Оглавление

Список принятых сокращений **9**
От автора **11**
Предисловие **14**
Введение **18**

Глава 1. Технология программирования программных систем **21**
1.1. Технология программирования сложных систем 21
1.1.1. Создание систем программирования 22
1.1.2. Системы синтеза, композиции и сборки программ 26
1.1.3. Технология программирования систем для ЭВМ 28
1.1.4. Развитие технологии сборочного программирования 30
1.1.5. Интерфейс и метод сборки в технологии программирования 32
1.2. Технология модульного проектирования систем 34
1.2.1. Интерфейсы модулей и их функции 36
1.2.2. Определение модульной структуры ПС 38
1.2.3. Матричное представление графов из модулей 41
1.2.4. Отношение достижимости модулей графов 42
1.2.5. Операции построения модульных структур 44
1.2.6. Процесс построения модульных структур 48
1.2.7. Отладка и тестирование модулей 50
1.3. Сборочное программирование и перспективы развития 53
1.3.1. Отечественные линии продуктов в АИС 54
1.3.2. Перспективы развития ТП (по А. П. Ершову) 55
Контрольные вопросы и задания 56

Глава 2. Программная инженерия / Software Engineering (SE) программных продуктов **57**
2.1. Инженерия компьютерных систем 58
2.2. Области знаний ядра SWEBOK 60
2.2.1. Программные требования 61
2.2.2. Проектирование ПО 63
2.2.3. Конструирование ПО 65
2.2.4. Тестирование ПО 67
2.2.5. Сопровождение ПО 69
2.2.6. Управление конфигурацией ПО 71
2.2.7. Управление инженерией ПО 72
2.2.8. Процесс SE 75
2.2.9. Инструменты и методы ПО 78
2.2.10. Качество ПО 80

2.3. Модели ЖЦ ПС 83
2.3.1. Каскадные модели 84
2.3.2. Итерационные модели 85
2.4. Парадигмы программирования SE 89
2.4.1. Парадигма событийно-управляемого программирования 89
2.4.2. Согласованное программирование и параллельные вычисления 90
2.4.3. Парадигма ООП 92
2.4.4. Agile-технологии 93
Контрольные вопросы и задания 95

Глава 3. Новые подходы к разработке изменяемых ПП из КПИ 96
3.1. Инженерия изготовления ПП в Product Line 97
3.1.1. Вариабельность продуктов и семейств на линиях SPLE 97
3.1.2. Процессы разработки ПП в SPLE 99
3.1.3. Процессы инженерии доменов 100
3.2. Инженерия повторного использования КПИ/Reuse 101
3.2.1. Разновидности КПИ 102
3.2.2. Спецификация КПИ 103
3.3. Подходы к созданию вариантов ПП 104
3.4. Моделирование изменяемых систем по К. Чернецки 107
3.4.1. Основы метода генерации ПП 107
3.4.2. Метод генерации доменов и приложений 107
3.5. Применение инженерии SPLE в системе Grid 109
3.5.1. Средства разработки ПС в ETICS Grid 110
3.5.2. Сборка, тестирование и конфигурация систем в ETICS 111
3.6. Фабрики программ и AppFab 112
3.6.1. Системные AppFab 114
3.6.2. Фабрики интеграции разнородных компонентов и данных 115
Контрольные вопросы и задания 115

Глава 4. Модели и методы проектирования вариантов систем 117
4.1. Определение вариантов КПИ в ПС и СПС 117
4.1.1. Определение КПИ и модели вариантности ПС и СПС 118
4.1.2. Введение вариантных характеристик в архитектуру ПС 118
4.1.3. Связь варианта с точками вариации 119
4.1.4. Модель КПИ с вариантами и механизмами вариации 120
4.1.5. Описания внешних и внутренних связей вариантов 122
4.2. Базовые модели проектирования систем 130
4.2.1. Модели MDD, MDA 130
4.2.2. Модели систем для разных платформ PIM и PSM 132
4.3. Моделирование систем в языке UML 135
4.3.1. Стратегия использования UML 135
4.3.2. Описание в UML веб-сайта оплаты услуг 136
4.4. Аспекты управления вариабельностью ПС 144
4.4.1. Функции управления вариабельностью 146
4.4.2. Подход к разработке вариабельной СПС из КПИ 151
4.4.3. Процесс сборки КПИ в современных средах 154
Контрольные вопросы и задания 155

Глава 5. Определения характеристик систем методами анализа и извлечения знаний 156
5.1. Моделирование функциональных и нефункциональных характеристик ПП 157
5.1.1. Метод анализа иерархий 157
5.1.2. Моделирование характеристик качества байесовской сетью 160
5.1.3. Неиерархические модели качества 162
5.1.4. Подход к оценке качества процессов линий 165
5.2. Модели представления знаний 167
5.2.1. Продукционные системы 167
5.2.2. Логические модели 168
5.2.3. Фреймы 169
5.2.4. Семантические сети 170
5.2.5. Методы многомерного анализа данных 171
5.2.6. Инструменты извлечения (добывания) знаний 172
5.3. Методы поиска и извлечения знаний 175
5.3.1. Методы машинного обучения 176
5.3.2. Методы извлечения знаний 176
5.3.3. Применение технологии Mining к системам 180
Контрольные вопросы и задания 182

Глава 6. Формальные методы спецификации, верификации и доказательства правильности систем 183
6.1. Методы и языки формальной спецификации 183
6.1.1. Классификация методов и языков спецификации программных систем 184
6.1.2. Темпоральные и параллельные спецификации 186
6.2. Формальные методы спецификации моделей систем 186
6.2.1. Формальное описание моделей систем в языке Z 187
6.2.2. Описание базовых конструкций языка VDM 189
6.2.3. Описание динамических систем концепторным языком 192
6.3. Формальные методы доказательства программ 195
6.3.1. Подходы к доказательству правильности спецификаций 196
6.3.2. Языки формальной спецификации 197
6.3.3. Характеристика базовых методов доказательства 199
6.3.4. Модель доказательства корректности перестановки данных в векторе 202
6.3.5. Верификация и валидация программ 205
6.4. Подходы к верификации моделей систем и характеристик 208
6.4.1. Логическое описание свойств моделей и их анализа 208
6.4.2. Инструменты верификации модели FM в Product Family 212
6.4.3. Корректность диаграмм характеристик онтологии OWL 214
Контрольные вопросы и задания 217

Глава 7. Теория и методы проектирования моделей доменов и систем ... 218
7.1. Теория моделирования систем из объектов на уровнях 219
7.1.1. Обобщающий уровень проектирования 221
7.1.2. Структурный уровень моделирования 222

7.1.3. Характеристический уровень проектирования ... 224
7.1.4. Поведенческий уровень проектирования ... 226
7.2. Операции над классами объектов ... 227
7.3. Формальные основы объектного анализа ... 228
7.4. Определение моделей ОМ, ПС, МХ ... 230
7.5. Теория моделирования ПС из компонентов ... 233
7.5.1. Компонентная модель системы ... 235
7.5.2. Формальные модели компонентного проектирования ... 236
7.5.3. Операции над компонентами в компонентной среде ... 239
7.5.4. Компонентная алгебра ... 240
7.6. Связь компонентной и объектной моделей ... 242
7.7. Технология конфигурационной сборки ... 246
7.8. Реализация технологии сборки в ИТК ... 251
Контрольные вопросы и задания ... 252

Глава 8. Применение теоретического аппарата фундаментальных и общих типов к неструктурным данным BIG DATA ... 254
8.1. Фундаментальные ТД в ЯП ... 255
8.2. ТД стандарта GDT ... 264
8.2.1. Формальный синтаксис GDT ... 264
8.2.2. Генератор сложных ТД GDT ... 267
8.2.3. Преобразование ТД ISO/IEC 11404—96 ... 269
8.2.4. Преобразование данных для связи компонентов в ЯП ... 270
8.3. Неструктурированные ТД ... 271
8.4. Подход к решению проблемы обмена разными ТД ... 273
8.4.1. Операции преобразования ТД ... 274
8.4.2. Подход к генерации FDT ⇔ GDT ... 275
8.5. Практика реализации разнородных ТД ... 276
8.6. Средства поддержки приложений с Big Data ... 278
8.7. Характеристика стандартов ISO ... 278
Контрольные вопросы и задания ... 280

Глава 9. Онтологический метод концептуального моделирования доменов ... 282
9.1. Описание доменов средствами онтологии ... 284
9.2. Основные понятия онтологии представления ПрО ... 286
9.3. Средства проектирования доменов в DSL ... 287
9.3.1. Структура описания домена в DSL ... 287
9.3.2. Трансформация модели домена в DSL к ЯП ... 289
9.3.3. Онтология домена «Вычислительная геометрия» ... 290
9.3.4. Процессы домена ЖЦ ISO/IEC 12207 ... 291
9.3.5. Формализация онтологической модели ЖЦ ... 295
9.3.6. Описание процессов ЖЦ средствами DSL и Protégé ... 296
9.4. Средства Protégé для представления онтологии ... 298
9.4.1. Разработка онтологии процесса тестирования ... 299
9.4.2. Онтология тестирования в Protégé ... 305
9.4.3. Плагины для графического представления онтологии ... 306

9.5. Онтология КПИ 307
Контрольные вопросы и задания 311

Глава 10. Методы экспертирования, тестирования и оценки качества ПС 312
10.1. Экспертирование компонентов и систем 312
10.1.1. Модель экспертизы процессов 312
10.1.2. Оценка процессов 314
10.1.3. Задачи экспертной оценки объектов 316
10.1.4. Технология экспертного оценивания объектов ПрО 317
10.2. Подходы к процессу тестирования ПС и СПС 318
10.2.1. Тестирование видов систем в SWEBOK 320
10.2.2. Задачи процесса управления тестированием 322
10.2.3. Инженерия процесса тестирования ПрО 323
10.2.4. Модели тестирования СПС 324
10.2.5. Задача определения оптимального времени тестирования ПС 327
10.3. Оценивание качества ПС и СПС 329
10.3.1. Моделирование характеристик качества ПС 331
10.3.2. Задачи управления качеством ПС 333
10.3.3. Модель требований для обеспечения качества ПС 333
10.3.4. Система прогнозирования безотказной работы ПС 335
10.3.5. Анализ достижения уровня качества 336
10.3.6. Задачи оценки показателей качества сложных систем 338
Контрольные вопросы и задания 339

Глава 11. Методы разработки веб-систем в Semantic Web 340
11.1. Структура Semantic Web 341
11.2. Задание информации в Semantic Web 343
11.2.1. Базовые элементы Semantic Web 344
11.2.2. Представление онтологии в Semantic Web 345
11.2.3. Стандарты консорциума W3C 347
11.3. Веб-технологии 347
11.3.1. Язык BPMN для спецификации бизнес-процессов 350
11.3.2. Процессы и операции бизнес-процессов 351
11.3.3. Типы объектов, событий и задач в BPMN 351
11.3.4. Моделирование семантики бизнес-процессов 352
11.3.5. ЖЦ бизнес-процессов 353
11.4. Веб-сервисы 355
11.4.1. Базовые модели сервисов SOA и SCA 358
11.4.2. Вычисление задач в Cloud computing 359
11.4.3. Язык WSDL 362
11.4.4. Языки RDF и OWL 363
11.5. Языки описания систем и протоколов Интернета 365
Контрольные вопросы и задания 366

Глава 12. Основы Software Engineering Method and Theory 368
12.1. Основные теоретические достижения SE 370
12.1.1. Определение новых дисциплин в SE 370

12.1.2. Теоретические аспекты онтологии ПП ... 376
12.1.3. Подход к обучению предмету SE ... 376
12.2. Основные сущности SEMAT. Практика, теория, образование ... 377
12.3. Задачи повышения компетенции инженеров ПС ... 379
12.3.1. Назначение модели компетенции специалистов ... 380
12.3.2. Понятие навыков в стандарте SWECOM ... 380
Контрольные вопросы и задания ... 381
Заключение ... 382
Словарь терминов ТП и SE ... 383
Список рекомендуемой литературы ... 391
Новые издания по дисциплине «Программирование» и смежным дисциплинам ... 396
Приложение 1. Моделирование систем диаграммами UML ... 397
Приложение 2. Сборка систем в .NET ... 406
Приложение 3. Взаимодействие систем в Интернете ... 416
Приложение 4. Первые всесоюзные конференции по технологии программирования (1968—1992) ... 425

Список принятых сокращений

АСУ — автоматизированная система управления
БД — база данных
ГоР — готовые ресурсы
ЖЦ — жизненный цикл
ИС — информационная система
КП — комплексы программ
КПИ — компоненты повторного использования
МХ — модель характеристик
ОКМ — объектно-компонентный метод
ООП — объектно ориентированные программы
ОС — операционная система
ПИ — программная инженерия
ПО — программное обеспечение
ПП — программный продукт
ППП — пакет прикладных программ
ППр — программирующая программа
ПрО — предметная область
ПС — программная система
СП — система программирования
СПС — семейство программных систем
ТД — тип данных
ТЛ — технологическая линия
ТП — технология программирования
ТПР — технология подготовки разработки
ЭВМ — электронно-вычислительная машина
ЯП — язык программирования
ACM — Association for Computing Machinery
CS — Computere Science
DE — Domain Engineering
DSL — Domain Specific Language
FM — Feature Model
GDM — Generative Domain Model
PIM — Platform Independed Model
PLE — Product Line Engineering
PSM — Platform System Model
SE — Software Engineering
SWEBOK — Software Engineering Body Knowledge

От автора

Теоретическими и прикладными аспектами автоматизации создания разных систем обработки информации на отечественных ЭВМ и на современных компьютерах автор занимается с самого начала своей научной деятельности. Исследования и разработки систем относятся к ТП и ПИ.

В период появления ЭВМ и разработки систем программирования для них в отделе Е. Л. Ющенко с 1960 г. проводились семинары по изучению синтаксического анализа ЯП в работах Н. Хомского, К. Самельсона, Ф. Бауэр, Р. Флойд и др. Стояла задача создать системы программирования для новых ЭВМ («Проминь», «М-20», «Днепр-1», «Днепр-2» и др.). В результате сформировался метод синтаксического контроля программ в разных ЯП. Метод реализован в технологических комплексах и системах программирования с языков Алгол, Кобол, Фортран, Автокод и др. Аналогичные разработки проводились в стране для других машин («МЭСМ», «БЭСМ», «Стрела», «Урал», «М-50» и др.). Созданы отечественные трансляторы ТА1—ТА4, которые вошли в состав системного математического обеспечения первых ЭВМ. Трансляторы с языков Алгамс и Кобол реализованы на машине «Днепр-2» и внедрены в ГДР в 1970-е гг. На их основе совместно со специалистами из Киева реализована АСУ ТП металлургической промышленности ГДР, которая проработала до 1992 г.

В 1970-х гг. в стране развернулись работы по автоматизации разного рода систем из программ (АПРОП, ПРОЕКТ, АЛЬФА и др.). Автор реализовала метод сборки модулей, записанных в разных ЯП (Алгол-60, Кобол, Фортран, PL/1, Assebler и др.), на единой системе ЭВМ. Подобные работы проводились специалистами в Москве, Ленинграде, Таллине и др. и в США. В АПРОП впервые реализована связь модулей в разных ЯП с помощью интерфейса (межмодульного и межъязыкового), осуществляющего обмен данными и их преобразование посредством интерфейсных функций (64) библиотеки в ОС ЕС. Система вошла в состав комплекса ПРОТВА и передана 52 организациям ВПК, а также в организации Прибалтики и Средней Азии. В результате сформировался новый вид программирования — сборочный. В составе специалистов комплекса ПРОТВА по технологии создания бортовых систем автор получила звание лауреата премии Кабинета министров СССР (1985).

Дальнейшим развитием сборочного программирования является метатехнология (ТПР) создания ТЛ производства программ в системе АИС «Юпитер» для Военно-морского флота СССР (1982—1991) в АН УССР. По пяти созданным линиям было изготовлено в АИС более 500 научных программ и программ обработки данных. Среди программ были такие, которые обеспечивали надежность технических и программных средств

АИС. Система передана заказчику. Идея линий получила развитие в проекте института SE USA (2001, http://www.sei.com/productline) и положила начало созданию вариабельных (изменяемых) продуктов семейств СПП (2004). Эти работы развиваются и в нашей стране рамках проекта Российского фонда фундаментальних исследований (РФФИ).

А. П. Ершов предвосхитил развитие сборочного программирования как доказательного (1986). В рамках фундаментальных проектов в отделе «Программная инженерия» Национальной академии наук Украины проведено изучение и развитие теории генерирующего (К. Чернецки), объектного (Г. Буч) и компонентного программирования (1992—2013). Создана теория объектно-компонентного моделирования объектных систем, которая апробирована в проекте информатизации НАН Украины (В. Н. Грищенко). Разработаны новые идеи и концепции верификации, тестирования и оценивания качества объектов и ПС (Г. И. Коваль, Т. М. Коротун). По ним защищено пять диссертаций сотрудников отдела и шесть магистерских работ Московского физико-технического института (МФТИ) и Киевского национального университета имени Тараса Шевченко (КНУ). Дальнейшее развитие этого метода состоит в создании разных вариантов систем и продуктов, в проведении стандартизации ГоР и reuses — КПИ в языке WSDL, в исследовании основных положений веб-семантики, веб-сервисов и онтологического представления ряда предметных областей (ЖЦ ПС, КПИ и вычислительная геометрия) с участием студентов КНУ, где автор преподавала ТП и ПИ с 1967 г., а также в МФТИ (2001—2016).

Современные ИТ-технологии Всемирной паутины и системы Skype ставят задачу по обеспечению взаимодействия программ, расположенных в разных точках планеты. В перспективе в информационном сообществе разрабатываются научные сервисы для поддержки *e*-коммерции, *e*-науки, *e*-обучения и т.п.

Развитие технических ресурсов ЭВМ идет быстрыми темпами по пути совершенствования и стандартизации микроэлементов (диоды, триоды, транзисторы и др.). Методом сборки из них изготавливают большие и малые компьютеры, очень малые настольные и карманные, а также мобильные телефоны и приборы для применения в медицине, космосе, авиации и т.п.

Такого прогресса еще не достигнуто в области изготовления сложных программных и информационных систем. ТП и ПИ находятся на стадии формирования готовых многоразовых ресурсов типа модуль, объект, компонент для конфигурационной сборки сложных систем. Эти объекты будут минимизироваться подобно техническим наноэлементам, и из них будут собираться большие системы в миниатюрном виде.

В данном учебнике нашли отражение основные положения ТП, сформировавшейся в СССР, и ПИ изготовления ПО и продуктов в разных вариантах. В нем более углубленно и с теоретической точки зрения рассматриваются методы проектирования отдельных компонентов систем, их верификация и тестирование, а также сборки готовых КПИ, ГоР в сложные структуры с обеспечением их правильности и качества ПП в конфигурационные варианты систем семейства. Проблема изменяемости/вариабельно-

сти нашла отражение в отечественной технологии объектно-компонентного моделирования вариантов сложных систем из артефактов, reuses и КПИ.

Отдельные разделы учебника апробированы автором при чтении лекций в МФТИ (2001—2016) и Н. В. Лаврищевой в КНУ. Новые теоретические результаты в SE отображены в ряде дипломных работ студентов на сайте (http://www.sestudy.ua-univ.net, 2011), студенческой фабрики программ КНУ (http://www.programsfactory.univ.kiev.ua, 2012) и в статьях и докладах учеников автора на международных конференциях ICTERY (2011—2013), УкрПрог (1998—2014), TAAPSD'2008—2015, Science and Information — 2015, АПСПИ — 2015 и др.

На практических занятиях студенты изучают линии Product Line/ Product Family, Streaming assembly and conveyor К. Чернецки, Дж. Гринфилд, К. Ленц и др. С участием студентов и сотрудников отдела «Программная инженерия» реализована веб-онтология таких доменов, как вычислительная геометрия, КПИ и ЖЦ стандарта ISO/IEC 12207. Разработаны методы экспертирования, верификации и тестирования систем, предложен математический аппарат оценки характеристик качества продуктов. Ряд результатов исследований и разработок отражен на модифицированном сайте http://www.ispras.ru/lavrischeva/SE.

Автор глубоко благодарна сотрудникам отдела (В. Н. Грищенко, Н. В. Лаврищева, Л. Г. Усенко, Е. И. Гришкевич, Т. В. Соколова, М. В. Семенюк, А. Т. Вишня, Д. С. Хоролец, Г. И. Коваль, Т. М. Коротун, Л. П. Бабенко, О. А. Слабоспитская, Н. Т. Задорожная, В. М. Зинькович, Е. И. Моренцов, Л. И. Куцаченко и др.), которые долгие годы развивали и реализовывали отдельные новые идеи в области ТП и SE. Семь сотрудников защитили кандидатские диссертации.

Выражаю признательность многим студентам (А. Островский, А. Аронов, А. Диденко, И. Радецкий, В. Бураков, В. Черный, Е. Гриценко, Т. Гончаренко, А. Поляченко, А. Колесник, А. Мороз, А. Бондаренко, М. Тарановский, Е. Баборико и др.), которые принимали активное участие в реализации поставленнях мною задач в области ТП и SE, писали по ним дипломы и магистерские работы. Особенно хочу поблагодарить студентов А. Колесник, А. Стеняшина, А. Шевченко, которые защитили кандидатские диссертации по современным вопросам создания изменяемых систем.

Хочу выразить благодарность В. П. Иванникову, А. К. Петренко, А. Н. Томилину, Л. Е. Карпову за признание моих работ в области ТП и ПИ, поддержавших проведение исследований и разработок вариабельных прикладных и операционных систем в рамках фундаментального проекта РФФИ.

Предисловие

С учетом направлений ТП, SE и SEMAT (SE Methods and Theory) в учебнике излагаются результаты теоретических исследований и разработок отечественных и зарубежных специалистов в области ТП и инженерии ПП и семейств СПС.

В области ТП сформировались сборочное программирование, метод моделирования объектных систем из КПИ, а также экспертирование, тестирование и оценивание качества ПС. Определен подход к обеспечению взаимодействия ГоР между собой в среде Интернет. Представлена теория типизации данных (фундаментальных, общих GDT — стандарт ISO/IEC 11404–2007) и предложены новые механизмы обработки неструктурированных данных (больших данных) типа Big Data. Описаны методы онтологии ЖЦ, КПИ и вычислительной геометрии.

При освоении данного учебника обучающийся должен:

знать

- теорию модульного программирования систем из разнородных языковых модулей, интегрируемых вместе с помощью интерфейса;
- теоретические основы и методы проектирования вариабельных систем на основе модели характеристик, моделей систем и модели конфигурации;
- зарубежные методы формальной спецификации объектов предметной области, верификации и доказательства с использованием методов Флойда, Дейкстры, Турского и др.;
- теорию объектно-компонентного моделирования предметных областей с применением математических и логических операций;
- теорию формальных и общих ТД (FDT, GDT) и некоторые новые механизмы генерации неструктурированных больших данных типа Big Data структур FDT, GDT;
- онтологические основы представления научных и инженерных доменов (ЖЦ стандарта ISO/IEC 12207, домены КПИ и «Вычислительная геометрия»);
- парадигмы программирования (модулей, компонентов, сервисов, объектов, аспектов и др.) к виду отдельных самостоятельных элементов повторного использования КПИ и взаимодействующих между собой через интерфейс, определенный автором в 1970-х гг. и широко используемый во всех общесистемных инструментах (IBM, Microsoft, Intel, Apple и др.) и сегодня в фабриках программ;
- классификацию дисциплин ПИ и характеристику новых дисциплин в Semantic Web;
- новые теории и методы SEMAT;

уметь

- создавать операционные, прикладные бизнес-системы;
- анализировать новые подходы и методы, которые появляются в SEMAT;
- представлять итоги проделанной работы в виде отчетов, рефератов, статей с учетом стандартных требований с привлечением современных средств редактирования и печати;

владеть

- общенаучными методами исследований и творчески применять их при проведении разработки крупных программных проектов;
- методами проектирования современных вариабельных систем, широко используемых за рубежом;
- владеть методами представления данных, анализа и синтеза программной и системной информации, извлекать знания из готовых действующих систем методами Mining для последующего применения в других системах, а также формировать выводы из полученных знаний.

Материал данного учебника состоит из 12 глав. В конце каждой главы для студентов приводятся контрольные вопросы и задания, а также список используемой литературы.

Глава 1. Технология программирования программных систем. Изложены основы технологии программирования в СССР. Рассмотрены системы программирования и трансляторы с первых ЯП (Алгол, Кобол, PL/1, Алгол-68 и др.). Определены сформировавшиеся методы синтеза, сборки и тестирования программ. Представлена модульная ТП больших систем. Дано определение сборочной технологии, обеспечивающей интеграцию и объединение разнородных КПИ и GoP.

Глава 2. Программная инженерия / Software Engineering программных продуктов. Представлена ПИ изготовления ПО, ПС и ПП средствами стандарта SWEBOK, включающего 10 областей знаний. Дано описание этих областей знаний. Изложены парадигмы программирования, сложившиеся в рамках SE. Рассмотрены модели ЖЦ, оформившиеся в процессе разработки ПО, ПС и ПП. Рассмотрены парадигмы программирования типа Agile.

Глава 3. Новые подходы к разработке изменяемых ПП из КПИ. Изложены методы производства ПП на Product Lines США с использованием КПИ и GoP. Определена модель изменяемости (вариабельности) FM, предложенная К. Похлом. Рассмотрены другие подходы к конфигурационной сборке (К. Чернецки, К. Ленц и др.), а также Grid-системы и фабрики программ — AppFab. Представлена технология разработки вариантов систем методом конфигурационной сборки.

Глава 4. Модели и методы проектирования вариантов систем. Рассмотрена концепция формирования вариантов систем из КПИ, артефактов и продуктов. Определены внутренние и внешние характеристики КПИ и вариантные точки для внесения изменений. Задана технология управления вариантами в языке C#. Рассмотрены базовые модели (MDA, MDD, PIM, PSM, UML) моделирования систем. Представлены методы генерации, сборки и управления вариабельностью.

Глава 5. Определения характеристик систем методами анализа и извлечения знаний. Рассматривается моделирование функциональных характеристик с помощью методов анализа иерархий, байесовских сетей, методов Minning и извлечения знаний. Описаны инструменты анализа и извлечения знаний из готовых систем. Приведены предметные области, в которых используются методы извлечения знаний и реинженерии систем.

Глава 6. Формальные методы спецификации, верификации и доказательства правильности систем. Дана классификация методов спецификации моделей систем (Z, VDM, B, UML и др.) и теория доказательства программ с помощью утверждений, пред- и постусловий. Описаны методы статической и динамической верификации на основе онтологии OWL, а также представлена логика спецификации модели FM и средств анализа таких моделей.

Глава 7. Методы проектирования моделей доменов и систем. Рассмотрен метод ОКМ моделирования доменов из объектов с использованием логико-математического аппарата и построения графовой объектной модели из объектов и их интерфейсов. Определена компонентная теория разработки и изменения систем из объектов и компонентов. Изложен метод управления конфигурационной сборкой ПС из КПИ, хранящихся в репозиториях или библиотеках.

Глава 8. Применение теоретического аппарата фундаментальных и общих типов к неструктурным данным Big Data. Приведена аксиоматика ТД: фундаментальные, общие GDT (ISO/IEC 11404–2007) и неструктурированные данные в Big Data. Рассмотрены механизмы формального описания разных фундаментальных, общих ТД и данных, генерируемых с разных устройств. Определены функции преобразования разнотипных данных, передаваемых между разнородными системами и средами.

Глава 9. Онтологический метод концептуального моделирования доменов. Описан базис онтологии в языке OWL и подход к описанию систем средствами языка DSL. Показаны онтологии доменов — «Вычислительная геометрия», ЖЦ ПС и КПИ, сформированные в процессе обучения студентов МФТИ. Описана оригинальная концептуальная модель ЖЦ в DSLToolMS.Net и в языке Protégé, а также модель тестирования ПС, реализованная в среде Eclipse.

Глава 10. Методы экспертирования, тестирования и оценки качества ПС. Изложены подход к экспертизе процессов и программ, тестирование ПС и сбор данных об ошибках. Тестирование проводится с использованием модели вариантов и нового метода расчета оптимального времени тестирования ПС. Рассмотрена модель требований, прогнозирования безотказной работы ПС и оценки качества ПС и ПП.

Глава 11. Метод разработки веб-систем в Semantic Web. Дано определение понятий Semantic Web. Приведены основные компоненты и стандарты комитета W3C (BPMN, BPML, SOAP, OWL и др.) для описания веб-систем и онтологий. Рассмотрены веб-сервисы, модели SOA, SCA и набор композитных сервисов, используемых при решении бизнес-задач. Изложены методы организации вычислений ПС в Cloud Computing и с Big Data.

Глава 12. Основы Software Engineering Method and Theory (SEMAT). Изложены теоретические аспекты дисциплин SE и ЖЦ. Рассмотрены подходы к разработке новых методов и теории SEMAT, ориентированных на представление и получение теоретических знаний и навыков по специальности SE у студентов и разработчиков ПП.

Приложение 1. Моделирование систем диаграммами UML.

Приложение 2. Сборка систем в .NET

Приложение 3. Взаимодействие систем в Интернете.

Приложение 4. Первые всесоюзные конференции по технологии программирования (1968—1992).

Введение

Программирование возникло в связи с появлением первых ЭВМ как способ задания последовательности операций для решения математических и физических задач на ЭВМ. Первоначально программы описывались в языках, близким к машинному языку. Потом появились ЯП (Адресный, Автокод, Алгол, Кобол, Фортран, ПЛ/1 и др.), с помощью которых описывались разного рода программы. Для того чтобы программу понимала машина, необходимо было ее переводить на язык машины. Их перевод осуществлялся с помощью появившихся средств автоматизации — программирующих систем (трансляторов и интерпретаторов).

Отрабатывалась новая техника описания алгоритмов математических, физических и других задач в виде некоторой формальной схемы программы. Сформировались теории А. А. Ляпунова, В. М. Глушкова, А. П. Ершова, Е. Л. Ющенко и др. Эти теории были ориентированы на формализацию логических схем программ и машин и их переписывание в ЯП, для которых были разработаны программирующие программы (трансляторы, интерпретаторы, генераторы) на ЭВМ.

Разработкой трансляторов с разных ЯП для отечественных ЭВМ занимались научные школы программирования (ИПМ, М. Р. Шура-Бура, Москва; ЛГУ, С. С. Лавров, Ленинград; СОАН, А. П. Ершов, Новосибирск; ИК, В. М. Глушков, Киев и др.). Позже такие школы были созданы в Риге, Таллине, Минске, Кишиневе, Ереване и др. В этих школах сформированы направления отечественной ТП, основу которых составляли ЯП, методы и средства автоматизации программ на разных ЭВМ. ТП охватывала не только трансляцию программ, но и их отладку и организацию вычислений под управлением специально созданных программ операционных систем для каждой ЭВМ.

Для координации и финансирования работ по ТП в СССР был создан Госкомитет по науке и технике (ГКНТ). Он поддерживал разработки и развитие научных основ вычислительной техники и ТП. В рамках ГКНТ создавались государственные комиссии с привлечением известных специалистов страны для приемки новых средств вычислительной техники и технологий программирования, изготовленных в разных организациях страны.

В 1970-х гг. проводилась массовая разработка программных модулей для решения разных задач на ЭВМ. ГКНТ ставит задачи автоматизации разных отраслей промышленности, крупных военно-промышленных комплексов (ВПК), АСУ, АСУ ТП, САПР и др. ГКНТ финансировал системы автоматизации модульного программирования («АПРОП», «ПРОЕКТ», «АЛЬФА», «СИРИУС» и др.), технологический комплекс «РТК»

с *Р*-графическим языком задания программ и документов АСУ, систем «ПРОТВА», «РУЗА» (В. В. Липаев) по изготовлению больших программ для ВПК и др. Главным элементом разработки систем стал модуль, выполняющий некоторую функцию и описанный в стандартном виде. По решению ГКНТ для хранения готовых отдельных модулей и программ общего назначения созданы государственные и республиканские Фонды алгоритмов и программ (1970—1992). В стране был создан в Калинине (ныне Тверь, 1980) первый завод для сборки ПО АСУ из готовых программ разных фондов. После двух лет работы этот завод перестал существовать из-за отсутствия необходимого набора готовых программ и КПИ.

Созданные системы автоматизации «АПРОП», «ПРОТВА», «РТК» и др. внедрялись в разные организации страны и служили способом автоматизации разных задач. Кроме того, были созданы системы АСУ в нашей стране, а также в Болгарии, ГДР и др.

Для обсуждения проблем автоматизации разных систем в стране ГКНТ проводил всесоюзные конференции по ТП. Первая Всесоюзная конференция по программированию прошла в Киеве (1965), вторая — в Новосибирске (1978) и последующие конференции по ТП проводились в 1979, 1982—1992 гг. В работе каждой конференции принимали участие тысячи специалистов, вовлеченных в автоматизацию разных задач. В организации этих конференций принимали участие видные специалисты страны в ТП: В. М. Глушков, А. П. Ершов, Е. А. Жоголев, С. С. Лавров, Л. Н. Королев, Э. Х. Тыугу, М. Р. Шура-Бура, Е. Л. Ющенко, и др.

Инженерия ПО (Software Engineering; SE) сформировалась для первых зарубежных ЭВМ, аналогично ТП, и была представлена П. Науром на конференции НАТО (1968). В США проводились разработки ПО систем общего назначения для разных ЭВМ (IBM, Intel, Apple, Sun и Microsoft), а также прикладных систем в ЯП для этих ЭВМ.

Были созданы международные комитеты ISO/IEC, ACM и IEEE для проведения работ по созданию средств автоматизации ПО ЭВМ и систем. Комитет выпустил стандарт разработки систем: SWEBOK (2001, 2004, 2007, 2014); ISO/IEC 12207 — ЖЦ ПО, оценки качества ПО, стандарт описания многоразовых КПИ (Reusebility, 1987—2012) и др.

SWEBOK (Software Engineering Body Knowledge) сформировался в США (www.swebok.com). Это процесс разработки ПО в виде областей знаний, отражающих последовательность процессов проектирования ПО систем. Для поддержки областей знаний разработаны специальные CASE-инструменты (Rational Rose, COM, CORBA, UML и др.), а также стандарт ISO/IEC ЖЦ 12207, регламентирующий процессы разработки систем согласно областям знаний SWEBOK. В его последних версиях определены новые дисциплины SE (теория, экономика, менежмент и др.) и новые парадигмы программирования: объектно ориентированные; клиент-серверные, компонентные, сервисные, системы реального времени, бизнес-системы и др.

Начиная с 2000 г. многие крупные фирмы и отдельные специалисты за рубежом проводят работы по обеспечению изменяемости (вариабельности) систем и ПП. Так, в SEI USA разработан метод производства ПП —

Product Line/ Product Family со сборкой КПИ (компонентов, reuses, assets и др.) по заказу потребителей. Теоретическую основу концепции вариабельности сформулировали зарубежные специалисты (К. Похл, К. Чернецки, Т. Бёгнер и др.). Определена модель характеристик FM (Feature Model), включающая внешние характеристики (свойства) создаваемого ПП. Разработаны новые методы верификации FM и конфигурационной сборки ПП из КПИ и других ГоР. Представлены отдельные аспекты Semantic Web и создания ПП с помощью сервисов Интернет.

Основные разработчики SE постоянно развивали новые теоретические методы изготовления ПО систем. В 2009 г. вышел новый стандарт SEMAT, ориентированный на дальнейшее развитие теории и методов изготовления ПП и систем, в том числе логического и математического моделирования систем и моделей (http://www.semat.org). Для формального доказательства этих моделей необходимы математический аппарат и формальные языки спецификации типа VDM, RSL, Z, UML и доказательства программ методами верификации моделей и систем. Появились новые методы анализа характеристик систем средствами иерархических и байесовских сетей, а также методы анализа и извлечения знаний Minning.

Развитию теории проектирования ПС из КПИ и ГоР способствовали сформулированные автором формальные дисциплины SE (теория, инженерия, экономика, менеджмент и производство), теория ОКМ, а также стандарт ICO/IEC PMBOK, нацеленный на управление программными проектами и обнаружение рисков в разработке качественных систем.

Глава 1
ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

Под **технологией программирования** понимается совокупность методов, средств и инструментов для проведения процесса программирования задач предметной области с целью получения ПП с заданными свойствами и функциями. Методами могут быть: модульный, объектный, компонентный и др. К средствам относятся ЯП, а также Analytic, Domain Specific Language, VDM, SADT, SSADM, OMT и др. Инструменты — это системы программирования, отладчики, редакторы и CASE-системы (UML1, UML2, Rational Rose и др.) и ТЛ разработки вариантов некоторого продукта, в которых отображается регламентированная последовательность действий над объектом разработки, начиная от требований, задания схемы (графа) системы из ее элементов и действий по их реализации в конкретной технологической среде. Каждое действие поддерживается системным или специальным инструментом, на вход которого подается промежуточный результат и на выходе — готовый выходной код.

В ТЛ указываются оборудование, на котором надо работать, и технология его использования с заданным порядком выполнения операций и действий на линии. Выходной результат линии — продукт и документация, а также инструкции по ее применению, получению вариантов продуктов и проведению модификаций ПП в процессе эксплуатации и замены непригодных инструментов или функций новыми, более совершенными. Технологический процесс — это общее базовое понятие в индустрии ПП, компоненты которого могут применяться по-разному, в разных вариантах, в зависимости от требований и условий использования. Таким образом, на практике долгие годы в СССР использовалась технология программирования, сформировавшаяся для первых ЭВМ.

В данной главе рассматриваются фундаментальные основы ТП, система программирования или программирующая программа (ППр), технология модульного программирования, широко используемая в СССР для производства программных продуктов и систем различного назначения.

1.1. Технология программирования сложных систем

Программирование вычислительных задач начало развиваться на цифровых ЭВМ с помощью языков представления алгоритмов и программ, их преобразования в коды машин средствами ППр или систем программирования (трансляторов, интерпретаторов) с языков описания программ (Адрес-

ный, Алгол-60, Фортран, Кобол, Алгол-68 и др.). Трансляторы разрабатывались для отечественных ЭВМ («БЭСМ», «М-20», «М-50», «Минск-32», «Стрела», «СМ1-3», «Днепр-2» и др.) в рамках ТП. Создавались системы автоматизации методами синтезирующего, сборочного программирования, технологического комплекса «РТК» с Р-графическим языком задания программ и документов и технологии В. В. Липаева[1] по изготовлению больших программ для ВПК и др.

1.1.1. Создание систем программирования

Первые советские монографии по программированию — «Элементы программирования» А. И. Китова, И. А. Криницкого (1956), «Быстродействующие электронные машины АН СССР» С. А. Лебедева (1952), «Общее описание БЭСМ и методика выполнения операций» С. А. Лебедева и В. А. Мельникова (1958) и др. — освещали аспекты создания программ решения разных математических задач на ЭВМ. Они открыли эру алгоритмизации вычислительных задач с помощью языков, близких ЭВМ. Это также работы «Алгоритмы и машинное решение задач» (В. А. Трахтенгерц, 1957), «Начальные сведения о решении задач для ЭВМ» (А. А. Ляпунов и Г. А. Шестопалов, 1957), «Граф-схемное программирование» (А. А. Ляпунов, А. П. Ершов), «Адресное программирование» (Е. Л. Ющенко, 1962) и др.

Основные элементы отечественной ТП

Программирующая программа. Первая идея ППр возникла в 1953 г. у А. А. Ляпунова, который читал курс по теории представления алгоритмов с помощью операторных граф-схем в МГУ. На языке операторных схем программа представлялась схемой, в виде управляющего графа и совокупности спецификаций его операторов. В основе входного языка каждой из ППр лежал общий концептуальный базис, фиксирующий типы операторов и их спецификации. Программа ППр-1 сделана в 1954 г. с участием Э. З. Любимского, А. П. Ершова в МГУ под руководством М. Р. Шура-Бура. Это первая ППр в мировой практике. ППр-1 стала прототипом для проектирования других ППр на машинах «БЭСМ», «Стрела», «М-20» и др. В результате были созданы фундаментальные алгоритмы трансляции и положено начало системному и теоретическому программированию. Важную роль сыграли теоретические работы: А. П. Ершова «Введение в теорию программирования» (1974), В. М. Глушкова, Г. Е. Цейтлина и Е. Л. Ющенко «Алгебра. Языки. Программирование» (1974), М. Р. Шура-Бура, Э. З. Любимского «Транслятор ТА2 на языке АлМО» (1970) и др.

Идея ППр развивалась и в ВЦ АН УССР в период создания машины МЭСМ А. А. Лебедевым. В. М. Глушков в статье «Об одном методе автоматизации программирования» (журнал «Проблемы кибернетики». 1957. № 2. С. 181—189) и А. А. Стогний (там же, с. 190—199) в статье «О прин-

---

[1] См.: *Липаев В. В.* Программная инженерия. Методические основы. М. : СИНТЕГ, 2006; *Его же*. Программная инженерия сложных заказных программных продуктов. М. : Изд-во института системного программирования, 2014.

ципах построения специализированной ППр» определили библиотечный метод алгоритма решения систем дифференциальных уравнений. ПП использовались на разных машинах: «УМШН», «Проминь», «Днепр», «Днепр-2» и др. Трансляторы для этих машин были реализованы в адресном языке Е. Л. Ющенко и в специализированных языках типа Автокод.

Библиотеки стандартных программ. Е. А. Жоголев (МГУ) разработал стандартную программу, которая осуществляла статическую загрузку и связывание программ из библиотеки численных методов. Интерпретирующая система ИС-2 (М. Р. Шура-Бура) использовалась на машинах «М-20», «БЭСМ» и др. В ней реализованы метод динамического вызова библиотечных подпрограмм, способы связывания, подкачки и замены. ИС-2 стала неотъемлемой частью разных ОС ЭВМ, поставляемых пользователям[1].

Операционные системы. Для «БЭСМ-6» была сделана операционная система (ОС) в Институте точной механики и вычислительной техники под руководством В. П. Иванникова, которая долгое время выполняла общие функции управления памятью, задачами и данными. Дальнейшее развитие ОС получила в вычислительной системе «Электроника ССБИС» (1985—1991). Ее цели и задачи опубликованы в ряде статей, в том числе и в итоговой статье[2]. В ней представлены ОС и ее ядро. В это ядро входят стандартные средства ОС, средства взаимодействия с IBM и Эльбрус, набор протоколов обслуживания программ и система программирования с ЯП (Фортран, С, Паскаль и др.). Базовая система программирования включает новые средства поддержки абстрактных ТД, кластерного сборочного конвейера для загрузки модулей в ОС и в библиотеку программ трансляторов, а также отладчик программ на ЯП и сервисные средства обслуживания пользователя.

Системы программирования

С 1960 г. в стране появилось много ЯП: Адресный, Алгол-60, Фортран, Кобол, Пролог, Ада и др.

Одним из первых языков, используемых для программирования математических задач и ПП, был Адресный язык.

Адресный язык (Е. Л. Ющенко, Б. В. Гнеденко, В. С. Королюк) был близок к языку математики. Алгоритм в этом языке записывался набором строк операторов, в каждой из которых — одно или несколько действий — формул. Для задания порядка операторов используются метка и операторы безусловного перехода. Переменные обозначались буквами и соответствовали ячейкам машины. Содержимое некоторого адреса отмечалось указателем, адресом второго ранга и использовалось для многократного перехода. Этот язык также применялся для описания программ вычислительного характера и реализации ПП для машин «Киев», «Урал» и «Днепр».

---

[1] См.: *Жоголев Е. А.* Система стандартных подпрограмм / Е. А. Жоголев [и др.]. М. : ГИФМЛ, 1958.

[2] *Иванников В. П., Гайсорян С. С., Томилин А. Н.* Системное программное обеспечение вычислительной системы «Электронника ССБИС. SORUCOM.2014» // Труды Межд. конференции «Развитие ВТ и ее ПО в России и в странах бывшего СССР. История и перспективы», 15—17 октября 2014 г., Казань. С. 117—125.

Язык Алгол-60. Он возник в США как универсальный ЯП научных и инженерных задач[1]. Этот язык в ВЦ АН СССР был принят в качестве языка описания научных задач. Сформировалось следующие направления разработки трансляторов с языка Алгол (ТА): ТА1 — С. С. Лавров (ЛГУ, 1962), ТА2 — М. Р. Шура-Бура и Э. З. Любимский (ИПМ, 1965), ТА-3 (Альфа-система) в русской версии языка Алгол-60 (СО АН СССР, 1966), ТА-4 — Е. Л. Ющенко, Е. М. Лаврищева — для УВК «Днепр-2» (ИК АН СССР, 1967). Позднее, в 1970-х гг. делались еще трансляторы: с языка Алгол-68 (А. Н. Терехов, ЛГУ), с языка Кобол (Л. П. Бабенко для «Днепр-2» и О. Н. Романовская для «Минск-32») и др.[2]

На основании этого языка в трансляторе ТА-1 реализованы простая схема трансляции, стековый подход к трансляции выражений, процедур без рекурсий. В трансляторе ТА-2 разработан оригинальный алгоритм реализации процедур, механизм управления памятью (оперативной и внешней) и метод таблично-управляемой генерации кода. В ТА-3, АЛЬФА-трансляторе[3] осуществлена оптимизация (выражений, циклов, процедур, памяти и др.) для повышения эффективности выходного кода, который был близок коду, созданному вручную. В ТА-3 реализованы также операции над многомерными значениями и комплексными ТД. В трансляторе Д-АЛГАМС (ТА-4) — СМ-метод табличного представления БНФ и набора семантических программ их трансляции[4]. Новым подходом к трансляции системы программирования «Днепр-2» было создание общего арифметического блока для двух языков Алгол и Кобол.

Трансляторы ТА1—ТА3 и ОС для новых ЭВМ («М-20», «СТРЕЛА», «БЭСМ») были реализованы в машинном коде этих машин, а транслятор ТА4 — на языке, близком Адресному языку, — Автокод «Днепр-2». Трансляторы АКД, Д-АЛГАМС и Кобол входили в состав общесистемного ПО машины «Днепр-2». Трансляторы с этих языков были сданы в составе ОМО (ОС и систем программирования АКД, Д-АЛГАМС, Кобол) УВК «Днепр-2» Госкомиссии СССР в 1967 г. Комиссию возглавлял А. В. Дородницын. В ее состав входили Л. Н. Королев, В. М. Глушков, А. Н. Кухарчук и др. Эта машина была приобретена ГДР в 1971 г., ПО машины было внедрено в вычислительном центре немецкой металлургической фирмы BMHW и работало до 1991 г. С помощью этих систем совместно специалистами ГДР и Украины разрабатывалась АСУ ТП металлургическими процессами. Работала она там до 1992 г. Следует отметить, что одновременно для «Минск-32» делался транслятор с языка Кобол сотрудниками ИНЭУМ Белоруссии.

---

[1] См.: Сообщение об алгоритмическом языке Алгол-60 / под ред. А. П. Ершова. М., 1960.

[2] См.: *Лаврищева Е. М., Грищенко В. Н.* Связь разноязыковых модулей в ОС ЕС. М. : Финансы и статистика, 1982.

[3] См.: *Бабецкий Г. И.* Система автоматизации программирования АЛЬФА // Журнал вычислительной математики и математической физики. 1965. Т. 5. № 2.

[4] См.: *Yushchenko E. L., Lavrishcheva E. M.* A method of analyzing programs based on a machine language // Springer. 1972. Vol. 8. №. 2. P. 219—223; и др.

Системы программирования включали трансляторы, отладчики и редакторы программ. Э. З. Любимский (1963) предложил промежуточный язык — АлМО — для перевода теста в любом ЯП в этот язык, а затем текста АлМО в код ЭВМ[1]. АлМО — это некоторая абстрактная машина, отражающая особенности класса ЭВМ в СССР. Этот и другие языки (Эпсилон, Сигма) стали языками-посредниками при трансляции программ с различных ЯП. Их смысл состоял в замене трансляции с m входных языков в n машинных языков, т.е. «из m в один» и «из одного в n». АлМО реализован для основных машин того времени («М-20», «БЭСМ-6», «Минск-2», «Урал-1») и проверен для трансляторов с Алгол-60 и Фортран[2].

Одной из систем программирования с ЯП была система с языка Алгол-68 (Г. С. Цейтин, А. Н. Терехов, ЛГУ, 1968—1991), переведенного на русский язык в 1979. Многие годы разрабатывались варианты трансляторов в ЛГУ, Минске, Новосибирске (Бета). Однако наиболее реальный транслятор был сделан под руководством А. Н. Терехова для первых ЭВМ — СМ1, СМ2, а потом для ОС единой серии ЭВМ. НИЦЕВТ финансировал эту работу. Это была одна из самых сложных и трудных разработок в стране. В работе «Процессы идентификации и структура транслятора с языка Алгол-68» А. Н. Терехов описал наработанные многими членами рабочей группы по Алгол-68 вопросы реализации транслятора и программ перевода языковых конструкций сначала в промежуточный язык, а потом в код машины. История развития и создания системы с языка Алгол-68 описана в статье «Алгол-68 и его влияние на развитие программирования в СССР в России».

Опыт разработки программ на Алгол-60 и первые трансляторы обсуждались на Первой Всесоюзной конференции по программированию в Киеве (1965), Второй конференции в Новосибирске (1978) и на последующих конференциях по ТП (1979, 1982—1992). В работе каждой конференции принимали участие более 1000 человек. Это объясняется большой популярностью данной проблематики в стране. По линии ГКНТ (Госкомитет по науке и технике), ГКНТИ (ГКНТ и информатике) СССР в Алуште под руководством проф. Е. Л. Ющенко проводились Всесоюзные конференции (1981—1990) по разным аспектам ТП. На них обсуждались новые теоретические и прикладные аспекты проектирования, разработки, тестирования различных видов систем для ЭВМ, а также задачи эксплуатации и сопровождения ПО. В развитие отечественной ТП внесли вклад ведущие специалисты академических институтов АН СССР: В. М. Глушков, И. В. Вельбицкий, А. П. Ершов, Э. З. Любимский, В. С. Лавров, М. Р. Шура-Бура, К. Л. Ющенко, Э. Х. Тыугу и др. ГКНТ СССР финансировал работы по разработке средств автоматизации ТП. Были специальные постановления Кабинета министров СССР, направленные на развитие средств вычис-

---

[1] См.: *Шура-Бура М. Р., Любимский Э. З.* Транслятор с языка Алгол-60 // Журнал вычислительной математики и математической физики. 1964. Т. 4. № 1.

[2] См.: *Жоголев Е. А.* Система стандартных подпрограмм; *Лавров С. С.* Основные понятия и конструкции ЯП. М. : Финансы и статистика, 1982; *Лаврищева Е. М.* Транслятор с языка Д-АЛГАМС для УВК «Днепр-2» / Е. М. Лаврищева [и др.]. Киев : Изд-во ИК АН УССР, 1970; *Лаврищева Е. М., Грищенко В. Н.* Связь разноязыковых модулей в ОС ЕС. М. : Финансы и статистика, 1982.

лительной техники и технологии изготовления ПП и сдачи их в фонды алгоритмов и программ для использования другими программистами.

Р-технология. Для конструирования структур произвольных программ с помощью визуальных *Р*-графов и схемной реализации программ разработано устройство синтаксического контроля программ за несколькими отечественными патентами[1]. Выполнена структурная интерпретация синтаксиса и семантики ЯП с использованием этого языка. Графический стиль программирования использовался во многих организациях военно-промышленного комплекса СССР. Затем был создан технологический комплекс программиста для проведения синтаксического анализа программ в *Р*-языке.

Ведение новых СП в эксплуатацию

Созданные в стране СП (трансляторы с входных языков типа Алгол (ТА1-ТА4), средства отладки программ в системе) передавались в организации, где находились новые ЭВМ, вместе с документацией, включающей:

- описание входного языка;
- руководство программиста;
- руководство системного программиста;
- инструкцию по запуску и сопровождению.

Эти системы проходили опытную эксплуатацию в течение нескольких лет на примерах задач (вычислительных, экономических, планирования, управления и др.). Находились не только собственные ошибки в программах, но и в СП и ОС. Системные ошибки устранялись разработчиками систем, которые потом передавали новую версию организациям пользователей. В качестве примера можно привести ОС 360 IBM, которая выдала организациям-пользователям более 200 версий на протяжении двух десятков лет. Потом был сделан стандарт ОС IBM 370. Аналогично было с отечественными СП. Появлялись другие языки ЯП (Ада, Фортран, Пролог и др.), СП которых проходили длительную отработку, пока они получали статус промышленного применения. Документация на системы размещалась в фондах алгоритмов и программ для приобретения любыми пользователями. Позднее Кабинет министров СССР принял решение о межведомственной сдачи ОС и СП. В состав комиссий по приемке систем входили видные специалисты страны в области ЭВМ. Однако стандартной версии ОС и СП почти никто из разработчиков систем не сделал. На некоторых системах в закрытых организациях страны до сих пор работают отдельные ОС и СП.

1.1.2. Системы синтеза, композиции и сборки программ

Система «ПРИЗ» разработана академиком ЭССР Э. Х. Тыугу для синтеза программ в *языке Утопист* и ЯП на основе задания семантической модели ПрО решения математических задач в ППП. Метод синтеза семан-

---

[1] См.: *Вельбицкий И. В.* Технология программирования. Киев : Техника, 1984; *Lavrischeva E. M.* Developmentof Domestic Programming Technology // Journal Cybernetics and Systems Analyses. 2014. Vol. 50. № 3. May. P. 452—264.

тических программ в PL/1, Фортран, Assembler реализован методом подстановки прикладным задачам семантики их реализации в ЯП и операциями системы «ПРИЗ» в ОС единой серии[1].

Композиция программ. Композиционное программирование[2] служило способом объединения функций и данных такими цепочками: «данные — функция — имя», «функции — композиция — дескрипция». Программной поддержкой этих цепочек являлась система «ДЕФИПС», которая обеспечивала построение программ из функций, заданных на некотором множестве именованных данных, дескрипций и денотатов (значений). Семантика задается ординарными функциями обработки операций, интерфейсных функций, арных функций и именованных данных. Операции композиции — это подкласс стандартных композиций и декомпозиционных функций.

Сборка программ. Индустриальный выпуск сложных программ В. М. Глушков определил как сборочный конвейер Форда. По его мнению, фабрика оборудуется линиями программирования разных программ и их интеграции в новую программную структуру. Базисом сборки были модули — главный элемент программирования. Обмен готовыми модулями потребовал разработки *паспорта*, который содержал описание входных и выходных данных, а также операторов вызова (Call, RMI и др.) других модулей. Модули описывались в разных ЯП и имели паспорт, задающий связь (интерфейс) с другими модулями. Это описание упрощало процесс сборки разноязыковых модулей, записанных в широко используемых ЯП (Алгол, Кобол, Фортран, ПЛ-1, Модула и др.). Позднее (1994) для описания интерфейса были созданы язык IDL (Interface Definition language) и брокер объектных запросов (ORB) в системе CORBA в рамках объектного подхода. Описание интерфейса через посредников *stub* и *skeleton* стало основным элементом сборки любых модулей, объектов и компонентов.

Вопросом сборки модулей занимались Е. П. Жоголев, Е. М. Лаврищева, В. И. Орлов, В. Г. Волховер и др.[3] В 1980-х гг. бурно развивалось направление автоматизации прикладных систем и ППП. В Институте кибернетики Академии наук УССР был выполнен ряд проектов, финансируемых ГКНТ СССР. Это проекты: система автоматизации программ — «АПРОП» (Е. М. Лаврищева); комплекс программиста «ТКП» (И. В. Вельбицкий); система алгоритмических алгебр «Мультипроцессист» (Г. Е. Цейтлин) и др. Средства автоматизации разработаны и в ряде других научных центров СССР: система «ПРИЗ» (ЭССР, Э. Х. Тыугу), системы «Альфа», «Бета» (Новосибирск, А. П. Ершов), система автоматизации математических задач (ИПМ АН СССР, Д. Н. Корягин), система модульного программирования (МГУ, Е. А. Жоголев), система послойного проектирования ПО (Ростов-на-Дону, В. И. Фуксман) и др.

---

[1] См.: *Кахро М. И., Тыугу Э. Х.* Инструментальная система программирования на ЕС ЭВМ (ПРИЗ). М. : Финансы и статистика, 1981; *Тыугу Э. Х.* Концептуальное программирование. М. : Наука, 1984.

[2] См.: *Редько В. Н.* Композиции программ и композиционное программирование // Программирование. 1978. № 5. С. 17—26.

[3] См.: *Волховер В. Г., Иванов Л. А.* Производственные методы разработки программ. М. : Финансы и статистика, 1983.

При выполнении научных проектов в СССР сформировались разные аспекты ТП для производства прикладных систем, АСУ и ППП с использованием готовых модулей, названных компонентами повторного использования (КПИ). Выходило постановление Кабинета министров СССР «Программы — продукты производственного назначения» (1974) и постановления ГКНТ СССР о том, что программы общего назначения должны были сохраняться в фондах алгоритмов и программ республиканского значения (1976—1992). В рамках Всесоюзной программы по ТП проводилась разработка стандарта ГОСТ 9126 и качества ПС.

Отработка разных аспектов создания сложных программ и ППП осуществлялась на всесоюзных конференциях по ТП (1968, 1979, 1980, 1986, 1987, 1991, 1992 и др.). Наиболее распространенной тематикой были технология модульного создания систем и интерфейс связей модулей и программ. Эти вопросы рассматривались на Международной конференции «Интерфейс СЭВ — 1987». Краткая характеристика работ на конференциях по ТП в СССР приведена в приложении 4.

1.1.3. Технология программирования систем для ЭВМ

В 1960—1970-х гг. для решения разных народно-хозяйственных задач применялись новые отечественные ЭВМ («БЭСМ», «МЭСМ», «М-20», «Днепр-1», «Днепр-2», «Стрела» и др.). Для каждой их них сформированы ТП, включающие набор методов и средств для разработки сложных систем. К ним относились СП и средства отладки программ (на правильность синтаксиса и семантики). Позже появились методы тестирования отдельных программ и систем. С их помощью находились функциональные ошибки, отказы в связи с недоработками как ОС, так и изготовляемых систем.

Определение программной системы и методы ее построения

Под **системой** понимается совокупность взаимодействующих элементов, работающих совместно для достижения заданных целей. Система первоначально была иерархической структурой, в которой в качестве элементов могли быть подсистемы, каждая из которых функционировала самостоятельно и связывалась с другими для обмена данными. Так как работа системы зависит от аппаратуры ЭВМ, то формировалась системотехника как технология создания систем, охватывающая все аспекты создания и модернизации программных и технических средств этих систем.

Результаты, полученные при системном тестировании систем, оценивались на надежность и качество. На основе опыта программирования и тестирования систем были опубликованы монографии В. В. Липаева — «Надежность ПС» и «Качество ПС» и «Проектирование комплексов программ». В это же время вышла в свет работа А. Ф. Кулакова «Качество программ ЭВМ». Появилась теоретическая возможность оценивать качество технических средств и программ. Методы тестирования программ и систем независимо от используемой ЭВМ и видов программ проводили проверку правильности их работы. Сформировался определенный порядок проведения разработки систем — так называемые модели ЖЦ (итерационная, спиральная и др.), которые включают следующие этапы создания системы:

• определение целей и задач в техническом задании на разработку системы;

• проектирование структуры системы и разработка отдельных ее структурных элементов;

• тестирование элементов и систем;

• испытания системы на множестве данных и тестов;

• подготовка к тиражированию системы для передачи потребителям;

• эксплуатация системы и ее сопровождение;

• снятие с эксплуатации (утилизация).

В 1990-х гг. появится стандарт ЖЦ (1996), в котором были регламентированы этапы ЖЦ для разработки разных сложных программ, а в 2007 г. — окончательный вариант стандарта ЖЦ ICO/IEC 12207.

ТП специализированных ЭВМ

В рамках ВПК разрабатывались (в строгой секретности) специализированные ЭВМ («ПРА-6.0», «МАПА», «АРГОН», «АОУ6» и др.), а также военная техника (5Э89, Э26Б 5Э51 и др.) для решения функциональных задач космических кораблей, подводных лодок, самолетов и др.[1] Они использовались как радиолокационные средства:

• для обнаружения воздушных целей;

• управления средствами противовоздушной обороны;

• телекоммуникационных средств передачи данных на командные пункты и средств поражения противоракетной техники (ПРО и ПВО) и стратегии перехвата закрытых данных;

• оценки надежности систем обработки информации в режиме реального времени и др.

Было создано более 300 образцов военной техники, которые надо было поставить на решение конкретных задач ВПК. Для этого при Министерстве радиопромышленности СССР был создан комитет под руководством главного конструктора В. В. Липаева. Главная его задача состояла в том, чтобы на специализированных ЭВМ создать ПО, основанное на принципах разработки языков и программ как для универсальных ЭВМ. Для этих целей разрабатывались системы «ЯУЗА», «РУЗА» и «ПРОМЕТЕЙ». В НИЦЕВТ в бортовую машину «АРГОН» включалась система команд машин ЕС ЭВМ. Затем потребовалось разрабатывать новые интерпретаторы и кросс-системы, с помощью которых усовершенствовались процессы разработки функциональных программ на бортовых ЭВМ в ВПК. На этих машинах создавались специальные средства разработки, отладки и испытания программ в режиме разделения времени.

Разработка системы «ПРОТВА» проводилась по ТП, с помощью которой решались следующие задачи:

• планирование трудоемкости и длительности создания ПО;

• прогнозирование и оценивание реального состояния качества в зависимости от обнаруженных ошибок;

---

[1] *Липаев В. В.* Фрагменты истории развития отечественного программирования для специализированных ЭВМ в 50—80-е годы. М. : СИНТЕГ, 2003.

• оценка ресурсов ЭВМ по памяти и производительности для реализации ПС;

• прогнозирование и оценка состояния качества ПС, в зависимости от длительности и характеристик обнаруженных ошибок.

В качестве языков использовались: автокоды, макроязыки и ЯП. Для них строились СП и системы проверки правильности по аналогии с большими ЭВМ. Для программистов вводились рекомендации по размеру программного модуля (100—200 строк текста) и тестам проверки до 10 строк. Было сформировано правило: число условий в тестах для покрытия тестами структуры модуля пропорционально квадрату строк текста программы. Было получено ограничение на размер программного модуля не более 300 строк. Эти вещи делались для того, чтобы снизить сложность и повысить качество системы в целом.

При работе с проектом «ПРОМЕТЕЙ» наибольшее внимание уделялось вопросам взаимодействия модулей. Для этого в рамках проекта «ПРОМЕТЕЙ» использовалась система «АПРОП», которая реализовала интерфейсные связи разноязычных модулей. Она была адаптирована в среду системы «ПРОМЕТЕЙ». В ней были предусмотрены средства тестирования интерфейсов связываемых модулей и методы системного тестирования. Для системы «ПРОМЕТЕЙ» была разработана документация согласно требованиям ГОСТа. Система «ПРОМЕТЕЙ» была сдана государственной комиссии и передана в ЕРНУЦ (Ереван). Она успешно применялась на многих предприятиях ВПК и после эксплуатации представлена к премии Кабинета министров СССР (1985).

1.1.4. Развитие технологии сборочного программирования

Технология программирования развивалась по трем направлениям, сформулированным в статье В. М. Глушкова «Фундаментальные основы и технологии программирования» (Программирование. 1980. № 2):

• модульная система автоматизации производства программ («АПРОП») из стандартизированных программных заготовок в сложные системы[1];

• метод формализованных технических заданий для проектирования сложных программных комплексов «ПРОЕКТ»[2];

• *Р*-технология программирования для автоматизации проектирования систем средствами графического *Р*-языка для представления структур программ и данных в АСУ[3].

Модульная система автоматизации программ «АПРОП» разрабатывалась исходя из тезиса В. М. Глушкова — конвейерная сборка разнородных моду-

---

[1] См.: *Глушков В. М., Стогний А. А., Лаврищева Е. М.* Система автоматизации производства программ (АПРОП). Киев : Изд-во ИК АН УССР, 1976; *Лаврищева Е. М., Грищенко В. Н.* Связь разноязыковых модулей в ОС ЕС; *Лаврищева Е. М.* Вопросы объединения разноязыковых модулей в ОС ЕС // Программирование. 1978. № 1.

[2] См.: *Глушков В. М., Капитонова Ю. В., Летичевский А. А.* О применении метода формализованных технических заданий к проектированию программ обработки структур данных // Программирование. 1978. № 6. С. 5—12.

[3] См.: *Вельбицкий И. В.* Технология программирования; *Вельбицкий И. В., Ходаковский В. Н., Шолмов Л. И.* Технологический комплекс автоматизации программ на машинах ЕС ЭВМ и БЭСМ-6. М. : Статистика, 1980.

лей и интерфейсов (Е. М. Лаврищева) и в системе «ПРОЕКТ» (Ю. В. Капитонова, А. А. Летичевский). На этой основе создавались ППП (И. Н. Молчанов); ППП математического и статистического типов (И. В. Сергиенко, В. Н. Редько, А. С. Стукало); *Р*-технология (И. В. Вельбицкий), «ТЕРЕМ» (Н. М. Мищенко). Этими работами был внесен весомый вклад в сборочное создание ПП на ЕС ЭВМ.

Система АПРОП разрабатывалась по договору с Институтом приборостроения (Москва) в составе технологии создания программ для бортовых систем «ПРОТВА» в рамках Министерства радиопромышленности СССР, реализованной под руководством В. В. Липаева[1]. В эту систему были внедрены интерфейс и библиотека интерфейсных функций преобразования нерелевантных ТД, описываемых в модулях на разных языках и в разных платформах. Было разработано стандартное описание интерфейса, содержащее описание входных и выходных параметров связываемых между собой модулей. Это описание стало средством интеграции модулей и КПИ. Система «АПРОП» вошла в состав комплексов «ПРОТВА» и «РУЗА», которые реализовали технологию создания бортовых систем для ВПК. Комплекс «ПРОТВА» был представлен на премию СМ СССР (1985).

В ТП сформировался новый вид — сборочное программирование для объединения разнородных модулей (программ) в более сложные структуры. Эта технология развивалась и в отделе В. М. Глушкова для семейства трансляторов с широко используемых ЯП. Основные положения этого вида сборки базировались на идее выделения общих средств в ЯП ОС ЕС, системной реализации компонентов языковых процессоров и сборки из них трансляторов (СПТ ТЕРЕМ). В СПТ разработаны общие компоненты языковых процессов в классе языков ОС ЕС. СПТ включала ЯП ОС ЕС (Алгол, Кобол, ПЛ1 и др.) и построена для МВК с макроконвейерной организацией вычислений и средствами внесения изменений в программы ЯП. Более половины универсальных модулей для ЯП реализованы и вошли в ядро СПТ «ТЕРЕМ», которая сдана Госкомиссии в 1989 г.

В системе «ПРОЕКТ» реализован **метод формализованных технических заданий** для проектирования дискретных систем с применением теории дискретных преобразований. Основу метода составляет формальное описание функций системы с помощью языка L2, аналогичного языку Аналитик, и доказательства правильности функций с помощью дискретных преобразователей и с применением алгебраических операций и средств обработки разных структур данных. К структурам данных относятся простые (числа, символы) и сложные данные (массивы, указатели). Вводится понятие схемы программы, содержащей множество переходов и состояний. Дается описание алгоритма проектирования системы по данному методу и системы обработки данных с применением метода К. Флойда. Метод прошел внедрение на практике при разработки ПО системы «МАЯК».

---

[1] См.: *Липаев В. В.* Технология проектирования комплексов программ. М. : Радио и связь, 1982; *Липаев В. В.*, *Позин Б. А.*, *Штрик А. А.* Технология сборочного программирования. М., 1992.

***P*-технология** создана под руководством И. В. Вельбицкого для конструирования структур программ с помощью визуальных *P*-графов и схемной их реализации. Было разработано устройство синтаксического контроля программ, запатентованное и за рубежом. В нем выполнена структурная интерпретация синтаксиса и семантики ЯП с использованием *P*-языка. Затем был создан ТКП для проведения анализа программ в *P*-языке и комплекс РТК для машин «БЭСМ-6», «ЕС ЭВМ» и «СМ ЭВМ». РТК использовался в различных закрытых организациях СССР для обработки текстовой информации и генерации средств производства программ для АСУ, САПР и ИС.

1.1.5. Интерфейс и метод сборки в технологии программирования

Концепция интерфейса в рамках системы АПРОП включала межмодульный, межъязыковой и технологический. Первое определение понятия *интерфейса* и языка его описания сформулировано в проекте системы в 1976 г.[1] Идея интерфейса намного опередила зарубежные разработки. Был создан язык MIL (*Module Interface Language*) для интерфейса модулей. В зарубежной литературе интерфейс появился позднее. К нему относились программный интерфейс API (*Application Programs Interface*), языковый интерфейс IDL (*Interface Definition Language*), научный интерфейс SIDL (*Scientifically IDL*) и др.

Межмодульный интерфейс — это связующее звено между двумя объединяемыми программными объектами, выполняет функции передачи, приема и преобразования разнотипных данных. Ему затем соответствовал брокерный Stub, Skeleton в системе CORBA (1994)[2] , используемый и сейчас.

Межъязыковой интерфейс — средства представления и преобразования структур данных в ЯП с помощью примитивных функций системных библиотек, которые обеспечивают взаимно однозначную передачу данных между разнородными программами (например, преобразование матрицы по строкам в Фортране в матрицу по столбцам в PL/1 и обратно). Библиотека таких функций была создана в рамках системы «ПРОТВА» (1982—1987) для среды ОС ЕС (PL/1, Algol, Fortran, Cobol, Assembler) и передана в 52 организации СССР по актам внедрения.

Технологический интерфейс — совокупность методов и средств для поддержки процессов ТЛ, включая нормативные, методические документы и формы преобразования данных на процессах ЖЦ. Данный интерфейс включал операции контроля процессов, оценки заданных в требованиях показателей качества, изменения состояния продукта на процессах ЖЦ и др. Методика создания ТЛ (1987) проверена на шести линиях заказа АИС «Юпитер-470» и стала первой работой по формализации ТЛ разра-

---

[1] См.: *Глушков В. М., Стогний А. А., Лаврищева Е. М.* Система автоматизации производства программ (АПРОП).

[2] См.: *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. М. : Мир, 2002.

ботки ПС. Альтернативой ТЛ явились продуктовые линии (ProductLines/ ProductFamily, 2004) института SEI США (http://www.sei.com.edu/ productlines). Концепция технологического интерфейса автора, Г. И. Коваль и Т. М. Коротун соотмечена грамотами европейского конкурса «Интерфейс-СЭВ» (1987)[1].

Созданная нами концепция интерфейса модулей была автоматизирована. Роль генератора интерфейсных модулей-посредников выполняла подсистема реализации интерфейса в системе «АПРОП» (1975—1982). Она описана в монографии «Связь разноязыковых модулей в ОС ЕС» под авторством Е. М. Лаврищевой и В. Н. Грищенко. Эти средства использовались многими в стране и способствовали сокращению объема работ при сборке разноязыковых объектов.

Таким образом, было сформировано сборочное программирование, объектами которого являются любые готовые программы, модули и КПИ, а также интерфейс как способ объединения, комплексирования (по В. В. Липаеву), сборки ПС и семейств ПС из готовых объектов. Он занял свое место среди других методов программирования. Как показали исследования[2], сборка и сейчас остается главным инструментом на многочисленных фабриках программ и сервисов, работающих в среде операционных сред (MS.Net, CORBA, Java, VSphere, Unix, Linex и др.).

Академик А. П. Ершов считал, что «сборочное программирование эффективно, поскольку готовые запрограммированные модули позволяют быстро решить любые задачи из определенной проблемной области для ЕС ЭВМ и мини-, микро- и макро-ЭВМ»[3]. Сборка стала важным технологическим решением в индустрии создания продукции производственно-технического назначения.

Метод сборки разноязыковых программ:

- опубликован в зарубежной прессе *(Lavrishcheva E. M.* Modular design of large programs. Springer. 1980. Vol. 16. № 2. P. 244—249);
- защищен Е. М. Лаврищевой в докторской диссертации «Методы, средства и инструменты сборочного программирования» (1989);
- описан в монографиях «Связь разноязыковых модулей в ОС ЕС», «Сборочное программирование» (Е. М. Лаврищева, В. Н. Грищенко, 1991) и в книге «Технология сборочного программирования» (В. В. Липаев, Б. А. Позин, А. А. Штрик, 1992);
- развит в ОКМ моделирования ПС из объектов и компонентов и обеспечивает сборку КПИ и ПС (описывается в гл. 7 данного учебника);
- представлен в статье Е. М. Лаврищевой «Парадигмы программирования сборочного типа в программной инженерии» (УкрПРОГ. 2014. № 2—3. С. 121—133).

---

[1] *Коваль Г. И., Коротун Т. М., Лаврищева Е. М.* Технологический интерфейс для разработки систем // Труды Межд. конф. «Интерфейс-СЭВ», 1987. С. 97—110.

[2] В частности: *Лаврищева Е. М., Грищенко В. Н.* Сборочное программирование.

[3] *Ершов А. П.* Опыт интегрального подхода к актуальной проблеме ПО // Кибернетика. 1984. № 4. С. 11—21. См. также: *Его же*. Два облика программирования // Кибернетика. 1982. № 6. С. 122—125.

1.2. Технология модульного проектирования систем

Технология модульного программирования основывается на понятии модуля, который осуществляет преобразование множества исходных данных X во множество выходных данных Y и задается в виде отображения:

$$M : X \to Y.$$

На множества X, Y и отображение M накладывается ряд ограничений и дополнительных условий, позволяющих отделить модуль как самостоятельный программный элемент. Модуль обладает множеством свойств и характеристик. Эти свойства проявляются на трех основных процессах разработки системы: проектировании, разработке и выполнении[1].

Проектирование ПС включает определение следующих свойств модулей:

- выполнение одной или более взаимосвязанных функций;
- логическая законченность функции;
- независимость одного модуля от других (внутренняя логика данного модуля не связана с логикой работы других);
- замена отдельного модуля без нарушения структуры;
- вызов других модулей и возврат данных вызвавшему модулю;
- уникальность именования модуля и др.

Процесс разработки модулей основан на следующих свойствах:

- описание модулей на одном из ЯП и представление в виде КПИ;
- обязательное описание характеристик модуля в паспорте;
- задание ограничений на размер модуля.

Процесс выполнения ПС основывается на следующих требованиях к свойствам характеристик модуля:

- возможность использования КПИ в различных точках вариативности ПС;
- передача данных между модулями через вызов CALL;
- изменение передаваемых данных и др.

Не все свойства одинаково важны в процессе разработки ПС. Однако для модульного проектирования характерны две тенденции — усиление *внутренних связей* в модуле и ослабление *внешних связей*. Хорошо спроектированный модуль обладает значительно более сильными внутренними связями, чем внешними. При выполнении этого условия процесс сборки модулей можно рассматривать как сочетание неделимых программных единиц с определенными свойствами без учета их внутренней структуры. С учетом этого выделены свойства модулей, наиболее важные для процесса сборки:

1) логическая законченность процесса проектирования;
2) заменяемость отдельного модуля в ПС;
3) возврат управления вызывающему модулю;
4) обращение одного модуля к другим;
5) раздельная компиляция внешнего модуля;
6) использование модуля в различных точках ПС;
7) спецификация паспорта и текста модуля в ЯП.

---

[1] См.: *Лаврищева Е. М., Грищенко В. Н.* Связь разноязыковых модулей в ОС ЕС. — 1982, 136 с.

Приведенные свойства позволяют выбрать аксиоматический подход к определению модуля.

Модулем в рамках метода модульного программирования называется программный объект, характеризующийся свойствами 1—7 и реализующий отдельную функцию Это определение задает границы применения понятия «модуль».

Определение типов связей модулей

Под *типом связи* будем понимать: 1) связь по управлению; 2) связь по данным.

1. **Связь по управлению** характеризуется наличием или отсутствием среды ЯП и механизмом вызова.

Среда ЯП определяется как совокупность программных средств окружения модулей с ЯП, включающих:

- системные средства процессов выполнения библиотечных программ связи с ОС, вычисление стандартных функций, обработки внутренних структур данных и т.д.;
- внутренние структуры данных, точки передачи и управления.

Механизм вызова через CALL в ЯП.

Связь по управлению описывается следующей функцией:

$$CP = K_1 + K_2,$$

где K_1 — коэффициент механизма вызова; K_2 — коэффициент перехода от среды ЯП вызывающего модуля к среде ЯП вызываемого.

Для стандартного механизма вызова $K_1 = 1$, для нестандартного $K_1 = 1 + a$ ($a > 0$). Здесь a зависит от количества отличных от стандарта характеристик вызова.

К характеристикам вызова относятся: способ определения точки входа в вызываемый модуль; механизм передачи управления; формирование адреса возврата в вызывающий модуль; доступ к списку параметров; метод сохранения и восстановления регистров вызывающего модуля.

Коэффициент K_2 зависит от количества операций, необходимых для перехода от среды вызывающего модуля к среде вызываемого и наоборот. Аналитического выражения для K_2 не существует, но можно указать параметры, от которых он зависит. К ним относятся: количество библиотечных модулей, входящих в среду; количество выполняемых ими функций; количество структур данных, входящих в среду; ЯП вызывающего и вызываемого модулей.

Если вызывающий и вызываемый модули написаны на одном ЯП, то $K_2 = 0$. Для остальных случаев $K_2 > 0$.

2. **Связь по данным** между модулями. Различают регулярную и нерегулярную связь. Регулярная характеризуется целенаправленным обменом определенного множества данных при каждой активизации вызываемого модуля с помощью оператора вызова CALL. Нерегулярная — непостоянством обмениваемых данных через посредников.

К этому типу связи относится обмен данными с глобальными именами или внешними файлами. Эта связь характеризует *сложность* ПС, описываемую следующей функцией:

$$CI = \sum_{i=1}^{n} K_i F(x_i), \qquad (1.1)$$

где K_i — весовой коэффициент для i-го параметра; $F(x_i)$ — функция количества элементов простых типов для параметра x_i.

Коэффициенты $K_i = 1$ — для простых переменных и $K_i > 1$ — для сложных ТД. $F(x_i) = 1$, если x_i — простая переменная, и $F(x_i) > 1$ — для сложных ТД. Необходимо отметить, что в формулу (1.1) входят данные, принадлежащие к регулярной и нерегулярной информационной связи. Определение простых и сложных ТД будет приведено ниже.

1.2.1. Интерфейсы модулей и их функции

В модульном программировании важное место занимает интерфейс, под которым понимается взаимосвязь программных объектов (модулей, программ, языков и т.п.), обеспечивающая условия их совместного функционирования.

Интерфейс разрабатывается как самостоятельный ПП. В основе построения интерфейсов лежат:

- стандартизация объектов, их свойств и характеристик;
- разработка формализованных правил (механизмов) связи стандартизованных объектов;
- средства автоматизации механизмов связи объектов.

При этом имеет место несколько видов интерфейсов: межъязыковый, межмодульный, межпрограммный, пользовательский.

Межъязыковый интерфейс — это связь различных ЯП по типам содержащихся в них данных, методам их организации и способам отображения этих средств соответствующими системами программирования.

Основными функциями межъязыкового интерфейса является решение следующих четырех задач.

1. Обеспечение перехода от среды функционирования одного ЯП к среде функционирования другого. При связи разноязыковых модулей необходимо осуществить переход от среды ЯП вызывающего модуля к среде ЯП вызываемого. Перед возвратом управления необходимы обратные операции.

2. Передача управления между разноязыковыми модулями. Решение данной задачи связано с решением предыдущей.

Если реализованы средства перехода от одной среды функционирования к другой, то передача управления между разноязыковыми модулями не отличается от передачи управления между одноязыковыми. В этом случае задача для межъязыкового интерфейса сводится к контролю за последовательностью обращения от вызывающего модуля к вызываемому.

Необходимо отметить, что решение данной задачи предполагает механизм непосредственного взаимодействия модулей без дополнительных средств обеспечения передачи управления.

3. Обеспечение доступа к общим данным через механизмы нерегулярной информационной связи в зависимости от места расположения информации. Поэтому при модульном подходе основным механизмом обмена

является аппарат передачи данных через параметры оператора вызова CALL ЯП высокого уровня.

4. Реализация механизма передачи данных через параметры вызова. Задачей межъязыкового интерфейса в этом случае будет преобразование данных из списка фактических параметров вызывающего модуля к представлению, согласующемуся с описанием формальных параметров вызываемого модуля.

Преобразование направлено на устранение различий, как языковых, так и связанных с реализацией данных системами программирования. Оно выполняется перед и после выполнения вызываемого модуля.

Межмодульный интерфейс — это обеспечение связи модулей, записанных в разных ЯП, и управление модульными структурами:

1) разработка межмодульного интерфейса основана на решении задачи сопряжения пар модулей и задачи управления модулями;

2) описание разнородных модулей;

3) представление модулей в системах программирования с ЯП.

1. **Различия** в языковых средствах ЯП являются следствием неодинаковости синтаксического и семантического представления ТД ЯП, их функциональных возможностей. К ним относятся:

- механизм конструирования новых ТД, который отсутствует в языках Фортран, ПЛ/1, Кобол и имеется в языках Паскаль, Ада, Симула-67, Модула-2, Си, CLU;
- некоторые предопределенные типы, которые отсутствуют в некоторых ЯП (например, нет символьного типа в Фортране);
- формат представления разный (логический тип в языке ПЛ/1 представлен как битовая строка);
- динамические ТД отсутствуют в Фортране и Коболе и имеются в языках Паскаль, Ада, Симула-67 и др.;
- организация внешних файлов в ЯП различна;
- дескрипторы для представления структурных ТД имеются в некоторых ЯП и не требуются в Фортране и Коболе;
- представление некоторых структурных типов различается в различных ЯП (массивы в Фортране располагаются по столбцам, в других ЯП — по строкам).

2. **Проблемы сопряжения** связаны с описаниями модулей, вызваны несоответствием задания формальных и фактических параметров, и состоят в следующем:

- описания ТД, областей значений переменных, индексов массивов и т.п. задаются однозначно;
- с одним формальным параметром сопоставляется несколько фактических и наоборот;
- изменение порядка следования параметров.

3. **К проблемам реализации** модулей относятся:

- особенности передачи управления (наличие среды функционирования);
- различия во внутреннем представлении однородных ТД для различных СП;
- различия в структуре и организации внешней памяти для однородных файлов.

Из приведенного выше следует, что проблема передачи управления между разноязыковыми модулями носит не принципиальный характер, а является следствием реализации конкретных СП с ЯП. Это подтверждается также тем, что аналогичные проблемы для ЯП, реализованных на мини- и микроЭВМ, практически отсутствуют или незначительны.

Управление построением и обработкой модульных структур программ как одной из задач межпрограммного интерфейса основывается на реализации следующих четырех функций.

1. Комплектование программных средств различного уровня сложности. Этот процесс выполняется несколькими процессами. На каждом приходится иметь дело с программами различного уровня сложности, готовности и отлаженности. Комплексирование (сборка) разнородных программ происходит в рамках задачи межпрограммного интерфейса.

2. Обеспечение построения различных видов программных структур (простая и динамическая структуры). Построение этих программных структур и составляет вторую задачу управления модульными структурами.

3. Анализ правильности функционирования модульной структуры. Для выполнения операций над модульными структурами необходимо исследовать модульные структуры объектов для обеспечения правильности их функционирования. В частности, эти исследования состоят в выполнении операций определения доступности модулей и их именования, анализа циклов в последовательностях обращений между модулями и др. Поэтому в управлении обработкой модульных структур заключается третья задача межпрограммного интерфейса.

4. Проведение отладки и тестирования межмодульных переходов способствует построению модульных структур. При решении этих задач используются средства межъязыкового интерфейса и средства управления модульными структурами. Средства сопряжения модулей позволяют обеспечить тестирование передаваемых параметров, а средства управления модульными структурами — отслеживание цепочек выполнения модулей.

К задачам управления модульными структурами относятся:

1) выполнение операций над модульными структурами;
2) построение новых структур на основе модулей;
3) построение отладочной среды модульной структуры;
4) сборка модулей в агрегат с учетом решения задач 1)—3).

При решении первых трех задач четвертая является тривиальной, и ее обычно выполняет специальный компонент ОС — редактор связей в ОС и др.

1.2.2. Определение модульной структуры ПС

Для представления ПС из модулей используется математический аппарат теории графов, в котором графом G называется пара объектов $G = (X, \Gamma)$, где X — конечное множество, называемое множеством вершин, а Γ — конечное подмножество прямого произведения $X \cdot X \cdot \Gamma$, называемое множеством дуг графа. Из данного определения следует, что граф G фактически является мультиграфом, так как две его вершины могут быть соединены несколькими дугами. Для отличия таких дуг они нумеруются целыми положительными числами.

Программная структура, состоящая из модулей, описывается ориентированным графом. В нем каждая дуга соответствует оператору CALL и соединяет вершину, соответствующую вызывающему модулю, с вершиной, соответствующей вызываемому модулю.

Определение 1.1. Модель модульной структуры ПС — это объект, описываемый тройкой $T = (G, Y, F)$, где $G = (X, \Gamma)$ — ориентированный граф, являющийся графом модульной структуры; Y — множество модулей, входящих в программный агрегат; F — функция соответствия, ставящая каждой вершине X-графа элемент множества Y.

Функция F отображает X на Y:

$$F: X \to Y. \tag{1.2}$$

В общем случае элементу из Y может соответствовать несколько вершин из X, что характерно для динамической структуры ПС.

Определение 1.2. Модульной структурой ПС называется пара $S = (T, \chi)$, где T — модель модульной структуры ПС; χ — характеристическая функция, определенная на множестве вершин X графа модулей G.

Значение функции χ определяется следующим образом:

- $\chi(x) = 1$, если модуль, соответствующий вершине $x \in X$, включен в состав ПС программного агрегата;
- $\chi(x) = 0$, если модуль, соответствующий вершине $x \in X$, не включен в состав ПС и к ней имеются обращения из других модулей.

Определение 1.3. Две модели модульных структур $T_1 = (G_1, Y_1, F_1)$ и $T_2 = (G_2, Y_2, F_2)$ тождественны, если $G_1 = G_2$, $Y_1 = Y_2$, $F_1 = F_2$. Модель T_1 изоморфна модели T_2, если $G_1 = G_2$, между множествами Y_1 и Y_2 существует изоморфизм φ, а для любого $x \in X \Rightarrow F_2(x) = \varphi(F_1(x))$.

Определение 1.4. Две модульные структуры $S_1 = (T_1, \chi_1)$ и $S_2 = (T_2, \chi_2)$ тождественны, если $T_1 = T_2$ и $\chi_1 = \chi_2$. Модульные структуры S_1 и S_2 называются изоморфными, если T_1 изоморфна T_2 и $\chi_1 = \chi_2$.

Понятие изоморфизма модульных структур и их моделей необходимо для введения уровня абстракции, на котором определяются операции над модульными структурами. Для изоморфных объектов операции будут интерпретироваться одинаково без ориентации на конкретный состав модулей. Данные операции определяются над парами (G, χ).

Введенные выше определения описывают модульные структуры общего вида. Далее рассматриваются структуры специального вида, особенности которых состоят в следующем:

1) граф модульной структуры G имеет один или несколько компонентов связности, каждая из которых представляет ациклический граф, т.е. не содержит ориентированных циклов;

2) в каждом компоненте выделена единственная вершина, которая называется корневой и характеризуется тем, что не существует входящих в нее дуг и соответствующий ей модуль программного агрегата выполняется первым (для данного компонента связности);

3) циклы допускаются только для случая, когда соответствующий некоторой вершине модуль имеет рекурсивное обращение к самому себе. Обычно такая возможность реализуется компилятором с соответствующего ЯП и данный тип связи не рассматривается межмодульным интер-

фейсом. Поэтому такие дуги не включаются в граф. Исключение из рассмотрения других типов циклов связано с тем, что некоторые модули должны будут помнить историю своих вызовов, чтобы правильно вернуть управление. А это противоречит свойствам модулей;

4) пустой граф G_0 соответствует пустой модульной структуре.

В дальнейшем под *графом модульной структуры* понимается схема модулей, удовлетворяющая указанным выше условиям. Типичный пример графа приведен на рис. 1.1.

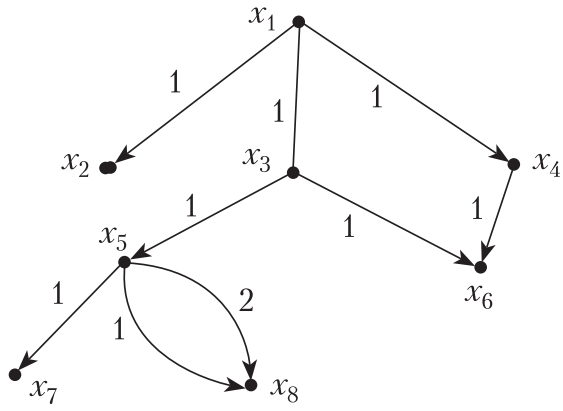


Рис. 1.1. **Пример графа модульной структуры программы**

Вершины x_1, x_2, ..., x_8 составляют множество X. Все дуги пронумерованы. Из модуля, соответствующего вершине x_5, имеются два обращения (два оператора вызова) к модулю, соответствующему вершине x_8. Множество дуг графа имеет вид $\Gamma = \{(x_1, x_2, 1), (x_1, x_3, 1), ..., (x_5, x_8, 1), (x_5, x_8, 2)\}$. В дальнейшем этот граф будет использоваться для иллюстрации операций над модульными структурами. Так, на рис. 1.2 показана структура графа, соответствующая модулям x_5 и x_6, а на рис. 1.3 — модульные структуры для трех видов сегментов.

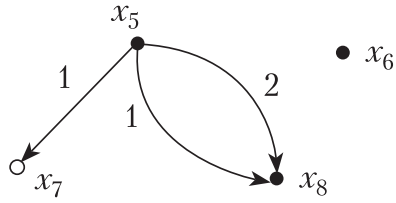


Рис. 1.2. **Граф структуры модулей x_5 и x_6**

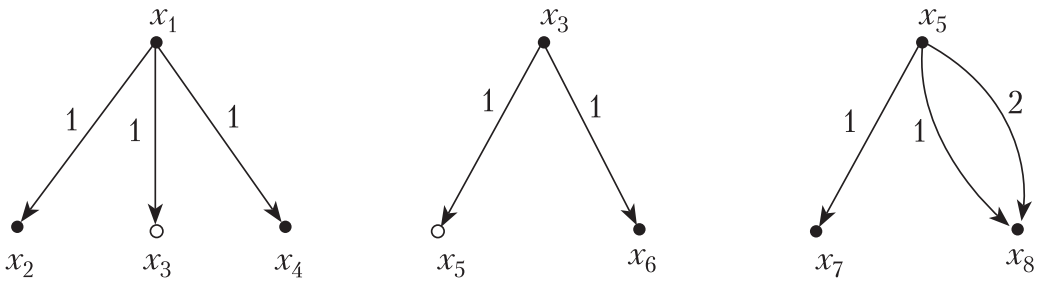


Рис. 1.3. **Графы модульных структур для трех видов сегментов**

1.2.3. Матричное представление графов из модулей

Для определения основных операций над модульными структурами используем матричное представление их графов. Матричные представления используются в различных работах. Например, матрицы вызовов и матрицы достижимости эквивалентны матрицам смежности ориентированных графов.

В настоящей работе в качестве матричного представления используется матрица вызовов. Элемент матрицы m_{ij} определяет количество обращений (операторов вызова) из модуля, соответствующего индексу i, к модулю, соответствующему индексу j. Кроме матрицы вызовов для дальнейшего анализа используется характеристический вектор, для каждого компонента i которого $V_i = \chi(x_i)$. Для графа модульной структуры, приведенной на рис. 1.1, характеристический вектор и матрица вызовов имеют вид

$$V = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Проведем анализ матриц вызовов и характеристических векторов для графов модульных структур, соответствующих различным типам программных агрегатов. Для модулей с графами, представленными на рис. 1.2, векторы и матрицы имеют следующий вид:

$$V_5^m = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, M_5^m = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix};$$

$$V_6^m = (1), M_6^m = (0).$$

Только один элемент характеристического вектора равен единице и только в одной строке матрицы находятся ненулевые элементы. Для рис. 1.3 векторы и матрицы записываются в таком виде:

$$V_3^s = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, M_3^s = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; V_1^s = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$M_1^s = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}; V_5^s = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, M_5^s = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Для программы с графом, представленным на рис. 1.1, характеристический вектор и матрица вызовов совпадают с V и M соответственно и определяются в формуле (1.2). Все элементы V равны единице.

В комплексе программ характеристический вектор и матрица вызовов имеют следующий вид:

$$V^c = \begin{pmatrix} V_1^p \\ V_2^p \\ \dots \\ V_n^p \end{pmatrix}, M^c = \begin{pmatrix} M_1^p & 0 & \dots & 0 \\ 0 & M_2^p & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & M_n^p \end{pmatrix}.$$

Здесь V_i^p и M_i^p $(i = \overline{1, n})$ обозначают характеристический вектор и матрицу вызовов для графа i-й программы, входящей в комплекс.

В дальнейшем матричное представление используется для анализа операций над модульными структурами.

1.2.4. Отношение достижимости модулей графов

Пусть $G = (X, \Gamma)$ — граф модульной структуры, x_i, x_j — вершины, принадлежащие X. Если в графе G существует ориентированная цепь от x_i к x_j, то вершина x_j — достижима из вершины x_i. Справедливо следующее утверждение: если вершина x_j достижима из x_i, а x_l — из x_j, то x_l достижима из x_i. Доказательство этого факта очевидно. Рассмотрим бинарное отношение на множестве X, которое определяет достижимость между вершинами графа. Введем обозначение $x_i \to x_j$ для достижимости вершины x_j из x_i. Отношение транзитивно. Обозначим через $D(x_i)$ множество вершин графа G, достижимых из x_i. Тогда равенство

$$\overline{x}_i = \{x_i\} \bigcup D(x_i)$$

определяет транзитивное замыкание x_i по отношению к достижимости.

Докажем следующую теорему.

Теорема 1.1. Для выбранного компонента связности графа модульной структуры любая вершина достижима из корневой, соответствующей данному компоненту, т.е. выполняется равенство (x_l — корневая вершина)

$$\overline{x}_l = \{x_l\} \bigcup D(x_l) = X.$$

Доказательство. Предположим, вершина x_i $(x_i \in X)$ недостижима из x_1. Тогда $x_i \notin \overline{x}_l$ и множество $X' = X \setminus \overline{x}_l$, непусто. Поскольку выбранный компонент графа связанный, то существуют вершина $x_j \in \overline{x}_1$ и цепь $H(x_i, x_j)$, ведущая от x_i к x_j. Исходя из ацикличности графа G, в X' должна существовать простая цепь $H(x_l, x_j)$, где в вершину x_l не входят дуги (данная цепь может быть пустой, если X' состоит только из x_i). Рассмотрим цепь $H(x_l, x_j) = H(x_l, x_i) \bigcup H(x_i, x_j)$. Это означает, что модуль x_j достижим из вершин x_1 и x_i и обе вершины не содержат входящих дуг. А это противоречит определению графа модульной структуры с единственной корневой вершиной. Теорема доказана.

Основное требование для обеспечение достижимости — это отсутствие неориентированных циклов в графе. Исходя из графа на рис. 1.4, отмечаем, что граф содержит ориентированный цикл и модули, соответствующие вершинам x_4, x_5, x_6, никогда выполняться не будут. Таким образом, результаты теоремы 1.1 усиливают требование отсутствия ориентированных циклов в графе модулей.

Для анализа матричного представления отношения достижимости графа модульной структуры рассмотрим граф матрицы достижимости, приведенной на рис. 1.1.

Коэффициент $a_{ij} = 1$, если модуль, соответствующий индексу l, достижим из модуля, соответствующего индексу i. Следующие результаты основаны на известной теореме из теории графов.

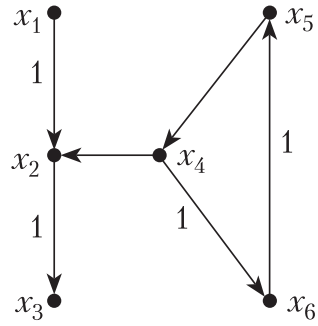


Рис. 1.4. **Граф с ориентированным циклом**

$$A = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Теорема 1.2. Коэффициент m_{ij} l-й степени матрицы смежности M определяет количество различных маршрутов, содержащих l дуг и связывающих вершину x_i с вершиной x_j-ориентированного графа.

Рассмотрим следующие три следствия из этой теоремы[1].

Следствие 1.1. Матрица $\bar{M} = \sum_{l=1}^{n} M^t$, где M — матрица смежности ориентированного графа с n вершинами, совпадает с точностью до числовых значений коэффициентов с матрицей достижимости A.

[1] Доказательство этой теоремы приводится в работе (книга): *Лаврищева Е. М., Грищенко В. Н.* Сборочное программирование. Киев : Наукова думка, 1991. 287 с.

Доказательство. В ориентированном графе, содержащем n вершин, максимальная длина пути без повторяющихся дуг не может превышать n. Поэтому последовательность степеней матрицы смежности M^i, где $i = 1, 2, ..., n$, определяет количество всех возможных путей в графе с количеством дуг $\leq n$. Пусть коэффициент $\overline{m}_{ij}$ матрицы M отличен от нуля. Это означает, что существует степень матрицы M^i, у которой соответствующий коэффициент m_{ij} также отличен от нуля. Следовательно, существует путь, идущий от вершины x_i к x_j, т.е. вершина x_j достижима из x_i. Данное следствие определяет связь матрицы вызовов графа модульной структуры, совпадающей с матрицей смежности M, с матрицей достижимости A и определяет алгоритм построения последней.

Следствие 1.2. Пусть для некоторого i в последовательности степеней матрицы смежности M^i существует коэффициент $m_{ii} > 0$. Тогда в исходном графе существует ориентированный цикл.

Доказательство. Пусть $m_{ii} > 0$ для некоторого l. Следовательно, x_l достижима из x_i, т.е. существует цикл. Согласно теореме 1.2 данный цикл имеет l дуг (в общем случае повторяющихся).

Следствие 1.3. Пусть n-я степень матрицы смежности M^n ациклического графа совпадает с нулевой матрицей (все коэффициенты равны нулю).

Доказательство. Если граф ациклический, то в нем максимально простой путь не может иметь больше чем $n - 1$ дуг. Если в M^n имеется коэффициент, отличный от нуля, то должен существовать путь, состоящий из n дуг. А этим путем может быть только ориентированный цикл. Следовательно, все коэффициенты M^n для ациклического графа равны нулю. Данное следствие предоставляет необходимое и достаточное условие отсутствия циклов в графе модульной структуры.

Для ациклических графов отношение достижимости эквивалентно частичному строгому порядку. Транзитивность отношения достижимости рассмотрена выше. Антисимметричность следует из отсутствия ориентированных циклов: если вершина x_j достижима из x_i, то обратное неверно. Введем обозначение $x_i > x_j$, если вершина x_j достижима из вершины x_i.

Пусть $G = (X, \Gamma)$ — ациклический граф, соответствующий некоторой модульной структуре. Рассмотрим убывающую цепь элементов частично упорядоченного множества X:

$$x_{i1} > x_{i2} > ... > x_{in}, ...,$$

где через знак «>» обозначено отношение достижимости. Поскольку X конечно, то цепь обрывается $x_{i1} > x_{i2} > ... > x_{in}$. Вершина x_{in} не имеет исходящих дуг, т.е. элемент x_{in} минимальный (ему соответствует модуль, который не содержит обращения к другим модулям). Максимальный элемент во множестве X — корневая вершина.

1.2.5. Операции построения модульных структур

Ранее было отмечено, что операции над модульными структурами выполняются на уровне абстракции, определяемом понятием изоморфизма модульных структур. Над изоморфными структурами операции выполня-

ются одинаково, поэтому преобразование модульных структур будет рассматриваться как изменение их графов и характеристических функций, т.е. $S = (G, \chi)$.

Пусть $S_1 = (G_1, \chi_1)$ и $S_2 = (G_2, \chi_2)$ — две модульные структуры с графами $G_1 = (X_1, \Gamma_1)$ и $G_2 = (X_2, \Gamma_2)$ соответственно. Введем следующие обозначения:

- $D(x)$ — множество вершин, достижимых из вершины x;
- $D^*(x)$ — множество вершин, из которых достижима вершина x.

Для одинаковых вершин, входящих в графы G_1 и G_2, будут использоваться одинаковые обозначения.

Рассмотрим основные операции над модульными структурами.

Операция **объединения**

$$S = S_1 \bigcup S_2$$

предназначена для формирования модульной структуры комплекса и формально определяется следующим образом: S_1 и S_2 — любые модульные структуры, которые удовлетворяют условиям

$$G = G_1 \oplus G_2, X = X_1 \oplus X_2, \Gamma = \Gamma_1 \oplus \Gamma_2,$$

где символ $\oplus$ обозначает прямую сумму, и

$$\chi(x) = \chi_1(x), \text{ если } \chi \in X_1,$$
$$\chi(x) = \chi_2(x), \text{ если } \chi \in X_2.$$

Одинаковые вершины, входящие в G_1 и G_2, операцией объединения модульных структур рассматриваются как разные объекты. Поэтому характеристический вектор и матрица вызовов для модульной структуры S определяются так:

$$V_{1,2} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, M_{1,2} = \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix},$$

где $V_{1,2}$ и $M_{1,2}$ — характеристические векторы и матрицы вызовов для модульных структур S_1 и S_2 соответственно. Операция объединения ассоциативна, но не коммутативна — порядок следования операндов определяет порядок выполнения компонентов комплекса. Необходимо отметить, что если операнды S_1 и S_2 удовлетворяют условиям определения модульных структур, то результат S будет удовлетворять тем же требованиям.

Операция объединения модульных структур увеличивает число компонентов связности соответствующего графа. Кроме того, графы структур, соответствующие операндам, сами могут иметь несколько компонентов связности. Для остальных операций графы модульных структур операндов и результата имеют единственный компонент связности.

Рассмотрим операцию **соединения**. Через x_i и x_j обозначим корневые вершины для графов G_1 и G_2 модульных структур S_1 и S_2 соответственно. Если данные структуры удовлетворяют условиям:

- множество $X' = X_1 \bigcap X_2$ непусто;
- вершина $x_j \in X'$ и $\chi(x_j) = 0$;

• $D^*(x) \cap D(x) = 0$ для каждого $x \in X'$, где $D^*(x) \in X_1$ и $D(x) \in X_2$, то операция соединения, обозначаемая

$$S = S_1 + S_2, \tag{1.3}$$

определяется следующим образом:

$$G = G_1 \cup G_2, X = X_1 \cup X_2, \Gamma = \Gamma_1 \cup \Gamma_2,$$

и для характеристической функции χ выполняются:

- $\chi(x) = \chi_1(x)$, если $x \in X_1 \setminus X'$;
- $X(x) = \max(\chi_1(x), \chi_2(x))$, если $x \in X'$;
- $\chi(x) = \chi_2(x)$, если $x \in X_2 \setminus X'$.

Первое условие означает, что в графах G_1 и G_2 имеются общие вершины. Согласно второму условию корневая вершина G_2 принадлежит общей части и для S_1 объект, соответствующий x_j, еще не включен в модульную структуру. Третье условие запрещает существование циклов в графе результата. Действительно, если существует $x_n \in D^*(x) \cap D(x)$, то $x_n > x$ и $x > x_n$, что означает существование цикла. Так как не все S_1 и S_2 удовлетворяют приведенным выше условиям, то операция частичная.

Определим принадлежность результата операции соединения к классу рассматриваемых модульных структур. Поскольку X' непусто, то граф G имеет один компонент связности. Корневой вершиной графа G является x_i. Сам граф G не имеет ориентированных циклов, т.е. ацикличен. Таким образом, S принадлежит к классу рассматриваемых модульных структур.

Операция соединения не коммутативна и в общем случае не ассоциативна. Чтобы показать последний факт, рассмотрим результат $S = (S_1 + S_2) + S_3$, где корневые вершины графов G_2 и G_3 входят в состав вершин графа G_1 и $X_2 \cap X_3 \neq 0$. Тогда результат $S_2 + S_3$ не определен.

Рассмотрим операцию **проекции**. Пусть $S_1 = (G_1, \chi_1)$ — модульная структура и $x_i \in X_1$. Операция проекции модульной структуры на вершину графа S_1, обозначаемая как $S = P_{rxi}(S_1)$, определяется следующим образом:

$$G(X, \Gamma), X = \overline{x}_i, \Gamma = \{(x_i, x_j, K) \mid x_i, x_j \in X\},$$

и для характеристической функции

$$\chi(x) = \chi_1(x), \text{ если } x \in X.$$

Операция проекции определяет из модульной структуры S_1 подструктуру S. Проверим принадлежность S классу рассматриваемых модульных структур. Если граф модульной структуры S_1 связан и ацикличен, то теми же свойствами будет обладать и граф S. Существует единственная корневая вершина x_i в графе G. Таким образом, модульная структура S принадлежит рассматриваемому классу.

Операция **разности** для модульных структур определяется следующим образом. Пусть $S_1 = (G_1, \chi_1)$ — модульная структура и $x_i \in X_1$. Операция разности выполняется между модульной структурой и ее проекцией

на вершину x_i соответствующего графа (x_i не является корневой вершиной графа G_1). Формально операция разности модульной структуры

$$S = S_1 - P_{r_{xi}}(S_1)$$

определяется следующим образом:

$$G = \{X, \Gamma), X = (X_1 \setminus \bar{x}_i) \cup X'$$

$$\Gamma = \{(x_i, x_j, K) \mid x_i, x_j \in X\},$$

где множество X' состоит из таких элементов, для которых

$$X' = \{x'_j \mid (x_l \in X_1 \setminus x_i) \ \& \ (x'_j \in \bar{x}_i) \ \& \ (x_l, x'_j, K) \in \Gamma\}.$$

Характеристическая функция χ определяется так:

- $\chi(x) = \chi_1(x)$, если $x \in X_1 \setminus \bar{x}_i$;
- $\chi(x) = 0$, если $x \in X'$.

Во множество X включаются вершины, которые не вошли во множество $\bar{x}_i$, и те из вершин $\bar{x}_i$, в которые входят дуги из вершины $X_1 \setminus \bar{x}_i$ (множество X'). Характеристическая функция для элементов $x' \in X'$ равна нулю. Операция разности модульных структур определена так, чтобы быть обратной к операции соединения, т.е. чтобы выполнялось равенство

$$S - P_{r_{xi}}(S) + P_{r_{xi}}(S) = S.$$

Проверим принадлежность S, определяемой в формуле(1.3), к классу рассматриваемых модульных структур. Если граф G, связан и ацикличен, то этими же свойствами будет обладать граф G_1. Корневая вершина G совпадает с корневой вершиной G_1. Таким образом, S удовлетворяет условиям определения модульной структуры, приведенным в п. 1.2.2.

Пусть S^* обозначает множество модульных структур, заданное на прямом произведении $G^* \cdot \chi^*$, где G^* и χ^* — соответственно множество графов и характеристических функций. Обозначим через Ω множество введенных операций над модульными структурами и предикаты C и R, рассмотренные выше:

$$\Omega = \{\bigcup, +, -, P_r, C, R\}.$$

Этим мы определяем частичную алгебраическую систему $U = (S^*, \Omega)$ над множеством модульных структур. Согласно определению типов программных агрегатов и операций над модульными структурами необходимо отметить следующие операции:

- объединения — применяется для программ и комплексов, результатом является комплекс;
- соединения — применяется для модулей и сегментов, результатом может быть сегмент или программа;
- проекции — используется для программ и сегментов, результатом является модуль или сегмент;
- разности — применяется для программ и сегментов, результатом может быть модуль или сегмент.

1.2.6. Процесс построения модульных структур

Рассмотрим основные задачи построения программных агрегатов и алгоритмы их решения.

1. Дано множество модулей, входящих в состав программы, и имя главного модуля. Построить модульную структуру программы.

2. Дано множество модулей, входящих в состав программы. Имя главного модуля неизвестно. Построить структуру из модулей.

3. Дано множество модулей, реализующих некоторые функции предметной области, и имя главного модуля для одной из программ. Построить модульную структуру программы.

4. Даны множество модулей реализации функции предметной области и последовательность имен главных модулей нескольких программ. Построить модульную структуру для комплекса программ.

5. Заменить в модульной структуре один или несколько модулей новыми.

6. Выделить из модульной структуры объекты и включить их в другую структуру. В этой задаче под объектами понимается любая часть модульной структуры, т.е. любой подграф ее графа.

При решении данных задач будут использоваться введенные операции над модульными структурами. На практике применяется матричное представление объектов, а в качестве операций рассматриваются преобразования соответствующих матриц. Так, множества и характеристические функции представляются векторами, графы — матрицами вызовов и т.д. Аналогичные представления применяются и для объектов, используемых в алгоритмах решения задач. Технические аспекты реализации матричных представлений объектов и операций принципиальных трудностей не представляют и в данном учебнике не рассматриваются.

Рассмотрим решение каждой из поставленных задач, применяя операции над модульными структурами.

Задача 1. Пусть X^p — множество вершин графа, соответствующее множеству модулей программы. Упорядочим его так, чтобы для каждых $x_i, x_j \in X^p$, из условия, что модуль, соответствующий x_i, вызывает модуль, соответствующий x_j, следовало, что $i > j$. Если X^p таким образом не удается упорядочить, то в графе модульной структуры возникнут циклы, что противоречит необходимости ацикличности графа.

Обозначим вершину, соответствующую главному модулю, через x_1. Основной метод решения данной задачи состоит в постепенном наращивании модульной структуры «сверху вниз». С каждым модулем, соответствующем вершине x_i, связана модульная структура $S_i^m = (G_i^m, \chi_i^m)$, где $G_i^m = (\chi_i^m, \Gamma_i^m)$. Множество χ_i^m включает x_i и вершины, соответствующие модулям, к которым имеются обращения из данного модуля. Множество Γ_i^m соответствует множеству вызовов из модуля, соответствующего x_i. Характеристическая функция для x_i равна единице и нулю — для остальных вершин. Если из модуля, соответствующего x_i, нет обращения к другим модулям, то $X_i^m = \{x_i\}$, $\Gamma_i^m = 0$, $\chi(x_i) = 1$. Пусть Y — множество вершин графа, для которых $\chi = 0$.

Алгоритм приведен ниже:

1. Ввод начальных значений: $S = S_i^m$; $Y := X_1^m \setminus \{x_1\}$.
2. Если $C(S) = 1$, то переход на п. 6.
3. Выберем первый элемент $x_i \in Y$. Множество Y непусто и конечно.
4. Наращивание модульной структуры $S := S + S_i^m$ и изменение множества $Y := (Y \setminus \{x_i\}) \cup (x_i^m \setminus \{x_i\})$.
5. Переход на п. 2.
6. Выход.

Докажем корректность и сходимость данного алгоритма. На шаге 4 выполняется операция соединения структур. Эта операция корректна, так как ее условия выполняются:

$$x_i \in X \cap X_i^m,$$

$$\chi(x_i) = 0, X_i^m(x_i) = 1,$$

циклы отсутствуют согласно упорядоченности множества X^p.

Рассмотрим изменение множества Y на шаге 4. Согласно результатам п. 1.2.4 в графе модульной структуры обязательно существуют вершины, из которых исходят дуги. Исходя из этого факта и конечности множества X^p, следует, что существует x_i, такое что $Y \setminus \{x_i\} = 0$ и $x_i^m \setminus \{x_i\} = 0$. В результате условие шага 2 истинно и осуществляется переход на п. 6 — завершение работы алгоритма.

Необходимо отметить, что данный алгоритм имеет практическое воплощение в процессе комплексирования, основанном на применении специальных компонентов ОС — редактора связей, построителя задач, комплексатора и т.д.

Задача 2. Для решения второй задачи достаточно упорядочить множество X^p, как и для условия первой. Тогда первый элемент X^p соответствует главному модулю и можно воспользоваться алгоритмом решения первой задачи.

Задача 3. Для решения третьей задачи проведем аналогичное упорядочение модулей ПрО. Единственное условие состоит в том, чтобы $x_1 \in X$ соответствовал главный модуль программы. Тогда можно воспользоваться алгоритмом решения первой задачи.

Задача 4. Пусть X^c — множество вершин, соответствующих модулям реализации функции ПрО, и $x' = \{x_1, x_2, ..., x_n\}$ — множество вершин, соответствующих главному модулю программы ($X' \subset X^c$). Алгоритм решения следующий.

1. Начальное значение: $i := 1$, $S := 0$.
2. Выбрать $x_i \in X'$ и построить модульную структуру с главным модулем, соответствующим корневой вершине x_i. Построение выполняется согласно алгоритму третьей задачи. Пусть S_i^p — соответствующая модульная структура.
3. $S := S \cup S_i^p$. На данном шаге используется операция объединения модульных структур.
4. $i := i + 1$. Если $i \le n$, то переход на шаг 2.
5. Выход.

Корректность и сходимость данного алгоритма очевидны и не требуют доказательства.

Задача 5. Пусть $S = (G, \chi)$ — модульная структура с графом $G = (X, \Gamma)$. Необходимо заменить модуль программы, соответствующий вершине $x_i \in X$, новым модулем x_i'. Сложность данной задачи состоит в том, что с заменой модуля может измениться модульная структура S, так как x_i в общем случае является корневой вершиной для некоторого графа. Данный алгоритм будет иметь следующий вид.

1. Построить модульную структуру S_i' для корневой вершины x_i' согласно алгоритмам задач 1—3 (предполагается, что множество модулей для данной структуры известно до ее построения).
2. $S := S - P_{rx_i}(S)$. Исключается соответствующая часть модульной структуры S.
3. Переобозначим x_i' через x_i, а S_i' — через S_i. Выполним операцию: $S = S + S_i$.
4. В результате предыдущей операции в графе G модульной структуры могут существовать вершины с $\chi = 0$, т.е. $C(S) = 0$. Тогда необходимо применить алгоритм первой задачи, полагая в качестве множества Y множество таких вершин.

Выполнение операции соединения на шаге 3 должно удовлетворять необходимым условиям.

Задача 6. Данная задача подобна задаче 5. Пусть $S_1 = (G_1, \chi_1)$ — исходная модульная структура с графом $G_1 = (X_1, \Gamma_1)$ и $x_i \in X_1$. Граф $G_2 = (X_2, \Gamma_2)$ модульной структуры $S_2 = (G_2, \chi_2)$ имеет вершину x_i с $\chi(x_i) = 0$. Тогда алгоритм решения данной задачи следующий.

1. $S_i = P_{rx_i}(S_1)$.
2. $S := S_2 + S_i$. Операция соединения модульных структур должна удовлетворять необходимым условиям.
3. Если $C(S) = 0$, то выполнять действия, аналогичные шагу 4 пятой задачи.

Исходя из приведенных выше алгоритмов, для решения задач 1—6 использовались четыре операции над модульными структурами.

1.2.7. Отладка и тестирование модулей

Применение данных методов предполагает, что отдельные модули, входящие в модульную структуру, прошли процесс автономной отладки и тестирования. На средства автономной отладки и тестирования никаких ограничений не накладывается.

Отладка модульной структуры

Задачи отладки модульных структур состоят в проверке правильности построения модульной структуры и выполнения программного агрегата, соответствующего данной модульной структуре. Рассмотрим эти задачи более подробно.

Проверка правильности построения модульной структуры. Существует два способа проверки. Первый основан на анализе результатов процесса редактирования связей (сборки модулей), выполняемого специальной про-

граммой ОС. Данный способ позволяет выявить грубые ошибки — отсутствие модулей в модульной структуре, к которым есть обращения. Эти ошибки — следствие реального отсутствия модулей или неверного имени в операторе вызова LINK.

Второй способ основан на анализе самой модульной структуры, который заключается в следующем:

1) визуальный анализ графа модульной структуры. Проверку правильности построения непосредственно осуществляет сам разработчик ПС. Отображение графа модульной структуры на экране терминала или вывод его на печать;

2) анализ матриц, описывающих модульные структуры и основанных на результатах п. 1.2.3. Для проверки правильности построения модульных структур используются матрицы вызовов и достижимости. Результат анализа — установка существования циклов, числа маршрутов и достижимости между каждой парой модулей (данная информация используется для определения количества маршрутов при тестировании).

Перейдем к рассмотрению второй основной задачи отладки модульных структур.

Проверка правильности выполнения модульной структуры. Решение данной задачи тесно связано с реализацией проблем межъязыкового интерфейса. Проверка правильности выполнения модульной структуры предполагает отслеживание в динамике последовательности передач управления и данных между взаимодействующими модулями. Такая возможность позволяет фиксировать цепочки выполняемых модулей и выполнять трассировку передаваемых данных. Результаты этих операций отображаются на экране или выводятся на печать. На основе их анализа определяются:

- правильность последовательности вызовов модулей в соответствии с графом модульной структуры и в зависимости от входных данных;
- правильность передаваемых данных между взаимодействующими модулями согласно описанию их типов в списке формальных и фактических параметров;
- последний выполняемый модуль в момент аварийной ситуации при выполнении программного агрегата;
- некоторые виды зацикливания в модулях при выполнении программных агрегатов.

Реализация этих функций возможна только при применении шестого метода комплексирования — использовании промежуточных модулей связи, в которые включаются средства для отладки модульных структур агрегатов. Они предназначены для фиксации момента входа в вызываемый модуль и выхода из вызываемого модуля; отображения и хранения значений входных параметров согласно описаниям их ТД при передаче управления вызываемому модулю; отображения и хранения значений выходных (и, возможно, входных) параметров согласно описаниям их ТД при возврате управления вызывающему модулю.

Средства отладки могут включать и дополнительные возможности: изменение значений передаваемых параметров при выполнении программного агрегата с использованием режима диалога; отображение файлов

печати (значений параметров, цепочек вызовов модулей и т.п.) на экран после работы каждого модуля или агрегата в целом.

Тестирование модулей и сложных структур

Тестирование заключается в динамической проверке поведения программы на конечном множестве тестовых данных, выбранных из входного пространства, на соответствие ожидаемому поведению.

Динамическое тестирование всегда предполагает выполнение программы, а *конечное* — создание большого количества тестов, которые могут быть выполнены в ограниченные сроки наблюдения за поведением программы.

Методы тестирования в основном отличаются подходами к выбору множества тестовых данных из входного пространства. Основная цель тестирования — обнаружить дефекты в ПС и установить ее функциональную пригодность, удобство применения, производительность и др.

Тестирование на протяжении процесса разработки сложной структуры из модулей выполняется на нескольких уровнях. Для каждого определяются категория объектов тестирования (ПС, компоненты, отдельные модули) и набор проверяемых тестируемых характеристик.

На каждом уровне тестирование повторяется многократно, образуя циклы: тестирование — исправление — повторное тестирование.

В современной практике тестирования все виды действий, начиная с планирования до оценки результатов тестирования, должны интегрироваться в четко определенный, документируемый и контролируемый процесс тестирования. Это облегчает взаимодействие между разработчиками, группой тестирования и руководством проекта, а также позволяет сделать процесс видимым, повторяемым и измеряемым.

Связь тестирования с другими видами проверки. Тестирование имеет много общего с процессами верификации и валидации (*V&V*). Общность процесса тестирования с процессами *V&V* заключается в единстве состава и структуры планов, рекомендуемых стандартом IEEE Std. 829, а также объектов и применяемых методов. Отличие этих процессов состоит в условиях их применения. Тестирование выполняется *всегда*, для всех объектов ПО системы независимо от ее критичности. Процессы *V&V* в современной трактовке стандартов IEEE Std. 1012 и ISO/IEC 12207 — поддерживающие процессы, которые могут применяться к выбранным объектам тестирования для проверки планов тестирования и подтверждения того, что выполненное тестирование адекватно уровню критичности ПС. По отношению к процессу тестирования *V&V* выполняет контрольную функцию и подтверждает соответствие объектов установленным требованиям.

Тестирование ПС тесно связано с отладкой и собственно программированием, но охватывает гораздо более широкий круг проблем и участников — программистов, тестировщиков, аналитиков и инженеров по качеству.

Виды и уровни тестирования. Традиционно выделяются три уровня тестирования ПО: автономное или модульное (*unit testing*), интеграционное (*integrating testing*) и системное (*system testing*). В стандарте ISO/IEC 12207 прослеживаются *четыре* уровня тестирования:

- модульное (в процессе «Построение ПО»);
- интеграционное (в процессе «Сборка ПО»);
- тестирование ПС (как процесс);
- системное (в процессе «Испытания ПС»).

Модульное тестирование предполагает проверку функционирования модулей в изоляции друг от друга. Оно обычно выполняется разработчиками с доступом к коду и чередуется с отладкой. Объектами тестирования являются отдельные процедуры, программные модули и компоненты, состоящие из тесно связанных модулей.

Цели модульного тестирования — обнаружить дефекты в реализации функций объектов и подтвердить соответствие объекта спецификациям проекта (техническому проекту).

Наиболее полно вопросы систематического модульного тестирования освещены в стандарте IEEE 1008—87 «Standard for Software Unit Testing».

Интеграционное тестирование предназначено для проверки правильности взаимодействия между программными объектами, протестированными автономно.

Современные систематические стратегии интеграции определяются архитектурой ПС и моделью разработки (обычно итерационной с приращениями).

Интеграционное тестирование выполняется при каждой сборке новой версии ПС с целью выявления дефектов в интерфейсах между интегрируемыми компонентами и подтверждения их соответствия проекту архитектуры ПС.

Тестирование ПС заключается в проверке функционирования интегрированной версии системы в моделируемой среде. Цели тестирования на этом уровне — выявить дефекты в реализации внешних функций ПС и подтвердить соответствие спецификации функциональным требованиям.

Системное тестирование ориентировано на оценку качества сопряжения модулей ПС с другими не программными компонентами современных систем.

На этом уровне тестирования производится проверка соответствия ПС целям качества, установленным в требованиях к системе, таким как надежность, устойчивость, производительность и др., а также *внешним интерфейсам* с другими системами, средой, аппаратным обеспечением. Большая часть функциональных отказов и дефектов должна быть идентифицирована и устранена на предыдущих уровнях тестирования. Здесь могут быть выявлены дефекты, связанные с неудовлетворительными техническими характеристиками функционирования всей системы.

Задачи системного тестирования часто объединяют в единый процесс. При этом тестирование технических характеристик ПС должно выполняться отдельно.

1.3. Сборочное программирование и перспективы развития

Основу сборочного программирования составляют метод сборки, интерфейс и КПИ. Метод сборки — это способ соединения разноязычных объек-

тов в ЯП, который базируется на теории спецификации и отображении типов и структур данных ЯП передаваемых между модулями разнородных данных[1].

Модульный посредник осуществляет интерфейс с другим модулем через оператор вызова (CALL, RPC, RMI и др.) с формальными параметрами, которые проверяются на соответствие ТД. Результат согласования — операции эквивалентного преобразования данных в модуле-посреднике для связываемых модулей.

Сущность сборочного программирования состоит в том, что он является:

- одним из методов программирования и подчиняется общим закономерностям и функциям;
- формой поддержки повторного использования КПИ в ПС за счет стандартизации описания и паспортизации данных.

Объекты сборки, модули или другие готовые элементы могут обладать общими свойствами и методами, например образовывать классы в ЯП (в C++ несколько библиотек классов).

Необходимые условия применения этого метода:

- наличие большого количества разнообразных КПИ;
- паспортизация объектов сборки;
- наличие набора правил взаимосвязи объектов и линий с операциями установления связей между КПИ в процессе сборки ПС.

1.3.1. Отечественные линии продуктов в АИС

Подход к построению ТЛ сформировался при выполнении Всесоюзного проекта АИС «Юпитер» (1982—1991) автоматизации Военно-морского флота СССР. Разработан метод технологической подготовки работ для создания специальных ТЛ[2].

ТЛ комплектуется из процессов, которые соответствуют некоторой ПрО, стандартных инструментов и модулей реализации специфики функций ПрО системными средствами и нормативно-методическим обеспечением. Набор процессов ТЛ создается с учетом требований межведомственного стандарта ГОСТ 3918—1999. Процессы поддерживаются отобранными методами, средствами и инструментами, которые преобразуют элементы процессов, потом тестируют и выполняют в среде проекта.

Для ТЛ подбираются ГоР, КПИ, средства и инструменты реализации отдельных функций или частей программ. На проекте АИС «Юпитер» было разработано пять ТЛ в комплекте документов для разработки сложных программ для Военно-морского флота. Каждый исполнитель собирает модули, КПИ по операциям ТЛ. В проекте, как на фабрике программ, появились новые инженерные роли — менеджер, разработчик, верификатор, валидатор, эксперт, тестировщик, сборщик, оценщик качества и др. Каждый из них выполнял соответствующую работу на созданных ТЛ.

---

[1] См.: *Лаврищева Е. М., Грищенко В. Н.* Сборочное программирование. Основы индустрии программных продуктов. Киев : Наукова думка, 2009. — 471 с.

[2] См.: *Лаврищева Е. М.* Технологическая подготовка разработки систем : Препринт / Изд-во ин-та кибернетики им. В. М. Глушкова. Киев, 1987; *Ее же.* Технологическая подготовка и программная инженерия // УСиМ. 1988. № 1.

1.3.2. Перспективы развития ТП (по А. П. Ершову)

Академик А. П. Ершов выступил на конференции «Технология программирования» в 1986 г. Ниже приводится фрагмент его выступления[1].

«Следует различать ТП как технологическую теорию, как конкретный способ организации и осуществления создания, распространения и сопровождения ПП и как процедуру индивидуальной деятельности профессионала, разрабатывающего или принимающего участие в разработке ПП.

Технологии и методологии — это всегда наука, в то время как метод входит в технологию составной частью.

Говоря о технологии профессионального, производственного программирования, следует отметить принципиальную важность отчуждаемости и тиражирования ПП. Сколько-нибудь технология начинается тогда, когда она охватывает ЖЦ ПП.

ТП — это совокупность методологических положений, организационно-административных установлений, инструментально-технических средств, их информационного и ПО, регламентирующая и поддерживающая производственную деятельность людей, вовлеченных в процесс создания, распространения и сопровождения ПП.

Законченная ТП должна:

- охватывать весь ЖЦ ПП;
- способствовать применению методологии, повышающей уровень достоверности, надежности и доказательности программирования;
- обеспечивать организационную управляемость и контролируемость производственных процессов;
- обеспечивать устойчивость ПП по отношению к смене технических средств;
- обеспечивать развиваемость ПП в связи с изменением условий функционирования целевой системы, использующей этот продукт, и других условий».

Пример достигнутой ТП на мировом уровне приведен фирмой IBM.

Ориентиром для развития ТП в советских условиях является статья А. П. Ершова в журнале «Программирование» (1986, № 3). В ней он говорит о трех поколениях интегральной промышленной ТП и ее перспективах.

«**Первое поколение** (*организационное программирование*) 1975—1985 гг.

ЯП не формализован. Переход от прототипа к программной версии не формализован.

ЯП — Фортран, Кобол, ПЛ/1, Ассемблер.

База знаний отсутствует.

Развитие продукта — версионное.

Второе направление (*сборочное программирование*) 1985—1995 гг.

Язык спецификации регламентирован.

Переход от протопипа к промышленной версии регламентирован. Язык разработки — формализованный язык высокого уровня со средствами модуляризации.

ЯП объединен с языком разработки, допуская комплексирование с ассемблерными модулями...»

Третье направление (*доказательное программирование*) 1995—2005 гг.

Язык разработки формализован и содержит систему формальных преобразований, необходимых для доказательства программ.

---

[1] Всесоюзная конференция ВПК1 «Программирование» (1968) и ВПК2 (1979). Труды ВПК1, ВПК2.

ЯП объединен с языком разработки.

БД проекта машинизирована.

Применение ЭВМ в проекте — полное.

Развитие продукта — эволюционное — *адаптивное*.

...Было бы полезно выработать норматив по технологии *второго поколения*, который не затрагивая конкретного методологического или языкового наполнения, унифицировал бы:

- общую этапность разработки ПП;
- нормативы производительности и надежности;
- организационно-документационную структуру;
- вычислительную инструментальную среду;
- **межмодульный интерфейс**, поддерживающий принцип *сборочного программирования...*»

Идеи академика А. П. Ершова получили широкое развитие в нашей стране и за рубежом. В 1995 г. созданы системы CORBA и UML. В CORBA реализован брокер ORB, который осуществляет сборку разноязыковых объектов и обработку обмениваемых данных.

На данный момент приведена конкретная реализация процесса сборки КПИ в системе .NET (Приложение 2)[1]. В ней показан процесс сборки и преобразования разных ТД. Процесс сборки используется в системах общего назначения (IBM, VS.Net, CORBA, Oberon и др.) в виде системных фабрик AppFab и коммерческих системах (ЕПАМ, ИБК и др.).

Контрольные вопросы и задания

1. Дайте определение ТП.
2. В чем суть ТП?
3. Определите основы модульного программирования.
4. Определите пути развития технологии систем, агрегатов.
5. Определите смысл систем программирования с ЯП.
6. Что такое синтез, сборка и композиция программ?
7. Дайте определение технологии модульного проектирования.
8. Определите тестирование модульных структур.
9. Назовите виды тестирования программ, КПИ, агрегатов.
10. Рассмотрите основы сборочного программирования.
11. Опишите метод сборки модулей с использованием интерфейса.
12. Какие виды программирования по А. П. Ершову вы знаете?
13. Что такое доказательное программирование?

---

[1] См.: *Радецкий И. О.* Один подход к обеспечению взаимодействия сред MS.Net и Eclipse // Проблемы программирования. 2011. № 2. С. 45—52.

Глава 2
ПРОГРАММНАЯ ИНЖЕНЕРИЯ / SOFTWARE ENGINEERING (SE) ПРОГРАММНЫХ ПРОДУКТОВ

Альтернативой термина ТП является термин **«программная инженерия»** (SE), впервые произнесенный на конференции НАТО в 1968 г. П. Науром. SE развивалась многие годы в направлении совершенствования инженерии разработки ПП и систем с учетом формируемого ЖЦ ПО, представленного позднее в стандарте ISO/IEC 12207—1996, 2007. В 2001 г. комитеты ACM и IEEE выпустили первый вариант SWEBOK, который регламентировал 10 областей знаний для разработки ПО разного назначения (www.swebok.com).

С давних пор все новые теории в физике, математике и других естественных науках подтверждались в практической жизни через инженерию приборов, механизмов, систем и компьютеров.

Инженерия — это совокупность обобщенных и систематизированных знаний или наука о методах, способах, приемах, средствах автоматизации и порядке их использования при производстве и применении некоторого продукта.

Этот термин используется во всех областях человеческой деятельности (хозяйственной, промышленной, экономической, финансовой, программной и др.). В компьютерных науках (Computer Science, CS) этот термин применяется к инженерии компьютеров, систем и ПО:

- Computer Engineering — инженерия компьютеров;
- Systems Engineering — инженерия систем;
- Software Engineering — инженерия ПО.

Официальное определение SE представлено комитетом ACM и IEEE и в Curricula-2004, и именно на него ориентируются в разных университетах мира. Комитет постоянно развивает ядро знаний SWEBOK в направлении добавления новых знаний, методов и средств для эффективного их применения в инженерии программ, реинженерии и реверсной инженерии ПП. Описанию SWEBOK посвящено очень много монографий[1].

---

[1] В том числе учебник автора «Методы и средства программной инженерии» (М. : Изд-во МФТИ, 2007). Он был представлен на конкурсе Microsoft в МГУ (2006) и получил призовое место, а также право к публикации. Следующий учебник, «Программная инженерия» (Киев : Академпериодика, 2009) прошел конкурс «Учитель-новатор» (2007), а ее автор получила соответствующий сертификат фирмы Microsoft.

Распространенный в СССР термин ТП по отношению к инженерии SE имеет более широкий смысл, поскольку ТП включает теорию, методы, средства и инструменты формального проектирования и реализации прикладных систем и ППП по математике и физике.

Стандарт SE (2001, 2004, 2007) ориентирован на разработку разного рода программных систем. Круг этих систем расширен в стандарте SE (2013) и включает следующие специализированные системы: оперативные системы, системы клиент-сервер, распределенные, параллельные и сетевые системы, системы мобильных вычислений и др.

Далее дается краткое определение названных инженерий исходя из материалов энциклопедии «Computer science» (1992), научных работ и стандартов.

2.1. Инженерия компьютерных систем

В инженерии компютерных систем можно выделить дисциплины Computer Science (Electronic Engineering, EE; Computer Engineering, CE; Software Engineering, SE; Engineering System, ES) и дисциплины обработки информации (Information Technology, IT; Information Systems, IS) (рис. 2.1).

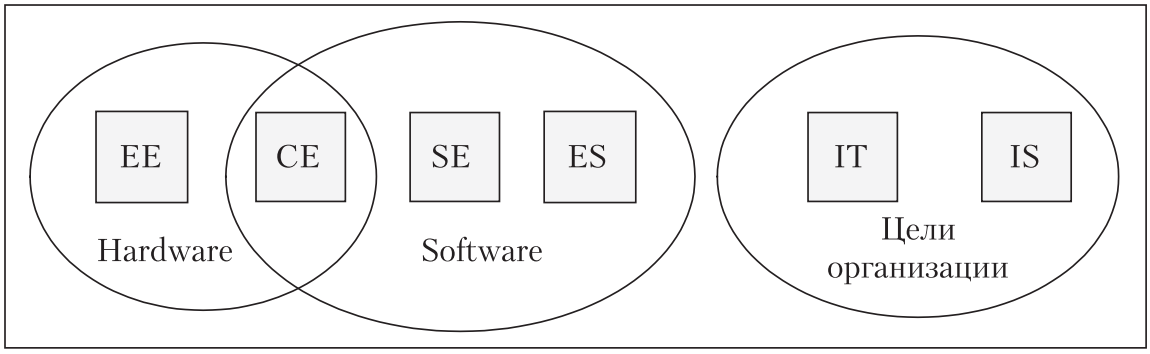


Рис. 2.1. **Дисциплины CS и обработки информации**

Инженерия компьютеров (*Computer Engineering*) — это теория и принципы построения компьютеров (фреймворки микропроцессов, кластеров, суперкомпьютеров и т.п.) и системного ПО (ОС, трансляторов, компиляторов, редакторов и т.д.) для управления их работой. Основу составляют теории Тьюринга, фон Неймана, теория автоматов, алгоритмов и кибернетики, а также математика, логика, теория анализа систем и т.п. Технология компьютеров одна из наиболее развитых с теоретической и практической точки зрения, как в направлении элементной базы (устройства, блоки, микросхемы, карты и т.п.), так и их системной организации (фреймворки, кластеры и др.). Уровень компьютерной технологии в сотни раз выше уровня системной и программной технологии. Компьютерами обеспечено более 70% населения планеты.

Инженерия систем (*System Engineering*) — это теория, методы и принципы построения ИС, автоматизированных систем, систем управления и компьютерных систем (*Computer Systems*). Средствами компьютерных технологий моделируются различные типы компьютерных приложений

(Computer Applications, Domain Engineering, Family Engineering) и новых средств управления ИС (ОС, БД, СУБД и др.). К средствам этой технологии относятся принципы и теория АСУ В. М. Глушкова[1], а также дисциплины (комбинаторика, графика и т.п.) и теория управления экономикой, финансами и др.

Программная инженерия (*Software Engineering*) — это система методов, способов и дисциплин планирования, разработки, эксплуатации и сопровождения ПО, обеспечивающих промышленное производство ПП. Она охватывает все аспекты создания ПО от начала формулировки требований, разработки продукта и до его использования и сопровождения. Базис инженерии — теория алгоритмов, теория программирования, теория вычислений и распределенной, телекоммуникационной обработки данных в сетях. При массовом производстве ПП используются теории планирования, регулирования процессов и ресурсов, тестирования, измерения результатов, оценки риска и достижения качества.

Информационные системы (*Information Systems*) — это компьютерные системы обработки разнообразной информации на предприятиях и в бизнес-деятельности. Они являются инструментом для управления и обработки информации, обеспечения продуктивности и эффективности работы систем информатизации. ИС и поисковые системы Интернета выступают инструментами накопления, поиска и отбора различных информационных ресурсов, необходимых для применения их различными пользователями.

Информационные технологии с 1990-х гг. стали базисом компьютерной инфраструктуры современных корпораций, предприятий и государственных органов управления, на которых решаются различные задачи обработки информации глобального типа. На их разработку и подготовку ИТ-специалистов, поддержку разных видов информационных систем Интернета и доступ к ним всех желающих выделяются огромные финансовые ресурсы. Цели и задачи построения информационных систем и технологий их применения сформулировал академик В. М. Глушков. В России создан ряд стандартов по информационным технологиям[2]. Эти стандарты являются руководством к применению в информационной инфраструктуре любого предприятия.

Центральное место среди дисциплин принадлежит SE (рис. 2.2).

Она дает теоретические и технологические средства разработки системной, компьютерной и программной инженерии, а также ИТ и ИС.

Таким образом, зарубежные технологии ориентированы на выпуск компьютеров, систем, ИТ, ИС и ПО для членов компьютерного мирового сообщества. Они образуют дисциплину информатики и используются в каждой стране.

---

[1] См.: *Глушков В. М.* Кибернетика, ВТ, информатика (АСУ) // Избр. труды : в 3 т. Киев : Наук. думка, 1990; *Его же*. Основы безбумажной информатики. М. : Наука, 1982.

[2] ГОСТ Р ИСО/МЭК 15504—1—2009. Информационные технологии. Оценка процессов. Ч. 1. Концепция и словарь. Ч. 2. Проведение оценки. Ч. 3. Руководство по проведению оценки.

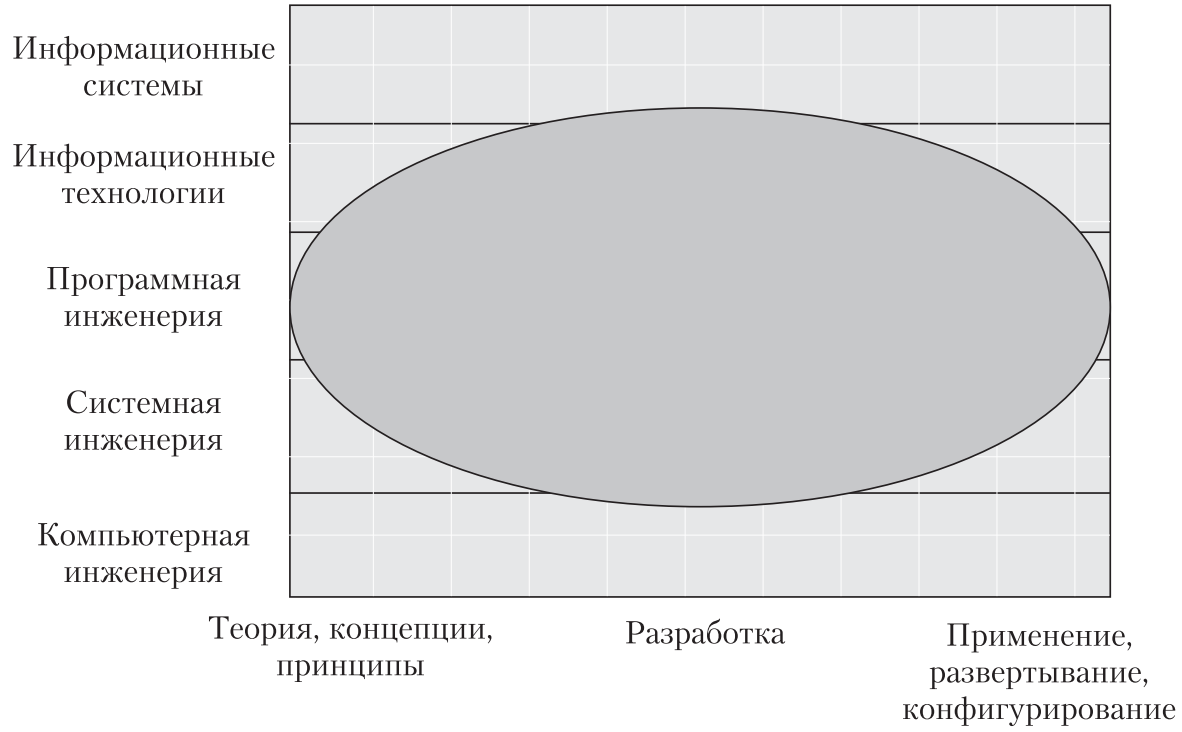


Рис. 2.2. **Набор дисциплин информатики**

2.2. Области знаний ядра SWEBOK

В начале 1990-х гг. мировое компьютерное сообщество пришло к необходимости систематизировать накопленные разнородные знания в отраслях компьютерных наук и информатики и зафиксировать их в виде *ядер знаний* SWEBOK.

Для создания ядра знаний по SE в 1993 г. совместными усилиями ACM (*Association for Computing Machinery*) и компьютерного сообщества института инженеров по электронике и электротехнике IEEE (Computer Society) был образован объединенный координационный комитет по ПИ SWECC (*Software Engineering Coordinating Committee*). В него вошли специалисты мирового уровня в области разработки ПО.

Основная задача SWECC — определение множества критериев и норм профессиональной практики SE, которые могли бы послужить основой для принятия практических решений при разработке ПО, сертификации продуктов и процессов, аттестации специалистов, а также организации учебного процесса.

В рамках комитета SWECC специализированные группы дали определение следующих направлений исследований:

1) ядра знаний и рекомендуемых практических приемов деятельности в SE;

2) норм профессиональной этики и стандартов по SE;

3) программ обучения студентов вузов по специальности SE.

SWEBOK (2004) определил работы по первому направлению. В 1998 г. был завершен Кодекс этики (как результат работы по второму направле-

нию), а в 2004 и 2007 гг. опубликованы новые рекомендации по подготовке программ обучения дисциплине SE, разработанные совместными усилиями IEEE и ACM[1].

Ядро знаний SWEBOK включает 10 различных *областей знаний*. Ядро знаний SE постоянно развивается. Текущая версия SWEBOK рассматривается как проект соответствующего международного стандарта, разработку которого осуществляет ISO/IEC J3333TC1/SC7 WG20. В 2005 г. вышла первая версия стандарта ISO/IEC TR 19759:2005 «Software Engineering — Guide to the Software Engineering Body of Knowledge — SWEBOK» и русский перевод программы Curricula «Рекомендации программной инженерии и информатики в университетах. Software Engineering 2004, Computing Curricula 2001». Ниже дается общее описание назначения этих областей знаний.

2.2.1. Программные требования

Область знаний «Программные требования» (*Software Requirements*) охватывает все аспекты **инженерии требований** — инженерной дисциплины извлечения, анализа, спецификации, документирования, проверки и управления требованиями к ПО на всех процессах ЖЦ систем.

Требования — это *свойства*, которыми должно обладать разработанное или адаптированное ПО, для того чтобы решать поставленные задачи.

С одной стороны, требования — это *множество функций*, которые должно выполнять ПО, чтобы удовлетворить потребности его заказчиков, с другой — это множество ограничений на *способ реализации* функций (например, язык, платформа, эффективные алгоритмы), с третьей — это множество условий выполнения функций ПО в определенной среде эксплуатации (с учетом конфигурации вычислительной среды, режима секретности и др.).

В SWEBOK различаются требования к продукту и процессу, функциональные и нефункциональные требования, а также системные требования, касающиеся взаимосвязанных программных и аппаратных подсистем. Требования могут иметь количественное представление (например, «количество запросов в секунду должно составлять...», «средний показатель ошибок не должен превышать 10% от объема вводимой информации» и т.п.). Значительная часть требований относится к характеристикам и подхарактеристикам качества: безотказность, надежность и др.

Эта область знаний включает следующие разделы:

- основы требований (*Requirements Fundamentals*);
- процесс инженерии требований (*Requirements Process*);
- выявление (извлечение) требований (*Requirements Elicitation*);
- анализ требований (*Requirements Analysis*);
- спецификация требований (*Requirements Specification*);
- проверка требований (*Requirements Validation*);
- практические соображения (*Practical Considerations*).

---

[1] Software Engineering 2004: Curriculum Guidelines for UndergraduateDegree Programs in Software Engineering. URL: http://sites.computer.org/ccse/SE2004Volume.pdf.

Раздел области знаний **«Процесс инженерии требований»** базируется на общем процессном подходе к работе с требованиями, в соответствии с которым каждое требование с момента его выявления должно идентифицироваться, определяться как элемент конфигурации и попадать в сферу непрерывного управления и *уточнения* с учетом мнений всех заинтересованных сторон и обеспечением компромисса между требованиями. Процесс инженерии требований требует поддержки, управления, оценки качества (зрелости) и совершенствования определенных ресурсов проекта, включая квалифицированный персонал — участников процесса (пользователей, аналитиков и др.).

Раздел **«Выявление требований»** — это извлечение информации из разных источников (договоров, материалов аналитиков по задачам и функциям системы и др.), проведение организационных мероприятий (наблюдений за выполнением бизнес-процессов, собеседований, собраний и др.) для определения спектра требований на разработку, согласование требований с заказчиком, исполнителем и всеми заинтересованными сторонами.

Раздел **«Анализ требований»** — изучение заявленных потребностей и целей пользователей, их классификация и преобразование под требования к системе, установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Требования классифицируются как функциональные и нефункциональные, внешние (по отношению к системе и ПО) и внутренние. *Функциональные требования* характеризуют функции системы и ее ПО, способы поведения ПО в процессе выполнения функций и методы преобразования входных данных в выходные (результаты). *Нефункциональные требования* определяют условия и среду выполнения функций (например, относительно защиты и доступа к БД, секретности, взаимодействия функциональных компонентов и др.). Моделирование требований и их декомпозиция до уровня программных компонентов завершаются на этапе проектирования архитектуры ПО и отражаются в специальном документе, по которому проводится *согласование требований* для достижения взаимопонимания между заказчиком и разработчиком.

Спецификация требований к ПО — это формализованное описание функциональных и нефункциональных требований к ПО и характеристикам его качества в *документе* спецификации требований, структура и содержание которого регламентируются стандартом, утвержденным для использования в проекте, например IEEE Std. 830:1998 «Рекомендованные приемы спецификации требований к ПО».

Проверка требований заключается в проведении процедуры верификации требований, сформулированных в документе спецификации требований. Ее цель — убедиться в том, что все требования заказчиков учтены и правильно поняты разработчиками. *Верификация требований* — это процесс проверки соответствия требований потребностям, их непротиворечивости, полноты и реализуемости, соответствия стандартам, а также их *пригодности для проведения проверки и подтверждения* соответствия с помощью приемочных тестов. Верификация проводится методами обзора или прототипирования, т.е. быстрого моделирования (отработки) отдельных требований на конкретном инструменте.

В разделе **«Практические соображения»** акцент делается на инженерии требований, управлении изменениями и трассировании программных требований, а также определении объема функциональности ПО согласно стандарту ISO/IEC 14143:1998 «Information technology — Software measurement — Functional size measurement».

2.2.2. Проектирование ПО

Проектирование ПО (*Software Design*) — процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы. Конечный результат проектирования — *проект* (или *дизайн*). В иностранной литературе по SE слово *design* определяет оба понятия — и процесс проектирования, и результат проектирования. Базовые понятия в области проектирования — цели, ограничения, возможные альтернативы, представления и решения. В соответствии с процессным подходом деятельность по проектированию организована в виде последовательных процессов «проектирования архитектуры системы» и «проектирования ПО», включая архитектурно-функциональное проектирование и детальное проектирование отдельных компонентов. Результат этого процесса — множество моделей и артефактов, фиксирующих основные решения.

При проектировании используются принципы и техники абстракции, связывания (*coupling*) и сцепления (*cohesion*), декомпозиции и модульности, инкапсуляции и сокрытия информации, разделения интерфейса и деталей реализации, необходимой достаточности, полноты и простоты.

Данная область знаний включает следующие разделы:

- основы проектирования ПО (*Software Design Fundamentals*);
- ключевые вопросы проектирования ПО (*Key Issue in Software Design*);
- структура и архитектура ПО (*Software Structure and Architecture*);
- анализ и оценка качества проекта ПО (*Software Design Quality Analysis and Evaluation*);
- нотации проектирования ПО (*Software Design Notations*);
- стратегия и методы проектирования ПО (*Software Design Strategies and Methods*).

Раздел **«Ключевые вопросы проектирования ПО»** определяет фундаментальные проблемы, поднимаемые вне зависимости от используемых методологий проектирования: обеспечение производительности, декомпозиция ПО на функциональные компоненты и их интеграция в систему, поведенческие аспекты (параллелизмы) в работе ПО, распределение компонентов в среде выполнения и их взаимодействие, организация данных и потоков управления, обработка событий и исключительных ситуаций, эргономические аспекты и др.

В разделе **«Структура и архитектуры ПО»** проект рассматривается как «многоаспектный» артефакт, произведенный в ходе процесса проектирования и отражающий множество независимых точек зрения на ПО (поведенческой, функциональной, структурной и др.).

Современная практика проектирования использует понятие архитектурных стилей и шаблонов проектирования.

Архитектурный стиль — это множество ограничений, определяющих семейство архитектур, удовлетворяющих этим ограничениям; или метамодель, определяющая высокоуровневую организацию ПО (например, клиент-серверные, послойные архитектуры и др.).

Шаблон проектирования (паттерн, *pattern*) — это эффективный способ решения характерных задач проектирования, образец типового решения общей проблемы в заданном контексте. Паттерн не есть готовое проектное решение, которое может быть непосредственно преобразовано в код, скорее это описание того, как решить задачу таким образом, чтобы решение можно было использовать многократно в различных ситуациях.

Архитектурный стиль включает макроархитектуру ПО и микроархитектуру ПО и использует другие паттерны: *структурные* («контейнер», «адаптер», «компоновщик» и др.), *креативные* («строитель», «фабрика», «прототип», «одиночка» и др.), *поведенческие* («интерпретатор», «посредник», «наблюдатель» и др.). Главная задача паттернов — это описание решений целого класса абстрактных проблем независимо от применяемого ЯП, что облегчает взаимодействие между разработчиками, использующими унифицированную терминологию.

Повторное использование архитектурных решений и компонентов — формирование *линий* (*семейств*) *продуктов* (*product lines*) на основе общего дизайна. В ООП используются *фреймворки* (каркасы), частично завершенные программные подсистемы, которые могут быть надлежащим образом дополнены в соответствии со спецификой решаемой задачи.

В разделе **«Анализ и оценка качества проекта ПО»** рассматриваются проблемы обеспечения и оценивания качества, непосредственно касающиеся проекта ПО. Делается различие между атрибутами качества, наблюдаемыми во время работы ПО (эффективность, безопасность, удобство применения и др.), и атрибутами качества, присущими проекту ПО как таковому (концептуальная целостность, корректность, полнота и др.).

Для анализа качества проекта рекомендуются *методики* статического анализа, моделирования и прототипирования, а также *меры и метрики* для оценки проекта по различным аспектам — размеру, структуре, функциям и качеству проектирования. Очевидно, что спектр метрик зависит от выбранного подхода к проектированию. В SWEBOK они классифицируются как *функционально ориентированные* (структурные) метрики проекта (на базе анализа иерархических диаграмм представления проекта) и *объектно ориентированные* (на базе диаграмм классов).

В разделе **«Нотации проектирования»** представлены две категории нотаций артефактов проекта — структурные и поведенческие.

Структурные нотации, как правило, графические, используются для представления структурных аспектов проекта — компонентов и их взаимосвязей, элементов архитектуры и их интерфейсов. К ним относятся формальные языки спецификаций и проектирования: ADL (*Architecture Description Language*), UML (*Unified Modeling Language*), IDL (*Interface Description Language*), диаграммы «сущность — связь» — ERD (*Entity-Relation Diagrams*), диаграммы классов и компонентов, диаграммы Джексона, структуры и схемы классов (*CRC Cards*) и др.

Поведенческие нотации используются для описания динамических аспектов функционирования систем и их компонентов. К ним относятся диаграммы потоков данных, диаграммы действий (потоков управления), диаграммы взаимодействия объектов, диаграммы переходов состояний, таблицы решений, блок-схемы, формальные языки спецификации, псевдокод, язык проектирования PDL и др.

Раздел **«Стратегии и методы проектирования ПО»** представляет общие стратегии проектирования и специализированные методы, предлагающие собственные нотации для описания различных аспектов проекта.

К *общим стратегиям* относятся: «разделяй и властвуй», «пошаговое уточнение», «снизу вверх», «сверху вниз», «абстракция данных», «применение паттернов и языков паттернов», «применение итеративного и инкрементного подходов» и др.

Функционально ориентированные (*структурные*) *методы* базируются на структурном анализе, структурных картах, диаграммах потоков данных и др. Они ориентированы на идентификацию функций и их уточнение сверху вниз и последующую разработку диаграмм потоков данных и описание процессов.

В методах *объектно ориентированного проектирования* ключевую роль играют наследование, полиморфизм и инкапсуляция, а также абстрактные структуры данных и представления объектов.

Методы *проектирования от данных* во главу угла при проектировании ставят структуры данных, которыми должны манипулировать программы, и базируются на применении методов Джексона, Варнье — Орра и др. для задания входных и выходных данных структурными диаграммами.

Методы *компонентного проектирования* ориентированы на построение программ-компонентов с четко описанными интерфейсами и зависимостями, которые могут разрабатываться автономно (с ориентацией на их повторное использование) и интегрироваться для совместного функционирования в системе, взаимодействуя посредством интерфейсов. Компонентное проектирование образует базис для других видов проектирования и программирования, например сервисно ориентированного.

В SWEBOK упоминаются также другие подходы к проектированию, включая *формальные* методы проектирования и методы *трансформации*.

2.2.3. Конструирование ПО

Конструирование ПО (*Software Construction*) — создание работающего ПО путем комбинирования методов кодирования, верификации, автономного и интеграционного тестирования и отладки компонентов. Границы между конструированием, проектированием и тестированием ПО существенно варьируются в зависимости от моделей ЖЦ и методологий разработки ПО.

Область знаний включает следующие разделы:

- основы конструирования ПО (*Software Construction Fundamentals*);
- управление конструированием ПО (*Managing Construction*);
- практические соображения (*Practical Considerations*).

Основу конструирования составляют концепции минимизации сложности ПО, ожидания изменений (готовности к изменениям), облегчения проверки, соблюдения стандартов (внешних и внутренних).

Управление конструированием базируется на моделях конструирования, планировании деятельности по конструированию и измерении артефактов процесса конструирования.

Модели конструирования описывают спектр выполняемых действий, их последовательность (взаимосвязь) и результаты. Виды моделей определяются применимыми стандартами ЖЦ, методологиями и приемами, например экстремальное программирование или RUP.

Планирование состоит в определении порядка создания и интеграции компонентов, распределении задач между исполнителями и действий по обеспечению качества ПО. Планирование регламентируется выбранной для проекта моделью конструирования.

Измерение в конструировании ориентировано на количественную оценку объема кода (разработанного, модифицированного, повторно используемого, удаленного), сложности кода; оценку статистических данных, касающихся верификации (инспекции) кода, интенсивности выявления и устранения дефектов в коде, а также трудоемкости конструирования. Данные измерений используются для управления конструированием с целью совершенствования самого процесса конструирования и повышения качества кода.

Практические соображения в данной области знаний касаются **проблем конструирования ПО** с помощью:

- лингвистических нотаций, основанных на строковых инструкциях и предложениях для представления сложных программных конструкций;
- формальных нотаций, основанных на точных, однозначных и формальных (математических) определениях;
- визуальных нотаций, которые основаны на непосредственном визуальном представлении элементов ПО (как, например, язык UML);
- кодирования с соблюдением хорошего *стиля кодирования*, включающего соглашения об именовании и структурировании исходного кода, использование классов, перечисляемых типов, переменных, поименованных констант; организацию исходного текста; использование структур управления, обработку ошибок и исключительных ситуаций, организацию доступа к параллельно используемым ресурсам, документирование кода и другие вопросы. В процессе конструирования должны использоваться внешние стандарты применяемых ЯП (например, Ада 95, C++ и др.), языков описания данных (XML, SQL и др.), средств коммуникации (COM, CORBA и др.), интерфейсов компонентов (POSIX, IDL, APL), сценариев (UML и др.);
- применения модульного и интеграционного тестирования согласно стандартам — IEEE Std. 829:1998 «IEEE standard for software test documentation» и IEEE Std. 1008:1987 «IEEE Standard for software unit testing» и *повторного использования* согласно процедур IEEE Std.1517:1999 «IEEE Standard for Information Technology — Software Lifecycle Processes — Reuse Process»;
- *обеспечения качества кода* с помощью стандарта ISO/IEC 9000.

2.2.4. Тестирование ПО

Тестирование ПО (*Software Testing*) заключается в проверке работы кода в *динамике* на *конечном* (ограниченном) наборе тестов, охватывающих *выбранное подмножество* сценариев использования ПО и тестовых данных, и сравнении полученных результатов с *ожидаемыми*.

Данная область знаний включает следующие разделы:

- основы тестирования (*Software Testing Fundamentals*);
- уровни тестирования (*Test Levels*);
- приемы тестирования (*Test Techniques*);
- метрики тестирования (*Test-related Measures*);
- процесс тестирования (*Test Process*).

Раздел **«Основы тестирования»** охватывает терминологию, основные проблемы тестирования и связь с другими областями знаний в SWEBOK. Базовые понятия в области тестирования определяются по стандарту IEEE Std. 610:1990 «Standard Glossary of Software Engineering Terminology». Рассматриваются ключевые проблемы, связанные с формированием критериев отбора (и адекватности) тестов, эффективности тестирования, идентификации дефектов, выбора «оракула» (который принимает решение о том, прошел или не прошел тест), тестируемости и др.

В разделе **«Уровни тестирования»** указаны уровни объектов ПО, для которых формируются тесты, и цели тестирования этих объектов. Выделены следующие уровни тестирования:

- *тестирование отдельных элементов* (*unit testing*), которое заключается в автономной проверке отдельных, изолированных и независимых частей ПО;
- *интеграционное тестирование*, ориентированное на проверку связей и способов взаимодействия (интерфейсов) отдельных компонентов;
- *тестирование системы*, предназначенное для проверки правильности функционирования системы в целом и выполнения сформулированных нефункциональных требований (по безопасности, надежности и др.), а также правильности внешних интерфейсов системы со средой окружения.

В разделе выделены следующие наиболее распространенные и обоснованные цели и, соответственно, *виды тестирования*: функциональное тестирование, регрессионное тестирование, тестирование эффективности, нагрузочное (стрессовое) тестирование, альфа- и бета-тестирование, тестирование конфигурации, тестирование производительности, надежности и др. К виду тестирования отнесены также методы проверки поведения системы на этапе испытаний ПО и его приемки.

В разделе **«Приемы (техники) тестирования»** представлена следующая классификация приемов тестирования:

- базирующиеся *на интуиции и опыте исполнителя* (тестирование по опыту (ad hoc), исследовательское тестирование);
- базирующиеся *на анализе спецификации* (по формальным спецификациям, на основе эквивалентного разделения программы на части, анализа граничных значений, таблиц решений, анализа переходов состояний и др.);
- базирующиеся *на анализе кода* (покрытие условий и решений в блок-схеме, анализ «жизненного цикла» переменных и др.);

• специальные приемы *обнаружения дефектов* (обнаружение дефектов на основе анализа рисков, «подсев» дефектов, мутационное тестирование);

• базирующиеся *на сценариях и условиях использования* (по операционному профилю, по специализированным методикам испытаний системы — на надежность, безопасность и др.);

• ориентированные *на определенный тип и архитектурную природу* ПО (приемы объектно ориентированного, компонентно ориентированного, веб-тестирования, тестирования на соответствие протоколам, тестирование систем реального времени и др.).

Измерения с помощью **метрик, связанных с тестированием**, используются для анализа качества ПО как инструмент оптимизации планирования и выполнения тестов, а также для управления процессом тестирования.

Для оценки программ в ходе тестирования рекомендуются метрики программ — размера, структурной сложности, частоты обращения к модулям; метрики (модели) дефектов и отказов — плотности дефектов, интенсивности отказов, роста надежности и др. Для оценки эффективности выполненных тестов рекомендуются метрики покрытия/глубины тестирования, оценки результатов мутаций и подсева дефектов, сравнительного анализа различных приемов тестирования.

В разделе **«Процесс тестирования»** указывается, что концепции, стратегии, техники и измерения тестирования должны быть объединены в *единый процесс тестирования* как деятельности по обеспечению качества на основе учета четырех элементов и связанных с ними факторов: людей, инструментов, регламентов и количественных оценок (измерений). Базовый стандарт по процессам ЖЦ ISO/IEC 12207 не выделяет всю деятельность по тестированию в качестве *единого* процесса, однако рассматривает соответствующие принципы работ по тестированию как неотъемлемую часть процессов ЖЦ.

Составной частью формализации процесса тестирования и основой для сертификации и оценивания организации по моделям зрелости и стандартам качества (например, CMMI) является *документация тестирования*. При ее составлении SWEBOK рекомендует руководствоваться стандартом IEEE Std. 829:1998 «Standard for Software Test Documentation» по документированию планов тестирования, спецификации процедур тестирования, отдельных тестов и др.

Существенным моментом в процессе тестирования является принятие решения о *завершении тестирования* (необходимом объеме тестирования). Тщательные измерения покрытия кода или охвата функциональности должны сочетаться с оценками *стоимости тестирования*, а также *рисков отказов* и стоимости устранения их последствий при эксплуатации системы.

Практические соображения относительно процесса тестирования, охватываемые данной областью знаний, касаются повторного использования тестов и шаблонов тестирования, а также отдельных работ по тестированию: планирования, генерации тестовых сценариев, создания тестовой среды, регламента выполнения тестов, анализа результатов тестирования и отчетности.

2.2.5. Сопровождение ПО

Сопровождение ПО (*Software Maintenance*) — совокупность действий по *эффективной* (с точки зрения затрат) *поддержке* функционирования ПО в период эксплуатации, включая внесение изменений при обнаружении ошибок в работе ПО, адаптацию ПО к новым условиям среды, повышение производительности и др. Хотя фаза сопровождения в ЖЦ ПО обычно начинается сразу после приемки/передачи продукта и действует в течение периода гарантии (или технической поддержки), предварительные работы начинаются до выпуска ПО и включают планирование деятельности по сопровождению системы и организацию перехода к ее полнофункциональному использованию. После выпуска ПО выполняется обучение обслуживающего персонала и модификация ПО, при условии сохранения целостности продукта.

Данная область знаний включает следующие разделы:

- основы сопровождения (*Software Maintenance Fundamentals*);
- ключевые вопросы сопровождения ПО (*Key Issue in Software Maintenance*);
- процесс сопровождения (*Maintenance Process*);
- приемы сопровождения (*Techniques for Maintenance*).

В разделе **«Основы сопровождения»** представлены базовые концепции и терминология, составляющие основу для понимания роли и содержания работ по сопровождению ПО, и сопоставлены положения стандартов IEEE Std. 1219:1998 «IEEE Standard for Software Maintenance», ISO/IEC 14764:1999 «Information technology — Software maintenance» и ISO/IEC 12207. Введены четыре *категории сопровождения*: корректирующее, адаптирующее, совершенствующее и профилактическое сопровождение.

Сопровождение, по сути, представляет собой *эволюционную разработку* системы. На работы по сопровождению приходится *бóльшая* часть ресурсов ЖЦ ПО, причем более 80% усилий связаны не столько с устранением отказов, сколько с другими работами. Это работы по улучшению дизайна, реализации расширений функциональных возможностей, созданию интерфейсов взаимодействия с другими (внешними) системами, адаптации к другим аппаратным платформам, обеспечению функционирования в среде телекоммуникационной инфраструктуры и т.п. вплоть до вывода ПО из эксплуатации.

Специалисты по сопровождению получают знания о продукте от разработчиков ПС. Организации или лица, на которые возлагается сопровождение системы, привлекаются к выполнению проекта задолго до его закрытия и непосредственно контактируют с разработчиками ПО, что позволяет снизить затраты на сопровождение системы после ввода в эксплуатацию.

В разделе **«Ключевые вопросы сопровождения ПО»** рассматриваются *технические*, *управленческие*, *стоимостные* и *измерительные* аспекты сопровождения.

«Техника» сопровождения охватывает вопросы быстрого изучения сопровождаемого ПО и локализации ошибок, использования тестирования, анализа последствий внесения изменений, сопровождаемости ПО (как показателя качества, обеспечиваемого при разработке ПО).

Сущность управленческих проблем заключается в поиске компромиссов в политике разработки и выпуска ПО (в срок и в рамках бюджета), с одной стороны, и политике дальнейшего сопровождения ПО, с другой; в выборе сопровождающей организации, подборе персонала сопровождения, организации сопровождения и др.

Вопросы оценки стоимости сопровождения решаются путем применения *параметрических моделей* и методов *экспертного оценивания* исходя из опыта по сопровождению ПО.

Для выполнения измерений при сопровождении SWEBOK рекомендует *общие метрики* (для всего ЖЦ ПО): размер, усилия, график работы, характеристики качества, а также *специализированные метрики* для оценки анализируемости, изменяемости, стабильности и тестируемости ПО — факторов, влияющих на уровень затрат по сопровождению ПО.

Процесс сопровождения должен определяться множеством выполняемых действий, входов и выходов и соответствовать требованиям IEEE Std. 1219:1998 или ISO/IEC 14764:1999.

Рассматриваются действия, характерные для процесса сопровождения:

- передача ПО от разработчика организации сопровождения;
- принятие/отклонение запросов на модификацию;
- извещение персонала сопровождения и отслеживание статуса запросов на модификацию и отчетов об ошибках;
- анализ влияния — анализ возможных последствий изменений, вносимых в существующую систему;
- поддержка ПО — консультационные услуги по запросам пользователей;
- ведение контрактов и обязательств об уровне услуг, предоставляемых сопровождающей организацией.

Кроме того, дополнительно определяются действия по планированию сопровождения, управлению конфигурацией, верификации и валидации ПО, оценке качества ПО, а также обучению персонала сопровождения.

Практические **приемы сопровождения** включают использование браузеров кода и других инструментальных средств для изучения ПО, а также таких методов эволюции ПО, как реинженерия, реверсная инженерия и рефакторинг.

Реинженерия — тщательное изучение функций и последующая радикальная переделка ПО путем его реорганизации, реструктуризации, перепрограммирования или настройки на другую платформу или среду. Цель реинженерии — не столько обеспечение удобства его сопровождения, сколько замена устаревшего ПО.

Реверсная (*обратная*) *инженерия* — восстановление спецификации (графов вызовов, потоков данных и др.) по исходному коду (например, после внесения в него множества изменений) с целью идентификации программных компонентов и связей между ними, формирования представления о ПО и обновления документации (в частности, для обеспечения последующей реинженерии).

Рефакторинг — трансформация ПО с сохранением архитектурных и технологических решений и без изменения его поведения, предпринимаемая с целью улучшения структурных характеристик и качественных

показателей (включая удобство сопровождения). Может рассматриваться как одна из форм реверсной инженерии.

2.2.6. Управление конфигурацией ПО

Управление конфигурацией или конфигурационный менеджмент (*Configuration Management*) — деятельность, связанная с контролем эволюции и целостности ПП в ходе выполнения проекта ПО системы путем идентификации всех артефактов, отслеживания их изменений и ведения отчетности о состоянии конфигурации ПО системы.

Работы по управлению конфигурацией ПО (далее — SCM, т.е. Software Configuration Management) включают тщательное планирование процесса SCM, определение характеристик элементов конфигурации, учет указанных характеристик и контроль за их изменениями, составление отчета о внесенных изменениях в конфигурацию и статусе их реализации; аудит конфигурации и проверку соответствия внесенных изменений заданным требованиям.

Данная область знаний включает следующие разделы:

- управление процессом (*Management of the SCM Process*);
- идентификация конфигурации ПО (*Software Configuration Identification*);
- контроль конфигурации ПО (*Software Configuration Control*);
- учет состояния конфигурации (*Software Configuration Status Accounting*);
- аудит конфигурации ПО (*Software Configuration Auditing*);
- управление выпуском и поставка ПО (*Software Release Management and Delivery*).

Управление процессом конфигурационного менеджмента осуществляется с учетом организационного контекста, особенностей и ограничений аппаратно-программной архитектуры системы и ПО (включая размер и сложность) и принятой модели процесса, устанавливаемой применяемыми стандартами и руководствами по SCM (например, ISO/IEC 12207, ISO/IEC 15504, CMMI).

Планирование процесса SCM охватывает распределение обязанностей и полномочий и способы отчетности; определение ресурсов и составление графика работ; подбор (разработку) инструментов (для ведения SCM-библиотек, запросов на изменения, управления кодом, сборки ПП, отчетности и др.), а также построение плана управления конфигурацией (например, в соответствии с IEEE Std.828:1998 «Standard for Software Configuration Management Plans»).

Для *контроля выполнения процесса SCM* используются метрики, состав которых определяется потребностями проекта и набором характеристик, значения которых могут быть получены с помощью применяемых инструментов управления конфигурацией (количество внесенных изменений в одном элементе, количество отвергнутых изменений, количество версий и др.).

Работы по **идентификации конфигурации** включают: определение контролируемых элементов (исходного и исполняемого кода, библиотек компо-

нентов, проектной документации, графиков работ и др.); определение схемы идентификации элементов и их версий; документирование функциональных и физических характеристик элементов; описание применяемых инструментов и приемов управления элементами конфигурации; определение базовой версии ПО; описание взаимосвязей между элементами, версиями и др.

Контроль конфигурации подразумевает управление изменениями в течение ЖЦ ПО. Включает анализ запросов на изменения и определение того, какие именно изменения должны быть сделаны и какие полномочия необходимы для их утверждения или отклонения, в чем суть поддержки реализации изменений и каков порядок формального утверждения «отхода» от проектных требований. Принятые (утвержденные) запросы на изменения реализуются с помощью определенных процедур и в определенном порядке с учетом взаимосвязи элементов (одновременно или последовательно). «Закрытие» изменений предполагает аудит конфигурации.

Учет состояния конфигурации подразумевает ведение всей информации об элементах конфигурации и проделанных изменениях, автоматизированный контроль и анализ внесенных изменений, а также генерацию отчетности. Отчетность может использоваться как основа для проведения различных количественных оценок в интересах руководства или процесса разработки либо для повышения эффективности процесса управления конфигурацией ПО.

Аудит конфигурации — это деятельность по оценке соответствия продуктов и процессов принятым стандартам, инструкциям, планам и процедурам. В ходе аудита определяется степень, в которой элемент конфигурации удовлетворяет заданным функциональным и физическим характеристикам. Различают неформальный аудит (в контрольных точках проекта) и формальный аудит двух типов: функциональный и физический аудит конфигурации, который завершается фиксацией *базовой версии* продукта. Цель функционального аудита — убедиться в том, что контролируемый программный элемент полностью соответствует заданным спецификациям, а физического — в том, что проект (дизайн) и документация точно согласуются с самим ПП. Может также проводиться выборочный внутренний аудит элементов базовых версий.

Управление выпуском версий и поставкой ПО включает отслеживание версий элементов конфигурации; обеспечение оперативного доступа к информации относительно элементов конфигурации и системы, к которым эти элементы относятся; сборку (объединение) компонентов; создание новых версий системы на основе существующей путем внесения изменений в конфигурацию; согласование версии продукта с требованиями. Управление выпуском охватывает идентификацию, упаковку и передачу элементов продукта и документации заказчику.

2.2.7. Управление инженерией ПО

Управление (менеджмент) инженерией ПО (*Software Engineering Management*) — приложение общих методов управления к инженерной деятельности по созданию ПО, осуществляемой в рамках *проектов* ПО и регламентированной *процессами* ЖЦ ПО.

Как любое управление, менеджмент ПО подразумевает планирование, координацию действий, мониторинг и измерение, контроль и принятие управленческих решений, а также отчетность по управлению проектом. Отличительные особенности управления SE связаны с учетом таких факторов, как: нестабильность требований потребителей, итеративность выполняемых процессов, высокий уровень новизны и сложности разработок (не позволяющий в полной мере использовать наработки и опыт), быстрота обновления технологий и т.п.

Распределение и координация взаимодействия людских, финансовых и технических ресурсов при реализации задач проекта возлагаются на менеджера проекта. В решении общих вопросов управления он должен руководствоваться таким источником знаний, как Руководство к PMBOK, а в решении специальных вопросов управления программными проектами — информацией из данной области знаний, а также положениями ISO/IEC 12207, где управление проектом рассматривается как необходимый *организационный процесс* ЖЦ, выполняемый в соответствии с политиками, целями и принятыми стандартами организации — разработчика ПО.

Данная область знаний включает следующие разделы:

- инициирование и определение рамок проекта (*Initiation and Scope Definition*);
- планирование проекта (*Software Project Planning*);
- выполнение проекта (*Software Project Enactment*);
- обзор и оценка (*Review and Evaluation*);
- закрытие (*Closure*);
- измерения в инженерии ПО (*SE Measurement*).

Раздел **«Инициирование и определение рамок проекта»** касается вопросов, связанных с принятием решений о начале программного проекта. Это вопросы эффективного выявления, анализа, специфицирования и проверки требований к ПО с использованием различных методов извлечения требований; определения целей, содержания и ограничений проекта; оценки осуществимости проекта с различных точек зрения (технической, технологической, финансовой, политической и др.); определения процедур пересмотра требований и анализа зависимостей и рисков для оценки влияния изменения требований.

Планирование проекта представляет собой *итеративный* процесс, базирующийся на анализе требований и содержания проекта, а также условий его осуществимости.

Первый шаг планирования предполагает выбор *модели разработки* (спиральной, эволюционной и т.п.) и *применяемых процессов* ЖЦ, а также методов и инструментов, используемых при управлении проектом, в частности для *декомпозиции проекта* в виде *задач* с ассоциированными входами, выходами и условиями завершения. Далее по каждой задаче специфицируются результаты выполнения (артефакты) и производится оценка трудоемкости и стоимости работ с помощью аналого-сопоставительных методов (по накопленным историческим данным) или экспертным путем. Идентифицируются связи и зависимости между задачами и потенциально

критические аспекты проекта (что может быть сделано, например, с помощью метода *анализа критического пути*). Для задач определяются ожидаемые сроки выполнения и расписание (график) работ (например, в форме *PERT-диаграмм*). Требования к ресурсам трансформируются в прогнозируемые затраты на проект. Все эти действия проводятся итеративно до тех пор, пока не будет достигнут консенсус между соответствующими заинтересованными лицами — в первую очередь менеджером и формируемой группой проекта, после чего осуществляется распределение обязанностей/ответственности людей, а также технических и финансовых ресурсов (с отражением, например, в *диаграмме Ганта*).

К данному разделу знаний SWEBOK отнесены также вопросы *управления рисками* и *управления качеством* в проекте, поскольку планирование проекта должно осуществляться с учетом дополнительных затрат ресурсов, в том числе и на эту деятельность. Управление рисками заключается в регулярной идентификации и анализе рисков, оценке их критичности и упорядочению по приоритетам и разработке мероприятий по устранению или смягчению рисков, начиная с наиболее критичных для проекта. Управление качеством заключается в планировании и осуществлении мероприятий, нацеленных на достижение характеристик качества ПП, указанных в спецификации требований.

Планироваться должен также процесс *управления* разработанным *планом проекта*, что особенно важно в условиях перманентных изменений требований и окружения проекта. Актуальность плана должна систематически контролироваться и при необходимости восстанавливаться.

Основой успешного **выполнения проекта** являются следование плану проекта, соблюдение процессов, представленных в плане, и управленческая деятельность по систематическому оцениванию и измерениям, мониторингу, контролю и отчетности по проекту. Существенным моментом для выполнения проекта выступает у*правление контрактами с поставщиками*, включающее подготовку и выполнение соглашений с поставщиками, мониторинг деятельности поставщиков, приемку поставляемых продуктов и интеграцию этих продуктов в рамках проектных работ.

Процесс выполнения проекта сопровождается *регулярными измерениями* (усилий, расходованных средств, выполненных объемов работ и др.), а результаты измерений *оцениваются* по отношению к плану проекта в ходе *мониторинга проекта*. Соблюдение плана, а также выход проекта за ограничения проверяются регулярно, через предопределенные интервалы времени, и сопровождаются оценками (и переоценками) известных рисков и выявлением новых и проверкой удовлетворения требований к качеству. Отчетность по результатам мониторинга обеспечивает базис для принятия решений и *регулирования* проекта, т.е. выполнения корректирующих действий, что, в свою очередь, может привести к изменению плана, отдельных работ, документов и других активов проекта. Выполнение проекта сопровождается также *регламентными отчетами* по состоянию проекта, направляемыми всем заинтересованным лицам.

Формальный **обзор и оценка проекта** выполняются в контрольных точках при достижении определенных вех проекта для оценивания *общего*

прогресса в достижении установленных целей и удовлетворении требований заинтересованных лиц, а также для оценивания *эффективности* процессов, работы персонала и инструментов и методов, которые применялись в заданном промежутке времени.

После выполнения всех планов, окончания процессов и подтверждения достижения *критериев завершения* проекта наступает **закрытие проекта**. На этой стадии оценивается успешность проекта, принимается и утверждается решение о его закрытии и передаче продукта заказчику и выполняются действия по архивированию проектных данных и обновлению измерений, анализу результатов, полученных в процессе работы над проектом, и улучшению процессов.

Измерения в инженерии ПО направлены на получение количественных оценок отдельных характеристик процессов, ресурсов и артефактов проекта с целью их использования при управлении инженерией ПО. Ключевые термины, а также методы измерений изложены в стандарте ISO/IEC 15939:2002 «Software Engineering — Software Measurement Process», который описывает информационную модель измерений и процесс, определяющий действия/работы и задачи, необходимые для реализации процесса измерений. В полном соответствии с этим стандартом в данном разделе рассматриваются следующие *вопросы*: формулирование требований в отношении измерений (уровень, цели и объекты измерений, *потребители результатов* измерений); выбор *метрик*, спектр и объемов *собираемых данных*; определение *процедур анализа* и ведения отчетности по измерениям; *планирование процесса* измерений и выделения ресурсов; организация измерений в подразделениях и *выполнение процесса* измерений (интеграция действий по измерениям в выполняемые процессы); анализ результатов измерений, *составление отчета* о полученном *информационном продукте* и его передача потребителю измерений; оценка хода измерений и совершенствование процесса измерений.

2.2.8. Процесс SE

Процесс SE (*Software Engineering Process*, SEP) как область знаний представлен в Руководстве к SWEBOK на двух уровнях описания.

Прежде всего (на нижнем уровне), SEP — это множество взаимосвязанных видов деятельности по *созданию программных систем* в рамках проектов, выполняемых на основе интегрированных процессов ЖЦ, адаптированных к конкретным проектам. В этом привычном для разработчиков понимании процесс SE рассматривается практически повсюду в Руководстве к SWEBOK.

Однако данная область знаний охватывает и другой аспект процесса SE, определяя его (на верхнем, метауровне) как множество взаимосвязанных видов деятельности по определению, реализации, оценке, измерению, управлению и совершенствованию собственно *процессов ЖЦ*, формирующих *базовый процесс SE* в организации (безотносительно к проектам). Именно этот аспект SEP имеется в виду при рассмотрении данной области знаний SWEBOK.

Данная область знаний включает следующие разделы:

- реализация и изменение процесса (*Process Implementation and Change*);
- определение процесса (*Process Definition*);
- оценивание процесса (*Process Assessment*);
- измерение процесса и продукта (*Process and Product Measurement*).

Раздел **«Реализация и изменение процесса»** описывает инфраструктуру, действия, модели и практические соображения по реализации SEP и его изменению. Изложение вопросов раздела базируется на стандартах IEEE/IEC 12207 и ISO/IEC 15504 «Information technology — Software process assessment».

Инфраструктура SEP может организовываться по типу *группы процесса SE* SEPG (*Software Engineering Process Group*) или по типу *экспериментальной фабрики* EF (*Experience Factory*). SEPG образуют специалисты организации, в обязанности которых (возможно наряду с работой по проектам) входят внедрение и совершенствование процессов ЖЦ. В отличие от SEPG экспериментальная фабрика никак не связана с выполнением проектов в организации, представляет собой отдельное подразделение (или другую организацию) и всецело ориентирована на институциализацию накопленного в масштабе организации коллективного опыта и извлеченных уроков, путем разработки, обновления и внедрения в проекты «пакетов опыта» (*experience packages*). Пакеты формируются из типовых активов процесса и могут включать, например, руководства, модели и методики, курсы обучения и др. Подразделения, работающие по проектам, в свою очередь, предлагают на рассмотрение EF свои наработки по внедренным и выполняемым процессам, что может послужить толчком к их усовершенствованию.

Помимо создания инфраструктуры управление процессами в области ПО включает еще три действия, выполняемые в рамках итеративного цикла: 1) *планирование* внедрения нового процесса или совершенствования существующего, а также разработку плана реализации или изменения процесса; 2) собственно *осуществление* реализации или изменения процесса; 3) *оценку* реализации процесса и отдачи от его внедрения.

В SWEBOK упомянуты две распространенные *модели реализации и изменения* SEP — модель QIP (*Quality Improvement Paradigm*) и модель IDEAL (*Initiating-Diagnosing-Establishing-Acting-Learning*), а также два подхода к *оценке реализации и изменения процесса*, первый из которых состоит в оценке самого процесса, второй — в оценке результатов (наработок) процесса.

Определение процесса может быть представлено процедурой, рекомендацией или стандартом и основываться на таких важных факторах, как природа работ (например, разработка или сопровождение), прикладная область, а также модель ЖЦ и уровень зрелости самой организации.

Модель ЖЦ задает высокоуровневое определение фаз (стадий) разработки ПО, включая основные работы и их взаимосвязи. Определение процесса SE обычно детальнее, чем модель, однако не описывает порядка выполнения работ во времени. Это определение может быть «выстроено» из определений выбранных процессов ЖЦ в соответствии с любой моделью.

Совокупность модели ЖЦ, процессов ЖЦ и приемов их выполнения определяет *методологию* поддержки ЖЦ ПО. Основным источником знаний по всем процессам ЖЦ ПО являются стандарт ISO/IEC 12207 (а также руководство к нему — ISO/IEC 15271:1998) и IEEE Std. 1074:1997 «IEEE Standard for Developing Software Life Cycle Processes» (стандарт по разработке процессов ЖЦ). Кроме того, существуют стандарты по отдельным процессам ЖЦ, например IEEE Std. 1219:1998 «IEEE Standard for Software Maintenance» или ISO/IEC 14764:1999 «Information technology — Software Maintenance», IEEE Std. 1540:2001 «IEEE Standard for Software Life Cycle Processes — Risk. Management», IEEE Std. 1517:1999 «Software Reuse Processes», ISO/IEC 15939:2002 «Software Engineering — Software Measurement Process» и др.

Процессы SE могут также определяться с учетом потребностей системы менеджмента качества, внедренной в организации и сертифицированной по стандарту ISO 9001:2001, который формулирует требования к процессам управления качеством.

Процессы могут определяться на различных уровнях абстракции и с использованием различных *нотаций*: диаграмм потоков данных; перечислений целей и результатов процессов (как, например, в ISO/IEC 15504); набора процессов и их декомпозиции в виде действий и задач, определенного на естественном языке (как, например, в ISO/IEC 12207); диаграмм переходов состояний, SADT, IDEF0 и др.

Для применения в конкретных проектах предопределенные (стандартизованные) процессы должны адаптироваться в соответствии с организационным контекстом, размером проекта, существующими регулирующими требованиями, отраслевой и ведомственной практикой и корпоративной культурой. Рекомендации по адаптации содержатся в ISO/IEC 12207 и ISO/IEC 15504.

Используемые в проектах методологии могут поддерживать автоматизированное определение процессов, обеспечивая их описание с использованием тех или иных нотаций, или предоставлять руководства по определению процессов.

Оценивание процесса проводится с использованием соответствующих моделей и методов.

Типовая модель оценивания (SPICE) определена в стандарте ISO/IEC 15504. Там же содержатся требования по соответствию других моделей (например, SW CMM, CMMI, Bootstrap и др.) этой типовой модели. На практике используются две архитектуры моделей оценивания: непрерывное оценивание и поэтапное оценивание. Выбор архитектуры обусловлен целями и возможностями организации.

Методы оценивания определяют порядок и способы получения количественных параметров оцениваемого процесса, которые характеризуют мощность этого процесса и направлены либо на совершенствование процессов, либо на оценку готовности (зрелости) организации для выполнения определенного контракта.

Измерение процесса и продукта этих процессов могут проводиться для идентификации сильных и слабых сторон процессов (в определенном

контексте их применения) и инициирования работ по их изменению, а также для оценивания процессов после того, как они реализованы и (или) изменены.

Кроме того, измерения составляют основу для оценивания ситуации и принятия решений при управлении проектом, для оценивания качества, размера, структуры и других характеристик ПП, а также эффективности применяемых технологий.

Ключевые понятия, термины и методы измерений в ПИ определены в стандарте ISO 15939:2002 «Software Engineering — Software Measurement Process» наряду с определением самого базового процесса измерения характеристик процессов и продуктов.

Подходы к измерению процессов классифицируются как *аналитические* и *эталонные* (*benchmarking*) и используются совместно, поскольку основываются на различных типах взаимодополняющей информации.

Аналитические подходы включают экспериментальное исследование процесса в специальной среде (наблюдение за процессом, моделирование поведения процесса и отклонений), обзор определения процесса (например, его сопоставление со стандартом ISO/IEC 12207), ортогональную классификацию дефектов (изъянов в процессе), анализ коренных причин (Root Case Analysis), статистический контроль процесса.

Подходы «от эталонов» базируются на идентификации «совершенной» организации процесса, приемов и инструментов и последующем использовании опыта (например, наработок таких организаций, как SEI и PMI).

2.2.9. Инструменты и методы ПО

Программные инструменты предназначены для автоматизированной поддержки выполнения процессов ЖЦ ПО (с охватом отдельных процессов и методов или всего ЖЦ). Методы, как и инструменты, варьируются по содержанию (в зависимости от охватываемой области приложения) и применяются на отдельных фазах ЖЦ или в течение всего ЖЦ.

Область знаний включает два раздела:

- инструменты SE (*Software Engineering Tools*);
- методы SE (*Software Engineering Methods*).

Раздел **«Инструменты SE»** описывает классы инструментов, используемых практически во всех областях знаний SWEBOK. Дополнительно рассматриваются вопросы интеграции инструментов и создания платформ разработки, применения метаинструментов и подходов к оцениванию инструментов. Краткая характеристика инструментов в предложенной SWEBOK классификации представлена в табл. 2.1.

Таблица 2.1

Классификация инструментов SWEBOK

Класс	Типы	Описание назначения и примечания
Инструменты работы с требованиями	Средства моделирования	Управление программными требованиями — извлечение, анализ, специфицирование и проверка
	Средства трассировки	Отслеживание проекций требований на всех этапах ЖЦ

Продолжение табл. 2.1

Класс	Типы	Описание назначения и примечания
Инструменты проектирования	В SWEBOK не классифицированы	Можно классифицировать по базовым нотациям (SADT, UML, DSL...), по целевым задачам (бизнес-проектирование, проектирование БД...)
Инструменты конструирования	Редакторы программ	Создание и модификация исходного кода и, возможно, документации (в IDE — интегрированных средах)
	Компиляторы и генераторы кода	Трансляторы с командной строкой. Однако есть тенденция интеграции с редакторами в IDE
	Интерпретаторы	Исполнение программ посредством эмуляции
	Отладчики	
Инструменты тестирования	Генераторы тестов	Разработка сценариев тестирования
	Средства выполнения тестов	Исполнение тестовых сценариев в контролируемом окружении
	Инструменты оценки тестов	Оценка результатов выполнения тестов
	Средства управления тестами	Поддержка всех аспектов процесса тестирования
	Инструменты анализа производительности (как инструменты *целевого анализа*)	Оценка поведения программы с точки зрения производительности. Другие инструменты *целевого анализа* — для функционального тестирования, тестирования безопасности, тестирования пользовательского интерфейса, нагрузочного тестирования и др.
Инструменты сопровождения	Средства визуализации	Инструменты, облегчающие понимание
	Инструменты реинженерии	Восстановление спецификации и архитектуры ПО по коду
Инструменты управления конфигурацией	Инструменты отслеживания	Отслеживание дефектов, расширений и проблем
	Инструменты управления версиями	
	Инструменты сборки и выпуска	Управление задачами сборки и выпуска продуктов, включая инсталляцию
Управление инженерией	Инструменты планирования и отслеживания работ	Календарное планирование работ по проекту, оценка трудоемкости и прогнозирование стоимости
	Инструменты управления рисками	Идентификация рисков, оценка последствий, мониторинг рисков
	Средства количественной оценки	Ведение измерений, поддержка программы измерений

Окончание табл. 2.1

Класс	Типы	Описание назначения и примечания
Инструменты поддержки процессов	Инструменты моделирования процессов	Более широкий класс инструментов моделирования, используемых, в частности, для описания и разработки модели процессов
	Средства управления процессами	Инструменты для управления проектами и управления конфигурациями

Методы SE классифицируются как:

- *эвристические* (неформальные): структурные, объектно ориентированные, ориентированные на данные;
- *формальные* (математически обоснованные): языки и нотации спецификирования, методы уточнения (трансформации), методы подтверждения/доказательства;
- *методы прототипирования*: стили прототипирования, цели прототипирования, приемы оценки результатов прототипирования.

2.2.10. Качество ПО

Качество ПО (*Software Quality*) — совокупность свойств ПП, обеспечивающих его способность удовлетворять *установленным* или *предполагаемым* требованиям заказчика (пользователя). В это определение вкладывается разный смысл в зависимости от контекста применения понятия и аспектов рассмотрения. Можно различать, например, рыночное качество ПО (*market-driven quality*), потребительское качество (*customer-driven quality*); качество ПО с позиций его сопровождения, обеспечения безопасности и т.п.; качество внутреннее, внешнее и эксплуатационное.

Залог повышения качества при процессном подходе — применение процессов проверки (верификации, тестирования, аудита и др.), нацеленных на своевременное обнаружение и устранение дефектов в ПО и изъянов в самих процессах ЖЦ. Даная область знаний SWEBOK охватывает вопросы *статической* проверки продуктов и процессов. Вопросы тестирования как способа *динамической* проверки работы ПО в данной области знаний SWEBOK не рассматриваются.

Область знаний включает следующие разделы:

- основы качества ПО (*Software Quality Fundamentals*);
- процессы управления качеством ПО (*Software Quality Management Processes*);
- практические соображения (*Practical Considerations*).

В разделе **«Основы качества ПО»** подчеркивается, что все вопросы качества должны рассматриваться в контексте *требований к программному продукту*, которые не только определяют необходимые характеристики качества ПО, но и обусловливают выбор методов их количественной оценки и критериев приемки ПО.

Инженеры ПО должны воспринимать вопросы качества как часть своей *профессиональной культуры*. Требования к характеристикам качества всегда являются результатом определенного *компромисса* (как по номенклатуре

и значениям характеристик, так и по *стоимости* обеспечения и *ценности* этих характеристик для заказчика). Стоимость качества дифференцируется как стоимость предупреждения дефектов, стоимость оценки качества, цены внутренних сбоев и внешних потерь. Большинство *альтернативных* компромиссных решений по качеству должно предлагаться и приниматься в процессе работы с требованиями, однако вопросы, касающиеся качества, могут подниматься на протяжении всего ЖЦ ПО и в рамках любых процессов ЖЦ.

Способы построения таксономий и *моделей качества* различны — иерархии, графические образы и т.п. *Базовый стандарт по качеству* ISO/IEC 9126-1:2001 «Software Engineering — Product Quality, Part 1: Quality Model» предлагает рассматривать качество на трех уровнях его обеспечения в артефактах проекта (внутреннее качество, внешнее качество и качество ПО в эксплуатации) и рекомендует иерархическую модель качества для каждого из этих уровней. Оценивается качество в соответствии со стандартом ISO/IEC 14598:1999 «Information Technology — Software Product Evaluation».

Непосредственное отношение к качеству создаваемого продукта имеет *управление качеством* (SQM, от Software Quality Management) и *качество процессов* ПИ. В области управления (менеджмента) качества важнейшими являются стандарты TickIT и ISO 90003:2004 «Software and Systems Engineering — Guidelines for the Application of ISO 9001:2000 to Computer Software», а в области качества (мощности, зрелости) процессов ЖЦ — стандарты CMMI и ISO/IEC 15504:2004 «Information Technology — Software Process Assessment» (модель SPICE, от Software Process Improvement and Capability Etermination). Эти стандарты взаимно дополняют друг друга, причем сертификация процессов по ISO 9001 помогает в достижении старших уровней зрелости по CMMI.

Понимание термина «*продукт*» в данной области знаний расширено включением *любых артефактов*, создаваемых в результате выполнения всех процессов ЖЦ, используемых для создания конечного ПП (например, спецификации системных требований и требований для программных компонентов, моделей, кода, тестовой документации и др.). Признаком высокой зрелости организации является наличие и использование собственных (исторических) данных относительно требований к качеству отдельных артефактов и результатов оценки их фактического достижения. Это дает возможность оценивать соответствие заданным характеристикам качества не только конечного продукта, но и промежуточных результатов/продуктов ЖЦ в рамках всех процессов SE.

Качество ПО может *повышаться* в ходе итеративного процесса его постоянного улучшения, требующего контроля, координации и обратной связи при управлении одновременно выполняемыми процессами: разработки, обнаружения, устранения и предотвращения дефектов, а также совершенствования процессов ЖЦ.

Раздел **«Процессы управления качеством ПО»** рассматривает во взаимосвязи процессы *обеспечения гарантии качества* (SQA, Software Quality Assurance), *верификации* и *валидации*, *обзора* (*совместного просмотра*)

и *аудита*, а также собственно *процесс управления качеством* (SQM), обеспечивающий функции контроля и оценки по всем аспектам процессов, продуктов и ресурсов, включая требования к процессам, измерения процессов и их результатов и установление обратной связи.

SQA, как отмечается в руководстве к SWEBOK, концентрируется *на процессах*. Роль SQA состоит в том, чтобы обеспечить планирование процессов, дальнейшее исполнение процессов на основе заданного плана и проведение необходимых измерений процессов. Верификация и валидация (*V&V*, *Verification and Validation*) применяются к *артефактам* проекта в его контрольных точках с целью проверки и подтверждения их правильности. Действия по *V&V* регламентируются базовыми стандартами IEEE 1012:1998 «Software Verification and Validation» и IEEE 1059:1993 «IEEE Guide for Software Verification and Validation Plans». Деятельность по обзорам и аудитам продуктов включает пять типов действий: управленческие обзоры, технические обзоры, инспекции, сквозной контроль и аудиты, и регламентируется стандартом IEEE 1028:1997 «IEEE Standard for Software Reviews».

Практические соображения в данной области касаются, прежде всего, определения *факторов*, влияющих на выбор стратегии SQM. Один из них — *критичность области применения* ПО. Комплексной характеристикой качества такого ПО является *гарантоспособность* (*dependability*) — совокупность свойств ПО, обеспечивающих его отказоустойчивость (*fault tolerance*), безопасность эксплуатации (*safety*), защищенность информации (*security*), а также удобство и простоту использования (*usability*). Другой важный фактор — уровень необходимой *целостности* ПО, связываемый с оценкой возможных последствий отказа ПО и вероятности его возникновения. Обеспечение целостности ПО требует привлечения приемов анализа опасностей окружению и угроз целостности информации.

В этом разделе высказываются также практические соображения относительно описания *характеристик дефектов* (рекомендуемый стандарт IEEE 1044:1993 «IEEE Standard for the Classification of Software Anomalies»), сбора и регистрации данных о дефектах, отчетов об устранении дефектов и т.п.

Дается классификация приемов SQM, включая статические и динамические, коллективные и индивидуальные, а также аналитические (формальные и неформальные) приемы. Подчеркивается, что тестирование может и должно применяться для решения таких задач SQM:

- оценка и тестирование инструментов, используемых в проекте (согласно ISO/IEC 14102:1995 «Information Technology — Guideline for the Evaluation and Selection of CASE Tools»);
- тестирование соответствия компонентов ПО и COTS-продуктов (от *commercial of-the-shelf*, готовый к использованию продукт, продукт «с полки») потребностям их использования в создаваемом продукте (согласно ISO/IEC 12119: 1994 «Information Technology — Software Packages — Quality Requirements and Testing»).

Эта область знаний охватывает также такие аспекты, как количественная оценка качества ПО с использованием метрик и оценка стоимость процесса SQM.

2.3. Модели ЖЦ ПС

ЖЦ ПС определяется как «...весь период существования системы от начала разработки до завершения ее использования»[1].

ЖЦ включает *упорядоченные процессы*, основные из которых:

- определение потребностей;
- анализ требований и формирование концепции;
- разработка;
- производство;
- внедрение/продажа;
- эксплуатация;
- сопровождение и поддержка;
- изъятие из эксплуатации.

Внутри каждого из этих процессов происходит дальнейшая детализация выполняемых действий по более мелким операциям (на более низком уровне).

Модели ЖЦ описывают *взаимосвязи* стадий ЖЦ с *процессом разработки ПС*, основные из которых:

- анализ требований;
- проектирование (предварительное и детальное);
- реализация;
- тестирование.

Наиболее известными *типами* моделей ЖЦ в настоящее время являются: последовательные и итерационные. На практике эти модели могут комбинироваться, образуя смешанные модели ЖЦ.

Каждая из моделей имеет свои достоинства и недостатки, которые должны исследоваться при выборе конкретной модели для проекта.

Модели ЖЦ могут использоваться:

- для организации, планирования, распределения ресурсов (трудозатрат и времени) и управления проектом разработки;
- организации взаимодействия с заказчиками и определения состава документов (рабочих продуктов), разрабатываемых на каждой стадии;
- анализа и (или) оценивания распределения ресурсов и затрат на протяжении ЖЦ;
- наглядного описания или в качестве основы для проведения финансовых расчетов с заказчиками;
- проведения эмпирических исследований с целью определения влияния моделей на эффективность разработки и общее качество ПП.

В стандарте ISO/IEC Life Sycle (ЖЦ) 12207 определена структура процессов, но не указаны способы их взаимодействия в рамках разных моделей. Рекомендации по возможному отображению процессов ЖЦ на основные модели разработки приведены в Руководстве по применению стандарта.

---

[1] *Скопин И. Н.* Модели жизненного цикла программного обеспечения. URL: http://www.computer-museum.ru/books/n_collection/models.htm.

2.3.1. Каскадные модели

Каскадная (*Waterflall*) (или стандартная) модель — наиболее известная модель разработки, по умолчанию предполагаемая стандартами (в частности, ГОСТ 34). Эта модель характеризуется набором стадий, выполняемых последовательно (рис. 2.3). Каждая стадия должна быть завершена до перехода к следующей, а создаваемые на ней рабочие продукты после их верификации и валидации должны быть «заморожены» и переданы на следующую стадию в качестве эталона. Пользователь видит работающий ПП в самом конце разработки. Наиболее жесткое ограничение этой модели — необходимость «замораживания» требований. При этом, чтобы минимизировать риск увеличения стоимости, допускаются только небольшие изменения.

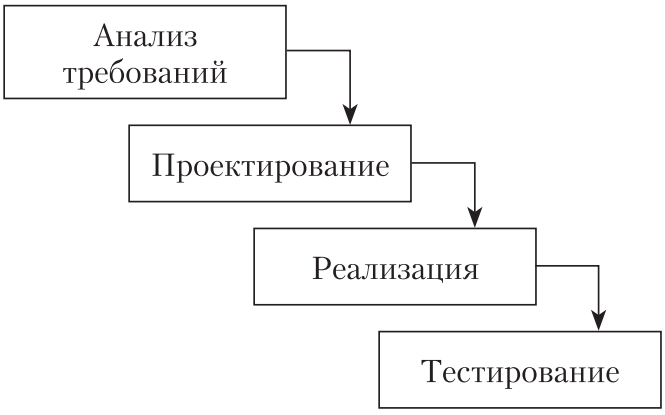


Рис. 2.3. **Каскадная модель**

С точки зрения качества ПС в этой модели стоимость исправления дефектов на стадии тестирования наибольшая (по сравнению с другими моделями), поскольку тестирование выполняется в самом конце разработки. Из-за недостатка времени на переделки и тестирование существует значительный риск выпуска ПС с серьезными дефектами.

Каскадная модель с обратной связью

Эта модель расширяет стандартную модель включением в нее *циклов обратной связи* для возврата на предыдущую стадию при изменении требований, проекта (конструкции) и по результатам инспекций или действий по *V*&*V* (рис. 2.4).

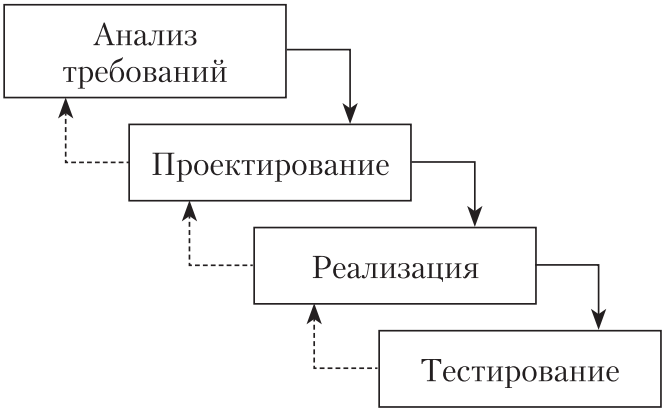


Рис. 2.4. **Каскадная модель с обратной связью**

Процессы *V&V*, выполняемые после завершения каждой стадии разработки, играют в этой модели важнейшую роль.

V-образная модель

V-образная (*V-shape*) модель расширяет каскадную модель включением в нее действий по *раннему планированию тестирования*.

В ней проводятся проверка и оценка тестопригодности требований на ранних стадиях разработки (посредством анализа, выполняемого при планировании тестирования), а также улучшаются планирование и распределение затрат на тестирование

Каскадная модель с прототипированием

Модель является модификацией *V*-образной модели с включением в нее прототипирования для моделирования требований и проекта (рис. 2.5). Прототипы служат для демонстрации и после разработки проекта выбрасываются, а реализация проекта может выполняться в другой среде.

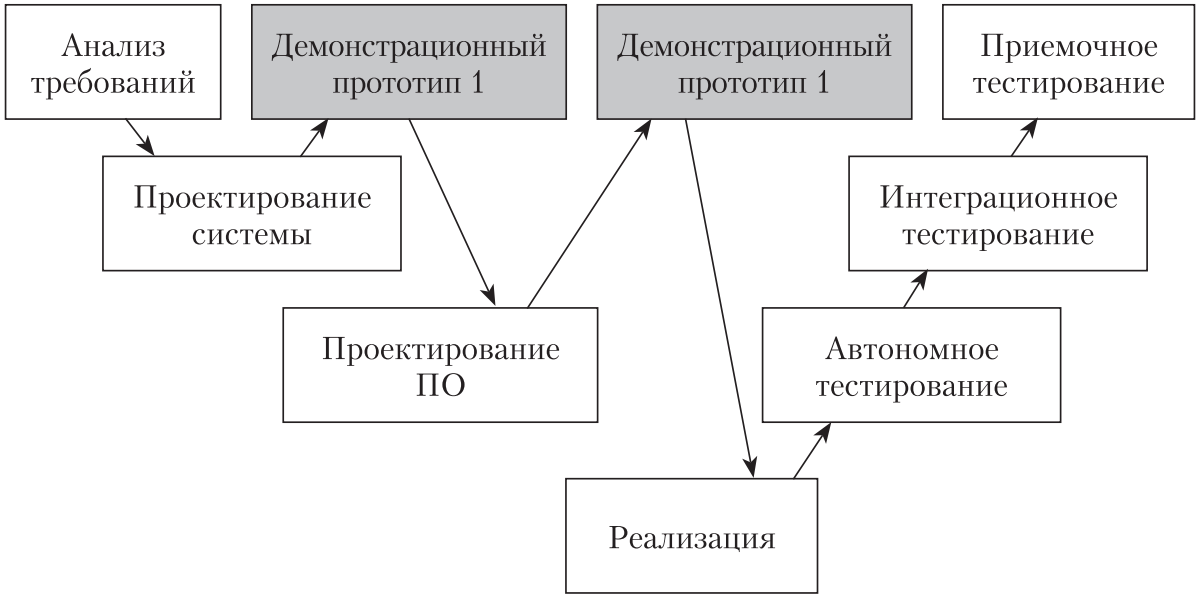


Рис. 2.5. **Последовательная модель с прототипированием**

Анализ и моделирование проектных решений проводятся методом прототипирования. При этом устраняются проблемы, связанные с неполнотой и нечеткостью задания требований.

При применении моделей возникают риски, когда требования не полностью понятны либо часто изменяются. В этом случае, если система большая, рекомендуется провести автономное тестирование отдельных элементов, потом всей системы. При нахождении ошибок проводится замена отдельных моделей либо частей системы.

2.3.2. Итерационные модели

Итерационные модели в целом можно разделить на два класса: модели с приращениями (*Incremental*) и эволюционные (*Evolutionary*). В соответствии со всеми этими моделям ПП разрабатывается *итерациями* и каждая итерация заканчивается выпуском работоспособной версии ПП.

Итерационные модели с приращениями

По моделям с приращениями ПП разрабатывается итерациями, с добавлением на каждой итерации функциональных возможностей. При этом вначале определяются *все* требования к ПС, а возможно, и разрабатывается предварительный проект. Дальнейшая разработка ПС разбивается на итерации. В каждой итерации разработка выполняется последовательно и завершается выпуском работоспособной версии ПП. В первой итерации реализуется набор основных требований, обеспечивающих базовую функциональность. Остальные итерации реализуются в порядке критичности требований для конечного пользователя. При появлении в середине итерации нового набора требований они откладываются до реализации следующей версии. В реальной жизни это допущение модели может нарушаться и допускается пересмотр требований. В разных моделях этой группы итерации могут выполняться последовательно или с перекрытием (новая итерация начинается до завершения предыдущей итерации или когда первая стадия предыдущей итерации завершена примерно на 80%).

На рис. 2.6 приведена структура такой модели.

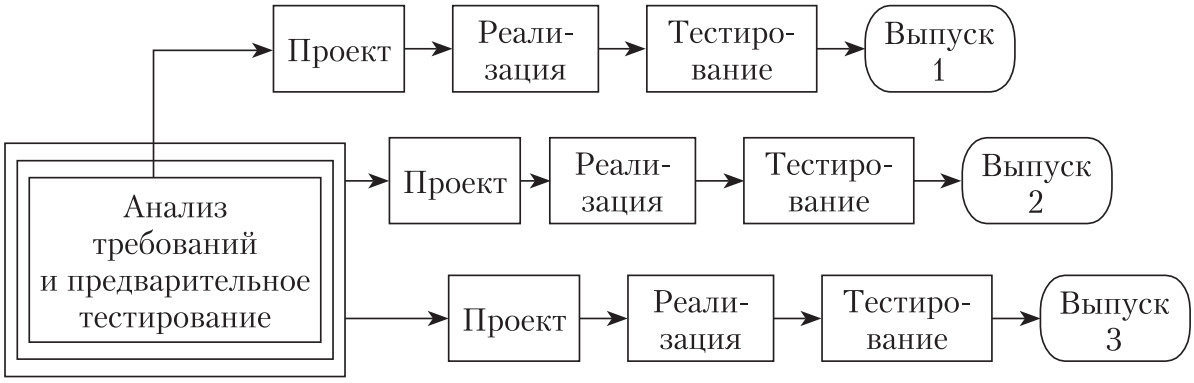


Рис. 2.6. **Итерационная модель с перекрытием итераций**

Итерационные модели с приращениями широко применяются для разработки коммерческих ПП, которые развиваются в течение длительного периода времени или для которых внешние требования изменяются слабо. Возникают риски, связанные с применением моделей и при несоответствии требований.

Эволюционные модели

В отличие от моделей с приращениями эволюционные модели применяются в тех случаях, когда *все* требования не могут быть определены сразу или известно, что они могут измениться. Разработка проекта по этим моделям также выполняется итерациями, но каждая итерация охватывает все стадии разработки — от анализа выбранного набора требований до выпуска версии. На каждой итерации выполняется прототипирование требований и проекта. К наиболее известным эволюционным моделям в настоящее время относятся спиральная модель и модель эволюционного прототипирования.

***Спиральная модель* (*Spiral*)** — разработана Б. Боэмом. Отражает управляемый риском процесс эволюции проекта от анализа до готовности продукта (рис. 2.7).

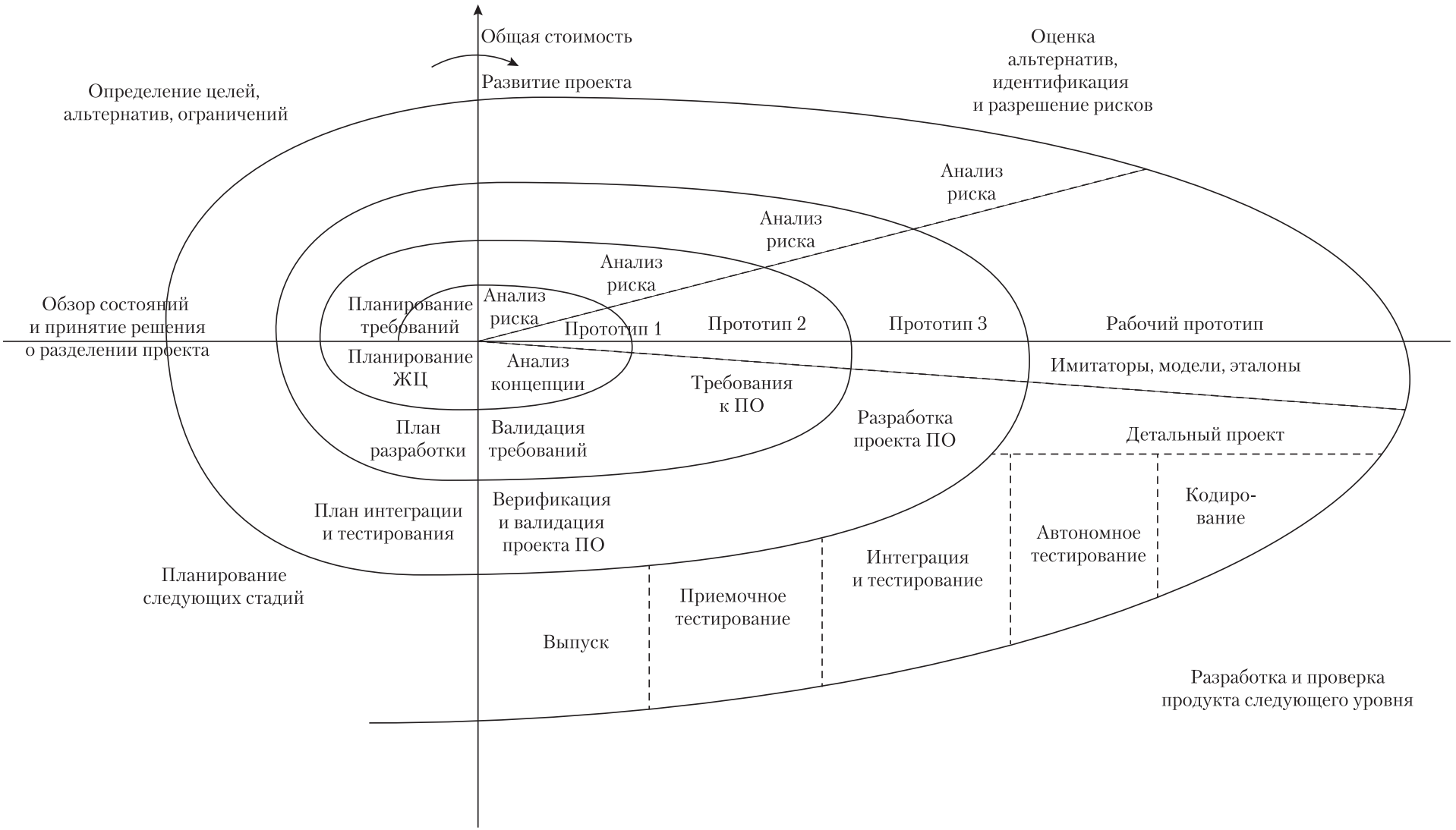


Рис. 2.7. **Спиральная модель ЖЦ**

При пользовании этой моделью анализируются разные виды рисков в работе системы или модели проектирования. Они касаются технических аспектов разработки. Финансовые риски устанавливают соотношение эффективость/затраты и используемых ресурсов, а также определяются риски эксплуатации системы.

Дальнейшим развитием этой модели является *Win-Win Spiral Model*, которая основана на привлечении к разработке разных категорий участников проекта и определении условий успеха (выигрыша) системы или подсистемы, которые обсуждаются на каждой итерации. Возможные альтернативные решения должны оцениваться по отношению к целям и ограничениям проекта.

Модель эволюционного прототипирования основана на применении эволюционного прототипирования в рамках всего ЖЦ разработки (а не только для моделирования требований). В литературе она часто называется моделью *быстрой разработки приложений* (RAD, Rapid Application Development). Моделирование включает следующие шаги.

1. Анализ применимости модели. Изучение возможности применения модели для проекта.
2. Обследование заказчика. Изучение потребностей пользователя и разработка плана создания прототипа.
3. Итерация разработки функционального прототипа. Создание и согласование прототипа интерфейса пользователя, определение нефункциональных требований и стратегии реализации системы.
4. Итерация проектирования и построения. Построение протестированной системы, удовлетворяющей всем функциональным и нефункциональным требованиям.

На шагах 3 и 4 разработчики выполняют определение прототипов, согласование сроков разработки, построение и проверку прототипов. Эти шаги выполняются итеративно и включают три итерации: начальное ознакомление, уточнение и согласование.

5. Реализация. Установка системы в среде заказчика, разработка документации и обучение.

Эта модель применяется для разработки не критических бизнес-приложений, для которых наиболее важными являются функциональные возможности. Ее применение предполагает тесное взаимодействие разработчика и пользователя. Разработка приложений обычно выполняется в среде мощных CASE-средств.

Структура модели представлена на рис. 2.8.

Модели ЖЦ используются практически в разных организациях разработчиков ПП. Комитет ISO/IEC разработал проект стандарта ЖЦ (1996), который проходил длительную апробацию. В этот проект вносились согласованные с 300 организациями изменения, и через 10 лет вышел окончательный вариант стандарта ISO/IEC Life Cycle 2007. Его структура и описание процессов приводятся в гл. 7 учебника.

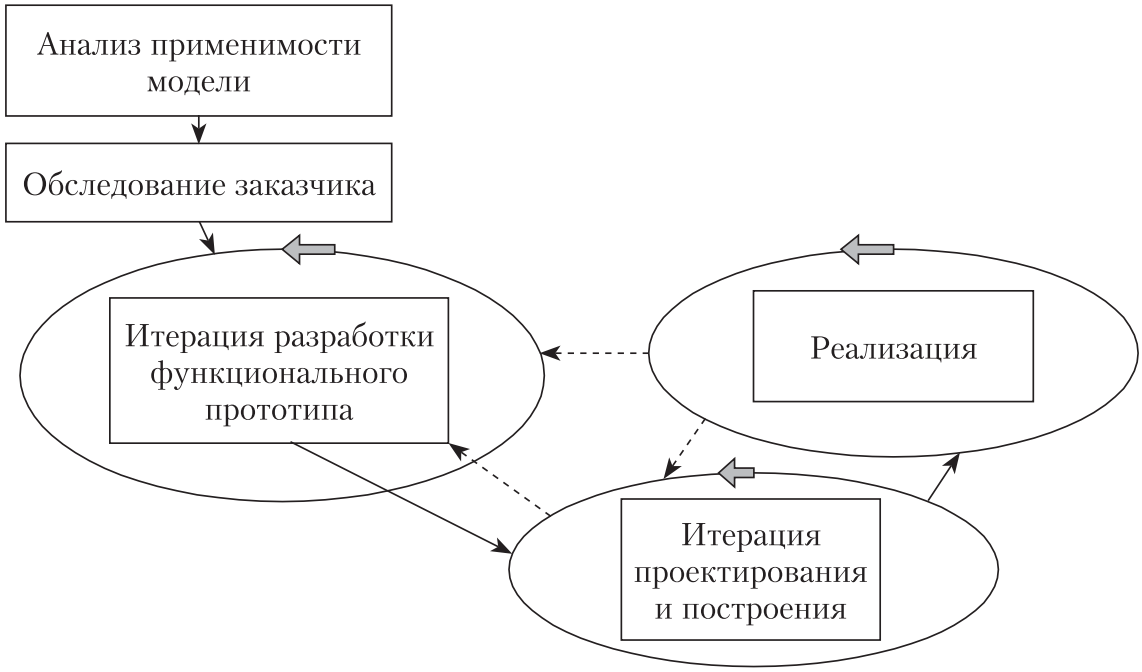


Рис. 2.8. **Модель эволюционного прототипирования**

2.4. Парадигмы программирования SE

SWEBOK включает ряд парадигм программирования[1]. В его учебные курсы по программированию включены следующие: *процедурное программирование* (курс CS1011 «Programming fundamentals»), *объектно ориентированное программирование* (курс CS102I «The Object-Oriented Paradigm»), а также *событийное и параллельное* (*event-based programming*) *программирование* и др. Им соответствуют курсы по ЯП (C++. C#, Basic, Java, Pyton и др.). В состав парадигм входят Agile-технологии. Далее рассматриваются цели и задачи названных парадигм.

2.4.1. Парадигма событийно-управляемого программирования

Стили программирования, обычно связываемые с итеративной парадигмой, основываются на концепции организации «*внутреннего*» *потока управления* выполнением инструкций программы, не связанного с наступлением событий вне ее. Однако существует множество задач, решаемых в реальном времени (в медицине, ядерной энергетике, управлении полетами и др.), а также задач, решаемых в интерактивном режиме, ожидающих, принимающих и обрабатывающих команды и наборы данных извне. Для эффективной реализации таких задач используются стили событийно-управляемого (или событийного) программирования, основанные на концепции организации *цикла обработки внешних событий*. Вместо того чтобы ожидать полного завершения ввода и обработки одного набора данных и лишь потом *после-*

[1] См.: *Лаврищева Е. М.* Парадигмы программирования сборочного типа в программной инженерии // УКРПрог-2014. № 2—3. С. 121—133.

довательно переходить к ожиданию следующего, в системе организуется цикл постоянного опроса очереди поступающих извне событий (будь то появление файла в каталоге, щелчок на кнопке мыши, наступление заданного момента времени или др.) и запускается *функция-триггер* для обработки очередного события. Событийное программирование заключается в подмене системных функций-триггеров «по умолчанию» нужными функциями, соответствующими каждому возможному событию.

Таким образом, программа в стиле событийного программирования обычно состоит из ряда маленьких прикладных программ, называемых *обработчиками событий*, которые запускаются в ответ на появление этих событий, и *диспетчера*, использующего очередь необработанных событий и имитирующего среду, управляемую прерываниями.

Событийное программирование (*event-oriented*, *event-based* или *event-driven programming*) — наиболее распространенная современная парадигма прикладного программирования, поддерживаемая множеством операционных систем и сред разработки программ (в частности, Windows).

Инженерия качества программ, разработанных с применением событийной парадигмы, заключается в тестировании операционных (сценарных) профилей (гл. 7) и в прогнозировании размера ПП методу FPA (гл. 8). К подобным стилям можно отнести согласованное (параллельное) программирование.

2.4.2. Согласованное программирование и параллельные вычисления

Резерв повышения производительности компьютеров — распараллеливание потока команд или потока данных. Распараллеливание данных подразумевает применение одной операции сразу к нескольким элементам массива данных, а параллелизм задач — разбиение вычислительного процесса на несколько подзадач (процессов), каждый из которых выполняется на своем процессоре.

Достаточно быстро были разработаны суперЭВМ, имеющие векторную или матричную архитектуру процессоров (транспьютеров), автономных, но способных обмениваться между собой результатами своих вычислений через *каналы связи*. А к началу 2006 г. главными элементами производственных программ ведущих компаний — разработчиков микропроцессоров стали *многоядерные процессоры*[1].

Параллелизм может быть обеспечен на двух уровнях: параллелизм уровня *микроопераций* и параллелизм уровня *процессов* — простых (послать, принять значение или процессы вычислительной модели) или структурных (последовательных или параллельных). Когда процесс выполняет инструкцию «Принять значение (из канала)», он входит в состояние ожидания, пока канал пуст. Как только в канале появляется значение, процесс его считывает и продолжает работу. В системе параллельных процессов каждый отдельный процесс обрабатывает события.

---

[1] См.: *Рыбаков А.*, *Золотарев С.* Программное обеспечение многоядерных систем. URL: http://old.osp.ru/os/2006/02/012.htm.

Активное внедрение новых аппаратных архитектур потребовало смены парадигмы программирования — перехода от последовательного программирования к *согласованному программированию* (*concurrent programming*), в частности *параллельному программированию* (*parallel programming* или параллельным вычислениям — *parallel computing*), связанному с параллельными потоками, порождением и обработкой асинхронных событий и др.

Согласованное (параллельное) программирование — это программирование в терминах взаимодействия некоторых одновременно работающих вычислителей (нескольких процессоров или многоядерного процессора).

При разработке параллельных программ используются специализированные языки, как, например, язык высокого уровня Occam (неразрывно связанный с транспьютерами), а также современные библиотеки и системы параллельного программирования PVM, LAM, CHMP и др. Три основных подхода к реализации этих систем различаются методами взаимодействия параллельных задач. Первый подход базируется на концепции *обмена сообщениями*, второй — на использовании *разделяемой памяти*, третий опирается на *стандарт POSIX* и объединяет оба подхода.

Наиболее известной реализацией первого подхода является спецификация MPI (*Message Passing Interface*) для языков C и Fortran, а второго подхода — спецификация OpenMP для Fortran, C, C++. OpenMP — это набор специальных директив компилятору (pragma), а также библиотечных функций и переменных среды. Компилятор, поддерживающий OpenMP, преобразует исходный код программы — вставляет соответствующие вызовы функций для параллельного выполнения областей кода, выделенных с помощью ряда специального директив компилятору. Поддержка спецификации OpenMP обеспечена во всех компиляторах Intel, начиная с шестой версии, в Microsoft C и C++ начиная с Visual Studio 2005, а также в GCC. В третьем подходе используется спецификация POSIX (*Portable Operating System interface for unIX*), введенная как международный стандарт ISO/IEC 9945—1:1990. В рамках POSIX можно реализовать параллельные вычисления на основе обмена сообщениями (аналогично MPI) или разделяемой памяти (как в OpenMP). Естественно, в POSIX допустима и любая комбинация этих методов.

В работе А. Рыбакова и С. Золотарева отмечается, что многоядерные процессоры — это единственное перспективное направление для дальнейшего развития компьютерной индустрии, поскольку «ресурсы трех слонов, на которых стояла вся индустрия, оказались исчерпанными: тактовая частота не растет; оптимизация исполнения в пределах одного ядра достигла предела, за которым рост сложности становится неоправдан; у дальнейшего наращивания размеров кэш-памяти процессоров остался лишь небольшой зазор для развития. Как следствие, старый добрый мир остался в прошлом. Ведущий эксперт корпорации Microsoft Херб Саттер в статье “Программное обеспечение и революция согласования” (ACM QUEUE, September 2005) называет происходящее революцией по масштабам существенно более мощной, чем распространение объектно ориентированного программирования в начале 1990-х гг. Тогда причиной перемен стало усложнение программных систем, нынешняя же революционная ситуация вызвана необходимо-

стью выполнять программы на многоядерных процессорах». Очевидно, что проблемы инженерии качества таких программ (основанных на принципах согласования и параллелизма) требуют отдельного рассмотрения и формирования новых направлений исследований.

2.4.3. Парадигма ООП

ООП является мощным и универсальным средством программирования и проектирования систем. ООП — парадигма, основанная на представлении ПрО в виде системы взаимосвязанных абстрактных *объектов* и их реализаций[1].

В этой парадигме различают два стиля — программирование, основанное на классах (*class-based programming*) и программирование, основанное на прототипах (*prototype-based programming*).

Программирование «на классах»

В отличие от стиля процедурного программирования, в котором данные и функции их обработки не связаны, в этом стиле определена новая структура данных — *класс* — по сути дела это тип объекта, в котором кроме данных (свойств, параметров объекта) содержатся функции их обработки (методы, допустимые действия объекта или с объектом).

Объект по отношению к своему классу является *экземпляром*. Класс определяет структуру и функциональность (*поведение*), одинаковую для всех экземпляров данного класса. Один класс отличается от других именем и обычно набором поддерживаемых интерфейсов. Интерфейсы, в свою очередь, представляют собой набор сообщений, которые можно посылать объекту. Экземпляр является носителем данных, т.е. обладает *состоянием*, меняющимся в соответствии с поведением, заданным классом. В языках ООП новый экземпляр создается через вызов конструктора класса (возможно, с набором параметров). Получившийся экземпляр имеет структуру и поведение, жестко заданные его классом.

Все объекты могут обмениваться между собой *сообщениями*. При получении объектом сообщения запускается соответствующий ему метод. Объект имеет ассоциативный *контейнер*, позволяющий получить по сообщению нужный метод для обработки. Если метод для обработки сообщения не найден, сообщение перенаправляется объекту-*предку*.

Принципами ООП являются:

- *наследование* — механизм установления отношений «потомок — предок» (порождать один класс от другого с сохранением всех свойств и методов класса-предка);
- *инкапсуляция* — свойство сокрытия реализации класса;
- *абстракция* — описание взаимодействия исключительно в терминах сообщений/событий в ПрО;
- *полиморфизм* — возможность подмены в сценарии взаимодействия объектов одного объекта другим объектом со сходной структурой.

---

[1] См.: *Буч Г.* Объектно-ориентированный анализ. М. : Бином, 1998; *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя : пер. с англ. М. : ДМК, 2000.

Многие современные языки специально созданы для программирования на классах, например Smalltalk, C++, Java, Python, PHP, Object Pascal (Delphi), VB.NET, Xbase++ и др. Однако следует отметить, что можно применять приемы ООП и для не объектно ориентированного языка и, наоборот, применение объектно ориентированного языка еще не означает, что код автоматически становится объектно ориентированным. Парадигма ООП на классах активно используется при проектировании сложных систем в языке UML[1].

Программирование по прототипу

В этом стиле программирования понятие класса отсутствует, а повторное использование (наследование) производится путем клонирования существующего экземпляра объекта — прототипа.

В отличие от стиля программирования, основанного на классах и концентрирующего внимание разработчика на таксономии классов и на отношениях между ними, стиль прототипного программирования заостряет внимание на поведении небольшого количества «образцов», которые далее используются как «базовые» объекты для создания других объектов.

Создание нового объекта выполняется одним из двух методов: *клонированием* существующего объекта либо созданием объекта «*с нуля*». Для второго метода предоставляются синтаксические средства добавления свойств и методов в объект. В дальнейшем для построенного объекта может быть получена полная копия — клон. В процессе клонирования копия наследует все характеристики своего прототипа, но с этого момента становится самостоятельной и может быть изменена. В некоторых реализациях копии хранят ссылки на объекты-прототипы, *делегируя* им часть своей функциональности; при этом изменение прототипа может затронуть все его копии. В других реализациях новые объекты обретают полную *независимость* от своих прототипов: прототип копируется один в один, со всеми методами и атрибутами, и копии присваивается новое имя (ссылка). Преимущество данного подхода в том, что создатель копии может ее менять, не опасаясь побочных эффектов среди других потомков своего предка. Кроме того, существенно снижаются вычислительные затраты на диспетчеризацию, так как нет необходимости обходить всю цепочку возможных делегатов в поисках подходящего метода или атрибута. В числе недостатков — трудности с распространением изменений в системе: модификация прототипа не влечет за собой немедленное и автоматическое изменение всех его потомков. Другой недостаток состоит в том, что простейшие реализации этой модели приводят к увеличенному (по сравнению с моделью делегирования) расходу памяти.

Каноническим примером прототипного языка является язык Self. Прототипный стиль программирования положен также в основу таких ЯП, как JavaScript, Squeak, NewtonScript, MOO, REBOL, Kevo и др.

2.4.4. Agile-технологии

В области SE специалисты Кент Бек (Kent Beck), Мартин Фаулер (Martin Folwer), Кен Шваубер (Ken Schwaber), Джефф Сазерленд (Jeff Sutherland)

---

[1] См.: *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя.

и другие образовали некоммерческую организацию *Agile Alliance*, пропагандирующую Agile Software Development, методологию eXtreme Programming, SCRUM, DSDM, ASD, Crystal, FDD, Pragmatic Programming и др.[1]

Методологии Agile ориентированы на тесное взаимодействие команды разработчиков с пользователями, итеративную модель ЖЦ с приращениями и быструю реакцию на изменяющиеся требования и продукто-ориентированный подход к разработке ПО. Такой подход к разработке ПС обеспечивает уменьшение зависимости не только от постоянных изменений функциональных требований заказчика, но и от проблем неритмичности общего финансирования и плана-графика разработки, а также быстрой смены технологической среды разработки. Самой широко известной Agile-методологией является **eXtreme Programming (XP)** (экстремальное программирование).

В XP реализуется принцип «коллективного владения кодом». В нем любой член группы может изменить не только свой код, но и код другого программиста и каждый модуль снабжается автономным тестом (*unit test*), обеспечивая возможность регрессионного тестирования модулей. Тесты пишут сами программисты, и они имеют право написать тесты для любого модуля. Таким образом, большинство ошибок исправляются на стадии кодирования, либо при просмотре кода, либо при динамическом тестировании.

SCRUM — гибкая методика управления проектом фирмы Advanced Development Methods, Inc., широко используется в сотнях организаций (Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox and Hewlett-Packard и др.). Эта методика задает итеративную модель ЖЦ с четко определенным процессом разработки, включающим анализ требований, проектирование, программирование, тестирование (http://agile.csc.ncsu.edu).

DSDM (*Dynamic Systems Development Method*) является методом разработки информационных систем консорциума DSDM. Он использует модель быстрой разработки RAD (*Rapid Application Development*) с прототипированием для достижения целей построения систем в условиях ограниченных ресурсов проекта.

ASD (*Adaptive Software Development*) — методология разработана Д. Хайсмитом (президентом Information Architects, Inc.) специально для экстремальных проектов и базируется на теории сложных адаптивных систем. В ней статический цикл разработки ПС включает *планирование — проектирование — программирование*, а динамический цикл: *размышление — взаимодействие — обучение*.

FDD (*Feature Driven Development*) — методология разработки ПС ориентирована на функциональные возможности, предложена Д. ДеЛуком для крупного банка (http://www.nebulon.com). FDD характеризуется модельно-ориентированным (*model-driven*) процессом разработки, включающим три *последовательные фазы*, в ходе которых формируется общая модель системы, и две *итеративные фазы*, выполняемые для каждой выделенной функции системы.

---

[1] Agile — манифест разработки ПО. URL: http://www.agilemanifesto/org/iso.

Контрольные вопросы и задания

1. Определите понятие технологии программирования.
2. Определите понятие SE.
3. Назовите основных разработчиков SE.
4. Перечислите инженерные дисциплины CS.
5. Дайте характеристику SWEBOK.
6. Определите области знаний — требования, проектирование и конструирование.
7. Что такое доменная инженерия ПО и систем?
8. Определите области знаний — управление проектом и качеством.
9. Определите понятие ЖЦ.
10. Приведите примеры моделей ЖЦ.
11. Определите инженерию КПИ.

Глава 3
НОВЫЕ ПОДХОДЫ К РАЗРАБОТКЕ ИЗМЕНЯЕМЫХ ПП ИЗ КПИ

Программные системы появились в 1980-х гг. для организации вычислений математических, экономических и других задач на ЭВМ. Базовым элементом таких систем были отдельные программы, подпрограммы и др. Ранее разработанные модули и программы становились готовыми для использования и накапливались в библиотеках стандартных программ или в банках модулей.

В эти годы Э. Дейкстра впервые ввел термин **«семейство программ»**. Он означал совокупность программ, которая обладает общими функциональными характеристиками, каждая из которых может изменяться независимо друг от друга. Программы и модули семейства описывались в ЯП и собирались из них с использованием процедур, функций, подпрограмм и т.п. Изменения вносились в любые элементы программ семейства на интуитивном уровне, а затем они повторно объединялись в одну систему.

Одновременно возникло понятие **программной системы** как совокупности отдельных программных объектов, которые реализуют взаимосвязанные функции некоторой ПрО для выполнения в заданной операционной среде. **Семейство программных систем (СПС)** — это совокупность ПС, которые определяются общим множеством понятий и свойств, присущих всему семейству, и множеством отдельных специальных свойств, которые присущи каждому отдельному члену семейства ПрО.

С начала 2000-х гг. Институт программной инженерии (SEI) США (www.sei.com) ввел термин **семейство программных продуктов** — product family (семейство продуктов, СПП) или **продуктовая линия** — product lines. В словаре ISO/IEC FDIS 24765:2009(E) — Systems and Software Engineering Vocabulary они определены как «группа продуктов или услуг, которые имеют общее управляемое множество свойств, удовлетворяющих потребностям определенного сегмента рынка или вида деятельности».

Таким образом, классическое определение семейства СПС эволюционировало к семейству продуктов, изготовляемых на продуктовых, технологических линиях фабрик производства ПС из готовых продуктов, КПИ типа reuses, которые накапливаются в библиотеках.

Для обеспечения изменчивости отдельных элементов семейства ПП Клаус Похл ввел понятие **вариабельности**, определяемое на модели FM-элементов, помеченных вариантными точками[1]. При этом термин

---

[1] См.:*Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение. СПб. : Питер, 2005; *Pohl K., Böckle G., Linden F. J.* Software Product Line Engineering: Foundions, Principles and Techniques. Springer Verlag, 2005.

«вариабельность» означал способность семейства ПС, Software Product Lines или отдельной системы, приложения и их компонентов к расширению, изменению, приспособлению или конфигурированию с целью использования в конкретной ПрО.

3.1. Инженерия изготовления ПП в Product Line

В рамках технологии Product Line определены следующие инженерии — *domain engineering* и *application engineering* (рис. 3.1).

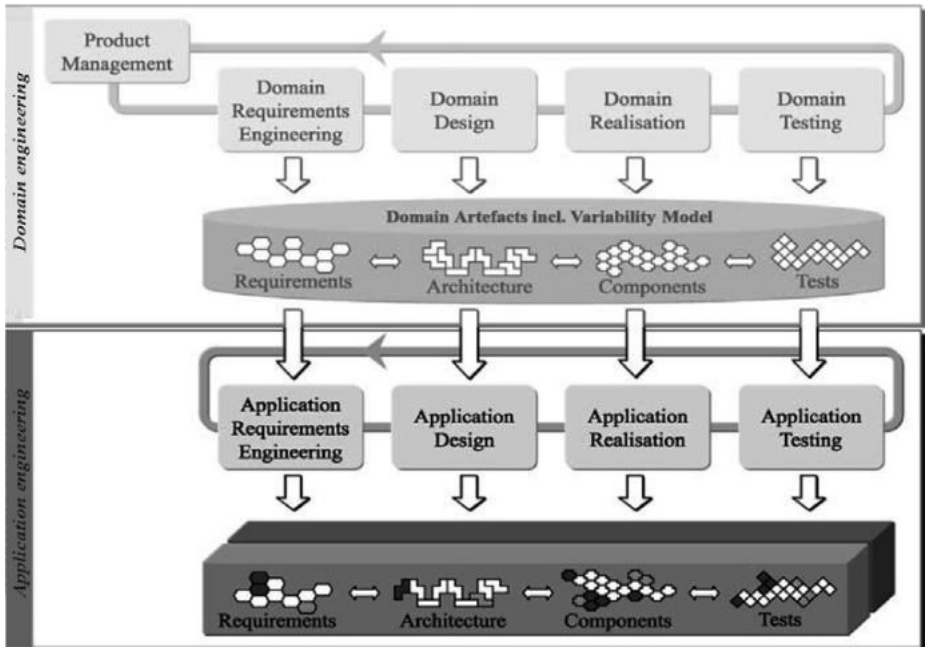


Рис. 3.1. **Структура Software Product Lines**

Процесс инженерии домена состоит в определении и реализации общности (*commonality*) элементов и обеспечении вариабельности ПП на продуктовой линии для получения вариантов продукта.

Инженерия домена включает вариабельность артефактов (*requirements*, *architecture*, *components*, *test cases* и др.) в архитектуре. Изменения в СПП вносятся на уровне домена и его приложений.

Модель вариабельности домена включает набор элементов, помеченных вариантными точками, по которым задается изменчивость объектов системы на линии ПП (*the software product line*) на основе констрейнов, устанавливающих зависимости между изменяемыми объектами модели *FM*-приложения (Application Variability Model), используемой для получения вариантов членов СПП.

Процесс Software Product Lines обеспечивает вариабельность СПП по модели FM, задаваемой на уровне требований, архитектуры, документации, тестов и др. FM реализуется в СПП и ПС семейства.

3.1.1. Вариабельность продуктов и семейств на линиях SPLE

Клаус Похл определил FM в Product Line для элементов системы, помечаемых вариантными точками.

При производстве коммерческих продуктов возникла необходимость создавать разные варианты ПП в связи с обнаруженными недоработками ПП или введением новых функций. В результате сформировалась и экспериментально отработана модель вариабельности ПП и семейств СПС. Инженерия SPLE с моделью FM разработана и представлена в работе Похла с соавторами[1] института SEI USA. Этим нововведением заинтересовались многие специалисты, стали проводиться конференции по проблеме вариабельности ПС, в том числе VAMOS 2007—2015.

Модель FM определяется аналитиками и разработчиками при проектировании архитектуры ПП. Она включает общие функциональные и нефункциональные характеристики элементов продукта (рис. 3.2), которые используются членами семейства при реализации задач автоматизируемого домена.

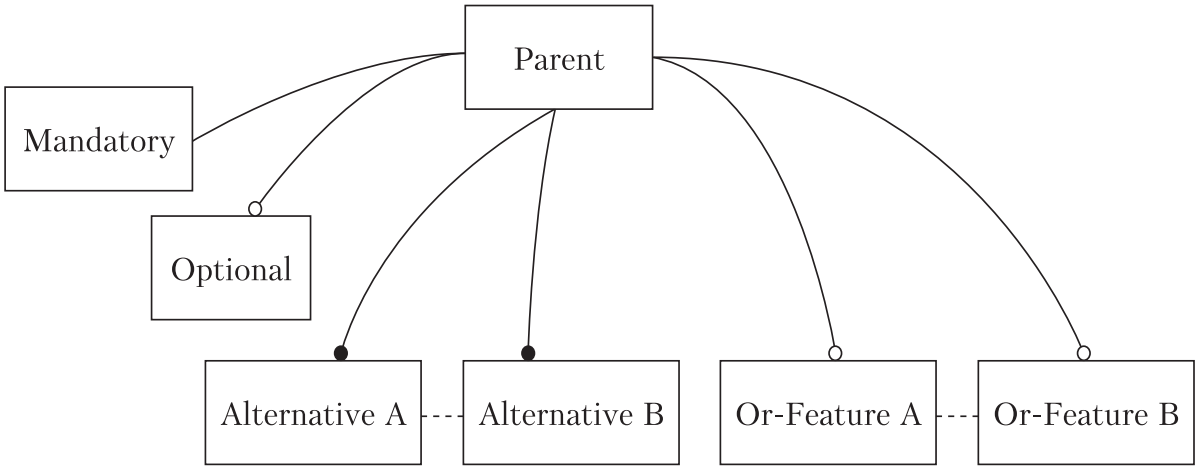


Рис. 3.2. **Вид модели FM**

Задача обеспечения вариабельности ПС базируется на методах разработки и конфигурирования продукта (*product configuration*) из готовых reuses и процессов управления конфигурацией (*configuration management*), направленных на определение и внедрение эффективных процедур разработки и эволюции систем, а также их адаптации к новым условиям функционирования в современных гетерогенных средах[2].

К главным аспектам вариабельности ПП и систем отнесены:

- моделирование вариабельности на уровне внешних характеристик (ВХ) объектов;
- реализация вариабельности ПС на уровне артефактов и архитектуры СПС;
- управление вариабельностью (планирование, контроль и регуляция объектов) в ходе ЖЦ ПС и ПП.

В основе SPLE лежит процесс создания ПП из базовых элементов, которые называют готовыми продуктами или КПИ. К ним относятся reuses,

---

[1] *Pohl K., Böckle G., Linden F. J.* Software Product Line Engineering: Foundions, Principles and Techniques.

[2] См.: URL: http://www.developer.com/news/techfocus/022398_dist1; www.01.ibm/com/software/ru/webspere.

assets, application и др. На их основе формируются варианты ПП и СПС из КПИ.

Преимущество применения КПИ и GoP в СПС заключаются в том, что они могут быть присоединены к разработанной ПС и СПС и тем самым добавить новую функциональность к ПС.

3.1.2. Процессы разработки ПП в SPLE

Разработка продукта включает четыре базовых процесса: определение требований к проекту, проектирование структуры домена, реализация домена, тестирование домена.

Требования к домену. Требования — это раздел инженерии домена для извлечения и документирования общих и изменчивых требований. На вход процесса подаются повторно используемые, текстовые документы модели изделий с изменчивостью. Выходом является общее техническое задание для проекта. Требования анализируются, чтобы идентифицировать те из них, которые являются общими для всего домена, и те, которые являются специфическими для отдельных приложений домена, которые различаются между собой.

Проектирование структуры домена. Этот подпроцесс служит для определения стандартной архитектуры на PL. Архитектура — это общая, высокоуровневая структура для всех линий Product Line. Проектирование домена включает механизмы конфигурации архитектуры для поддержки изменчивости на Product Line повторно используемых частей проекта и правил для разработки специфики приложений СПП по стандартной архитектуре.

Реализация домена — это детальное проектирование и пополнение КПИ для ПП. Каждый компонент планируется, разрабатывается и выполняется многократно в различных контекстах приложения на Product Line и в конфигурационном файле.

Тестирование домена — это процесс, который обеспечивает проверку правильности КПИ, требований, архитектуры и артефактов. При тестировании домена могут использоваться различные стратегии тестирования интегрированных элементов, которые содержат изменчивые части. После тестирования используются механизмы гарантии качества ПО, включая разного рода инспекции, обзоры и анализ работы системы и ее артефактов.

Документирование архитектуры включает:

- ортогональную (*orthogonal*) модель и с добавлением внутренней вариабельности, которая видна инженерам;
- реализацию артефактов (*realisation artifacts*) в проекте системы и их выполнение;
- повторно используемые компоненты ПС и их интерфейсы.

Испытание артефактов и ПП включает планирование тестов, тестовых случаев и сценариев выполнения тестов домена. План тестирования определяет стратегии тестирования домена, артефактов и тестовых случаев при выполнении списка и распределении ресурсов для тестовой активности домена. Тестовые случаи и сценарии записываются в детальных инструкциях для тестового инженера.

3.1.3. Процессы инженерии доменов

Согласно рассмотренным подходам К. Похла и К. Чернецки к разработке доменов в стандарт ISO/IEC 12207:2002:ЖЦ был включен процесс инженерии доменов (*Domain engineering process*) как новый процесс в модели процессов ЖЦ. Согласно этому стандарту процесс инженерии доменов охватывает ряд видов деятельности.

Анализ домена состоит:

- в определении границ домена и связей с другими доменами;
- выявлении, формальном описании общностей и отличительных особенностей внутри домена (постоянных и переменных требований) и в их включении в модели домена;
- формировании словарей для описания основных понятий в домене и взаимосвязей между активами в домене;
- классификации и документировании моделей;
- оценке моделей и словарей домена с учетом выбранной методологии моделирования.

Проектирование домена (*Domain design*) — это определение архитектуры домена на основе программных компонентов — специфичных ресурсов.

Кроме рассмотренного стандарта отметим проект стандарта комитета OMG (http://www.omg.org) относительно повторных компонентов RAS (Reusing Assets Specifications). В нем предлагается формат описания информации относительно повторно используемых ресурсов при разработки ПС, которое является обобщением описания с названием *asset* — вклада (актив) программного решения в виде готовой реализации некоторой задачи в заданном контексте некоторой программы, которая относится к проблеме разработки ПС, а контекст — к процессу разработки или выполнения ПП. Asset — это по сути дела компонент или КПИ, т.е. рабочий продукт или средство разработчика, которые могут быть использованы в других программных разработках. Понятия *asset* можно понимать и шире, аналогично понятию *готового ресурса* (ГоР) в ПИ.

Архитектура домена — каркас для КПИ, активов и формально определенных интерфейсов — должна согласовываться с моделью домена, стандартами организации и оцениваться на соответствие выбранной методологии архитектурного проектирования.

Технология инженерии домена, которая базируется на новом процессе модели ЖЦ (ISO/IEC 12207), понятии ресурса (*asset*), включает следующие стандартизированные подпроцессы:

- формирование ресурсов (*Asset provision*), т.е. разработка или приобретение ресурсов (активов), которые могут использоваться при компоновке новых программных систем или подсистем;
- разработка базы ресурсов (*Asset-based development*), в основе которой лежит концепция повторного использования (*software reuse*) — КПИ, обеспечивающая компоновку ПП домена;
- сопровождение ресурсов (*Asset maintenance*) — модификация и эволюция модели, архитектуры и продуктов домена за счет ГоР типа КПИ.

Данная технология предполагает разработку методик и инструментов для эффективного ее выполнения, а также для генерации системы из КПИ и компонентов многоразового применения на основе спецификаций требований к системе.

В результате применения технологии доменной инженерии в софтверной организации будет создаваться, поддерживаться и развиваться *архитектурный базис* из множества КПИ, хранящийся в репозитарии и учитывающий общие и специфические особенности разных сторон деятельности в доменах.

Основное требование к инженерии ПрО — обеспечение многоразового применения используемых решений для семейства ПС, а к инженерии приложений — производство (линейка) одиночной системы из КПИ по требованиям к ней.

3.2. Инженерия повторного использования КПИ/Reuse

Инженерия повторного использования компонентов (КПИ) — это систематическая и целенаправленная деятельность по подбору реализованных программных артефактов и представленных в виде КПИ, анализу их функций для добавления в качестве готовых в проектируемую систему и их интеграция с другими компонентами.

КПИ (или *Reuse*) — многоразовые компоненты. Повторность (*Reusebility*) (1987) — процесс создания ПС из reuses. В 2016 г. пройдут 15 конференций по Software Reuse, на них будут рассмотрены новые подходы и виды КПИ.

Основные свойства КПИ: функциональность, удобство использования и качество реализации.

КПИ — это некоторая функция с определенными атрибутами, обеспечивающими решение некоторой задачи; совокупность методов определенной сигнатуры и ТД, которые передаются и возвращаются после выполнения метода.

КПИ располагаются в каталоге (репозитории), с помощью которого можно понять, какие КПИ имеются, их функцию и как их можно соединить в программную структуру.

Инженерия повторного использования предусматривает **два процесса разработки** ПС из КПИ.

Первый процесс — это создание КПИ путем:

- изучения спектра решаемых задач ПрО, выявление среди них общих свойств и функций;
- построения компонентов, реализующих выявленные функции в виде КПИ;
- разработки каталога для хранения изготовленных компонентов и организации поиска необходимых компонентов по запросам пользователей.

Второй процесс — конструирование новых систем из готовых компонентов КПИ путем:

- понимания сущности новой системы, определения целей ее создания и предъявляемых к ней требований;

• поиска в каталоге готовых компонентов, которые подходят для использования в новой системе;

• сопоставления цели новой разработки с возможностями найденных КПИ и принятия решений о целесообразности и месте их применения в системе;

• интеграции КПИ в новую разработку с обеспечением интерфейса с подсистемами и другими компонентами.

Бизнес повторного использования, как любой бизнес, нуждается в специальных условиях по менеджменту всей инженерной деятельности инженерии систем из КПИ. Критерии успеха такого бизнеса определяются следующими предпосылками:

1) повторное использование КПИ требует меньших трудозатрат, чем разработка их как новых продуктов;

2) поиск пригодных КПИ для дальнейшего использования;

3) настройка компонентов на новые условия среды применения.

Основная парадигма КПИ — «писать — один раз, выполнять — много раз и где угодно». Архитектура, в которую встраивается готовый КПИ, поддерживает стандартные механизмы для работы с компонентами как со строительными блоками. Чтобы обеспечить высокий уровень использования КПИ, они должны обладать такими основными свойствами, как функциональность, удобство использования и качество реализации.

3.2.1. Разновидности КПИ

В качестве КПИ могут использоваться формализованные артефакты деятельности разработчиков ПС, которые отражают некоторую функциональность для применения в новых разработках. Под **артефактом** понимается реальная порция информации, которая может создаваться, изменяться и использоваться при выполнении деятельности, связанной с разработкой ПС различного назначения[1].

Артефактами КПИ могут быть:

• продукты процесса разработки ПС (требования, постановки задач, архитектура и др.);

• описания результатов процесса разработки ПС (спецификация, модель, каркас и т.п.);

• готовые компоненты ПС или отдельные части системы.

К КПИ выдвигаются такие требования, как независимость от конкретной платформы, наличие стандартного интерфейса и параметров настройки на новую среду, для взаимодействия в системе.

Разработке ПС с помощью КПИ соответствует модель ЖЦ со следующими общими этапами:

• анализ объектов и отношений реализуемой ПрО для выявления КПИ, обладающих общими свойствами, присущими группам объектов этой области;

• адаптация имеющихся в базе репозитария КПИ;

---

[1] См.: *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя.

• разработка интерфейсов компонентов и их размещение в репозитарии интерфейсов системы;

• интеграция КПИ и их интерфейсов с другими элементами создаваемой системы и формирование конфигурации этой системы.

На рынке ПП циркулируют следующие виды готовых компонентов:

• классы объектов, абстрактные классы, функции на ЯП;

• структуры данных и часто используемая информация (например, информационные ресурсы, веб-сервисы);

• API-, IDL-модули в библиотеках C++ (например, GUI, графика и др.);

• математические методы (Mat.Lab, Demral и др.);

• готовые решения — паттерны, фреймы, вклады, артефакты и др.

Примером КПИ могут быть контейнерные классы, которые хранят структуры данных с правилами их запоминания или выдачи очередного элемента, входящего в контейнер. Механизм контейнеров реализован в C++ в виде так называемых шаблонов (*templates*).

Все используемые КПИ должны настраиваться для повторного использования в ПС.

3.2.2. Спецификация КПИ

В качестве КПИ могут быть объекты, созданные в рамках объектно ориентированного программирования (например, огромная библиотека повторно используемых классов в C++) с наследованием их реализации.

Каждый компонент имеет:

• связь компонентов через интерфейсы на этапах разработки системы;

• инкапсуляцию в виде «черного ящика» без возможности вмешательства в выходной код;

• наследование интерфейсов, их изменение и настройку на применение;

• повторное использование исходного или выходного кода.

С общей точки зрения компонент определяется по-разному в зависимости от среды и функций, приведем некоторые их них[1].

Каждый КПИ должен быть специфицирован в ЯП, а их паспорта — в IDL, WSDL. Каждый элемент спецификации включает видимую или скрытую от пользователя информацию.

Информационная часть содержит следующие сведения: назначение, дата изготовления, условия применения (ОС, платформа и т.п.), возможности КПИ, тип среды окружения и др.

Внешняя часть — это интерфейс, который определяет взаимодействие компонента с внешней средой и с платформой, на которой он будет выпол-

---

[1] См.: *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя; *Лаврищева Е. М.* Новые теоретические основы технологии производства семейств программных систем в контексте ГП : *e*-монография / Е. М. Лаврищева [и др.]. Киев : Изд-во ИПС НАНУ, ВИНИТИ, 2011; *Batory D., O'Malley S.* The Design and Implementation of Hierarchical Soft ware Systems with Reusable Components // ACM Transactions on Software Engineering and Methodology. 1992. Vol. 1. № 4. P. 355—398; *Jacobson I., Griss M., Johnson P.* Software Reuse: Architecture, Process and organization for Business Success — Addison Wesley, Reading MA, 1997.

няться. *Внутренняя часть* компонента — это программный фрагмент кода, системная или абстрактная структура, представленные в виде каркаса компонента, спецификации и выходного кода. Данная часть содержит интерфейс (*interface*). В нем имеются операции вызова и описание параметров в языках IDL, APL и др. Интерфейс — видимая неизменная и обязательная часть спецификации компонента. Например, система Inspector Components изменяет некоторые параметры интерфейса компонента без вмешательства в код.

Параметры интерфейса определяются типом КПИ и включают инвариант спецификации с указанием типа и имени компонента, его входных и выходных параметров, методов компонента и др. В языке JAVA, например, типами компонентов могут быть: проекты, формы (AWT-компоненты), beans-компоненты, CORBA-компоненты, RMI-компоненты, стандартные классы-оболочки, БД, JSP-компоненты, сервелети, XML-документы, DTD-документы и т.п.

Компонент описывается в ЯП, которое не зависит от ОС (виртуальная машина JAVA) и от реальной платформы (в системе CORBA), где он будет функционировать. Объектом КПИ может быть паттерн — абстракция, которая содержит описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников. Он тоже специфицируется в подходящем ЯП, а информационная часть — в LDL или WSDL.

3.3. Подходы к созданию вариантов ПП

Многолетние исследования проблем индустрии ПП показали, что одной из ключевых проблем обеспечения работоспособности и качества ПП является сопровождение продукта с целью выполнения, обнаружения различных ошибок и внесения изменений в продукт. Особенно остро этот вопрос стоит в системах, которые изготовлены старыми традиционными способами. В связи с этим крупные производители общесистемных и коммерческих ПП подошли вплотную к реализации проблемы создания изменяемых, *вариабельных* ПП. Эта проблема стала в центре внимания разработчиков и научных специалистов. Опубликовано много статей и докладов научных конференций по вариабельности ПП. Главными среди этих подходов являются SPLE (2004—2015); Reusebility (1987—2015); Linux Variability (2008—2015), VAMOS (2007—2015) и др.

В рамках научных исследований в этой области изучаются методы и процессы производства ПП, ПС и их семейств СПС, а также механизмы вариабельности для управления реализацией разных вариантов программных продуктов.

Проанализированы новые подходы и методы построения ПП на основе модели FM из артефактов и готовых reuses, КПИ, а также приложений СПС.

Анализ показал, что работы по созданию систем, их версий и вариантов артефактов, КПИ и характеристик модели вариабельности ПП в мировой индустрии ведутся по трем основным направлениям:

1) моделирование вариабельности в требованиях посредством *диаграмм характеристик — FODA* и основанные на подходе *featureRSEB*, Рибича, *Forfamel*, *RequiLine* и *CBFM*;

2) моделирование вариабельности с помощью механизмов UML — *сценариев использования* (use cases);

3) моделирование вариабельности в пространстве проблем и решений К. Чернецки *ConIPF*, *CONSUL/Pure*:*Variants*, *GEARS*, подход Ван дер Хока, которые интегрируют вариабельность в существующий язык моделирования архитектуры, а также подходы Гома и Вебера по *VSL* (*Variability Specification Language*) и моделированию вариантов из изменяемости артефактов ПС.

Понятие вариабельности и подход к ее моделированию нашли отражение в VSL, AMPL (*Asset Model for Product Lines*), веб-инструментах KBuild и KConfig (http://www.Sap.org) и Pure:Variants (Pure Systems Website) в объектной модели и языке Prolog. Для моделирования вариабельности используется фрейворк *COVAMOF* (*ConIPF Variability Modeling Framework*) (рис. 3.3).

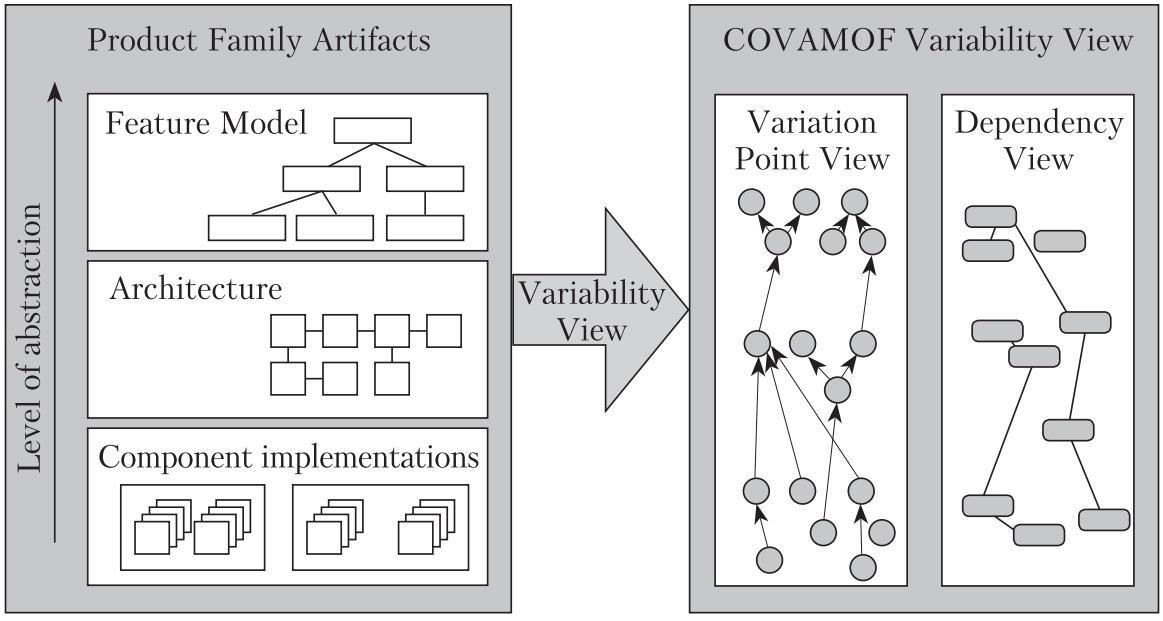


Рис. 3.3. **Ракурсы представления вариабельности в COVAMOF**

Инструмент *COVAMOF* может погружаться в среду MS Visual Studio и моделирует вариабельность, используя онтологию домена, реализованную как plug-in в Eclipse.

Характеристики СПП должны отвечать требованиям к ПП.

Между характеристиками и требованиями существует множество ($nm \times \times x_0 \times x_0$) связей. Каждая ПС имеет группу характеристик, каждая из которых относится к *вариантной характеристике* (ВХ) и входит в их коллекцию. Каждый член коллекции соответствует требованию к артефактам в СПП и присоединяется к ядру СПП в точках вариантности.

Точка вариантности — это место в ПС, по которой осуществляется выбор варианта в системе. ВХ транслируется в коллекцию вариантов в точ-

ках вариантности в ПС. Для обеспечения вариабельности СПС выполняются следующие шаги (рис. 3.4).

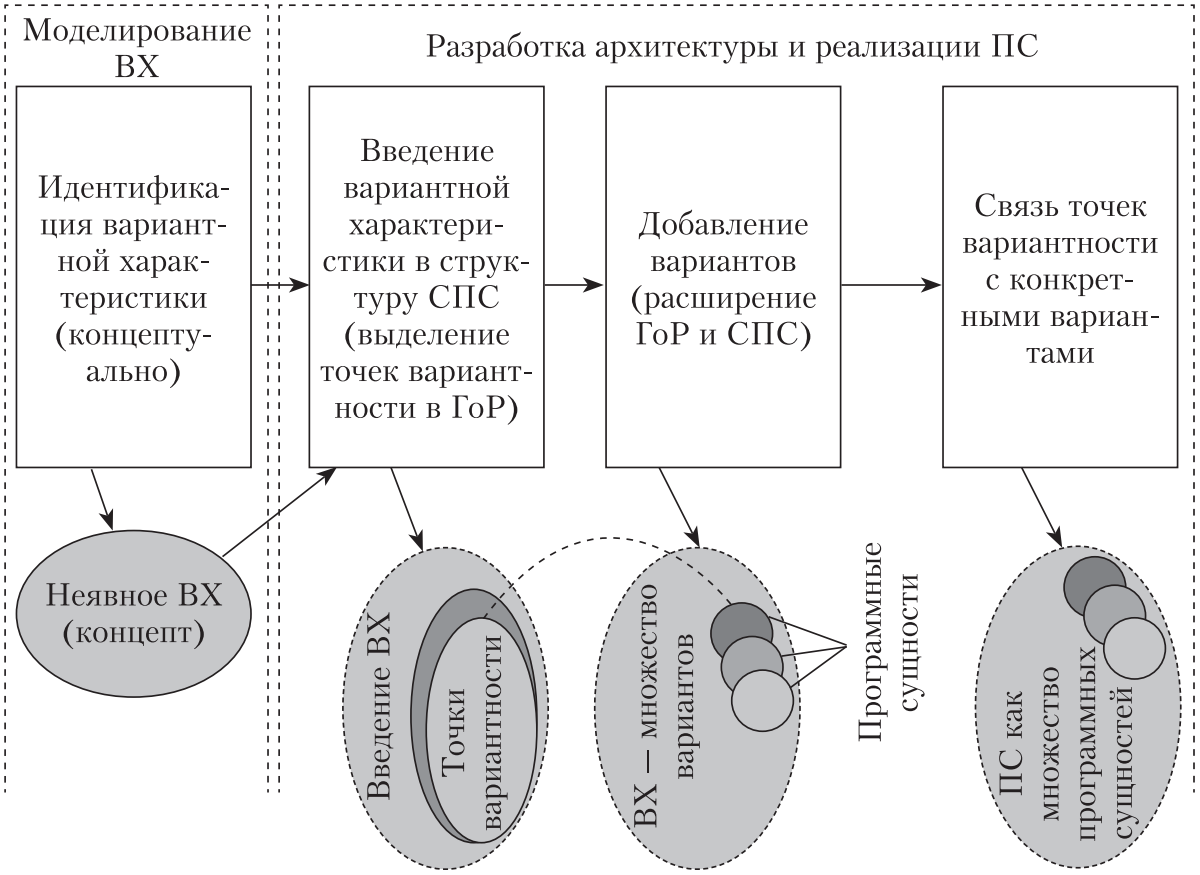


Рис. 3.4. **Процесс обеспечения вариабельности СПС**

1. *Идентификация вариабельности СПС* предусматривает наличие совокупности вариантов и точек вариации. Для ее идентификации применяются методологии FODA, FORM (*Feature-oriented Reuse Method*), RSEB (*Reuse-Driven Software Engineering Business*), Feature RSEB и т.п. На протяжении ЖЦ формируется программа ПС.

2. *Ограничение вариабельности СПП* состоит в обеспечении текущих потребностей и перспектив развития. В СПП вводится характеристика в форме множества точек вариации — мест в архитектуре, способы присоединения ВХ в СПП и реализация КПИ.

3. *Реализация вариабельности* выбирается по технологии реализации точек вариации. Решение относительно варианта ВХ будет использоваться для ПС и одновременно уточняется ссылка на элементы в файле конфигурации.

4. *Управление вариабельностью* состоит в планировании, учете и контроле реализации модели вариабельности, а также действий по регуляции уровня вариабельности СПП.

Таким образом, ЖЦ вариантной характеристики как объекта СПП включает переход состояний: концептуальное представление (неявная ВХ), множество точек вариации в СПС (введенная ВХ), множество вариантов сущностей (ВХ как множество вариантов реализации), конкретный фрагмент ПС, связанный с ВХ.

3.4. Моделирование изменяемых систем по К. Чернецки

Метод генерирующего программирования — ГП (*generative programming*) К. Чернецки[1] определяет *Feature-oriented* модель архитектуры системы в пространстве (области) проблем и пространстве (области) задач домена (рис. 3.5), аналогичный SPLE.

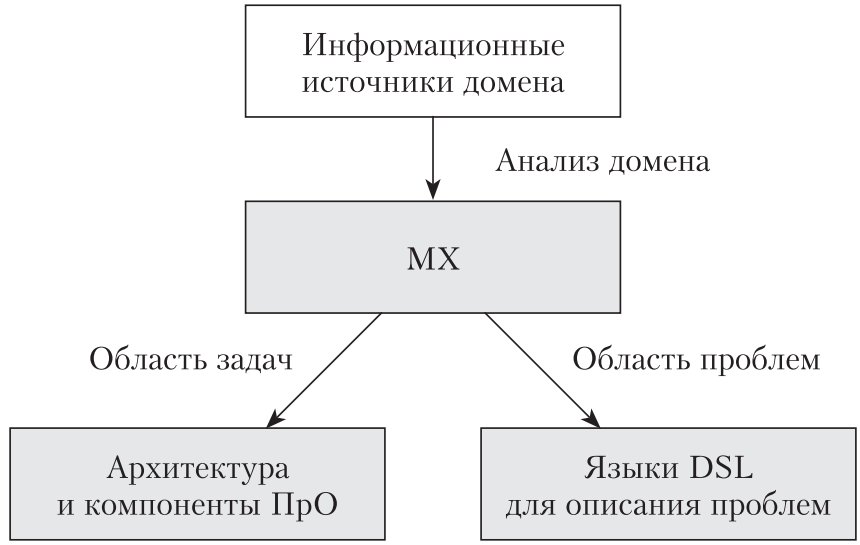


Рис. 3.5. **Структура инженерии домена**

В этом программировании характеристики членов семейства отображаются в модели ВХ как для КПИ, так и для ПС, СПС.

Построение из КПИ ПС с нужными свойствами основывается на решении задач классификации и типизации, а также идентификации архитектуры. Для КПИ должны задаваться спецификации функций и интерфейсов, а также способы хранения, поиска и выбора КПИ. На основе КПИ и ВХ проводится конфигурация ПП.

Из элементов этих областей обеспечивается *автоматизированное* построение новых ПП и СПП.

3.4.1. Основы метода генерации ПП

ГП, как и SPLE, включает два процесса разработки ПС — **инженерия предметной области** (*domain engineering*) и **инженерия приложений** (*application engineering*). Главным элементом программирования в этой парадигме является не только ПП, созданный из КПИ, но и семейство продуктов (*product family*). Элементы семейства генерируются на основе генерирующей модели (Generative Domain Model, GDM) в модели семейства.

3.4.2. Метод генерации доменов и приложений

Инженерия предметной области (доменов) включает:

- анализ домена, включая определение границ домена и его связей с другими доменами и выявление и формальное описание общностей

[1] См.: *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.

и отличительных особенностей внутри домена, — определение постоянных общих требований ко всему семейству программных продуктов как единому целому и специфичных переменных требований к его компонентам (объектам, аспектам), в совокупности покрывающих все требования к ПС;

- классификацию, документирование моделей домена и формирование терминологического словаря для описания основных понятий в домене и взаимосвязей между его активами; оценку моделей и словарей домена в соответствии с выбранной методологией моделирования;
- архитектурное проектирование домена заключается в создании ПП из КПИ. Используются горизонтальная и вертикальная типизация компонентов. К горизонтальным относят общие системные средства: графические пользовательские интерфейсы, СУБД, системные программы, библиотеки матриц, контейнеры, каркасы и т.п.; к вертикальным — медицинские, биологические и др.;
- формирование генерирующей модели домена на базе описанных требований;
- верификацию и тестирование элементов модели.

В архитектуре домена формально определены интерфейсы компонентов, которые фактически служат основой для объединения КПИ в систему, согласованную с моделью FM.

Инженерия приложений включает:

- разработку ПС по спецификации требований на базе КПИ и генерирующей модели домена GDM и FM, в результате чего генерируемая ПС приобретает все общие свойства своей платформы и наделяется специфичными (требуемыми) свойствами;
- разработку тестов и проведение тестирования ПС.

Результат инженерии — рабочий продукт ПС с общими и специфическими свойства. По модели FM и конфигурационной модели осуществляется сборка КПИ. При этом описание артефактов ПС трансформируется в ЯП, а затем в код (рис. 3.6).

При сборке общие характеристики и ограничения преобразуются к понятиям DSL и множеству компонентов из пространства задач и проблем для ПС и СПС.

Описание характеристик и ограничений ВХ приводится к описанию обобщенной архитектуры семейства ПС в языке ADL (*Arhitecture Description Languages*) и предметно-ориентированном языке DSL (*Domain Specific Language*). Трансформация описаний компонентов выполняется средствами ЯП. Конфигурационный файл содержит элементы ПС и их характеристики, необходимые для внесения изменений в ПС и СПС.

Модель приложения или домена в языке DSL (из пространства проблем) *трансформируется* в пространство решений путем генерации спецификаций в DSL в более простой ЯП.

Примером систем поддержки инженерии ПрО являются DEMRAL (библиотека численного анализа, распознавания речи, графовых вычислений и др.) и RSEB для прикладных систем.



Рис. 3.6. **Конфигурационная модель ПрО**

3.5. Применение инженерии SPLE в системе Grid

Анализ системы Grid (Седьмого рамочного проекта Европейского союза, 2004—2012) показал, что в ней подсистема ETICS реализована согласно технологии SPLE[1]. ETICS означает *e*-инфраструктура для тестирования, интеграции и конфигурирования ПО (рис. 3.7).

ETICS — это система управления малыми и большими программами путем конфигурации с учетом стандартов качества, сборка (*building*) пакетов из модулей и их тестирование в этой среде. Она используется для запуска удаленной сборки (*builds*) и тестирования на компьютерах пользователей для разных платформ. Система ETICS обеспечивает автоматизированное построение, конфигурацию, интеграцию и тестирование пользовательского проекта, которому задаются точки входа в веб-портал доступа к технологиям Grid. ETICS предоставляет интеграцию и каркас тестирования для задания синтаксиса, требований и моделей данных.

[1] См.: Grid — Distributed Computing at Scale. An overview of Grid and the Open Grid Forum. GWD-I Mark Linesch, HP. Marketing April 23, 2007. URL: http://www.ogf.org/ Public_Comment_Docs/Documents/Apr.

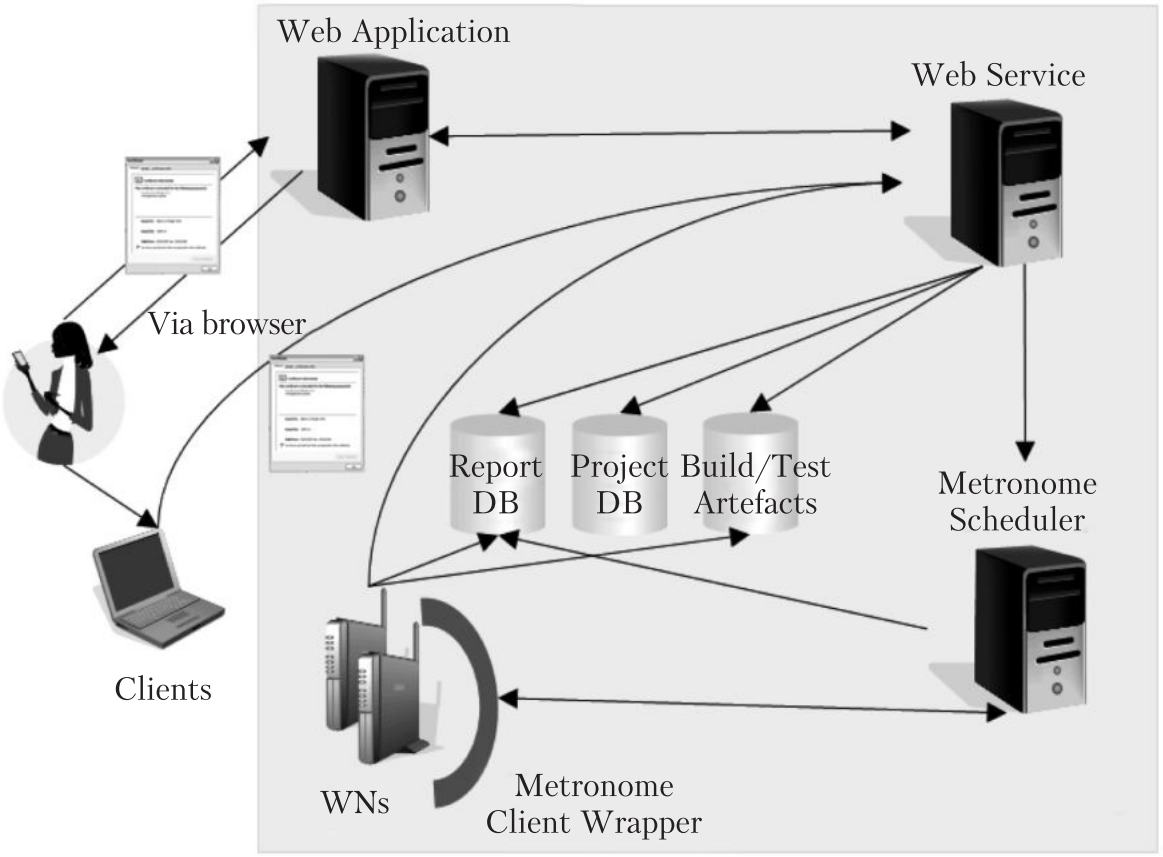


Рис. 3.7. **Инфраструктура Grid**

3.5.1. Средства разработки ПС в ETICS Grid

Динамические области ресурсов и интерфейсов создания и тестирования обеспечивают поддержку запуска построенных и тестированных наборов. Кроме того, все построенные и проверочные операции разработчики могут выполнять на удаленных узлах, чтобы разработать и проверить их на тестах и построить конфигурацию. Проект имеет возможность организовать и запомнить метаданные об их ПО и иметь доступ к информации к другим проектам. Репозиторий стандартных пакетов, документации и информации служит созданию новых конфигураций и тестовых наборов.

Построенный пакет содержит инструкции для работы с большими наборами исходных или комбинированных двоичных элементов из репозитория. При этом используются модель данных и их хранение согласно принципам представления стандартных моделей информации в CIM[1]. Вводятся формальные сущности, чтобы определить структуру ПО, построить конфигурацию с обеспечением информационной безопасности при получении результатов. Структура ПО проекта представляется объектами: *проект*, *подсистема* и *компонент*. Проект содержит подсистемы или компоненты, а подсистема — только компоненты (рис. 3.8).

---

[1] См. об этом: *Бей И.* Взаимодействие разноязыковых программ. Диалектика. М. : Вильямс, 2005; *Гринфильд Дж.* Фабрики разработки программ. Диалектика. М. : Вильямс, 2007.

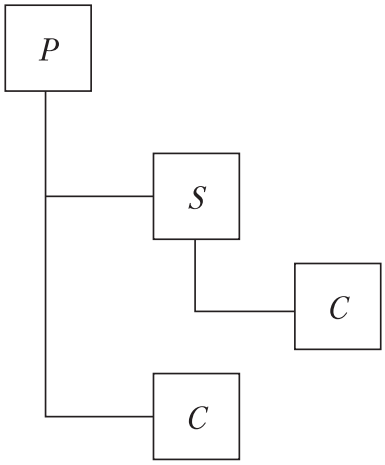


Рис. 3.8. **Связь объектов «проект», «подсистема» и «компонент»**

Три объекта — проект, подсистема, компонент — связаны с модулями и задают абстрактную идентичность проекта, подсистемы или тестового узла физического пакета, который строится или тестируется и выполняется без информации о версии. Каждый объект конфигурации является представителем информации о версии модуля.

Модель данных CIM описывает объекты и отношения между ними. Зависимости между объектами задаются в спецификации конфигурации и используются при проверке одних компонентов относительно других. Конфигурация модуля применяется при построении или тестовом выполнении компонента, который выбирает необходимую информацию из пространства проекта. Этот подход используется при контроле зависимости и обеспечении операций выполнения всего проекта.

Динамическая зависимость — взаимоотношение между названной конфигурацией и модулем (например ‘веб-сервис v. 1.0.0’ зависит от ‘tomcat’). В случае динамических зависимостей точная конфигурация применяется во время построения и ETICS выполняют построение конфигурации, которая использует глобальную информацию проекта (например, мандат проекта, где ‘tomcat v. 5.0.24’ должен применяться для всех компонентов в зависимости от ‘tomcat’) или версию (например ‘tomcat >= 5.0’ определяется как самая высокая и доступная версия в БД ETICS, удовлетворяющей подходам к использованию).

ETICS поддерживает несколько ОС: научный Linux (SLC3, SLC4, SL5), Ubuntu 7, Debian 4, версии Red Hat Linux, CentOS, MacOSX и Windows для 32- и 64-разрядных (ia64 и x64—86) CPUs.

3.5.2. Сборка, тестирование и конфигурация систем в ETICS

Конфигурационная сборка и тестирование в ETICS проходят три фазы.

1. Конфигурирование — это процесс регистрации программных компонентов для сборки и тестирования, создания конфигураций и присоединения команд контроля версий, команд сборки, команд тестирования, свойств, переменных и зависимостей. Функциональные возможности задачи предоставляются веб-приложением с помощью процесса сборки и тестирования (*Build and Test Web Application*) и клиентом. Командная строка (*Command-*

Line Client) обеспечивает доступ к сервису сборки и тестирования (*Build and Test Service*).

2. Локальная сборка/тестирование базируется на собственных компонентах и информации о конфигурации пользователя, зарегистрированного в системе Grid клиентом с помощью командной строки, позволяющей выполнять локальные сборки и тесты на машине пользователя.

3. Удаленная сборка/тестирование. Этот процесс (*engine Metronome*) позволяет запускать сборку и тесты на разных платформах Grid для подтверждения мобильности (*portability*) кода и генерации ряда отчетов о статическом и динамическом тестировании при разных условиях. Программные объекты могут быть следующих типов: «проект» (*P*), «подсистема» (*S*) и «компонент» (*C*).

Конфигурация должна быть присоединена к одному и только одному модулю (проекта, подсистемы, компонента), и каждый модуль может иметь одну или несколько конфигураций, подобно CVS, где модуль всегда имеет главную (HEAD) ветвь, но может иметь много других тегов и ветвей. В ней хранится упорядоченное в иерархию дерево конфигурации, содержащее все объекты ПС в соответствующем дереве модулей. Дерево модулей задает структуру проекта, а дерево конфигурации — отдельные версии проекта как подмножество конфигураций.

Команды CVS используются для управления кодом, сохраненным в системах CVS, Subversion или ClearCase. Команды сборки используются для сборки кода, генерации документации и пакетов, а также публикации артефактов сборки в стандартном формате.

3.6. Фабрики программ и AppFab

Фабрика программ — это интегрированная инфраструктура сборочного производства из готовых КПИ и ГоР новых ПП (систем, семейств систем, ИС, АСУ, АСУТП и др.).

Фабрика включает в себя комплекс системных средств, инструментов и сервисов, а также репозитории и библиотеки для накопления и выбора готовых КПИ. Ядро фабрики — операционная среда и методы изготовления (UML, компонентный, структурный, модульный, сервисный и др.) отдельных КПИ. В среду фабрики вводится набор веб-сервисов и веб-семантики для управления выбором необходимых сервисов из Интернета и их использования как КПИ при сборке и композировании новых ПП. К ним относятся:

- фабрика «ручной» сборки разноязычных программ И. Бея;
- фабрика потоковой сборки по UML Дж. Гринфильда;
- фабрика Г. Ленца на основе Use Case;
- каркас фабрики программ А. М. Авдошина;
- фабрика «Continiuos Integration» М. Фаулера;
- фабрика программ EPAM и др.

Фабрика сборки И. Бея. Базис этой фабрики — разноязычные программы, интерфейсные посредники, конфигурационные файлы. Программы описываются в ЯП (VC++, VBasic, Matlab, Java, Visual Works Smalltalk

и т.п.) и выполняются на платформах (MS.Net, HP, Apple, IBM и др.). В интерфейсном посреднике данные передаются друг другу. Для некоторых данных, типы которых нерелевантны и (или) их форматы зависят от другой платформы передачи данных (протоколы, вызовы, интерфейсные карты MIO-16E-2 и др.), проводилось их ручное преобразование. Описание интерфейсных связей проводилось с помощью RMI, RPC, Java Native Interface. Затем они реализовывались в виде *exe*-файлов. Принцип взаимодействия разноязычных программ апробирован на Domain, Application Models, Model Interconnection, Microsoft Communication Foundation и др.[1]

Фабрика программ Дж. Гринфильда. Для любой строящейся фабрики автором сформулированы аспекты производства ПП на основе UML с использованием моделей архитектур систем и компьютеров, механизмов интеграции разнородных компонентов с типами данных FDT и описанием интерфейса в языках (IDL, XML, RDF и т.п.). Главные объекты производства: *reuse*, которые накоплены в общепринятых хранилищах (библиотеках, репозиториях и т.п.) Интернета и имеют сертификаты качества; активы (assets), программы, приложения и системы; *модели*, шаблоны и инструменты UML для реализации ПП на линиях производства; *веб-сервисы*, процессы линий; методики измерения и контроля качества ПП и т.п. Анализ моделирования доменов в UML показал, что UML удобен для задания эскиза ПП, в котором используются компоненты и интерфейсы[2].

Фабрика ПО Г. Ленца. Ключевые элементы этой фабрики — схемы и шаблоны. Схема Software Factory — это линия реализации продуктов, которые могут быть произведены на этой фабрике. Технология базируется на модели MDD и поддерживается архитектурным фреймворком. Типичным примером шаблона является завод, обеспечивающий разработку проектов из многократно используемых компонентов в Visual Studio при реализации проектных решений и требований. Эти требования и решения описываются на языке DSL и используются на линии производства ПП[3].

Фабрики А. М. Авдошина. Основной ресурс и активы данной фабрики — архитектурный каркас, являющийся отправной точкой в разработке любого продукта на линии. В нем используются активы и ресурсы: классы, компоненты, конфигурации, образцы, проектные решения и т.п. Активы выбираются на стадии разработки линии фабрики. База знаний фабрики включает в себя разные пособия, справочники, статьи, примеры программ и программы-образцы. Моделирование понятий и основных концепций в языке DSL способствует генерации кода и конфигурации структуры продукта. Для автоматизации разработки ПП задается схема фабрики и уточняется набор ресурсов[4].

---

[1] См.: *Бей И.* Взаимодействие разноязыковых программ. Руководство программиста. М., 2005. С. 868.

[2] См.: *Гринфильд Дж.* Фабрики разработки программ. Диалектика. М. ; СПб. ; Киев, 2005. С. 285.

[3] См.: *Lenz G., Wienands C.* Practical Software Factories in .NET. From theory in practice — a primer reference and case study. Apress, 2007.

[4] См.: *Авдошин С. М., Белкин С. А.* Фабрики программного обеспечения // Новые информационные технологии. М. : Изд-во МИЭМ, 2007. С. 23—39.

Фабрика программ «Continiuos Integration» (непрерывная интеграция, повторная сборка) М. Фаулера (M. Fowler, K. Beck). Это автоматизированный процесс, обеспечивающий разработку, поиск изменений в коде и системе, сборку, контроль собранных версий, развертывание и тестирование приложения. На фабрике используется гибкие (*agile*) методологии, включая модульное (*unit*) тестирование, рефакторинг, стандарт кодирования. Каждое изменение в систему требует его интеграции на специальной машине, в среде которой проект будет развернут (*production enviroment*). Процесс интеграции выполняется на сервере интеграции (*continuous integration server*) на Java и CruiseControl.NET. Процесс интеграции включает: **Build** — компиляция исходных кодов и их сборка средствами Ant, Maven в Java и NAnt в .NET; **UnitTest** — тестирование модулей приложения; **Deploy** — развертка для выполнения после Build и UnitTest[1].

Фабрика EPAM предоставляет широкий спектр технологий и решений, процессов, сертифицированных по стандартам качества CMMI Level 4 и ISO 9001, а также инструментов управления проектами (http://www.epam-group.ru/#sthash.zPWS iln4. dpuf), аутсорсинга, внедрения, интеграции, тестирования и сопровождении. В СНГ создано более 15 филиалов.

3.6.1. Системные AppFab

В каждой ОС массового применения созданы фабрики программ. В библиотеках системы содержатся наборы документов, заготовок, КПИ, наборы данных и др. Они используются для сборки и интеграции из них приложений и систем для разных целей, включая управление предприятиями, большими данными и др. К фабрикам системного типа относятся:

1) фабрика ПО (Fabrics Software) для систем и сервисов;
2) AppFab в системе коллективной разработки VS.Net;
3) AppFab в системе Grid Европейского проекта;
4) AppFab IBM для создания доменов и бизнес-приложений;
5) AppFab CORBA для сборки разнородных ГоР;

Далее остановимся на краткой характеристике AppFab.

Разработана группой P&P и включает коллекцию ресурсов, блоков кода, образцов приложений и т.п. В состав фабрики вошли принятые схемы и стандарты, рекомендации, инструменты, шаблоны решений и запуска новых приложений.

Архитекторы модифицируют требуемые коды сервисных ресурсов модели SOA Visual Studio Industry Partners (VSIP) и устанавливают их на фабрике ПО. Используются модели Guidance Automation eXtensions (GAX) и Guidance Automation Toolkit (GAT) в Visual Studio. GAX — это среда исполнения в VSIP с использованием пакетов рекомендаций. Существует два варианта служб: веб-служба ASP.NET (ASMX) для Windows Communication Foundation (WCF) в среде .NET Framework 3.0. Версии для веб-службы WCF находятся в сообществе разработчиков фабрики ПО и GotDotN. В ее состав входят процессы предприятия, договоры и пакет

---

[1] См.: *Duval P.*, *Matyas S.*, *Grover A.* Continuous Integration: Improving Software Quality and Reducing Risk. Addison Wesley, 2009.

документаций и рекомендаций для приложения-образца Global Bank. Это аналогично модели SCA в IBM WebSphere.

3.6.2. Фабрики интеграции разнородных компонентов и данных

К ним относятся IBM WebSphere, Microsoft Biz Talk 2004, BEA WebLogic Oracle 10g, SAP NetWeaver, ИВК «Юпитер» (табл. 3.1).

Таблица 3.1

CASE-системы интеграции ресурсов

Платформа	Разработчик	Содержание платформы
IBM WebSphere	Корпорация IBM	Сервер приложений J2EE, брокеры обмена данными, КПИ, портал, workflow/BPM, средства EII, SCA
Microsoft Biz Talk 2004 и компоненты .Net	Корпорация Microsoft	Сервер приложений COM, брокеры обмена данными, RGB доставки, портал, workflow/BPMN
BEA WebLogic	Корпорация BEA Systems (в 2008 г. присоединенная к корпорации Oracle)	Сервер приложений J2EE, брокеры обмена данными, GoP, сервер прикладных приложений, портал, workflow/BPMN
Oracle 10g	Корпорация Oracle http://www.oracle.com	Сервер приложений J2EE, брокеры обмена данными, GoP, портал, workflow/BPM, средства EII
SAP NetWeaver	Корпорация SAP http://www1.sap.com/www.sap.ru	Сервер приложений J2EE/ABAP, брокеры обмена данными, портал, инструменты BPMN
ИВК «Юпитер»	Компания ИВК (Россия) http://www.ivk.ru/	Брокеры обмена данными, КПИ, среда выполнения, сертификация, защита данных

Подробное описание этих фабрик приведено в статье автора[1]. Таким образом, из описания современных платформ индустрии приложений следует, что имеются фабрики AppFab с базовыми средствами создания и выбора сервисов для Grid и Cloud Computing.

Контрольные вопросы и задания

1. Назовите авторов теории вариантности или вариабельности.
2. Что такое Product Line и Product Family?
3. Что такое инженерия доменов и приложений?
4. Дайте характеристику и назовите свойства компонентов и системы.
5. Определите шаги технологии в системе SPLE.

---

[1] См.: *Лаврищева Е. М.* Компонентная теория и коллекция технологий разработки индустриальных приложений из готовых GoP // Труды 4-й науч. конференции «Актуальные проблемы системной и программной инженерии», АПСПИ — 2015, 20—21 мая 2015 г. С. 101—119.

6. Определите сущность метода генерации К. Чернецки.
7. Что такое система Grid и каково ее назначение?
8. Какие средства Grid предоставляет пользователям?
9. Определите модели данных в Grid.
10. Определите назначение фабрики программ и приведите пример.
11. В чем сущность системных фабрик AppFab?

Глава 4
МОДЕЛИ И МЕТОДЫ ПРОЕКТИРОВАНИЯ ВАРИАНТОВ СИСТЕМ

Определение модели вариантных систем из КПИ и других ГоР начинается с задания требований к семейству систем СПС и их способности изменяться в заданном контексте. Требования задают согласованные модели вариантов и вариабельности FM в структуре и артефактах СПС. При разработке СПС для оценки качества ПС могут использоваться сети Байеса, дерево ценности и др. Предложена новая концепция поддержки требований к функциям управления вариантами и конфигурирования артефактами систем и семейств ПС.

Главным элементом ПС является ГоР, КПИ или просто программный компонент как самостоятельно реализованный элемент, обеспечивающий выполнение определенной совокупности прикладных сервисов, доступ к которым возможен только с помощью интерфейсов, определяющих функциональные возможности продукта и порядок обращения к ним.

КПИ, как правило, отображает некоторое типовое решение, имеет типовую архитектуру, характеристики и атрибуты (параметры) в его интерфейсной части для обмена данными и взаимодействия КПИ друг с другом в разных средах. По существу КПИ становится программным элементом ПП, удовлетворяющим функциональным требованиям к архитектуре ПС и среде взаимодействия.

4.1. Определение вариантов КПИ в ПС и СПС

Готовые КПИ могут быть обязательными, т.е. присутствуют во всех ПС данной СПС; необязательными, т.е. присутствуют лишь в некоторых ПС или существуют в нескольких вариантах; индивидуальными — создаются для заказчика и не планируются использоваться в других ПС.

Компонент повторного использования (КПИ)[1] — это готовый программный компонент (проектное решение, функция, шаблон и др.), который используется в ходе разработки не только самими разработчиками, но и другими пользователями.

Готовые КПИ помещаются в библиотеках среды создания ПС.

Тип компонента или КПИ может изменяться. Например, индивидуальные могут получить статус необязательных, а необязательные статус обязательных. В описании КПИ могут задаваться точками. Будем их назы-

[1] *Липаев В. В., Штрик А. А., Позин Б. А.* Технология сборочного программирования. 1992.

вать *точками вариантности*, в случае если может существовать отличие функциональности в отношении другого ПС и если они задают точку для возможности изменения некоторой функций КПИ или ПС.

Вариант — это отдельная функциональность, которая отмечается точкой вариантности. В конкретном случае вариантами могут отмечаться КПИ обязательного и необязательного типов. Для конкретной точки вариантности задается зависимость в виде идентификатора ассортимента или альтернативы варианта.

С помощью ограничений (взаимоисключающих или взаимодополняющих) обеспечивается использование одного варианта с другим. То есть закладывается проверка на взаимоисключающее использование КПИ в одной ПС или их зависимость друг от друга.

4.1.1. Определение КПИ и модели вариантности ПС и СПС

ПС и СПС имеют общее множество характеристик, соответствующих потребностям определенных функциональных вариантов домена, различаются способами воплощения этих характеристик в отдельные ПС и разрабатываются с использованием GoP или КПИ. Любая ПС создается не с нуля, а порождается моделью СПС путем выбора готовых КПИ из репозиториев или библиотек. Выбранный КПИ адаптируется к среде и методом сборки конфигурируется в новую структуру ПС.

Модель СПС, связанная с объектной и компонентной моделью, дополняется точками вариантности. В результате получается расширенная модель семейства вариантной системы (СВС).

С помощью ОКМ проектирования ПС строятся взаимосогласованные составные элементы модельной среды: объектные и компонентные модели ПС и артефакты СПС. Для СВС проводится их конкретизация. При проектировании объектной модели (ОМ) осуществляется последовательное представление объектов на обобщающем ($t = 1$), структурном ($t = 2$), характеристическом ($t = 3$) и поведенческом ($t = 4$) уровнях абстракции.

4.1.2. Введение вариантных характеристик в архитектуру ПС

Вариантные характеристики (ВХ) могут быть введены на всех уровнях проектирования ОМ системы — от архитектурного и характеристического вплоть до реализации объектов к компонентам. Эти уровни различаются операционными программными сущностями. Одна ВХ может отражаться на множестве разнотипных программных сущностей, которые вместе обеспечивают желаемую функциональность. Она проявляет себя в СПС множеством точек вариантности на разных уровнях абстракции ОМ.

Решения относительно введения ВХ принимаются с учетом:

- доступности (готовности) к реализации, удовлетворяющей потребностям связи объектов и добавления вариантов;
- размеров программных сущностей и артефактов, которые касаются введенных характеристик ВХ;
- количества точек вариации;
- стоимости сопровождения введенных точек вариации.

Если точка вариации введена на ранних стадиях ЖЦ ВХ, она попадает под контроль на всех следующих этапах разработки, в другом случае — она контролируется лишь в течение непродолжительного времени.

Если точка вариации открыта для добавления новых вариантов к их коллекции в этой точке, то в любой момент времени варианты могут быть добавлены или изъяты. Если точка вариации закрыта, это означает, что коллекция вариантов зафиксирована и не может изменяться.

Решение относительно того, когда и как добавлять варианты, принимает определенный разработчик системы, учитывая бизнес-стратегию и модель производства ПС на определенной линии по модели вариабельности.

Если бизнес-стратегия предусматривает возможность позднего добавления вариантов, например, посторонними поставщиками, это ограничивает выбор технологий реализации точек вариации, поскольку может возникнуть потребность оставлять их (точки) открытыми для добавления новых вариантов после компиляции или даже в период выполнения ПС.

На процесс добавления вариантов влияют процесс разработки ПС и используемые инструменты. При этом добавление вариантов может выполняться двумя путями, в зависимости от того, как реализована точка вариации коллекции вариантов в ПС. Варианты могут быть добавлены неявно, и это означает, что не существует представления. Коллекцией руководят специалисты вне разрабатываемой системы, например с использованием простого перечня имеющихся вариантов. В этом случае обеспечение требуемого варианта из коллекции неявных вариантов возлагается на разработчиков или пользователей. При явном добавлении вариантов коллекция вариантов присутствует в составе кода ПС. Это означает, что в системе достаточно информации, для того чтобы при необходимости самостоятельно найти необходимый вариант.

4.1.3. Связь варианта с точками вариации

Главная цель введения вариантных характеристик — отнесение на последующие стадии разработки ПС решения относительно конкретной характеристики, которую нужно реализовать. Конкретизация СПС до необходимого варианта определяется связью варианта с неизменной составляющей ПС в точке вариации.

Различают внутреннюю и внешнюю связь. При *внутренней связи* ПС включает функциональность, ассоциируемую со связываемым вариантом. Такая связь имеет место в период выполнения. *Внешняя связь* предусматривает, что существует лицо или инструмент, фактически выполняющие связь.

Решение относительно связи — внутренней или внешней обусловливается тем, выполняют ее разработчики ПС или конечные пользователи (например, настройка источника ODBC), а также тем, что оно должно быть прозрачным для пользователей. Внешняя связь имеет преимущества, поскольку она оставляет отметку в начальном коде, т.е. ПС должна содержать функциональность для связи. Таким образом, внешняя связь способствует снижению сложности исходного и выходного кодов системы.

При добавлении вариантов время связи ограничивает выбор возможных путей реализации точки вариации. ВХ вводится через множество точек вариации, которые должны связываться одновременно (как в случае связи в период выполнения) либо синхронизированно (когда, например, точка вариации, связываемая при компиляции, ассоциируется с вариантом, который был связан в точках вариации на уровне архитектуры).

При принятии решений относительно времени связи нужно руководствоваться правилом: чем позже выполняется связь, тем она больше стоит. Откладывание связи во время построения архитектуры на компиляцию предопределяет необходимость для разработчиков не только управлять всеми вариантами во время реализации, но и включать функциональность связи в выходной код для эффективного выполнения связи.

4.1.4. Модель КПИ с вариантами и механизмами вариации

Рекомендации относительно построения внешних спецификаций повторно используемых информационных ресурсов предоставил консорциум OMG в документе «Reusable Asset Specification (RAS)». Назначение этих спецификаций — упростить процедуру объединения брокером информации о КПИ для ее распространения, поиска и использования разными категориями пользователей.

Для поддержки механизмов реализации вариабельности КПИ в СПС предлагается модель структуры КПИ с вариантами (рис. 4.1).

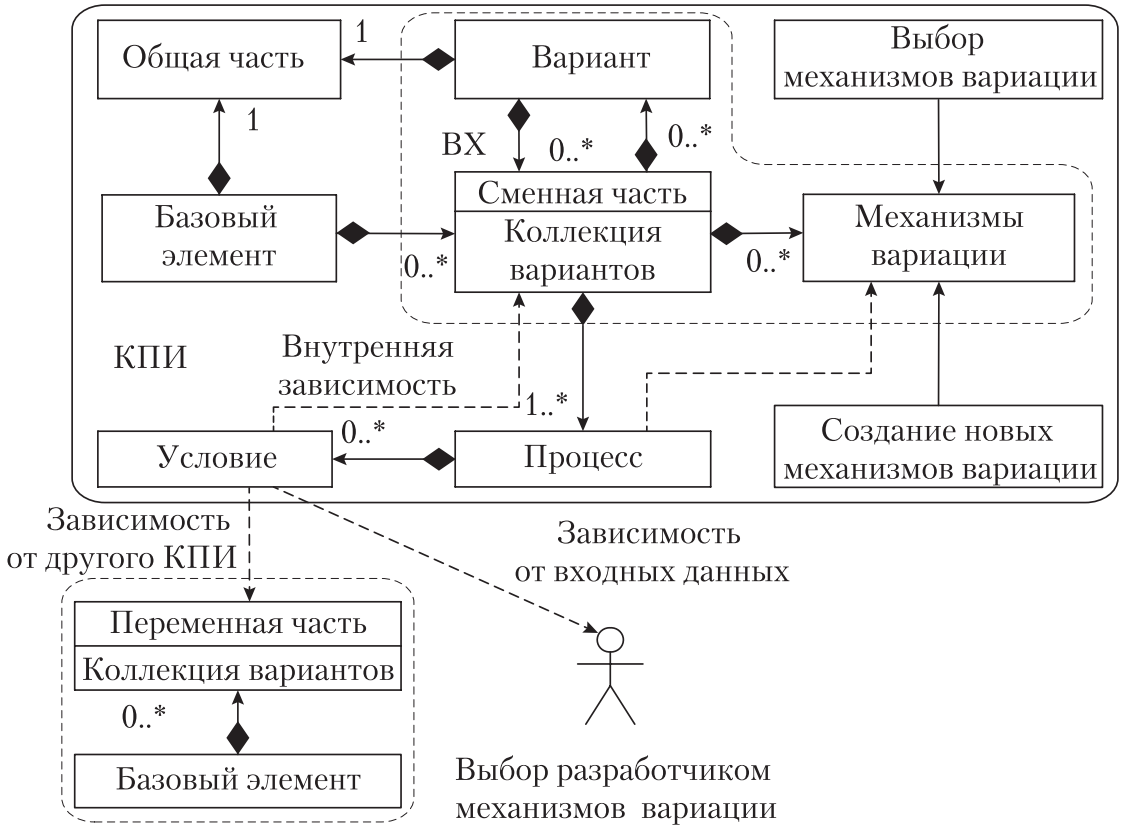


Рис. 4.1. **Модель структуры КПИ**

Эта модель представлена диаграммой классов в ООП, но это не означает, что все КПИ системы должны быть написаны в этом подходе. На данной

диаграмме базовый элемент является основным классом, который представляет КПИ и взаимодействует с ПС. Другими словами, объектом класса является базовый элемент, который и есть КПИ. Базовый элемент включает в себя общую часть и (необязательно) переменную часть. Если базовый элемент не имеет переменной части, его используют непосредственно при включению в конкретную ПС. Переменная часть — это локализация изменений в конкретной точке КПИ. Отсутствие общей части означает, что все КПИ должны быть разработаны для каждой ПС индивидуально и это не противоречит концепции КПИ.

Последующие объяснения относительно внутренней структуры КПИ, а также механизмов обеспечения его вариабельности ВХ будем сопровождать примером, в котором в качестве КПИ будут выступать элементы управления графического интерфейса пользователя (GUI), которые своими внешним видом и содержанием выполняемых действий могут адаптироваться к требованиям пользователя. Следовательно, для элемента управления общей частью может выступать механизм графического вывода на экран и для всех вариантов элементов управления он будет одинаковым. Переменную часть будет представлять набор параметров, которые модифицируют поведение и вид элемента управления.

Каждая переменная часть КПИ может содержать несколько механизмов вариации, которые поддерживают создание или выбор вариантов. Механизм вариации может быть интегрирован непосредственно в базовый элемент. Примером такой интеграции является способ изменения параметров элемента управления (размер, цвет и т.п.). Переменная часть задает точку вариации, а механизмы вариации выступают как средство конкретизации КПИ.

Во многих случаях переменная часть включает два механизма: один для создания нового варианта, а другой — для поддержки существующего, который выбирается из репозитория модельной среды.

Механизм вариантности включает все необходимое для внесения любых изменений. В случае его отсутствия соответствующие КПИ не будут широко использоваться в СПС, а это будет свидетельствовать о том, что действия процесса разработки КПИ и, соответственно, ПС с использованием КПИ должны быть скоординированы. В примере элемент управления механизма вариантности предоставляет возможность разработчикам ПС изменять его свойства, а разработчикам КПИ — добавлять новые.

Результатом применения механизма вариантности к переменной части КПИ является *вариант*, который содержится в общей части и может быть включен в переменную часть. Обычно появление нового варианта предопределяется применением нового механизма вариантности, например, при добавлении нового механизма связывания (*building*) элемента управления типа список. Получается новый вариант поведения базового элемента. Элемент управления должен работать во всех ПС, использующих этот механизм. Созданный вариант для одной ПС может быть использован в других ПС путем выбора механизма вариантности.

Для поддержки механизмов вариантности при построении ПС в СПС предлагается в описание переменной части включать информацию об использовании условий ее адаптации к потребностям ПС. Процесс при-

менения ПС будет объяснять, как использовать механизмы вариантности при создании или выборе варианта ПС.

Описание процесса может содержать условия использования. Если условий нет, то он является безусловным и выполняется каждый раз, когда создается новый вариант ПС из базовых компонентов. Условия использования переменной части определяют зависимость от коллекции вариантов, которые влияют на конкретный продукт. Коллекция может быть дополнена вариантами из другой переменной части или задается разработчиками ПС и, таким образом, будет отображать альтернативу, которая избрана для решения проблемы в конкретной ПС.

Возможны следующие зависимости переменных частей: внутренняя зависимость — это зависимость одной переменной части КПИ от его другой переменной части, внешняя зависимость — это зависимость от переменной части базового КПИ.

4.1.5. Описания внешних и внутренних связей вариантов

Связи вариантов КПИ могут быть внешние и внутренние.

Внешние связи вариантов

Для организации внешних связей вариантов могут использоваться такие механизмы вариации, как макросы, условная компиляция кода, конфигурирование кода, генерация кода статических библиотек и динамически присоединенных библиотек. Рассмотрим их более детально.

1. **Макрос.** Это программный объект, который при обработке «разворачивается» в последовательность действий или команд. Макросы полностью или частично поддерживаются в Ada, C, C++, D, Erlang, Haskell, Common, Lisp, Nemerle, Ruby и др. Функциональность макроса разделена на две части. Общая часть находится в теле определения макроса, а варьирующиеся — в разных частях кода и содержат варьирующиеся параметры, как в примере ниже:

```
// Macros.h Macros предупреждение
#define Реестр Предупреждение(сообщение)
cout << " Предупреждение в строчке: " <<
__LINE__ << ": " << (сообщение) << endl; // Любой класс в ПС. *.
cpp
#include "Macros.h" // Включаем макрос.
Любой Класс::Метод()
{Реестр Предупрежден("Сообщение");};
```

2. **Условная компиляция кода.** Этот механизм вариации обеспечивает возможность контролировать включение или исключение сегмента собранного кода с применением параметров компиляции.

Пример:

```
string s;
#ifdef WIN32
s = .;
#ifdef VXWORKS
s = .;
```

```
#endif
Return s;
```

Здесь переменная часть представлена как результат вариации, сохраненный переменной *s*.

3. **Конфигурирование.** Исходный код каждого варианта находится в отдельном файле, а для выбора альтернатив используется инструмент управления конфигурацией.

4. **Генерация.** Генератор конвертирует заданную спецификацию, написанную на языке ПрО (DSL, например) или ЯП, который генерируется в выходной код. При этом исходный код, в свою очередь, может быть использован как часть продукта. Например, графический интерфейс пользователя может быть сгенерирован (*Silverlight*) из графической или текстовой спецификации.

5. **Организация статических библиотек.** Статические библиотеки содержат набор внешних функций, которые можно использовать в приложении после его компиляции. При этом код из библиотеки интегрируется в выполняемый файл. Вариабельность достигается путем выбора соответствующей библиотеки перед ее присоединением. В Unix-системах, например, библиотеки в основном располагаются в /lib /usr/lib/usr/local/lib, а в Windows-системах — в \System32 и имеют тип «ocx» или «lib».

6. **Динамически присоединяемые библиотеки.** Данный механизм вариантности основан на использовании DLL-библиотек, которые можно поместить в адресное пространство программы во время ее работы. Управление вариабельностью достигается путем реализации разных вариантов кода в отдельных библиотеках и написания исходного кода, который загружает определенные варианты в приложение во время его работы. Следовательно, знать все возможные варианты во время разработки программы не обязательно. В Unix-системах, например, они называются совместно используемыми объектами (*shared objects*) и имеют расширение «So». Mac OS X унаследовали от BSD работу с библиотеками, они находятся в директориях «bundles» и имеют тип «dylib».

Внутренние связи вариантов

При описании механизмов, предназначенных для внутренней связи, будем считать, что ПС создается в парадигме ООП, где каждый компонент ПС представлен классом с собственными данными и методами их обработки, а следовательно, все варианты, с которыми пользователь имеет дело, при реализации представляют собой не что иное, как простые манипуляции классами или другими конструкциями используемого ЯП.

Для реализация вариабельности ПС может быть выполнена операция: композиции, наследования, параметризации, перегрузки операторов и методов, динамической загрузки классов, рефлексии (отображения), шаблонов (паттернов), архитектуры и проектирования. Кратко охарактеризуем каждую из них.

1. **Композиция.** Механизм вариабельности устанавливается на основе связи между объектами типа «part» и размещения общей части в делегирующем классе, а переменной — в делегированном. На рис. 4.2 представлены UML-схема и соответствующий фрагмент кода, который описывает пове-

дение системы в случае изменения расположения панели с элементами управления в окне.

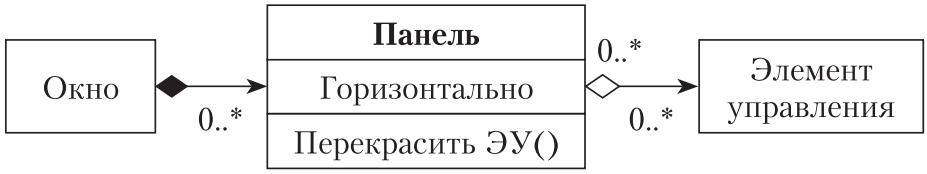


Рис. 4.2. **Композиция**

Окно не содержит логики относительно того, как именно отображать и группировать разные элементы управления. Окно делегирует эту функцию панели. Команда панель.Горизонтально = true является общей частью, которая выполняется одинаково в независимости от типа панели и элемента управления (ЭУ). Переменная часть находится в панели переписанного ЭУ и содержит соответствующую логику расположения и отображения элемента для разных ЭУ.

```
class Окно {
List<Панель> панели = new List<Панель>();
public Окно () {
foreach (Панель панель in панели) { панель.Горизонтально =
true;}
}
}
class Панель {
List<ЭлементУправления> ЭлементыУправления = new List
<ЭлементУправления>();
public bool Горизонтально {
get { return горизонтально; }
set {
горизонтально = value;
Перерисовать ЭУ();}
}
private bool Горизонтально;
private void Переписать ЭУ(){/*...*/}}
class ЭлементУправления {/*...*/
```

В данном примере общей частью является интерфейс работы с классом Панель и свойством Горизонтально. Перерисовываются ЭУ, которые содержат механизм вариации. Эти компоненты реализованы во всех других порожденных типах Панелей и ЭУ соответственно, а, следовательно, переменной частью является содержание каждого из этих методов и свойств. Вариантами являются объекты, которые сохраняются в коллекциях: List<ЭлементУправления> ЭлементыУправления и List<Панель> панели.

На данном примере связь между классами Панель и ЭУ является примером агрегации объектов — панель может содержать несколько ЭУ; связь между классами Окно и Панель — примеры композиции объектов, где к отношению «part-of» добавляется условие, что панель принадлежит только одному окну и уничтожается вместе с ним. Обобщением обоих отношений (агрегации и композиции) выступает ассоциация.

2. **Наследование.** Механизм базируется на таком принципе ООП, как наследование через установление отношений «наследник — предок» с возможностью порождать один класс объектов от другого с сохранением всех свойств и методов класса-предка, а следовательно, и руководить объектами, не имея полной информации о них.

Этот механизм разделяет общую функциональность в базовом классе и дополненную в классе наследника. Наследование имеет следующие виды категорий.

Множественное наследование (рис. 4.3). При таком механизме реализации вариабельности новый класс наследуется от нескольких базовых.

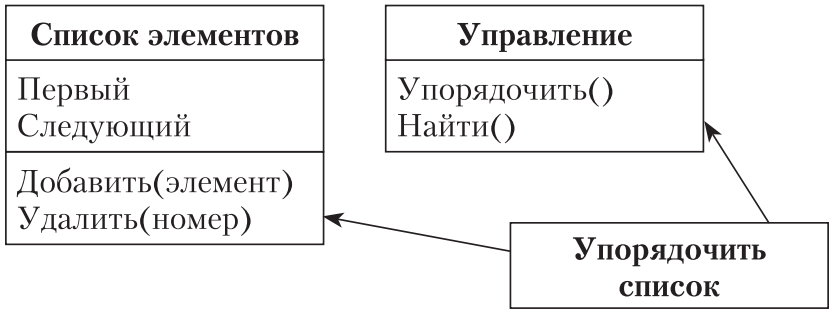


Рис. 4.3. **Множественное наследование**

Стандартное наследование (рис. 4.4). Предусматривает создание иерархической структуры кода, который содержит более общую функциональность на верхнем уровне (в базовом классе — ЭУ) для последующего использования ее на нижнем уровне, дополненном варьирующейся функциональностью (классы-наследники). Таким образом, классы становятся более специализированными.

Наследование с виртуальными методами. Механизм подобен стандартному наследованию, но методы базового класса могут быть загружены классом-наследником.

На данном рисунке класс управления содержит алгоритмы сортировки и поиска и не зависит от элементов, над которыми выполняется операция. Такой тип наследования поддерживается в ЯП: Eiffel, C, Dylan, Python, Perl, Curl, Common Lisp (через CLOS), OCaml.

Наследование, основанное на смесях. Смесь (mix) — элемент ЯП (абстрактный подкласс или модуль), который реализует вариант поведения родительского класса или модуля. При реализации вариабельности он используется для уточнения поведения других классов и не предназначен для порождения самостоятельно используемых объектов ЯП, которые поддерживают смеси, — это: Flavors, Smalltalk, Beta, CLOS, Ruby, D, Python, Scala. На рис. 4.5 изображена UML-схема реализации наследования со смесью.

На данном рисунке интерфейс потребности описывает через методы Смесь и получает информацию от предка. Любой класс, в котором будет использоваться Смесь, должен реализовывать методы интерфейса Услуги. Классы Алгоритмы1 и Алгоритмы2 в данном случае являются примесями. В классе упорядочен Список смешивания обоих классов. Ниже представлен фрагмент программы для реализации механизма вариации типа Смесь:

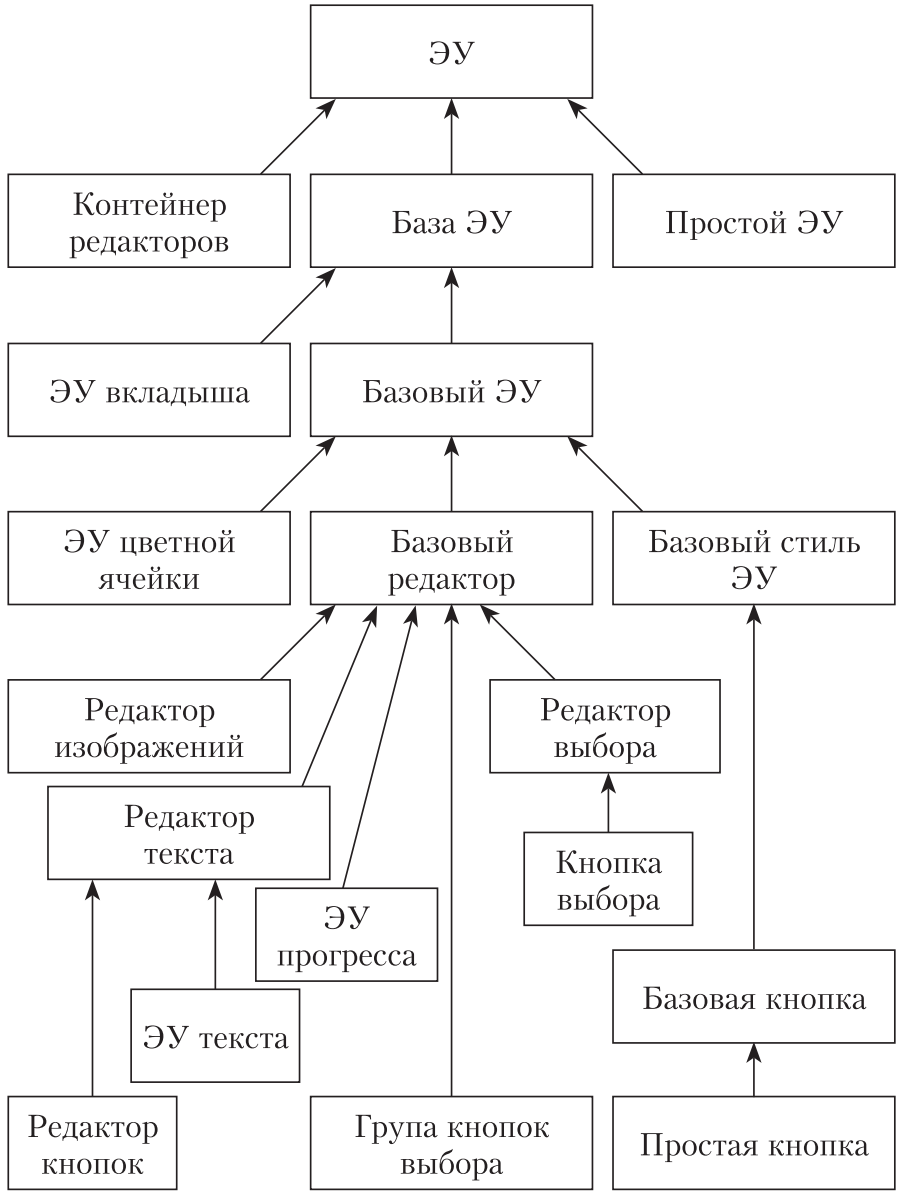


Рис. 4.4. **Пример стандартного наследования классов**

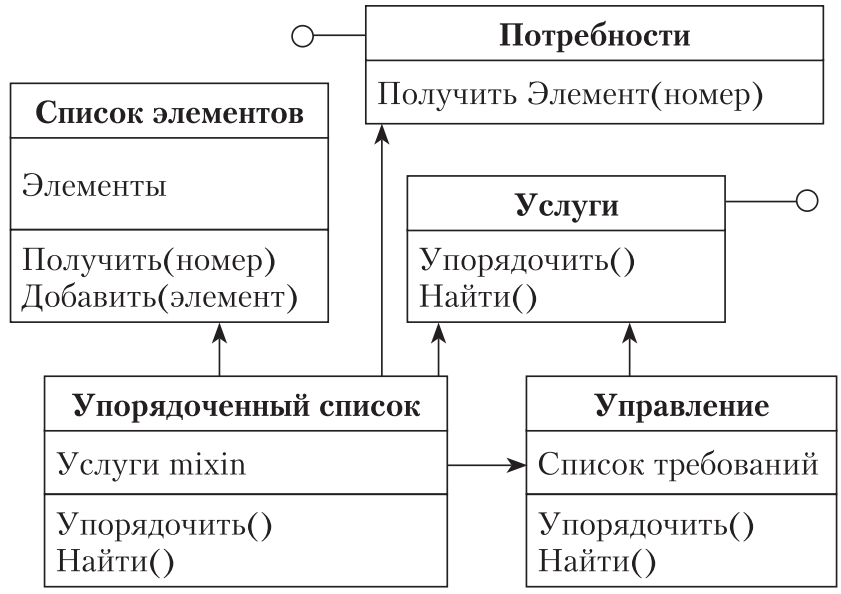


Рис. 4.5. **Имитация множественного наследования**

```
interface Услуги
{
void Упорядочить();
object Найти(/*элемент*/);
}
interface Потребность { object Получение Элемент(int номер);}
class Алгоритмы1: Услуги
{
private readonly Потребности список;
public Алгоритмы1(Потребности список) { this.список = список; }
public void Упорядочить()
{
/*Алгоритм упорядоченности списка, который находится в parent*/
}
public object Найти(/*элемент*/) {/*Алгоритм поиска в список.
ПолучитьЭлементы();*/}
}
class Алгоритмы2: Услуги
{
private readonly Потребности список;
public Алгоритмы2(Потребность список) {this.список = список; }
public void Упорядочить () { }
public object Найти(/*элемент*/) {/*...*/}
}
class СписокЭлементов
{
private object[] элементы;
public СписокЭлементов() {элементы = new object[10];}
public object Получить Элемент(int номер) {return элементы[номер];
}
public void ДобавитьЭлемент(object элемент) {/*ДобавитьЭлемент
*/ }
}
class Упорядоченный Список : СписокЭлементов, Потребности,
Услуги
{
private readonly Услуги mixin;
public Упорядочить Список()
base() {
if(/*ограничена память*/){this.mixin = new Алгоритмы1(this);
} else if (/*свободный процессор*/)
{this.mixin = new Алгоритмы2(this);}
}
public void Упорядочить() {mixin.Упорядочить();}
public object найти (/*элемент*/) {object элемент = mixin.Найти
(/*элемент*/); }
}
```

В этом фрагменте точка вариации представлена в коде командой this.mixin = new Алгоритмы2(this).

Если возникнет необходимость изменить алгоритм сортировки или поиска, который реализован в классе Алгоритмы1, достаточно изменить

Алгоритмы1 на другой класс, унаследованный от интерфейса Услуги. Переменная часть представлена классами Алгоритмы1 и Алгоритмы2, Общие — интерфейсы Потребности и Услуги и классы Список Элементов и Упорядоченный Список. Выбранный Вариант сохраняется в переменной private readonly Услуги mixin, а Механизм Вариации — это оператор if в конструкторе public Упорядоченный Список.

Наследование, основанное на объектах или шаблоне делегирования (*delegation pattern*). Это механизм вариации, который, в отличие от других, применяется на уровне объектов, а не классов. Это способ, которым объект внешне демонстрирует некоторое поведение, но фактически передает ответственность за выполнение соответствующего действия другому объекту, как в следующем примере, где *Принтер* делегирует объект типа Принтер.

```
class Принтер {//делегированный
void печатать() {/*..*/}}
class Принтер {// делегирующий
Принтер принтер = new Принтер();
void print() {принтер.печатать();}
}
```

3. **Параметризация.** Параметризованным типом может быть класс, структура, интерфейс или метод, в котором есть ссылка на место конкретизации параметров одного или нескольких типов, которыми он оперирует или которые сохраняются (конструкция вроде *placeholders* (*type parameters*)). Этот механизм вариации предусматривает наличие общего кода для работы с разными типами данных. Яркими представителями являются шаблоны и универсальные шаблоны. Они являются синтаксически и функционально подобными, но различаются внутренним механизмом реализации. Пример использования включает переменные: List<Панель> и List<ЭлементУправления>. Они могут сохранять массивы объектов любых типов, порожденных от типов Панели и ЭУ соответственно. При этом алгоритм работы остается неперемененным.

4. **Перегрузка.** Сущность этого механизма заключается в вариации типов, над которыми выполняется оператор или за которыми вызывается функция. То есть ЯП предоставляет возможность повторного использования имени для операции над другими типами. Выбор варианта осуществляет программная среда ЯП. Применяется к процедурам, функциям или операторам. Языки, которые поддерживают перегрузку: Ada, C++, C#, D, Erlang, F#, Groovy, Java, JavaScript, Haskell, Common Lisp, Nemerle, Scala, VB.NET, Delphi, Ocaml. Язык Java не поддерживает перегрузку операторов, но выполняет внутреннюю перегрузку оператора «+» для сочетания строк. В C#, например, можно перегружать методы, унарные и бинарные операторы.

Использование этого механизма можно рассмотреть на примере метода public object Найти(/*элемент*/). Система сама решит, что возвращать как результат: найденный элемент или его позицию в списке, если разработчик добавит еще один метод с таким же названием, но другим исходным параметром, public int Найти(/*элемент*/).

5. **Динамическая загрузка классов.** Сущность механизма заключается в том, что класс загружается в память лишь тогда, когда он действительно начинает использоваться так, как реализовано, например, в виртуальной машине языка Java. Для управления этой функциональностью используется (MainClass.class.getClass Loader ();).

6. **Отображение.** Механизм, основанный на способности программы модифицироваться, имеет собственное поведение. Такие манипуляции достигаются благодаря делению ПС на базовое и цель уровня. Цель уровня предоставляет информацию об избранных системных свойствах, поскольку базовый уровень, который включает логику программы, ориентирован на цель уровня, на изменение цели уровня и влияет на поведение базового уровня. В СПС общая функциональность расположена на базовом уровне и может меняться во время работы в зависимости от конфигурационной информации, которая используется на уровнях.

Пример реализации отображения в языке C#:

```
//Без отображения
Университет унив = new Университет ();
унив.СписокФакультетов();
//Отображение
Type type = Type.GetType(
«UnivercityNamespace. Университет «);
object univ = Activator.CreateInstance(t);
type.InvokeMember("СписокФакультетов  ",  BindingFlags.
InvokeMethod, null, univ, null);
```

7. **Шаблоны** (или паттерны) архитектуры и проектирования (*patterns*) могут использоваться в СПС, потому что большинство из них предоставляет готовые решения для управления вариабельностью (рис. 4.6).

Применение шаблонов покрывает некоторые предыдущие механизмы, такие как: агрегация, подобие и параметризация.

Класс Абстракция определяет абстрактный интерфейс для Клиента и содержит переменную исполнитель типа Исполнитель. Конкретизация Абстракции расширяет интерфейс, определенный Абстракцией.

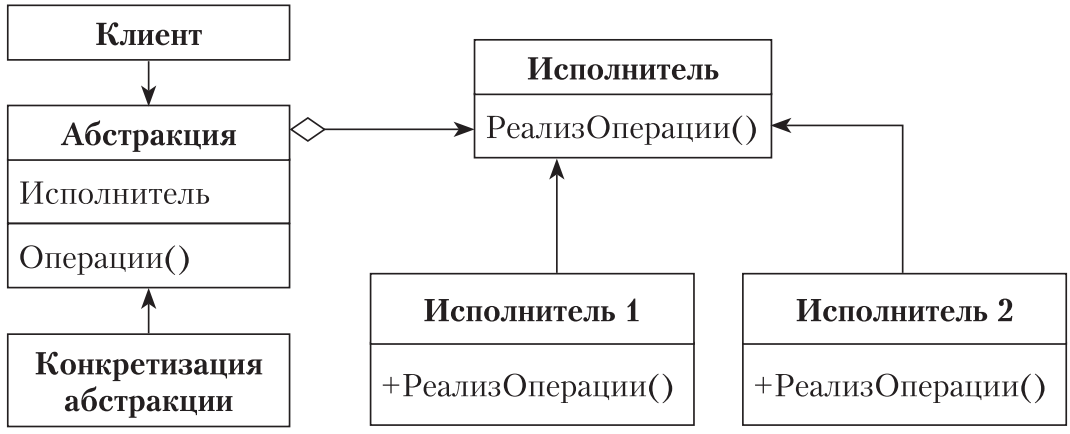


Рис. 4.6. **Шаблон**

Исполнитель определяет интерфейс для определенного исполнителя. Он не должен полностью отвечать интерфейсу Абстракции, т.е. он может

быть полностью отличным. Типичная ситуация, когда исполнитель предоставляет только примитивные операции, а Абстракция определяет операции высокого уровня, которые базируются на примитивных. Конкретный исполнитель определяет конкретные реализации простых операций. Налагая этот шаблон в схеме КПИ, можем увидеть, что Абстракция является общей частью. Переменная сохраняет вариант в функции Операция(), в котором находится механизм вариации, а Клиент является другим КПИ, зависимым от общей части, и не зависит от интерфейса определенных исполнителей.

Рассмотренные механизмы вариации не покрывают всю проблемную область обеспечения вариантов, но являются наиболее используемыми в парадигме ООП. Они могут быть применены в других парадигмах, а некоторые из них концептуально заложены в эти парадигмы и могут служить инструментами вариации ПС.

4.2. Базовые модели проектирования систем

Моделирование ПС и их семейств выполняется средствами КП в части построения отдельных КПИ и их сборки в конфигурационный вариант ПС или СПС на основе спецификаций компонентов в пространстве проблем. Это описание базируется на подходах FOD (*Feature-Oriented Development*) построения МХ ПрО, MDD (*Model Driven Development*) и MDA (*Model Driven Architecture*) и др. Архитектура системы может выполняться с помощью платформо-независимой модели PIM (*Platform Independent Model*) и платформо-зависимой модели PSM (*Platform Specific Models*). Концепции двухуровневого моделирования архитектуры MDA (*Model Driven Architecture*) и отображения PIM ↔ PSM соответствует концепции отображения пространства проблемы в пространстве решений.

4.2.1. Модели MDD, MDA

Модель MDD

Базируется на унифицированном языке моделирования UML с графической формой представления семантики. В MDD (*Model Domain Development*) определяется система с четко определенной семантикой, и ее артефакты могут быть преобразованы, трансформированы к коду реализации (аналогично тому, как код Java компилируется в байт-код). При MDD определяется модель приложения (*Application Model*), члены которого имеют общие функции, но зависят от платформы реализации компонентов и данных. Преимущество применения MDD для построения членов семейства заключается в том, что «управление» точкой вариантности для платформы выполняется автоматически путем трансформации моделей PIM — PSM. Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований, а также достижения качества ПС[1].

[1] См.: *Лаврищева Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства : учебник для вузов. 2-е изд. М. : Издательство Юрайт, 2016.

Приспособление подхода MDD к потребностям приложения может быть выполнено путем добавления к модели домена вариантных *точек* для управления альтернативными концепциями вариантной конфигурационной структуры. Это особенно необходимо при их уточнении или изменении некоторых КПИ в ПС. Выбор разных концепций представления модели домена будет предопределять появление разных прикладных моделей ПС, которые при генерации автоматически трансформируются к модели MDD.

Модель MDA

В основе этой управляемой архитектуры лежит идея разделении этапов моделирования и последующей реализации приложения на конкретной платформе. Сначала с помощью специальных средств создается общая и независимая от способов реализации модель приложения, а затем осуществляется ее реализация в некоторой среде разработки. Главная цель этой модели — функциональная совместимость, переносимость и повторное использование.

Архитектура MDA возникла на основе наличия ряда стандартов и технологий. Концептуальная основа MDA — спецификации OMA, ORB, CORBA, ООП, стандарт CWM, языки UML, XML, MOF.

MDA — это новый виток эволюции технологий программирования в плане описания процесса разработки в целом и с использованием современных средств представления, что позволяет автоматизировать создание приложений. MDA станет промышленным стандартом.

В классической модели MDA действует схема (рис. 4.7, *а*). С учетом потребностей доменов эта схема модифицируется (рис. 4.7, *б*).

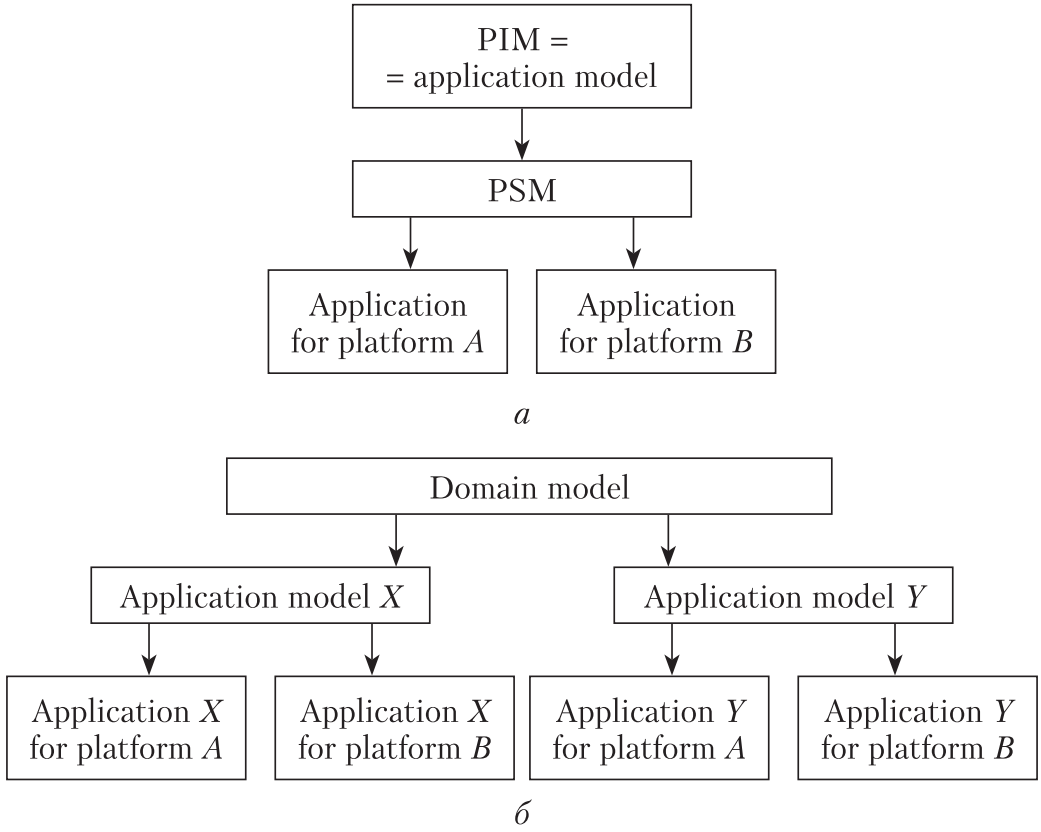


Рис. 4.7. **MDA — два варианта семейств систем**

Фактически спецификация ПС соответствует подходу MDD (см. рис. 4.7, *а*) и определяет модель семейства ПС из приложений, члены которого имеют общие функции, но различаются платформами реализации. Это видно на рис. 4.8, где показаны два способа реализации информационного взаимодействия прикладных приложений.

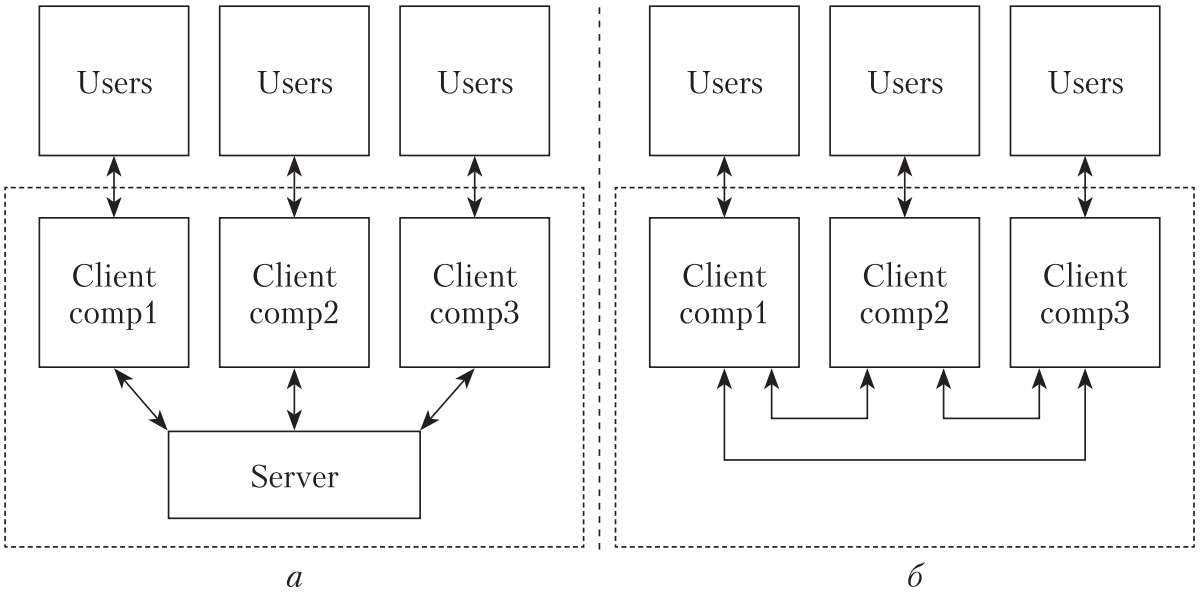


Рис. 4.8. **Два подхода к реализации взаимодействия приложений**:
а — централизованный доступ; *б* — взаимодействие парами

Выбор альтернативных платформ связан с точкой вариантности в семействе, которая находится над моделью ПС и невидима на уровне конкретной ПС из семейства СПС.

Преимущество применения MDD для построения члена семейства заключается в том, что «управление» точкой вариантности платформ выполняется через трансформацию PIM ⇔ PSM.

Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований к качеству и инфраструктуре, а также активов, которые реализуют альтернативные концепции. Приспособление подхода MDD к потребностям ГП может быть выполнено путем добавления *модели ПрО*, которая будет определять точки, где могут быть выбраны альтернативные концепции.

Выбор разных концепций из модели (ПрО) предусматривает появление разных прикладных моделей ПС, которые трансформируются автоматически в соответствии с подходом MDD.

Таким образом, преимущества этого подхода следующие: независимость модели от средств разработки; возможность переноса из одной операционной системы в другую и экономия ресурсов приложения одновременно для нескольких платформ; создание частей приложения.

4.2.2. Модели систем для разных платформ PIM и PSM

Проектирование ПС проводится, как правило, с помощью стандартных моделей и моделей ЖЦ (спиральной, водопадной, итерационной и др.). К стандартным моделям проектирования относятся MDA, MDD, PIM,

PSM и др. Модели MDA и MDD рассмотрены выше. Здесь дается характеристика моделей PIM и PSM, которые могут адаптироваться к разным платформам компьютеров.

Платформенно-зависимая модель PSM (*Platform Specific Model*) задает состав, структуру, функциональность системы применительно к конкретной платформе. Модель платформы задает технические характеристики, интерфейсы, функции платформы. Она используется при преобразовании модели PIM (*Platform Independent Model*) в модель PSM.

В зависимости от уровня детализации платформы модели могут содержать сведения о различных функциональных частях системы. Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего исполнение его назначения. Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным. Для приложений .NET чаще всего применяются возможности ADO.NET. Уровень пользовательского интерфейса содержит состав форм приложения, функциональные элементы управления. Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удается создать типовые шаблоны элементов управления для основных операций, то появляется возможность автоматической генерации форм и их содержания при создании приложений.

Вычислительно-независимая модель CIM (*Computation Independent Model*), структурная схема которой приведена на рис. 4.9, базируется на модели PIM для ее преобразования к платформо-зависимой модели с использованием языка моделирования UML.

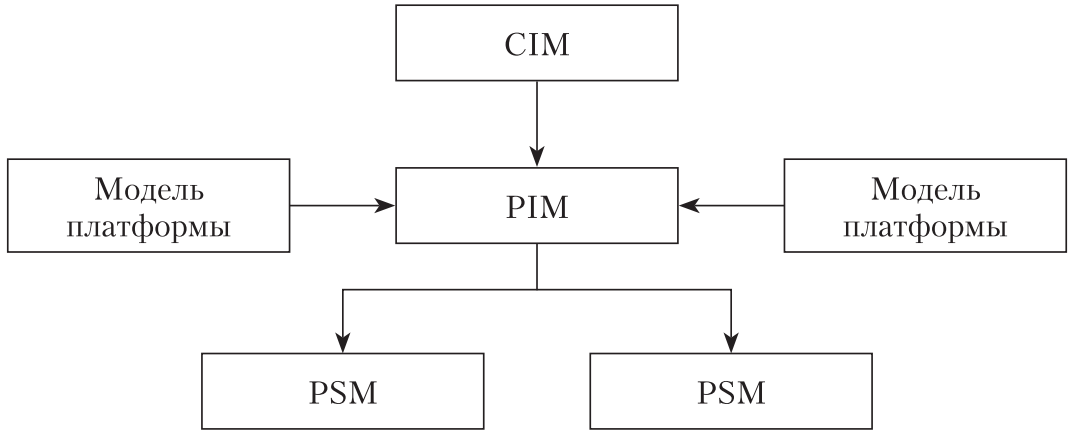


Рис. 4.9. **Структурная схема CIM**

Эта модель включает в себя разработку общих требований к системе, создание общего словаря понятий и описание окружения, в котором система будет функционировать. Сущности и понятия, описываемые в модели CIM, должны тщательно анализироваться и отрабатываться. Модель CIM должна отображать общую концепцию системы, используемую для программирования элементов приложения. Преобразование CIM в платформо-независимую модель PIM осуществляется средствами языка UML. Она включает в себя элементы, описывающие бизнес-логику, общую структуру системы, состав и взаимодействие подсистем, распреде-

ление функциональности по элементам и требования к пользовательскому интерфейсу. При этом модель PIM включается во все автоматизированные среды разработки приложений, использующих модель MDA.

Переход к платформо-зависимой модели определяется числом программных платформ, на которых будут функционировать элементы приложения. Отмечаются случаи, когда приложение (или его составные элементы) должны работать на нескольких платформах одновременно. Тогда модель PSM создается путем преобразования модели PIM к модели другой платформы. На этапе создания модели PSM разработка приложения осуществляется с помощью модели MDA, которая преобразуется к коду другой платформы.

Преобразование моделей PIM → PSM. На этом этапе общее описание системы на языке UML включает выполнение следующих действий:

1) разработка схемы преобразования (*mapping*);
2) маркирование (*marking*);
3) трансформация (*transformation*).

Для каждой платформы создается собственная схема преобразования, зависимая от возможностей платформы (формата, языков, средств обработки и др.). Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойств), так и механизмы ее представления (метамодель, используемые ТД, форматы данных платформы). В схеме преобразования свойствам метамодели и элементам модели PIM ставятся в соответствие свойства метамодели и элементы платформо-зависимой модели PSM.

При преобразовании PIM → PSM моделей может использоваться несколько схем преобразования. Для их связывания применяется марка (*mark*) как самостоятельная структуры данных, принадлежащая не моделям, а схемам преобразования, содержащая информацию о созданных связях (рис. 4.10). Процесс задания марок называется маркированием. Простейшим случаем является элемент модели PIM, который соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования этой модели. В процессе маркирования используются сведения о платформах, которые содержатся в модели платформы.

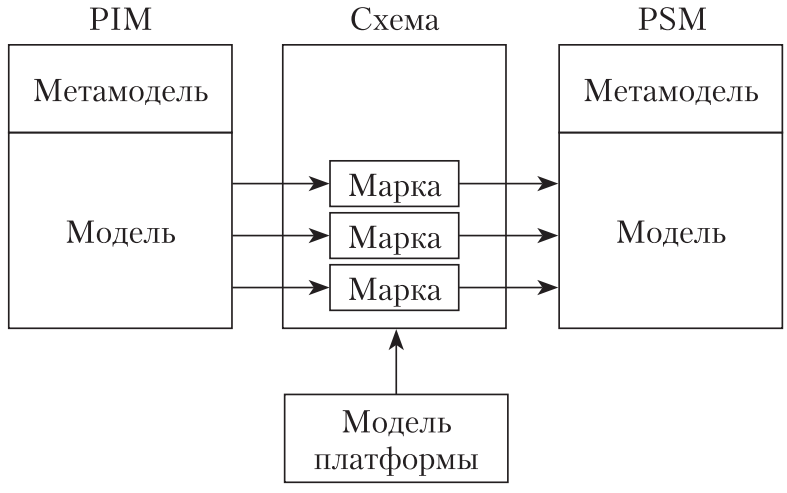


Рис. 4.10. **Схема преобразования метамоделей**

Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования документируется картой переноса элементов модели и метамодели.

4.3. Моделирование систем в языке UML

Процесс построения системы в среде UML проводится на этапе анализа ПрО и проектирования. В процессе анализа системы создается концептуальная объектная модель ПрО в виде диаграмм прецедентов. Она уточняет внешнее функциональное поведение системы. На этапе проектирования разрабатываются диаграммы классов и взаимодействия. Они образуют каркас системы. Для определения поведения классов объектов используются диаграммы состояний и деятельности. Размещение объектов фиксируется компонентными диаграммами, а расположение программных модулей в узлах (компьютеров) сети — диаграммами развертывания.

С помощью UML графических диаграмм описываются три основные объектные модели системы:

1) *структурная* (статическая) для описания структуры сущностей системы, включая классы, интерфейсы, отношения, атрибуты;

2) *модель поведения* (динамическая) описывает поведение объектов системы, включая взаимодействие и изменение состояний отдельных компонентов или всей системы;

3) *модель функционирования* описывает операции функционирования системы.

Этим моделям соответствуют структурные, поведенческие и аннотационные сущности, а также типы отношений: зависимость, ассоциация, обобщение, реализация. Есть также их вариации, например уточнения, трассировки, включения и расширения, агрегация и композиция.

Для проектирования систем используются следующие виды диаграмм:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграммы поведения, к которым относятся диаграммы состояний, деятельности, взаимодействия;
- диаграммы последовательности и кооперации;
- диаграммы реализации, к которым относятся диаграммы компонентов и развертывания.

Диаграммы UML описаны в Приложении 1 данного учебника.

4.3.1. Стратегия использования UML

В процессе анализа системы создается концептуальная ОМ ПрО в виде диаграмм прецедентов. Она уточняет внешнее функциональное поведение системы и используется в дальнейшем как проектная документация для обсуждения всеми заинтересованными сторонами, участвующими в разработке ПС.

Диаграммы UML являются основой работы над проектом (разработка других диаграмм, тестирование и др.). На этапе проектирования разраба-

тываются диаграммы классов и взаимодействия. Они могут использоваться для автоматической генерации каркасного кода. Для определения поведения классов со сложной динамикой реагирования на события могут быть задействованы диаграммы состояний и деятельности. Размещение объектов с программными модулями задается компонентными диаграммами, а расположение программных модулей в узлах сети — диаграммами развертывания.

На этапе кодирования построенные диаграммы реализуются средствами языков C#, C++, Java и другими языками ООП. Затем обосновывается правильность системы. При этом используются специальные ООП-средства тестирования и отладки. Модификация системы на этапе ее сопровождения происходит с помощью уже существующих объектов, что уменьшает стоимость и риски в новой версии системы. Изменения в реализации какого-либо объекта или добавление к нему новых методов не влияет на другие объекты системы.

4.3.2. Описание в UML веб-сайта оплаты услуг

Рассматривается ООАП автоматизированной онлайн-системы создания веб-сайтов. Для краткости назовем ее автоматизированный веб-построитель (АВП).

Интернет-пользователь имеет возможность построить собственный веб-сайт, пользуясь интуитивно понятным интерфейсом. Построение веб-сайта состоит из нескольких шагов, включая регистрацию. Далее пользователь может изменить любые свойства своего сайта, пользуясь теми же программными модулями и пользовательскими интерфейсами, какими он пользовался при создании. За использование сайта пользователю начисляется абонентская плата. Моделирование веб-сайта системы начнем с построения диаграммы вариантов использования.

Диаграмма вариантов использования (*Use Case Diagram*). В качестве актора выступает пользователь этой системы. Пользователь может производить две операции — менять свойства веб-сайта и производить финансовые расчеты. Эти операции выражены на диаграмме двумя вариантами использования: «ИзменитьСвойстваВебСайта» и «ОплатитьУслуги». У одного пользователя может быть от одного до нескольких веб-сайтов, что отображено на отношениях ассоциации кратностью «1...n».

Вариант использования «ИзменитьСвойстваВебСайта» состоит из трех вариантов использования: «ИзменитьТему», «ИзменитьДизайн», «ИзменитьРегистрацИнформ». Они связаны с «ИзменитьСвойстваВебСайта» отношениями включения. Это отношение передает поведение ПрО. Процессы изменения свойств веб-сайта или создания веб-сайта состоят из этих трех этапов. Варианты использования «СоздатьВебСайт» и «ИзменитьСвойстваВебСайта» связаны отношением расширения. Следующий вариант использования «ОплатитьУслуги» состоит из составляющих его трех вариантов использования: «СообщитьОбОплате», «ПровестиТранзакцию», «АнализироватьРезультатТранзакции». Они связаны с «ОплатитьУслуги» отношениями включения.

Процесс оплаты услуг состоит из расчета стоимости и уведомления клиента («СообщитьОбОплате»), приема оплаты («ПровестиТранзакцию»)

и анализа финансового состояния счета клиента («АнализироватьРезультатТранзакции»). В результате этого анализа принимается решение о продолжении предоставления услуг, если клиент их оплатил, или о прекращении их предоставления, если нет.

Инициализацию варианта использования «ИзменитьСвойстваВебСайта» производит актор — «Пользователь». Этот вариант использования и составляющие его варианты пассивны.

Процесс взаимодействия диктуется пользователем. Варианты использования «ИзменитьТему», «ИзменитьДизайн», «Изменить РегистрацИнформ» последовательны по отношению друг к другу и могут взаимодействовать с «Пользователем» в прямом или обратном направлении относительно порядка, представленного на рис. 4.11.

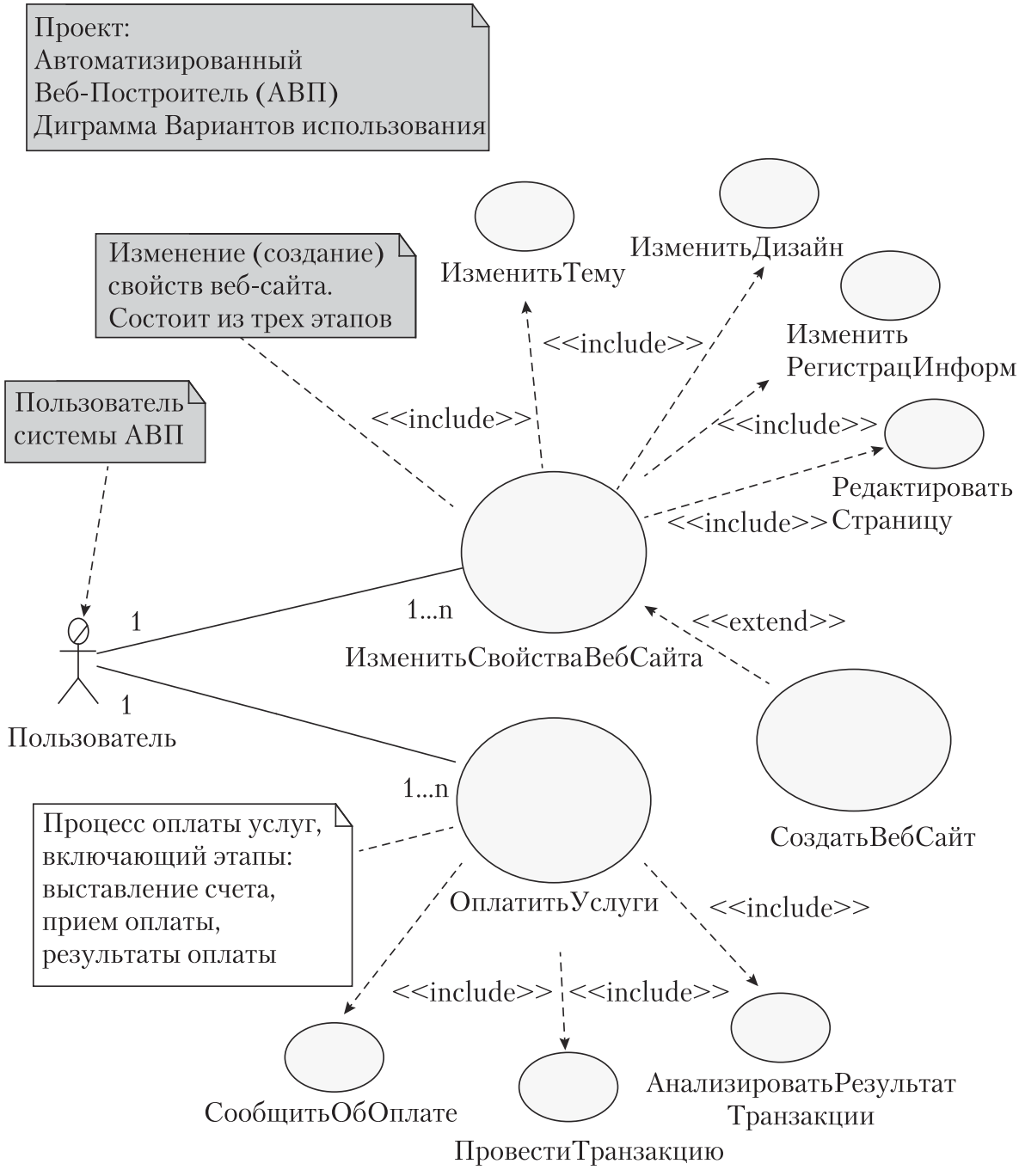


Рис. 4.11. **Диаграммы использования на сайте**

«ОплатитьУслуги» инициализируется через **вариант использования** «Пользователем» при первой же его регистрации в АВП. Здесь варианты использования «СообщитьОбОплате» и «АнализироватьРезультатТранзакции» активны и существуют на протяжении всего времени существования системы. Они являются своего рода процессами-демонами. Первый отвечает за финансовые расчеты и уведомление «Пользователя» о необходимости оплаты. После уведомления «Пользователь» инициирует вариант использования «ПровестиТранзакцию». Второй процесс-демон, представленный вариантом использования «АнализироватьРезультатТранзакции», взаимодействует с «Пользователем» путем посылки сообщений об анализе результата транзакции.

В диаграмме вариантов использования определены следующие основные классы (рис. 4.12).

BillingSystem. Класс отвечает за финансовые расчеты. Имеет атрибуты периода оплаты (*billing Period*) и баланс (*balance*). Посылает управляющие сообщения классам Customer, TariffPlan, ControlSystem. Получает сообщения от тех же классов.

TariffPlan. Класс управляет информацией по тарифным планам. Атрибутами является информация о ценах. Посылает сообщения классам Customer и BillingSystem. Получает сообщения от этих же классов.

Site. Представляет собой объект веб-сайта пользователя. Имеет операции по созданию и по управлению веб-сайтом. Получает управляющие сообщения от классов Customer и ControlSystem. Классы Template, PageSet, Ecommerce, PictureGallery связаны с классом WebSite отношением агрегации, так как эти классы являются его составными элементами.

Customer. Представляет собой класс пользователя. Имеет атрибуты имени (customerName) и контактной информации (contactInfo). Посылает управляющие сообщения классам WebSite и BillingSystem, получает сообщения от класса BillingSystem.

ControlSystem. Класс-демон, объект которого существует на протяжении работы системы. Управляет процессами запуска управляющих функций (расчет стоимости услуг, включение/отключение веб-сайтов). Посылает управляющие сообщения классам WebSite и BillingSystem. Получает сообщения от класса BillingSystem.

Для моделирования процесса редактирования содержимого страницы используется вариант «РедактироватьСтраницу».

Для процесса оплаты услуг (Use case «ОплатитьУслуги») используется **диаграмма кооперации**. Процесс оплаты услуг инициируется классом controlSystem путем посылки сообщения вызова операции calculateDebt() (рассчитать долг) класса BillingSystem. Сообщение представляет собой асинхронный поток управления.

Класс BillingSystem, в свою очередь, посылает сообщение вызова операции getTariffPlan() (получить тарифный план) класса TariffPlan. Поток управления синхронный. Классу TariffPlan для выполнения операции getTariffPlan() требуется вызов операции getCustomerInfo() (получить информацию о пользователе) класса Customer. Возвращаемые результаты этих сообщений представлены сообщениями в виде стрелок, обращенных в обратную сторону с кружком в начале. После получения

ответной информации от класса TariffPlan класс BillingSystem вызывает операцию sendBillingMessage() (уведомить пользователя об оплате) класса Customer путем посылки сообщения. В ответ класс Customer вызывает операцию acceptPayment() (принять оплату) класса BillingSystem путем посылки сообщения. После этого класс BillingSystem возвращает ответ классу ControlSystem на сообщение. Этот ответ обрабатывается в классе ControlSystem и как результат генерируется сообщение вызова операции changeStatus() (изменить состояние) класса WebSite.

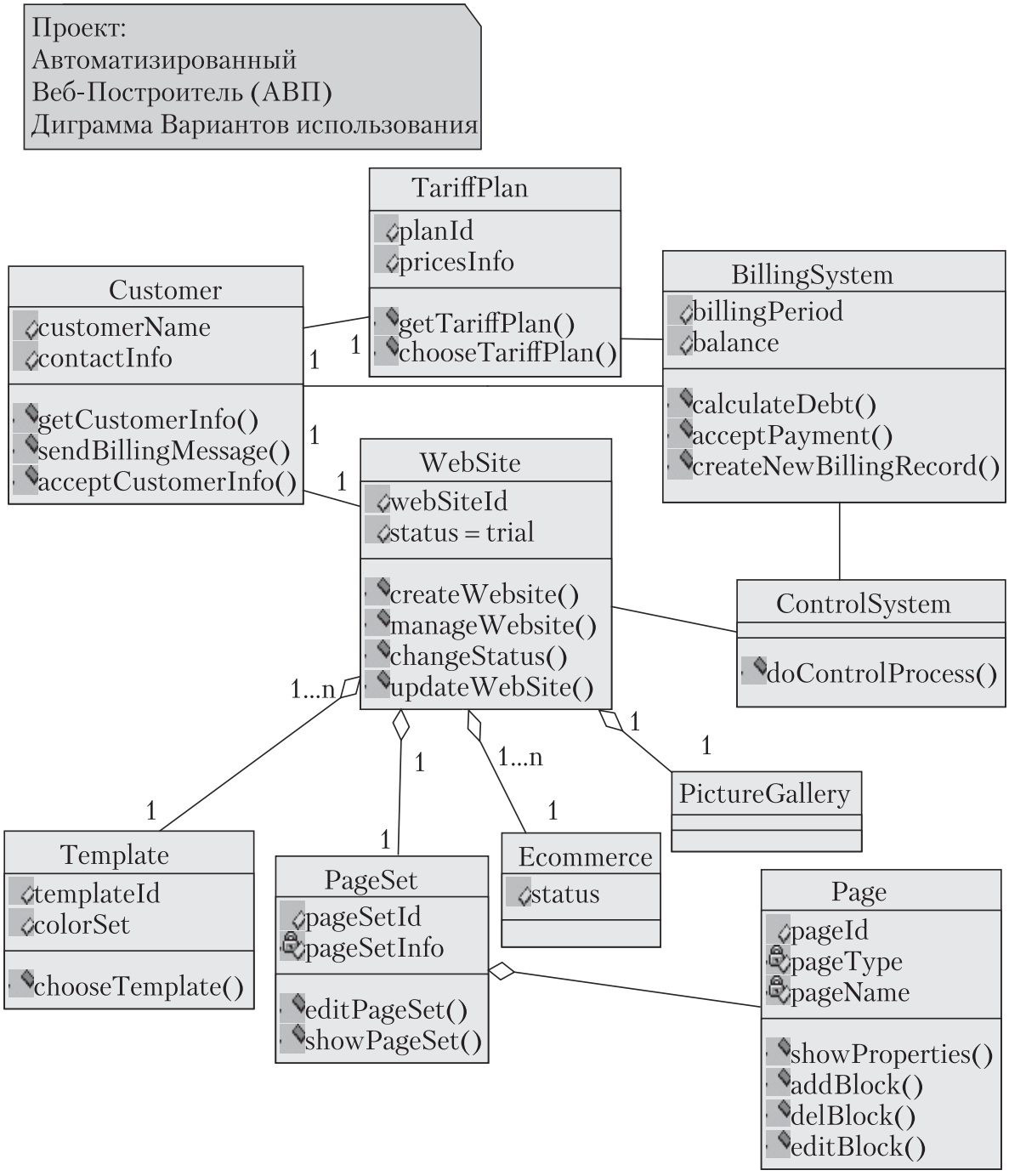


Рис. 4.12. **Диаграмма классов расчета оплаты за услуги**:
◇— агрегация с классом

Реально происходит следующее: система управления дает команду расчетной системе рассчитать стоимость оказанных пользователю услуг. Для

этого расчетная система запрашивает данные о тарифном плане и реквизитах пользователя. Далее происходит расчет стоимости услуг и посылается уведомление пользователю. После этого пользователь оплачивает (или не оплачивает услуги). На основании информации об оплате расчетная система выдает результат системе управления в виде ответа на сообщение. На основании этого результата система управления отключает или не отключает веб-сайт пользователя.

Процесс оплаты услуг (Use case «ОплатитьУслуги») инициируется классом controlSystem путем посылки сообщения вызова операции calculateDebt() (рассчитать долг) класса BillingSystem. Сообщение представляет собой асинхронный поток управления.

BillingSystem, в свою очередь, посылает сообщение вызова операции getTariffPlan() (получить тарифный план) класса TariffPlan. Поток управления синхронный. Классу TariffPlan для выполнения операции getTariffPlan() требуется вызов операции getCustomerInfo() (получить информацию о пользователе) класса Customer. Возвращаемые результаты этих сообщений представлены сообщениями в виде стрелок, обращенных в обратную сторону с кружком в начале. После получения ответной информации от класса TariffPlan, класс BillingSystem вызывает операцию sendBillingMessage() (уведомить пользователя об оплате) класса Customer путем посылки сообщения. В ответ класс Customer вызывает операцию acceptPayment() (принять оплату) класса BillingSystem путем посылки сообщения. После этого класс BillingSystem возвращает ответ классу ControlSystem на сообщение. Этот ответ обрабатывается в классе ControlSystem, и, как результат генерируется сообщение вызова операции changeStatus() (изменить состояние) класса WebSite.

Диаграмма кооперации для процесса создания веб-сайта (Use case «СоздатьВебСайт», рис. 4.13).

Процесс создания веб-сайта инициируется классом Customer путем посылки сообщения № 1 вызова операции createWebSite() (создать веб-сайт) класса WebSite. WebSite, свою очередь, посылает сообщение № 2 вызова операции chooseTariffPlan() (выбрать тарифный план) класса TariffPlan — запись о финансовом состоянии пользователя.

После получения ответа класс WebSite посылает сообщение № 3 вызова операции chooseTemplate() (выбрать графический шаблон) класса Template. Затем класс WebSite посылает сообщение № 4 вызова операции editPageSet() (редактировать набор страниц) класса PageSet. Регистрационная информация о пользователе вносится в систему посредством сообщения № 5 — вызов операции acceptCustomerInfo() класса Customer. Последняя операция № 6 — createNewBillingRecord() класса BillingSystem создает новую учетную запись.

Для процесса редактирования содержимого страницы (Use case «РедактироватьСтраницу»). Процесс редактирования инициируется классом Customer путем посылки сообщения № 1 вызова операции showPageSet() (показать набор страниц) класса PageSet. Затем идут сообщения № 2 и № 3, вызывающие операции showProperties() и editBlock() класса Page соответственно. После завершения редактирования изменения публику-

ются на веб-сайте путем вызова операции updateWebSite() класса WebSite (сообщение № 4).

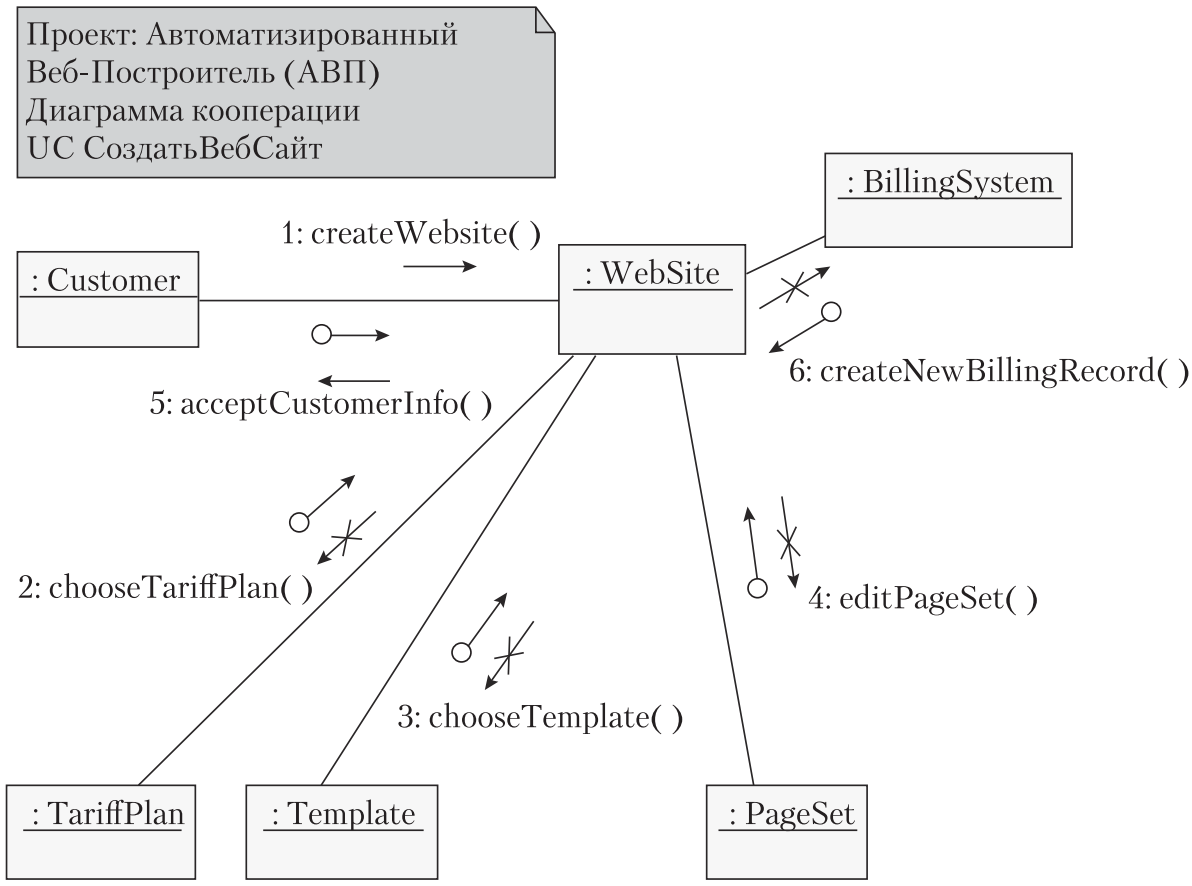


Рис. 4.13. **Диаграмма «Создать веб-сайт»**

Процесс создания веб-сайта (**диаграмма кооперации** Use case «СоздатьВебСайт») инициируется классом Customer путем посылки сообщения № 1 вызова операции createWebSite() (создать веб-сайт) класса WebSite. WebSite, свою очередь, посылает сообщение № 2 вызова операции chooseTariffPlan() (выбрать тарифный план) класса TariffPlan. Далее, после получения ответа класс WebSite посылает сообщение № 3 вызова операции chooseTemplate() (выбрать графический шаблон) класса Template. Затем класс WebSite посылает сообщение № 4 вызова операции editPageSet() (редактировать набор страниц) класса PageSet. Регистрационная информация о пользователе вносится в систему посредством сообщения № 5 — вызов операции acceptCustomerInfo() класса Customer. Последняя операция № 6 — createNewBillingRecord() класса BillingSystem — создает новую учетную запись о финансовом состоянии пользователя. **Диаграмма состояний** описывает класс BillingSystem(). Этот класс периодически производит процедуру расчета, что отображается переходом в себя наверху диаграммы (рис. 4.14).

Диаграмма деятельности показывает процесс расчета и оплаты услуг (Use case «ОплатитьУслуги»). Сначала происходят последовательные действия с целью расчета стоимости оказанных услуг. Далее происходит ветвление по альтернативным ветвям в зависимости от значения сторожевого условия. Если выполняется условие «[долга нет]», то происходит переход

в конечное состояние. В противном случае происходит разделение потоков на два параллельных (рис. 4.15).

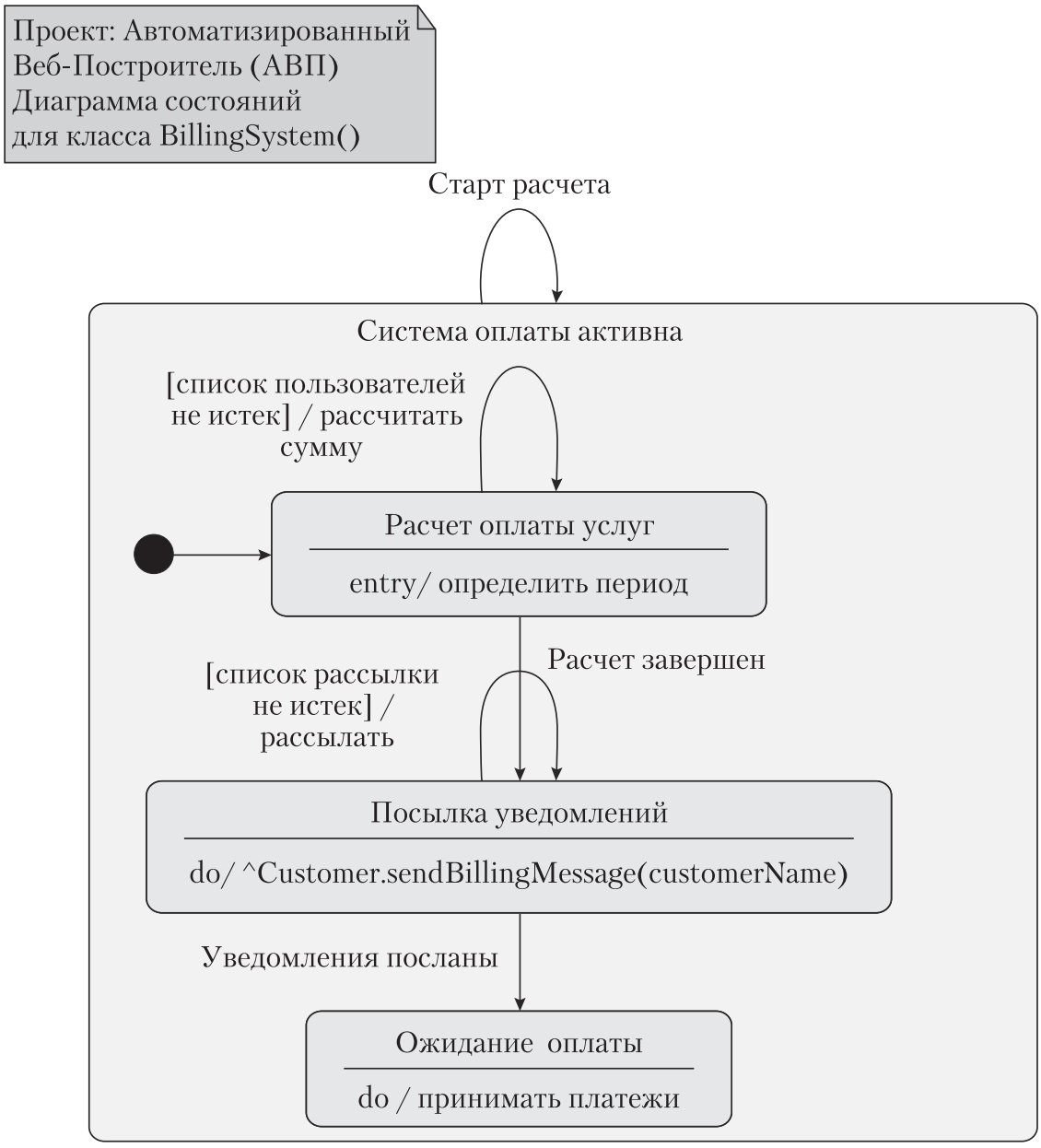


Рис. 4.14. **Диаграмма оплаты услуг**

В левой ветви происходят действия «Послать уведомление об оплате» и «Получить оплату», а в правой — «Получить оплату». Распараллеливание вызвано тем, что в общем случае пользователь может оплатить услуги, не дожидаясь уведомления. Далее параллельные потоки сливаются в один, и происходит ветвление в зависимости от сторожевого условия. При условии «[оплата не получена]» выполняется деятельность «Отключить услугу». Далее — выход в конечное состояние.

Диаграмма деятельности показывает структуру создания веб-сайта (Use case «СоздатьВебСайт») (рис. 4.16).

Моделирование структуры систем диаграммами UML представляет собой сложную проблему. Трансформация диаграмм к ЯП вызывает определенные трудности анализа и проверки правильности полученной системы.

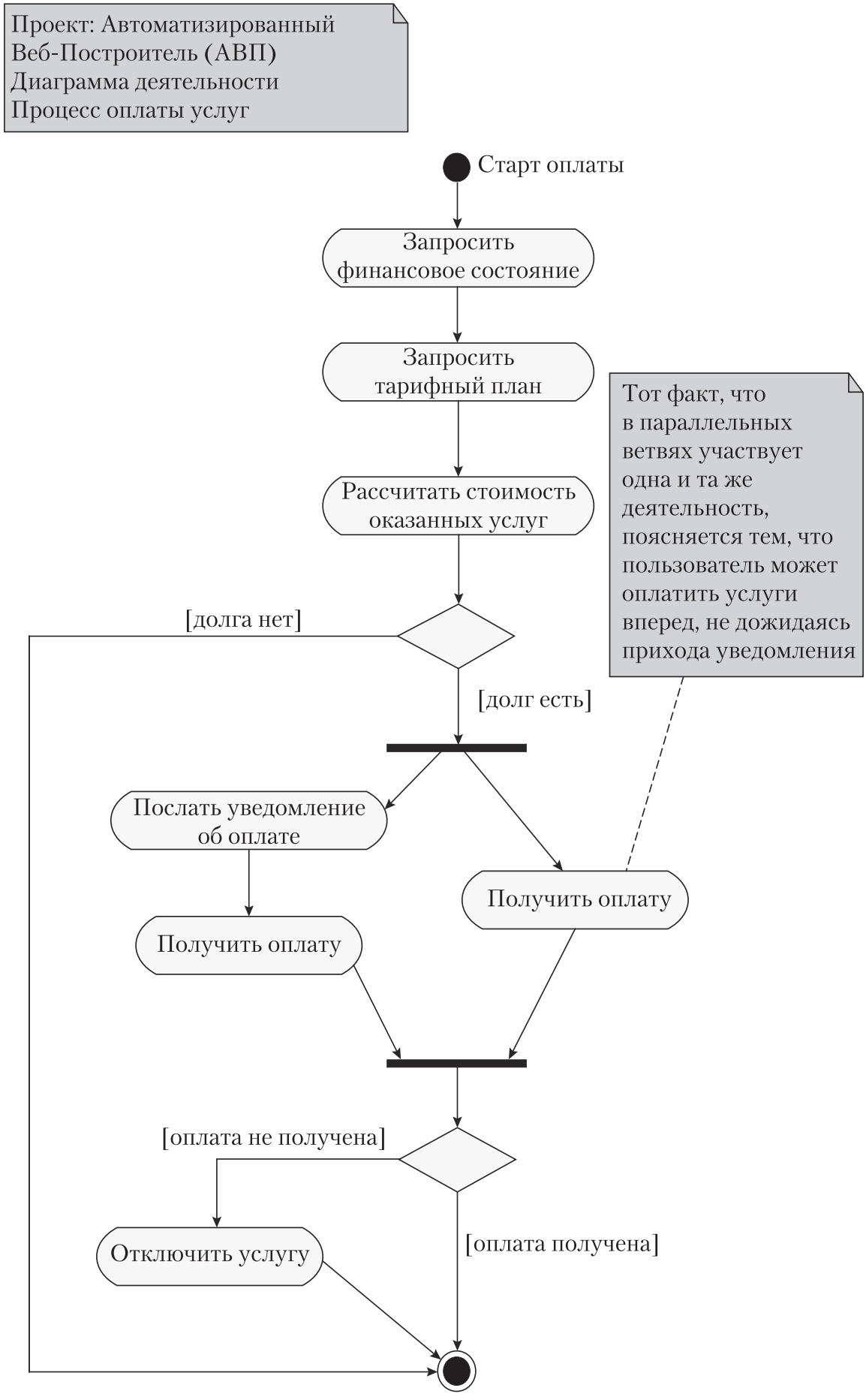


Рис. 4.15. **Диаграмма сайта «Оплата услуг» с потоками**

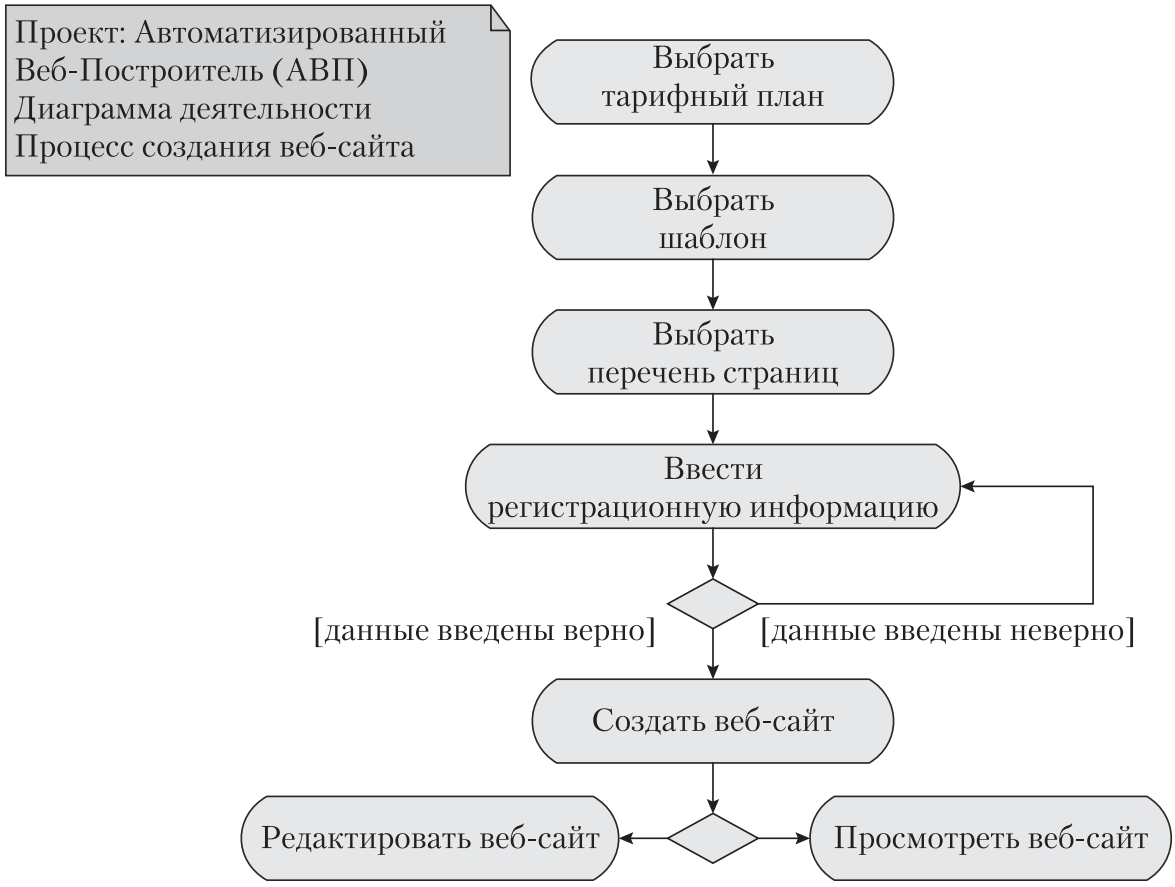


Рис. 4.16. **Диаграмма «Создать сайт оплаты услуг»**

4.4. Аспекты управления вариабельностью ПС

Потребности современного рынка программной продукции в разных предметных областях стимулируют организации-разработчики к быстрому созданию и выпуску на рынок высококачественных ПС и своевременной реакции на изменения заказчиков, появлению новых сред (Web, Grid), архитектурных платформ и т.п. Это, в свою очередь, ставит задачи совершенствования имеющихся методологий разработки ПС. Одна из новейших методологий базируется на идее построения семейств функционально подобных ПС для отдельных ПрО. СПС — это совокупность ПС, которые имеют общее множество характеристик, соответствующих потребностям определенных функциональных сегментов ПрО, различаются способами воплощения этих характеристик с использованием ГоР. ПС, члены СПС, не создаются «из нуля», а порождаются на основании общей модели СПС, которая конкретизируется в соответствии со спецификацией требований к ПС. Ресурсы СПС накапливаются в репозиториях среды разработки и выбираются для сборки их в ПС.

Определение 4.1. Модель вариабельности СПС — это пара

$$M_{var} = (SV; AV); SV = \langle G_1; \langle G_t, TR_t \rangle, t = 2, ..., 5; C; D \rangle, \qquad (4.1)$$

где SV — подмодель вариабельности в структуре семейства ПС/ПП; AV — подмодель вариабельности в артефактах (и ПС/ПП) семейства; $G_t = (F_t,$

L_t) — граф артефактов уровня $t = 1, ..., 5$, обусловленный моделью характеристик семейства ПС/ПП; TR_t — двусторонние связи трассируемости артефактов уровней $t-1$ и t; C и D — предикаты на $\otimes F_t$, задающие ограничения и зависимости артефактов.

Определение 4.2. Модель вариабельности в структуре СПС — это кортеж

$$SV = \langle\langle CF; \langle DR, TC\rangle; \langle AR, TD\rangle; \langle CM, FR, TS, TA\rangle; \langle ER, TF\rangle\rangle; \text{Constr; Dep}\rangle, \quad (4.2)$$

где $CF = \langle SF, LF\rangle$ — диаграмма характеристик, которая задает множество постоянных и переменных свойств ПС (SF), предлагаемых их потенциальным потребителям в ПрО и ограничения ($Constr$) и зависимости (Dep) между ними посредством связей обязательного и вариантного подчинения (LF);

$DR = \langle SF \cup SR, LF \cup LR\rangle$ — аналогичная диаграмма, где характеристики $f \in F$ детализированы и (или) дополнены характеристиками $r \in SR$, что подают постоянные и переменные «технические» требования к ПС со стороны аналитиков СПС, ограничения и зависимости которых подаются связками LR;

$TC = \{(r, f), r \in SR, f \in SF)\}$ — двусторонние связи трассировки между «техническими» требованиями к ПС и их свойствами для потребителя, которые конкретизируются требованиями;

$AR = \langle AC, LC\rangle$ — эталонная архитектура СПС, т.е. описание взаимосвязей (LC) между формальными представлениями программных компонентов (AC) для реализации характеристик $f \in DR$ в определенной нотации для программной архитектуры (UML, xADL и т.п.);

$TD = \{(ac, f), ac \in AC, f \in SF \cup SR)\}$ — двусторонние связи трассировки между представлениями компонентов $ac \in AC$ для реализации характеристик $f \in DR$ и этими характеристиками;

CM — формальные представления базовых программных компонентов для реализации компонентов архитектуры при составлении элементов компонентного каркаса (FR);

$TS = \{(ts, cm), cm \in CM\}$ — формальные описания тестов (ts) для компонентов с представлениями cm;

$TA = \{(cm, ac), cm \in CM, ac \in AC)\}$ — двусторонние связи трассировки между программными компонентами и компонентами архитектуры AR;

$TL = \{(fr, lc), fr \in FR, lc \in LC)\}$ — двусторонние связи трассировки между элементами каркаса для составления компонентов и их взаимосвязей;

ER и TF — соответственно, ER-модели БД для обработки ПС и СПС и двусторонние связи трассировки ее элементов с компонентами $cm \in CM$.

Ключевой проблемой инженерии семейства ПС является обеспечение вариабельности артефактов разработки членов СПС. Для решения этой проблемы предлагается теоретический аппарат вариабельности в артефактах СПС[1].

---

[1] См.: *Лаврищева Е. М.* Теоретические аспекты управления вариабельностью в семействах программных систем / Е. М. Лаврищева [и др.] // Вестник КНУ, серия физ.-мат. наук. 2011. № 1. С. 151—158; *Ее же.* Программная инженерия. Парадигмы, технологии и CASE-средства; *Ларищева Е. М., Слабоспицкая О. А.* Поход к построению объектно-компонентной модели семейства программных продуктов // Проблемы программирования. 2013. № 1. С. 21—33. *Lavrischeva E. M.* Software engineering. New disciplines and E-learning them for development of applied systems. Progressive Academic Publishing // European Journal of Engineering and Technology. 2015. № 3 (3). P. 36—63. URL: http://www.idpublications.org/ejet-vol-3-no-3-2015.

Определение 4.3. Модель вариабельности в артефактах СПС — это тройка

$$AV = \langle g_1; \langle g_t, tr_t \rangle; t = 2, ..., m; \langle p_t, tr_t \rangle; t = m + 1, ..., 5; AF \rangle, \quad (4.3)$$

$$AF = \langle AFt; REP; DBF \rangle; CAS = \cup_{t=1...4} Aft, \quad (4.4)$$

где g_t и p_t — подграфы G_t, описывающие артефакты, реализуемые артефактом типа m с идентификатором id_m и реализующие его при разработке ПС/ПП; элементы $AV(id_m)$ — сужения соответствующих элементов SV;

$$REP = \langle RPR \cup RPA \cup RPE \rangle; RPE = CM^* \cup TS^*,$$

где $AFt = \{f(aft)\}$ — формальное представление артефактов СПС типа $t = 1,4$ (aft); RPR, RPA, RPE — репозитории ресурсов СПС типа t: документов требований к ПС ($t = 1$), типичных архитектур *ПС* ($t = 2$) и программных ресурсов ($t = 3$); DBF — физическое представление БД для обработки ПС — членами СПС; CM^*, TS^* — множества программных КПИ, описанных в CM в выражении (4.2), и тестов для них.

4.4.1. Функции управления вариабельностью

Вариабельность — способность семейства ПС, отдельной системы или артефакта к расширению, изменению, приспособлению или конфигурированию с целью использования в ПрО ISO/IEC FDIS 24765: 2009.

Вариабельность обеспечивается на уровнях формирования требований к ПС, МХ СПС, архитектуры, документации, тестов и т.п. В целом вариабельность может быть реализована как в СПС, так и в конкретных ПС. В первом случае обеспечивается вариабельность производственной линии, которая поддерживает существование семейства как множества ПС. Она касается множества базовых элементов любой природы (программных и непрограммных), которые содержатся в репозитории. Во втором случае обеспечивается вариабельность составляющих ПС, что обусловливает эволюционность ПС после ее порождения и «отлучения» от СПС.

При управлении вариабельностью задаются:

- точка вариантности в формальном представлении элементарного артефакта СПС и способы реализации его для ПС;
- вариант — элементарный артефакт СПС одного типа с соответствующим ему вариантным артефактом, который представляет собой его описание в ПС;
- ограничение — предикат, определенный на декартовом произведении декартовых множеств точек вариантности и вариантов;
- зависимость — предикат, который определяет допустимые взаимосвязи между точкой вариантности и множеством вариантов.

Адаптация известного цикла управления Деминга к современным представлениям эффективной организации программной индустрии позволяет выделить в процессе разработки СПС четыре функции количественного управления вариабельностью:

1) F_1— планирование реализации вариабельности в артефактах СПС (на уровнях инженерии ПрО и инженерии приложений);

2) F_2 — реализация вариабельности в артефактах СПС;

3) F_3 — мониторинг состояния СПС в аспекте вариабельности;

4) F_4 — актуализация СПС по результатам мониторинга.

Для унификации управления вариабельностью в процессах инженерии ПрО и инженерии приложений введены следующие требования R (*Requirements*), определяющие функций F_1—F_4:

1) R_1 — обоснованность — наличие объективных оснований принятия решений для F_1—F_4;

2) R_2 — согласованность — одинаковость способа выработки и реализации решений на всех уровнях абстракции и на этапах процесса разработки СПС;

3) R_3 — масштабность — независимость способа выработки и реализации этих решений от объема функциональных возможностей, охватываемых СПС;

4) R_4 — трассировка отслеживания связей вариабельности на всех уровнях абстракции и на всех этапах процесса разработки СПС;

5) R_5 — визуализация вариабельности и связей между ними.

Обеспечение сформулированных требований R_1—R_5 для функций F_1—F_4 требует модельную среду для их реализации.

Необходимым элементом этой среды должна быть специальная **интегрированная модель вариабельности** в СПС (*VM*), которая позволит, по данным о текущей или прогнозируемой структуре СПС, определять оценку уровня вариабельности и степени ее соответствия потребностям ПрО по относительной шкале.

Для обеспечения сформулированных требований R_1—R_4 для функций F_1—F_4, принцип построения ортогональной модели вариабельности применен к модели вариабельности в структуре СПС. Именно из моделей *SV* (формулы (4.1, 4.2)) и *AV* (формулы (4.3, 4.4)) изъяты вершины всех уровней, которые подают общие черты всех ПС со СПС (т.е. вершины, не связанные в соответствующих графах ни одними связями вариантного подчинения).

Полученные в результате двумерные графы распределены на множества всех точек вариантности в СПС (*VP*) и вариантов для них (*VAR*) вместе со сбереженными связями трассировки из множеств *TF*, *TA*, *TD* (см. формулу (4.2)). В этих множествах выделены «фрагменты», соответствующие типам отображенных артефактов — уровням моделей *SV* (4.2).

Дополнительным требованием решения задач по управлению вариабельностью являются полученные «блоки» третьего измерения количественных оценок вариабельности (на уровне, соответствующему блоку с моделью *SV* и некоторой шкалы отношений, данные для которых предоставляют этот блок). Учитывая необходимость постоянного соблюдения баланса между множеством точек вариантности и вариантов в СПС, предусмотрено два класса оценок вариабельности СПС — уровни и соответствия потребностям СПС. Соотношение между заданными измерениями представлены в интегрированной модели вариабельности (рис. 4.17).

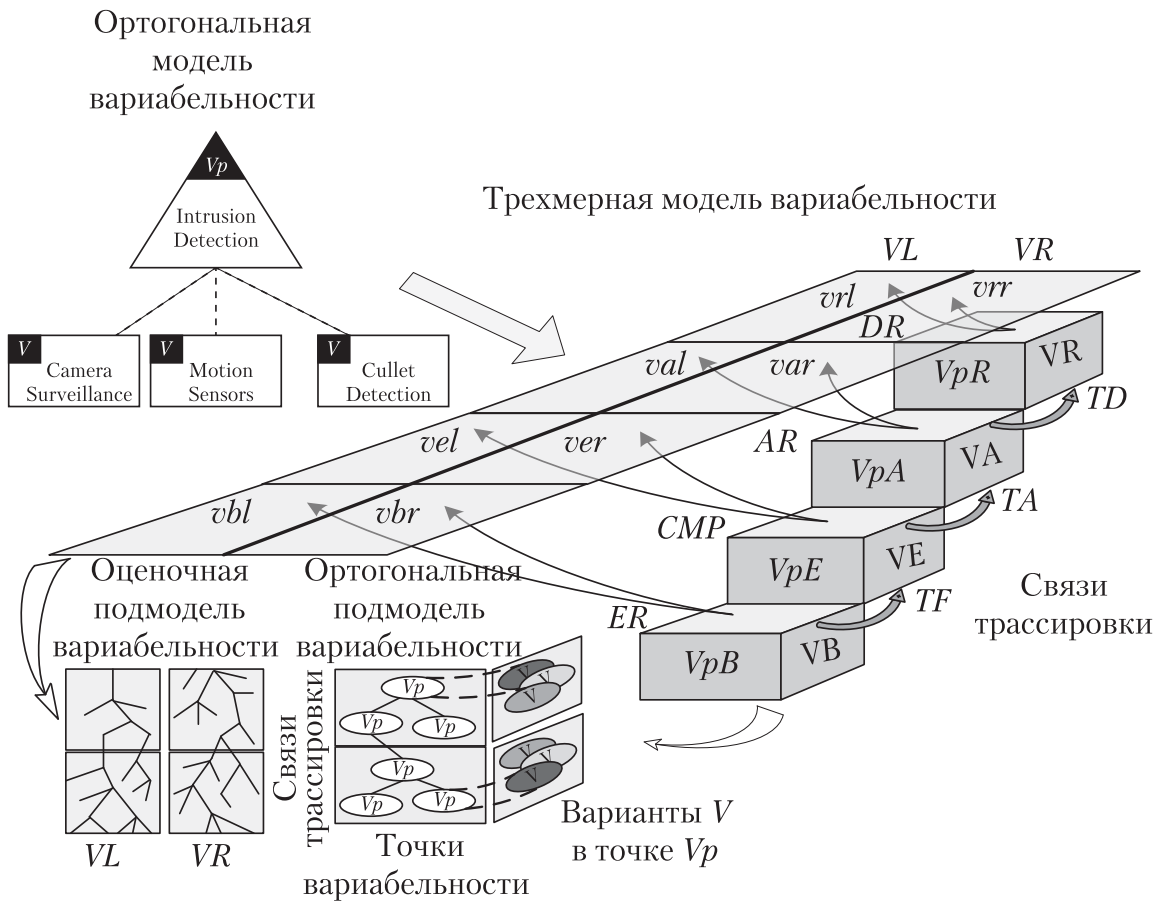


Рис. 4.17. **Структура интегрированной модели вариабельности**

Определение 4.4. Интегрированная модель вариабельности — это структурированный кортеж[1]

$$VM = \langle EVM;\ VP;\ VAR\rangle;\ EVM = \langle VL,\ VR\rangle; \tag{4.5}$$

$$VP = \langle VpR,\ VpA,\ VpE,\ VpB\rangle;\ VAR = \langle VR,\ VA,\ VE,\ VB\rangle; \tag{4.6}$$

$$VL = \langle vrl, val, vel, vbl\rangle,\ VR = \langle vrr, var, ver, vbr\rangle,$$

где *VpR*, *VpA*, *VpE*, *VpB* — множества точек вариантности в структуре СПС, которые подают, соответственно, характеристики ПС, компоненты архитектуры, элементы каркаса и поля таблиц БД (*VpB*), с сохраненными связями трассировки, ограничения (*Constr*) и зависимости (*Dep*) в формуле (4.1);

VR, *VA*, *VE*, *VB* — множества вариантов для точек вариантности из множества (4.5), на которых сохранены связи трассировки с ограничениями (*Constr*) и зависимостями (*Dep*), указанными в формуле (4.2);

VL — вложенная оценочная модель уровня вариабельности, детализированная типами ее проявлений — соответственно, в требованиях (*vrl*), компонентах архитектуры (*val*), программных артефактах (*vel*), обрабатываемых данных (*vbl*);

VR — аналогичная оценочная модель уровня соответствия вариабельности в СПС потребностям ПрО из сферы охватывания СПС, детализирован-

[1] *Лаврищева Е. М., Слабоспитская О. А.* Подход к построению объектно-компонентной модели семейства программных продуктов (укр.) // Проблемы программирования. 2013. № 3. С. 14—26.

ная соответственно для требований (*vrr*), компонентов архитектуры (*var*), программных артефактов (*ver*), обрабатываемых данных (*vbr*).

Для представления фрагментов (см. выражение (4.5)) вложенной оценочной модели предлагаются три универсальные модели преимуществ. Выбор среди них определяется текущей ситуацией оценивания вариабельности в процессе разработки СПС.

В частности, индивидуальное стохастическое прогнозирование вариабельности с высокой степенью неопределенности ее факторов (в начале этапов процесса инженерии ПрО и отношений), наиболее целесообразное с помощью байесовской сети. Для индивидуального или коллективного выбора оптимального среди нескольких вариантов решений из планирования или реализации вариабельности на этих этапах наиболее пригодна аналитическая иерархия Т. Саати.

Коллективное оценивание действительного уровня вариабельности и степени ее соответствия потребностям ПрО в конце этапов инженерии ПрО и инженерии приложений может эффективно выполняться посредством дерева ценности. Для сохранения преемственности получаемых оценок в процессе разработки СПС предусматривается согласование отмеченных моделей путем сопоставления им одного набора терминальных вершин.

В частности, индивидуальное стохастическое прогнозирование вариабельности с высокой степенью неопределенности ее факторов (в начале этапов процесса инженерии ПрО и инженерии отношений), наиболее целесообразно с помощью *байесовской сети*. Для индивидуального или коллективного выбора оптимального среди нескольких вариантов решений по планированию или реализации вариабельности на этих этапах наиболее пригодна аналитическая иерархия Т. Саати. Наконец, коллективное оценивание действительного уровня вариабельности и степени ее соответствия потребностям ПрО в конце этапов процесса инженерии ПрО и инженерии приложений может эффективно выполняться с помощью *дерева ценности*. Для сохранения преемственности получаемых оценок в процессе разработки СПС предлагается согласовать отмеченные модели путем сопоставления им набора терминальных вершин. Его состав отвечает структуре унифицированного паспорта ресурса $CT(rs)$ (см. выражение (4.5)), который и предоставляет фактические данные для их оценивания.

Главная задача поддержки функций F_1—F_4 состоит в предоставлении оценок вариабельности для решения и их выполнения. Тогда модель *VM* сможет выполняться только вместе с адекватной моделью самого СПС с отмеченными данными.

Известны два типа подходов к моделированию вариабельности. При одном из них предлагается интегрировать вариабельность в существующие модели (*FD* или *UML*), а при другом — строить специальные модели вариабельности.

Модели вариабельности, которые предлагаются в первом подходе, ограничены применением только для функции F_3 — проверки соответствия структуры СПС потребностям ПрО. К тому же они используют принципиально несопоставимые нотации для представления вариабельности артефактов СПС разных типов, которое существенно осложняет обеспечение требований согласованности и трассировки.

Среди специальных моделей вариабельности единственной известной моделью, практически лишенной последнего недостатка и поэтому избранной для преобразования в *VM*, является так называемая ортогональная модель вариабельности (*OVM*). Соответствующая ей модель СПС должна согласованно задавать состав и взаимосвязи всех проявлений вариабельности — как в структуре СПС, так и во всех артефактах процессов инженерии ПрО и инженерии приложений.

Таким образом, в состав модельной среды для функций F_1—F_4 вносится трехмерная модель вариабельности, которая сопоставляется измерениям ее ортогональной модели и вложенной оценочной модели.

Дальнейшая конструктивизация модели M_{var} с помощью метода ОКМ преобразует ее в объектно-компонентную модель семейства ПС/ПП. Метод ОКМ отражает принцип сборки ПС/ПП из КПИ. Он основан на обобщении понятия объекта с помощью теории Фреге и установлении теоретической связи между объектным анализом ПрО и компонентным методом создания ПС. Метод предполагает логико-математическое моделирование доменов ПрО с помощью четырех специальных графов ее объектов. Для формирования и использования графа ПрО используется алгебра объектного анализа, внешняя внутренняя компонентная алгебра и алгебраическая система преобразования неэквивалентных ТД, передаваемых между разнородными объектами, взаимодействующими в структуре ПС[1].

Модель вариантных характеристик M_{var} последовательно преобразуется в объектную модель семейства ПС/ПП на четырех уровнях проектирования, выделенных в ОКМ[2]. *Объектная модель* — четверка графов, последовательно конструктивизирующих друг друга:

$$OM = \langle (G_1, TR_1); (G_2^t, TR_2); (G_3^t, TR_3); G_4 \rangle,$$

где G_1 — граф объектов ПрО, создаваемый на обобщающем уровне проектирования; G_2 — представление МХ на характеристическом уровне; G_3 — архитектурно-компонентная модель семейства структурного уровня; G_4 — интерфейсная модель взаимодействия КПИ на поведенческом уровне.

Связи трассируемости TR_t в *OM* сопоставляют объектам функций моделируемых СПС из ПС/ПП и обрабатываемых ими данных (т.е. вершинам графа G_1) методы и данные (отображаемые вершинами графов G_2 и G_3), необходимые для взаимодействия этих объектов в ПС/ПП.

В поддержку сборки ПС/ПП, моделируемых с помощью *OM*, предложено ее дальнейшее преобразование в компонентную модель. Его суть — реализация методов объектов, выделенных в *OM*, за счет КПИ с соответствующими интерфейсами[3]. При этом терминальным объектам в G_3 и их интерфейсам в G_4 соответствует только один терминальный КПИ.

---

[1] См.: *Лаврищева Е. М., Слабоспитская О. А.* Технология моделирования изменяемых программных продуктов и систем // XII Международная научно-практическая конференция «Teoretical and applied aspect of program systems devepopment — 2015». Киев. 23—26 ноября 2015 г.

[2] См.: *Лаврищева Е. М.* Теория объектно-компонентного моделирования изменяемых программных систем // Труды ИСП РАН, 2015. URL: http://www.ispras.ru/preprins/docs/prep_29_2016.pdf.

[3] См.: *Lavrischeva E.* Formal Fundamentals of Component Interoperability in Programming // Cybernetics and Systems Analysis. 2010. Vol. 46. № 4. P. 639—652.

Модель представляет собой четверку:

$$CM = \langle RC; In; ImC; Fim; D \rangle,$$

где RC — терминальные КПИ, соответствующие терминальным объектам модели OM; In — множество интерфейсов КПИ, в параметрах которых задаются точки вариантности; ImC — множество реализаций терминальных КПИ в заданной среде; $Fim(\cdot)$ — множество функций преобразования входных параметров интерфейсов; D — множество элементов данных из сигнатур интерфейсов терминальных КПИ.

Наконец, *модель семейства* ПС/ПП имеет вид

$$M_{SPF} = \langle [(OM, CM)]; KPV; PRG; RPC; M_{FM}; M_{var},; MK \rangle,$$

где OM — объектная модель; CM — компонентная модель; M_{FM} — традиционная МХ семейства ПС/ПП; KPV — множество ГоР/КПИ; PRG — предикат принадлежности к KPV; RPC — сборочный предикат, определяющий операции сборки КПИ/ГоР в семействе; MK — модель конфигурации.

Разработка вариабельных ПС/ПП в их семействе осуществляется на основе описанных моделей путем управления вариантами ПС/ПП за счет конфигурирования КПИ/ГоР согласно универсальной модели M_{var} либо ее объектно-компонентному уточнению (OM, CM).

4.4.2. Подход к разработке вариабельной СПС из КПИ

Анализ интенсивности работ по созданию всемирных профильных библиотек разных артефактов систем и их онтологий[1] позволяет сделать вывод о том, что имеется устойчивая тенденция к развитию теории и практики инженерии изготовления новых ПрО, модели которых должны интегрироваться в контексты накопленных мировых нормализованных информационных ресурсов. Рассматриваются ГоР и КПИ, которые экономят время и стоимость создаваемого продукта.

К базовым артефактам систем, которые используются для накопления и использования в СПС, относятся следующие типы ГоР:

- типовые задачи ПрО;
- типовые классы объектов, их атрибуты и методы;
- типовые требования к разработке;
- пред-, постусловия и инварианты;
- специальные КПИ (фреймы, паттерны, диаграммы UML и др.);
- типовые показатели качества и метрики оценки качества ПП.

Обобщая понятие ГоР, можно сказать, что ими могут быть модели входных и выходных продуктов и систем на процессах разработки (табл. 4.1). ГоР и КПИ разрабатываются и накапливаются в библиотеках (репозиториях) и выбираются при необходимости.

Главным в разработке СПС из ГоР являются процессы инженерии ПрО и приложений из КПИ, аналогично SPLE (рис. 4.18).

---

[1] См.: *Чернецки К.*, *Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.

Таблица 4.1

Артефакты продуктов на процессах разработки ПС

Процесс разработки	Категории артефактов продуктов
Инженерия требований	Участники формулируют требования, элементы требований. Варианты использования и интерфейсы систем
Проектирование	Архитектурные составляющие: спецификации компонентов, объектов, интерфейсов
Кодирование	Спецификация компонентов и объектов
Тестирование	Элементы тестов, варианты интегрированных тестов, методики тестирования
Сопровождение (управление изменениями)	Спецификация прогнозируемой вариантности компонентов, кода, элементов тестов
Управление конфигурацией	Спецификации выпусков вариантов и версий проекта, компонентов, объектов, документации

Ключевая проблема методологии СПС — обеспечить вариабельность артефактов процесса разработки семейства и его членов. Методология разработки СПС включает теоретический аппарат согласованного моделирования вариабельности в структуре и артефактах СПС, а также функции управления вариабельностью в заданной среде.

На схеме, идентичной SPLE, приведена собственная трактовка двух проссесов: инженерия домена и инжнерия приложений

Процесс инженерии домена (ПрО) предназначен для определения общих и отличительных особенностей ПС в составе СПС с позиций функциональных и нефункциональных требований. Создаются артефакты, которые реализуют общие характеристики ПС и обеспечивают вариабельность СПС, свойственную множеству ПС. Артефакты — это ГоР и КПИ, подвергаются тестированию и образуют платформу СПС. Эта платформа наряду с программными компонентами включает модели требований, архитектуры, тестов и т.п. При построении систем обработки данных, характерными признаками которых являются сложные структуры данных ПрО и взаимодействие с СУБД для работы с большими данными, в состав платформы входят модели данных ПС.

Инженерия приложений предназначена для порождения ПС на общей платформе из множества ГоР и КПИ путем связи заложенной в них вариабельности и в соответствии с требованиями для каждой ПС.

Процессы инженерии выполняются параллельно, взаимодействуют через рабочие продукты репозитория среды разработки и обмениваются согласованным способом.

Процесс организационного управления координирует работы на процессах инженерии ПрО и инженерии приложений, а также обеспечивает планирование, контроль работ по управлению вариабельностью создаваемых СПС.

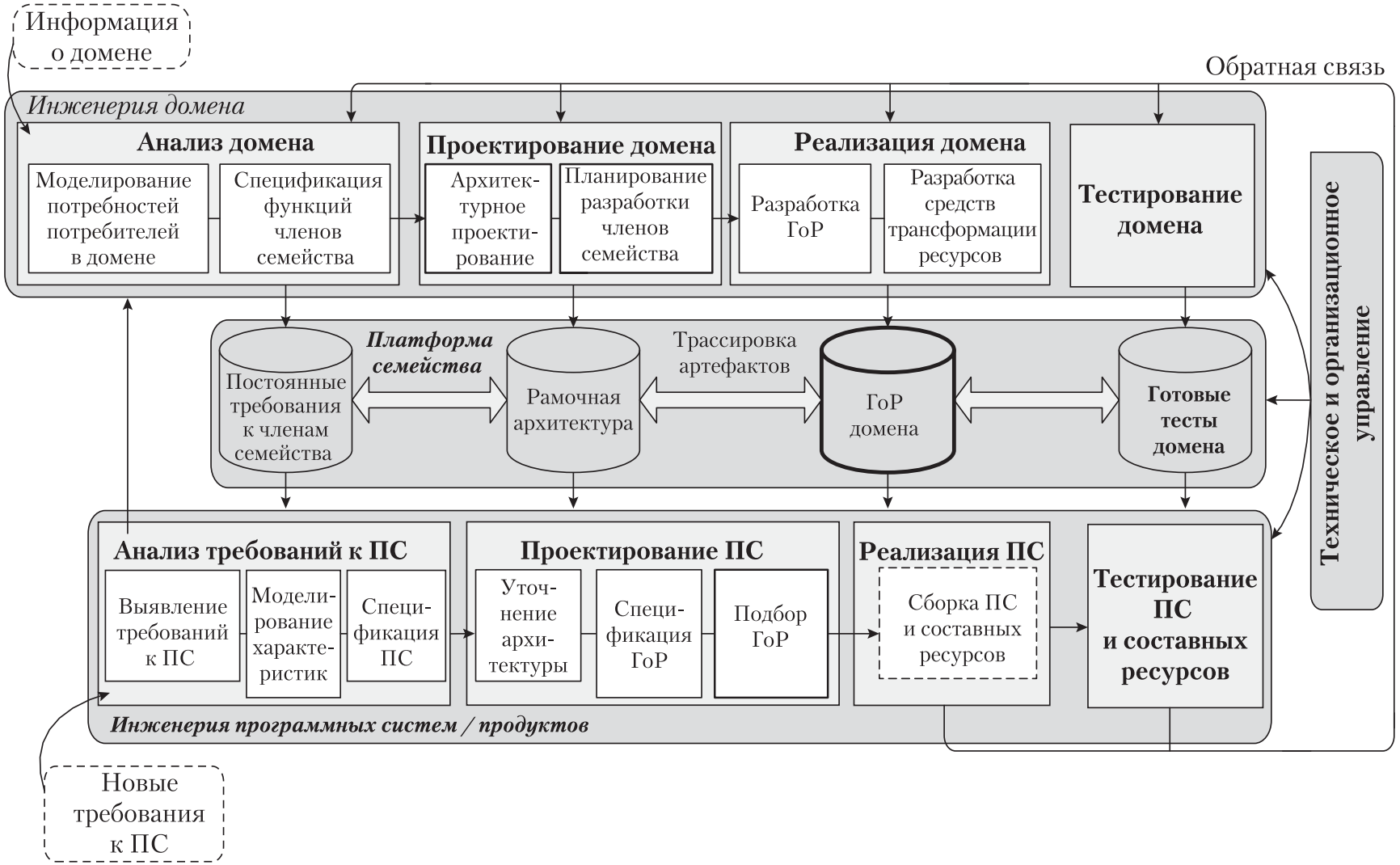


Рис. 4.18. **Процессы инженерии ПрО и приложения**

4.4.3. Процесс сборки КПИ в современных средах

В инженерии приложений и доменов имеется процесс конфигурационной сборки КПИ и GoP на основе модели вариабельности FM и ПС. В сборку входит процесс проверки павильности использования GoP FM на соответствие требованиям заказчика. После этого принимается решение о достаточности или недостаточности его использования в СПС[1].

Впервые идея сборки программ реализована в системе АПРОП. Она описана в монографии «Связь разноязыковых модулей в ОС ЕС» (1982). В ней приводится теория построения разноязыковых отдельных модулей/КПИ, их сборка в ПС и трансформация несовместимых ТД связываемых разнородных КПИ через интерфейсный посредник. В нем генерируются операции сборки и функции преобразования незквивалентных ТД. Идея отечественной сборки разнородных объектов получила развитие в системе COBRA через *stub* и *skeleton* (1994). Этот метод стал на долгие годы основным во многих системах общего назначения (IBM, MS.Net, CORBA и др.).

В настоящее время имеется много систем сборки. К средствам поддержки интеграции или сборки разных программных элементов относятся Make, Apache Ant, Apache Maven и др. Рассотрим их.

Make — это кросс-платформенная система автоматизации сборки ПС из исходного кода. Она генерирует файлы управления сборкой, например Makefile в системах Unix для сборки посредством make.

Apache Ant — Java-утилита для автоматизации процесса сборки ПП. Ant — платформонезависимый аналог UNIX-утилиты Make, но с использованием языка Java, приспособленным для Java-проектов. Самая важная непосредственная разница между Ant и Make состоит в том, что Ant использует XML для описания процесса сборки и его зависимостей, тогда как Make имеет собственный формат Makefile. По умолчанию XML-файл, называемый build.xml, осуществляет сборку.

Apache Maven — программный инструмент для управления (*management*) Java, проектами и сборщика (*build*) разных программ. По принципам функционирования он подобен Apache Ant, но имеет более простую *build*-модель конфигурации, которая базируется на формате XML. Двигатель ядра может динамически загружать плагины из репозитория, который обеспечивает доступ ко многим версиям разных Java-проектов с открытым кодом от Apache и других организаций и отдельных разработчиков.

Gradle — система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но в отличие от них представляет DSL на языке Groovy вместо традиционной XML-подобной формы представления конфигурации проекта.

Пример сборки КПИ приведен в Приложении 2 (сборка систем из КПИ в MS.INT).

---

[1] См.: *Лаврищева Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства.

Контрольные вопросы и задания

1. Сформулируйте основные понятия вариантов КПИ и СПС.
2. Перечислите четыре функции управления вариабельностью.
3. Дайте определение модели вариабельности в структуре.
4. Определите элементы вариабельности в артефактах.
5. Назовите элементы интегрированной модели вариабельности.
6. Определите графовую модель вариабельности.
7. Что такое конфигурационная сборка артефактов?
8. Определите сущность технология сборки по модели FM.

Глава 5
ОПРЕДЕЛЕНИЯ ХАРАКТЕРИСТИК СИСТЕМ МЕТОДАМИ АНАЛИЗА И ИЗВЛЕЧЕНИЯ ЗНАНИЙ

В проблематике разработки ПП и семейств систем СПС сложилось достаточно много формальных методов определения характеристик систем, а также методов при разработке продукта, начиная с требований к продукту и заканчивая его изготовлением, а также методов реинженерии и извлечения знаний из систем для построения моделей внешних и внутренних характеристик ПП[1].

Одной из важных проблем является представление и извлечение знаний из систем и используемых данных. Знания представляются в конкретной форме и хранятся в базах знаний. Языком представления знаний являются языки описания логических моделей и система предикатов первого порядка. К типичным моделям представления знаний относятся: логическая модель, продукционная модель, модель, основанная на фреймах, и модель семантической сети. Они будут описаны ниже[2].

К методам анализа и извлечения знаний относятся: кластерный и факторный анализ, логлинейный анализ, каноническая корреляция, пошаговая линейная и нелинейная регрессия, анализ временны́х рядов и деревьев классификации.

В отличие от традиционной проверки гипотез разведывательный анализ данных применяется для нахождения связей между переменными в ситуациях, когда отсутствует (или недостаточно) априорное представление о природе этих связей. При таком анализе учитывается и сравнивается большое число переменных, а для поиска закономерностей используются разнообразные средства. К инструментам поиска и извлечения знаний относятся Oracle Data Mining Suite, Scenario, WizWhy и др.

В русле развития логических подходов и создания *псевдофизических логик* — специальных логик для описания восприятия человеком процессов, протекающих в реальном мире, рассмотрена работа Д. А. Поспелова «Представление знаний. Опыт системного анализа» (опубликована в 1985 г. в ежегоднике «Системные исследования»). Псевдофизические

---

[1] См.: *Колесник А. Л.* Модели и методы разработки семества вариантных программных систем : автореф. дис. ... канд. техн. наук. Киев, 2013; *Лаврищева Е. М., Коваль Г. И* Теоретические аспекты управления вариабельностью в семействах программмных систем // Вестник КНУ им. Тараса Шевченко. Серия физ.-матем. наук. 2011. № 1. С. 45—57.

[2] См.: *Дюк В. А.* Data Mining — интеллектуальный анализ данных; *Елманова Н.* Введение в Data Mining. КомпьютерПресс, 2003; URL: https://sites.google.com/site/upravlenieznaniami/tehnologii-upravlenia-znaniami/data-mining#TOC-Classification; http://www.statistica.ru.

логики включают пространственные логики, логики действий и норм, причинно-следственные логики. Они обладают рядом отличительных особенностей: логика отношений; логика на нечетких шкалах; аксиомы утверждений, вытекающие из восприятия мира человеком; связи, существующие между отдельными логиками. В результате были разработана прикладная семиотика, рассматривающая вопросы применения знаков и знаковых систем в системах представления, обработки и использования знаний при решении различных практических задач[1].

5.1. Моделирование функциональных и нефункциональных характеристик ПП

Множество конкретных целевых нефункциональных характеристик семейства ПС определяется лишь после того, как достаточно полно определены функциональные потребности ПрО и архитектурные возможности их реализации. Нефункциональные характеристики (НФ) могут быть декомпозированы по уровням и отдельным объектам архитектуры. Такая декомпозиция будет содействовать *оптимизации архитектуры* с учетом общих функциональных и нефункциональных требований к СПС, а также построению ПС — членов семейства с требуемым качеством системы.

Нефункциональные характеристики СПС — это характеристики качества и ресурсов проекта — время, трудозатраты, а также программно-технические ограничения эксплуатации (сетевое, Интранет, Интернет), оборудование, производительность ПО и т.п.

Модель НФ СПС должна включать *модель ресурсов* СПС. В ней задаются *ресурсные ограничения* (разработки СПС и ПС и выполнения ПС), в рамках *адекватной* модели качества.

Для обеспечения соответствия требованиям к моделям нефункциональных характеристик СПС в НФ привлекаются методы интеллектуального анализа данных, например метод анализа иерархий и байесовские сети[2].

5.1.1. Метод анализа иерархий

Метод анализа иерархий перспективен в инженерии качества. С его помощью можно оптимальным способом распределять целевые требования к качеству ПС по множеству ее компонентов в иерархической функциональной архитектуре и оценивать взнос разных компонентов в достижение качества. Метод предоставляет также преимущества в построении сложных иерархических моделей качества, обеспечивая возможности *сочетания разных характеристик* в одной модели, распределения их по приоритетам и при необходимости определения интегрального показателя качества ПС.

---

[1] См.: *Сахаров А. А.* Принципы проектирования и использования многомерных баз данных (на примере Oracle Express Server) // СУБД. 1996. № 3; *Codd E. F., Codd S. B., Salley C. T.* Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. Codd & Date, Inc., 1993.

[2] См.: URL: http://www.statistic.ru.

Анализ иерархии проводится в три шага.

Шаг 1. Попарное сравнение *элементов* одного уровня иерархической структуры (компонентов архитектуры, характеристик качества и т.п.), начиная с уровня 2, с позиций их важности по отношению к каждому элементу высшего уровня (в соответствии с установленными связями элементов, определенных в иерархической структуре системы).

Кроме того, преимущества могут определяться по принципу «преобладает N раз» (2, 3 и до 9 раз), а также устанавливаться путем непосредственного измерения ценности элементов или интенсивности их использования (с нормированием значений таким образом, чтобы их сумма равнялась единице).

По результатам попарного сравнения элементов одного уровня по отношению к элементу высшего уровня строится квадратная матрица A размером $nn \times x_0 \times x_0$ элементов

$$a_{11} = a_{22} = ... = a_{nn} = 1, a_{ij} = \frac{1}{a_{ji}} x_0 \ (u, j = 1, ..., n; n = k, l, m),$$

где n — количество сравниваемых элементов структуры; k, l, m — количество элементов на уровне 2, 3 соответственно.

Поскольку попарное сравнение для предоставления преимущества тому или другому элементу иерархии является субъективным процессом, суждением лиц, которые принимают решение (ЛПР), они могут быть несогласованными, т.е. такими, для которых не выполняются условия транзитивности и кардинальности. Первое условие означает, что для любых компонентов x, $в$, z и отношения R, из того, что xRy и yRz, вытекает xRz.

Второе условие заключается в выполнении соотношения $a_{ij} = \frac{a_{i-1,j}}{a_{i-1,i}} x_0$ при $u > 1$, $u < j$, из которого вытекает линейная зависимость между строками матрицы.

Шаг 2. Определение относительных оценок всех элементов одного уровня иерархии — *коэффициентов относительного веса*, w_i, $u = 1, ..., n$ — которые образуют вектор относительного веса W размерности n (или вектор приоритетов).

Матрица парных сравнений может быть согласованной или несогласованной.

Если матрица согласованная, относительный вес i-го элемента структуры вычисляется по формуле

$$w_i = \frac{\sum_{j=1}^{n} a_{ij}}{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}}, \sum_{i=1}^{n} w_i = 1.$$

Но поскольку при больших размерах матриц сложно сразу определить, согласованная матрица или нет, а также для получения более точных относительных оценок элементов иерархии выполняется вычисление *собственного вектора* и *собственных значений матрицы* из уравнения

$$A \cdot W = \begin{pmatrix} 1 & a_{12} & \dots & a_{1n} \\ \frac{1}{a_{12}} & 1 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ \frac{1}{a_{1n}} & \frac{1}{a_{2n}} & \dots & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} = \lambda_{\max} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} = \lambda_{\max} \cdot W,$$

где $\lambda_{\max}$ — максимальное собственное значение матрицы A порядка n; w_i — коэффициенты относительного веса элементов матрицы A, a; $W = (w_1, w_2, ..., w_n)$ — главный собственный вектор, который означает $\lambda_{\max}$.

Полученные элементы собственного вектора нормализуются значениями $\alpha = \sum_{i=1}^{n} w_i$, которое обеспечивает единое решение при $\sum_{i=1}^{n} w_i = 1$.

Если матрица A имеет $n - 1$ собственных значений λ, равных нулю, и одно $\lambda_{\max} = n$, то она является согласованной. В случае нарушения согласованности матрица будет иметь несколько собственных значений $\lambda \neq 0$, однако при небольшой несогласованности мнений экспертов одно из собственных значений будет существенно больше других и будет показывать уровень несогласованности матрицы сравнений.

Чтобы оценить уровень согласованности суждений специалистов, используются индекс согласованности *CI* и коэффициент согласованности *CR*, которые определяются по формулам

$$CI = \frac{\lambda_{\max} - n}{n-1}, CR = \frac{CI}{E(CI)},$$

где $E(CI)$ — математическое ожидание *CI* для случайно сложенной матрицы парных сравнений A ($nn \times x_0$), получено Т. Саати по данным 50 000 экспериментов (табл. 5.1).

Таблица 5.1

Индексы согласованности *CI* для матриц размером *n*

N	1	2	3	4	5	6	7	8	9	10
E(*CI*)	0	0	0,52	0,89	1,11	1,25	1,35	1,40	1,45	1,49

Критерий приемлемости парного сравнения элементов в матрицах размером $n_3 \geq x_0 \geq x_0$ является таким: для $n = 3$, а для $n > 3$ $CR < 0{,}1$. В случае превышения установленного уровня матрица парных сравнений строится заново.

Шаг 3. На этапе иерархического синтеза относительные взвешивающие коэффициенты элементов иерархии *ї структуры каждого уровня, которые являются *локальными* (в пределах уровня), дальше агрегируются с соответствующими относительными взвешивающими коэффициентами элементов, которые находятся уровнем выше. В результате определяются *общие взвешивающие коэффициенты*, которые задают относительную важность элементов структуры более низких уровней в сравнении с элементом наивысшего уровня (или элементами выбранного промежуточного уровня в иерархической структуре).

Например, рассмотрим иерархическую структуру ПС с такими уровнями:

1 — *целевая характеристика* качества Q_{nc};

2 — *функциональные характеристики* ПС F_1, ..., F_k;

3 — множество программных *композиций* компонентов (программных применений) Z_1, ..., Z_k, предназначенных для поддержки выполнения функций;

4 — множество независимых *повторно используемых компонентов* архитектуры ПС (модулей) M_1, ..., M_m.

В методе МАИ определяются:

1) *векторы локальных приоритетов* функций, композиций и компонентов, а именно

- $U = (u_1, u_2, ..., u_k)$, вектор коэффициентов относительного веса функций в Q_{nc},
- $V_i = (v_{i1}, v_{i2}, ..., v_{il})$, $i = 1, ..., k$, векторы коэффициентов относительного веса композиций для каждой функции,
- $W_i = (w_{i1}, w_{i2}, ..., w_{im})$, $u = 1, ..., l$, векторы коэффициентов относительного веса компонентов для каждой композиции;

2) *векторы общих* (*глобальных*) *приоритетов* композиций и компонентов. Общий вес i-й композиции рассчитывается по формуле

$$V_i^* = \sum_{j=1}^{k} u_j \cdot v_{ij} x_0.$$

Для всех композиций и по отношению ко всем функциям вектор общих взвешивающих коэффициентов определяется так:

$$V^* = U \cdot \begin{pmatrix} V_1^* \\ V_2^* \\ ... \\ V_l^* \end{pmatrix}, (v_1^*, v_2^*, ..., v_l^*) = (u_1, u_2, ..., u_k).$$

Так же обозначается вектор общих весовых коэффициентов для всех компонентов, к которым обращаются композиции:

$$\begin{pmatrix} v_{11} & v_{12} & ... & v_{1l} \\ v_{21} & v_{22} & ... & v_{2l} \\ ... & ... & ... & ... \\ v_{k1} & v_{k2} & ... & v_{kl} \end{pmatrix}, (w_1^*, w_2^*, ..., w_m^*) = (v_1^*, v_2^*, ..., v_l^*) \begin{pmatrix} w_{11} & w_{12} & ... & w_{1m} \\ w_{21} & w_{22} & ... & w_{2m} \\ ... & ... & ... & ... \\ w_{l1} & w_{l2} & ... & w_{lm} \end{pmatrix}.$$

Полученные общие взвешивающие коэффициенты для множества функций, композиций и отдельных компонентов определяют значимость присутствия определенного атрибута качества в каждой из этих конструкций для обеспечения целевой характеристики качества ПС. Их значения могут далее использоваться при оценивании качества.

5.1.2. Моделирование характеристик качества байесовской сетью

Одним из перспективных подходов к моделированию качества ПС с целью последующего управления качеством является байесовский под-

ход[1], в котором обоснована целесообразность его применения для моделирования причинно-следственных связей факторов качества и их влияния на целевую характеристику качества ПС.

Подход приобрел практическую значимость лишь с появлением байесовских *сетей доверия* (*Bayesian Belief Network*), на базе которых строятся графические байесовские модели (БМ) качества.

Основные преимущества их использования при разработке СПС и отдельных членов семейства — поддержка *прогнозирования* заданного показателя ПС и *диагностики* наиболее достоверных причин (источников) его недостижения. К тому же аппарат байесовских сетей позволяет строить достаточно сложные модели (сотни вершин), которые не могут быть реализованы другими методами[2].

Хотя байесовские сети известны давно, широкое практическое применение они начали находить лишь с появлением эффективных алгоритмов распространения вероятности по сети, а также инструментальных средств построения байесовских сетей.

В основе теории байесовских сетей лежат формула Байеса и правило сети, которое является обобщением правила умножения вероятности и предоставляет средство вычисления совместимого распределения вероятностей случайных событий.

Пусть A_1, A_2, ..., A_n — случайные величины (СВ), связанные с наступлением случайных событий. Тогда по правилу сети

$$P(A_1A_2, ..., A_n) = P(A_1 \mid A_2, ..., A_n) \cdot P(A_2 \mid A_3, ..., A_n) \cdot ... \cdot P(A_{n-1} \mid A_n) \cdot P(A_n).$$

Байесовская сеть — это ориентированный ациклический граф, каждая вершина которого представляет собой событие, описанное случайной величиной (переменной) A_u, которая может находиться в нескольких состояниях (принимать несколько значений). С каждой вершиной априори связываются параметры, соответствующие закону распределения случайной величины. Дуга между любыми двумя вершинами A_u и A_j $(u \neq j)$ устанавливает причинно-следственную связь «A_u *вызывает* A_j». В сети справедливо отношение условной независимости вершин: вершина-потомок A_j не зависит от вершин-предков, которые находятся выше вершины-отца (т.е. A_u). Это позволяет упростить правило работы сети.

Пусть a_1, ..., a_n — множество всех возможных конфигураций значений множества СВ A_1, A_2, ..., A_n в вершинах БМ, а запись Предки (A_u) означает множество предков вершины A_u. Тогда распределение вероятностей для A_1, A_2, ..., A_n на множестве конфигураций $P(a_1, a_2, ..., a_n) = \prod_{i=1}^{n} P(a_i \mid Предки(a_i))$.

На рис. 5.1 дан пример БМ с пятью вершинами A_1, A_2, A_3, A_4, A_5, переменные в которых могут принимать лишь значения True (T) и False (F).

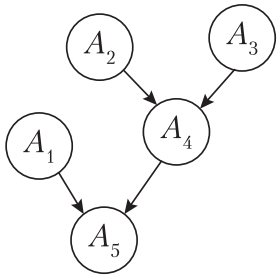
---

[1] См.: URL: https://sites.google.com/site/upravlenieznaniami/tehnologii-upravlenia-znaniami/data-mining#TOC-Classification.

[2] *Коваль Г. И.* Байесовские сети для оценки и прогнозирования качества ПО // Проблемы программирования. 2005. № 2. С. 15—23.

По правилам сети с учетом условной независимости A_5 от A_2 и A_3 имеем

$$P(A_5A_4A_3A_2A_1) = P(A_5 \mid A_4A_1) \cdot P(A_4 \mid A_2A_3) \cdot P(A_1) \cdot P(A_2) \cdot P(A_3).$$



$P(A_4 = T \mid A_2 = T, A_3 = T) = 0{,}4,\ P(A_4 = F \mid A_2 = T, A_3 = T) = 0{,}6$
$P(A_4 = T \mid A_2 = T, A_3 = F) = 0{,}3,\ P(A_4 = F \mid A_2 = T, A_3 = F) = 0{,}7$
$P(A_4 = T \mid A_2 = F, A_3 = T) = 0{,}6,\ P(A_4 = F \mid A_2 = F, A_3 = T) = 0{,}4$
$P(A_4 = T \mid A_2 = F, A_3 = F) = 0{,}5,\ P(A_4 = F \mid A_2 = F, A_3 = F) = 0{,}5$

$P(A_1 = T) = 0{,}8,\ P(A_1 = F) = 0{,}2$
$P(A_2 = T) = 0{,}6,\ P(A_2 = F) = 0{,}4$
$P(A_3 = T) = 0{,}7,\ P(A_3 = F) = 0{,}3$

$P(A_5 = T \mid A_1 = T, A_4 = T) = 0{,}7,\ P(A_5 = F \mid A_1 = T, A_4 = T) = 0{,}3$
$P(A_5 = T \mid A_1 = T, A_4 = F) = 0{,}3,\ P(A_5 = F \mid A_1 = T, A_4 = F) = 0{,}7$
$P(A_5 = T \mid A_1 = F, A_4 = T) = 0{,}8,\ P(A_5 = F \mid A_1 = F, A_4 = T) = 0{,}2$
$P(A_5 = T \mid A_1 = F, A_4 = F) = 0{,}3,\ P(A_5 = F \mid A_1 = F, A_4 = F) = 0{,}7$

Рис. 5.1. **Пример БМ с вероятностными значениями переменных**

Если значения $A_1 = A_2 = A_3 = T$ (с вероятностью $P(A_i) = 1$, $i = 1, ..., 3$), а значение $A_4 = T$ неизвестно, то получаем вероятность того, что $A_5 = T$, т.е. $P(A_5 = T \mid A_4 = T, A_3 = T, A_2 = T, A_1 = T) = P(A_5 = T \mid A_4 = T, A_1 = T) = 0{,}7$ БМ предоставляет полное совместимое распределение вероятности всех возможных событий, вычисляя априорную вероятность каждой комбинации значений всех переменных, которые представляют эти события. Когда значение какой-то переменной становится известным (очевидным фактом), первичное совместимое распределение вероятности вычисляется. С учетом этого факта наблюдается переходный процесс замены априорных вероятностей значений каждой переменной апостериорными, т.е. пересматриваются равные вероятности событий, которые моделируются байесовской сетью.

Таким образом, посредством БМ можно формулировать предположение о существовании зависимости между разными переменными, а затем последовательно «распространять» получаемые объективные данные наблюдений по всей сети.

Параметры каждой вершины БМ должны быть представлены таблицей вероятностей вершины (для дискретных СВ), функцией распределения вероятности (для непрерывных СВ) или детерминированной функцией от значений СВ в вершинах-родителях.

В проблематике моделирования НФ к СПС применимы также следующие методы:

- методы *анализа деревьев ошибок* (FTA) или деревьев событий (ETA);
- методы *сценарного подхода* (методы SAAM, ATAM, CDAV, FAAM), согласно которым архитектура должна покрывать все сценарии ее возможного использования, а также иметь высокие показатели качества с помощью характеристик интероперабельности (*interoperability*), расширяемости (*extensibility*), модифицируемости (*modifiability*) и т.п.

5.1.3. Неиерархические модели качества

Стандарт качества ISO/IEC 9126 включает большое количество измерений качественных показателей, однако они не устанавливают причинно-

следственных взаимосвязей, существующих между характеристиками и подхарактеристиками, измеряемыми атрибутами качества и множеством их прямых и побочных измерений.

Параллельно с разработкой иерархических моделей проводились исследования способов учета и представления каузальных взаимосвязей в проблематике качества ПС.

В 1999 г. А. Абраном и Л. Буглионе была предложена модель QEST (*Quality Factor + Economic, Social and Technical dimensions*). Эта трехмерная структура-оболочка позволяет одновременно представлять зависимость значений параметров продукта, которые отображают три аспекта его измерения (три точки зрения на продукт).

Три измерения модели образуют геометрическую фигуру — правильный четырехгранник (тетраэдр, рис. 5.2).

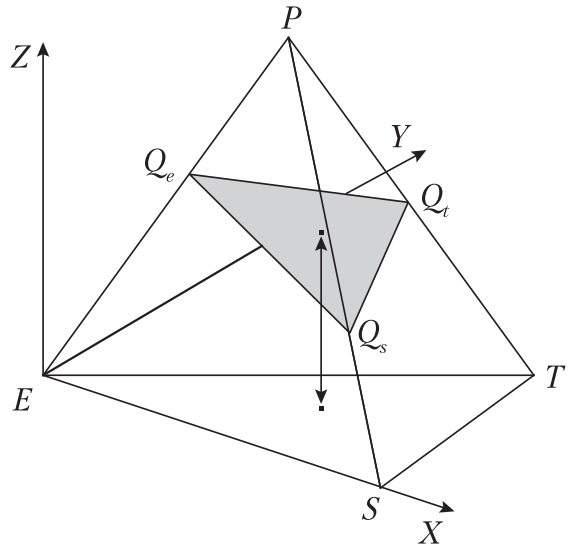


Рис. 5.2. **Модель QEST для оценки качества ПС**

Стороны тетраэдра представляют нормализованные значения измеренных величин по каждому из измерений, а вершина — нормализованное оптимальное значение для каждого измерения

Модель QEST приспособлена для совместимого представления количественных и качественных измерений показателей разрабатываемого ПП, которые включаются в интегральный показатель качества.

Измерения модели позволяют определить:

1) экономический аспект (*E*). Точка зрения менеджеров на стоимость и график разработки (поставки) продукта;

2) социальный аспект (*S*). Точка зрения пользователей на характеристики эксплуатационного качества (качества в использовании) продукта;

3) технический аспект (*T*). Точка зрения разработчиков на обеспечение, измерение и оценивание достижения требований к качеству продукта.

Вершина (*P*) тетраэдра описывает максимальный уровень качества продукта проекта.

Стороны правильного тетраэдра (пирамиды) равны единице. Все три измерения нормализованы на интервале {0,1}. Значения по измерению являются взвешенными суммами списка *n* значений нормализованных

коэффициентов, которые отражают каждую из трех точек зрения экспертов на продукт (номинальные оценки). Эти значения откладываются на сторонах пирамиды $\{Q_e, Q_s, Q_t\}$. Полученные точки соединяются линиями. Образуется сечение пирамиды, которое «отделяет» лучшие и худшие оценки качества продукта. В ходе выполнения проекта формируются «реальные срезы» эффективности проекта. Эффективность проекта измеряется с помощью трех геометрических объектов:

- расстояние от сечения до основы пирамиды. Чем больше расстояние от текущего среза до основы, тем выше эффективность;
- площадь сечения. Чем меньше площадь, тем выше эффективность;
- объем нижней части пирамиды. Чем больше объем под сечением, тем выше эффективность проекта.

В 2001 г. А. Абраном и Н. Кецеси была предложена схема GDQA (*Graphical Dynamic Quality Assessment*).

В этой схеме сначала определяется иерархическая модель качества ПС, которое включает декомпозицию артефактов в категории процессов, продуктов и ресурсов проекта и их приоритизацию, «взвешивание» и нормализацию для последующего сопоставления (рис. 5.3).

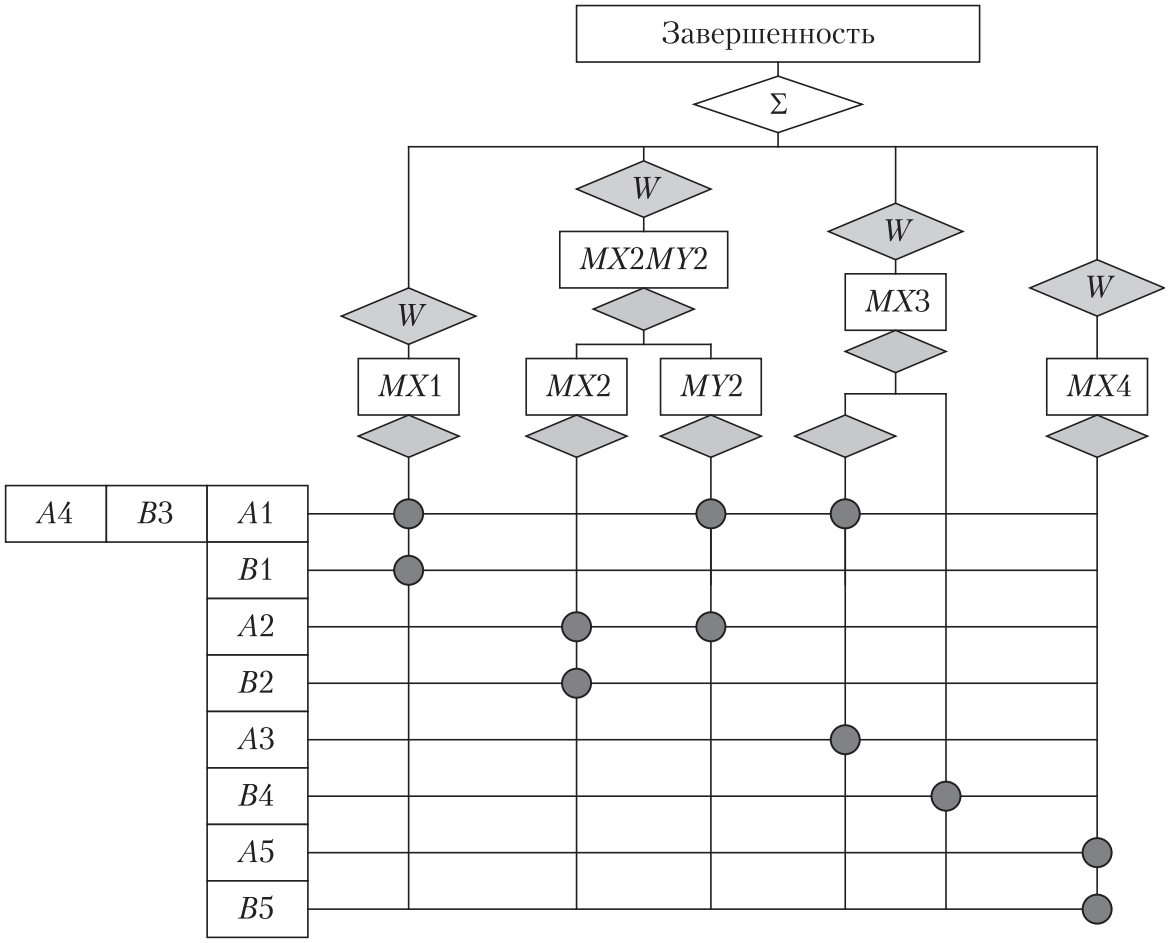


Рис. 5.3. **Схема GDQA для измерения «завершености ПП»**:

● — непосредственное измерение; ◆ — вычисление метрики;

W — взвешивание; Σ — суммирование

Дальше устанавливается взаимосвязь факторов качества, которые находятся на разных уровнях, поскольку некоторые атрибуты качества нижнего уровня могут вносить вклад в несколько характеристик верхнего уровня иерархии. После этого выполняется отображение атрибутов соответствующих измерений по метрикам качества, а затем определяются входные переменные, значение которых можно установить, анализируя проектные документы, код и отчеты о тестировании и т.п.

На приведенном рисунке дан пример применения схемы GDQA для подхарактеристики «завершенность» качества «надежность».

Круги на схеме указывают, для каких метрик используются те или другие собираемые данные.

5.1.4. Подход к оценке качества процессов линий

Ни один из описанных выше типов моделей и формальных методов применительно к моделированию семейства ПС не соответствует требованиям к моделям.

Наиболее близок удовлетворению установленных требований подход Prometheus[1].

Это целевой подход к измерению и оцениванию качества на линии ПП, которые интегрируют количественные и качественные методы моделирования.

Подход обеспечивает следующее:

- оценивание начиная с ранних стадий ЖЦ;
- эффективное обучение по ходу разработки нескольких вариантов/релизов продукта;
- уточнение модели качества проекта;
- интеграцию количественных и качественных методов;
- комбинирование разных контекстов индивидуальных взглядов на качество ПС (разработчиков, пользователей) и оцениваемых объектов (процессов, продуктов, ресурсов).

Подход включает три фазы: спецификация модели, применение модели для оценивания качества и «упаковка» добытых знаний по внешним характеристикам для накопления опыта по проектам. Схема выполнения шагов спецификации модели дана на рис. 5.4.

Целевые требования к качеству СПС определяются и специфицируются с применением методологии измерения «Цель — вопрос — метрика» (GQM, *Goal — question — measure*)[2].

Соответственно методологии они конкретизируют список измеряемых характеристик и подхарактеристик оцениваемых программных объектов семейства. Потом устанавливаются взаимосвязи между характеристиками (подхарактеристиками) и измерениями.

Рассматриваются два типа связей — *декомпозиция* характеристик (сверху вниз) и *влияние* уровня (значение) одной характеристики на другой.

---

[1] См.: *Осукра С., Саэки Ю.* Приобретение знаний : пер. с япон. М. : Мир, 1990.

[2] См.: *Codd E. F., Codd S. B., Salley C. T.* Providing OLAP (On-line Analytical Processing) to User-Analysts: An ITMandate.

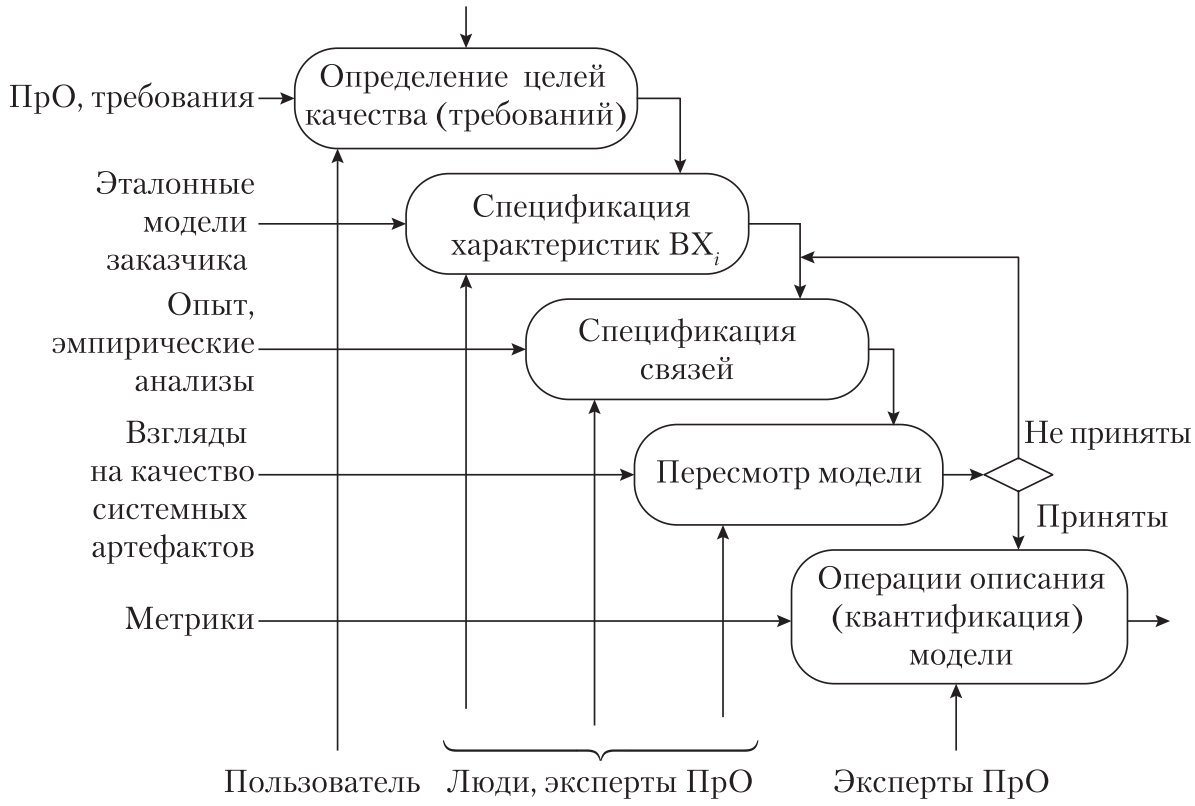


Рис. 5.4. **Порядок спецификации модели в подходе Prometheus**

Разработанная модель проверяется с позиций ее реализации. Сначала достигается консенсус всех заинтересованных сторон, потом определяются практические возможности и ресурсы сбора данных для вычисления значений метрик по модели. Это делается ради достижения компромисса между трудозатратами на применение модели и практической ценностью ее использования. Оценивается относительная важность характеристик, и в модели остаются лишь наиболее влиятельные и легкие в измерении характеристики. В конце выполняется квантификация модели — «наполнение» данными (исходными значениями) характеристик и связей применения метрик.

Метрики могут иметь не количественное, а качественное (интуитивное, экспертное) представление (с использованием порядковой шкалы, например уровень значений «очень высокий», «высокий», «средний», «низкий», «очень низкий»). Для квантификации связей (влияний), а также комбинирования количественных и качественных данных применяются байесовские сети.

Подход к моделированию характеристик активов СПС заключается в *комбинировании разнотипных моделей* соответственно уровням архитектуры, а именно:

- иерархических моделей на уровне отдельных компонентов;
- аналитических моделей зависимостей для композиций компонентов (что фактически оценивает качество интерфейсов компонентов);
- байесовских моделей применительно к частям архитектуры, для которых не могут быть установлены аналитические зависимости и на момент оценивания отсутствуют объективные данные в базе знаний конфигурации, хранящей значения по измерению показателей качества;

• метода анализа иерархий для балансирования требований к характеристикам качества и (или) определения *интегрального показателя качества* СПС.

Таким образом, в результате анализа факторов, которые влияют на характеристики качества, определено множество требований, характеристик и подходов к их моделированию. В частности, установлено, что наиболее приемлемыми являются подход, который заключается в *комбинировании разнотипных моделей* соответственно уровням архитектуры СПС (иерархических, аналитических и байесовских моделей), а также метод анализа иерархий для балансировки требований к характеристикам СПС.

5.2. Модели представления знаний

Наиболее распространенными моделями представления знаний являются: продукционные системы, логические модели, фреймы, семантические сети[1].

5.2.1. Продукционные системы

В продукционных системах знания представляются продукциями в виде совокупности специальных информационных единиц, которые имеют следующую структуру: *Имя продукции*, *Предпосылка*, *Условие*, *Постусловие*.

Предпосылки устанавливают на множестве правил некоторый порядок и приоритет их использования. Условия определяют возможность использования того или другого правила. Ядро продукции «Если А, то В» характеризует преобразование, которое составляет содержание продукционного правила. Постусловие говорит о том, что́ следует делать, когда данное продукционное правило выполнено.

В общем случае продукционная система включает следующие компоненты:

- БД, которая содержит множество фактов;
- базу правил, которая содержит набор продукций;
- интерпретатор (механизм логического вывода) или правила работы с продукциями.

База правил и БД образуют базу знаний. Факты в БД представляют собой краткосрочную информацию и в принципе могут меняться в ходе работы продукционной системы по мере накопления опыта. Правила являются более долговременной информацией и предназначены для определения гипотез (новых фактов) из того, что уже известно.

Продукционные системы разделяются на два типа — с прямыми и обратными выводами. При прямом выводе размышление ведется от данных до гипотез, а при обратном выполняется поиск доказательств или возражений некоторой гипотезы. Часто используются комбинации прямой и обратной цепи размышлений.

---

[1] См.: *Лорьер Ж. Л.* Системы искусственного интеллекта : пер. с фр. М. : Мир, 1990; *Осукра С.*, *Саэки Ю.* Приобретение знаний; *Уано Х.*, *Исидзуке М.* Представление и использование знаний : пер. с япон. М. : Мир, 1987.

Представление знаний при помощи продукций называют иногда «плоским», так как в продукционных системах отсутствуют средства для установления иерархии правил. Объем базы знаний продукционных систем растет линейно, по мере включения в нее новых фрагментов знаний, в то время как в традиционных алгоритмических системах, которые используют деревья решений, зависимость между объектом базы знаний и количеством собственно знаний является логарифмической.

5.2.2. Логические модели

Логические модели представления знаний реализуются средствами логики предикатов[1].

Предикатом называется функция, которая принимает только два значения — истина и ложь — и предназначена для выражения свойств объектов или связей между ними. Выражение, в котором утверждается или не утверждается наличие каких-нибудь свойств у объекта, называется высказыванием. Константы служат для именования объектов ПрО. Логические предложения или высказывания образуют атомарные формулы. Интерпретация предиката — это множество всех допустимых связываний переменных с константами. Связывание представляет собой подстановку констант вместо переменных. Предикат считается общезначимым, если он истинен на всех возможных интерпретациях. Говорят, что высказывание логически вытекает из заданных предпосылок, если оно истинно всегда, при истинных предпосылках.

Наиболее простым языком логики является исчисление высказываний, в котором отсутствуют переменные. Любому высказыванию можно приписать значение «истина» или «ложь». Отдельные высказывания могут соединяться связями И, ИЛИ, НЕ, которые называются булевыми операторами. Основы исчисления высказываний составляют правила образования сложных высказываний из атомарных.

Логические операторы есть в большинстве ЯП. Однако исчисления высказываний — недостаточно выразительное средство для обработки знаний, поскольку в нем не могут быть представлены предложения, которые включают переменные с кванторами.

Исчисление предикатов с кванторами (логика предикатов) является расширением исчисления высказываний, в котором для выражения отношений между объектами ПрО могут использоваться предложения, включающие не только константы, но и переменные.

В общем случае модели, основанные на логике предикатов, описываются формальной системой, которая задается четверкой:

$$\text{M} = (T, P, A, \Pi),$$

где T — множество базовых элементов или алфавит формальной системы; P — множество синтаксических правил, при помощи которых можно построить синтаксически корректные предложения; A — множество аксиом или некоторых синтаксически правильных предложений, заданных

[1] *Rukhaia Kh.* One methods Constructing a Formal systems / Rukhaia Kh. // Applied Mathematics, Informatics and Mechanics. 2006. V11. №2. P. 81– 89.

априорно; П — правила продукций (правила вывода или семантические правила), при помощи которых можно расширять множество *A*, добавляя в него синтаксически правильные предложения.

Главное преимущество логических моделей представления знаний состоит в возможности непосредственно запрограммировать механизм вывода синтаксически правильных высказываний. Примером такого механизма может быть процедура вывода, построенная на методе резолюций. Однако с помощью правил, которые задают синтаксис языка, невозможно установить истинность или ложность того или другого высказывания. Причем это распространяется на все языки. Высказывание может быть построено синтаксически правильно, но быть совсем бессодержательным.

5.2.3. Фреймы

Фрейм определяет структуру данных для представления стереотипных ситуаций. Модель представления знаний на основе фреймов использует концепцию организации памяти, понимания и обучения человека, предложенную М. Минским (1997). Фрейм (дословно — «рамка») — это единица представления знаний, детали которой могут изменяться с текущей ситуацией. Фрейм в любой момент может быть дополнен разнообразной информацией — о способах его применения, последствиях этого применения и др.

Структура фрейма состоит из характеристик стереотипной описываемой ситуации и их значений, которые называются соответственно слотами и заполнителями слотов.

Незаполненный фрейм называется протофреймом, а заполненный — экзофреймом. Роль протофрейма как оболочки в экзофрейме особенно важна. Эта оболочка позволяет выполнять процедуру внутренней интерпретации, благодаря которой в памяти системы находятся не бессодержательные сведения, а имеющие достаточно определенное, известное системе содержание.

Слот может содержать не только конкретное значение, но и имя процедуры, что позволяет вычислять его по заданному алгоритму, а также одну или несколько продукций (эвристик), при помощи которых это значение определяется. В слот может входить не одно, а несколько значений. Иногда слот включает компонент, называемый фасетом и задающий диапазон или перечень его возможных значений. Фасет указывает также предельные значения заполнителя слота.

Кроме конкретного значения, в слоте могут храниться процедуры и правила, которые вызываются при необходимости определить это значение. Среди них выделяют процедуры-демоны и процедуры-слуги. Первые запускаются автоматически при выполнении некоторого условия, а вторые активизируются только по специальному запросу.

Совокупность фреймов, которая моделирует какую-нибудь ПрО, представляет собой иерархическую структуру, в которой фреймы собираются при помощи родовидовых связей. На верхнем уровне иерархии находится фрейм, содержащий наиболее общую информацию для остальных фреймов. Фреймы имеют способность наследовать значение характеристик

своих родителей, которые находятся на более высоком уровне иерархии. Эти значения могут передаваться по умолчанию фрейму, который находится ниже в иерархии, но если последний содержит собственные значения данных характеристик, то в качестве истинных принимаются именно собственные значения. Это обстоятельство позволяет без осложнений учитывать во фреймовых системах различного рода исключения.

Различают статические и динамичные системы фреймов. В системах первого типа фреймы не могут быть изменены в процессе решения задачи, а в системах второго типа это допускается.

О системах, основанных на фреймах, говорят, что они являются объектно ориентированными. Каждый фрейм отвечает некоторому объекту ПрО, а слоты содержат данные, которые описывают этот объект, т.е. в слотах находятся значения признаков объектов. Фрейм может быть представлен в виде списка свойств, а если использовать средства БД, то в виде записи.

Наиболее ярко преимущества фреймовых систем представления знаний проявляются в том случае, когда родовидовые связи изменяются нечасто и ПрО насчитывает немного исключений. Во фреймовых системах данные о родовидах хранятся явно, как и знания других типов. Значения слотов представляются в системе в единственном экземпляре, поскольку включаются только в один фрейм, который описывает наиболее общие понятия из всех тех, что содержит слот с данным именем. Такое свойство систем фреймов обеспечивает экономное размещение базы знаний в памяти компьютера. Еще одно преимущество фреймов заключается в том, что значение любого слота будет рассчитано при помощи соответствующих процедур или найдено эвристическими методами. То есть фреймы позволяют манипулировать как декларативными, так и процедурными знаниями.

К недостаткам фреймовых систем относят их относительно высокую сложность, что проявляется в снижении скорости работы механизма вывода и в увеличении трудоемкости внесения изменений в родовидовую иерархию. Поэтому при разработке фреймовых систем больше внимания уделяют демонстративным способам отображения и эффективным средствам редактирования фреймовых структур.

5.2.4. Семантические сети

Семантическая сеть описывает знание в виде сетевых структур. В качестве вершин сети выступают понятия, факты, объекты, события и т.п., а в качества дуг сети — отношения, которыми связаны вершины между собой.

Семантические сети часто рассматривают как общий формализм для представления знаний. Частичным случаем таких сетей являются сценарии, в которых в качестве отношений выступают казуальные отношения или отношения типа «цель — средства».

Вершины сети соединяются дугой, если соответствующие объекты предметной области находятся в каком-нибудь отношении.

Наиболее распространенными типами отношений являются следующие:

БЫТЬ ЭЛЕМЕНТОМ КЛАССА — значит, что объект входит в состав данного класса;

ИМЕТЬ — позволяет задавать свойства объектов;

ЯВЛЯЕТСЯ СЛЕДСТВИЕМ — отображает причинно-следственные связи;

ИМЕТЬ СМЫСЛ — задает значение двух объектов.

Как и в системе на фреймах, в семантической сети могут быть представлены родовидовые отношения, которые позволяют реализовать наследование свойств от объектов-родителей. Это обстоятельство приводит к тому, что семантические сети приобретают все недостатки и преимущества представления знаний в виде фреймов. Преимущества состоят в простоте и наглядности описания ПрО. Однако последнее свойство с усложнением семантической сети теряется, кроме того, существенно увеличивается время вывода. Также к недостаткам семантических сетей относят сложность обработки различного рода исключений.

5.2.5. Методы многомерного анализа данных

Очень часто извлечение знаний трактуется как «смесь статистики, средств искусственного интеллекта и анализа баз данных» и до последнего времени не признавалось полноценной областью интересов для специалистов по статистике. Однако благодаря большому практическому значению эта проблематика сейчас интенсивно разрабатывается и вызывает большой интерес (в том числе и в ее статистических аспектах) и в ней достигнуты важные теоретические результаты.

Есть, однако, важное отличие процедуры извлечения знаний от классического разведывательного анализа данных: системы извлечения знаний в большей степени ориентированы на практическое использование полученных результатов, чем на выяснение природы явления. Иными словами, при извлечении данных не очень важен конкретный вид зависимостей между переменными задачи. Выяснение природы функций, участвующих в процессе, или конкретной формы интерактивных многомерных зависимостей между переменными не является главной целью этой процедуры. Основное внимание уделяется поиску решений, на основании которых можно было бы строить достоверные прогнозы. Таким образом, в области извлечения данных принят такой подход к анализу данных и извлечению знаний, которые иногда характеризуют словами «черный ящик». При этом используются не только классические приемы разведывательного анализа данных, но и такие средства, как нейронные сети, которые позволяют строить достоверные прогнозы, не уточняя конкретный вид тех зависимостей, на которых такой прогноз основан.

Методы многомерного разведывательного анализа специально разработаны для поиска закономерностей в многомерных БД (последовательностей одномерных данных). К ним относятся:

1) кластерный анализ;
2) факторный анализ;
3) анализ дискриминантных функций;
4) многомерное шкалирование;

5) логлинейный анализ;
6) каноническая корреляция;
7) пошаговая линейная и нелинейная регрессия;
8) анализ соответствия;
9) анализ временны́х рядов и деревьев классификации.

К основным средствам разведывательного статистического анализа относятся:

- процедура анализа распределения переменных (например, чтобы обнаружить переменные с несимметричным или негауссовским распределением, в том числе и бимодальные);
- просмотр корреляционных матриц с целью поиска коэффициентов, которые превышают по величине определенные пороговые значения;
- анализ многовходовых таблиц частот (например, «послойный» последовательный просмотр комбинаций уровня управляющих переменных).

В отличие от традиционной проверки гипотез разведывательный анализ данных применяется для нахождения связей между переменными в ситуациях, когда отсутствует (или недостаточно) априорное представление о природе этих связей. При разведывательном анализе учитывается и сравнивается большое число переменных. Вычислительные методы разведывательного анализа данных включают основные статистические средства, а также более сложные, специально разработанные средства многомерного анализа, предназначенные для поиска закономерностей в многомерных данных.

5.2.6. Инструменты извлечения (добывания) знаний

Система Scenario предназначена для выявления взаимосвязей в данных. Анализируемая информация представляется в виде таблиц, столбцами которых задаются факторы (переменные), описывающие ПрО. При этом один из этих столбцов выделяется пользователем как целевой показатель. Scenario фокусирует внимание аналитиков на измеряемых показателях и факторах, что позволяет им переходить от качественных рассуждений к построению более точных количественных моделей. В Scenario реализовано автоматическое выявление зависимостей по критерию χ-квадрат (метод CHAID) для нахождения однородных групп (сегментов) данных с «аналогичным» поведением относительно целевого показателя.

Система использует метод деревьев решений и строит правила классификации, которые можно применять к новым данным для построения прогнозов относительно возможных значений целевого показателя. Ядро системы составляют методы статистики. Основная статистическая информация отображается методами наглядной графики, а более детальные сведения пользователь в случае необходимости может запросить сам. Собственно методы интеллектуального анализа данных являются основанием для реализации типовых видов анализа, которые используются аналитиками и лицами, принимающими решение.

Поскольку этот метод почти всегда требует значительных затрат времени и вычислительных ресурсов, в Scenario реализованы три стратегии анализа.

1. Для проведения быстрого начального анализа задачи, отсева незначительных факторов и фокусирования внимания на ключевых параметрах рекомендуется использовать режим **исследования**. Этот режим позволяет выигрывать во времени за счет некоторого снижения точности результатов. Поскольку в дальнейшем допускается проведение более детального анализа, требования к надежности не выдвигаются. Размер выборки не ограничивается.

2. Более точный анализ может быть проведен в режиме **тестирования**. Использование этого режима обеспечивает очень высокую точность и достаточную надежность результатов. Для его проведения требуется не менее 300 записей в БД.

3. Наилучшие по точности и надежности результаты можно получить в режиме **верификации**. Этот режим, требующий не менее 1000 записей, рекомендуется использовать для проверки и уточнения результатов, полученных в ходе тестирования.

Oracle Data Mining Suite

Комплект Oracle Data Mining Suite Release 3.7 — это ПП Data Mining, который характеризуется легкостью использования и богатой функциональностью для разрешения сложных проблем. Этот комплект выявляет скрытые закономерности в данных, строит модели предвидения и помещает свои «предвидение» и «интуицию» в БД для использования другими прикладными программами и пользователями.

Oracle Data Mining Suite — это «производитель» ценной новой информации для других «потребителей» данной организации. Data Mining Suite предоставляет легкий в использовании, интуитивно понятный пользователю интерфейс. Oracle Data Mining Suite предлагает мастера-утилиты для упрощения и автоматизации шагов Data Mining. Например, мастер Key Fields в Oracle Data Mining Suite автоматические находит наиболее нужные переменные, которые максимально влияют на решение некоторого конкретного вопроса. Мастер Model Seeker автоматически строит большинство моделей Data Mining, показывает интерактивные графы и таблицы результатов, а также рекомендует наилучшие модели.

Oracle Data Mining Suite предлагает многопроцессорные компьютеры (SMP) для «добычи» огромных массивов данных и извлечения из них более ценной информации. Данный инструмент может извлекать данные из разнообразных сетевых источников данных, транспортировать информацию из БД Oracle и использовать технологию прямого доступа к данным Oracle.

На фазе построения моделей Oracle Data Mining Suite предоставляется множество алгоритмов — C&RT, классификация и регрессия, также известные как деревья решений, нейронные сети, алгоритм нахождения ближайших k соседей и кластеризация.

Oracle Darwin Data Mining Suite

В состав линейки продуктов Oracle 9i входит инструментальная среда класса Data Mining под названием Darwin. Эта система поддерживает все

этапы технологии извлечения знаний, включая постановку задачи, подготовку данных, автоматическое построение моделей, анализ и тестирование результатов, использование моделей в реальных приложениях.

На этапе подготовки данных обеспечивается доступ к любым реляционным БД, текстовым файлам, SAS-файлам. Darwin поддерживает построение пяти различных типов моделей — нейронные сети, классификационные и регрессионные деревья решений, ближайшие k-окрестности, байесовское обучение и кластеризацию.

Развитые графические средства предоставляют широкие возможности для анализа полученных результатов, верификации моделей на тестовых наборах данных, оценки точности и устойчивости результатов. Уточненные и проверенные модели можно включать в существующие приложения путем генерации их описаний на C, C++, Java, а также разрабатывать новые специализированные приложения с помощью входящей в состав Darwin среды разработки (SDK).

Важной особенностью системы Darwin, выделяющей его среди других средств извлечения знаний, являются его технические характеристики: работа в архитектуре «клиент-сервер», широкое использование техники параллельных вычислений, высокая степень масштабируемости при увеличении вычислительных ресурсов. Все это позволяет выполнять процедуры автоматического анализа данных огромных объемов, достигая очень высоких временны́х показателей.

Oracle Personalization

Эта система помогает компаниям установить истинно однозначные (1 : 1) отношения через Интернет и выполняет оценку вероятности того, что данному клиенту подойдут эти рекомендации, выполняет улучшенную навигацию по сайту на основании интересов и профилей клиентов. Personalization позволяет пользователям создавать «механизмы рекомендаций», которые предоставляют индивидуальные рекомендации веб-сайтам на основании анализа поведения веб-клиентов. Модели предвидения Oracle Personalization могут модифицироваться периодически, т.е. ежедневно, еженедельно, ежемесячно, и использоваться механизмами рекомендаций.

WizWhy — система поиска логических правил

Система WizWhy предприятия WizSoft (http://www.wizsoft.com) является современным представителем подхода, который реализует ограниченный перебор. Хотя авторы системы не раскрывают специфику алгоритма, положенного в основу работы WizWhy, вывод о наличии здесь ограниченного перебора был сделан по результатам тщательного тестирования системы (изучались результаты, зависимости времени их получения от числа параметров, которые анализируются, и т.п.). В WizWhy ограниченный перебор используется в варианте, который модифицировался с применением дополнительного алгоритма «Apriori», исключающего из анализа логические события с низкой частотой.

Алгоритмы ограниченного перебора были предложены в середине 1960-х гг. М. М. Бонгардом для поиска логических закономерностей в дан-

ных. С тех пор они доказали свою эффективность при решении множества задач в разнообразных областях.

Эти алгоритмы вычисляют частоты комбинаций простых логических событий в подгруппах данных. Примеры простых логических событий: $X = a$; $X < a$; $X > a$; $a < X < b$, где X — любой параметр, a и b — константы. Ограничением является длина комбинации простых логических событий. На основании анализа вычисленных частот делается вывод о полезности той или другой комбинации для установления ассоциации в данных, для классификации, прогнозирования и т.п.

Для большей убедительности авторы WizWhy противопоставляют свою систему нейросетевому подходу и алгоритмам построения деревьев решений и утверждают, что WizWhy, владея более высокими характеристиками, вытесняет другие ПП с рынка Data Mining.

5.3. Методы поиска и извлечения знаний

Data Mining — процесс анализа данных, направленный на поиск новой полезной информации, обычно представленной в форме ранее неизвестных отношений между переменными[1].

Однако совокупность большого количества таких данных (Big Data), накопленных за несколько лет, может стать источником дополнительной, гораздо более ценной информации, которую нельзя получить на основе одной конкретной записи, а именно — сведений о закономерностях, тенденциях или взаимозависимостях между какими-либо данными. Подобного рода информация обычно используется при прогнозировании, стратегическом планировании, анализе рисков и предприятия. Отсюда процесс ее поиска и получил название Data Mining (mining по-английски означает «добыча полезных ископаемых», а поиск закономерностей в огромном наборе фактических данных действительно близок к этому). Термин Data Mining обозначает не столько конкретную технологию, сколько сам процесс поиска корреляций, тенденций, взаимосвязей и закономерностей посредством различных математических и статистических алгоритмов: кластеризации, создания субвыборок, регрессионного и корреляционного анализа.

Цель этого поиска — представить данные в виде, четко отражающем процесс поиска, а также построить модель, при помощи которой можно прогнозировать процессы, критичные для планирования изготовления продуктов в СПС. Нужно отметить, что традиционная математическая статистика, долгое время остававшаяся основным инструментом анализа данных, равно как и средства оперативной аналитической обработки данных (Online Analytical Processing, OLAP), не всегда могут успешно применяться для решения таких задач, для которых заранее сформулированы гипотезы.

---

[1] См.: *Поспелов Д. А.* Представление знаний. Опыт системного анализа.

Основой технологии Data Mining является концепция шаблонов, отражающих закономерности, свойственные подвыборкам данных. Поиск шаблонов производится методами, не использующими никаких априорных предположений об этих подвыборках.

Средства Mining отличаются нестандартностью и неочевидностью разыскиваемых шаблонов и на основании имеющихся данных способны находить такие взаимозависимости самостоятельно и строить гипотезы об их характере. Тут могут использоваться OLAP для лучшего понимания характера закономерностей, которые следует искать.

5.3.1. Методы машинного обучения

Существует множество различных инструментов Mining, но в основе большинства из них лежат две технологии: визуализация (визуальное представление информации) и машинное обучение. *Визуализация* используется для поиска исключений, общих тенденций и зависимостей проекта. *Машинное обучение* — для поиска зависимостей в уже отлаженном проекте. Обучение предполагает применение следующих методов:

- дерева решений;
- категоризации тестов:
- ассоциативных правил;
- генетических алгоритмов;
- нейронных сетей.

5.3.2. Методы извлечения знаний

К методам извлечения знаний относятся: нахождение подобного, дерево решений, алгоритм C4.5, вероятность и кластеризация.

Категоризация текстов

Это сопоставление текстов, написанных на некотором языке с предопределенным компонентом, основанном на содержании данных. Эта задача находит широчайшее применение в таких сферах, как отбор необходимых текстов в электронных библиотеках при поступлении нового материала, фильтрация содержимого по отдельным признакам. Автоматическая категоризация текста играет важную роль в задачах фильтрации почты или сортировки файлов в иерархии каталогов, определении ключевых тем и выборе статей по ним, нахождении документов, соответствующих каким-то меняющимся критериям, которые сложно задать с формальной точки зрения. Данная задача также играет особую роль в фильтрации информации в работе поисковых систем.

Метод нахождения подобного

Для примера при рассмотрении метода нахождения подобного может быть применен модифицированный метод Роччио. В его формулировке веса, соответствующие некоторому терму, являются комбинацией весов в оригинальном запросе и в последующей оценке релевантности документов:

$$x_j = \alpha \cdot x_{q,j} + \beta \cdot \frac{\sum\limits_{i \in rel} x_{i,j}}{n_r} + \gamma \cdot \frac{\sum\limits_{i \in non-rel} x_{i,j}}{N - n_r}.$$

Параметры α, β и γ контролируют относительную важность оригинального вектора запроса, положительные и отрицательные отклики. В контексте классификации текстов не существует исходного запроса, соответственно, $\alpha = 0$. Для облегчения можно полагать также, что $\gamma = 0$, таким образом переходя к простому среднему весов положительных исходов.

Деревья решений, алгоритм C4.5

При рассмотрении метода деревьев решений может применяться алгоритм C4.5 — фундаментальный алгоритм в реализации Microsoft Desicion Trees. Он был впервые предложен Р. Квинланом (Quinlan) в 1993 г. Суть его заключается в рекурсивном разделении данных. Данный алгоритм рассматривает все возможные данные, которые могут разделить исходный набор, и выбирает тест, который дает наилучшую выходную цепочку.

Вероятностный подход

При рассмотрении метода классификации Байеса используем вероятностный подход для каждой категории, по которой строится теория о принадлежности документа какому-то классу:

$$P(C = c_k \mid \vec{x}) = \frac{P(\vec{x} \mid C = c_k)P(C = c_k)}{P(\vec{x})}.$$

Данная величина трудно вычислима, и для ее оценки используются допущения о том, что все X_1, ..., X_n являются независимыми величинами. Это упрощает вычисления до формулы

$$P(\vec{x} \mid C = c_k) = \prod_i P(x_i \mid C = c_k).$$

Несмотря на тот факт, что допущение о независимости слов в целом не является истиной, данный классификатор достаточно эффективен. Дальнейшим развитием этого метода являются байесовские сети.

Методы кластеризации

Стоит также упомянуть о методе кластеризации. Данный метод можно исследовать с применением метода k-ближайших соседей (что является основой алгоритма Microsoft Clustering). Суть его в следующем. Предположим, что нам необходимо классифицировать объект S, признаковое описание которого представлено вектором $x(S)$. Для каждого из классов K_1, ..., K_l вычисляется условная вероятность принадлежности $P(K_i|x)$. Объект S относится к тому классу, для которого условная вероятность принадлежности максимальна. Данное решающее правило минимизирует вероятность ошибочной классификации. Его обычно называют байесовским решающим правилом или оптимальным байесовским классификатором. К сожалению, байесовское решающее правило не может быть реализовано в подавляющем большинстве практических задач из-за того, что вероятностное распределение неизвестно.

Однако мы можем попытаться оценить условные вероятности принадлежности $P(K_i|x)$, используя информацию, содержащуюся в обучающей выборке.

Метод *k*-ближайших соседей

Оценка условной вероятности $P(K_i|x)$ в методе k-ближайших соседей ведется по ближайшей окрестности V_k точки $x(S)$ в признаковом пространстве, содержащей по крайней мере k признаковых описаний объектов обучающей выборки. В качестве оценки $P(K_i|x)$ выступает отношение $\frac{k_i}{k}$, где k_i — число объектов из класса K_i в окрестности V_k. Естественно, что поиск ближайшей окрестности должен основываться на использовании некоторой функции расстояний, заданной на множестве пар точек признакового пространства. В качестве такой функции расстояний, в частности, может выступать евклидова метрика. Точность распознавания методом k-ближайших соседей существенно зависит от числа k, оптимизация которого может производиться по обучающей выборке. При этом в качестве оптимального берется то число ближайших соседей, при котором оценка точности распознавания с использованием режима скользящего контроля максимальна. Основным недостатком метода k-ближайших соседей является снижение его эффективности при малых объемах выборки и высокой размерности признакового пространства.

Данный алгоритм можно разбить на следующие шаги.

Шаг 1. Выбираются k исходных центров кластеров $y_1(l)$, $y_2(l)$, ..., $y_k(l)$ $z_1(l)$, $z_2(l)$, ..., $z_k(l)$. Этот выбор производится произвольно, например первые k результатов выборки из заданного множества объектов.

Шаг *l*. На l-м шаге итерации заданное множество объектов $\{x\}$ распределяется по k группировкам по следующему правилу:

$$x \in T_j(l), \text{ если } \left\|x - y_j(l)\right\| < \left\|x - y_i(l)\right\|$$

для всех $i = 1, 2, ..., k$, $i \neq j$, где $T_j(l)$ — множество образов, входящих в кластер с центром $y_j(l)$. В случае равенства решение принимается произвольным образом.

Шаг *l* + 1. На основе результатов шага l определяются новые центры группировок $y_j(l + 1)$, $j = 1, 2, ..., k$, исходя из условия, что сумма квадратов расстояний между всеми объектами, принадлежащими множеству $T_j(l)$, и новым центром данной группировки должна быть минимальной.

Центр $y_j(l + 1)$, обеспечивающий минимизацию $J_j = \sum_{x \in T_j(l)} \left\|x - z_j(l+1)\right\|^2$, $j = 1, 2, ..., k$, является выборочным средним, определенным по множеству $T_j(l)$. Следовательно, новые центры кластеров определяются как

$$y_j(l+1) = \frac{1}{N_j} \sum_{x \in T_j(l)} x, j = 1, 2, ..., k,$$

где N_j — число выборочных объектов, входящих в множество $T_j(l)$. Очевидно, что название алгоритма «k внутригрупповых средних» определяется способом, принятым для последовательной коррекции назначения центров кластеров.

Равенство $y_i(l + 1) = z_j(l)$ при $j = 1, 2, ..., k$ является условием сходимости алгоритма, и при его достижении выполнение алгоритма заканчива-

ется. Полученные множества $T_j(l), j = 1, 2, ..., k$, образуют искомые кластеры. В противном случае последний шаг повторяется.

В противовес данному алгоритму можно поставить алгоритм локализации аномалий с применением метода кластеризации.

Метод локализации аномалий

Метод заключается в последовательной обработке двух компонентов. Цель первого компонента — выбрать наилучшую комбинацию атрибутов x_i, дающих наиболее выразительную и контрастную кластеризацию. Второй компонент находит кластеры в пространстве этих фиксированных наборов атрибутов x_i. Рассмотрим принцип работы данного компонента. Предположим, что нам удалось разделить пространство значений гиперплоскостями на набор N-мерных прямоугольников. Рассмотрим одну из этих областей E_i. Пусть n — число точек в данной области, N — общее число записей в исследуемом наборе. Область E_i может быть рассмотрена как прямое произведение элементов генеральной совокупности: $E_i = S_1, ..., S_m$. Обозначим количество точек со значением j-го атрибута, попадающих в сегмент S_j, как M_j. Если точки не образуют кластер в пространстве атрибутов x_i, которые считались независимыми атрибутами, то относительная плотность точек в E_i примерно равна произведению одномерной относительной плотности точек в сегменте S_j:

$$\frac{n}{N} \approx p_j = \frac{M_1 \cdots M_m}{N^m}.$$

Существенно большее значение $\frac{n}{N}$ означает, что E_i должно быть рассмотрено как часть кластера. В случае $m = 1$ первое приближение данного равенства заведомо верно. Таким образом, минимально возможная размерность кластеризуемого пространства m равна двум.

Для того чтобы найти кластеры, состоящие из прямоугольных областей с аномальной плотностью точек, используется следующая процедура. Для каждой области E_i с количеством точек больше, чем $Np_i = \frac{M_1 \cdots M_m}{N^{m-1}}$, подсчитывается вероятность того, что высокая плотность точек в данной ячейке является результатом статистической флуктуации. То есть для каждой области E_i мы определяем значение $s_i = b\left(n, N, \frac{M_1 \cdots M_m}{N^m}\right) = b(n, N, p_i)$, где $b(k, K, p)$ — хвост биномиального распределения с количеством испытаний K и вероятностью p.

Далее создается список всех таких E_i, отсортированный по возрастанию значений s_i. Обозначим полученную последовательность ячеек как $\{E'_j\}$. Для каждой такой области E'_j мы знаем количество точек, лежащих в ней, — n_j и значение p_j. Для каждого j подсчитывается значение $S_{CUM_1} = b\left(\sum_{i=1}^{j} n_i, N, \sum_{i=1}^{j} p_i\right)$. Для обозначения будем считать, что значение j,

при котором S_{CUM_1} является минимальным, обозначается J_{BEST}. Соответствующее ему минимальное значение S_{CUM_1} обозначим S_{BEST}. Это значение соответствует наиболее контрастному, наиболее значимому расщеплению всех областей E_i на «плотные» и «разреженные». Множества прилегающих областей из $\{E'_j\}$, для которых $j \le j_{BEST}$ (наиболее плотные области), называются **кластерами**.

Теперь рассмотрим процедуру, которая выбирает наилучшую комбинацию атрибутов для кластеризации. Цель данного этапа — найти подмножества множества данных, которые имеют максимальное значение некоторого параметра, определяющего качество кластеризации. В случае «жесткой» модификации алгоритма локализации аномалий таким критерием является $1 - S_{BEST}$. В случае «мягкой» модификации количество точек в кластере служит критерием. Кроме того, требуется, чтобы процедура кластеризации выявляла как минимум два кластера и $1 - S_{BEST}$ должно быть больше, чем некоторый доверительный уровень.

Используя данные наборы алгоритмов, предлагается выработать критерии применимости, а также параметры, влияющие на получение оптимального результата. Также должно быть проведено сравнение эффективности работы данных алгоритмов с точки зрения как временны́х, так и качественных показателей.

Для обучения могут быть использованы тестовые базы агенства Reuters (равно как и для проверки работы). Также работу алгоритмов можно проверить по применению их к базе интернет-библиотеки (http://www.lib.ru), где присутствует в конечном виде разделение литературы по жанрам, что может являться достоверным критерием правильности работы и эффективности каждого из алгоритмов.

С помощью полученных данных выработывается комбинированный алгоритм для классификации (категоризации) текстов, который можно опробовать в такой задаче, как категоризация интернет-почты по признаку наличия рекламных сообщений.

5.3.3. Применение технологии Mining к системам

Рассматриваются разные области применения данной технологии: банковское дело, финансы, бизнес, медицина, автомобильная промышленность, торговля и др.

Банковское дело

В настоящее время банки являются накопителями огромного количества разнообразных данных о клиентах, что помогает расследовать случаи мошенничества с кредитными картами, прогнозировать изменение клиентуры, предлагать новые услуги клиентам. Основные задачи технологии Data Mining следующие:

- *выявление мошенничества* с кредитными картами;
- *сегментация клиентов* путем разбивки клиентов на различные категории, маркетинговая политика предлагает различные виды услуг разным группам клиентов;
- *изменений клиентуры* путем построения прогнозных моделей ценности для своих клиентов для обслуживания каждой категории.

Телекоммуникация

В этой области методы Data Mining помогают компаниям продвигать свои программы маркетинга и ценообразования, чтобы удерживать существующих клиентов и привлекать новых. Среди типичных мероприятий приведем следующие:

- *анализ записей о подробных характеристиках вызовов*. Назначение такого анализа — выявление категорий клиентов с похожими стереотипами пользования их услугами и разработка привлекательных наборов цен и услуг;
- *выявление лояльности клиентов*. Data Mining можно использовать для определения характеристик клиентов, которые, один раз воспользовавшись услугами данной компании, с большой долей вероятности останутся ей верны. В итоге средства, выделяемые на маркетинг, можно тратить там, где отдача больше.

Страхование

Страховые компании в течение ряда лет накапливают большие объемы данных. К примеру, страхование автогражданской ответственности предполагает анализ предварительной статистики для каждого водителя. Методы Data Mining обеспечивают:

- *выявление мошенничества* путем поиска определенных стереотипов в заявлениях о выплате страхового возмещения, характеризующих взаимоотношения между юристами, врачами и заявителями;
- *анализ риска* путем выявления сочетаний факторов, связанных с оплаченными заявлениями, что уменьшает их потери по обязательствам.

Автомобильная и авиационная промышленность

При сборке автомобилей производители должны учитывать требования каждого отдельного клиента, поэтому им нужна возможность прогнозировать популярность определенных характеристик и знание того, какие характеристики обычно заказываются всеми. Проводится:

- *политика гарантий*. Производителям нужно предсказывать число клиентов, которые подадут гарантийные заявки, и среднюю стоимость заявок;
- *поощрение часто летающих клиентов*. Авиакомпании могут обнаружить группу клиентов, которых данными поощрительными мерами можно побудить летать больше. Например, одна авиакомпания обнаружила категорию клиентов, которые совершали много полетов на короткие расстояния, не накапливая достаточно миль для вступления в их клубы, поэтому она таким образом изменила правила приема в клуб, чтобы поощрять число полетов так же, как и мили.

Медицина

Известно много экспертных систем для постановки медицинских диагнозов. Они построены главным образом на основе правил, описывающих сочетания различных симптомов разных заболеваний. С помощью таких правил узнают не только, чем болен пациент, но и как нужно его лечить. Правила помогают выбирать средства медикаментозного воздействия,

определять показания и противопоказания, ориентироваться в лечебных процедурах, создавать условия наиболее эффективного лечения, предсказывать исходы назначенного курса лечения и т.п. Технологии Data Mining позволяют обнаруживать в медицинских данных шаблоны, составляющие основу указанных правил.

Молекулярная генетика и генная инженерия

Задача обнаружения закономерностей в экспериментальных данных стоит остро в молекулярной генетике и генной инженерии. Здесь она формулируется как определение так называемых маркеров, под которыми понимают генетические коды, контролирующие те или иные фенотипические признаки живого организма. Такие коды могут содержать сотни, тысячи и более связанных элементов. На развитие генетических исследований выделяются большие средства. В последнее время в данной области возник особый интерес к применению методов Data Mining.

Прикладная химия

Методы Data Mining находят широкое применение в прикладной химии (органической и неорганической). Здесь нередко возникает вопрос о выяснении особенностей химического строения тех или иных соединений, определяющих их свойства. Особенно актуальна такая задача при анализе сложных химических соединений, описание которых включает сотни и тысячи структурных элементов и связей.

Можно привести еще много примеров различных областей знания, где методы Data Mining играют ведущую роль. Особенность этих областей заключается в их сложной системной организации. Они относятся главным образом к такому уровню организации систем, закономерности которого не могут быть достаточно точно описаны на языке статистических или иных аналитических математических моделей. Данные в указанных областях неоднородны, гетерогенны и часто отличаются высокой размерностью.

Контрольные вопросы и задания

1. Дайте определение методу анализа иерархий.
2. Дайте определение понятия байесовских сетей.
3. В чем смысл иерархических моделей качества?
4. Определите модели представления знаний.
5. Что такое продукционные системы?
6. Определите логические модели.
7. Определите фреймы.
8. Определите семантические сети.
9. Когда используется многомерный анализ данных?
10. Назовите инструменты добывания знаний.
11. Определите методы извлечения знаний.
12. Определите методы машинного обучения.
13. В каких областях применяются технологии Mining?

Глава 6
ФОРМАЛЬНЫЕ МЕТОДЫ СПЕЦИФИКАЦИИ, ВЕРИФИКАЦИИ И ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ СИСТЕМ

Одной из важных задач проектирования систем является разработка спецификаций отдельных элементов, включая описание семантики и функциональной структуры системы, спецификацию входных и выходных данных программы. Основу формальных методов спецификаций составляет задание определенных свойств программ с помощью математических методов, которые потом доказываются. Особенность использования спецификаций состоит в определении понятий системы и установлении связей между заказчиком и проектировщиком, между проектировщиком и разработчиком, а также между разработчиком и тестировщиком. Также спецификация выступает как документация к программному коду системы.

Одним из методов разработки ПС являются UML (*Unified Modeling Language*) и соответствующие CASE-инструменты (Rational Rose, MS Visio, Rational Software и др.). Эти инструменты поддерживают разработку ПС на всех этапах ЖЦ (см. Приложение 1). Результатом будет представление ПС с помощью графических диаграммных элементов *use case*, не обладающих формальной семантикой. Это может привести к неоднозначности толкования функций системы. Проектирование в системе UML, а также в языках MSC, SDL наглядно, но в них отсутствуют средства доказательства свойств элементов системы и системы в целом.

Спецификации основываются на аппарате математической логики, способствующей описанию программы как математических объектов, содержащих формальное описание семантики на одном из формальных языков RSL, VDM, Z, B и др. Элементы системы, описанные на этих языках, требуется обрабатывать с помощью специально разработанных математических средств, обеспечивающих доказательство определенных свойств программ и правильности семантики.

6.1. Методы и языки формальной спецификации

Формальная спецификация используется при описании модели системы и требований к ее поведению в терминах языка.

Метод формальной разработки ПС — это набор методов и средств, основанных на математических принципах (моделирование, математическая логика, теория множеств, теория конечных автоматов, алгебра и т.д.),

которые позволяют формально специфицировать программу, а потом проводить верификацию с учетом требований к ПС. Как правило, кроме формальных методов используются стандартные языки. К ним относятся: VDM, RAISE, Z, CSP, CCS.

Формальный язык (язык спецификаций, формальной нотации) — это язык с определенным синтаксисом и семантикой. Такие языки формальных спецификаций используются для написания свойств отдельных понятий проблемной области. Методы формальной разработки базируются на использовании некоторого языка спецификаций и средствах формальных соображений для построения ПС.

Существует ряд подходов к методам формальной разработки ПС:

- операционный — разработка системы базируется на описании модели, имеет свойства разрабатываемой системы;
- аксиоматический — разработка базируется на аксиоматическом описании поведения системы;
- алгебраический — разработка базируется на алгебраическом описании свойств действий;
- гибридный — объединяет операционный подход с аксиоматичным или алгебраическим.

Шаги при формальной разработке объединены и сопровождаются анализом, который базируется на проверке свойств системы. Проверка свойств системы осуществляется средствами формальных соображений с помощью двух основных подходов:

- автоматическое доказательство теорем (*theorem proving*) — программное доказательство логических теорем (свойств системы). В основе автоматического доказательства теорем лежит аппарат математической логики;
- проверка модели (*model checking*) — процесс проверки модели заданной ПрО.

6.1.1. Классификация методов и языков спецификации программных систем

В основу классификации языков и методов спецификаций программ можно положить разные критерии: назначение, вид, полноту и форму представления и т.д. Таким образом, получаем различные виды классификаций языков и методов спецификаций ПрО и ПС. Языки спецификации бывают статические и динамические (*executable* — Z, RSL, VDM, B).

Имеются следующие типы спецификаций:

- поведенческие (*behavioral*) — описывают ограничения на поведение объекта спецификации, а именно на функциональные возможности, безопасность и выполнение;
- структурные (*structural*) — описывают ограничения на внутренний состав объекта спецификации, а именно на использование и состав, отношение зависимости;
- взаимодействия (*interaction*) — описывают ограничения на взаимодействие между двумя или большим количеством объектов спецификации, а именно на соответствие интерфейсов и протоколов.

Методы спецификаций традиционно классифицируют по подходам к представлению модели. В частности, выделяют:

- модель-ориентированные подходы к спецификации. Их целью является построение абстрактной модели специфицированной информационной системы. Такие методы спецификаций базируются на описании состояний (*state-based*). Например, VDM, Z, RAISE, B и др.;
- методы, ориентированные на свойства (алгебраические). Их целью является описание системы в терминах желаемых свойств без построения явной модели. Такие методы спецификаций базируются на описании действий (*action-based*). Примеры: OBJ, Larch, Anna, Clear, ateCharts, CSP, CCS.

Различие между модель-ориентированными и алгебраическими методами не очень четкое. На практике модель-ориентированные спецификации часто описывают аспекты спецификации системы с помощью аксиом, свойственных спецификациям, ориентированным на свойства. А алгебраические спецификации часто описывают наборы базовых ТД при моделировании свойств, но при этом используют построение модели спецификации системы. Методы, которые являются одновременно модель-ориентированными и ориентированными на свойства, называются гибридными (в частности, VDM, Anna, Z, RSL, UML).

Языки спецификаций классифицируются на следующие базовые типы:

- транзиторные спецификации (спецификации переходов состояний);
- темпоральные (временны́е) логические спецификации;
- параллельные спецификации;
- алгебраические спецификации;
- аксиоматические спецификации.

Язык спецификаций может поддерживать несколько типов.

Спецификации абстрактной модели

Явно описывают поведение в терминах модели. Спецификация использует четко определенные типы (множества, последовательности, отношения, функции) и определяет операции на моделях. Спецификация включает модель типа, инвариант, пред- и постусловия.

Особенностями таких спецификаций являются явное моделирование состояний в моделях, возможность выполнения, возможность построения объектов в иерархическом порядке.

Спецификации абстрактной модели поддерживаются на языках VDM, Z, RAISE, UML/OCL.

Алгебраические спецификации

Определяют поведение совокупности отношений эквивалентности, описывающих свойства объектов и операции над ними.

Спецификация включает функции и отношения.

Особенностями таких спецификаций являются наличие описаний функций, поддержка абстракции данных, они особенно применимы для абстрактных ТД.

Алгебраические спецификации поддерживают такие языки, как OBJ, Larch, Anna, Clear.

Транзиционные системы

Описывают поведение системы набором состояний и определяют операции перехода между состояниями или отношениями. Спецификация включает состояния и переходы. Их особенность — задание общего пространства состояний, текстовые и графические заметки, модульность, управление системами.

Спецификации состояний и переходов поддерживаются такими языками, как StateCharts, CCS, UML/OCL (диаграммы конечных автоматов, деятельности).

Аксиоматические спецификации

Определяют поведение с помощью логических формул, а также задают входные, промежуточные и выходные утверждения.

Спецификация включает операции связи между входными и выходными параметрами, аксиомы с пред- и постусловиями.

Особенностями аксиоматических спецификаций являются:

- простота понимания;
- широкие возможности применения;
- возможность расширения системы;
- использование методов доказывания.

Аксиоматические спецификации поддерживают такие языки, как VDM, Anna и Z.

6.1.2. Темпоральные и параллельные спецификации

Эти спецификации явно определяют поведение, системные состояния и наборы событий в порядке их исполнения и синхронизации. Параллельные спецификации определяют действия в терминах параллельных событий. Спецификация включает состояния, переходы, события, упорядочения и синхронизацию. Особенности таких спецификаций — мощный механизм спецификации, уровень синхронизации спецификации.

Временны́е логические спецификации поддерживаются языками TLS (*Temporal Logic Specification*), Petri nets, GIL CSP, UML/OCL (диаграммы конечных автоматов, деятельности, коммуникации, взаимодействия, последовательности, синхронизации).

6.2. Формальные методы спецификации моделей систем

Формальные методы более всего связаны с математическими техниками спецификаций, верификации и доказательства правильности создаваемых программ. Эти методы содержат математическую символику, формальную нотацию, аппарат вывода и поэтому редко используются на практике рядовыми программистами. Кроме того, эти средства не вкладываются в процессы ЖЦ, в которых регламентированы все основные и дополнительные процессы (управление качеством, проектом, ресурсами и др.). За многие годы своего существования (более 20 лет) названные выше формальные методы — VDM, Z, RAISE и др. — начали использоваться в реальных проектах, а также в университетских и академических организациях.

Считается, что формальные техники трудно использовать практически, особенно в таких системах, как системы реального времени, где особенно важно применение формальных методов для создания более надежных программ, стоимость разработки которых возрастает, так как затрачивается много времени на спецификацию и верификацию.

Существует разрыв между бурно развивающимися инженерными методами (например, оценка качества, стоимости ПС, измерение процесса, продукта и др.) проектирования и формальными методами спецификации и верификации программ.

Следует отметить, эти средства не нашли отражения в документе SWEBOK. В связи с тем, что SEMAT основное внимание уделяет теории разработки сложных систем, данные формальные методы найдут применение.

Отсуствует и стандарт для одного из языков спецификации, поскольку требуется длительная практическая их апробация в конкретных условиях создания ПС и в соответствии с этапами ЖЦ.

Методы верификации программных компонентов используются в минобороны Великобритании для спецификации задач безопасности военных систем и проверки их на полноту и непротиворечивость, а также в атомной энергетике Канады при создании безопасных атомных электростанций и др.

В технике формального проектирования ПС отмечен значительный прогресс в создании обобщенного UML языка визуальной спецификации требований системы и модели ПрО. Этот язык прошел массовую апробацию в практически создаваемых системах и в конкретных инструментальных поддержках. В результате UML доведен до стандарта и отодвинул на второй план такие средства спецификации, как RAISE, несмотря на то что для него существует программная поддержка Raise Tools, которая задает спецификацию системы, ее верификацию и доказательство корректности свойств системы.

Далее рассматриваются некоторые техники спецификации моделей программ и методы формального доказательства.

6.2.1. Формальное описание моделей систем в языке Z

Приведем наиболее известные из этих моделей:

- обобщенная модель, компоненты которой задаются с помощью языка спецификации *Z*-схем;
- модель системы в виде ациклического графа, формально создаваемая с помощью композиционной теории;
- формальная модель модульной структуры, которая задается в виде графа, имеет язык его спецификации.

Обобщенная модель является формальной, создается с помощью языка спецификации *Z*-схем и включает в себя описание:

- создания узла управления моделью;
- назначения метки узла графа;
- задания внешних характеристик и параметров модулей.

Созданная модель включает в себя:

- совокупность *Z*-схем с набором деклараций и ограничений;
- инварианты схем для верификации модели.

На определение обобщенной модели повлияли следующие операции:

- выбор модуля из библиотеки шаблонов, переименование шаблонов или их конкретизация соответственно определению узла графа, где зафиксирован шаблон;
- связь модулей в модульную структуру системы, модули которой имеют интерфейс *порта* после конкретизации *слота* и отмечаются меткой соответствующей *Z*-схемы;
- верификация связей модулей, которая включает в себя проверку атрибутов и ТД;
- создание нового шаблона, инкапсуляция его интерфейса и соответствующей схемы.

Графовая модель создается с помощью композиционной теории в виде ациклического графа, узлы которого — модули, а дуги — интерфейс между модулями.

Каждому модулю модели соответствует сервис как провайдера (*provider*), так и потребителя (*consumer*) услуг. Дуга графа от модуля *M* к модулю *N* определяет интерфейс, который может предоставлять модуль *N* для модуля *M*.

Концепция сервиса поддерживается протоколами сети при взаимодействии компонентов, которые транспортируются через сеть в целях отправки множеств значений параметров серверу сети.

Базис формального метода представления модели — спецификация интерфейсов модулей для двух вышеказанных пользователей сервисов. Интерфейс удовлетворяет следующим свойствам:

- разделение (*separable*) модулей с точки зрения проектирования и спецификации, интерфейс которых не требует описания среды модуля;
- композиция (*composition*), когда модули соединяются с помощью механизмов композиционной теории в виде системы с гарантированными связями между ними.

Данная теория базируется на модели интерфейса для взаимодействия модулей системы между собою через сетевые протоколы (TCP, UDP и др.). Каждый протокол передает разным модулям этой модели параметры для нахождения требуемого сервиса.

Модель интерфейса задает связь модуля со средой в виде дискретного события (*event*), которое возникает только тогда, когда модуль и среда действуют вместе, и создают событие, рассматриваемое на стороне интерфейса.

Таким образом, интерфейс специфицирован как множество последовательностей событий для наблюдения за поведением модулей модели. Предположим, что *S* определяет спецификацию модуля *M*, которая включает в себя все возможные случаи его поведения и задает разные ситуации, которыми могут быть:

- событие интерфейса или состояние наблюдателя системы;
- анализ, который является конечным или представлен неопределенной последовательностью;
- формализм определения этой последовательности;
- условия выполнения события.

Эти предположения разрешают обеспечить разноуровневую систему взаимодействия модулей модели через механизм протоколов и систему наблюдения за событиями.

Проект модели VDM системы состоит из двух модулей: модуля управления CONT и модуля распределения памяти STOR.

Модуль CONT имеет следующую спецификацию действий:

$$\text{CONT} = (\text{coin} \rightarrow \text{reguest} \rightarrow \text{response} \rightarrow \text{choc} \rightarrow \text{cont}).$$

Общее назначение этой спецификации состоит в том, что потребитель может вставлять данные в *coin* для отправки запроса (*reguest*) модулю памяти. Этот модуль дает ответ (*response*) на запросы и имеет следующую спецификацию:

$$\text{STOR} = (\text{reguest} \rightarrow \text{response} \rightarrow \text{stor}).$$

Модель VM определяет параллельную обработку модулей CONT, STOR, а также связь со средой.

При этом спецификация VM имеет вид

$$\text{VM} = (\text{CONT} \parallel \text{STOR}) \setminus \{\text{reguest, response}\} = (\text{coin} \rightarrow \text{choc} \rightarrow \text{VM}).$$

6.2.2. Описание базовых конструкций языка VDM

Язык VDM появился в венской лаборатории как метод для описания языка ПЛ/1, разработки трансляторов и систем со сложными структурами данных[1]. Главная цель этого языка — спецификация программ и данных. Язык VDM предоставляет математическую символику для описания спецификации. В нем содержатся следующие ТД:

X — натуральные числа с нулем;

N — натуральные числа без нуля;

Int — целые числа;

Bool — булев тип;

Qout — строки символов;

Token — знаки и специальные обозначения операций.

Функция в языке — это определение свойств структуры данных и операций над ними, аппликативно или императивно. В первом случае функция специфицируется через комбинацию других функций и базовых операций (через выражения), что соответствует синониму «функциональный». Во втором случае значение определяется описанием алгоритма, что соответствует синониму «алгоритмический». Например, спецификация функции вычисления минимального значения из двух переменных имеет вид: $\min N_1 N_2 \rightarrow N_3$.

Алгоритмически описание значения этой функции имеет вид:

$$\min(x, y) = \text{if } x < y \text{ then } x \text{ else } y.$$

---

[1] См. об этом: *Bjorner D., Jones C. B.* The Vienna Development Methods (VDM): The Meta Language. Vol. 61 of Lecture Notes in Computer Science. Heiderberg : Springer Verlag, 1978.

Объекты языка VDM. Элементы данных, которыми оперируют функции, могут образовывать множества, деревья, последовательности, отображения, а также формировать новые более крупные объекты.

Множество может быть конечным и обозначаться X-set. При работе с множеством используются операции $\in$, $\subseteq$, $\cup$, $\cap$ и др. Язык включает в себя правила проверки правильности задания этих операций. Пример: $x \in A$ будет корректным только тогда, когда A является подмножеством из множества, которому принадлежит x.

Дистрибутивное объединение подмножеств имеет вид

$$\textbf{union}\ \{(1, 2), (0, 2), (3, 1)\} = (0, 1, 2, 3).$$

Списки (*последовательности*) — это цепочки элементов одинакового типа из множества X. Операция **len** задает длину списка, а **inds** — номер элемента списка.

Например, **inds** lst $=(i \in X\ [f \le i \le \text{len}])$.

К операциям относятся: взятие первого (головы) элемента списка — **hd** и остатка (хвоста) после удаления первого элемента из списка — **tl**.

Например, **hd** $(a, b, c, d) = (a)$, **tl** $(a, b, c, d) = (b, c, d)$.

Могут использоваться также операции конкатенации (соединение двух списков) и дистрибутивной конкатенации.

Дерево — это конструкция **mk**, позволяющая объединять в комплекс объекты разной природы (последовательности, множества и отображения). В деревьях могет использоваться также конструктор для обозначения составных объектов и деструктор для именования констант, вносимых в ранее определенный составной объект.

Например, пусть t — переменная типа Время, значением которой является 10:30, тогда конструкция имеет вид

let **mk** – Время $(h, m) = t$ **tin** определяет значение $h = 10$, а $m = 30$.

Отображение — это конструкция **map**, позволяющая создавать абстрактную таблицу из двух столбцов: ключей и значений. Все объекты таблицы принадлежат одному ТД — множеству. Операция dom строит множество ключей, а rng — множество всех его значений. Кроме того, есть такие операции: исключить строку, слить две таблицы и др.

Приведенные конструкции используются для спецификации программы и, в частности, начального состояния с инвариантными свойствами этого состояния. Инвариантным свойством (inv) является функция, содержащая описание типов аргументов, результата и самой функции. При спецификации программы средствами VDM задаются пред- и постусловия, аксиомы и утверждения, необходимые для проведения доказательства правильности специфицированной программы. *Предусловие* — это предикат с операцией, к которой будет обращаться программа после получения ее начального состояния или в случае нерегулярной ситуации.

Утверждение задает описание операций по проверке правильности программы в разных ее точках. Операторы программы изменяют ее состояние, а операции утверждений извлекают информацию из нее после изме-

нении состояния (например, после операции работы с БД). На основе этой информации устанавливается правильность или неправильность выполнения операции. При возникновении непредвиденной в аксиомах описания программы ситуации к программе должны предусматриваться определенные действия.

Постусловие — это предикат, который является истинным после выполнения предусловия, завершения текущей операции и выполнения инвариантных свойств программы.

Метод VDM ориентирован на пошаговую детализацию спецификации программ. На первом уровне строится грубая спецификация — модель программы в языке VDM, которая постепенно уточняется, пока не получится окончательный текст в ЯП. Разработка спецификации проводится по следующей схеме:

1) выбор терминов для спецификации программы;

2) описание понятий и объектов с использованием денотат, идентифицируемых с помощью некоторого имени;

3) описание инвариантных свойств программы;

4) определение операций (например, ввести объект, удалить и др.), которые изменяют состояние программы и сохраняют инвариантные свойства.

При переходе от одного шага детализации к другому модель программы детализируется и постепенно становится ближе к конечному описанию. Функции программы уточняются при детализации структуры программы и задания поведения модели.

Реальная разработка спецификации производится итерационно, на первом уровне она служит для проверки только свойств модели программы при заданных ограничениях и независимо от среды. Далее уточняются и расширяются спецификация и набор формальных утверждений. И так до тех пор, пока окончательно не будет завершен процесс спецификации программы и проведено доказательство.

Для демонстрации возможностей VDM языка рассмотрим задачу «Поиск», которая занимается поиском имени компонента C в каталоге (catalR) репозитария компонентов. Затем это имя сравнивается с заданным именем в запросе пользователя. В случае совпадения имен проверяются параметры, из каталога извлекается код компонента для передачи пользователю.

Спецификация переменных программы «Поиск»

1. repoz :: = developers : dl → **Init — set**
2. catalR : cat → **Init — set**
3. role : **inst** → facet
4. facet :: = autors : Milk
5. title : N
6. user :: = developer : **Itn**
7. free : **Bool**,

где developer — разработчик компонента *C*; facet — переменная, в которую посылается код компонента, выбранного из каталога catalR репозитория repoz при совпадении имен в каталоге и запросе; role — переменная, в кото-

рой хранится текущий элемент из репозитария, найденный по фасету компонента с номером *N* для user; autors : Milk — имя разработчика компонента; free : Bool — переменная, которая используется для задания признака: компонент не найден или к нему никто не обращался.

Описание инвариантных свойств программы

8. **type inv** — repoz: repoz → Bool
9. **inv** — repoz (dev) =
10. **let** mk repoz (*cd*, *c*, role) = dev **in**
11. (∀ *i* ∈ dom *cd*)
12. (∀ *i* ∈ *cd* (*i*))
13. ((∃ *j* ∈ dom *cd* & *i* ∈ *c*) & ∃ *a* ∈ elems role (*i*), autors (*a* ∈ dom *cd*) & free = false ⇔ developer = catalR & facet (N) = role).

Проверяется список имен в каталоге, который содержит *N* элементов, представляющих собой элементы типа **set**, совпадающие с именем в запросе, после чего результат сохраняется в role.

Доказательство инвариантных свойств программ должно проводиться автоматизированным способом с помощью специально созданных инструментальных средств поддержки VDM-языка.

Таким образом, можно сделать вывод о том, что методы систематического и теоретического программирования постоянно развиваются и пополняются новыми возможностям.

6.2.3. Описание динамических систем концепторным языком

Для постановки сложных математических задач (суммирование бесконечных рядов, теоретико-множественных операций с бесконечными множествами, гильбертова оператора и др.) и задач искусственного интеллекта (игры, распознавание образов и др.) предложен *общематематический процедурный язык*, так называемый концепторный язык (КЯ) — язык формального их описания[1]. В этом языке процесс описания сложной задачи проводится путем обоснования математической постановки решения задачи, формального алгоритмического описания.

Средства спецификации сложных задач

Основу КЯ составляет теоретико-множественный язык, который содержит декларативные и императивные средства теории множеств Цермело — Френкеля. Ядро включает набор элементов (типы, выражения, операторы) и средства определения новых типов, выражений и операторов.

Декларативные средства КЯ — это типизированный, многосортный логико-математический язык задания *выражений* и структуризации множества значений (денотат). Выражения состоят из термов и формул: термы обозначают объекты ПрО, а формулы — утверждения об объектах и отношениях между ними. К конструкторам составных типов и формул относятся: функторы, предикаты, конекторы и субнекторы.

---

[1] См.: *Коваль В. Н.* Концепторные языки. Доказательное проектирование. Киев : Наукова думка, 2001.

Функтор — это конструктор, преобразующий термы в термы. *Предикаты* превращают термы в формулы, *конекторы* включают в себя логические связки и кванторы для преобразования одной формулы в другую. Субнектор (дескриптор) — это конструктор построения термов из выражений и формул. Конструкторы термов — это традиционные арифметические и алгебраические операции над числовыми множествами и вещественными функциями. Конструкторы формул включают в себя предикаты, состоящие из предикатных и числовых символов, а также конекторы, состоящие из логических связок, кванторов и конструкторов теории множеств.

Императивные средства КЯ — это операторы и процедуры для описания объектов ПрО с помощью концепторов, состоящих из разделов для определения объектов решаемой задачи и действий над ними. Каждый концептор — это именованный набор определений и действий со следующей структурой описания:

концептор К (< список параметров >)
<список импортных параметров>
<определение констант, типов, предикатов>
<описание глобальных переменных>
<определение процедур>
начало К
<тело концептора>
конец К.

Концептор — это декларативное описание объектов и императивное описание операторов вычисления выражений тела. Рассматривается два случая: 1) декларативный концептор состоит из определений параметров и типов; 2) императивный концептор — это тело из операторов задач. Декларативные концепторы задают описание объектов и понятий, связанных с математической постановкой задачи, а описание метода ее решения задается с помощью императивных концепторов. Концепторное описание — это формальная спецификация задачи, которую можно трансформировать до алгоритмического описания и верификации.

Если полученный концептор неэффективен, то для повышения эффективности строится алгоритм, эквивалентный данному концептору, путем аппроксимации концепторного решения для замены неконструктивных объектов и неэффективных операций конструктивными и более эффективными аналогами.

Формализация КЯ

Общая схема формализации декларативной и императивной частей КЯ расширяется логико-математическими языками, традиционными структурными операторами (присваивание, последовательность, цикл и т.п.), а также теоретико-модельными (денотационными) и аксиоматическими средствами для формализации неконструктивной семантики КЯ.

Денотационный подход состоит в определении *семантики* языка путем подстановки каждому выражению соответствующего элемента из множества денотатов в функции φ интерпретации символов сигнатуры языка. Каждым константе $c \in C$, функциональному символу $f \in F$ и предикатному

символу $p \in P$ сопоставляется объект из множества денотат. Этот способ интерпретации семантики выражений и операторов языка аналогичен денотационной семантике ЯП. Главное отличие семантики КЯ от семантики программ — ее неконструктивность. С каждым КЯ можно связать некоторую дедуктивную теорию, которая отражает свойства концепторов.

Формальная дедуктивная теория строится путем выделения из множества всех формул подмножества аксиом и правил вывода. Для каждой пары R_1, R_2 формул дедуктивной теории и каждого оператора I создается *операторная* формула $\{R_1\}I\{R_2\}$ с утверждением: если R_1 истинно перед выполнением оператора I, то завершение оператора I обеспечивает истинность R_2, т.е. формула R_1 — предусловие, а R_2 — постусловие оператора I. С помощью неконструктивных объектов и неразрешимых формул этой теории можно адекватно описывать свойства неэффективных процедур.

Аксиоматическое описание КЯ — это аксиомы и утверждения относительно концепторного описания и проведения дедуктивного доказательства и верификации этого описания.

Логико-алгебраические спецификации

При использовании этих спецификаций ПрО представляется в виде алгебраической системы с помощью соответствующих носителей, сигнатуры и трех принципов. *Первый принцип* — описание логико-алгебраической спецификации ПрО и уточнение понятий, *второй принцип* — описание свойств ПрО в виде аксиом, которые формулируются в языке предикатов первого порядка и хорновских атомарных формул, и наконец, *третий принцип* — это определение термальных моделей из основных термов спецификации. Логико-алгебраические спецификации можно ограничить хорновскими формулами из-за простоты аксиом, упрощающих процесс автоматического доказательства теорем. Отношения в сигнатурах спецификаций заменяются булевыми функциями.

Доказательное проектирование КЯ

Средства концепторной спецификации сложных, алгоритмически не разрешимых задач положены в основу формализованного описания *поведения дискретных систем*. Для описания свойств аппаратно-программных средств динамических систем применяются логико-алгебраические спецификации КЯ. Техника описания таких систем включает два этапа.

На первом этапе дискретная система S рассматривается как черный ящик с конечным набором входов, выходов и состояний. Области значений входов и выходов — произвольные, а функционирование системы S — это набор частичных отображений и операций алгебраической системы. Они образуют *частичную алгебру*, формальное описание которой выполняется с помощью алгебраических спецификаций и является программой моделирования состояний дискретной системы[1].

На втором этапе система S детализируется в виде совокупности взаимозависимых подсистем $S_1, ..., S_n$, каждая из которой описывается алгебра-

[1] См.: *Коваль В. Н.* Концепторные языки. Доказательное проектирование.

ической спецификацией. В результате получается спецификация системы S из функций переходов и выходов, для которых необходимо доказывать корректность. Процесс детализации выполняется на уровне элементной базы или элементарных программ и сопровождается доказательством их корректности. В конечном счете получается система S, эквивалентная исходной абстрактной спецификации[1]. Рассмотрим один из примеров доказательства.

Пусть требуется построить спецификацию натуральных чисел из множества этих чисел и сигнатуры операций $\Sigma = (+, \times, \leq)$. При построении используются число 0 и функция следования s: $N \to N'$. Спецификация состоит из следующих аксиом:

1) $x + 0 = x$;
2) $x + s(y) = s(x + y)$;
3) $x \times 0 = 0$;
4) $x + s(y) = s(x \times y) + x$;
5) $0 \leq x$;
6) $x \leq y \supset s(x) \leq s(y)$;
7) $s(x) \leq s(y) \supset x \leq y$.

При этом алгебраические системы становятся многоосновными алгебрами, а аксиомы — спецификациями: тождественными и квазитождественными. Именно *алгебраические* спецификации являются наиболее используемыми, поскольку для них существуют относительно эффективные алгоритмы выполнения, которые преобразуют спецификации ЯП высокого уровня. Поэтому такие языки называют языками выполненных логико-алгебраических спецификаций. Их операционная семантика основана на *переписывании термов*, а создаваемая алгебраическая спецификация получает логическую семантику, используемую при доказательстве теорем.

6.3. Формальные методы доказательства программ

Формальные методы тесно связаны с математическими техниками спецификаций, верификацией и доказательством правильности программ. Эти методы содержат математическую символику, формальную нотацию и аппарат вывода. Правила доказательства громоздки, и поэтому на практике редко используются рядовыми программистами. Однако с теоретической точки зрения они развивают логику применения математического метода индукции при проверке правильности программ. На основе спецификации программ проводится частичное и полное доказательство правильности программ[2].

---

[1] Примеры доказательства таких систем приведены в работе: *Коваль В. Н.* Концепторные языки. Доказательное проектирование.

[2] См.: *Hoare T., Misra J.* Verified software: Theories, Tools, Experiments. Vision of Grant Challenge project. Microsoft Research Ltd and the University of Texas at Austin, 2005; *Jacobson I., Booch G., Rumbaugh J.* The Unified Software Development Process. N. Y. : Addison-Wesley, 1999.

Под доказательством *частичной правильности* понимается проверка выполнения свойств данных программы с помощью утверждений, которые описывают то, что должна получить эта программа, когда закончится ее выполнение, в соответствии с условиями заключительного утверждения. *Полностью правильной программой* по отношению к ее описанию и заданным утверждениям будет такая программа, которая частично правильная и заканчивается ее выполнение при всех данных, удовлетворяющих ей.

Для доказательства частичной правильности используется метод индуктивных утверждений, сущность которого состоит в следующем. Пусть утверждение A связано с началом программы, B — с конечной точкой программы и утверждение C отражает некоторые закономерности получения значений переменных в одной из точек каждого пути в программе. Если при выполнении программа попадает в i-ю точку и справедливо утверждение A_i, а затем она проходит от точки i к точке j, то будет справедливо и утверждение A_j.

Теорема 6.1. Если выполнены все действия программы методом индуктивных утверждений, то она частично правильна в соответствии с утверждениями A, B, C.

Требуется доказать, что если выполнение программы закончится успешно, то утверждение B будет справедливым. Согласно индукции, если при проходе через некоторую точку программы утверждение C будет справедливым, то при прохождении n-точек оно также будет справедливым. Таким образом, если программа прошла n-точку и утверждения A и B справедливы, то тогда всякий проход из n-й точки в $n + 1$ точку и утверждение A_{n+1} будут справедливыми, что и требовалось доказать.

6.3.1. Подходы к доказательству правильности спецификаций

Одним из современных направлений в области проверки правильности программ является использование формальных методов для доказательства их правильности, т.е. привлечение математического аппарата для доказательства того, что спецификация алгоритма программы задает решение некоторой задачи, для которой она разработана.

В формальных методах нет рутинного написания текста на ЯП, а есть анализ текста и описание поведения программы в стиле, близком математической нотации, путем рассуждений и доказательств, принятых в математике. Формальные методы в программировании появились одновременно с самим программированием, на которое оказали влияние работы А. А. Маркова, А. А. Ляпунова, Ю. И. Янова, а также К. А. Хоара (формальные нотации языка описания взаимодействующих процессов) и др.

В 1970-х гг. появились формальные спецификации, которые близки ЯП и предоставляют средства, облегчающие рассуждения о свойствах формальных тестов и сближающие их с математической нотацией. Несмотря на это, исследования формальных методов носили в основном академический, теоретический характер, поскольку извлечь из них практическую пользу в программировании соизмеримо с огромными затратами, связанными с описанием формальных спецификаций и проведением по ним доказательства правильности программ.

Формальное математическое доказательство программ основывается на спецификациях алгоритмов программ, аксиомах, описаниях некоторых условий, называемых предварительными предусловиями, утверждениями и постусловиями, определяющими заключительное условие получения правильного результата программой.

Спецификация — это формальное описание функций и данных программы, с которыми эти функции оперируют. Различают видимые данные, т.е. входные и выходные параметры, а также скрытые данные, которые не привязаны к реализации и определяют интерфейс с другими функциями.

Предусловия — это ограничения на совокупность входных параметров и **постусловия** — ограничения на выходные параметры. Пред- и постусловие задаются предикатом, т.е. функцией, результатом которой будет булевская величина (true/false). Предусловие истинно тогда, когда входные параметры входят в область допустимых значений данной функции. Постусловие истинно тогда, когда совокупность значений удовлетворяет требованиям, задающим функциональность. Постусловие фактически задает формальное определение критерия правильности получения результата.

Доказательство с помощью утверждений, которые составляются в формальном языке и служат способом проверки правильности программы в заданных ее точках. Создается набор утверждений, каждое из которых использует предусловия и последовательность инструкций, приводящих к проверке промежуточного результата относительно отмеченной точки программы, для которой сформулировано данное утверждение. Если утверждение соответствует конечному оператору программы, где требуется получить окончательный результат, то с помощью заключительного утверждения и постусловия делается окончательный вывод о частичной или полной правильности работы программы.

К традиционным методам проверки правильности относятся:

1) методы доказательства правильности программ;
2) верификация и валидация программ.

Далее рассматриваются особенности этих методов с учетом анализа конкретных работ[1] применительно к разным объектам ПС.

6.3.2. Языки формальной спецификации

Языки спецификаций, используемые для формального описания свойств программ, являются языками более высокого уровня, чем ЯП. Проведем их классификацию по следующим категориям: универсальные языки с общематематической основой (например, RAISE, Z, API, VDM и др.); языки спецификации отдельных проблемных областей (например, ЯП, трансляторы, БД и др.); специализированные языки спецификации (например, языки таблиц, логики, равенств и подстановок и др.); языки,

---

[1] См.: *Буч Г.*, *Рамбо Д.*, *Джекобсон А.* Язык UML. Руководство пользователя; *Burstall R. M.* Program proving as hand simulation with a little induction. Proc. IFIP Congress 74, North–Holland, 1974. P. 80—89; *Hoare T.*, *Misra J.* Verified software: Theories, Tools, Experiments. Vision of Grant Challenge project; *Jacobson I.*, *Booch G.*, *Rumbaugh J.* The Unified Software Development Process.

ориентированные на спецификацию параллельных процессов (например, CIP-L, Concurrent Pascal, Ada-68 и др.).

Спецификация программы — это точное, однозначное и недвусмысленное описание с помощью математических понятий и терминов, а также правил синтаксиса и семантики языка спецификации. В языке спецификаций могут быть понятия и конструкции, которые нельзя выполнить на компьютере, их необходимо представлять последовательностью операций, функций, понятных для интерпретации.

Описание задачи в языке спецификации включает в себя описание общего контекста всех понятий, через которые определяются другие понятия, участвующие в формулировке задачи или модели ПрО. Описание задачи представляется формально в виде аксиом, утверждений, пред- и постусловий и других формализмов, требующих для их реализации не систем программирования, а специального математически ориентированного аппарата для доказательства или верификации описания задач, в частности специальных интерпретаторов или метасистем.

Дадим краткую характеристику видов языков спецификации.

Универсальный язык спецификации имеет общематематическую основу, сущность которой поясним на примере языка VDM. Данный язык имеет следующие математические средства:

1) логические средства логики первого порядка, включая кванторы;
2) арифметические операции;
3) средства образования множеств, в том числе описание множеств с помощью логических формул и операций над множествами;
4) средства описания конечных последовательностей (кортежей, списков) и операции над ними;
5) средства описания конечных функций и операции над ними;
6) средства описания древовидных структур;
7) средства построения областей или множества объектов, включая произведения, объединения и рекурсивные определения;
8) определение функций с помощью выражений и равенств, включая рекурсивные определения;
9) процедурные средства ЯП (операторы присваивания, цикла, выбора, выхода);
10) операции композиции, аргументами и результатами которых могут быть функции, выражения, операторы.

В этом языке не хватает графовых средств, деревьев, средств управления и параллелизма. Однако содержащихся в языке VDM средств достаточно для формальной математической спецификации задач различного типа.

Общая характеристика языков RCL и Z приведена выше.

Языки спецификации областей включают в себя средства:

1) задания процессов управления, взаимодействия и параллельного выполнения;
2) спецификации ЯП и трансляторов;
3) спецификации БД и знаний;
4) спецификации пакетов прикладных программ и др.

Каждый из этих языков имеет специализированные средства, отображающие специфические особенности соответствующей области. Языки первого типа, в отличие от ЯП, позволяют специфицировать процессы управления вычислениями, передачей сообщений и взаимодействием объектов в распределенных системах. Лексический и синтаксический анализ трансляторов хорошо описывается языками регулярных выражений КС-грамматик в форме Бэкуса — Наура в виде спецификации контекстных зависимостей синтаксиса ЯП. Такого типа языки называются метаязыками. Для спецификации семантики языков используется формализм равенств. Техника описания ЯП в целях перевода основывается на атрибутных грамматиках и абстрактных типах данных. Задача описания ЯП и перевода — очень сложная, для ее описания используются денотационный, алгебраический и атрибутный подходы, а также логические термины с ориентацией на верификацию.

Языки спецификации — это языки с ориентацией на равенства и подстановки с операционной семантикой (Лисп, Рефал); логические языки; языки операций (APL) над последовательностями и матрицами; табличные языки; сети, графы и др.

Язык логики предикатов с набором базисных функций используется для записи пред- и постусловий, инвариантов и верификации. Отдельные операции логики предикатов используются также в языках логического программирования (например, Пролог).

Основой математических объектов являются равенства и подстановки. Для определения семантики равенства бывает трех классов: операционное, денотационное и аксиоматическое. Операционная семантика связана с подстановками (замена, продукция) и определяется в терминах операций, приводящих к вычислениям алгоритмов. При этом фиксируются порядок и динамика выполнения операций. Денотационный подход к семантике предпочитает статическое описание в терминах математических свойств объектов, а аксиоматической — специфицирует свойства объектов в рамках некоторой логической системы, содержащей правила вывода формул и (или) интерпретаций.

Продукция или правила подстановки общего вида — это $\lambda \rightarrow \rho$, где λ и ρ — произвольные слова в фиксированном алфавите. Нормальный алгоритм Маркова представляет собой упорядоченный набор правил, некоторые из них отмечены как завершающие. Применение правила $\lambda \rightarrow \rho$ к слову φ состоит в подстановке слова ρ вместо самого левого слова λ в φ. Вычисление заканчивается, когда применяется завершающее правило, порождающее одно слово.

Завершая рассмотрение разных типов языков спецификации, можно сказать, что средства этих языков оказывают влияние на верификацию программ, перевод спецификаций в формы, приближающие их к эффективному выполнению программ, и разработку новых моделей вычислений. Эти языки постепенно станут привычными ЯП.

6.3.3. Характеристика базовых методов доказательства

Наиболее известными формальными методами доказательства программ являются метод рекурсивной индукции или индуктивных утверж-

дений Флойда, метод структурной индукции Хоара, метод рекурсивных индукций Маккарти и др.[1]

Метод Флойда основан на определении условий для входных и выходных данных и в выборе контрольных точек в доказываемой программе так, чтобы путь прохождения по программе пересекал хотя бы одну контрольную точку. Для этих точек формулируются утверждения о состоянии и значениях переменных в них (для циклов эти утверждения должны быть истинными при каждом прохождении цикла-инварианта).

Каждая точка рассматривается для индуктивного утверждения того, что формула остается истинной при возвращении в эту точку программы и зависит не только от входных и выходных данных, но и от значений промежуточных переменных. На основе индуктивных утверждений и условий к аргументам программы создаются утверждения с условиями проверки правильности этой программы в отдельных ее точках. Для каждого пути программы между двумя точками устанавливается проверка на соответствие условий правильности и определяется истинность этих условий при успешном завершении программы на данных, удовлетворяющих входным условиям.

Формирование таких утверждений является довольно сложной задачей, особенно для программ с высокой степенью параллельности и взаимодействия с пользователем. Кроме того, трудно проверить достаточность и правильность самих утверждений.

Доказательство корректности применялось для написанных программ и тех, которые разрабатываются методом последовательной декомпозиции задачи на несколько подзадач, для каждой из них формулируются утверждения с учетом условий ввода и вывода в точки программы, расположенные между входными и выходными утверждениями. Суть доказательства истинности выполнения условий и утверждений относительно заданной программы и составляет основу доказательства ее правильности.

Данный метод доказательства способствует уменьшению числа ошибок и сокращению времени тестирования программы, а также проверки спецификаций на полноту, однозначность и непротиворечивость.

Метод Хоара — это усовершенствованный метод Флойда, основанный на аксиоматическом описании семантики ЯП исходных программ. Каждая аксиома описывает изменение значений переменных с помощью операторов этого языка. Основное внимание уделяется формализации операторов перехода и вызовов процедур с помощью правил вывода, каждое из которых задает индуктивное высказывание для каждой метки (точки) и функции исходной программы.

Оператор перехода рассматривается как выход из циклов и аварийных ситуаций. Данный метод имеет некоторые ограничения на параметры процедур. Система правил вывода дополняется механизмом переименования глобальных переменных и условиями к аргументам и результатам, а также правильностью задания данных программы.

---

[1] См.: *Лаврищева Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства.

Описание с помощью системы правил утверждений об операторах программы громоздко и отличается неполнотой, поскольку все правила предусмотреть невозможно. Данный метод проверялся экспериментально на ограниченном множестве программ, а средства автоматизации метода не были реализованы.

Метод рекурсивных индукций Маккарти состоит в структурной проверке функций, работающих над структурными типами данных, изменяет структуры данных и диаграммы перехода во время символьного выполнения программ.

Техника символьного выполнения включает в себя моделирование выполнения кода при использовании символов для изменяемых данных. Тестовая программа имеет детерминированное входное состояние при вводе данных и разных условий ее выполнения. Благодаря тому, что каждая строка кода должна выполняться самостоятельно, фиксируются состояния и значения переменных программы, а также проводится их проверка.

Выполняемая программа рассматривается как серия изменения состояний, т.е., каждой логической части программы соответствует упорядоченная серия изменения состояний. Самое последнее состояние части программы считается выходным состоянием, и если оно получено, то программа считается правильной. Данный метод обеспечивает высокое качество исходного кода.

Метод Дейкстры включает в себя два подхода к доказательству правильности программ. Первый подход основан на модели вычислений, оперирующей с историями результатов вычислений при работе программ, анализе пути прохождения и правил обработки большого объема информации.

Второй подход базируется на формальном исследовании текста программы с помощью задаваемых предикатов первого порядка, которые применяются к асинхронным программам. В процессе выполнения каждая из этих программ получает некоторое состояние, которое запоминается в так называемом сборщике мусора, который по окончании программы очищается.

Метод Дейкстры основывается на перевычислении, математической индукции и абстрактном описании программы.

Перевычисление используется для проверки границ вычислений проверяемой на правильность программы с помощью инвариантных отношений.

Математическая индукция применяется для организации прохождения циклов и рекурсивных процедур с помощью необходимых и достаточных условий утверждений повторных вычислений в программе.

Абстракция позволяет ослабить количественные ограничения, которые накладываются методом перевычислений.

Доказательство программ по данному методу можно рассматривать как доказательство теорем в математике с использованием аппарата математической индукции при формальном доказательстве правильности, а также как система статической проверки правильности программ с целью обнаружения в них ошибок.

Метод математической индукции позволяет доказать истинность некоторого предположения $P(n)$ в зависимости от параметра n для всех $n \geq n_0$

и тем самым доказать случай $P(n_0)$. Исходя из истинности $P(n)$ для любого значения n, доказывается $P(n + 1)$, что достаточно для доказательства истинности $P(n)$ для всех $n \geq n_0$.

Этот путь доказательства используется для утверждения A относительно программы, которая при своем выполнении достигает определенной точки. Проходя через эту точку n раз, можно получить справедливость утверждения $A(n)$ и провести доказательство:

1) что справедливо $A(1)$ при первом проходе через заданную точку;

2) если справедливо $A(n)$ (при n-проходах через заданную точку), то справедливо и $A(n + 1)$ при прохождении через заданную точку $n + 1$ раз.

Чтобы доказать, что некоторая программа правильная, надо корректно описать то, что эта программа делает, т.е. ее логику. Такое описание называется правильным высказыванием или утверждением. Утверждение о правильности программы будет справедливым, если она успешно завершится.

Проверка правильности программ — это автоматизированный метод Хоара в системе автоматизации — СПРУТ, основанный на системе проверки утверждений о программах на ЯП Паскаль.

На вход данной системы поступает программа на Паскале с аннотациями и точками для организации проверки программы. Генератор обрабатывает условия проверки правильности для осуществления доказательства арифметических и логических выражений.

В систему вводятся аксиомы и списки подцелей, с помощью которых устанавливается соответствие между аргументами аксиом и параметрами формул. Процесс завершается заключением о правильности программы или о найденных ошибках.

6.3.4. Модель доказательства корректности перестановки данных в векторе

Рассматривается формальное доказательство экспериментальной программы, заданной структурной логической схемой. Для нее составляются утверждения с помощью логических операторов, задаваемых комбинациями логических переменных (true/false), логическими операциями (конъюнкция, дизъюнкция и др.) и кванторами всеобщности и существования (табл. 6.1)[1].

Таблица 6.1

Логические операции

Название операции	Пример	Значение
Конъюнкция	x & y	x и y
Дизъюнкция	x * y	x или y
Отрицание	$\neg x$	Не x
Импликация	$x \rightarrow y$	Если x, то y

[1] См.: *Лаврищева Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства. М. : Издательство Юрайт, 2016. — 284 с.

Окончание табл. 6.1

Название операции	Пример	Значение
Эквивалентность	$x = y$	x равнозначно y
Квантор всеобщности	$\forall\, x\, P(x)$	Для всех x условие истинно
Квантор существования	$\exists\, x\, P(x)$	Существует x, для которого $p(x)$ истинно

Алгоритм программы состоит в построении для одномерного массива целых чисел T длины N (array T [1:N]) эквивалентного массива T' той же длины N, что и массив T, и его элементы должны располагаться в порядке возрастания их значений. Иными словами, цель данного алгоритма — сортировка элементов исходного массива T по их возрастанию. Доказательство правильности алгоритма сортировки элементов массива T проведем с использованием ряда утверждений относительно элементов этого алгоритма, который описывается П1—П6.

П1. Входное условие алгоритма задается в виде начального утверждения:

A_{beg}: (T[1 : N] — массив целых) & (T'[1 : N] массив целых).

Выходное утверждение A_{end} запишем как конъюнкции таких условий:

(а) (T — массив целых) & (T' — массив целых);

(б) ($\forall i$, если $i \le N$, то $\exists\, j$ ($T'(i) \le T'$ (j)));

(в) ($\forall i$, если $i < N$, то ($T'(i) \le T'$ ($i + 1$));

т.е. A_{end} — это

(T — массив целых) & (T' — массив целых)

& $\forall i$, если $i \le N$, то $\exists\, j$ ($T'(i) \le T'(j)$)

& $\forall i$, если $i < N$, то ($T'(i) \le T'(i + 1)$).

Для расположения элементов массива T в порядке возрастания их величин в массиве T' используется алгоритм пузырьковой сортировки, суть которого заключается в предварительном копировании массива T в массив T', а затем сортировке элементов согласно условию их возрастания.

Операторы алгоритма размещены в прямоугольниках. Условия выбора альтернативных путей представлены как условие, в кружках заданы точки с начальным A_{beg} и конечным A_{end} условиями и состояниями алгоритма, кружок с нулем задает начальное состояние, а с одной звездочкой — состояние после обмена местами двух соседних элементов в массиве T', кружок с двумя звездочками — состояние после обмена местами всех пар за один проход всего массива T'.

Кроме уже известных переменных T, T' и N, в алгоритме использованы еще две переменные: i — целое и M — булева переменная, значением которой являются логические константы true и false.

П2. Для доказательства того, что алгоритм действительно обеспечивает выполнение исходных условий, рассмотрим динамику выполнения этих условий последовательно в определенных точках алгоритма.

Заметим, что указанные точки делят алгоритм на соответствующие части, правильность любой из которых обосновывается в отдельности.

Так, оператор присваивания означает, что для всех i ($i \le N$ & $i > 0$) выполняется ($T'[i] := T[i]$).

Результат выполнения алгоритма в точке с нулем может быть выражен утверждением:

$$(T[1:N] - \text{массив целых}) \;\&\; (T'[1:N] - \text{массив целых}) \;\&\; (\forall i \text{ если } i \le N\,(T[i] = T[i])).$$

Доказательство очевидно, поскольку за семантикой оператора присваивания (поэлементная пересылка чисел из (T в T')) сами элементы не изменяются. К тому же в данной точке их порядок в T и T' одинаковый. Итак, получили, что выполняется условие «б» исходного утверждения.

Заметим, что первая строка доказанного утверждения совпадает с условием исходного утверждения A_{end}, которое остается справедливым до конца работы алгоритма и поэтому в следующих утверждениях приводиться не будет.

Оператор $(i < N)\; T'(i) > T'(i+1) \to T'(i)$ и $T'(i+1)$ изменяет местами элементы массива.

В результате работы оператора будет справедливым такое утверждение:

$$\exists\, i, \text{ если } i < N, \text{ то } T'(i) < T'(i+1),$$

которое является частью условия «в» утверждения A_{end} (для одной конкретной пары смежных элементов массива T').

Очевидно также, что семантика оператора обмена местами не нарушает условие «б» выходного утверждения A_{end}.

Алгоритм выполняет всевозможные замены данных в каждой паре смежных элементов массива T' за один проход через T', т.е. оператор обмена работал один или больше раз. Однако пузырьковая сортировка не дает гарантии, что достигнуто упорядочение за один проход по массиву T', поскольку после очередного обмена индекс i увеличивается на единицу независимо от того, как соотносится новый элемент $T'(i)$ с предшествующим элементом $T'(i-1)$.

В этой точке также справедливо утверждение:

$$\exists\, i, \text{ если } i < N, \text{ то } T'(i) < T'(i+1).$$

Часть алгоритма, обозначенная точкой с двумя звездочками, выполняется до тех пор, пока не будет упорядочен весь массив, т.е. не будет выполняться условие «в» утверждения A_{end} для всех элементов массива T':

$$\forall i, \text{ если } i < N, \text{ то } T'(i) < T'(i+1).$$

Итак, выполнение исходных условий обеспечивается соответствующей семантикой операторов преобразования массива и порядком их выполнения.

Доказано, что выполнение программы завершено успешно, что означает ее правильность.

П3. Этот алгоритм можно представить в виде серии теорем, которые доказываются. Начиная с первого утверждения и переходя от одного преобразования к другому, определяют индуктивный путь вывода, т.е. если

одно утверждение является истинным, то истинно и другое. Иными словами, если первое утверждение — A_1 и первая точка преобразования — A_2, то первой теоремой будет: $A_1 \to A_2$.

Если A_3 — следующая точка преобразования, то второй теоремой будет: $A_2 \to A_3$.

Таким образом, формулируется общая теорема: $A_i \to A_j$, где A_i и A_j — смежные точки преобразования.

Последняя теорема формулируется так, что условие, «истинное» в последней точке, отвечает истинности выходного утверждения: $A_k \to A_{\text{end}}$.

Следовательно, можно возвратиться потом к точке преобразования A_{end} и к предшествующей точке преобразования.

Доказали, что $A_k \to A_{\text{end}}$, а значит, верно $A_j \to A_{j+1}$ и т.д., пока не получим, что $A_1 \to A_0$.

П4. Далее специфицируются утверждения *if — then*.

П5. Чтобы доказать, что программа корректная, необходимо последовательно расположить все утверждения, начиная с A_1 и заканчивая A_{end}, что подтверждает истинность входного и выходного условий.

П6. Доказательство программы завершено.

6.3.5. Верификация и валидация программ

Верификация и валидация (*verification and validation — V&V*) предназначены для анализа, проверки правильности выполнения и соответствия ПО спецификациям и требованиям заказчика. Данные методы проверки правильности программ и систем соответственно означают:

- верификация — это проверка правильности создания системы в соответствии с ее спецификацией;
- валидация — это проверка правильности выполнения заданных требований к системе.

Верификация помогает сделать заключение о корректности созданной системы после завершения ее проектирования и разработки. Валидация позволяет установить выполнимость заданных требований и включает в себя ряд действий для получения правильных программ и систем, а именно:

- планирование процедур проверки и контроля проектных решений и требований;
- обеспечение уровня автоматизации проектирования программ CASE-средствами;
- проверка правильности функционирования программ методами тестирования на наборах целевых тестов;
- адаптация продукта к операционной среде и др.

Валидация выполняет эти действия путем просмотра и инспекции спецификаций и результатов проектирования на этапах ЖЦ для подтверждения того, что имеется корректная реализация начальных требований и выполнены заданные условия и ограничения. В задачи верификации и валидации входят проверки полноты, непротиворечивости и однозначности спецификации требований и правильности выполнения функций системы.

Верификации и валидации подвергаются:

- основные компоненты системы;
- интерфейсы компонентов (программные, технические и информационные) и взаимодействия объектов (протоколы и сообщения), обеспечивающие выполнение системы в распределенных средах;
- средства доступа к БД и файлам (транзакции и сообщения) и проверка средств защиты от несанкционированного доступа к данным разных пользователей;
- документация к ПО и к системе в целом;
- тесты, тестовые процедуры и входные данные.

Иными словами, основными систематическими методами правильности программ являются:

- *верификация* компонентов ПС и валидация спецификации требований;
- *инспектирование ПС* для установления соответствия программы заданным спецификациям;
- *тестирование* выходного кода ПС на тестовых данных в конкретной операционной среде для выявления ошибок и дефектов, вызванных разными недоработками, аномальными ситуациями, сбоями оборудования или аварийным прекращением работы системы (см. гл. 9).

Стандарты ISO/IEC 3918—99 и 12207 включают в себя процессы верификации и валидации. Для них определены цели, задачи и действия по проверке правильности создаваемого продукта (включая рабочие, промежуточные продукты) на этапах ЖЦ и соответствия его требованиям.

Основная задача процессов верификации и валидации состоит в том, чтобы *проверить и подтвердить*, что конечный ПП отвечает назначению и удовлетворяет требованиям заказчика. Эти процессы позволяют выявить ошибки в рабочих продуктах этапов ЖЦ, без выяснения причин их появления, а также установить правильность ПП относительно его спецификации.

Эти процессы взаимосвязанные и определяются одним термином — «верификация и валидация» (*V&V*).

При верификации осуществляется:

- проверка правильности перевода отдельных компонентов в выходной код, а также описаний интерфейсов путем трассировки взаимосвязей компонентов в соответствии с заданными требованиями заказчика;
- анализ правильности доступа к файлам или БД с учетом принятых в используемых системных средствах процедур манипулирования данными и передачи результатов;
- проверка средств защиты компонентов на соответствие требованиям заказчика и проведение их трассировки.

После проверки отдельных компонентов системы проводятся их интеграция, а также верификация и валидация интегрированной системы. Систему тестируют на множестве наборов тестов для определения адекватности и достаточности этих наборов для завершения тестирования и установления правильности системы.

Идея создания международного проекта по формальной верификации была предложена Т. Хоаром, она обсуждалась на симпозиуме по верифицированному ПО в феврале 2005 г. в Калифорнии. Затем в октябре этого же

года на конференции IFIP в Цюрихе был принят международный проект сроком на 15 лет по разработке «целостного автоматизированного набора инструментов для проверки корректности ПС».

В нем сформулированы следующие основные задачи:

- разработка единой теории построения и анализа программ;
- построение всеобъемлющего интегрированного набора инструментов верификации для всех производственных этапов, включая разработку спецификаций и их проверку, генерацию тестовых примеров, уточнение, анализ и верификацию программ;
- создание репозитария формальных спецификаций и верифицированных программных объектов разных видов и типов.

В данном проекте предполагается, что верификация будет охватывать все аспекты создания и проверки правильности ПО и станет панацеей от всех бед, связанных с постоянным возникновением ошибок в создаваемых программах.

Многие формальные методы доказательства и верификации специфицированных программ прошли практическую апробацию. Проделана большая работа международного комитета ISO/IEC в рамках стандарта ISO/IEC 12207:2002 по стандартизации процессов верификации и валидации ПО. Проверка корректности формальными методами разных объектов программирования является перспективной.

Репозитарий является хранилищем программ, спецификаций и инструментов, применяемых при разработках и испытаниях, оценках готовых компонентов, инструментов и заготовок методов. На него возлагаются следующие общие задачи:

- накопление верифицированных спецификаций, методов доказательства, программных объектов и реализаций кодов для сложных применений;
- накопление всевозможных методов верификации, их оформление в виде, пригодном для поиска и выбора реализованной теоретической идеи для дальнейшего применения;
- разработка стандартных форм для задания и обмена формальными спецификациями разных объектов программирования, а также инструментов и готовых систем;
- разработка механизмов интероперабельности и взаимодействия для переноса готовых верифицированных продуктов из репозитария в новые распределенные и сетевые среды для создания новых ПС.

Данный проект предполагается развивать в течение 50 лет. Более ранние проекты ставили подобные цели: улучшение качества ПО, формализация сервисных моделей, снижение сложности за счет использования ПИК, создание отладочного инструментария для визуальной диагностики ошибок и их устранения и др. Однако коренного изменения в программировании не произошло ни в смысле визуальной отладки, ни в достижении высокого качества ПО. Процесс развития продолжается.

Новый международный проект по верификации ПО требует от его участников не только знаний теоретических аспектов спецификации программ, но и высокой квалификации программистов для его реализации в ближайшие годы.

6.4. Подходы к верификации моделей систем и характеристик

Сформировалась теория моделирования изменений создаваемых ПС из готовых продуктов (семейство продуктов — Product Families или Product Lines). Она включает средства формального графового представления МХ, по которому осуществляется управление вариабельностью членов семейства с учетом требований и автоматизированная обработка функций системы с проведением анализа возникающих нерегулярных ситуаций и установлением соответствия между вариабельными точками программы для внесения изменений.

Метод верификации характеристик выполняется путем:

1) построения модели объекта. Необходимо построить модель, которая была бы адекватна верифицированному объекту. Полученная модель должна иметь небольшое число состояний, которое является условием ее эффективной проверки. Построенная модель для программных объектов имеет большое количество состояний, которые не подразделяются на управляющие и вычислительные состояния;

2) формулировки требований в терминах выбранной темпоральной логики. Выполнение этого этапа сталкивается с проблемами, поскольку требования должны формулироваться в терминах модели и необходимо обеспечение их адекватности начальной спецификации объекта;

3) верификации модели с целью проверки формальных требований. В сравнении с первыми двумя этапами этот этап достаточно хорошо автоматизирован.

6.4.1. Логическое описание свойств моделей и их анализа

Истоки формальной верификациии и доказательства правильности программ лежат в венском методе (VDM). В нем дана система доказательств программ с помощью формальных методов проверки правильности или неправильности объекта в соответствии со спецификацией свойств программ. Объектами верификации стали модель системы и характеристик ПрО, требования к разработке новой ПС семейства Product Lines. Появились новые методы, основанные на *model checking*[1].

Формальная проверка моделей объектов

В них свойства объектов, которые подлежат верификации в МХ, описываются средствами линейной темпоральной логики (*Linear Temporal Logic*, LTL) или логики деревьев вычислений (*Computational Tree Logic*, CTL).

Подходами к формальной верификации объектов являются *дедуктивный анализ* (верификация на основе логического вывода) и *верификация по модели* (*model checking*).

Дедуктивный анализ требует большого объема ручной работы и высокой квалификации специалистов. Он применяется редко и только для тех объ-

[1] См.: *Карпов Ю. Г.* Model Checking. Верификация паралельных и распределенных программ. СПб. : БХВ-Петербург, 2010.

ектов, которые являются функционально критическими (например, должны обеспечивать безопасность функционирования или защиту информации).

Верификация на модели *model checking* применима только к моделям объектов с конечным числом состояний. Особенность метода верификации на модели заключается в том, что проверка проходит автоматически и для ее выполнения не нужны особенные знания и время. Этот метод работает не с реальным объектом, а с его моделью. Верификация моделей состоит в формальной проверке того, удовлетворяет ли заданная модель объекта формальным спецификациям и выдает ли правильные результаты.

Жизнеспособность метода верификации по модели подтверждена тем, что группа его авторов (Э. Кларк, А. Емерсон и Д. Сифакис) была награждена премией Тьюринга в области информационных технологий. Суть метода верификации — математическая формулировка требований к создаваемым программам и алгоритмы формальной проверки требований (рис. 6.1).

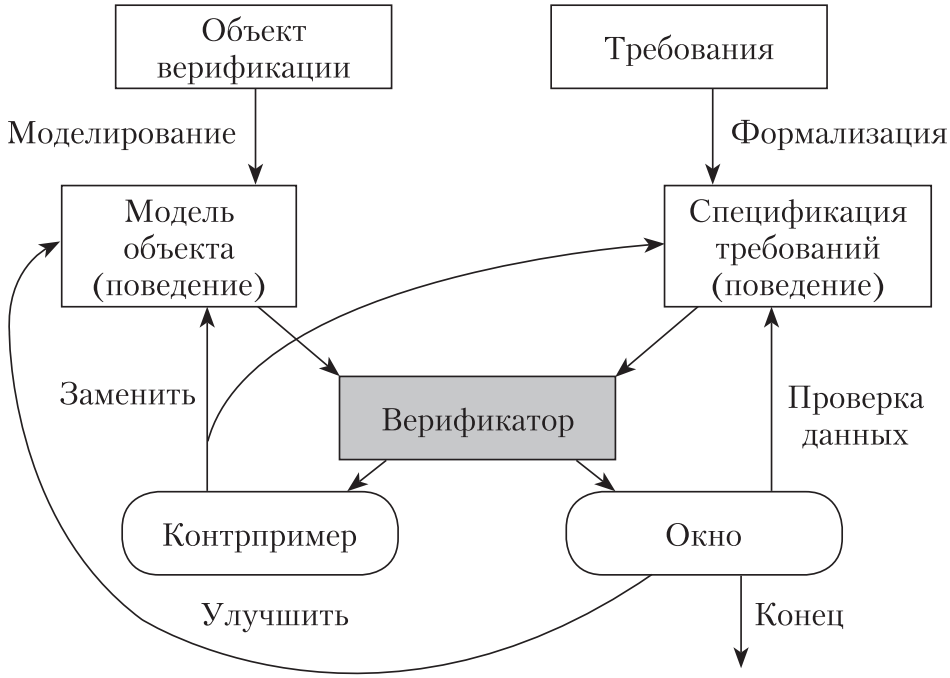


Рис. 6.1. **Схема метода *model checking***

Эта модель использует так называемую модель Крипке, которая формально задается в виде: $M = (S, S_0, R, L)$, где S — множество состояний, S_0 — множество начальных состояний — отношение переходов $L: S \to 2^{AP} x_0$ — функция разметки.

Как правило, спецификация поведения модели задается языком формальной темпоральной логики. Потом формулируются утверждения, истинность которых необходимо проверить. По описанию модели осуществляется автоматическая верификация, в результате которой устанавливается, что модель либо удовлетворяет спецификации, либо нет.

Хотя метод проверки на модели позволяет убедиться в том, что разработанная модель объекта соответствует заданной спецификации, однако определить, охватывает ли заданная спецификация все свойства объекта, невозможно. Разработчик модели абстрагируется от несущественных свойств

модели. Если модель удовлетворяет требованиям — верификатор сообщает об этом. Если же возникает ошибка, верификатор предоставляет контрпример, который показывает, при каких условиях могло возникнуть данное несоответствие. Пример представляет собой сценарий, в котором модель ведет себя нежелательным способом и показывает ошибки в модели, чтобы ее пересмотреть. Хотя это также может означать, что формальные требования неправильные, в том смысле, что верификатор проверяет то, чего разработчик не желал проверять.

Если ошибок не найдено, разработчик может усовершенствовать описание модели с учетом дополнительных свойств и размера модели.

Алгоритмы метода обычно базируются на полном пересмотре пространства состояний модели: для каждого состояния проверяется, удовлетворяет ли пространство сформулированным требованиям. Алгоритмы гарантированно завершаются, поскольку модель конечна.

Основное отличие этого метода от классической (хоаровской) верификации заключается в том, что свойства объекта проверяются динамически. В отличие от него классический метод проверяет, отвечает ли состояние переменных на выходе объекта условиям, которые накладываются на входное состояние переменных.

Основная сложность, которую приходится преодолевать в ходе проверки на модели, связана с эффектом «комбинаторного взрыва» в пространстве состояний. Эта проблема возникает, например, в системах, которые состоят из большого количества взаимодействующих компонентов, а также в системах, которые имеют структуры данных, способные принимать много значений.

Инструментальная поддержка этапов верификации выполнена в инструментах BLAST и JavaPathFinder (JPF).

Инструмент BLAST (*Berkeley Lazy Abstraction Software Verification Tool*) обеспечивает верификацию *C*-программ, которая проверяет, удовлетворяет ли *C*-объект требованиям к поведенческим свойствам интерфейсов. Инструмент и документация доступны по адресу http://mtc.epfl.ch/software-tools/blast. Разработан BLAST-плагин для Eclipse.

Java PathFinder (JPF) — это верификатор для Java-программ. В базовом виде представляет собой виртуальную машину Java Virtual Machine (JVM), которая используется для систематической проверки всех выполняемых путей программы и поиска таких нарушений, как блокировка (*deadlocks*) или исключительные ситуации (*unhandled exceptions*). В отличие от традиционных отладчиков JPF предоставляет информацию о полном пути, который приводит к дефекту. JPL — программное средство, которое может использоваться автономно или встраиваться в системы типа IDE. Доступен по адресу http://www.sourceforge.net/projects/javapathfinder/ или http://is.ifmo.ru/verification.

Верификация MX

Автоматический анализ моделей MX выполняется в два шага:

- модель MX транслируется в определенное *логическое представление* (*logic representation*);

• для изъятия информации из полученного логического представления количества возможных вариантов характеристик (членов СПС) используются существующие *решатели* (solvers), множество вариантов, которое удовлетворяет определенному критерию, поиску минимально используемой конфигурации и т.п.

На данный момент не существует оптимального логического представления и решателя для всех операций, которые могут выполняться относительно МХ, а возможности инструментов для анализа МХ зависят от конкретных решателей и их эффективность можно повысить, используя несколько решателей.

Исследование логического представления МХ проводится с помощью средств: CSP (*Constraint Satisfaction Problem*), SAT (*Boolean satisfiability problem* или *Propositional satisfiability problem*) и BDD (*Binary Decision Diagram*).

Представление модели CSP состоит из множества переменных, конечных областей значений этих переменных и ограничений относительно значений переменных. Задача CSP решается путем поиска состояний (значений переменных), в которых все ограничения удовлетворяются. Решатели CSP могут оперировать числовыми значениями, такими как области целых чисел. Это представление может использоваться для выполнения операций над МХ. Однако решатели программирования в ограничениях (Constraint Programming Solvers) неэффективны при выполнении некоторых операций на моделях, в частности при вычислении количества возможных комбинаций характеристик.

Более детально об CSP можно прочитать на сайте: http://www.ucalgary.ca/~sillito/ruby-csp/doc/classes/AI/CSP.html.

В свободном доступе есть реализация CSP в составе jOpt (*Java-based implementation Optimization Programming Language*) по адресу http://www.sourceforge.net/project/ showfiles.php?group id=112044.

Решатель jOpt's CSP solver основан на методе искусственного интеллекта согласования дуг (Arc Consistency), который охватывает создание графа из вершин и дуг из определенных пользователем переменных и ограничений. Этот граф используется для сокращения областей переменных в ходе достижения согласованности дуг (ограничений). jOpt's solver поддерживает разные алгоритмы сокращения областей на разных уровнях согласованности по широкому спектру типов доменов (целых, плавающих и т.п.).

Рассмотрена задача выполнимости булевых формул (SAT). Решение такой задачи заключается в определении выполняемости пропозициональной формулы или определении логического значения переменной, при который формула верна. Формула — это выражение, которое состоит из множества булевых переменных (литералов), соединенных операторами логики. Нет ни одного подхода, по которому атрибуты МХ могли бы транслироваться в SAT для выполнения таких операций, как максимизация или минимизация значений атрибута.

В свободном доступе по адресу http://www.sat4j.org есть библиотека SAT4J решателей в Java. Разработан новый пропозициональный механизм

в Eclipse, частью которого является эта библиотека (сообщение http://mail-archive.ow2.org/sat4j-dev/2008-03/msg00001.html).

BDD (*Binary Decision Diagram*) решений включает структуру данных, используемую для репрезентации булевой функции. BDD — это направленный ациклический граф, образованный группой вершин принятия решений (*decision nodes*) и двумя терминальными вершинами, названными 0-терминальной и 1-терминальной (один корень и два подкорня). Каждая вершина графа представляет переменную в булевой функции и имеет две дочерние вершины, которые отображают назначение переменной значений 0 и 1. Все пути от корня к 1-терминальной вершине представляют назначение переменных, для которых соответствующая булевая функция есть true (истина), а все пути к 0-терминальной вершине — назначение переменных, для которых булевая функция — false (ложь).

Хотя размер диаграмм может сокращаться по некоторым установленным правилам, слабым местом такого рода представлений является размер структуры данных, который колеблется в зависимости от упорядоченности переменных. Вычисление наилучшего построения переменных — сверхсложная (NP) проблема. Однако проблемы с памятью очевидны, компенсируются существенной эффективностью по времени, обеспечиваемой BDD-решателями.

В отличие от решателей CSP и SAT, которые не способны найти общее количество решений для MX среднего и большого размера за разумное время, решатели BDD могут это сделать по крайней мере относительно подсчета операций.

В свободном доступе по адресу http://javabdd.sourceforge.net/ содержится Java-библиотека для манипулирования диаграммами BDD.

FAM (*Feature Model Analyser*) — интегрированный анализатор характеристических моделей. FAM задается каркас (фреймворк), который автоматически проводит анализ MX. Он интегрирует некоторые из множества наиболее применимых логических представлений и решателей и благодаря этому обеспечивает оптимизацию процесса анализа.

Текущая версия FAMA (отсутствующая в свободном доступе) является полноценным инструментом для редактирования и анализа MX, поддерживает моделирование в стиле FODA, Feature-RSEB, экспорт/импорт MX с XML и XMI, интегрирует представление CSP, SAT и BDD, реализована как плагин для Eclipse.

Кроме указанных методов интерес представляет метод анализа MX, который базируется на дескриптивной логике (*description logic based analyses*). Его сущность заключается в трансформации MX в онтологию (OWL DL) и последующем применении автоматизированных средств рассуждения (*description logic reasoners*).

6.4.2. Инструменты верификации модели FM в Product Family

Модель характеристик формируется в процессе разработки ПП аналитиками и разработчиками и включает общие функциональные и нефункцио-

нальные характеристики элементов системы, которые используются членами Families при решении задач[1].

Модель вариабельности домена (*Domain Variability Model*) содержит набор элементов, помеченных вариантными точками, и их характеристики, по которым реализуется изменчивость объектов линии ПП (*Software Product Line*) на основе констрейнов, устанавливающих зависимости между изменяемыми объектами, используемыми при создании разных вариантов продуктов.

Модель вариабельности приложения (*Application Variability Model*) устанавливает связи зависимостей характеристик и констрейнов модели домена. Она вводится для каждого приложения ПП. Процесс моделирования модели вариабельности является базовым для всех ПС и семейства (домена) СПС.

Множество вариантов одной характеристики образуют коллекцию. Каждый член коллекции ассоциируется с разработанными артефактами в СПС и присоединяется к ядру СПС в точках вариантности, в которой будет использоваться в процессе выполнения системе. Вариантная характеристика транслируется в коллекцию вариантов из заданных точек вариантности в ПС. Обеспечение вариабельности ПС базируется на методах разработки и конфигурирования продукта (*Product Configuration*) из готовых reuses и процессов управления конфигурацией (*Configuration Management*), направленных на определение и внедрение эффективных процедур разработки и эволюции систем, а также их адаптацию к новым условиям функционирования в современных гетерогенных средах.

Анализ и особенности моделирования, проверки и верификации МХ семейства приводятся ниже.

Feature Model Plugin или **Feature Plugin** (FMP). Инструмент реализован как плагин для Eclipse. Он поддерживает моделирование, построение МХ и конфигурирования СПС на ее основе. Использует BDD-решатель для определения количества возможных комбинаций характеристик в МХ, но не имеет средств другого анализа характеристических диаграмм. Не поддерживает атрибутные модели характеристик (*attributed feature models*) и не может включать более чем один решатель для проведения анализа.

xFeature. Основан на XML-инструменте моделирования характеристик, также реализован как плагин для Eclipse. Поддерживает моделирование СПС и порождаемых ПС. Не поддерживает автоматического анализа МХ.

CaptainFeature. Это инструмент моделирования характеристик, который использует нотацию FODA для построения МХ. Включает интегрированный конфигуратор для специализации созданных моделей. Не поддерживает автоматический анализ моделей.

Requiline. Это инструмент инженерии требований для эффективного управления линиями ПП. Из описания группы характеристик и требований «выводится» конфигурация продукта. Включает функцию контроля

---

[1] См.: *Pohl K.*, *Bockle G.*, *Van der Linden F. J.* Software Product Line Engineering: Foundations, Principles and Techniques.

(*checker*) согласованности и не выполняет других операций анализа. Проводится в таких средах функционирования, как Microsoft.NET, Oracle DBMS.

Pure::Variants. Инструмент поддерживает моделирование характеристик, конфигурирование, базовые операции анализа с помощью средств Prolog-решателя с ограничениями (*constraint solver*).

AHEAD Tool Suite (ATS). Набор инструментов для разработки линий продуктов, поддерживает модуляризацию характеристик и их компоновку. Может выполнять определенные операции анализа МХ посредством SAT-решателей и сохранять ее в виде грамматики. Доступен по адресу http://www.cs.utexas.edu/~schwartz/ATS.html.

Из представленных инструментов FeaturePlugin поддерживает анализ МХ на примере (demo) использования верификатора FeaturePlugin (рис. 6.2).

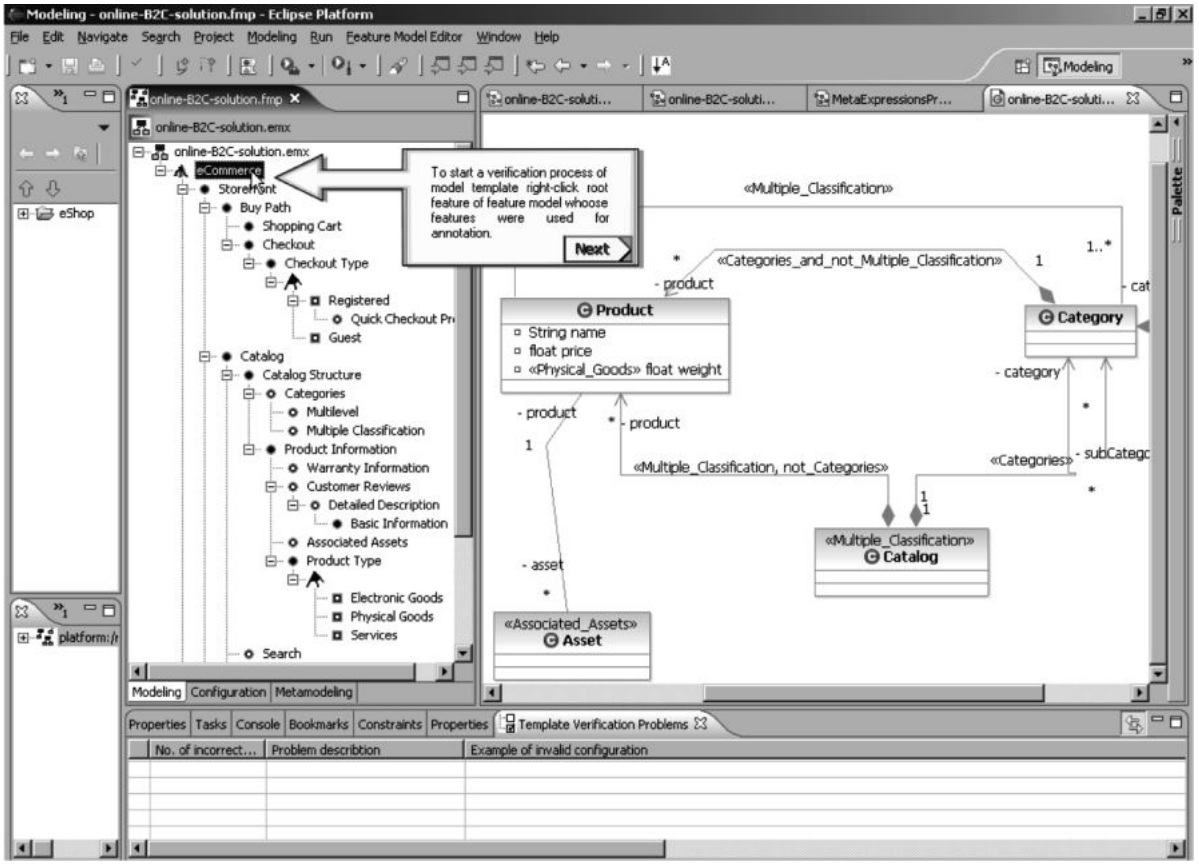


Рис. 6.2. **Общий вид компонента**

6.4.3. Корректность диаграмм характеристик онтологии OWL

При выборе определенной конфигурации характеристик для разрабатываемой ПС (члена СПС) в модели МХ необходимо проверять ее корректность, базируясь на ограничениях, определенных в этой характеристической модели. Поскольку модели могут эволюционировать с приобретением знаний о домене через добавление или изъятие характеристик, необходимо также проверять, сохраняется ли правильность конфигурации.

Инструмент для проверки корректности определенной конфигурации характеристик учитывает ограничение в модели на основе ряда требований:

• автоматическое выявление несогласованности;

• обеспечение эффективного по времени рассуждения для валидации конфигурации (путем постоянного реконфигурирования) системы характеристик;

• масштабируемость (порядок числа характеристик от десятков до тысяч);

• выразительность. Взаимосвязи между характеристиками (особенно в случае большого их количества) могут быть достаточно сложными, потому система рассуждения должна обеспечивать средства удобного представления и выполнения рассуждений, например графически средствами Builder C++;

• помощь при отладке моделей в плане предоставления объяснений об их несогласованности.

В уже упомянутой книге (*Biorner D., Jones C. B.* The Vienna Development Methods (VDM): The Meta Language. Vol. 61 of Lecture Notes in Computer Science) представлен подход к моделированию и верификации диаграмм характеристик с использованием OWL-онтологии Semantic Web. Предложены методы трансформации MX в онтологии и проиллюстрированы на примере MX для линии продуктов GPL (*Graph Product Line*, операция графами). GPL — семейство классических графовых приложений в домене CS, члены которого реализуют один или несколько алгоритмов работы с графами того или другого типа (*directed* или *undirected*, *weighted* или *unweighted* и т.п.).

Наиболее известный редактор онтологии — Protégé. С точки зрения верификации MX интерес представляет одна из версий этого инструмента — плагин Protégé-4.

Protégé-4 — редактор онтологии Semantic Web на Java, который предоставляет пользователю возможность создавать приложения. В частности, Protégé-OWL plugin[1] позволяет редактировать OWL-онтологию и подключать двигатель рассуждения (*reasoning engines*).

В статье «Description Logic Reasoners»[2] описано, каким образом Protégé-OWL взаимодействует с исследователем и как используется APE для выполнения рассуждений. Существует стандарт «DIG interface», который специфицирует общий способ подключения DL-исследователей (DIG — DL Implementation Group). Любой DIG-совместимый исследователь можно подключить как plug in в приложение, которое имеет DIG-интерфейс.

По адресу http://www.cs.man.ac.uk/~sattler/reasoners.html подан сжатый обзор популярных DIG-совместимых исследователей. Среди множества представленных в нем инструментов рассматривались лишь некоммерческие, основанные на C++ и Java, а среди них исследователи могут быть встроены в Protégé-OWL.

Для последующего исследования выбраны инструменты FaCT++, Pellet и RACER. Они поддерживают автоматический анализ подкатегорий клас-

---

[1] URL: http://protege.stanford.edu/.

[2] URL: http://protege.stanford.edu/plugins/owl/api/ReasonerAPIExamples.html.

сов и верификацию их согласования (*class subsumption and consistency reasoning*), а также выполнение некоторых запросов на OWL-онтологии. Фрагмент формы Protégé с выбором исследователя представлен на рис. 6.3.

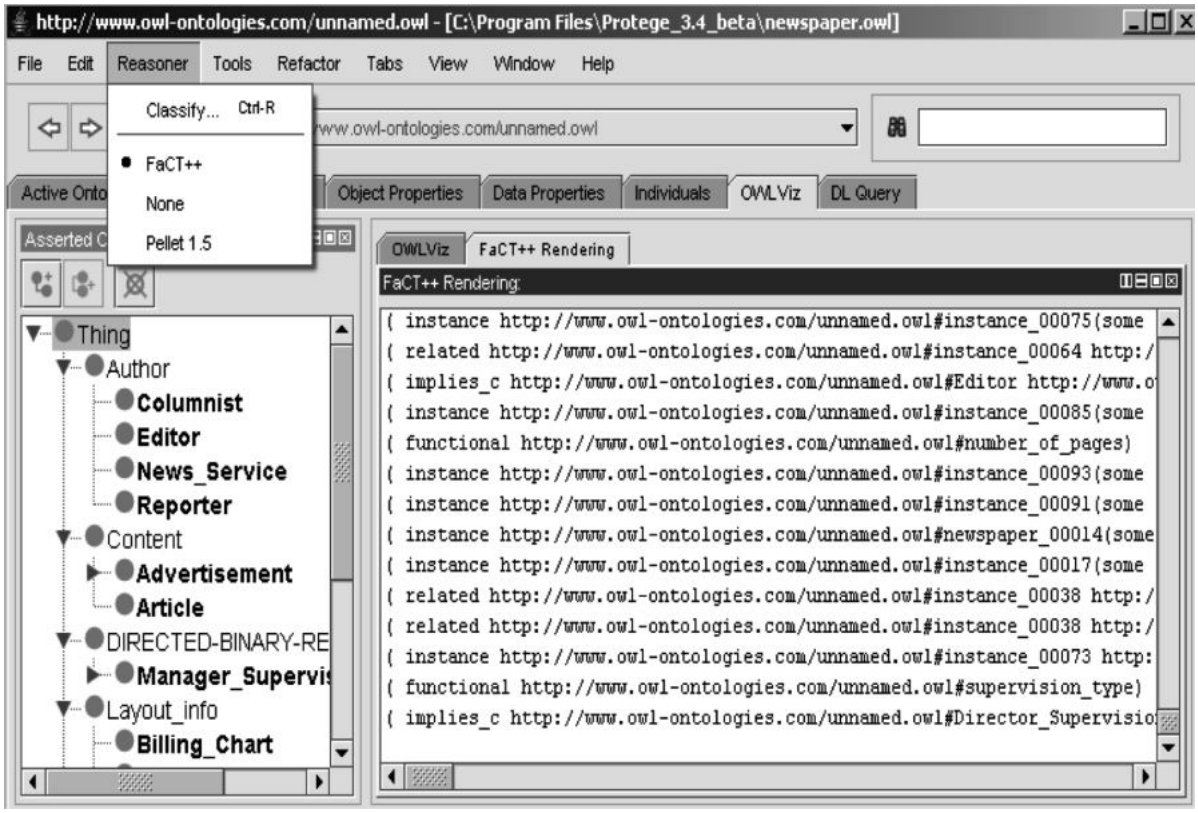


Рис. 6.3. **Форма с выпадающим меню для пользователя в Protégé-4**

FACT++ (*Fast Classification Terminologies*). Это новое поколение известного FaCT для OWL-DLngfe1049srsid10762568. Использует алгоритмы, заложенные в своем предшественнике, но имеет другую внутреннюю архитектуру. Реализован на C++, что повышает его переносимость на другие платформы.

Инструмент доступен в виде бинарного файла и открытого исходного кода. Для использования FaCT++ необходим компилятор GNU gcc v3.3 или выше, а также библиотека парсинга XML parsing library Xerces-C++, которая доступна по адресу http://xml.apache.org/xerces-c.

PELLET. Основан на Java исследователя для OWL DL с открытым кодом. Может использоваться вместе с библиотеками OWL API. Pellet API обеспечивает функции валидации, проверки согласованности онтологии, классификации таксономии, проверки ограничений прав использования онтологии и получения ответов на подмножество RDQL-запросов.

RACER (*Renamed Abox and Concept Expression Reasoner*). Это коммерческий (со временным trial-доступом) инструмент исследователя. Поддерживает не только OWL, но и формулировку правил, рассуждений относительно ограничений и получения ответов на запросы в других языках (например, в языке SPARQL). Racer имеет специальный язык запросов nRQL (*new Racer Query Language*).

Контрольные вопросы и задания

1. Назовите формальные методы проверки правильности.
2. Какие процессы проверки зафиксированы в стандарте?
3. Какие объекты входят в доказательство правильности программ?
4. Назовите основные методы доказательства корректности программ и базис этих методов.
5. Определите типы логических операций, используемых при логическом доказательстве корректности программ.
6. В чем смысл техники формального доказательства?
7. Сформулируйте основные задачи верификации и валидации программ.
8. В чем различие верификации и валидации?
9. Определите основы процесса тестирования.

Глава 7
ТЕОРИЯ И МЕТОДЫ ПРОЕКТИРОВАНИЯ МОДЕЛЕЙ ДОМЕНОВ И СИСТЕМ

В момент расцвета ООП и создания систем CORBA, UML, Rational Rose был сформирован оригинальный подход к моделированию систем из объектов и трансформации их к программным компонентам. Его основу составляет четырехуровневое моделирование системы в виде графа из объектов с помощью логико-математических операций метода ОКМ. Метод ОКМ апробирован в системе информатизации НАН Украины. Главным реализатором компонентной методологии метода ОКМ выступил В. Н. Грищенко[1].

Построение моделей ПрО (доменов) получило название концептуального моделирования. В нем каждая модель отображает присущую ПрО систему понятий, характерные их свойства, отношения друг с другом и правила поведения. Модель домена ориентирована на понимание ее семантики человеком. Она содержит существенные понятия (денотат — смысл понятия, концепт — семантика денатата), их размер, общие характеристики, их зависимости и содержание.

К современным средствам моделирования ПрО относятся: онтологические модели OWL (*Web Ontology Language*), ODM (*Ontology Definition Metamodel*); модели предметных областей — GDM (*Generative Domain Model*), FM (*Feature Model*), MDD (*Model Driven Development*), MDA (*Model Driven Architecture*), SOA (*Service-Oriented Model*); языки моделирования UML1, UML2 и др.; логико-алгебраические спецификации (RAISE, RCL, Z, VDM) моделей доменов в виде алгебраической системы из объектов, сигнатуры операций, аксиом и утверждений. Техника спецификации состоит в декомпозиции ПрО объектами с входами, выходами и состояниями, задаваемыми аксиомами и предикатами, алгебраическими операциями и операциями теории множеств.

Согласно ОКМ описание функций объектов трансформируется в программные компоненты с изоморфным отображением интерфейсных данных взаимодействующих объектов. Для многоразового использования функции объектов сертифицируются в языках WSDL, IDL как КПИ и размещаются в репозитории готовых артефактов и интерфейсов. Отдельные

---

[1] См.: *Грищенко В. Н.* Метод объектно-компонентного проектирования программных систем // Проблемы программирования. 2007. № 2. С. 113—125; *Его же*. Теоретические и прикладные аспекты компонентного программирования : автореферат дис. ... докт. физ.-мат. наук. Киев, 2007.

аспекты теории ОКМ с участием В. Н. Грищенко нашли отражение в следующих публикациях[1].

7.1. Теория моделирования систем из объектов на уровнях

ОКМ обеспечивает логико-математическое моделирование системы на четырех уровнях (обобщенном, структурном, характеристическом и поведенческом). Каждый уровень детализирует и уточняет объекты и задает модель системы в виде графа объектов и их отношения на множестве объектов ПрО с постепенным уточнением их денотатов и концептов по теории Фреге и определением характеристик объектов в MF. Задание структуры системы и действий над объектами (объединение, удаление, замена и др.) осуществляется математическими и логическими операциями ($\cup$, $\cap$, /, $\Diamond$, $\oplus$, – , $\supseteq$, $\rightarrow$, =, &, $\forall$, $\exists$, $\in$). На первом уровне проводится декомпозиция ПрО системы объектами. На следующих определяются их внешние характеристики FM и создается графовая объектная модель (ОМ), в вершинах которой находятся объекты, а дуги задают направление связи (интерфейсы) объектов, передаваемых между объектами. Для объектов определяется их поведение при выполнении функций в заданной среде. Объекты графа переводятся к программным компонентам, которые погружаются в компонентную среду. Объекты и компоненты этих моделей собираются в репозитории компонентной среды и могут конфигурировать в разные варианты ПП ПС. По аспектам вариабельности и разработке объектных систем, работающих с нестандартнми данными, защищено три диссертации.

Математическое моделирование объектной модели

Теория объектного моделирования ПрО построена с использованием денотационной теории Фреге и базовых понятий метода Г. Буча (классы, полиморфизм, наследование и др.). Объект выделяется на уровне анализа ПрО с привлечением логико-математических формализмов для задания и уточнения функций объектов, их характеристик в ОМ и отображения взаимоотношений и поведения объектов.

Согласно теории Фреге объект — это именуемая часть реального домена с определенным уровнем абстракции, заданная в виде денотат I, знаков II и концептов III (рис. 7.1). Каждый объект ОМ как сущность принадлежит некоторому множеству объектов $O = (O_0, O_1, ..., O_n)$, где $O_i = O_i\ (Na_i, Den_i, Con_i)$ и параметры означают — знак (имя), денотат и концепт объекта. Поведение концепта определяется на множестве предикатов $P = (P_1, P_2, ..., P_r)$ с помощью функции задания $Con_i = (P_{i1}, P_{i2}, ..., P_{is})$.

---

[1] См.: *Лаврищева Е. М.* Методы программирования. Теория, инженерия, практика; *Лаврищева Е. М.*, *Грищенко В. Н.* Сборочное программирование. Основы индустрии программных продуктов; *Лаврищева Е. М.* Теория объектно-ориентированного моделирования программных систем. URL: http://www.ispran.ru/preprints/docs/prep_29_2016.pdf.

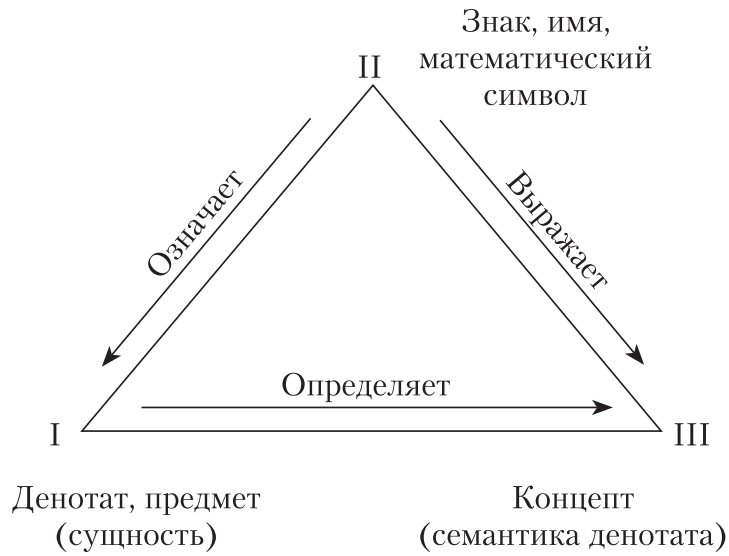


Рис. 7.1. **Треугольник Фреге для определения объектов**

Аксиома 7.1. Предметная область, которая моделируется из объектов, сама является объектом.

Аксиома 7.2. Моделируемая ПрО может быть отдельным объектом в составе другой ПрО.

При моделировании объект ПрО имеет хотя бы одно свойство или характеристику и уникальную идентификацию во множестве объектов и предикатов свойств и отношений.

Свойство объекта определяется на множестве объектов ПрО унарным предикатом, который получает значение истины при наличии внешней или внутренней характеристики объекта.

Характеристика — это свойство, определенное на множестве внешних и внутренних характеристик с условием получения истины из области значений их свойств.

Отношение определяется бинарным предикатом на множестве объектов ПрО, принимающих значение истины на заданной паре отношений типа $IS - A \mid Part - Of$.

Уровни логико-математического моделирования ПрО

Четырехуровневое объектное моделирование ПрО является развитием модульной парадигмы декомпозиции ПрО из функциональных и интерфейсных объектов и их определения и уточнения на I—IV уровнях логико-математическим аппаратом (рис. 7.2).

I. **Обобщающий уровень** определяет базовые понятия ПрО — объекты без учета их сущности и свойств.

II. **Структурный уровень** определяет место объектов в структуре модели ПрО и устанавливает отношения между ними.

III. **Характеристический уровень** задает общие и специфические свойства и характеристики концептов объектов.

IV. **Поведенческий уровень** определяет поведение и изменение объектов в зависимости от событий, которые они создают при их выполнении.

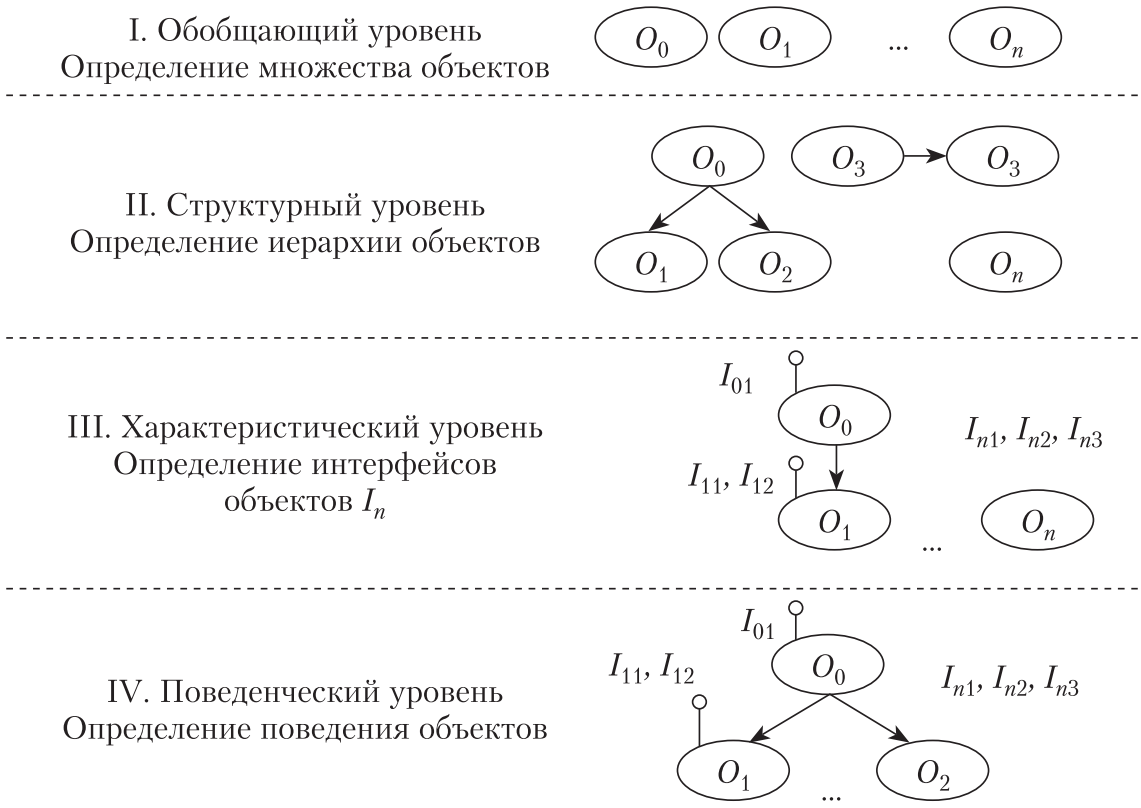


Рис. 7.2. **Уровни проектирования ОМ**

Каждый из уровней выполняет определенные функции абстрактного вида, которые рассматриваются ниже.

7.1.1. Обобщающий уровень проектирования

Этот уровень задает наивысшую меру абстракции отображения ПрО в ОМ с использованием рассмотренной концепции обобщения с целью определения сущностей ПрО и представления их в виде объектов ОМ как базовых понятий модели. Результат — это спецификации объектов вида:

<имя объекта> <концепт>,

где *имя объекта* — идентификатор из символьной строки литер и десятичных цифр; *концепт* — текст, который определяет <денотат объекта>.

Выделение сущностей осуществляется с учетом изменений, которые определяются соответствующими понятийными структурами — треугольниками Фреге — и фиксируются идентификаторами и концептами. Результат этого проектирования — ОМ, в которой размещены именованные объекты ПрО, которые образуют множество $O = (O_0, O_1, ..., O_n)$, где O_0 — объект ПрО и множество денотатов и концептов объектов после декомпозиции и композиции объектов ПрО.

Декомпозиционное изменение денотата. Формирование новых денотатов однородных объектов

$$decds(O_i): O_i \rightarrow \{O_{i1}, ..., O_{ik}\},$$

где $O_i = O_i\,(Name_i, Den_i, Con_i) \forall\, j\ Con_{ij} = Con_i; Den_i = Den_{i1} \cup ... \cup Den_{ik}$.

Формирование новых денотатов неоднородных объектов:

$$decdn(O_i) : O_i \rightarrow \{O_{i1}, ..., O_{ik}\},$$

где $O_i = O_i\,(Name_i, Den_i, Con_i)\ \forall\ j\ Con_{ij} = \varnothing$; $Den_i = Den_{i1} \cup ... \cup Den_{ik}$.

Композиционное изменение денотат. Композиция формирования однородных денотатов объектов:

$$comds(O_{i1}, ..., O_{ik}) : \{O_{i1}, ..., O_{ik}\} \rightarrow O_i,$$

где $O_i = O_i\,(Name_i, Den_i, Con_i)\ \forall\ j\ Con_i = Con_{ij}$; $Den_{i1} \cup ... \cup Den_{ik} = Den_i$.

Композиция формирования неоднородных денотатов объектов:

$$comdn(O_{i1}, ...,O_{ik}) : \{O_{i1}, ..., O_{ik}\} \rightarrow O_i,$$

где $O_i = O_i\,(Name_i, Den_i, Con_i)\ \forall\ j\ Con_i = \varnothing$; $Den_{i1} \cup ... \cup Den_{ik} = Den_i$.

Расширение концепта объекта. Если $P_t \in P$, $P_t \in Con_i\ P_t(O_i)$ принимают значение инстины, то $conexp(O_i, P_t) : O_i \rightarrow O_i$, где $O_i' = O_i'(Name_i, Den_i, Con_i')$, $Con_{ij} \cup \{P_t\} = Con_i$.

Сужение концепта объекта. Если $P_t \in Con_i$, то $connar\,(O_i, P_t) : O_i \rightarrow O_i'$, где $O_i' = O_i'(Name_i, Den_i, Con_i')$, $Con_{ij}' = Con_{ij} : P_t$.

Аксиома 7.3. Если $P_1 \in P$, $P_t \in Con_i\ P_t(O_i)$ принимает значение истины, то $conexp(O_i, P_t) : O_i \rightarrow O_i$, где $Con_{ij} \cup \{P_t\} = Con_i$.

Для множества объектов $O = (O_0, O_1, ..., O_n)$ выполняется отношение

$$\forall\ i\ [(i > 0)\ \&\ (O_i \in O_0)]. \qquad (7.1)$$

Между определенными элементами множества O существуют отношения принадлежности ($\in$). На этом уровне выделяются сущности ПрО и описываются как объекты в виде базовых понятий ОМ. Выделение сущностей осуществляют с учетом различий, определяющих соответствующие им понятийные структуры, исходя из треугольника Фреге. Объекты получают имена, идентификаторы и концепты. Результат этого уровня анализа — ОМ. В ней на данном уровне размещены только имена объектов ПрО.

7.1.2. Структурный уровень моделирования

Объекты уже определены на обобщающем уровне абстракции. Каждый объект задается как множество или элемент множества.

Пусть $O = (O_0, O_1, O_2, ..., O_n)$, исключая из множества O элемент O_0, который не принадлежит другим элементам, получаем множество $O' = (O_1, O_2, ..., O_n)$.

Выражение (7.1) на этом уровне трансформируют в следующее:

$$\forall i\ \exists j\ [(i > 0)\ \&\ (j > 0)\ \&\ (i \neq j)\ \&\ (O_i \in O_j)]. \qquad (7.2)$$

В выражении (7.2) каждый из объектов является множеством или элементом определенного множества и к ним применяются операции теоретико-множественной алгебры. Также это выражение определяет отношение «часть — целое», экземпляризации, агрегации.

Выражение $OS = (OS_1, OS_2, OS_m)$ и $OS \subseteq O'$. Тогда IS: $O' \to O$ есть ограничение тождественного отображения O' на себя. Пусть $S = (S_1, S_2, ..., S_m)$ — множество объектов, которое определено на данном уровне, и TS: $OS \to S$, где каждому OS_i соответствует S_i. Тогда определяется отображение между объектами на обобщающем и структурном уровнях $TS \cdot IS$: $O' \to S$.

Обозначим Ω — совокупность теоретико-множественных операций ($\cup$, $\cap$, /, $\Diamond$, $\oplus$), а $\Sigma = (O', \Omega)$ — алгебраическую систему для структурного уровня. Сущность теоретико-множественных операций:

- *объединение* $\cup(A, B)$, где A и B — объекты со статусом множеств. Результат применения операции: новый объект-множество, которое получено объединением множеств A и B;
- *пересечение* $\cap(A, B)$, где A и B — объекты-множества. Результат применения операций: новый объект-множество, которое является пересечением множеств A и B;
- *вычитание* $D = A \oplus B$ при $B \subset A$. Результат применения операции: новый объект-множество со всеми элементами A, которые не входят в B;
- *симметричное вычитание* $S = A : B$, где $A \cap B$ и не $A \subset B$, не $B \subset A$. Результат применения операции: новый объект-множество с множеством элементов, которые принадлежат A или («или» — распределенное) B.

В результате применения таких операций к каждой паре объектов проводится корректное выявление структурной упорядоченности и определение новых объектов, которые не выявлены на обобщающем уровне.

Таким образом, на данном уровне осуществляется решение следующих задач:

- определение и фиксация структурного порядка объектов;
- расширение описания объекта в структуре ПрО;
- определение новых объектов, связанных с заданными;
- формирование объектного графа ПрО.

Операция вычитания обеспечивает структурную упорядоченность объектов или системы их взаимодействий путем применения отношения принадлежности. Выделение объекта или фиксации его внешних различий осуществляется на основе анализа его структурного типа и определения места в объектном графе ПрО.

Результат проектирования на этом уровне — граф ОМ (рис. 7.3) с установленными отношениями следующего вида:

- множество вершин графа задает взаимно однозначное отображение множества объектов на уровне;
- для каждой вершины существует хотя бы одна связь (структурная) с другими вершинами графа;
- существует лишь одна вершина O_1 графа G, которая отображает ПрО в целом.

Построенный граф G корректируется и реструктурируется с помощью новых объектов и проведения контроля этого графа на структурную упорядоченность.

В процессе структурной упорядоченности граф G расширяется путем:

- *конкретизации объектов*, при которой объект получает внешнее свойство;

• проверки *принадлежности объектов* данному множеству объектов с помощью заданных свойств.

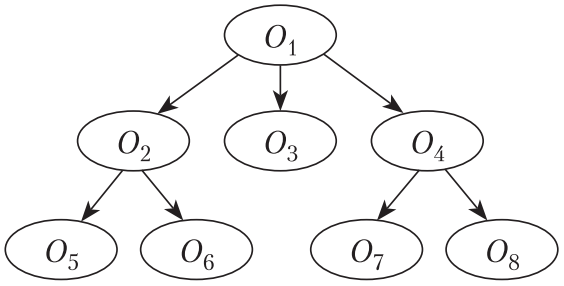


Рис. 7.3. **Структура графа $G = \{O\}$**

При решении задачи принадлежности задается критерий, общий для каждого объекта — элемента множества. Частным случаем отношения принадлежности является отношение «*Part—of*»,

Контроль полноты и неубыточности графа ОМ обеспечивает устранение продублированных объектов в разных множествах.

7.1.3. Характеристический уровень проектирования

Объекты определены на структурном уровне абстракции и на этом уровне используют логико-алгебраические операции для модификации объектного графа $G = \{O\}$ и конкретизации свойств, характеристик объектов и системы предикатов P над множеством объектов ПрО.

На входе — объектный граф G со спецификациями объектов. Результат уровня — откорректированный и расширенный новыми объектами граф ОМ с дополненными характеристиками и свойствами. Для задания характеристик и свойств (предикатов) строится алгебра.

Пусть $P = (P_1, P_2, ..., P_r)$ — множество предикатов на множестве S и $OA = (OA_1, OA_2, OA_k)$, где для каждого OA_i существует P_j, такой, что P_j на OA_i принимает значение истины.

Тогда $IA : S \to EA$ есть ограничение тождественного отображения S на себя. Пусть $A = (A_1, A_2, ..., A_k)$ — множество объектов, что определены на данном уровне, и $TA : OA \to A$, где каждому OA_i соответствует A_i, т.е. определяется отображение между объектами на структурном и характеристическом уровнях $TA \cdot IA : S \to A$ и $\Lambda = (A, P)$ определяется алгебры характеристического уровня, в соответствии с которой для каждого из объектов формируется его концепт. Если $O' = (O_1, O_2, ..., O_n)$ — совокупность объектов O, а $P' = (P_1, P_2, ..., P_r)$ — множество унарных предикатов, которые связаны со свойствами объектов O, то концепт объекта O_i является множеством утверждений, которые построены на основе предикатов с P', что принимают значение истины для соответствующего объекта. То есть $Con_i = \{A_{ik}\}$ и $P_k(O_i) = \text{true}$. Согласно структуре концептов объекты определяются отношением типа «род-вид».

Выражение $A = (O', P')$ определяет систему концептов объектов O' и предикатов P' с помощью 0-*арных*, *унарных* и *бинарных* операций.

Аксиома 7.4. Каждый объект ПрО имеет хотя бы одну характеристику, которая задает семантику и уникальную идентификацию во множестве объектов ПрО.

Для объектов определяются внешние и внутренние характеристики.

Внешние характеристики предназначены для задания проблемной ориентации объектов и их статуса как элементов множеств. Каждое внешнее свойство характеристики отмечается в FM.

Внутренние характеристики объекта предназначены для определения статуса объектов и критерия формирования множества из объектов, эквивалентных внешнему свойству.

Характеристики задаются в виде списка или множества свойств и могут соответствовать разным модельным характеристикам. Любой объект множества или элемент его может соответствовать только одной внешней и внутренней характеристике.

Основное условие — это принадлежность объекта к множеству эквивалентнх свойств объекта-множества.

На характеристическом уровне функциональным объектам определяются виды связи — интерфейс, в котором передаются внешние характеристики другим элементам, связанным на графе отношением принадлежности.

Формируется граф $G = \{O, I, R\}$, в котором O — множество объектов (функций); I — множество интерфейсов и отношений R (*relations*) между объектами. В граф G добавляются интерфейсные I (рис. 7.4).

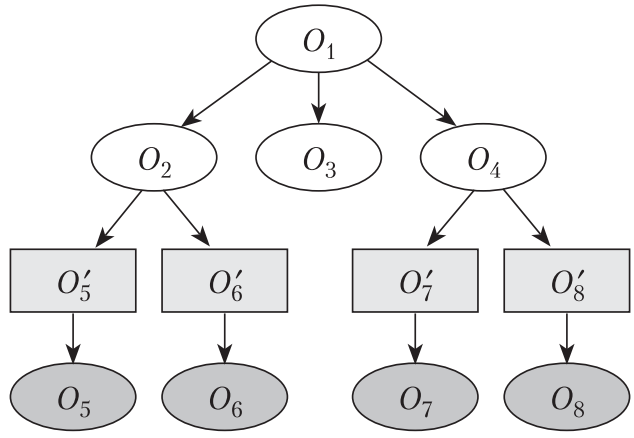


Рис. 7.4. **Граф *G* на множестве объектов и интерфейсов**:

O_i — локальные объекты; O_i — интерфейсные объекты;

O'_i — удаленные объекты

Вершины данного графа G задают функциональные объекты — O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8 и интерфейсные объекты — O'_5, O'_6, O'_7, O'_8, которые размещаются в репозитории, а дуги соответствуют отношениям между всеми видами объектов.

Элементы графа O_1—O_8 описываются в ЯП, а интерфейсные объекты O'_5—O'_8 — в специальном языке IDL (*Interface Definition Language*) системы CORBA. Параметры внешних характеристик интерфейсных объектов передаются между объектами через интерфейсы и помечаются как *in* (входной), *out* (выходной), *inout* (входной и выходной) в языке IDL.

Множество функциональных объектов графа G соответствует методу реализации этих объектов ПрО. При конкретизации объекты графа G уста-

навливают связь с другими объектами через интерфейсные параметры *in*, *out*, *inout* множества интерфейсов *I*.

Интерфейсным объектам графа соответствуют описание данных, методы их передачи по запросам к удаленным процедурам или методам и возможным операциям преобразования этих данных к соответствующим форматам среды выполнения.

Для графа *G* на рис. 7.4 можно построить программы P_1—P_5 с использованием операторов объединения (сборки) *link*:

1) $P_1 = O_2 \cup O_5$, *link* $P_1 = In\, O_5' (O_2 \cup O_5)$;
2) $P_2 = O_2 \cup O_6$, *link* $P_2 = In\, O_6' (O_2 \cup O_8)$;
3) P_3;
4) $P_4 = O_4 \cup O_7$, *link* $P_4 = In\, O_7' (O_4 \cup O_7)$;
5) $P_5 = O_4 \cup O_8$, *link* $P_4 = In\, O_8' (O_4 \cup O_8)$;
6) $P_0 = (P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5)$.

Результатом связи двух объектов графа (например, O_{25} и O_{47}) является интерфейсный объект, в котором множество входных интерфейсов совпадает с множеством интерфейсов объекта-приемника, а множество исходных интерфейсов — с множеством исходных интерфейсов объекта-передатчика.

Аксиома 7.5. Расширенный граф *G* с интерфейсными объектами структурно упорядочен (наверх), проконтролирован на полноту, избыточность, в нем устранены дублирующие элементы.

Объекты могут иметь несколько интерфейсов и наследовать интерфейсы других объектов в том случае, когда они предоставляют данные из множества входных интерфейсов.

Множество объектов и интерфейсов графа отмечаются общими характеристиками ОМ. Они считаются достоверными, если выполняется следующее условие: каждая внутренняя характеристика эквивалентна внешней характеристике объекта. Если это условие не выполняется, то такой элемент удаляется из множества *O* и графа соответственно.

Для реализации бинарного отношения «*Part—of*» пары «объект-элемент — объект-элемент» используются операции:

- *агрегации* для определения совокупности объектов, внешних характеристик с указанным исполнителем подмножеством свойств и порождения нового объекта-множества с внутренней характеристикой, которая состоит из этого подмножества;
- *детализации*, которая является обратной по отношению к операции агрегации и позволяет определить список объектов, из которых состоит агрегированный объект-множество.

7.1.4. Поведенческий уровень проектирования

Входными данными для этого уровня являются ОМ на предыдущих уровнях. На этом уровне определяется последовательность состояний объектов и процессы перехода состояний. Взаимосвязи между объектами формируются с помощью бинарных предикатов, которые связаны со свойствами объектов, устанавливающих взаимосвязи между объектами.

Данный уровень базируется на системе предикатов, определенных на подмножестве *SA* и множестве объектов *S*, которые отображают сущ-

ность объектов ПрО и определяют их поведение, без учета внутренних свойств.

Совокупность предикатов задается через $P = (P_1, P_2, ..., P_r)$. Тогда если поставить в соответствие множеству *SA* алгебраическую модель *A*, в которой *SA* является основным множеством, то на данном уровне абстракции объекты определяются как элементы множества *SA* с использованием предикатов с *P* и на этих элементах принимается значение «*истина*».

В общем случае этот уровень позволяет рассматривать зависимость от алгебры *A* (элементы множества *SA* и *P*) на характеристическом уровне, с отображением зависимости от времени реальных объектов ОМ и их характеристик.

Пусть $B = (B_1, B_2, ..., B_k)$ — множество объектов, которые определены на поведенческом уровне, и каждому A_i отвечает B_i ($A_i \rightarrow B_i(t)$). Тогда отображение *TB* определяется между объектами на характеристическом и поведенческом уровнях *TB*: $A \rightarrow B$.

Определение состояния объектов зависит от конкретного значения параметра времени на отрезке часовой оси (отрезок отвечает длительности существования объекта) $SA_j = SA_j(t)$, в которой SA_j — объекты — элементы множества *SA* с ассоциируемой совокупностью предикатов *P*, которые принимают на объектах значение «*истина*».

Теоретически существует зависимость самих предикатов относительно времени, т.е. зависимость вида $P_i = P_i(t)$. Однако для упрощения модели подобная зависимость в данной работе не учитывается, а возможные часовые изменения рассматриваются как новые предикаты.

Понятие времени — это абстрактное понятие, оно соответствует конкретному параметру системы, значения которого упорядочены, и каждому из них соответствуют состояния объектов.

Взаимосвязи между объектами формируются на основе бинарных предикатов, которые связаны со свойствами объектов *O*, и детализируются к уровню взаимосвязей между состояниями объектов. Зависимость от параметра времени достигается внесением в ОМ специального объекта *Timer*, основное назначение которого заключается в рассылке специальных сообщений текущего времени с определенным уровнем дискретности. Каждый объект имеет соответствующий метод, который анализирует полученное значение и выполняет переход к другому состоянию или оставляет его без изменений.

7.2. Операции над классами объектов

Объекты ОМ группируются в классы объектов. Класс — это определенное множество объектов, имеющих общие переменные, структуру и поведение. Объект является экземпляром класса. Поведение экземпляра определяется операциями его создания, уничтожения, сериализации и операциями над переменными класса. Совокупность внешних переменных и методов класса определяет интерфейс, с помощью которого экземпляры классов взаимодействуют между собой.

Одним из основных свойств объектов является инкапсуляция, которая реализуется с помощью интерфейсов, включающих метод и атрибуты.

Атрибут может определять внешние переменные экземпляра класса, которые используют методы выборки значения переменной (*get*-метод) и присвоения ему нового значения (*set*-метод). Каждый объект может иметь несколько интерфейсов, которые определяют его функциональные свойства. Упорядочение объектов ОМ выполняется с учетом отношения принадлежности, а элементов класса — с помощью множества целых чисел. Конкретизацией объекта являются:

- *класс* — объект, который задает собой множество;
- *экземпляр класса* — объект — элемент определенного множества, который сам является классом;
- *объединенный класс* — множество, которое является прямой суммой нескольких других множеств;
- *класс-пересечение* — множество, которое является общей частью других множеств;
- *агрегированный класс* — множество, которое является подмножеством определенного декартового произведения нескольких других множеств.

Объекты объединяются в классы в соответствии с общими характеристиками, т.е. ОМ имеет следующий вид:

$$OM = (Oclass, GK),$$

где $Oclass = \{Oclass_i\}$ — множество классов объектов функций или методов с общими свойствами; GK — объектный граф, который устанавливает связи и отношения между классами и его экземплярами.

Каждый класс представляется в виде:

$$Oclass_i = (ClassName_i, Meth_i, Field_i),$$

где *ClassNamei* — имя класса; $Meth_i = \{Methj^i\}$ — множество методов; $Field_i = = \{Field_n^i\}$ — множество переменных, которые определяют состояние экземпляров класса.

Пусть $Pfield_i \subset Field_i$ — множество внешних переменных (*public*). Каждому $Pfield_n^i \in Pfield_i$ соответствует метод *get* <$Pfield_n^i$> и *set* <$Pfield_n^i$> для задания и выборки значений соответствующих переменных как атрибутов объектов и интерфейсов ОМ, в которых методы объектов отображаются программными компонентами.

Множество методов имеет вид

$$Imeth^i = Meth_i \cup \{get <Pfield_n^i>\} \cup \{set <Pfield_n^i>\}.$$

Ему сопоставляется интерфейс $Ifunc_i$, который состоит из методов, входящих в $Imeth^i$.

7.3. Формальные основы объектного анализа

Алгебра объектного анализа имеет вид

$$\Sigma = (O', I', A'),$$

где $O' = (O_1, O_2, ..., O_n)$ — множество функциональных объектов; $I' = (I_1, I_2, ..., I_n)$ — множество интерфейсов; $A' = (A_1, A_2, ..., A_n)$ — множество (Action — A') операций над элементами множества O.

Каждая из операций A' имеет приоритет и арность и задает связь с соответствующими концептами объектов и операциями множества $A' = \{decds, decdn, comds, comdn, conexp, connar\}$, где *decds*, *decdn* — декомпозиции; *comds*, *comdn* — композиции и *conexp*, *connar* — сужение.

Семантика взаимодействия объектов графа G, функционирующего в среде «клиент — сервер», содержит: O — множество функциональных объектов $O = O_1, O_2, ..., O_k$; I — множество интерфейсных объектов *In*, *Out*, в котором *In* — множество входных интерфейсных объектов, где задаются данные клиента для передачи их серверному объекту $O_k \in O$, $In(O_k)$, и *Out* — множество выходных интерфейсных объектов сервера $O_k \in O$, $Out(O_k)$ и *Inout* — промежуточные интерфейсы.

Модель взаимодействия объектов основывается на свойствах и характеристиках объектов модели ОМ, которые представлены интерфейсами I в языке IDL (для задания параметров *In*, *Out*) и операциями принадлежности: $O_k \in O$, $In(O_k)$ — множество входных (*In*) интерфейсных объектов клиента; $O_k \in O$, $Out(O_k)$ — множество выходных (*Out*) интерфейсных объектов сервера.

Утверждение 7.1. Операция взаимосвязи двух объектов O_i, O_k дает объект, в котором множество *In* входных интерфейсов совпадает с множеством выходных интерфейсов *Out* объекта O_k, а множество выходных интерфейсов *Out* — с множества входных интерфейсов объекта-клиента O_i:

$$O_k = (Out(O_k), In(O_k)), O_l = (Out(O_l), In(O_l)),$$
$$O_k \cdot O_l = (Out(O_k), In(O_l)).$$

Утвержение 7.2. Взаимодействие объектов $O_k \cdot O_l$ является корректным, если объект-сервер O_k обеспечивает сервис, необходимый объекту-клиенту O_i, т.е.

$$\forall I_m \in In(O_k) \Rightarrow \exists I_n \in Out(O_l) \wedge I_m = I_n.$$

Удаленные объекты могут иметь несколько интерфейсов и наследовать интерфейсы других объектов ($O_k \leftarrow O_l$). Тогда последние предоставляют сервис всего множества исходных интерфейсов: $O_k \leftarrow O_l \Rightarrow Out(O_k) \subseteq Out(O_l)$.

В случае когда объект унаследовал другой объект, в котором множество выходных интерфейсов содержит все его интерфейсы, а множество входных интерфейсов содержит только интерфейсы, необходимые для предоставления сервиса, имеем

$$O_k \leftarrow O_l = \begin{pmatrix} Out(O_k) \cup Out(O_l), \\ \{I_m : (I_m \in In(O_k) \cup In(O_l)) \wedge \exists I_n \in Out(O_k \leftarrow O_l) : exec(I_n, I_m)\} \end{pmatrix}.$$

Над объектами и интерфейсами выполняются операции:

- проекции *объекта на интерфейс*, в котором множество интерфейсов *In* содержит один входной интерфейс, а множество интерфейсов *Out* — только те интерфейсы, которые необходимы для задания сервиса;
- проекция *объекта на объект* как проекция объекта на множестве интерфейсов объекта;

• проекция *объекта со взаимодействующим объектом* задается равенством:

$$(O_k \cdot O_l)[I_m] = O_k[I_m] \cdot O_l.$$

Унаследованный объект делегирует все интерфейсы и имеет свойства:

• *транзитивности* $\forall O_{1,2,3} \in O : O_1 \leftarrow O_2, O_2 \leftarrow O_3 \Rightarrow O_1 \leftarrow O_3$;

• *симметричности* $\forall O_k \in O \Rightarrow O_k \leftarrow O_k$.

Операция параллельного выполнения программ ПС в распределенной среде не всегда является симметричной: $O_k \leftarrow (O_{l_1} \| O_{l_2}) \neq O_k \leftarrow (O_{l_2} \| O_{l_1})$.

Результат отображения объекта на объект через интерфейс осуществляется с помощью множества входных $O_k[O_1] = O_k\ [In(O_i)]$ или выходных интерфейсов $O_k[O_1] = O_k\ [Out(O_i)]$.

Модель ОМ в конкретной распределенной среде выполняется с помощью сообщений, передаваемых на основе графа G в сеть от объекта через интерфейсную вершину к удаленному объекту и обратно. Такая схема взаимодействия поддерживается многими современными распределенными средами.

7.4. Определение моделей ОМ, ПС, МХ

Элементами модельной среды являются: модель ПрО, объектная, компонентная модель ПС и модель характеристик артефактов и системы[1].

Объектная модель выглядит так:

$$\text{ОМ} = \left\langle G_1^t; G_2^t; G_3^t; G_4^t \right\rangle,$$

где G_1 — граф объектов ПрО на обобщающем уровне ее проектирования ($t = 1$); G_2 — *FM* характеристического уровня ($t = 2$); G_3 — архитектурная (объектная и компонентная) модель ПС ($t = 3$); G_4 — интерфейс взаимодействия объектов на поведенческом уровне ($t = 4$).

Объектам функций G_1^t и их характеристикам соответствуют методы и данные (уровня 2, 3), необходимые для их реализации в ПС и обеспечения их взаимодействия.

Компонентная модель ПС — развитие ОМ, методы объектов которой реализуются программными КПИ для одного и только одного ее объекта и интерфейсами взаимодействия между ними. Она имеет следующий вид:

$$\text{КМ} = \langle RC, In, ImC, Fim \rangle,$$

где *RC* — базовые компоненты множества *C*; *In* — интерфейс компонентов; *ImC* — реализация базового компонента; *Fim*(...) — функции, превращающие интерфейс и множество элементов данных в сигнатуру интерфейса.

---

[1] См.: *Лаврищева Е. М.* Компонентная теория и коллекция технологий для разработки индустриальных приложений из готовых ресурсов // Труды Четвертой научно-практической конференции «Актуальные проблемы системной и программной инженерии», АПСПИ-2015, 20—21 мая 2015 г. С. 101—119; *Лаврищева Е. М.* Программная инженерия. Парадигмы, технологии, CASE-средства.

Определение 7.1. *Предметная область* — это совокупность понятий, концептов выделенных объектов и их характеристик. Объекты ПрО уточняются, структурно упорядочиваются посредством теоретико-множественных операций (селекции, пересечения и др.) и объединяются согласно общим признакам и характеристикам в классы и суперклассы.

Модель ПрО имеет вид:

$$M_{\text{ПрО}} = (M_{\text{о}}, Mi_{\text{ис}}, M_{\text{ох}}, M_{\text{ПС}}, P, D),$$

где $M_{\text{о}}$ — множество объектов и отношений между ними, заданных в ОМ; $Mi_{\text{ис}}$ — модель интерфейса объектов ОМ; $M_{\text{ох}}$ — множество общих характеристик объектов и внутренних, присущих каждому объекту и обеспечивающих конфигурационную сборку объектов ПрО; $M_{\text{ПС}}$ — модель ПС, которая реализует задачи и функции ПрО; P — множество предикатов с порядком и условиями выполнения объектов, их функциональных и нефункциональных характеристик модели свойств FM и взаимосвязи объектов ПрО, методы которых обеспечивают их программную реализацию в ПС; D — множество данных ПрО, которые необходимы для выполнения отдельных компонентов и ПС и могут сохраняться в БД.

Определение 7.2. *Программная система* (*ПС*) — это совокупность различных программных артефактов (объектов, компонентов, сервисов), их реализаций и конфигурационной сборки в заданной среде. Семейство программных систем (СПС) — это совокупность ПС, которые определяются общим множеством понятий и характеристик, присущих каждому отдельному члену СПС.

Модель ПС — это множество артефактов, КПИ, функций (объектов), интерфейсов и данных:

$$M_{\text{ПС}} = (C_L, M_f, Ms, M_i, M_d),$$

где C_L = компонентные элементы в языках $L = L_1, L_2, ..., L_N$; $M_f = (O_1, O_2, ..., O_r)$ — множество функциональных объектов ПрО; $Ms = (Ms_{in}, Ms_{out}, Ms_{inout}\}$ — множество сервисов (s) по обработке интерфейса — входного Ms_{in}, выходного Ms_{out} и общих данных Ms_{inout}; M_i — множество интерфейсов в языке IDL; M_d — множество данных и метаданных ПС.

Определение 7.3. *Вариантность* — это способность модели ПС к замене некоторых функциональных КПИ системы новыми артефактами или ПС путем конфигурирования новых ПС или семейств. Она обеспечивается FM, где каждая характеристика ассоциируется с функцией новых КПИ в ПС, который присоединяется или заменяется в точках вариантности.

Модель вариантных характеристик имеет вид

$$M_{VC} = (SV; AV); SV = (G_t, TR_t), t = 2, ..., 5; C; D;$$

$$AV(idm) = (g_t, tr_t); t = 2, ..., m; (p_t, tr_t); t = m + 1, ..., 5; c; d\rangle,$$

где SV — подмодель вариабельности в структуре семейства ПС/ПП; AV — подмодель вариабельности в артефактах (и ПС/ПП) семейства; $G_t = (F_t, L_t)$ — граф артефактов уровня $t = 1, ..., 5$, обусловленный моделью характе-

ристик семейства ПС/ПП; TR_t — двусторонние связи трассируемости артефактов уровней $t-1$ и t; C и D — предикаты на $\otimes F_t$, задающие ограничения и зависимости артефактов; g_t и p_t — подграфы G_t, описывающие артефакты, реализуемые артефактом типа m с идентификатором idm и реализующие его при разработке ПС/ПП; остальные элементы $AV(idm)$ — сужения соответствующих элементов SV.

Управление вариабельностью семейства ПС/ПП (в его структуре и артефактах) — основа разработки изменяемых ПС/ПП из ГоР.

Подмодель AV служит определению процесса разработки ПС из КПИ, которые имеют паспорта, сохраняемые в репозитории. Подмодель SV отображает функциональные и вариантные характеристики КПИ и продукта, а также аспекты отношения между ними на разных уровнях модели. Модель SV конкретизируется в специальную линию разработки КПИ и конфигурирования их в ПС.

Определение 7.4. *Точка вариантности* — это место в интерфейсе модели ПС, где осуществляется выбор варианта системы. Вариантная характеристика представляет собой множество вариантов или определенное количество точек вариантности ПС.

Точки вариантности обрабатываются конфигуратором и позволяют трансформировать ПС путем замены одних КПИ другими, более функциональными или корректными.

Вариантным характеристикам объектов в графе G_1 соответствуют варианты методов и данных (в графах второго и третьего уровней G_2 и G_3), поддерживающие взаимодействие этих объектов в изменяемых ПС/ПП.

Процесс управления вариабельностью имеет следующий вид.

1. *Идентификация вариабельности СПС*. Вариантная характеристика предусматривает наличие совокупности ее вариантов и точек вариации.
2. *Ограничение вариабельности СПС* задается с учетом текущих потребностей и перспектив эволюции.
3. *Реализация вариабельности*. В зависимости от принятых ограничений может быть выбрана соответствующая технология реализации точек вариации.
4. *Управление вариабельностью* (см. рис. 3.4).

Результатом сопоставления определений алгебраических систем на каждом уровне и изоморфизма этих систем является

Теорема 7.1. Отображение объекта множества O есть его реализация во множестве компонентов C.

$\tau : O \to C$ является изоморфизмом объектной и компонентной моделей ПС как алгебраических моделей типа $\langle 2, 2, 2, 2, 2\rangle$

$$\langle O; CH_O, VC_O, IM_O, EQ_O, EX_O\rangle;$$

$$\langle C; CH_C, VC_C, IM_C, EQ_C, EX_C\rangle.$$

Следствие 7.1. Отображение, которое сопоставляет объекту его интерфейс (КПИ) и для которого выступает входным, является изоморфизмом алгебраических моделей типа 2 (O; R), (I; R) и (I; R), (C, R). Эти модели превращают точки вариабельности / варианты сложных объектов в точки вариабельности / варианты интерфейсов и КПИ в системе.

Приведенные модели метода моделирования систем из артефактов (*reuses*, *object*, *services*, *components*) служат основой их трансформации к программному виду, который получается путем конфигурации моделируемой системы на основании *модели конфигурации*, имеющей вид

$$M_{konf} = (\text{ОМ}, M_{\text{ПрО}}, M_{\text{ПС}}, FM_{var}, M_{\text{вз}}),$$

где $M_{\text{вз}}$ — модель взаимодействия отдельных элементов системы.

На основании модели конфигурации осуществляются:

- формирование набора артефактов и ресурсов ПС для ПрО;
- выделение общих и вариантных характеристик ПС для построения FM и модели ПрО;
- набор элементов ПС или подбор ГоР, накопление их в базе конфигурации;
- планирование многократного использования ресурсов для ПС в точках вариантности, (подобно use case UML или точек вариабельности в стандарте SEI IEEE—15534), используемых для внесения или удаления некоторых функций внешними и внутренними операциями;
- реализация модели вариабельности из артефактов и ресурсов, в том числе КПИ;
- сборка ресурсов в ПС и их адаптация к новым условиям среды функционирования;
- управление вариантами ПС с заменой отдельных функций в ПС;
- управление моделью взаимодействия в гетерогенной среде.

Модель взаимодействия имеет вид

$$M_{\text{вз}} = \{M_{\text{пр}}, M_{\text{сис}}, M_{\text{сред}}\},$$

где $M_{\text{пр}} = \{Com, Int, Pro\}$ — модель программного ресурса; $M_{\text{сис}} = \{FPC, Int, Pro\}$ — модель системы; $M_{\text{сред}} = \{Envir, Int, Pro\}$ — модель среды.

Модель взаимодействия базируется на интерфейсе и механизмах передачи данных по сети для обеспечения взаимодействия систем, миграции элементов систем или ПС из одной среды в другую и для вычисления задач на определенных данных.

В качестве примера студенты МФТИ (А. Островский[1] и И. Радецкий[2]) реализовали модели взаимодействия систем:

Visual Studio ↔ Eclipse,

CORBA ↔ Visual Studio,

IBM ↔ Eclipse.

7.5. Теория моделирования ПС из компонентов

Одна из ключевых задач компонентного проектирования систем — построение КПИ, которыми могут быть программы, модули и объекты,

---

[1] *Островский А. В.* Подход к обеспечению взаимодействия программных сред Java и Ms.Net // Проблемы программирования. 2011. № 2. С. 37—44.

[2] *Радецкий И. О.* Один подход к обе6спечению взаимодействия сред Ms.Net и Eclipse // Проблемы программирования. 2011. № 2. С. 45—52.

пригодные для повторного использования в новой разработке ПС. Проектирование системы из компонентов — естественное расширение ООП. Компоненты не являются объектами, а предоставляют им необходимые ресурсы и сервис. На данный момент сформировались теория о компонентах, их спецификации и определении интерфейсов, способах их сборки и конфигурирования в систему.

Для моделирования ПС разработан новый метод ОКМ. Он обобщает понятие объектов как элементов реального мира и основывается на свойствах и характеристиках, которые определяются и уточняются с помощью логико-математических формализмов задания концептов и необходимых и достаточных свойств и характеристик, которыми объекты отличаются друг от друга. Метод обеспечивает отображение объектов ОМ в программные компоненты и интерфейсы, осуществляя последовательный переход от объектов к компонентам и их интерфейсам[1].

В нем конфигурационная сборка программных элементов выполняется средствами формального математического аппарата, который включает модели компонента, компонентную среду и компонентную систему. Эти модели реализуются с помощью компонентной алгебры, а также формализмов описания интерфейсов и выполнения системы преобразования передаваемых ТП на основе заданной сигнатуры интерфейсов взаимодействующих компонентов в ПС.

Рассмотрим множество программных компонентов $C = (c_1, c_2, ..., c_r)$, где $c_i \in C$ — произвольный компонент, использующийся при разработке ПС. Каждый компонент характеризуется некоторой совокупностью свойств (например, расширяемость, взаимозаменяемость и др.).

Пусть $P_1, P_2, ..., P_k$ — предикаты свойств компонентов на множестве C, т.е. $P = <P_1(C), P_2(C), ..., P_k(C)>$.

Над $c_i \in C$ возможны некоторые операции, результаты выполнения которых также являются элементами множества C. Тривиальными примерами таких операций могут быть объединение нескольких объектов в один или разделение некоторого компонента на отдельные части.

Обозначим через ΣCR — алгебру, содержащую множество компонентов C и операций R над компонентами C. Каждая операция имеет собственную арность в зависимости от ее семантики. В целом существуют операции, которые задаются на подмножестве целых, а также в качестве результатов могут использоваться подмножества (например, предыдущие операции объединения и разделения).

Среди множества операций рассматриваются те, которые сохраняют свойства компонентов. Обозначим эти операции через $R = (R_1, R_2, ..., R_m)$. Тогда для любых $c_1, c_2, ..., c_r$, принадлежащих C, и любой R_i, $i = 1, 2, ..., m$, с учетом арности операции R_i и ее результата справедливо одно из двух выражений:

$$R_i(c_1, c_2, ..., c_r) \in C \text{ или } R_i(c_1, c_2, ..., c_r) \notin C.$$

---

[1] См.: *Лаврищева. Е. М.* Компонентная теория и коллекция технологий разработки индустриальных приложений из готовых ресурсов // Труды Четвертой научной конференции «Актуальные проблемы системной и программной инженерии». С. 101—119; *Ее же.* Программная инженерия. парадигмы, технологии и CASE-средства.

Множество $C = (c_1, c_2, ..., c_r)$ и операции $R_1, R_2, ..., R_m$ определяют алгебру компонентов. Примерами реальных операций над компонентами могут быть операции расширения интерфейсов, рефакторинга и др.

Множество элементов C, рассмотренное выше, является математической абстракцией. В реальности мы имеем некоторое множество компонентов C, в котором $c_i \in C$.

Применив к этому множеству операции $R_1, R_2, ..., R_m$, можно получить множество C, которое будем называть замыканием множества C. Этот термин выбран для того, чтобы подчеркнуть, что C' имеет максимально возможное множество компонентов, которые можно получить на основе множества C. В соответствии с рассмотренными выше операциями получаем, что $c_i \in C$.

В реальных процессах конструирования систем рассматривается множество компонентов C (или C'), но при этом учитывается, что для него справедливы все положения соответствующей компонентной алгебры.

7.5.1. Компонентная модель системы

Под компонентной моделью системы понимается такое абстрактное представление конструируемой системы, в которой элементами являются реальные компоненты, обеспечивающие реализацию функций системы и выполнение других (нефункциональных) требований к системе. Для одной и той же системы существует множество компонентных моделей в зависимости от концепций проектирования с использованием компонентного подхода. Рассмотрим обобщенный уровень представления модели, используя функции компонентов и совокупность контрактов (интерфейсов) между ними.

Пусть $\prod A_i$ — множество интерфейсов, относящихся к определению функций компонентов. Каждому A_i можно сопоставить интерфейс In_i, который описывает интерфейс как клиент-серверное взаимодействие с соответствующими методами и структурами данных. В соответствии с этим каждому интерфейсу можно сопоставить пару In_i и Out_i, которые будем называть определяющим представлением In_i и реализующим представлением Out_i (что соответствует входному и выходному интерфейсу). In_i определяет условия и цель интерфейса со стороны клиента, а Out_i задает аспект реализации интерфейса со стороны сервера.

После того как все In_i и Out_i определены, их можно группировать в различных сочетаниях для элементов C. Рассмотрим произвольную совокупность In_i, Out_j, где $i \neq j$. В нее не входят одновременно определяющее и реализующее представления для одного и того же интерфейса.

Каждая полученная совокупность $C_i \cup I_v = \{C_i, I_v, Out_j, Inout_i^j\}$ входит в модель компонента или является шаблоном компонента. Фактически каждый шаблон содержит некоторое множество определяющих и реализующих представлений интерфейсов. По этим представлениям в дальнейшем осуществляется сопоставление шаблонов и интерфейсов реальных программных компонентов.

Модель компонентной системы имеет вид:

$$M_{кс} = (CL_m(Lm_1, ..., Lm_n), P(P_i, ..., P_m), CL_n(In_1, ..., In_k), D_i),$$

где CL_m — компоненты из множества реализаций в разных языках L; $P(P_i, ..., P_m)$ — множество предикатов, соответствующих процессам сборки или

конфигурации КПИ в ПС на основе реализаций компонентов CL_m и интерфейсов In; CL_n — множество интерфейсов компонентов; D_i — множество данных.

Модель компонентной системы $M_{кс}$ состоит из множества функций (объектов), реализаций (КПИ, *reuses*), предикатов, интерфейсов и данных.

Условие целостности системы заключается в существовании для каждого компонента C_1 из C, имеющего исходный интерфейс CIn_1^m, компонента C_2 с соответствующим входным интерфейсом CIn_2^m, и контракта $Cont_{12}^{im} = (CIn_1^u, CIn_2^m, IMap_{12}^{im})$, входящего в состав множества C.

Процесс построения $M_{кс}$ включает в себя создание компонентной среды, определение начальных компонентов и определенного множества интерфейсов в соответствии с функциональными требованиями к компонентной системе. Суть моделирования системы состоит в том, чтобы представить модель $M_{кс'}$, такую, чтобы для любого элемента системы существовал компонент из C или он мог быть получен из C' посредством конечного числа допустимых операций компонентной алгебры.

На практике такие условия могут выполняться не всегда (например, разрабатывается компонент с принципиально новой функциональностью). В этом случае реализуются дополнительные компоненты и расширяется множество C.

7.5.2. Формальные модели компонентного проектирования

Для установления связей между объектно-ориентированным и компонентным программированием предлагается объектное и компонентное представление программы, которая создаются по ОМ и интерфейсной модели.

Объектная модель системы имеет вид

$$OSyst = (OClass, G),$$

где $OClass = (OClass_i)$ — множество классов; G — объектный граф.

Интерфейсная модель в общем случае задается выражением $ISyst = (IFunc, IG)$, где $IFunc = (IFunc_i)$ — множество интерфейсов; IG — интерфейсный граф, который эквивалентен графу G модели ОМ. Для классов $OClass$ определяются условия, когда они допускают представление их как элементов множества интерфейсов в интерфейсном графе. Для всех классов объектной модели справедлива следующая теорема.

Теорема 7.2. Существует единственное изоморфное отображение объектной модели $OSyst$ и интерфейсной модели $ISyst$.

Эта теорема определяет условия существования эквивалентного отображения между объектным и компонентным представлениями системы. Как следствие, на этапах анализа и проектирования компонентной системы могут применяться модели и методы объектно ориентированного программирования, а на последующих этапах — собственные модели компонентного программирования, которые определяются дальше.

Дадим определение компонентных элементов ОКМ.

Программный компонент — это самостоятельно реализованный программный объект, который обеспечивает выполнение определенной совокупности прикладных функций, доступ к которым возможен только через интерфейс и операции вызова других компонентов.

Компонент отображает типовое решение или функцию и при последующей композиции с другими объектами и компонентами имеет типичную интерфейсную часть для обмена данными в разных средах. То есть компонент становится неделимым и инкапсулированным объектом, который удовлетворяет функциональным требованиям и требованиям компонентной среды.

Модель компонента — это обобщение типовых решений, касающихся сущности объектов, их архитектуры, структуры, свойств, характеристик, которые постепенно переходят или отображаются в компонент.

Формально модель компонента имеет вид

$$Comp = (CName, CInt, CFact, CImp, CServ),$$

где $CName$ — уникальное имя компонента; $CInt = \{CInt^i\}$ — множество интерфейсов, связанных с компонентом; $CFact$ — интерфейс управления экземплярами компонента; $CImp = \{CImp_j\}$ — множество реализаций компонента; $CServ = \{CServ_i\}$ — множество системных сервисов.

Множество $CInt = CIntI \cup CIntO$ состоит из входных $CIntI$ и выходных $CIntO$ интерфейсов. Различие между ними заключается в том, что для входных интерфейсов компонент имеет собственные реализации, а для выходных интерфейсов реализации находятся в других компонентах. Интерфейс $CFact$ определяет методы, которые необходимы для управления экземплярами компонента (поиск, создание, уничтожение и т.п.).

Возможные отношения между объектами: наследование, экземпляризация, интерфейс, связывание и взаимодействие.

Модель интерфейса имеет вид

$$CInt_i = (IntName_i, IntFunc_i, IntSpec_i),$$

где $IntName_i$ — имя интерфейса; $IntFunc_i$ — функциональность (совокупность методов); $IntSpec_i$ — спецификация интерфейса (описания типов, констант, других элементов данных, сигнатур методов и т.д.).

Необходимым требованием существования компонента является условие его целостности:

$$\forall CInt_i \in CIntI \; \exists \, CImp_j \in CImp \; [Provide(CInt_i) \subseteq CImp_j],$$

где $Provide(CInt_i)$ означает функциональность, которая обеспечивает реализацию методов интерфейса $CInt_i$. Для взаимодействия двух компонентов $Comp_1$ и $Comp_2$ существует следующее необходимое условие: если $CInt_{i1} \in \in CIntO_1$, то должен существовать $CInt_{k2} \in CInt_{I2}$, такой, что $Sign(CInt_{i1}) = = Sign(CInt_{k2})$ & $Provide(CInt_{i1}) \subseteq CImp_{j2}$, где $Sign(\cdot)$ означает сигнатуру соответствующего интерфейса.

Наличие знака включения в данной формуле означает, что избранная реализация компонента может обеспечить поддержку не только необхо-

димого интерфейса, но и других возможностей. Практические технологии и ЯП (CORBA, Com, Java, C# и др.) содержат необходимые средства. Интерфейс может иметь несколько реализаций, которые различаются особенностями функционирования (например, операционной средой, средствами хранения данных и т.д.).

Условия распределенной среды и проблемы возникновения и устранения угроз компонентам (и каркасам) ставят перед разработчиками ПС задачу управления атрибутами безопасности и др., а также защиты компонентов от разрушений и выходов из тупика.

Функциональные элементы компонентной модели могут быть равными, эквивалентными или подобными.

Определение 7.5. Два компонента $Comp_1$ и $Comp_2$ являются тождественными (равными), если тождественны их соответствующие составные части. Как следствие, замена $Comp_1$ на $Comp_2$ не влияет на компонентную систему, к которой принадлежит $Comp_1$.

Определение 7.6. Два компонента $Comp_1$ и $Comp_2$ являются эквивалентными, если эквивалентны их соответствующие составные части, кроме имен компонентов. Замена $Comp_1$ на $Comp_2$ не меняет функциональности компонентной системы при условии установления соответствия между именами в самой системе.

Определение 7.7. Два компонента $Comp_1$ и $Comp_2$ подобны, если эти компоненты имеют тождественное множество интерфейсов. Замена $Comp_1$ на $Comp_2$ сохраняет начальную функциональность компонентной системы, к которой можно добавить новые функциональные свойства.

Главной средой использования компонентов является **компонентная среда**, модель которой имеет вид

$$CE = (NameSpace, IntRep, ImpRep, CServ, CServImp),$$

где $NameSpace = \{CName_m\}$ — множество имен компонентов среды; $IntRep = \{IntRep_i\}$ — репозиторий интерфейсов компонентов среды; $ImpRep = \{ImpRep_j\}$ — репозиторий реализаций; $CServ = \{CServ_r\}$ — интерфейс множества системных сервисов; $CServImp = \{CServImp_r\}$ — множество реализаций для системных сервисов.

Модель компонентной среды рассматривается как множество серверов приложений, где разворачиваются компоненты-контейнеры, экземпляры которых реализуют функциональность компонента. Взаимосвязь контейнера с сервером осуществляется через стандартизированные интерфейсы (*CFact*). Связь между компонентами, которые развернуты в разных серверах, обеспечивается реализациями интерфеса *CServ*.

Определение 7.8. *Каркасом компонентной среды* называется среда, для которой совокупность имен компонентов, интерфейсов и реализаций суть пустые множества, т.е. $FW = (\varnothing, \varnothing, \varnothing, CServ, CServImp)$.

Пусть $FW_1 = (\varnothing, \varnothing, \varnothing, CServ_1, CServImp_1)$ и $FW_2 = (\varnothing, \varnothing, \varnothing, CServ_2, CServImp_2)$ — два каркаса.

Определение 7.9. Каркас FW_1 совместим с каркасом FW_2, если существует отображение $SMap$: $CServ_1$ —> $CServ_2$, такое, что $SMap(CServ_1) \subseteq \subseteq CServ_2$.

Совместимость каркасов и компонентных сред означает, что в среду $CSer_1$ должен входить сервис именования, а в $CServ_2$ — аналогичный сервис и между этими сервисами должна существовать такая связь, при которой сервис первой среды — это именование объектов среды и наоборот. Аналогичная ситуация должна существовать и для других системных сервисов, так как любая компонентная среда определенного типа использует те же сервисы, а их связи определяют совместимость. В частности, взаимодействие компонентов, расположенных в разных серверах, поддерживают сервисы этих серверов. Если совместимость между серверными сервисами существует, то такие компоненты могут входить в состав общей компонентной системы.

7.5.3. Операции над компонентами в компонентной среде

Далее рассматривается набор операторов, задаваемых математическими операциями $\{\cup, \cap, \oplus, \Diamond, -\}$ для задания и обработки компонентов в компонентной среде.

Операция добавления компонента к компонентой среде обозначается $\oplus$. Добавление компонента к среде выполняется согласно следующим правилам:

$C \oplus CE_1 = CE_2$;

$CE_2.NameSpace = \{C.CName\} \cup CE_1.NameSpace$;

$CE_2.InRep = \{C.(CIn_i, CName)\} \cup CE_1.InRep$;

$CE_2.ImRep = \{C.(C_j, CName)\} \cup CE_1.ImRep$.

Аксиома 7.6. Операция добавления компонента к компонентной среде коммутативна: $C \oplus CE = CE \oplus C$.

Аксиома 7.7. Операция добавления компонента к компонентной среде ассоциативна: $C_1 \oplus (CE \oplus C_2) = (C_1 \oplus CE) \oplus C_2$.

Утверждение 7.1. Каждая непустая компонентная среда CE может быть представлена в виде: $CE = C_1 \oplus C_2 \oplus ... \oplus C_n \oplus FW$.

Операция удаления компонента из компонентной среды обозначается знаком $\Diamond$ и подчиняется следующим правилам:

$CE_1 \Diamond C = CE_2$;

$\exists\ CName_m : CName_m \in CE_1.NameSpace \cap (CName_m = C.CName) \Rightarrow CE_2.NameSpace = CE_1.NameSpace \setminus \{C.CName\} \cap CE_2.IntRep = CE_1.IntRep \setminus \{(\forall i: IntRep^i.CName = C.CName)\ IntRep^i\} \cap CE_2.ImpRep = CE_1.ImRep \setminus \{(\forall j\ \&ImtRep^j.CName = C.CName)\ ImRep^j\}$.

Теорема 7.3. Для любого компонента C и компонентной среды CE выполняется равенство $(C_2 \oplus CE) \Diamond C = CE$.

Доказательство. Представим операцию добавления компонента к среде:

$C \oplus CE = (\{C.CName\} \cup CE.NameSpace\}, \{C.(CIn_i, CName)\} \cup CE.InRep, \{C.(CIm_j, CName)\} \cup CE.ImRep, CSe, CSeIm) = CE$.

Тогда операция удаления компонента из среды имеет вид:

$CE' \setminus C = (CE'.NameSpace \setminus \{C.CName\},$

$\{CE'.InRep \setminus C.(CIn_i, CName)\},$

$CE'.ImpRep \setminus \{C.(CIm_j, CName)\}, CSe, CSeIm) = CE$.

Операция замены компонента задается знаком «−» и имеет вид

$$CE.NameSpace(C_1) - C_2 = (CE \Diamond C_1) \oplus C_2.$$

Операция **объединения** (обозначается $\cup$) компонентных сред выполняется согласно следующим правилам:

$CE_1 \cup CE_2 = CE_3$;

$CE_3.NameSpace = CE_1.NameSpace \cup CE_2.NameSpace$;

$CE_3.InRep = CE_1.InRep \cup CE_2.InRep$;

$CE_3.ImRep = CE_1, ImRep \cup CE_2.ImRep$.

Теорема 7.4. Операция объединения компонентных сред ассоциативна:

$$(CE_1 \cup CE_2) \cup CE_3 = CE_1 \cup (CE_2 \cup CE_3).$$

Доказательство:

$\forall C \in (CE_1 \cup CE_2) \cup CE_3 \Rightarrow C \in CE_1 \cap C \in CE_2 \cap C \in CE_3$;

$\forall C \in CE_1 \cup (CE_2 \cup CE_3) \Rightarrow C \in CE_1 \cap C \in CE_2 \cap C \in CE_3$.

Аксиома 7.8. Операция объединения компонентных сред коммутативна:

$$CE_1 \cup CE_2 = CE_2 \cup CE_1.$$

Утверждение 7.4. Для любых компонентных сред выполняется:

$$CE \cup FW = FW \cup CE = CE.$$

Утверждение 7.5. Для любой компонентной среды $CE \cup FW = FW \cup CE = CE$.

Утверждение 7.6. Для двух произвольных компонентных сред CE_1, CE_2 и компонента C всегда выполняется:

$$C \oplus (CE_1 \cup CE_2) = (C \oplus CE_1) \cup CE_2 = (C \oplus CE_2) \cup CE_1.$$

Утверждение 7.7. Для любого компонента C и компонентной среды CE всегда выполняется соотношение: $C \in CE$.

Процесс построения КС включает в себя создание компонентной среды, определение начальных компонентов и формирование множества контрактов в соответствии с функциональными требованиями к компонентной системе.

7.5.4. Компонентная алгебра

Для трансформации моделей компонентного проектирования к выходному коду разработана компонентная алгебра — внешняя, внутренняя и эволюционная:

$$\Sigma = \{\varphi_1, \varphi_2, \varphi_3\},$$

где $\varphi_1 = \{CSet, CESet, \Omega_1\}$ — внешняя алгебра; $\varphi_2 = \{CSet, CESet, \Omega_2\}$ — внутренняя алгебра; $\varphi_3 = \{Set, CESet, \Omega_3\}$ — алгебра эволюции компонентов.

Внешняя компонентная алгебра

Эта алгебра двухосновная и имеет следующий вид:

$$\varphi_1 = \{CSet, CESet, \Omega_1\},$$

где $CSet$ — множество компонентов C; $CESet$ — множество компонентных сред; Ω_1— множество операций над этими элементами. Операции над компонентами такие:

- инсталляция (развертывание компонента) $CSet_2 = Cset \oplus CESet_1$;
- объединение компонентных сред $CESet_3 = CESet_1 \cup CESet_2$;
- удаление компонента из компонентной среды $CESet_2 = CESet_1 / CSet$.

Внутренняя компонентная алгебра

Эта алгебра имеет следующий вид:

$$\varphi_2 = \{CSet, CESet, \Omega_2\},$$

где $Cset = \{OldComp, NewComp\}$ — множество старых *OldComp* компонентов в системе и множество новых компонентов *NewComp*, включая вновь разработанные или преобразованные старые компоненты к новому *NewComp*; *OldComp* = (*OldCName*,*OldInt*, *CFact*, *OldImp*, *CServ*) включает интерфейсы, реализации в серверной среде; *NewComp* = (*NewCName*, *NewInt*, *CFact*, *NewImp*, *CServ*) включает интерфейсы, реализации этого компонента, как необходимые элементы любого компонента, в том числе и нового компонента в серверной среде.

$$\Omega_2 = \{addImp, addInt, replImp, replInt\},$$

где *addImp* — операция добавления реализации; *addInt* — операция добавления интерфейса; *replImp* — операция замещения реализации компонента; *replInt* — операция замещения интерфейса компонента.

$$\Omega_3 = \{O_{refac}, O_{reeng}, O_{rever}\} \text{ — алгебра эволюции } (\varphi_3),$$

где O_{refac} — рефакторинг для построение компонентной ПС и возможностью внесения изменений; O_{reeng} — реинжиниринг для обеспечения замены, переименование компонентов и (или) их интерфейсов, а также добавление новых компонентов в ПС и в компонентную среду; O_{rever} — реверсная инженерия обеспечивает восстановление структуры компонента по его выходному коду (обратная инженерия).

Операции **алгебры эволюции** обеспечивают изменение компонентов и интерфейсов с помощью специальных операций, определенных ниже.

Модель рефакторинга компонентов:

$$M_{refac} = \{O_{refac}, \{CSet = \{NewComp^n\}\},$$

где $O_{refac} = \{AddOImp, AddNImp, ReplImp, AddInt\}$ — операции рефакторинга; пара ($CSet$, O_{refac}) — элемент компонентной алгебры эволюции.

Операция рефакторинга носит комплексный характер, так как методы, входящие в этот интерфейс, требуют реализации компонента в составе контейнера. Поэтому реализация этой операции связана с существованием нового типа контейнера. Согласно классификации методов рефакторинга операция расширения интерфейса управляет экземплярами компонентов и относится к расширенной классификации. Множество операций рефакторинга *AddOImp*, *AddNImp*, *ReplImp*, *AddInt* включают в себя операции добавления и замещения реализаций компонентов и интерфейсов.

Модель реинженерии компонентов:

$$M_{reeng} = \{O_{reeng}\,\{CSet = \{NewComp^n\}\},$$

где O_{reeng}= {*rewrite*, *restruc*, *adop*, *supp*, *conver*} — операции реинженерии; пара ($CSet$, O_{reeng}) — элемент компонентной алгебры эволюции.

Алгебра реинжиниринга $\sum^{Reeng} = (CSet, Reeng)$ используется при условии нарушения целостности компонентов или изменения функциональности. Эта алгебра может быть построена лишь при условии нарушения целостности представления компонентов. Кроме операций рефакторинга (*Refac*), во множество *Reeng* входят операции, которые удаляют из компонента существующий интерфейс или изменяют его сигнатуру. Это усиливает условие целостности, так как другие компоненты, которые обращаются к нему, не могут иметь доступа к необходимой функциональности. Исходя из таких условий, вместо алгебры реинжиниринга в состав формальных методов компонентного программирования необходимо включить модель реинжиниринга.

Модель реверсной инженерии компонентов:

$$M_{rever} = \{O_{rever}\{CSet = \{NewComp^n\}\},$$

где $O_{rever} = \{restruc, design\}$ — операции реверсной инженерии; пара ($CSet$, O_{reeng}) — элемент компонентной алгебры эволюции.

Алгебра реверсной инженерии $\sum^{Rever} = (CSet, Rever)$ используется при условии восстановления структуры и спецификации ПС. В ней могут использоваться операции реинженерии $Reeng \subset Rever$. Особенность *Rever* состоит в том, что ее операции определены не полностью, а лишь частично. Реверсная инженерия обозначает полную перестройку ПС из компонентов в некоторой среде, которые могут изменяться в зависимости от смысла функции и применяемых операций изменения компонента. Операция *Reeng* может определять трансформацию не только целевого компонента, но и других связанных компонентов. Таким образом, операции эволюционной алгебры обеспечивают изменения конкретного компонента и относятся к классу операций в других методах программирования, которым соответствует операции алгебры *эволюции*. Базовая суть этих операций — изменение имен, интерфейсов, реализаций и связей между элементами, а также операции преобразования структуры и функций компонентов.

7.6. Связь компонентной и объектной моделей

Результатом сопоставления определений алгебраических систем и изоморфизма этих систем является следующая теорема.

Теорема 7.5. Отображение $O \rightarrow C$ создает сложный объект из КПИ и его реализацию, изоморфизм объектной и компонентной моделей задается моделью типа <2, 2, 2, 2, 2>.

Отображение сопоставляет каждому объекту входной интерфейс. Изоморфизмы алгебраических моделей типа 2 (O; RO), (I; RI) и (I; RI), (C, RC) превращают точки вариабельности / варианты сложных объектов в точки вариабельности / варианты интерфейсов и КПИ.

Связь компонентной модели с объектной ОМ проследим с помощью процесса функционирования компонентов *Create* и интерфейса *Cfac*, порождающего экземпляры компонентов:

$$Cfact.Create: Comp \rightarrow \{Cins_k^{ij}\}, Cins_k^{ij} = (Iins_k^{ij}, IntFunc^i, ImpFunc^j),$$

где $Cins_k^{ij}$ — экземпляр k-компонента, который предоставляет свою функциональность с помощью интерфейса $IntFunc^i$ и обеспечивает реализацию этого интерфейса с помощью $ImpFunc^j$; $Iins_k^{ij}$ — уникальный идентификатор экземпляра компонента.

Пусть существует некоторая объектная система, представленная диаграммой классов:

$$Osyst = (OClass, G),$$

где $OClass = \{OClass^i\}$ — множество классов; G — объектный граф, отражающий связи и отношения между классами и экземплярами.

Каждый класс представлен в виде

$$OClass^i = (ClassName^i, Method^i, Field^i\},$$

где $ClassName^i$ — имя класса; $Method^i = \{Method_j^i\}$ — множество методов; $Field^i = \{Field_n^i\}$ — множество переменных, определяющих состояние экземпляров класса.

Переход от объектной к компонентной структуре связан с преобразованием объектов и интерфейсов ОМ к компонентам и интерфейсам (рис. 7.5).

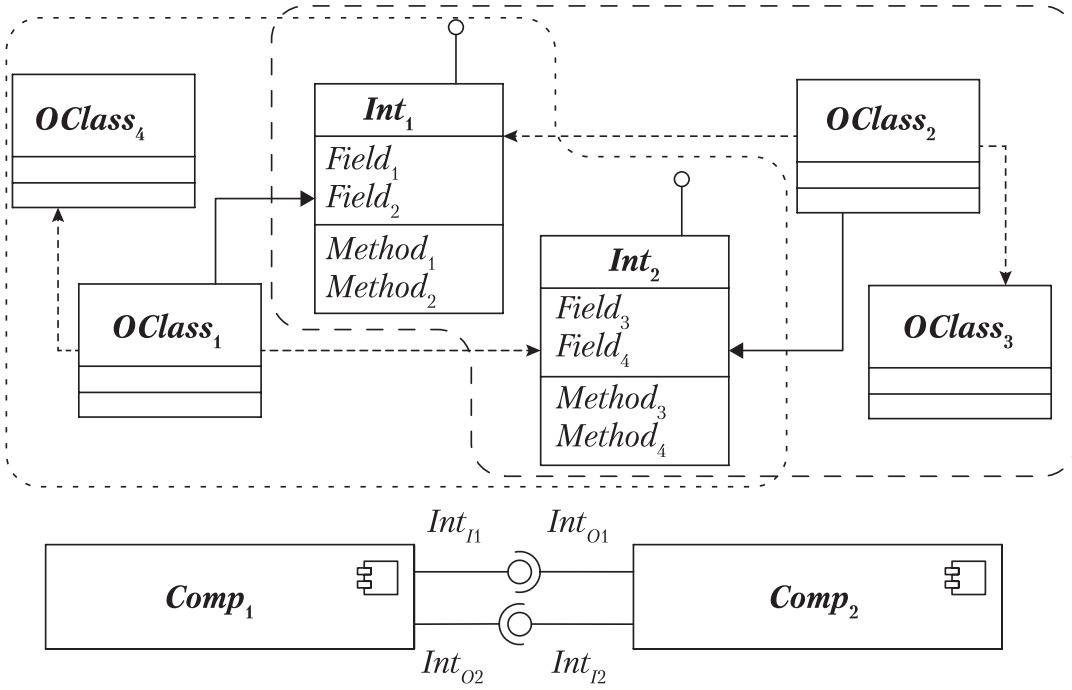


Рис. 7.5. **Структура связи компонентов и интерфейсов в классе**

На рисунке показаны структура компонентов $Comp_1$ и $Comp_2$, соответствующие интерфейсы. Int_1, Int_{O1} и Int_{I2}, Int_{IO2} в классе компонентов $OClass_1$, $OClass_2$ и $OClass_3$.

Каждый *Oclass* представляется как модель класса

$$OClass^i = \{ClassName^i, Method^i, Field^i\},$$

где $OClass = \{OClass^i\}$ — множество классов; $ClassName^i$ — имя класса; $Method^i = \{Method_j^i\}$ — множество методов; $Field^i = \{Field_n^i\}$ — множество переменных, определяющих состояние экземпляров класса.

Пусть $Pfield^i \subset Field^i$ — множество внешних переменных (*public*), доступные извне. Каждому $Pfield^i_n \in Pfield^i$ ставятся в соответствие методы *get* $<Pfield^i_n>$ и *set* $<Pfield^i_n>$ для присвоения и выборки значений переменной. Другими словами, эти переменные становятся атрибутами в современных компонентных моделях.

В других классах вместо непосредственного обращения к таким переменным будут использоваться указанные методы.

Введем новое множество методов:

$$Imethod^i = Method^i \cup \{get <Pfield^i_n>\} \cup \{set <Pfield^i_n>\},$$

которым сопоставим интерфейс $Ifunc^i$, состоящий из прототипов методов, которые входят в $Imethod^i$.

Вместе с *Osyst* рассмотрим систему $Isyst = (Ifunc, IG)$, где $Ifunc = \{Ifunc^i\}$ — множество интерфейсов; *IG* — интерфейсный граф, идентичный графу *G*.

Класс $Oclass^i$ порождает свои экземпляры (объекты) $Obj_k{}^i = \{ObjName^i_k, Method^i, Field^i\}$, которым в системе *Isyst* будут соответствовать интерфейсные элементы $Iobj^i_k = \{Iname^i_k, Ifunc^i\}$.

Для каждого такого элемента не определена реализация соответствующего интерфейса. Сопоставив некоторому интерфейсу реализацию $ImpFunc^j$ (которая обеспечивает выполнения методов интерфейса), сформируем элемент $Iobj^{ij}_k = \{Iname^i_k, Ifunc^i, ImpFunc^j\}$, который, по своей сути, эквивалентен экземпляру компонента $Cins^{ij}_k = (Iins^{ij}_k, IntFunc^i, ImpFunc^j)$.

Основные расхождения определяются следующими факторами. Во-первых, выбор реализации в процессе развертывания, а не на ранних стадиях, как это есть в ООП. Во-вторых, экземпляр объектного класса порождается его описанием и не может содержать элементов больше, чем существует в самом классе или его суперклассах. Противоположно этому реализация компонента может поддерживать несколько не связанных между собою интерфейсов. Эти две системы (объектная и интерфейсная) эквивалентны.

При построении компонентной системы используются соответствующие инструментальные средства. Согласно логико-математическому проектированию создается ОМ и соответствующая интерфейсная система без конкретизации реализации этих интерфейсов. Результат такого проектирования — совокупность интерфейсов, для которой рассматривается задача покрытия интерфейсов соответствующими компонентными реализациями. При этом нет необходимости учитывать реализацию функциональности ПС в виде конфигурации компонентов, которая выполняется путем сборки компонентов в стандартной форме и развертывания ПС.

Типы отношений между компонентами в ОКМ задаются в виде

$$Comp_n = (Cname_n, Ci_n, Cfact_n, Cimp_n, Cserv).$$

К ним относятся следующие отношения:

- *наследование* двух компонентов определяется установлением отношения наследования для входных интерфейсов (как наследования в ООП);

• *экземпляризация* создается соответственно определенному входному интерфейсу $CIntI^i$. Выражение $CIns_k^{ij} = (IIns_k^{ij}, IntFunc^i, ImpFunc^j)$ описывает экземпляр компонента *Comp*. Здесь $IIns_k^{ij}$ — уникальный идентификатор экземпляра, $IntFunc^i$ — функциональность интерфейса $CIntI^i \in CInt$, $ImpFunc^j$ — программный элемент, который обеспечивает реализацию $CImp^i \in \in CImp$;

• *интерфейс* между компонентами $Comp_1$ и $Comp_2$ в виде выражения $Cont_{12}^{im} = (CInt_1^u, CInt_2^m, IMap_{12}^{im})$, где $CInt_1^u \in CInt_1$ — исходный интерфейс первого компонента, $CInt_2^m \in CInt_2$ — входной интерфейс второго компонента; $IMap_{12}^{im}$ — отображение соответствия между методами, входящими в состав обоих интерфейсов с учетом сигнатур и ТД, которые передаются. Отношения контракта существует, если компонент $Comp_2$ имеет реализацию интерфейса $CInt_2^m$, выполняющего функциональность $IntFunc_1$ и интерфейс $CInt_1^n$;

• *связывание* компонентов $Comp_1$ и $Comp_2$ с помощью отношения контракта $Cont_{12}^{im}$, в котором между экземплярами $CIns_{1k}^{ij} = (IIns_{1k}^{ij}, IntFunc^i, ImpFunc^j)$ и $CIns_{2p}^{mq} = (IIns_{2p}^{mq}, IntFunc^m, ImpFunc^q)$ существует отношение связывания через контракт $Cont_{12}^{im}$, который описывается выражением *Bind* $(IIns_{1k}^{ij}, IIns_{2p}^{mq}, Cont_{12}^{im})$.

Между объектным и компонентным представлениями систем существует неоднозначность, которая порождается тем, что определенный компонент может иметь несколько интерфейсов *Isyst*. В случае если каждый отдельный компонент использует один интерфейс, то существует единое эквивалентное отображение между объектными и компонентными представлениями.

Пример объектного и компонентного описания сложной системы из объектов и компонентов показан на рис. 7.6.

В нем используются:

$Comp_1(OC_{11}, OC_{12})$, $Comp_2$ (IC_2, OC_2), $Comp_3$ (IC_3, IC_4, OC_3) — компоненты ПрО с входными и выходными параметрами;

O_1, O_2, O_3, O_4 — классы объектов ОМ;

IO_2, IO_3, IO_4 — входные объектные интерфейсы;

IC_2, IC_3, IC_4 — входные параметры компонентов (штрихпунктирные линии).

В интерфейсном представлении *ISyst* существуют компонентные интерфейсы IC_2, IC_3, IC_4 (штрихпунктирные линии).

При этом $OC_{11}, OC_{12}, OC_2, OC_3$ — исходные интерфейсы в компонентной модели. Между объектными и компонентными интерфейсами установлено однозначное отображение, а для объектной и компонентной моделей такого отображения не существует, так как компонент $Comp_3$ имеет два входных интерфейса, т.е. функциональность классов O_3 и O_4 реализована в одном компоненте.

В ОКМ объектный анализ — это первая фаза объектно-компонентного метода проектирования ПС, при котором выявляются объекты и строится ОМ, адекватно отображающая ее структуру, объекты и отношения между ними и т.п. Главная задача второй фазы — проектирование конкретных компонентов и ПС по результатам моделирования ПрО. В ОКМ процессы

определения объектов начинаются с отдельных сущностей ПрО и заканчиваются заданием компонентов с учетом их поведения в компонентной среде. Объектная модель отображается в компонентную модель путем интерфейсов для компонентов на основе компонентной алгебры.

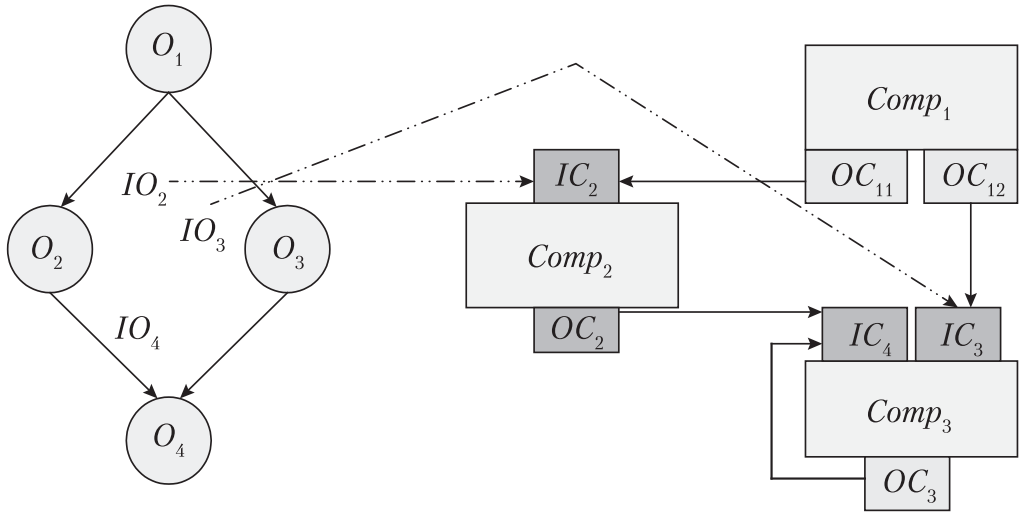


Рис. 7.6. **Структура программы в ОКМ**:
⟶ — связи между объектами; ⇢ — информационные связи

7.7. Технология конфигурационной сборки

Основные задачи конфигурационной сборки ГоР в ПС теоретически и практически выполнены Л. В. Колесником и О. А. Слабоспитской[1]. Они состоят в следующем:

- поддержка операций добавления требований, их трассировка в вариантах, используемых ГоР на всех уровнях;
- порождение новых вариантов архитектуры и компонентов в точках вариантности для реализации установленных требований и ограничений к ПС;
- поддержка эффективных решений по разработке новых вариантов ГоР при проведении анализа «затраты/преимущества» с учетом данных в базе знаний проекта ПС;
- управление конфигурацией ПС путем выбора вариантов ГоР для ПС;
- планирование вариабельности и расширением готовых компонентов новыми вариантами и контролем полноты и непротиворечивости.

Основные задачи управления конфигурацией КПИ состоят в следующем:

- сборка ресурсов по моделям ПС и *Mvar* (контроль, верификация ресурсов, тестов);
- систематическое отслеживание внесенных изменений в отдельные части конфигурации, аудит и контроль изменений;
- поддержка целостности конфигурации путем аудита;

---

[1] См.: *Lavrischeva E.*, *Stenyashin A.*, *Kolesnyk A.* Object-Component Development of Application and Systems. Theory and Practice // Journal of Software Engineering and Applications, 2014. URL: http://www.scirp.org/journal/jsea.

• ревизия конфигурации с учетом требований к ПС;

• трассировка изменений конфигурации при сопровождении и эксплуатации ПС;

• согласование объектно-компонентных и вариабельных моделей МХ ПС;

• доказательство изоморфного отображения объектных и компонентных моделей ПС путем трансформации объектов и данных в структуре ПС.

При сборке ГоР в ПС используется модель вариабельности ВХ, содержащая все характеристики функций системы. Она является инструментом изменения и настройки конфигурируемых артефактов ПС по требованию заказчика. Вариабельные элементы используются на всех уровнях представления и взаимосвязи ресурсов в ПС.

ГоР в стандартном виде накапливаются в репозиториях среды разработки ПС. Они отбираются, адаптируются и интегрируются в систему ПС.

Основную роль в этих процессах выполняет конфигуратор ИТК[1]. Он обеспечивает связь КПИ и их интерфейсов с вариантами отдельных рабочих продуктов ПС, которые находятся в репозитории, и устанавливают логику вариантности.

Применение конфигуратора КПИ необходимо в инженерии СПС, поскольку варианты каждого КПИ имеют контексты использования в разных ПС и находятся под контролем конфигуратора ИТК. На некотором уровне система как член СПС также является вариантом.

Для управления конфигурацией СПС из КПИ репозиторий ИТК дополнен механизмами хранения, введения (изъятия) вариантов КПИ и сопоставления их между собою, а также принятия решений по сборке, объединению разных вариантов КПИ и с учетом требований к показателям качества ПС, стоимости и др.

Управление конфигурацией СПС начинается на линии подбора, поиска и выбора КПИ в репозитории, сборки их в виде конфигурационного файла ПС или СПС в среде ИТК.

Конфигурационная сборка реализована (рис. 7.7) с помощью рассмотренной модели вариабельности и включает[2]:

• *модель инженерии* для разработки отдельных КПИ и сборки (*reuses*, *assets*, *services*) в продукт СПС;

• *процессную модель* — для обеспечения повторного использования (*forreuse*) и разработки с использованием КПИ (*with reuse*);

• *модель управления вариантами ПС* путем конфигурации ПС из КПИ.

Разработка вариантов ПС и СПС (*AS*) базируется на следующих аксиомах и предположениях.

---

[1] ИТК — URL: http://www.7dragons.ru/ru.

[2] См.: *Колесник А. Л.* Подходы к конфигурированию компонентов повторного использования // Проблемы программирования. 2011. № 4. С. 57—66; *Slabospitskaya O., Kolesnyk A.* The Model for Enhanced Variability Management Process in Software Product Line // Preproc. of 4th International United Information Systems Conference (UNISCON 2012). Crimean State Humanitarian University, Yalta, Ukraine, June 1—3, 2012. URL: http://www.uniscon.org/images/uniscon/downloads/ UNISCONPapers.zip; *Колесник А. С.* Модели и методы разработки семейства вариантных программных систем : автореф. дис. ... канд. техн. наук. Киев, 2013. — 22 с.

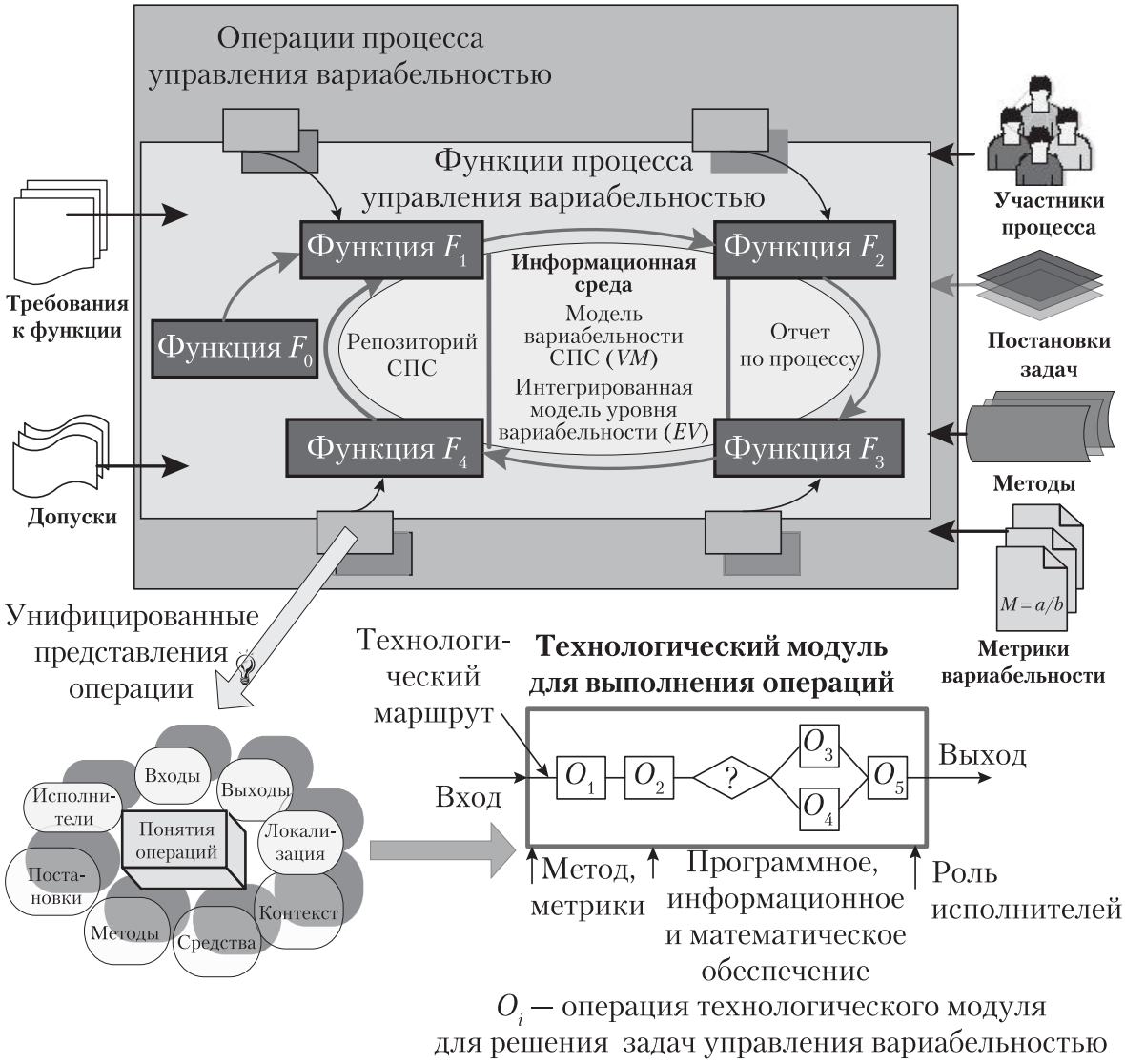


Рис. 7.7. **Схема процесса управления вариабельностью СПС**

Аксиома 7.9. Технология задает циклическую последовательность процессов разработки ПП и актуализации СПС.

Аксиома 7.10. Каждую терминальную характеристику модели системы реализует один и только один базовый КПИ.

К операциям управления вариабельностью отнесены функции F_1—F_4 модели МХ с использованием ГоР КПИ и предикаты проверки соответствия этих функций заданным требованиям к СПС.

Управление вариабельностью СПС основывается:

- на точках вариантности или вариантных артефактах СПС;
- предикатах на множестве точек вариантности и вариантов и взаимосвязях ними.

Для управления вариабельностью используется цикл управления Е. Деминга, основанный на следующих функциях:

1) F_1 — планирование вариабельности в артефактах и архитектуре СПС;
2) F_2 — реализация вариабельности в артефактах СПС;
3) F_3 — системный мониторинг вариабельности архитектуры СПС;
4) F_4 — актуализация СПС для выполнения ПС и СПС.

Функции F_1—F_4 выполняются с учетом заданных требований (*Requirement* — *R*) R_1—R_4 к артефактам и архитектуре СПС:

1) R_1 — для обоснования выполнения функций F_1—F_4;

2) R_2 — для реализации этих функции на процессах разработки СПС;

3) R_3 — для принятия решений о правильности выполнения функциональных свойств СПС;

4) R_4 — для трассировки связей между элементами модели вариабельности на множестве точек вариантности и вариантов СПС.

Проверка требований R_1—R_4 относительно функций F_1—F_4 на соответствие потребностям заказчика проводится по интегрированной модели вариабельности СПС (*VM*), которая позволит определять оценку уровня вариабельности и степени ее соответствия потребностям СПС. В ней ортогональная модель вариабельности (*OVM*) определяет проявление вариабельности по F_1—F_4 в структуре СПС и в артефактах процессов инженерии доменов и инженерии приложений.

Управление конфигурацией выполняется в *модельной среде* конфигуратора, в которую входят:

- схемы формального описания артефактов;
- схема решения квадратного уравнения модели ВХ с отмеченными точками вариантов;
- БД, включающая требования, архитектуру, набор базовых ГоР и ПС;
- конфигуратор, объединяющий артефакты в ПС и СПС.

В качестве примера взят метод решения квадратного уравнения. Ему задаются запросы, которые выполняются по схеме, приведенной на рисунке. Данные, КПИ и результаты сборки ПС и СПС размещаются в репозитории. В процессе управления конфигурацией собираются данные для проведения ряда процедур, включенных в первоначальный план конфигурации:

- управление сборкой КПИ и инструментов построения ПС;
- управление процессами с соблюдением плана развития ПС и СПС;
- управление средой (программное и аппаратное обеспечение), на которых размещена СПС;
- взаимодействие членов команды, которые работают над проектом СПС;
- отслеживание дефектов и обработка этих дефектов.

Среди операций выполняются такие, которые являются стандартными в процессе конфигурации.

Отчетность конфигурации — учет и отчетность всей необходимой информации о состоянии процесса разработки ПС.

Аудит конфигурации — гарантия того, что ПС содержит запланированные функции в соответствии с документом спецификации, в том числе требования, архитектуру и документацию для пользователя.

На конфигураторе реализуются сборка ГоР и преобразование исходного кода в код для выполнения на компьютере. Одним из шагов **сборки** является процесс компиляции исходного кода, где файлы превращаются в промежуточный код или в код выполнения в среде VS.Net. Для сложных

программ после компиляции устанавливается связь (нахождение реального положения всех внешних функций), которая выполняется специальной программой — связником. Процесс связи представляет собой замену относительных адресов функций внешних библиотек на реальные адреса, используемые в программе при выполнении. В этом случае конфигуратор является системой автоматизации и предоставления интерфейса пользователя для сборки ПС из набора КПИ в среде СПС.

Модель характеристик — это дерево свойств системы, представленных диаграммой характеристик, которое используется во время всего процесса разработки ПС в СПС.

Диаграмма характеристик — это визуальное представление свойств модели характеристик с точками вариантности в виде дерева (рис. 7.8).

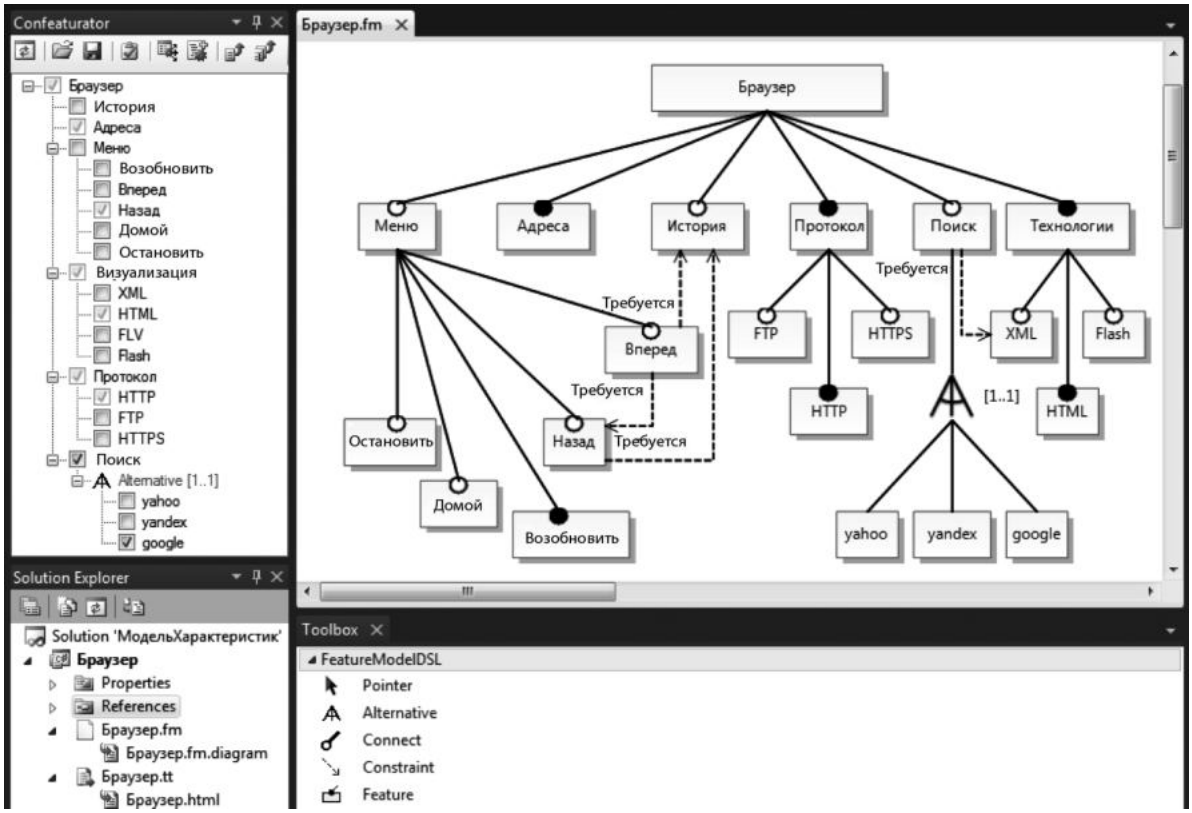


Рис. 7.8. **Диаграмма характеристик браузера конфигуратора для управления разработкой СПС**:

— точка вариантности; Адреса — вариант КПИ обязательного типа;

Меню — вариант КПИ необязательного типа;

----- — зависимость вариантов от точки вариантности;

[1..1] — альтернатива вариантности нескольким точкам для внесения в ПС;

---➤ — ограничения на использование вариантов

На рис. 7.8 представлена структура браузера для реализации диаграмм характеристик СПС. Диаграмма характеристик в браузере описывается в языке DSL и реализуются средствами Windows Workflow Foundation. Словарь терминов данного языка включает обычные термины, инструкции технологии WF (Workflow) и артефакты.

7.8. Реализация технологии сборки в ИТК

Объекты и компоненты предложенных моделей OM, KM и FM трансформируются к программным реализациям в языке WSDL и сохраняются в репозитории ИТК. Эти модели используются при конфигурировании артефактов и компонентов в выходную ПС. Компоненты системы определяют алгоритмы функций и данных, которые могут быть FDT, GDT либо нестандартные ТД (*Big Data*). Интерфейсы данных задаются в IDL. Готовые ГоР или КПИ имеют сертификат (паспорт, описание функций, интерфейс и данных для обмена). **Паспорт** имеет стандартный вид (рис. 7.9).

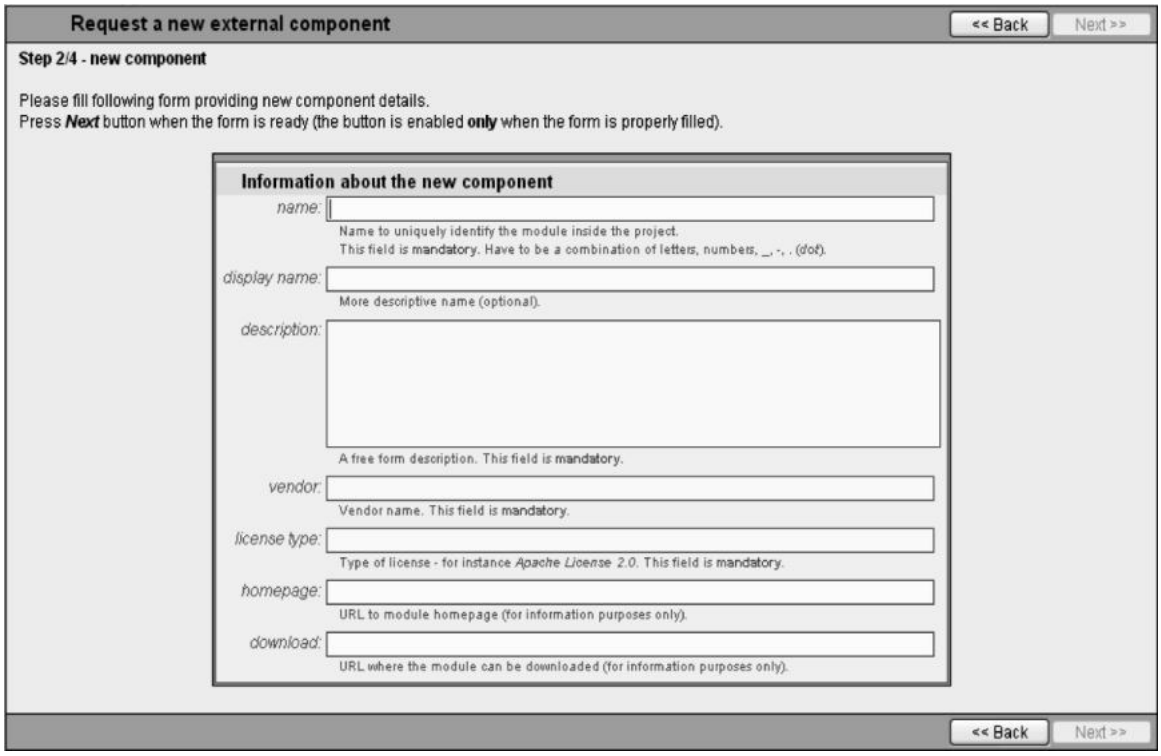


Рис. 7.9. **Формат стандарта паспортных спецификаций в Grid**

В нем дается описание в языке WSDL:

- *название* функции;
- *ID*-идентификатор ресурса;
- *содержание* (функции);
- *параметры* вызова других программ;
- *инструменты* поддержки выполнения программы;
- *необязательные атрибуты* (дата, состояние, версия, право доступа, автор, дата создания, срок пригодности, правила приобретения и т.п.).

Все артефакты системы сохраняются в репозитории и извлекаются оттуда, когда их надо использовать. По имени ресурса осуществляются поиск и анализ применимости ресурса, а потом их конфигурационная сборка с учетом модели MX.

Для поддержки ОКМ разработана интегрированная среда веб-сайта ИТК, включающая готовые системные компоненты — Eclipce, Protégé, VS.Net, CORBA, Java и др.

Компоненты ИТК, которые реализуют технологии разработки ПС из готовых КПИ, следующие[1]:

- ведение репозитория с помощью сертифицированных в языке WSDL и занесение их в репозиторий (компонентов и интерфейсов);
- поиск (выбор) КПИ пользователем и анализ их семантики для возможного применения в новой ПС;
- сборка отдельных КПИ путем конфигурации КПИ в вариант некоторой ПС;
- использование языка DSL для формального описания отдельных артефактов и их реализация в инструментах DSL Tools VS.Net/Eclipse-DSL в код XML с оформлением их паспорта в WSDL и занесением в репозиторий;
- онтология доменов — ЖЦ стандартов ISO/IEC 12207 и ISO/IEC 11404 GDT в DSL и вычислительной геометрии в Protégé;
- оценка стоимости (метод COCOMO II) и качества (ISO/IEC 9126) продукта с использованием результатов тестирования и испытания системы;
- обработка интерфейсных данных, передаваемых между компонентами, и трансформация их данных к соответствующим форматам с помощью интерфейсных функций библиотеки VS.Net;
- тестирование разнородных ресурсов в ЯП и ПС из них;
- организация взаимодействия программ и сред с помощью реализованных студентами трех моделей CORBA ↔ Eclipse-Java, VS.Net C# ↔ Eclipse, Модель Basic ↔ C++;
- электронное обучение программированию языкам C# VS.Net и Java (по учебнику: *Хабибулин И.* Java 7. Наиболее полное руководство. СПб. : БХВ-Петербург, 2012), а также собственного курса «Программная инженерия» по электронному учебнику (укр.) на сайте (http://programsfactory.univ.kiev.ua).

Описанная ОКМ-технология разработки ПС из готовых КПИ представлена на модифицированном сайте ИТК (URL: http://www.7dragons.ru/ru). К этим сайтам обращаются (по оценкам Google-статистики, более 100 000 респондентов за 2011—2015 гг.) преподаватели и студенты ряда университетов (России, Украины, Европы, США, Канады и др.). Основное его назначение — подготовка бакалавров и магистров по специальности «Программная инженерия».

Контрольные вопросы и задания

1. Определите теорию моделирования систем из объектов.
2. Определите обобщающенный уровень проектирования ОМ.
3. В чем состоят задачи структурного уровня моделирования ОМ?
4. Определите характеристический уровень проектирования ОМ.
5. Каковы задачи поведенческого уровня проектирования ОМ?

---

[1] См.: *Лаврищева Е. М.* Компонентная теория и коллекция технологий разработки индустриальных приложений из готовых ресурсов // Труды Четвертой научной конференции «Актуальные проблемы системной и программной инженерии». С. 101—119.

6. Определите операции над классами объектов.
7. Что составляет основы объектного анализа?
8. Определите набор моделей компонентного проектирования.
9. Назовите операции над компонентами в компонентной алгебре.
10. Назовите этапы конфигурационной сборки компонентов.
11. Какие модели относятся к базовым моделям проектирования систем?

Глава 8
ПРИМЕНЕНИЕ ТЕОРЕТИЧЕСКОГО АППАРАТА ФУНДАМЕНТАЛЬНЫХ И ОБЩИХ ТИПОВ К НЕСТРУКТУРНЫМ ДАННЫМ BIG DATA

На данное время в компьютерном информационном пространстве накоплено большое количество разнородных программ, которые используются постоянно или частично в разных средах для вычисления многих задач. При появлении новых компьютеров, мейнфреймов, кластеров, систем распределенных вычислений (типа Grid, Big Data, Cloud) проблема описания и накопления готовых программ по-прежнему актуальна. При вычислении программы обмениваются данными, которые могут быть представлены как фундаментальные данные, содержащиеся во всех ЯП, общие ТД стандарта ISO/IEC 11404 и неструктурные данные, поступающие в программу в виде пространственных зрительных образов, отчетов и наборов данных, а также генерируемые с разных датчиков или специализированной аппаратуры.

Наиболее используемыми средствами представления данных при решении различных задач являются FDT (*Fundamental Data Type*) и GDT (ISO/IEC 11404 — *General Data Types*), независимые от ЯП. При решении современных E-science задач сформировались новые неструктурированные данные, которые поступают с разных датчиков при исследовании Земли и атмосферы (проект Grid). Они входят в класс больших данных BigData и требуют нестандартных приемов для их анализа и обработки.

Тип данных — это фундаментальное понятие в теории программирования, которое определяет множество значений, набор операций, которые можно применять к этим значениям и способам их хранения.

Данные, которыми оперируют программы в современных ЯП, относятся к простым, структурным и сложным ТД FTD. Аксиоматика FTD разработана в 1970-х гг. специалистами (С. Дейкстра, Н. Вирт, В. Турский, П. Наур, Н. Агафонов и др.). В этой аксиоматике сложные ТД (массив, файл, таблица и др.) рассматриваются как агрегатная структура, которая создается из простых ТД. Операции над значениями типа задаются аксиомами и функциями, устанавливающими взаимно однозначное соответствие переданных данных. При их несоответствии производится преобразование к соответствующим ТД другого ЯП. Система аксиом определяет структуру множества значений типа, принадлежность ему отдельных элементов, их свойства и связь с другими ТД.

Теория структурной организации данных воплощена в ЯП Паскаль, Модула-2, Ада и др. Неструктурированные данные, которые генериру-

ются устройствами и датчиками в различных физико-технических экспериментах, для проведения вычислений, как правило, приводятся к форматам стандартных структур данных.

Тип данных — это фундаментальное понятие в теории программирования. ТД определяет множество значений, набор операций, которые можно применять к этим значениям, и способ реализации хранения значений и выполнения операций. Данные, которыми оперируют программы, относятся к определенным ТД.

Роберт Хиндли (1960) исследовал типизацию в комбинаторной логике и в языках, основанных на теории лямбда-исчисления. В 1980-х гг. он исследовал полиморфную систему типов для языков функционального программирования.

Тип (сорт) — устойчивая и независимая совокупность элементов в выделенном множестве ПрО.

Полиморфный тип — представление набора типов как единственного типа.

Математический тип представляется множеством всех значений, принадлежащих типу, и предикатной функцией, определяющей принадлежность объекта к данному типу. Тип в математике — «целое число» и не имеет ограничений, а в программировании ограничения связаны с диапазоном значений и объемом занимаемой памяти. Выход за границу целого не приводит к исключительной ситуации.

ЯП по способу определения ТД разделяются:

1) на языки с полиморфными ТД (например, Basic — тип вариант, Prolog, Lisp — списки). Переменные принимают значения любого типа и возвращают значения того типа. Выполнение $a + b$ трактуется как сложение чисел, если оба члена имеют числовые значения, или как конкатенация строк, если они имеют строковые значения, и недопустимые, если типы не совместимы. Это называется динамической типизацией, а в ООП и в теории чисел — полиморфным типом;

2) языки с неявным определением типа. Например, в языке Basic различают строковые типы с добавлением массивов;

3) языки с типом пользователя. Все типы практически совместимы. Их применяют в любых выражениях, а транслятор совершает необходимое преобразование. В языке Ада типы строго типизированы и каждая операция требует строгого задания типов. Никаких преобразований типов не выполняется.

8.1. Фундаментальные ТД в ЯП

Для анализа основных типов используется теория структурной организации данных. Она базируется на формализованном подходе к определению типов, основанном на аксиоматизации каждого типа и правилах выполнения операций над объектами. Система аксиом определяет структуру множества значений типа, принадлежность ему отдельных элементов, их основные свойства, связь с другими ТД.

Для каждой операции, выполняемой над переменными рассматриваемого типа, определяются типы операндов и результата. Теория структурной организации данных ориентирована на ее применение в ЯП. Во множестве ЯП основные положения данной теории воплощены в языках Паскаль, Модула-2, Ада и др.

В дальнейшем для обозначения ТД используются синтаксические конструкции ЯП (например, Паскаль).

Существуют четыре предопределенных ТД: целый (INTEGER), вещественный (REAL), булевый (BOOLEAN), символьный (CHARACTER). Они характеризуются тем, что на уровне архитектуры большинства ЭВМ имеются средства для обработки соответствующих ТД, так как они существуют практически во всех ЯП. Остальные являются производными и образуются с помощью средств конструирования новых ТД.

Все ТД делятся на *простые* и *структурные*. К простым относятся перечислимые и числовые, к структурным — массивы, записи, множества, списки, последовательности и т.д. Далее рассматриваются ТД: перечислимые на основе булевого и символьного типов; числовые на основе целого и вещественного типов; массивы и записи как объекты структурных типов.

Другие типы, представленные в теории структурной организации данных ЯП, рассматриваются без подробного анализа.

Простые ТД

Общее обозначение перечислимого типа имеет вид type $T = \{x_1, x_2, ..., x_n\}$. Здесь T — имя типа, а $x_1, x_2, ..., x_n$ — имена всех значений типа T, образующих множество значений X. Операции над перечислимыми типами включают бинарные операции отношения и унарные операции pred и succ, определяющие соответственно предыдущий и последующий элементы во множестве X. Все операции отношения ($<, \leq, >, \geq, =, \neq$) при построении алгебраических систем будут заменены одной ($\leq$), определяющей линейную упорядоченность множества X.

Примерами перечислимых типов могут служить булевый и символьный типы. Множество значений булевого типа состоит из двух элементов — false и true. Множество операций Ω, кроме перечисленных выше, включают операции булевой алгебры &, V, ¬. Запишем алгебраическую систему, соответствующую булевому типу:

$G^b = <X^b, \Omega^b>$;
X^b = {false, true};
Ω = {&, V, ¬, pred, succ, ≤}.

U^b имеет тип (2, 2, 1, 1, 1; 2) согласно арности соответствующих операций и предикатов.

Булевый тип в ЯП записывается в виде type T^b = (false, true). Если он стандартный (Boolean), то это описание в текстах модулей может опускаться.

Множество значений X символьного типа состоит из букв, цифр и специальных символов (знаков арифметических операций, знаков препинания и т.д.). Множество операций совпадает со множеством операций для любого

перечисленного типа. Алгебраическая система U^c, соответствующая символьному типу, имеет вид

$G^c = <X^c, \Omega^c>$;
$X^c = \{..., 'A', ..., 'X', ..., '0', '1', ..., '9'\}$;
$\Omega^c = \{\text{pred}, \text{succ}, \leq\}$.

Множество X упорядочено согласно внутреннему представлению символов для ОС ЕС.

Алгебраическая система U^c имеет тип <1,1; 2> согласно арности соответствующих операций и предикатов.

Символьный тип средствами ЯП записывается следующим образом: type T^c = (..., 'A', ..., 'X', ..., '0', ..., '9'). Если он стандартный (CHAR), то это описание в модулях может быть опущено. Операция ord, присваивающая каждому символу его порядковый номер во множестве X^c, и char, определяющая по порядковому номеру символа его значение, являются по существу операциями преобразования типов и поэтому во множество Ω^c не включены.

Для перечисленных типов характерны следующие аксиомы, которые в дальнейшем будут использоваться для анализа операций преобразования ТД:

$X.\min \in X$;
$X.\max \in X$;
$(\forall x \in X) \,\&\, (x \neq X.\max) \Rightarrow \text{succ}(x) \in X.$;
$(\forall x \in X) \,\&\, (x \neq X.\max) \Rightarrow \text{succ}(x) \neq X.\min$.

Здесь X.min и X.max обозначают соответственно минимальный и максимальный элементы множества X.

Практическое использование числовых типов всегда содержит ограничения, определяемые архитектурой ЭВМ (конечное значение количества разрядов слова памяти для представления чисел) или явным описанием в модулях (для ограничения диапазона значений отдельных элементов). Поэтому все числовые типы без нарушения общности анализа могут быть рассмотрены как отрезки. В общей форме отрезок записывается в виде type T = (X.min ... X.max). Здесь X.min и X.max обозначают соответственно минимальный и максимальный элементы отрезка. Для любого $x \in X$ выполняется условие $x.\min < x < x.\max$. Для стандартных числовых типов (INTEGER и REAL) приведенное выше описание в модулях может быть опущено. В этом случае элементы X.min и X.max не определены и зависят от конкретной реализации транслятора с ЯП на конкретном типе ЭВМ.

Над переменными целого типа и типов, для которых целый тип является базовым, выполняются те же операции, что и в случае перечисленных типов. Кроме того, добавляются операции целочисленной арифметики: унарный минус, +, −, ×, div (целочисленное деление) и mod (получение остатка от деления). Алгебраическая система G^i, соответствующая целому типу, будет иметь вид

$G^i = <X^i, \Omega^i>$;
$X^i = \{X^i.\min, X^i.\max + 1, ..., X^i.\max\}$;
$\Omega^i = \{+, \times, \text{div}, -, \leq\}$.

Во множестве Ω^i операция «–» соответствует унарному минусу. Остальные операции выражаются через операции, включенные в Ω. Алгебраическая система G^i имеет тип (2, 2, 2, 1; 2) согласно арности операций и предикатов. Форма записи целого и отрезков целого типа в ЯП имеет вид

type $T^i = (X^i.\min, ..., X^i.\max)$.

Над переменными вещественного типа и типов, для которых вещественный тип является базовым, выполняются операции отношения и обычные арифметические операции для действительных чисел (унарный минус, +, –, ×, /). Алгебраическую систему G^r, соответствующую вещественному типу, запишем следующим образом:

$G^r = <X^r, \Omega^r>$;
$X^r = \{x \mid X^r.\min \leq x \leq X^r.\max\}$;
$\Omega^r = \{+, \times, /, -, \leq\}$.

Во множестве Ω^r операция «–» соответствует унарному минусу. Алгебраическая система Ω^r имеет тип (2, 2, 2, 1; 2) согласно арности операций и предикатов.

Форма записи вещественного и отрезков вещественного типа в ЯП имеет вид

type $T^r = (X^r.\min, ..., X^r.\max)$.

Необходимо сделать замечание относительно порядка выполнения операций над любыми типами:

- все операнды приводятся к базовому типу;
- операция выполняется, как над объектами базового типа;
- для полученного результата выполняется обратный переход от базового к исходному типу.

Если результат принадлежит множеству значений данного типа, то операция выполняется верно. В противном случае результат операции не определен.

В дополнение к приведенным аксиомам для числовых типов будут использоваться следующие:

$(\forall x \in X) \Rightarrow T(T^0(x)) = x$;
$(\forall x_1 \in X)\ \&\ (\forall x_2 \in X) \Rightarrow (x_1 \leq x_2) \equiv (T^0(x_1) \leq T^0(x_2))$.

Здесь T^0 обозначает базовый тип для типа T. Операции $T^0(x)$ и $T(x)$ определяют преобразование значения к соответствующему типу. В этих обозначениях выполнение арифметических операций для числовых типов будет определяться следующим образом ($\oplus$ — любая двухместная арифметическая операция):

$(\forall x_1 \in X)\ \&\ (\forall x_2 \in X) \Rightarrow (x_1 \oplus x_2) \equiv T\,(T^0(x_1) \oplus T^0(x_2))$.

Операции сравнения числовых типов выполняются простой проверкой типов.

Структурные ТД

Отличительной особенностью структурных ТД в сравнении с простыми является то, что они содержат несколько упорядоченных элементов, обработка которых проводится как над целыми объектами, так и на уровне отдельных элементов. Структурные типы строятся из базовых типов и отличаются функциями конструирования и механизмами обработки. В качестве основных структурных типов в работе рассматриваются массивы и записи.

Массивы. Функция конструирования массива на основе базовых типов состоит в определении отображения из множества индексов на множество значений его элементов: $M : I \to Y$, где I — множество индексов, Y — множество значений элементов массива. В общем случае отображение M может не быть взаимно однозначным, если в различных элементах массива (элементы с разными индексами) содержатся одинаковые значения. Множество I является множеством значений перечислимого типа или отрезка целого типа. Элементы множества Y могут быть элементами любого типа, допускаемого в теории структурной организации данных.

Над массивами могут выполняться следующие операции:

- отношение для упорядоченных массивов (определяется как совокупность операций отношения для всех элементов массивов);
- сложение и вычитание однотипных массивов, т.е. массивов с одним и тем же множеством индексов (определяется как совокупность соответствующих операций над всеми элементами массивов с одинаковыми индексами);
- умножение двухмерных массивов по правилам умножения матриц.

Операция умножения накладывает ограничения на область значений индексов массивов, связанных с правилами умножения матриц, и не является общей для всех типов массивов. Операции сложения и вычитания выполняются только для числовых массивов. Поэтому они не входят в состав множества общих операций над массивами. Алгебраическая система, соответствующая ТД «массив», имеет следующий вид:

$$G^{\alpha} = < X^{\alpha}, \Omega^{\alpha} >;$$
$$X^{\alpha} = \{x \mid (\forall x_1 \in X^{\alpha}) \,\&\, (\forall x_2 \in X^{\alpha}) \Rightarrow I(x_1) = I(x_2)) \,\&\, (Y(x_1) \cup Y(x_2) \subset \overline{Y}(X^{\alpha}))\}, \Omega^{\alpha} = \{\leq\}.$$

Здесь $I(x)$ обозначает множество индексов для массива x, $Y(x)$ — множество значений элементов массива x, $\overline{Y}(X^{\alpha})$ определяет множество значений элементов для всех массивов, принадлежащих рассматриваемому типу. Второе выражение обозначает, что к данному типу принадлежат только те массивы, у которых множества индексов совпадают, а множество значений их элементов принадлежат одному и тому же множеству, характеризующему рассматриваемый тип. В раннее приведенных обозначениях отображение примет следующий вид (I — постоянно для всех x):

$$x : I \to Y(x),\ Y(x) \subset \overline{Y}(x).$$

Множество Ω^{α} состоит только из одного предиката. Поэтому алгебраическая система G^{α} фактически является алгебраической моделью типа 2.

Тип данных «массив» в ЯП записывается в виде

type T^a = array $T(I)$ of $T(\overline{Y})$,

где T^a — ТД «массив»; $T(I)$ — ТД индексов массива; $T(\overline{Y})$ — ТД для множества значений элементов массивов типа T^a.

Операции, определенные выше, выполняются над массивами как над единым структурным значением. Кроме того, над элементами множеств I и Y могут выполняться операции, соответствующие их типам данных. Определим операцию селектора для элементов массива. Пусть $I' \subset I$. Через E обозначим вложение $I' \to I$. Отображение $E \cdot M : I' \to Y$ называется ограничением отображения M на x и обозначается $M \mid I'$. Если I' состоит из одного элемента $I' = \{k\}$, то $M|\{k\}$ будет определять элемент множества Y, соответствующий индексу k. Заменяя M на x (согласно определению массива), получим обозначение операции селектора для элементов массива $x|\{k\}$. В ЯП обычно элемент массива обозначается в виде $x[k]$.

Необходимо отметить, что в предыдущем анализе рассматривались одномерные массивы. Многомерные массивы определяются рекурсивно. Данные типа $T(Y)$ в описании массива T^a, в свою очередь, могут быть массивами и т.д. При этом подходе последовательность предложений

type T^a = array $T(I^1)$ of $T(Y^1)$,
type $T(Y^1)$ = array $T(I^2)$ of $T(Y^2)$

эквивалентна следующей записи:

type T^a = array $T(I^1 \times I^2)$ *of* $T(Y^2)$.

В последнем предложении множество индексов массивов, принадлежащих типу T^a, представлено в виде прямого произведения множеств значений для типов $T(I^1)$ и $T(I^2)$. Операция селектора будет определяться в виде $\{\{x\}|\{i\}\}|\{j\}$, если $i \notin I^1$ и $j \notin I^2$. В ЯП элемент массива в этом случае будет обозначаться через $x[i][j]$ или $x[i, j]$.

Запись. Структурный тип «запись», как и массив, состоит из нескольких компонентов, которые могут быть разнородными, т.е. принадлежать различным простым или структурным типам. Функция конструирования записей представляет конкатенацию отдельных компонентов. Множество значений типа «запись» является прямым произведением множества значений ее компонентов. Ко множеству операций, выполняемых над записями, относятся операции отношения. При этом сравниваются только однотипные структуры (типы компонентов записей и их порядок следования одинаковые) и для каждого компонента записи выполняется операция отношения, которая интерпретируется как соответствующая операция для типа, соответствующего данному компоненту.

Пусть запись состоит из n компонентов. Каждый m-компонент ($m = 1, 2, ..., n$) имеет тип T^{v_m} и ему соответствует алгебраическая система $U^{v_m} = <X^{v_m}, \Omega^{v_m}>$. Индекс v_m соответствует одному из индексов для рассматриваемых в работе ТД.

Алгебраическая система для записи будет иметь вид

$G^z = <X^z, \Omega^z>$,
$X^z = \{x \mid (x = x^{v_1} \times ... \times x^{v_m})\ \&\ (x^{v_1} \in X^{v_1})\ \&\ ...\ \&\ (x^{v_n} \in X^{v_n})\}$, $\Omega^z = \{\leq\}$.

Аналогично массиву алгебраическая система G^z является моделью типа 2. Общая форма представления типа для записи имеет вид

type $T^z = (S_{v1} : T^{v_1}, ..., S_{v_n} : T^{v_n})$.

Здесь $S_{v_1}, ..., S_{v_n}$ — селекторы, а $T^{v_1}, ..., T^{v_n}$ — ТД для компонентов записи. Средствами ЯП запись $(S_{v_1}, ..., S_{v_n})$ — имена компонентов записи — описывается следующим образом:

type T^z = record
$S_{v_1} : T^{v_1}$
:
$S_{v_n} : T^{v_n}$
end.

Операции выполняются над записью как над единым структурным значением. Для обработки отдельных компонентов введем операцию селектора, аналогично соответствующей операция для массива. Пусть $I = \{S_{v_1}, ..., S_{v_n}\}$, $I' \subset I$. Обозначим через E вложение $I' \to I$. Пусть x — переменная типа «запись». Введем в рассмотрение множество $X^v = \{X^{v_1}, ..., X^{v_n}\}$. Тогда между I и X^v существует однозначное соответствие $M : I \to X^v$. Ограничение отображения M на I' обозначим через $M|\{S_{v_m}\}$. Если I состоит из одного элемента $I' = \{S_{v_m}\}$, то $M|\{S_{v_m}\}$ будет определять S_{v_m}-ю запись компонента. Заменяя M на x, получаем $x|\{S_{v_m}\}$. В ЯП компонентные записи обозначаются в виде $x \cdot S_{v_m}$, где S_{v_m} — имя соответствующего компонента.

Проведенный анализ относится к фиксированным записям. Для вариантных записей, последовательность компонентов которых определяется специальным признаком, можно поступить следующим образом. Признак является переменной перечислимого типа с конечным множеством значений. Каждому конкретному значению признака соответствует определенный вид записи. Поэтому вместо одной вариантной записи рассматривается семейство фиксированных записей для каждого значения признака. Семейство конечное, так как конечно число элементов перечислимого типа. Анализ вариантной записи будет сведен к перебору полученного семейства и обработке конкретной фиксированной записи. Поэтому в дальнейшем без снижения общности результатов будут рассматриваться только фиксированные записи.

Сложные ТД

Выше были рассмотрены основные структурные ТД — массивы и записи, которые встречаются в большинстве ЯП. Кроме них находят применение: множества, объединения, динамические объекты данных, списки, последовательности, стеки, деревья и др. Некоторые из этих типов являются стандартными в конкретных ЯП, а другие реализуются путем моделирования соответствующих структур и операций над ними. В данной работе для структурных типов приводится описательная информация и детальный анализ не производится. Это вызвано следующими причинами:

- некоторые типы являются собственностью одного или малого числа ЯП и не имеют аналогов в других ЯП;

• анализ некоторых типов, моделируемых средствами некоторого ЯП, может быть сведен к анализу базовых типов, которые подробно рассмотрены выше;

• реализация некоторых типов и выполнение операций над ними недостаточно формализованы.

Необходимо отметить, что чем сложнее ТД, тем разнообразнее операции над объектами этого типа и тем менее формализовано представление этих операций в ЯП.

Множества. Общая форма записи ТД «множество» следующая:

type T = powerset T^0,

где T определяет тип множества и T^0 является базовым типом для элементов множества.

Примером ЯП, в котором реализован аппарат множеств, является Паскаль.

Обычно T — перечислимый или целый тип. Для T реализованы все основные операции над множествами как математическими объектами — объединение, пересечение, разность, операции включения, определения тождественности и нетождественности. Операции селектора представляют собой выбор элемента типа T^0 объекта типа T. Операция конструирования формирует из одного или нескольких элементов типа T^0 объект типа T.

Объединения. Общая форма записи для объединения имеет вид:

type T = union (T^{v_1}, ..., T^{v_n}),

где T — тип объединения; T^{v_1}, ..., T^{v_n} — базовые типы.

В общем, на базовые типы ограничения не накладываются. Любой объект типа T имеет два компонента — значение и признак, по которому определяется один из типов T^{v_1}, ..., T^{v_n} для данного значения. Механизм реализации объединения подобен механизму реализации вариантных записей. Различие состоит в том, что сам признак скрыт, в отличие от признака вариантной записи, в которую он входит в качестве отдельного компонента. Все операции над объектами типа аналогичны операциям над вариантными записями.

Динамические объекты данных. Этот ТД в различных вариантах реализован во многих ЯП: Паскаль, Ада, ПЛ/1, Си и др. Общая форма записи для этого типа имеет вид

type T = pointer to T^0,

где T — определяет ссылочный тип; T^0 — базовый тип.

Базовым может быть любой тип. Объект типа T представляет адрес объекта типа T^0. Фактически ссылочный тип не является структурным, так как переменные этого типа содержат только одно значение, как и объекты простых типов. Однако использование ссылочного типа отличается от использования простых типов. Операции над ссылочными типами не формализованы. Например, язык Паскаль допускает только одну операцию — настройку на элемент базового типа (аналогично операции при-

сваивания). В то же время язык Си допускает над ссылочными типами операции адресной арифметики.

Списки. Списки являются конструкциями ЯП Лисп или могут реализовываться программным моделированием. Во втором случае элемент списка описывается как запись, содержащая одну или несколько компонентов ссылочного типа, которые обеспечивают связь между элементами списка. К последним в этом случае могут быть применены операции, аналогичные операциям над фиксированными записями. Кроме них существуют операции, применяемые к целому списку: выбор начального элемента, получение остатка списка, соединение списков, их сравнение, инвертирование, поиск элементов (атомов) в списке и др. Необходимо отметить, что в ЯП Лисп эти операции принадлежат к средствам самого языка. Для языков, не имеющих стандартных средств обработки списков, аналогичные операции должны быть реализованы в виде отдельных процедур. Все это не позволяет определить фиксированное множество стандартных операций над списками, характерное для всех или большинства ЯП.

Последовательности. Общая форма имеет вид

type T = sequence T^0,

где T — тип последовательности; T^0 — базовый тип.

Последовательность является одним из вариантов списка, у которого каждый элемент содержит только одну ссылочную переменную, что обеспечивает одностороннюю связь. Операции над последовательностями аналогичны операциям над списками. Одной из разновидностей последовательности является строка. Для строки каждый элемент, кроме ссылочной переменной, содержит компонент символьного типа. Обработка символьных строк допускается в языке Снобол.

Стек. Стек — это специальным образом организованная память. Она подчиняется дисциплине обработки LIFO (последним пришел — первым обработан). Обычно отдельный элемент стека принадлежит простому типу. Однако, используя средства программного моделирования, можно реализовать обработку стеков, содержащих элементы любых типов. На практике стеки реализуются в виде массивов или списков. В отличие от других дополнительных структурных типов множество операций над стеками фиксировано и включает: инициализацию стека, занесение элемента в стек, выбор из стека, анализ элемента, находящегося на вершине стека.

Операции над стеками могут быть реализованы на аппаратном уровне, средствами ЯП или программным моделированием.

Деревья. Деревья являются списковыми структурами и служат для представления графов или других аналогичных объектов. Множество операций над деревьями аналогично множеству операций над списками. Реализация этих операций зависит от конкретных приложений.

Кроме рассмотренных структурных типов используются и другие — таблицы, файлы, всевозможные комбинации перечисленных выше типов и др. Полный анализ всех ТД и их преобразований выходит за рамки настоящей работы. Предлагаемый подход применяется для использования этих типов.

8.2. ТД стандарта GDT

Общие ТД GDT (*General Data Types*) представлены в стандарте ISO / IEC 11404—2007 и являются:

- независимыми от языка (*Independed Language*) ТД, используются для формального описания концептуальных ТД как формализация метаданных для элементов данных, понятий элемента данных и значений областей;
- объединением технологий с текущими ЯП, интерфейсами и изображением данных ЯП (Java, IDL, Express, XML и др.);
- поддержкой полуструктурированных и неструктурированных совокупностей данных, где ТД и навигация проводятся как для неизвестных или неопределенных заранее ТД, перспективных, устаревших и сохранившихся особенностей, такие как элементы данных и допустимые значения;
- расширяемыми и позволяют пользователю ТД GDT описывать их как расширения.

Данный стандарт устанавливает номенклатуру и распределенную семантику для набора ТД, которые чаще всего используются в ЯП и в интерфейсах программных систем. В стандарте специфицированы как примитивные (базовые, независимые от других) ТД, так и сложные, которые полностью или частично определены на основе простых ТД.

Понятие «независимый от языков» ТД означает, что специфицированные ТД составляют классы ТД, реальные представители которых используются в ЯП и других объектах и основываются на концепции ТД.

8.2.1. Формальный синтаксис GDT

В стандарте определен формальный язык спецификации ТД. Некоторые понятия получены на основе форм Бэкуса-Наура, используемых для определения этого языка. Слово «знак» применяется для нотации символов, используемых для определения синтаксиса, тогда как слово «символ» характерно для ссылок на символы в языке спецификации реальных ТД.

Стандарт GDT ISO/IEC 11404 определяет ТД, которые частично совпадают с ТД FDT и включают:

- примитивные ТД (*real*, *integer*, *char*, *boolean*) и др.;
- сложные ТД (массив, запись, последовательность, портфель и т.д.);
- сгенерированные ТД — это ТД, полученные в результате генерации ТД этого стандарта;
- генератор ТД — это операция, которая создает новый ТД.

Примитивные ТД GDT

Рациональный (*rational*) — математический ТД, который отвечает рациональным (действительным) числам.

Масштабированный (*scaled*) — семейство ТД, пространством значений которого является подмножество пространства рациональных чисел и каждый отдельный ТД имеет фиксированный знаменатель и предусматривает аппроксимацию его значений.

Комплексный (*complex*) — семейство ТД, каждый из которых задает числовой математический ТД в структуре комплексных чисел.

Пустой (*void*) — ТД, который задает объект с необходимыми синтаксическими и семантическими описаниями, но не несет никакой информации.

Основные понятия GDT

Пространство значений в GDT — это совокупность (коллекция) значений ТД, которая определяется одним из следующих способов:

1) перечислением;

2) аксиоматичным определением;

3) подмножеством уже определенного пространства значений, которое имеет тот же набор свойств;

4) комбинацией любых значений некоторого, определенного пространства значений с помощью специфицированной процедуры конструирования новых значений.

Каждое отдельное значение принадлежит только одному ТД, хотя оно может принадлежать и нескольким подтипам этого ТД.

Равенство. Для каждого пространства значений существует понятие равенства (*equality*), задаваемого следующими аксиомами.

Аксиома 8.1. Для любых двух значений (a, b) из пространства значений выполняется условие равенства b, специфицированное как $a = b$, или неравенства b, специфицированное как $a \neq b$.

Аксиома 8.2. Не существует пары таких значений (a, b) из пространства значений, для которых одновременно выполняются условия $a = b$ и $a \neq b$.

Аксиома 8.3. Для каждого значения a из пространства значений выполняется условие $a = a$.

Аксиома 8.4. Для любых двух элементов значений (a, b) из пространства значений $a = b$, тогда и только тогда, когда $b = a$.

Аксиома 8.5. Если для произвольных трех элементов значений (a, b, c) из пространства значений выполняются условия $a = b$ и $b = c$, то выполняется условие $a = c$.

Для каждого ТД операция равенства *Equal* определяется как свойство равенства пространства значений. Для любых значений a и b из пространства значений *Equal* (a, b) есть true, если $a = b$, и false в противном случае.

Порядок. Пространство значений упорядочено, если для него установлено отношение порядка (*order*), которое задается знаком меньше или равно (≤) и удовлетворяет правилам:

1) для каждой пары значений (a, b) из пространства значений выполняется условие $a \leq b$ или $b \leq a$ или оба этих условия;

2) для любых двух значений (a, b), если $a \leq b$ и $b \leq a$, то $a = b$;

3) для любых трех значений (a, b, c), если $a \leq b$ и $b \leq c$, то $a \leq c$.

Запись $a < b$ используется для нотации: $a \leq b$.

ТД упорядочен, если отношение порядка определяется на его пространстве значений. Тогда операция *InOrder* определяется для произвольных двух значений a и b из пространства значений *InOrder*(a, b) есть true, если $b \leq a$, и false в противном случае.

Ограниченность. ТД ограничен сверху, если он упорядочен и существует такое значение U из его пространства значений, при котором для всех значений s этого пространства выполняется условие $s \leq U$. Значение U образует верхнюю границу пространства значений. Аналогично ТД ограничен снизу, если он упорядочен и существует такое значение L из его пространства значений, что для всех s этого пространства выполняется условие $L \leq s$. Значение L образует нижнюю границу пространства значений. ТД называется ограничением, если его пространство значений имеет верхнюю и нижнюю границу.

Кардинальность. Пространство значений ТД основывается на математической концепции кардинальности (*cardinality*) и оно может быть конечным или бесконечным. ТД должен иметь кардинальность (мощность) своего пространства значений. В стандарте предусмотрены три важные категории ТД, пространство значений которых является:

1) конечным;
2) точным и бесконечным;
3) приближенным (пространство значений которой может быть конечным или бесконечным).

Точный и приближенный ТД. Если каждое значение в пространстве значений концептуального ТД можно отличить от другого значения в пространстве, то ТД считается точным (*exact*).

Математические ТД, которые имеют значения и не имеют определенного представления, называются приближенными (*approximate*). Пусть M — математический ТД, а C — соответствующий вычисляемый ТД, P — преобразователь пространства значений M в пространство значений C. Тогда для каждого значения v' в C существует соответствующее значение ТД v в M и такое действительное значение h, что $P(x) = v'$ для всех x в M и $|v - x| < h$. Таким образом, v' — это приближение C для всех значений M, находящееся в h-области значения v''. Кроме того, для одного значения v' в C существует более чем одно такое значение в M, так что $P(y) = v'$. Таким образом, C не является точной моделью M.

ТД называется числовым (*numeric*), если концептуально его значения определяются количественно (в системе нумерации). ТД, значения которого не имеют этого свойства, называется нечисловым (*non-numeric*).

Пространство значений. Пространство значений — это совокупность (коллекция) значений ТД, которая определяется одним из следующих способов:

- исчислением;
- аксиоматическим определением согласно основным положениям;
- как подмножество уже определенного пространства значений, которое имеет тот же набор свойств;
- как комбинация любых значений некоторого, уже определенного пространства значений с помощью специфицированной процедуры конструирования новых значений.

Каждое отдельное значение в пространстве принадлежит только одному ТД, хотя оно может принадлежать и нескольким подтипам этого ТД.

Каждый концептуальный ТД являются точным. Невычислимый ТД является бесконечным. Если каждое значение в пространстве значений концептуального ТД можно отличить от другого значения в пространстве этой модели, то ТД считается точным (*exact*).

Сгенерированные ТД

Сгенерированные ТД (*generated datatypes*) — это ТД, полученные в результате применения генератора ТД.

ТД, с которыми работает генератор, называются параметрическими или компонентными. Сгенерированный ТД семантически зависит от параметрических ТД, но имеет собственные характеристические операции. Важной характеристикой всех генераторов ТД является то, что генератор может применяться для многих разных параметрических ТД. Генераторы указателя и процедуры генерируют ТД, значения которых атомарные, тогда как генератор выбора и агрегатных ТД генерирует ТД, значения которых позволяют производить их декомпозицию.

Генератор ТД (*datatype generator*) — это концептуальная операция над одним или несколькими ТД, которая создает новый ТД. Генератор ТД оперирует с ТД, а не с его значениями и представляет собой:

1) набор критериев для характеристик ТД, над которыми будут выполнены операции;

2) процедуры конструирования, которые допускают набор ТД с данным критерием для создания нового пространства значений из пространств значений этих ТД;

3) набор характеристических операций, которые применяются в конечном пространстве значений для завершения определения нового ТД.

Агрегатный ТД (*aggregate datatype*) — это сгенерированный ТД, каждое значение которого получено из значений параметрических ТД. Параметрические ТД агрегатного ТД или его генератор включают в себя имена компонентов ТД. Генератор агрегатного ТД генерирует ТД с помощью алгоритмической процедуры пространства значений агрегатного ТД.

В отличие от других сгенерированных ТД агрегатный ТД обеспечивает доступ к компонентам значений через характеристические операции. Агрегатные значения разных типов различаются между собой свойствами, которые характеризуют отношение между компонентами ТД и отношение между каждым компонентом и агрегатным значением.

8.2.2. Генератор сложных ТД GDT

Выбор генерирует ТД. Каждое значение образуется из любого набора альтернативных ТД. Этот ТД логически учитывает их соответствие значению другого ТД с признаком (*tag*).

Указатель генерирует ТД, каждое значение которого устанавливает средства ссылки на значение другого ТД, специфицированного ТД *element-type*. Эти значения ТД указателя являются атомарными.

Процедура генерирует ТД, каждое значение которого является значением других ТД, которые называют параметром. Такой ТД включает в себя набор всех операций над значениями конкретной коллекции ТД, концептуально атомарных.

Агрегатные ТД

Генератор агрегатного типа генерирует ТД путем:

- применения алгоритмической процедуры к пространству значений его ТД для создания пространства значений агрегатного ТД;
- обеспечения набора характеристических операций, специфических для генератора.

Таким образом, многие свойства агрегатных ТД составляют свойства генератора, независимо от ТД компонентов. В отличие от других генераторов ТД для агрегатных характерно то, что значение компонентов агрегатного значения получается с помощью характеристических операций.

Ниже приведены агрегатные ТД.

Запись генерирует ТД, значения которого составляют совокупность значений компонентов ТД и каждая совокупность имеет значение для каждого компонента ТД, специфицированного фиксированным идентификатором поля *field-identifier*.

Набор генерирует ТД из пространства значений из поднаборов пространства значений ТД элемент с операциями, свойственными математическому множеству *set*.

Портфель генерирует ТД, значения которого составляют коллекции образцов значений ТД элемент. Многочисленные образцы того же значения могут подаваться в этой коллекции, а в каком порядке — несущественно.

Последовательность генерирует ТД, значениями которого являются упорядоченные последовательности значений ТД из значений, несвойственных этому ТД; одно и то же значение может встречаться многократно в этой последовательности.

Массив генерирует ТД, значения которого ассоциируются с произведением пространств одного или нескольких конечных ТД, которые называются индексными ТД. Пространство значений этого ТД таково, что каждому значению из пространства индексного ТД соответствует только одно значение элемента.

Таблица генерирует ТД, значения которого составляют коллекции значений из пространства одного или нескольких полей, такое что каждое значение из пространства задает ассоциации между значениями его полей.

Объявленные ТД

Объявленный ТД (*defined*) — это ТД, определенный с помощью описания типа type-declaration. **Type-identifier** — идентификатор некоторого объявленного типа, который ссылается на ТД или генератор ТД. **Actual-type parameters** соответствует номеру и типу formal-type parameters объявленых в type-declaration. Таким образом, каждый actual-type-parameter соответствует formal-type-parameter в соответствующей позиции списка formal-type-parameter-list. Если formal-parameter-type составляет type-specifier, то actual-type parameters будет value-expression, определять значение ТД, специфицированным как formal-parameter-type. Если он является «type», то actual-type-parameter будет type-specifier и иметь нужные свойства этого параметрического ТД.

Type-declaration идентифицирует type-identifier в type-reference с одним ТД, семейством ТД или генератором ТД. Если идентификатор типа type-identifier задает семью ТД, то type-reference ссылается на тот член семьи, пространство значений которого определяется посредством type-definition после замены каждого значения actual-type-parameters для всех входов formal-parametric-value. Если type-identifier задает генератор ТД, то type-reference означает ТД, который получается применениями генератора ТД к реальным параметрическим ТД.

Характеристические операции

Это операции, которые создают значение любого типа с помощью генератора ТД, создавая пространство значений параметрических ТД. Такие операции необходимы для выделения ТД по их названиям и генерации агрегатных ТД как композиции следующих операций:

1) с нулевой арностью для генерации значений этого ТД;

2) с унарной операцией (арности 1), которая превращает значение этого ТД в новое значение этого же ТД или в значение *boolean*;

3) с арностью 2, которая преобразуют пары значений этого ТД в значение этого же ТД или в значение *boolean*;

4) с *n*-арностью, преобразующей упорядоченные *n*-элементные группы значений, каждая из которых относится к определенному ТД и может быть параметрическим или агрегатным типом.

Практически не существует уникальной коллекции характеристических операций для заданного ТД. Одна коллекция операций для ТД (или генератора типов) достаточна для выделения этого ТД среди других из пространства значений той же мощности.

Таким образом, существует посимвольная замена, которая преобразует все пространство значений одного ТД (*domain*) в подмножество значений пространства другого ТД (диапазон, *range*) так, чтобы значения отношений и характеристических операций домена сохранялись в соответствующих значениях отношений и характеристических операций диапазона ТД.

8.2.3. Преобразование ТД ISO/IEC 11404—96

Преобразование с помощью LI языка стандарта. Этот язык обеспечивает преобразования ТД независимо от ЯП следующих видов:

- внешнее преобразование из внутренних ТД ЯП в LI ТД;
- внутреннее преобразование LI ТД в ТД ЯП;
- обратное внутреннее преобразование.

Суть *внешнего преобразования* ТД и генераторов ТД состоит в следующем:

- для каждого примитивного типа для сгенерированного внешнего ТД преобразование устанавливается связь с одним LI ТД;
- для каждого внутреннего ТД преобразование задает связь между допустимым его значением и эквивалентным значением соответствующего LI ТД;
- для каждого значения LI ТД определяется наличие значения любого внутреннего ТД, преобразуемого в LI ТД с его значением.

Внутреннее преобразование задает связь примитивного ТД или сгенерированного в LI ТД с конкретным внутренним ТД ЯП. Данное преобразование обладает следующими свойствами:

• для каждого LI ТД (примитивного или сгенерированного) преобразование определяет наличие этого ТД в ЯП;

• для каждого LI ТД преобразование определяет отношение между этим типом и эквивалентным значением соответствующего внутреннего типа ЯП;

• для каждого значения внутреннего ТД преобразование определяет, является ли это значение образом (после преобразования) какого-то значения LI ТД.

Обратное внутреннее преобразование для LI ТД состоит в преобразовании значений внутреннего ТД в соответствующее значение LI-типа при наличии соответствия и отсутствия двусмысленности. Это преобразование для ЯП является коллекцией обратных внутренних преобразований LI ТД.

8.2.4. Преобразование данных для связи компонентов в ЯП

Под **преобразованием данных** будем понимать методы трансформации и отображения ТД в формате одного компьютера к соответствующему представлению другого компьютера, а также методы устранения различий в описаниии ТД в разных ЯП.

К ним относятся:

• интерфейсы связи (RPC, RMI, IDL, API и др.) между разнородными компонентами;

• протоколы передачи данных (TCP/IP, UDP, GIOP и др.);

• классы функций преобразования различающихся типов и структур данных ЯП и генерации соответствующих новых ТД;

• системные процедуры по преобразованию передаваемых данных между разными объектами в ЯП.

Особую роль выполняют функции преобразования неэквивалентных ТД, используемых при обмене. В рамках системы АПРОП разработано множество функций преобразования ТД (64 функции), которые обеспечивают преобразование ТД в следующих случаях:

• несоответствие числа параметров в списках фактических и формальных параметров;

• несогласованность типов передаваемых параметров в вызывающей и вызываемой программах;

• несоответствие во множествах значений типов фактических и формальных параметров;

• изменение множества значений отдельных параметров;

• наличие нестандартных операций передачи данных;

• неверное описание ТД передаваемых параметров;

• отсутствие обратных преобразований ТД выходных параметров и др.

Взаимно однозначное соответствие данных, заданных в формальных параметрах вызова программы, устанавливается с помощью системных функций отображения одних ТД в другие[1].

---

[1] См.: *Лаврищева Е. М., Грищенко В. Н.* Сборочное программирование. Основы индустрии программных продуктов; *Лаврищева Е. М., Рыжов А. Г.* Применение теории общих типов данных стандарта ISO/IEC 11404 к данным Big Data // Journal Euroasia-science, 2016; *Лавріщева К., Стеняшин А.* Подход к трансформации общих ТД стандарта ISO/IEC 11404 для использования в гетерогенных середах // International Conference on High Performance Computing, October 8—10, 2012, Kyiv. P. 227—234.

8.3. Неструктурированные ТД

В 2008 г. в британском журнале «Nature» появился термин Big Data. Этот термин определял новый вид данных в электронной науке — в археологии, ядерной физике, биологии, медицине и др. К этим данным относятся данные измерений при проведении многочисленных физических экспериментов и обработке огромного количества документов в международном бизнесе и др. Данный термин касается наборов данных, которые превышают возможности их размещения в существующих БД и не соответствуют их форматам для представления в них. Большие данные накапливаются в хранилищах данных на разных серверах Интернета и требуют новых методов их представления и обработки при решении разного рода научных и коммерческих задач.

Проведена классификация больших данных, которая включает три группы: быстрые данные (*Fast Data*), их объем измеряется терабайтами; большая аналитика (*Big Analytics*) для данных, объем которых исчисляется петабайтами; глубокое проникновение (*Deep Insight*) — для данных порядка экзабайт, зеттабайт. Группы различаются не только объемами, но и качеством решений по их обработке.

Независимо от этой классификации стоит задача анализировать неструктурированные данные из класса Big Data, появившиеся в хранилищах данных, которые генерируются с разного рода датчиков исследования недр Земли и пространства, списков записей многочисленных страховых компаний, задаваемых разными изображениями, фотографиями, документами и др.

Неструктурированные ТД GDT

Это совокупность данных, которые концептуально:

- структурированы и имеют оба компонента ТД или метода доступа;
- наполовину структурированы и имеют один ТД или метод доступа к значениям;
- неструктурированы и имеют набор данных неодинаковой природы.

Такие данные представляются в форматах, основанных на XML и пространственных зрительных образах, отчетах. Эти данные накапливаются как большие объемы нереляционных и неструктурированных данных. Они интегрируются с существующими приложениями, соответствуют нормативным требованиям и расширяют возможности для пользователя при работе с информационными системами. Стандарт формального описания неструктурированных ТД еще не существует. К таким ТД относятся и большие данные.

Термин «большие данные» распространяется и на наборы данных, размер которых превосходит возможности типовых БД по занесению, хранению, управлению и анализу информации. Он определяет анализ больших объемов данных и их управление.

Согласно отчету McKinsey of Institute Big Data — это новый рубеж для инноваций, конкуренции и производительности больших данных.

Мировые репозитарии данных постоянно растут. В отчете аналитической компании IDC (Digital Universe Study) «Исследование цифровой

вселенной, 2011» и компании EMC предсказывалось, что общий мировой объем созданных и реплицированных данных составляет около 1,8 зеттабайта (1,8 трлн гигабайт) — примерно в 9 раз больше того, что было создано ранее.

Такие данные предполагают больше, чем просто анализ огромных объемов информации, поскольку они в основном представлены в формате, плохо соответствующем традиционному структурированному формату БД. К ним относятся веб-журналы, видеозаписи, текстовые документы, машинный код или, например, геопространственные данные и др. Большие данные хранятся во множестве разнообразных хранилищ. Корпорации могут иметь доступ к ним, но не иметь необходимых инструментов, чтобы установить взаимосвязи между этими данными. По прогнозам, количество данных на планете будет удваиваться каждые два года вплоть до 2020 г. Традиционные методы анализа информации не могут удовлетворять проблемам увеличения объемов данных, их накопления и обновления, что приводит к необходимости создавать технологии больших данных и формальные методы работы с ними.

Способы хранения неструктурированных данных

Приложения, основанные на работе с реляционными и нереляционными данными, в основном используют одну из трех архитектур:

1) реляционные данные находятся в БД, а данные в формате больших нереляционных двоичных объектов (BLOB) — в файловых системах или на файловых серверах;

2) реляционные данные находятся в БД, а нереляционные — в хранилище, предназначенном для BLOB-данных;

3) реляционные и нереляционные данные находятся в БД.

Кроме сложностей, связанных с хранением неструктурированных данных, требуется учитывать то, как эти данные будут использоваться приложениями:

- на этапе разработки используются структурированные и неструктурированные данные и составлении кода, отвечающего за создание, загрузку, обновление и удаление неструктурированных данных, требуется обеспечить транзакционную согласованность между источниками реляционных и неструктурированных данных;
- для индексирования неструктурированных данных и поиска;
- для извлечения метаданных, доступных в явной форме (например, из полей форм или из атрибутов файлов), и предоставления этих данных пользователю;
- для преобразования содержимого документа в форматы, пригодные для выполнения поиска и составления запросов (например, преобразование звуковых файлов в текстовые, в которых затем может быть выполнен поиск по запросу БД или при помощи обработчика полнотекстового поиска).

Каждый из перечисленных подходов имеет свои преимущества и недостатки. Например, хранение неструктурированных данных на файловых серверах или в хранилищах, предназначенных для хранения BLOB-данных, может снизить денежные затраты на память. Это усложняет разработку

приложений и управление ими, так как приложения должны поддерживать согласованность между записями в БД и системами, в которых находятся BLOB-данные, связанные с этими записями. С другой стороны, при хранении BLOB-данных в БД одна БД становится удобным централизованным хранилищем, но при этом происходит увеличение затрат и может произойти снижение быстродействия.

Пример неструктурированных данных, которые получены с датчика космических исследований (2012):

{58} n_vers:byte; {номер версии программы записи информации}
{59-62} n_krit:longint; {номер критерия экспресс-обработки}
{63-64} n_bad:integer; {количество кадров, отбракованных обработкой}
{65-76} fam_fiz:array[1..12]of char; {фамилия дежурного физика}
{77-116} skr_k:array[1..40] of byte; {информация с искровой камеры}
{117-118} n_vved; {количество введенных кадров с разными номерами}
{119—120} n_good:integer; {количество кадров, записанных в первичный банк кадров}

8.4. Подход к решению проблемы обмена разными ТД

При обмене данными требуется трансформация передаваемых данных в случае несовпадения ТД параметров, отличающихся форматами данных платформ вычислительной среды, количеством передаваемых параметров и др.

Задача взаимодействия пары разнородных программ состоит в определении взаимно однозначного соответствия между множествами фактических параметров $V = \{v_1, v_2, ..., v_k\}$ и формальных параметров $F = \{f_1, f_2, ..., f_k\}$ программ. Для отображения множеств V и F требуется провести разбиение множеств таким образом, чтобы каждому подмножеству из V соответствовало только одно подмножество из F. Для каждого $f^i \in F$ рассматривается полный прообраз $V^i \in V$. Различные прообразы V^i и V^j могут иметь или не иметь одинаковые элементы.

Если $V^i \cap V^j \neq \Theta$, то V^1 и V^j объединяются в одно подмножество. Соответственно будет проведено объединение в одно подмножество элементов F^i и F^j. Данная процедура применяется до тех пор, пока не будет исчерпано множество V. В результате получается два семейства подмножеств

$$\Pi = \{V^1, V^2, ..., V^m\} \text{ и } \Phi = \{F^1, F^2, ..., F^m\},$$

таких что

$$\bigcup_{t=1}^{m} V^t = V,\ V^t \bigcap V^{t1} = \Theta \text{ при } t \neq t_1,$$

$$\bigcup_{t=1}^{m} F^t = F,\ F^t \bigcap F^{t1} = \Theta \text{ при } t \neq t_1,$$

для которых существует однозначное отображение, записываемое в виде $A : \Pi \rightarrow \Phi$.

В зависимости от количества элементов во множествах V^t и F^t имеют место следующие случаи:

1) $|V^t| = |F^t| = 1$. Отображение A для данных подмножеств включает операции преобразования ТД;

2) $|F^t| > 1$ и $|V^t| = 1$. Означает, что одному фактическому параметру структурного ТД соответствует несколько формальных параметров скалярных типов или структурных с меньшим уровнем структурирования. Отображение A включает операции селектора отдельных компонентов и преобразования ТД;

3) $|V^t| > 1$ и $|F^t| = 1$. Означает соответствие нескольких фактических параметров одному формальному. Отображение A содержит операции преобразования типов и конструирование структурного типа с более высоким уровнем структурирования, чем у передаваемых параметров;

4) $|V^t| > 1$ и $|F^t| > 1$. Свидетельствует о существовании глубокой связи вызывающего и вызываемого модулей, которая зависит от внутренней логики функционирования модулей. Такие связи противоречат свойствам модулей, и поэтому данный случай не поддается формальному анализу при комплексировании модулей. На основе проведенного анализа свойств отображения A выделяются операции для информационного сопряжения модулей.

8.4.1. Операции преобразования ТД

Эти операции позволяют осуществлять преобразование ТД T^t_α в T^t_β и имеют вид

$$P^{tq}_{\alpha\beta} = (T^t_\alpha , T^q_\beta).$$

Здесь данные типа T^t_α преобразуются в T^q_β, α и β соответствуют языкам l_α и l_β. Предполагается, что множество ТД каждого ЯП упорядочено и индексы t и q определяют конкретные элементы этого множества. Для ЯП, имеющих средства конструирования новых типов, t и q будут функциями от других индексов, и упорядоченность типов может определяться тем, что новый тип t будет конструироваться из типов, для которых индексы не больше t.

Каждый ЯП имеет определенное множество предопределенных ТД и базовых операций конструирования, что определяет основу всего множества типов. Новый тип будет иметь индекс, функционально зависящий от индексов предопределенных типов и конкретных операций конструирования.

Фундаментальные ТД ЯП и способы их представления определены выше. Для проведения отображения ТД разработан подход, в котором каждый ТД из класса FDT задается алгебраической системой вида

$$G^t_\alpha = <X^t_\alpha, \Omega^t_\alpha>,$$

в которой t — ТД языков L; X^t_α — множество значений ТД; Ω^t_α — множество операций над этими ТД.

Простым, сложным ТД и большим данным поставлены в соответствие следующие классы алгебраических систем:

$$\Sigma_1 = \{G^b_\alpha, G^c_\alpha, G^u_\alpha, G^r_\alpha\},$$
$$\Sigma_2 = \{G^a_\alpha, G^z_\alpha, G^u_\alpha, G^e_\alpha\},$$
$$\Sigma_3 = \{G^s, G^{ns}\}.$$

В каждом классе этих систем преобразование ТД $t \to q$ для пары языков l_t и l_q основано на следующих свойствах отображений:

1) системы G^t_α и G^q_β — изоморфны, если их ТД q, t определены на одном и том же множестве ТД;

2) между значениями X^t_α и X^q_β ТД t, q существует изоморфизм, если множество операций Ω^t_α и Ω^q_β разные. Если множество $\Omega = \Omega^t_\alpha \cap \Omega^q_\beta$ не пустое, то имеет место изоморфизм двух систем $G^{T'}_\alpha = <X^t_\alpha, \Omega>$ и $G^{q'}_\beta = <X^q_\beta, \Omega>$. Если ТД различаются, например t — строка, а тип q — вещественное, то между множествами X^t_α и X^q_β не существует изоморфного соответствия.

Отображения сохраняют линейный порядок элементов в силу линейной упорядоченности элементов алгебраических системы из этих классов.

8.4.2. Подход к генерации FDT ⇔ GDT

В процессе изучения и перевода стандарта ISO/IEC 11404—2007 студентами КНУ была разработана схема генерации общих ТД к фундаментальным и наоборот, т.е GDT ⇔ FDT. Она базируется на библиотеке функций (процедур) в XML, некоторые из них разработаны в магистерских работах студентов КНУ.

Функции библиотеки выполняют соответственно (рис. 8.1):

- преобразование ТД ЯП$_1$, ..., ЯП$_n$;
- представление ТД FDT к виду специальных функций;
- преобразование GDT к виду FDT;
- эквивалентные отображения GDT ⇔ FDT.

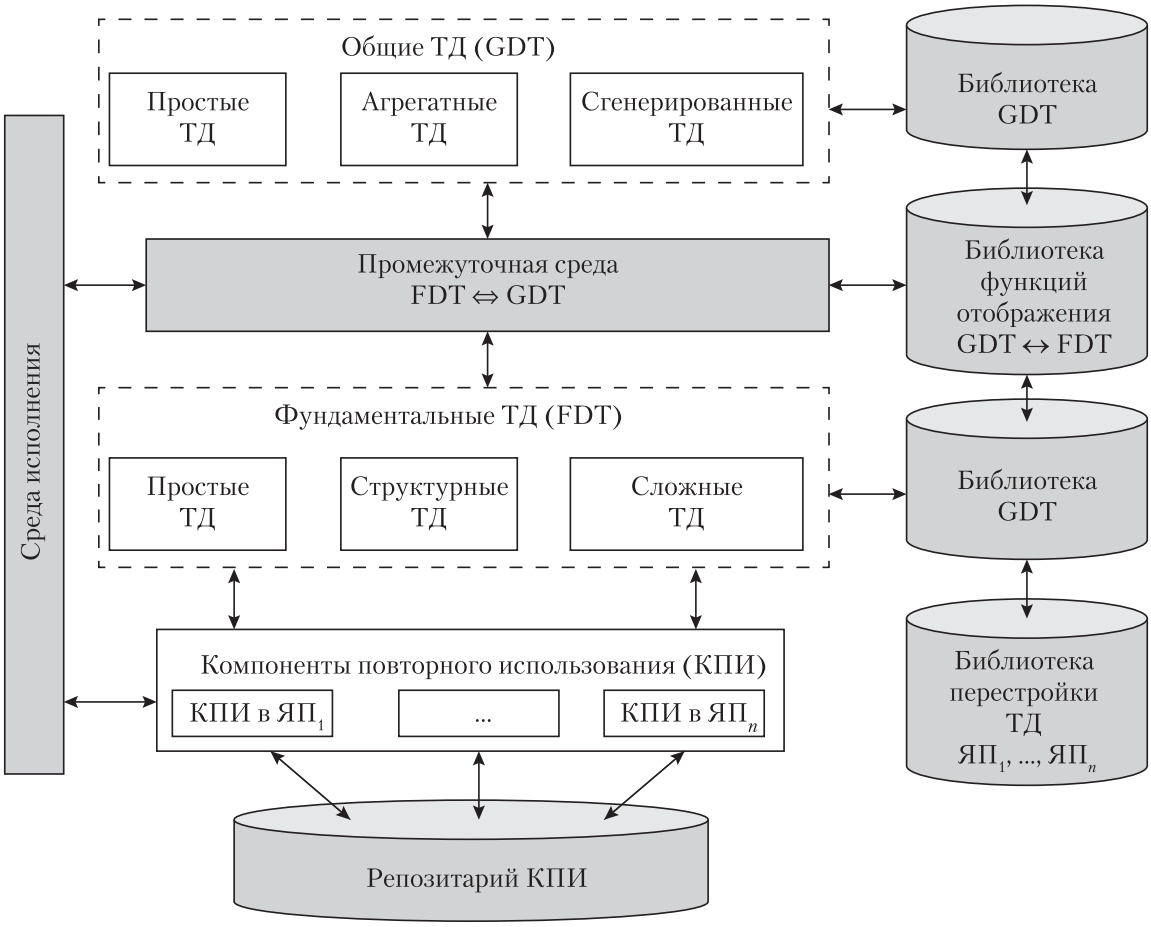


Рис. 8.1. **Схема преобразования GDT ⇔ FDT**

Реализованная библиотека ТД GDT ⇔ FDT является необходимым условием сборки разнородных объектов в новых ЯП и применения для нестандартных ТД в Big Data.

8.5. Практика реализации разнородных ТД

Система типов CTS (*Common Type System*) в Microsoft .NET является универсальной (рис. 8.2). CTS определяет способ объявления, использования и управления типами в среде CLR (*Common Language Runtime*), а также обеспечивает межъязыковую сборку. Ее основные задачи:

- межъязыковая интеграция и безопасность ТД;
- объектно ориентированная модель реализации ЯП;
- взаимодействие объектов, написанных на разных ЯП;
- библиотека типов-примитивов для всех FDT и частично GDT.

В системе выделены две группы типов: типы-значения (*value type*) и типы-ссылки (*reference type*). Эта система — общая для всех ЯП, которые выполняются в .NET и задаются пользователем в своей программе на ЯП. В системе существует механизм отображения типов CTS в ТД конкретных ЯП и наоборот.

Типы-значения — это статические типы, память для которых выделяется в стеке и освобождается после завершения работы программы. Эти типы встроены в CTS, их значения могут занимать память от 8 до 128 байтов. Перечень этих типов приведен в табл. 8.1 и на рис. 8.2

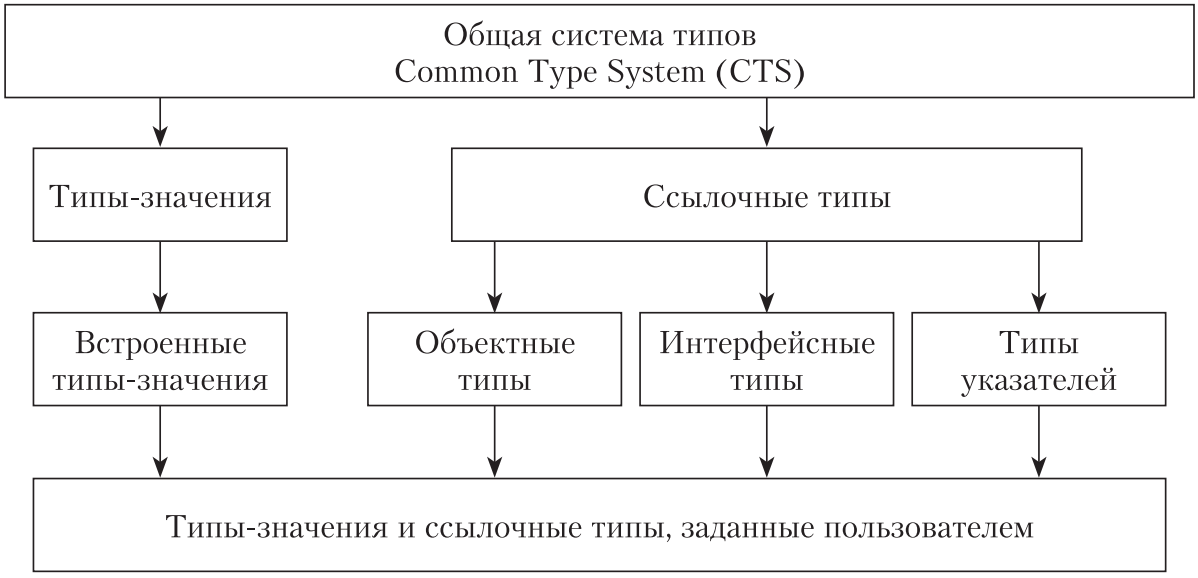


Рис. 8.2. **Структура системы типов MS.NET**

Типы-ссылки (или ссылочные (динамические) типы) используют указатели на объекты, которые они типизируют, а также механизмы централизованного хранения и освобождения памяти. Ссылочные типы включают: объектные типы (*object type*), интерфейсные типы (*interface type*), типы-указатели (*pointer type*).

Типы CTS — это стандартная система типов, которая описывает все ТД, поддерживается средой выполнения, определяет, как одни ТД, взаимодей-

ствуют с другими, и они будут представлены в форме метаданных .NET. Они могут использоваться после инициализации (операциями вызова, get и set и т.д.), а также преобразовываться и объединяться в пространства имен System.

CLR выявляет и загружает типы .NET и обеспечивает управление, безопасность и интеграцию программ в ЯП. Для того чтобы объединить компоненты, используются ЯП программ и язык спецификации CLS (*Common Language Specification*). CLR — это общая языковая среда, задающая общую систему типов, которая используется всеми ЯП в плане обработки целочисленных данных или данных с плавающей точкой и строк для всех ЯП. Этот язык позволяет выполнять интеграцию кодов (которые могут быть написаны на разных ЯП) в режиме дизайна, а не в режиме выполнения. Основное его назначение — соблюдение всех программных зависимостей, управление памятью и предоставление множества сервисов, облегчающих создание, внедрение приложений и улучшение надежности последних.

Язык CLS позволяет специфицировать следующие данные: классы, структуры, интерфейсы, типы, перечисления, а также встроенные ТД (см. табл. 8.1).

Таблица 8.1

Встроенные ТД CTS в C# и C++/CLI

ТД CTS	VS.NET	C#	C++/CLI
System.Byte	Byte	Byte	unsigned char
System.Sbyte	Sbyte	Sbyte	signed char
System.Int16	Short	Short	Short
System.Int32	Integer	Int	int or long
System.Int64	Long	Long	int64
System.UInt16	Ushort	Ushort	unsigned short
System.UInt32	UInteger	Uint	unsigned int or long
System.UInt64	Ulong	Ulong	unsigned _int64
System.Single	Single	Float	Float
System.Double	Double	Double	Double
System.Object	Object	Object	Object^
System.Char	Char	Char	wchar_t
System.String	String	String	String^
System.Decimal	Decimal	Decimal	Decimal
System.Boolean	Boolean	Bool	Bool

При трансляции программ в ЯП создается модуль DLL или EXE и сборщик (*assambly*) на языке IL. Код IL не зависит от платформы, на которой будет осуществляться выполнение. Промежуточный язык MSIL представляет собой промежуточный уровень в процессе взаимодействия исходного

кода, написанного на любом языке .NET, с машинным языком в виде кода на языке псевдо-ассемблера. При запуске кода на выполнение MSIL конвертируется в специфический код CPU. Благодаря тому, что CLR поддерживает один и больше JIT-компиляторов, один и тот же набор MSIL может быть скомпилирован и выполнен на разных платформах.

8.6. Средства поддержки приложений с Big Data

При разработке приложений, работающих с большими данными, используются следующие средства анализа и описания данных:

- A/B testing, crowdsourcing data fusion;
- integration genetic algorithms machine learning;
- natural language processing;
- signal processing simulation и visualisation;
- Parallel Processing Massiv;
- search-based applications data mining и др.

Большие данные могут также быть представлены как tensors, которые управляют вычислением, как, например, полилинейное обучение подпространств (*multilinear subspace learning*). Технологии обращения к большим данным включают массив параллельно-обрабатывающей (MPP) БД и извлечение данных на основе приложения, распределенные файловые системы, распределенные БД и инфраструктуры (приложения, хранение и вычисляющие ресурсы) Интернета.

К инструментальным средствам разработки приложений с большими данными относятся:

- Oracle JDeveloper;
- NetBeans;
- Pach for Eclipce и др.

Гавная проблема Big Data — анализ больших данных. Этим занимаются перечисленные фирмы, а также IBM, Microsoft, Yahoo, Google и др. Появились новые средства анализа данных:

- Apache Hadoop, Apache Hive, Apache Pig и др.;
- MapReduce;
- Revolution Analytics и др.

В настоящее время главной является Apache Hadoop.

Формальные механизмы работы с неструктурированными данными еще не сформировались. Их необходимо разработать.

8.7. Характеристика стандартов ISO

Стандарт ISO/IEC 11404-GDT

В 1996 г. вышел стандарт «Информационные технологии — ЯП, среды и системные интерфейсы ПО — Языки ТД».

Начиная с 1996 г. он применяется для обеспечения общей семантики ТД ЯП, БД и других систем, которые используют ТД. Стандарт был пере-

смотрен через 7 лет под названием «Информационные технологии — GDT для стандартизации и обеспечения совместимости данных».

Согласно стандарту ISO/IEC 11404 GDT ТД состоит из трех основных понятий: пространство значений, набор свойств и характеристические операции.

Пространство значений включает набор значений для каждого ТД, которые имеют определение:

- аксиоматических из фундаментальных понятий;
- совокупности пространства значений;
- совокупности произвольных значения из некоторых уже определенных пространств.

Каждый ТД содержит свойства из следующих характеристик:

- равенство между элементами пространства значений;
- наследование данных из области значений;
- конечное, бесконечное (счетное) или бесконечное (несчетное) множества значений пространства;
- точность по отношению к модели вычислений значений элементов области пространства;
- пространство концептуальных значений как точных и приближенных.

Например, вещественные числа могут быть приближенными на любой платформе компьютера.

Другой особенностью ТД являются характеристические операции над значениями из пространства значений. Например, ТД целое число стандарта ISO/IEC 11404 GDT имеет следующие характеристические операции:

- Equal(x, y: integer) — boolean is true if x and y определяют одно и то же значение и false в противном случае;
- Add(x,y: integer) — является операцией integer;
- Multiply(x, y: integer) — является математической мультиоперацией integer;
- Negate(x: integer) — является значением y в такой операции Add(x, y) = = 0 как integer;
- NonNegative(x: integer): boolean есть true if x = 0 or x может быть разработан путем итерации 1, т.е. if x = Add(1, Add(1, ... Add(1, Add(1,0)) ...)) else false;
- InOrder(x,y: integer): boolean = NonNegative(Add(x, Negate(y)));
- Quotient(x, y: integer) — integer, где 0 < y, является верхней границей множества всех integers z, таких, что Multiply(y, z) ≤ x;
- Remainder(x, y: integer) — integer, where 0 ≤ x and 0 < y, = Add(x, Negate(Multiply(y, Quotient(x, y))).

Стандарт предоставляет богатую библиотеку генераторов ТД (например, запись, массив и т.д.), которые могут быть использованы для создания сложных структур данных (поддерживаемых многими ЯП). Ниже приведен простой пример использования нотации стандарта ISO/IEC 11404:2007:

```
type employee_record = new
record
(
```

```
name: characterstring, // employee name
marital_status: state (single, married),
exemptions: integer, // number of exemptions for tax deduction
pay_rates: array (0..20) of pay_rate_type, // an array of
records
),
type pay_rate_type = new
record
(
code: characterstring, // pay code
wage: scaled(10,4), // hourly wages to 4 decimal digits
),
```

Стандарт изменения данных (Data interchange standards). Consensus-Building Process

Этот стандарт содержит следующие разделы:

- развитие с нормативной формулировкой;
- консенсус по вопросам политики с формальными уточнениями;
- обзор основных положений и пересмотр стандарта;
- внедрение новых технологий и снятие устаревшей технологии.

В стандартах обмена данными используются процессы разработки и технического обслуживания средств преобразования ТД. Ниже приведена схема для разработки программ с функциями изменения данных.

Requirements. Требования позволяют описывать спецификации и после завершения необходимо подтвердить их удовлетворение первоначальными требованиями, формальным нормам процесса идентификации потребностей.

Functionality. Функциональность помогает разграничить и содержать в рамках стандарта характеристики элементов.

Conceptual Model. Концептуальная модель описывает виртуальную реализацию моделей согласно теории операций и может быть использована для устранения неоднозначности в семантике. Результатом являются технические условия, их адаптация и внесение изменений в технологию.

Semantics. Семантика понятий описывается отдельно от концептуальной модели и привязки. Результатом является семантика взаимосвязи и (или) спецификация повышения степени интероперабельности.

Encodings. Привязки «стандарта поведения» (семантики) к конкретным кодировкам, форматам, интерфейсам и протоколам выполнения.

В рассмотренных стандартах приведены фундаментальные, общие и неструктурированные ТД, а также определен новый подход к работе с неструктурированными ТД, основанный на механизмах генерации FDT и GDT.

Контрольные вопросы и задания

1. Дайте представление о фундаментальных ТД в ЯП.
2. Опишите простые и сложные ТД в FDT.
3. Приведите формальный общий синтаксис GDT.

4. Расскажите про генерацию сложных ТД GDT к простым.
5. Дайте определение неструктурных ТД.
6. Какие имеются подходы к решению проблемы обмена ТД?
7. Назовите некоторые операции преобразования ТД.
8. Опишите сущность подхода к генерации FDT ⇔ GDT.
9. Сформулируйте подходы к реализации разнородных ТД.
10. Назовите средства поддержки работы с Big Data.
11. Расскажите, какие стандарты есть в области ТД.

Глава 9
ОНТОЛОГИЧЕСКИЙ МЕТОД КОНЦЕПТУАЛЬНОГО МОДЕЛИРОВАНИЯ ДОМЕНОВ

Онтология как метафизический термин возник еще в 1613 г. В информатике под онтологией понимается спецификация концептуальных знаний о некоторой ПрО (физике, медицине, биологии и др.) путем описания множества объектов и связей между ними. Формально онтология содержит понятия, организованные в таксономию их описания и правила вывода.

Концептуальная модель онтологии определяется в виде

$$O = <X, R, F>,$$

где X — конечное множество понятий ПрО или домена; R — конечное множество отношений между понятиями; F — конечное множество функций интерпретации.

Такая модель служит связью между людьми, компьютерными системами и средами. Она сохраняет информацию и может подавать необходимые знания о готовых объектах и программах разного назначения, которые находятся в современных библиотеках и репозиториях, реализующих объекты онтологии. Концептуальная модель может фиксировать знания, необходимые для аккредитации соответствующих учебных программ в вузах и сертификации специалистов в масштабе государства и международного сообщества.

На сегодня в Интернете функционирует ряд онтологий: Sensus с общими знаниями и понятиями естественного английского языка (более 70 000 терминов и их дефиниций); онтология понятий электронной коммерции; глобальная онтология продуктов и услуг (OOH); коммерческие онтологии SCTG, RosettaNet транспортных потоков товаров и т.п. Кроме того, действуют медицинские онтологии (Galen для определения клинического состояния; UMLS Национальной медицинской библиотеки США; ON9 для аттестации медицинских систем по общим параметрам), а также инженерные, химические онтологии и т.п. К средствам представления знаний относится также общероссийский веб-портал математических ресурсов, всемирная информационная математическая библиотека MathLab, Ret и др.

Таким образом, в основе представления любой системы знаний о ПрО лежат понятийная база, совокупность концептов (понятий) и отношений между ними. Следующим шагом нормализации знаний является зафиксированная классификация понятий в виде тезауруса. Процесс построения

модели ПрО, ориентированной на ее понимание человеком, называют *концептуальным моделированием*.

Каждая ПрО или *домен* имеет присущую ему систему понятий, систему «умалчивания» того, что должно считаться «общеизвестным» в рамках своего домена, собственные характерные свойства (атрибуты), отношения и правила поведения. Фактически концептуальная модель — это посредник между заказчиком и разработчиком. Степень ее формализации должна быть достаточной, чтоб обеспечивать точность и однозначность толкования разными носителями интересов в разработке и доступность для понимания.

По форме представления онтологии домена используется концептуальная модель ПрО, которая отображает систему понятий с характерными свойствами (атрибутами), отношениями и правилами поведения. Такая модель служит коммуникации (между людьми, между компьютерными системами) и сохранению информации в компьютерной среде.

Онтология задает соглашение об общем использовании понятий, которое включает средства представления предметных знаний и договоренностей о методах рассуждений.

Для *описания онтологии* используются язык OWL со спектром семантических языков разметки и словарь RDF для доступа к онтологическим знаниям в Интернете и обмена ими. Описание онтологии в языке OWL — это последовательность аксиом и фактов, информация о классах, свойствах и ресурсах идентификатора (ID) и веб-документов, которые импортируются через URI и могут ссылаться на XML-схему datatype по форме: <DatatypeID>:: = <URI ссылка>.

Факты — это информация в форме классов и свойств со значением ресурса. Аксиомы используются, чтобы сопоставить класс и свойство ID с частичными или полными спецификациями характеристик, давая другую логическую информацию о классах и свойствах. Каждая аксиома класса содержит совокупность более общих классов и совокупность локальных ограничений свойства. Класс может быть эквивалентом, подмножеством или пересечением более общих классов с ограничениями. В OWL аксиома класса содержит совокупность описаний, которые могут иметь вид обобщенных классов, ограничений, наборов ресурсов, булевых комбинаций описаний. Компания OMG для управления, взаимодействия и применения семантики доменов предложила метамодели ODM (*Ontology Definition Metamodel*).

Понятийная база домена определяет не только терминологию, которой должны пользоваться носители интересов, участвующие в процессе анализа требований, а также существенные отношения между понятиями и их интерпретацией.

Семантика понятия может быть охарактеризована совокупностью общих существенных признаков тех явлений и предметов, которые входят в это понятие. Совокупность явлений, охваченных понятием, называется его *объемом*. Отношения обеспечивают:

1) обобщение существенных признаков понятия, расширяя круг охваченных понятиями объектов и объема;

2) конкретизацию понятий путем добавления существенных признаков, благодаря чему содержание понятия расширяется, а объем сужается;

3) агрегацию, т.е. объединение ряда понятий в новое понятие, существенные признаки которого при этом могут образовывать сумму признаков;

4) ассоциацию как наиболее общее отношение, которое утверждает наличие связи между понятиями, не уточняя зависимости от содержания и объемов.

Для отдельных доменов могут использоваться специфические для них отношения. Онтология позволяет держать пользователя в возможном пространстве предопределенных понятий, содержание которых зафиксировано и понятно как разработчику, так и заказчику.

Примеры описания онтологии домена «Вычислительная геометрия» (выполнено группой студентов КНУ во главе Т. А. Лихо) и домена ЖЦ стандарта ISO/IEC 12207 приводятся ниже[1].

9.1. Описание доменов средствами онтологии

Для отображения знаний в определенной проблемной области используются концепты. Источниками знаний могут быть терминологические или толковые словари в близких проблемных областях, требования на разработку ПС и другие документы.

Начальным шагом построения онтологии является выявление существенных объектов ПрО, установление отношений между ними и предоставление им уникальных наименований. Для классов объектов выбираются имена, уникальные в границах домена.

Источником поиска объектов ПрО могут быть те понятия проблемной области, которым соответствуют отдельные задачи по достижению определенных профессиональных целей в компьютерном виде и могут быть использованы при обращении к системе. Таким образом, происходит последовательная декомпозиция сложной проблемы, которую можно выявить в домене ПрО:

1) проблема трансформируется в совокупность целей, необходимых для ее применения;

2) каждая из целей трансформируется в совокупность возможных примеров использования системы, которые отражаются в сценарии;

3) сценарии трансформируются в процессе их анализа в совокупность взаимодействующих объектов.

Определенная таким образом цепочка *проблема — цели* или *работы — сценарии — объекты* отображает степени концептуализации для дости-

---

[1] См.: *Лаврищева Е. М.* Подход к формальному представлению онтологии жизненного цикла программных систем // Весник КГУ, серия физ.-мат. наук. 2013. № 4. С. 140—149; *Lavrischeva E.* Ontological Approach to the Formal Specification of the Standard Life Cycle // Science and Information Conference-2015, Jule 28—30, London, UK. P. 965—972. URL: http://www.conference.thesai.org; *Lavrischeva E.* Ontology of Domains. Ontological Description Software Engineering Domain—The Standard Life Cycle // Journal of Software Engineering and Applications. 2015. № 8. P. 1—15.

жения понимания проблемы, последовательного снижения сложности ее частей и повышения уровня формализации моделей последних. Для объектов устанавливаются атрибуты, связи и состояния.

Анализ ПрО состоит в моделировании ее характеристик. Первая задача, которая решается при построении семейств ПС, — это функциональность в данной ПрО. Традиционным подходом к ее решению являются построение характеристических диаграмм (*feature diagram*) и DSL описание ПрО. Распространение онтологического подхода в разных отраслях науки обусловило проведение исследований применительно к моделированию ПрО и конкретизации подхода FDD.

Онтология применяется для описания понятий и их связей в домене, удобном для разработки члена СПС в языке XML.

Одной из новых сфер применения онтологического подхода являются описания ПрО в DSL и трансформация этих моделей в программный код.

Проектирование домена в языке DSL включает в себя анализ ПрО, в который входят:

1) идентификация ПрО;
2) сбор знаний о ПрО;
3) кластеризация этих знаний в терминах семантических понятий и операций над ними;
4) проектирование проблемного языка типа DSL, который хорошо описывает понятия ПрО и их использование.

Следующий шаг — это реализация ПрО, которая включает:

1) создание библиотек для реализации семантических понятий;
2) создание компиляторов для варианта языка DSL;
3) описание в DSL программ ПрО и их компиляцию.

Моделирование ПрО называется доменной инженерией (*domain engineering*). Для моделирования используются следующие методологии анализа ПрО: ODM (*Organizational Domain Modeling*), FODA (*Feature-Oriented Domain Analysis*), DSSA (*Domain Specific Software Architectures*). Кроме того, сложились подходы к разработке семейств: Lucent, FAST (*Family-Oriented Abstraction, Specification and Translation*). Семейства программ отождествляются с линиями программных продуктов (Product Line / Product Family).

Разработка некоторой онтологии предусматривает глубокий структурный анализ ПрО и включает в себя следующие действия:

1) выделение концептов — базовых понятий данной ПрО;
2) определение «высоты дерева онтологии» — количество уровней абстракции;
3) распределение концептов по уровням;
4) построение связей между концептами;
5) консультации с разными специалистами для исключения противоречий и неточностей.

Процесс построения онтологии является итеративным.

Методология построения онтологии допускает обозначение целей и области применения создаваемой онтологии, а также построение онтологии, которая включает следующие действия:

1) фиксацию знаний о ПрО, определение основных понятий и их отношений в выбранной ПрО; создание точных непротиворечивых определений для каждого основного понятия;

2) выбор или разработку специального языка для представления онтологии;

3) непосредственное задание фиксированной концептуализации на выбранном языке представления знаний;

4) обеспечение возможности использовать знания ПрО;

5) отделение знаний об ПрО от оперативных знаний.

9.2. Основные понятия онтологии представления ПрО

Существуют языки спецификации онтологии (Ontolingua, CycL, OKBC, OCML, Flogic). Более современные языки, основанные на веб-стандартах, такие как XOL, SHOE или UPML, RDF(S), DAML, OIL, OWL, созданы специально для задания онтологий этими средствами.

В целом различие между традиционными и веб-языками спецификации заключается в выразительных возможностях описания ПрО и в некоторых механизмах логического вывода понятий.

Важное направление использования языков онтологии связано с построением ПС, которые оперируют знаниями в онтологии.

На сегодняшний день язык XML (*Extensible Markup Language*) фактически стал стандартом разметки данных для их сохранения и обмена информацией между разными пользователями. XML активно используется в самых разных областях научной и коммерческой деятельности, и в ближайшее время его популярность будет только расти. Однако XML и его реализации представляют низкий уровень описания ресурсов и необходимые средства автоматической трансформации моделей ПрО (и онтологии) в XML-схемы, пригодные для работы приложений.

Связь языков моделирования с онтологией ПрО. Основной задачей моделирования ПрО, которая используется для разработки архитектуры ПС, является UML. Путем последовательного применения моделей можно сгенерировать описание классов и заготовок программного кода для выбранных ЯП и платформ.

Альтернативный подход с подобными целями — онтологический, который называется *Ontology-Driven Software Development*. Он позволяет описывать классы, которые отображают понятие ПрО. В отличие от предыдущего модели ПрО могут не только использоваться для генерации кода, но и являются «выполняемыми» артефактами.

К общим средствам описания онтологий ПрО относятся классы, слоты как свойства классов и фасеты, описывающие конкретные типы и возможные диапазоны значений. В аксиомах определяют дополнительные правила. Абстрактные классы являются контейнерами конкретных классов и могут содержать абстрактные атрибуты. Атрибуты понятий ПрО в системе Protégé называются слотами.

Конкретные классы содержат слоты, которым могут быть даны значения (экземпляры атрибутов).

Слоты в Protégé описывают свойства классов и экземпляров (возможные атрибуты). Согласно фреймовой модели представления знаний, которая используется в Protégé, слот — это фрейм. Слоты определяются независимо от любого класса, и один и тот же слот может принадлежать разным классам.

Фасеты позволяют вводить ограничение на типы и диапазоны значений экземпляров (значений атрибутов), подобные соответствующим понятиям XML-схемы. Кардинальность слота определяется количеством значений слота, ограничениями для типов значений (например, целое, символьное и т.п.) экземпляров класса и граничных значений (min, max). Фасеты определяют ограничение на присоединение слота к фрейму класса.

Слоты-образцы (*template slot*) и собственные (*own*) слоты. Слот можно присоединить к фрейму (класса) одним из двух способов: как слот-образец или как собственный слот. Собственный слот, присоединенный к фрейму, описывает свойства объекта, представленного фреймом. Классы также могут иметь собственные слоты. Например, документация класса является собственным слотом, присоединенным к классу, поскольку описывает сам класс, а не экземпляры класса.

ЖЦ онтологии включает в себя такие процессы: проектирование, валидация, поддержка, развертывание, оценивание, обновление, визуализация, интеграция, повторное использование.

Один из источников домена — требование к системам, предоставляющим сведения о совокупных свойствах домена, которые должны быть трансформированы в базу знаний домена и КПИ, нацеленных на решение его задач.

9.3. Средства проектирования доменов в DSL

Описание в DSL ПрО требует компиляции или интерпретации. При этом могут применяться стандартные компиляторы или специальные инструменты. Альтернативным подходом к реализации DSL является расширение существующих языков. К средствам производства программ для ПрО относятся: RUP компании Rational, стандарт зрелости процессов и коллективов разработчиков CMM — SE института SEI при университете Карнеги — Меллона, UML, языки ООП — C++, C#, Java и DSL (*Domain Specific Language*) и др.

9.3.1. Структура описания домена в DSL

DSL — это язык описания специфики и модели домена, предметной области, которые используются для генерации членов семейства систем домена.

Целью модели являются составные языки и правила для образования правильных моделей в DSL. К ним относятся[1]:

- абстрактный синтаксис для определения правил языка описания модели ПрО;
- конкретный синтаксис — применяется для написания нотаций, которые используются для описания моделей;

---

[1] *Рябко Д. М.* Подход к реализации среды разработки для DSL // Проблемы программирования. 2011. № 4. С. 3—12.

• семантический смысл — полная система правил поведения отдельных языковых конструкций модели.

Схема описания некоторой проблемы в GPL и DSL дана на рис. 9.1.

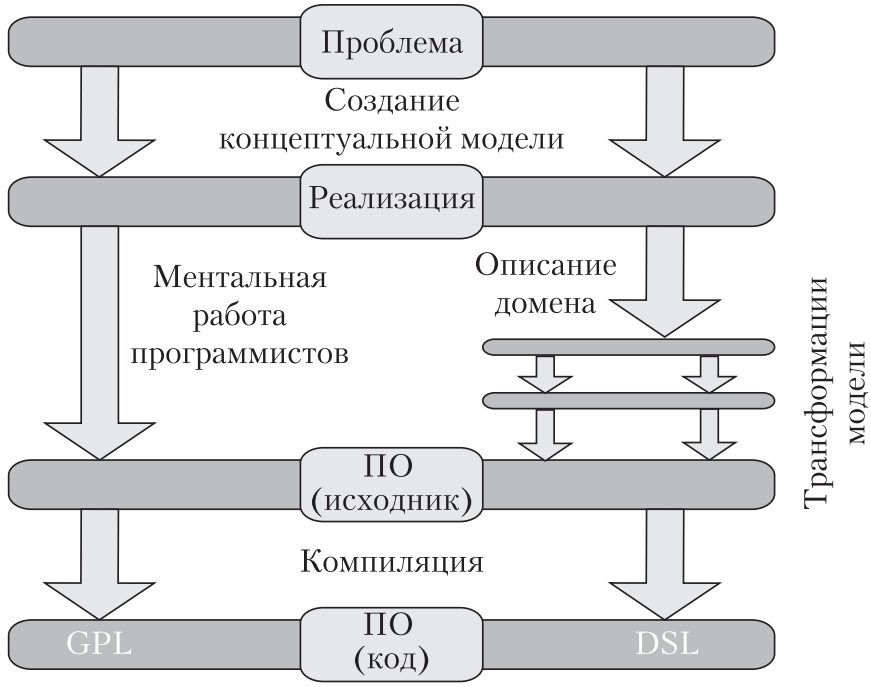


Рис. 9.1. **Схема описания проблемы в GPL и DSL**

Модель ПрО задается подмоделями ($M_{\text{про}i}$) и является метамоделью. Описание каждой модели $M_{\text{про}i}$ выполняется соответствующим проблемно ориентированным DSLi-языком. Это описание трансформируется в соответствующий ЯП реализации (рис. 9.2).

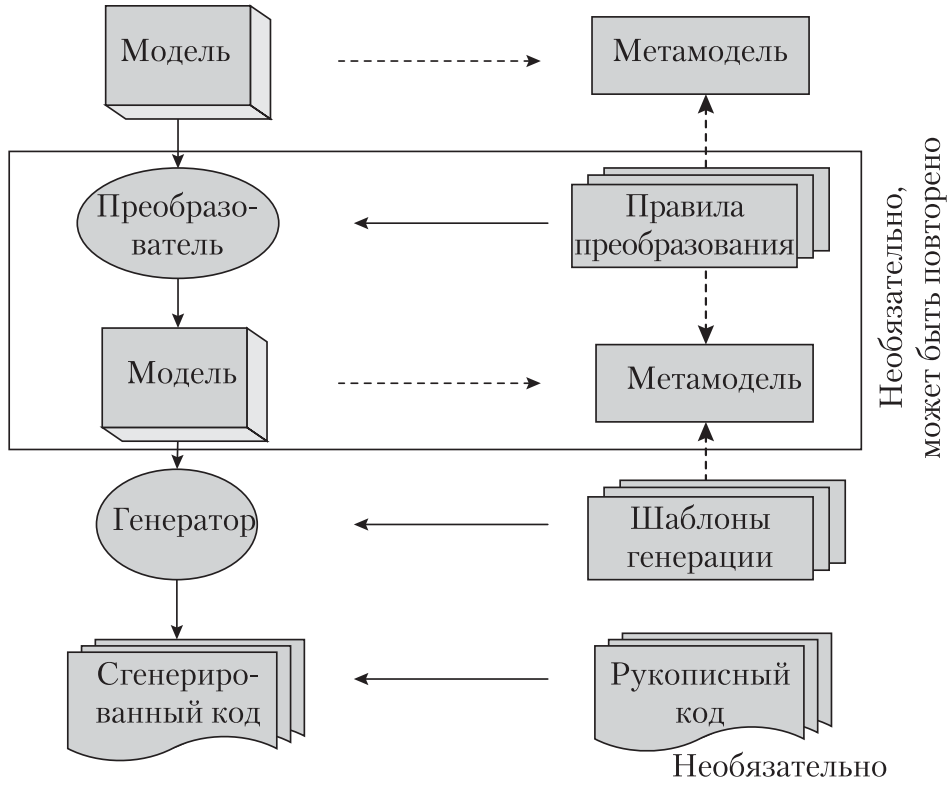


Рис. 9.2. **Модели и компоненты разработки в DSL**

Каждая из этих подмоделей отображает понятие и специфику соответствующей системы из семейства систем домена. На их пересечении находятся общие понятия, характеристики и ограничения, которые отображаются в МХ ПрО.

9.3.2. Трансформация модели домена в DSL к ЯП

Процесс преобразования моделей ПрО в языке DSL выполняется на разных уровнях (рис. 9.3).

Трансформация описания моделей ПрО проводится с помощью средств модельно-управляемой разработки MDD (*Model Driven Development*), архитектура системы которой на двух уровнях платформ — независимой PIM (*Platform Independent Model*) и зависимой PSM (*Platform Specific Models*).

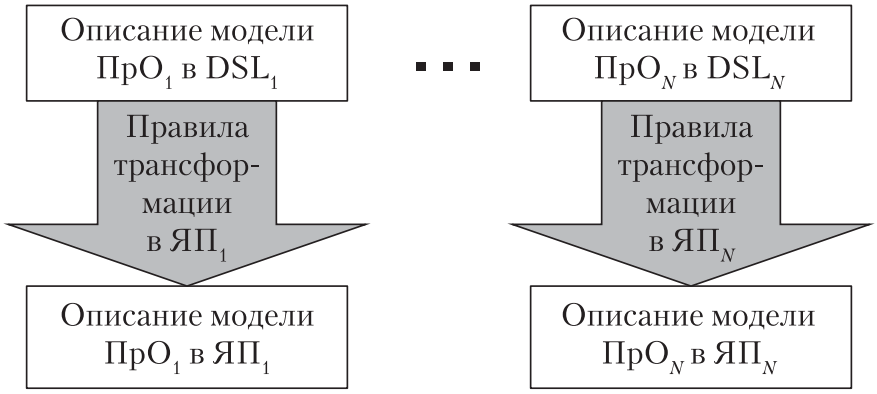


Рис. 9.3. **Трансформация описания моделей ПрО в DSL**

Концепции двухуровневого моделирования архитектуры MDA и отображения PIM ↔ PSM соответствует идеологии построения семейства ПрО (рис. 9.4).

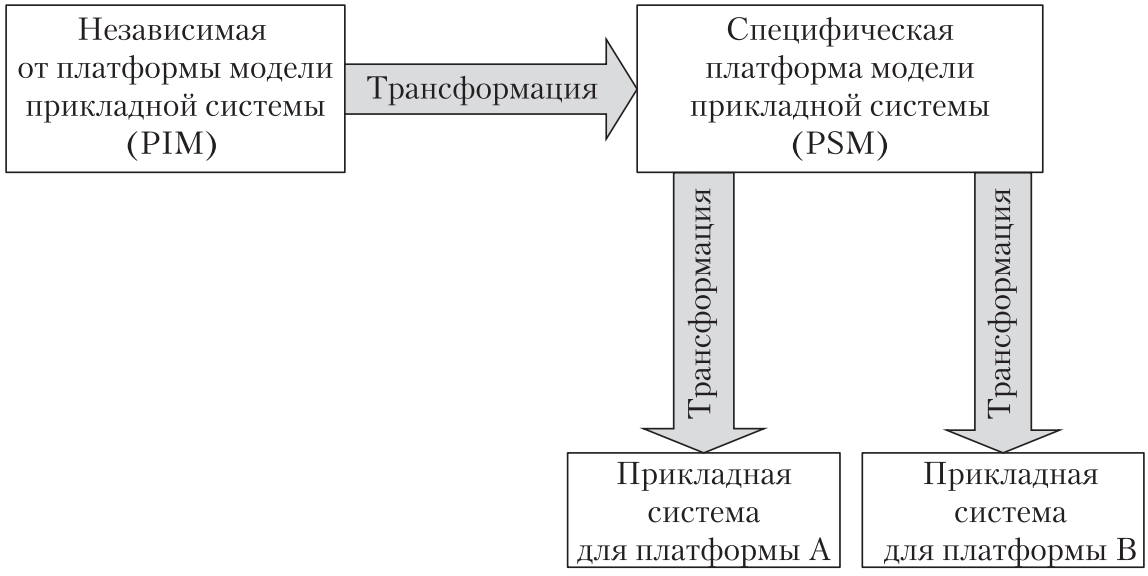


Рис. 9.4. **Схема трансформации модели ПрО**

Методология базируется на отображении описаний моделей прикладных систем (*Application model*) в DSL-языках в ЯП. Выбор альтернативной платформы соответствует точке вариантности в семействе систем. Эта

точка невидима на уровне конкретной ПС и при управлении вариантами платформы автоматические преобразует PIM ↔ PSM.

9.3.3. Онтология домена «Вычислительная геометрия»

ПрО «Вычислительная геометрия» (*computational geometry*) — предмет преподавания в современном университете, состоит из разделов: комбинаторная и численная вычислительная геометрия.

Комбинаторная вычислительная геометрия рассматривает геометрические объекты как дискретные сущности. *Численная вычислительная геометрия* имеет дело с представлением объектов реального мира в форме, пригодной для дальнейшей компьютерной обработки. Она рассматривается как дальнейшее развитие геометрии и как раздел компьютерной графики. Данный раздел геометрии содержит разработанные эффективные алгоритмы и структуры данных для решения задач, которые заданы в терминах базовых геометрических объектов: точек, отрезков, многоугольников, многогранников и др.

Вычислительная геометрия сосредоточивается на вычислительной сложности как алгоритме работы над очень большими наборами данных из десятков или сотен миллионов точек. Основные ее задачи можно классифицировать разными способами и с разными критериями. Различают следующие задачи: статические задачи и задачи геометрического поиска.

К *статическим задачам* относятся:

- выпуклая оболочка, которая имеет набор точек, необходимых для нахождения наименьшего выпуклого многоугольника, содержащего все точки;
- сечение отрезков как задача нахождения сечений в наборе отрезков;
- триангуляция Делона;
- диаграмма Вороного, которая имеет набор точек, разделяющих пространство на сектора, каждая точка которых является более близкой к точке из набора;
- задача нахождения ближайшей пары точек из набора точек, чтобы найти кратчайшее расстояние;
- евклидов кратчайший путь: соединение двух точек евклидового пространства (с полигональными препятствиями) кратчайшим образом;
- триангуляция многогранника как задача разбиения его на внутренние треугольники.

Задачи геометрического поиска на входных данных состоят из двух частей: пространство поиска и запросов. Пространство поиска требует предварительной обработки для обеспечения эффективного выполнения нескольких запросов, а именно:

- региональный поиск для обработки набора точек и эффективного поиска точек, которые содержатся в запрошенном регионе;
- локализация точек в пространстве проблем на регионы, и определение структуры данных, где находится данная точка;
- поиск ближайшей соседней точки для эффективного ее использования;
- трассирование луча для заданного набора объектов в пространстве со структурой данных, которые пересекают запрошенный луч.

Динамические задачи требуют постоянного изменения входных данных. Алгоритмы решения таких задач базируются на динамических структурах данных и дополнительных вычислительных ресурсах. Региональный поиск, построение выпуклой оболочки можно проводить над множеством точек, которые изменяются. Задачи вычислительной сложности связаны с поиском структур данных, их модификацией и обеспечением ответов на запросы.

Вариация — это задача, в которой курсор мыши лежит внутри многоугольника или он постоянно перемещается, при этом многоугольник не изменяется. Аналогично можно проверить определенный летательный аппарат, который изображен на экране радара, не пересекая границ страны. Такие задачи можно считать задачами геометрических запросов, а в CAD-системах — динамическими.

Таким образам, к класам задач *вычислительной геометрии* относятся: статические задачи, задачи геометрического поиска, динамические задачи и задачи вариации. Описание понятий и задач онтологии «Вычислительная геометрия» выполнялось на языке Protégé. Платформа Protégé поддерживает два основных способа моделирования онтологий с помощью редакторов Protégé-Frames и Protégé-OWL. Онтологии, построенные в Protégé OWL, могут быть экспортированы во множество форматов, включая RDF (RDF Schema), OWL и XML Schema. Protégé используется для решения задач, связанных со знаниями. Онтология задач вычислительной геометрии представлена на рис. 9.5.

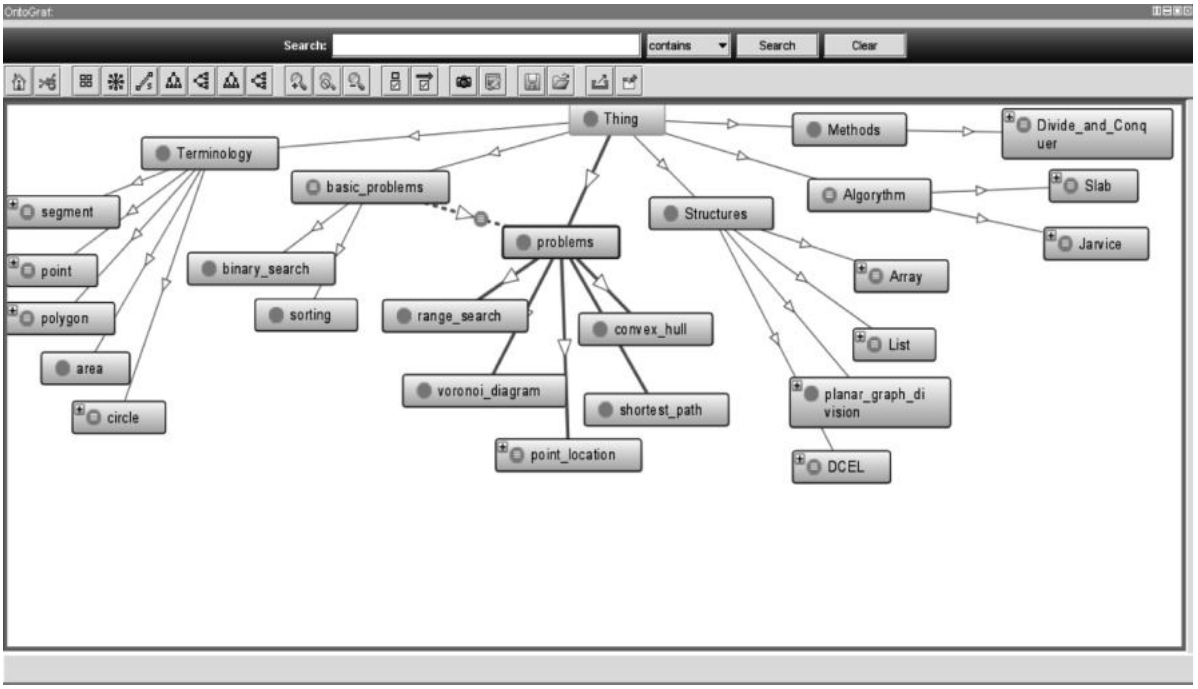


Рис. 9.5. **Схема онтологии «Вычислительная геометрия»**

9.3.4. Процессы домена ЖЦ ISO/IEC 12207

Объектом моделирования домена ЖЦ являются основные процессы стандарта ISO/IEC 12207 ЖЦ (требования, проектирование, конструи-

рование, тестирование и сопровождение), процессы поддержки и организации разработки (экспертиза, верификация, тестирование), сбор данных об ошибках и отказах, которые используются для оценки разных показателей качества ПП, в частности надежности.

Построение формальной модели домена выполняется методами онтологии путем аккумуляции знаний о моделях, ресурсах, доменах, которые созданы в некоторой операционной среде MS.Net.

Аспектом производства программ для домена является генерация, которая базируется на представлении знаний о специфике ПрО и накопленных знаний, а также на методах, средствах и инструментах SE, которые необходимы для линий автоматизированного изготовления отдельных ресурсов (компонентов) и ПС.

Процесс генерации рассматривается как последовательная трансформация промежуточных продуктов одного процесса ЖЦ в продукты следующего процесса путем использования систем программирования и других систем преобразования предметно ориентированных описаний членов ПС с применением КПИ.

При реализации ЖЦ стандарта ISO/IEC 12207 сформирована таблица процессов и подпроцессов ЖЦ (табл. 9.1).

Таблица 9.1

Процессы ЖЦ

№ п/п	
1. Категория «Основные процессы»	
1.1	Заказ (договор)
1.1.1	Подготовка заказа, выбор поставщика
1.1.2	Мониторинг деятельности поставщика, принятие потребителем
1.2	Поставка (приобретение)
1.3	Разработка
1.3.1	Выявление требований
1.3.2	Анализ требований к системе
1.3.3	Проектирование архитектуры системы
1.3.4	Анализ требований к системе
1.3.5	Проектирование системы
1.3.6	Конструирование (кодировка) системы
1.3.7	Интеграция системы
1.3.8	Тестирование системы
1.3.9	Системная интеграция
1.3.10	Системное тестирование
1.3.11	Инсталляция системы

Окончание табл. 9.1

№ п/п		
1.4	Эксплуатация	
	1.4.1	Функциональное применение
	1.4.2	Поддержка потребителя
1.5	Сопровождение	
	2. Категория «Процессы поддержки»	
2.1	Документирование	
2.2	Управление конфигурацией	
2.3	Обеспечение гарантии качества	
2.4	Верификация	
2.5	Валидация	
2.6	Общий обзор	
2.7	Аудит	
2.8	Решение проблем	
2.9	Обеспечение применимости продукта	
2.10	Оценивание продукта	
	3. Категория «Организационные процессы»	
3.1	Управление	
	3.1.1	Управление на уровне организации
	3.1.2	Управление проектом
	3.1.3	Управление качеством
	3.1.4	Управление риском
	3.1.5	Организационное обеспечение
	3.1.6	Измерение
	3.1.7	Управление знаниями
3.2	Усовершенствование	
	3.2.1	Внедрение процессов
	3.2.2	Оценивание процессов
	3.2.3	Усовершенствование процессов

Стандарт ISO/IEC 12207—2007 (табл. 9.2) включает в себя 17 процессов, 74 подпроцесса и 232 технологические операционные задачи (действия). Их необходимо и достаточно для проектирования систем с помощью процессного подхода. Некоторые системные фирмы поддержки реализуют фрагменты или варианты этого стандарта.

Процессы стандарта ЖЦ — это основные, вспомогательные процессы поддержки разработки ПС и организационные. Они представлены в виде

схем, удобных для онтологического описания. Основные процессы даны на рис. 9.6.

Таблица 9.2

Процессы, подпроцессы и задачи ЖЦ

Класс	Процесс	Действие	Задача
Основные процессы	5	35	135
Процессы поддержки	8	25	70
Организационные процессы	4	14	27
Всего	17	74	232

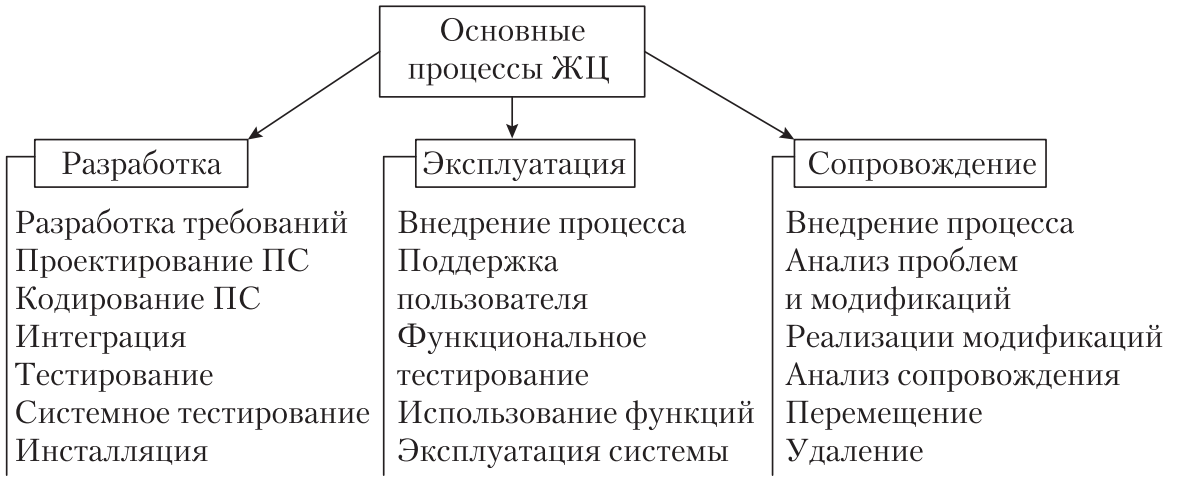


Рис. 9.6. **Схема основных процессов ЖЦ ПС**

Вспомогательные процессы регламентируют дополнительные действия по проверке продукта, управлению проектом и качеством (рис. 9.7).

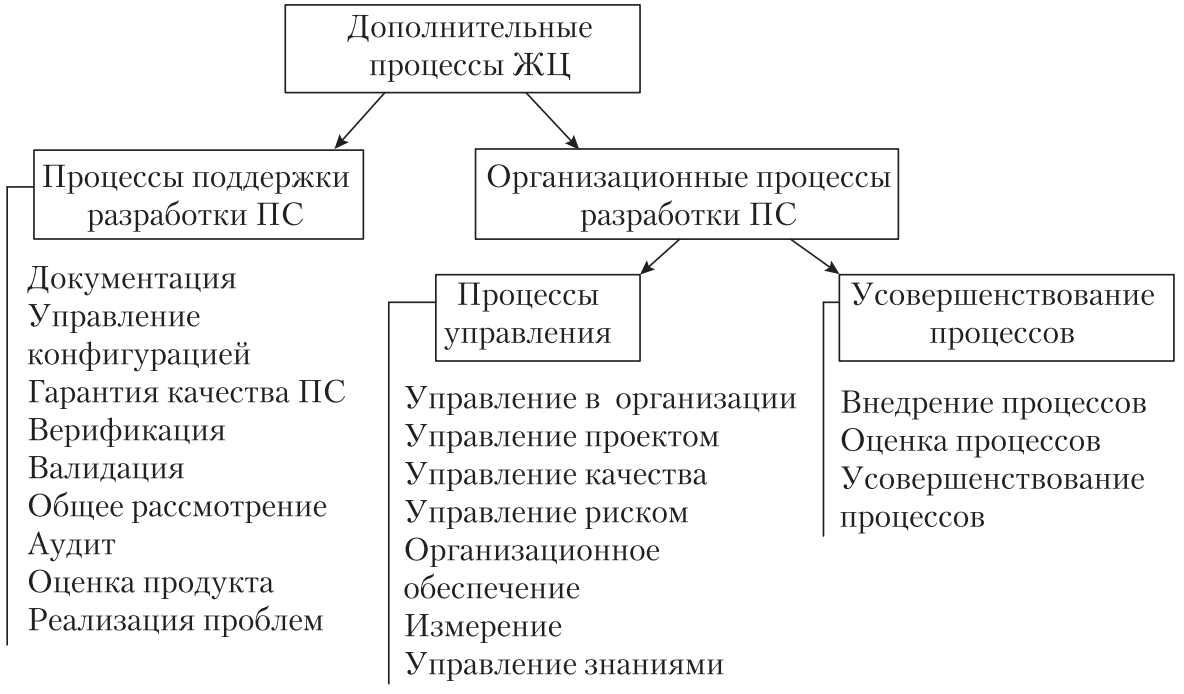


Рис. 9.7. **Схема вспомогательных процессов ЖЦ ПС**

Как правило, в зависимости от целей конкретного проекта на ПП главный разработчик и менеджер выбирают процессы, действия и задачи, выстраивают определенную схему ЖЦ для применения в конкретном программном проекте.

Описание семантики процессов, парадигм и методов их выполнения (объектные, компонентные, сервисные и др.) приведены в ядре знаний SWEBOK (www.swebok.com). В каждой технологии программирования сложных ПС с использованием стандарта ЖЦ применяются теоретические, прикладные методы, стандарты качества, общие и фундаментальные ТД (ISO/IEC 15404, ISO/IEC 9126, ISO/IEC 11404 GDT и др.), а также методики этих стандартов.

Задача автоматизации стандартного ЖЦ возникла при выполнении фундаментального проекта по ТП ИПС НАНУ (2007—2011) и создании комплекса ИТК. Была поставлена цель — автоматизировать ЖЦ и обеспечить генерацию разных его вариантов при изготовлении отдельных ПС из ГоР. Первый эксперимент по реализации ЖЦ проведен с участием студентов КНУ 4-го курса кафедр ИС, ТТП, МФТИ и двоих аспирантов. Участники разработки изучили современные онтологичные средства и средства визуального представления процессов ЖЦ — WWF (Windows Workflow Foundation), DSL Tools VS.Net, Protégé и др. На основе этих средств было реализовано описание онтологии ЖЦ в графическом и XML-видах в рамках систем DSL Tools VS.Net и Protégé. С вариантом онтологии ЖЦ автор настоящего издания выступала на международных конференциях TAAPSD'12—15, ICTERY — 2013 и «Science and Information — 2015» (Лондон). Идея автоматизации ЖЦ прозвучала впервые на этой конференции, и комитет IEEE в ее рамках предложил сделать патент на эту работу.

9.3.5. Формализация онтологической модели ЖЦ

На основе схем процессов ЖЦ и табл. 9.1 и 9.2 сделано формальное его описание, основанное на понятиях — процесс P (*Process*), операции A (*Action*) и задачи T (*Task*). Это описание имеет вид

$$M_{\text{ЖЦ}} = (P_k, A_м, T_n),$$

где $P_k = (P_{1k}, P_{2k1}, P_{3k2})$, $P_{1k,\ k=1-5}$ — основные процессы ЖЦ; $P_{2k1,\ k1=1-8}$ — дополнительные процессы ЖЦ; $P_{3k2,\ k2=4}$ — организационные процессы; $A_м = (A_{kr}, A_{k1l}, A_{k2j})$ — операции и задачи процесса.

В них задачи означают:

A_{kr}, $r = 1—35$ — операции основных процессов ЖЦ;

A_{k1l}, $l = 1—25$ — операции процессов поддержки ЖЦ;

A_{k2j}, $j = 1—14$ — операции организационных процессов ЖЦ;

$T_n = (T_{nk}, T_{nl}, T_{nj}) - T_{nk}$, $k = 1—135$ — задачи основных процессов ЖЦ;

T_{nl}, $l = 1—70$ — задачи процессов поддержки ЖЦ;

T_{nj}, $j = 1—27$ — задачи организационных процессов ЖЦ.

Описание содержания задач в стандарте не приведено. Их семантика задается при реализации формального описания.

Для представления структуры ЖЦ используется графический язык DSL, а для отображения семантики процессов ЖЦ могут использоваться ЯП (Java, C++, C# и др.), ориентированные на реализацию вычислительных действий программ. Язык DSL содержит общие абстракции для отображения классов объектов ПрО, типов процессов и действий, а также отношений между ними. Описание в этом языке сводится к языкам XML, WSDL и др.

Модель ПрО ЖЦ описана в языке DSL и может быть трансформирована к другой модели с более низким DSL. Это позволяет интегрировать между собой разные части процессов системы, написанные на разных языках DSL. Иными словами, ПрО ЖЦ может быть описана на одном уровне абстракции, а затем преобразована к языку более низкого уровня абстракции. Модель ПрО дополняется повторными компонентами и объектами и уточняется характеристиками, настроенными на процессы и операции (действия), которые есть в классе языков онтологии. На их основе формируется информация для конфигурационной структуры ПС с учетом аппаратных и программных ресурсов, необходимых при выполнении процессов ЖЦ.

Подход к автоматизации стандарта ЖЦ средствами онтологии является новым и оригинальным. В основе реализации лежит структура процессов ЖЦ и их взаимосвязи, а также подход к генерации отдельных вариантов автоматизации ЖЦ для конкретных применений.

Средствами представления процессов ЖЦ могут быть: языки OWL (*Web Ontology Language*), ODSD (*Ontology-Driven Software Development*), XML (*Extensible Markup Language*); действующие системы моделирования доменов — ODM (*Organizational Domain Modeling*), FODA (*Feature-Oriented Domain Analysis*), DSSA (*Domain-Specific Software Architectures*), DSL (*Domain Specific Language*), Eclipse-DSL Tools VS.Net, Protégé и т.п. Иными словами, имеются разнообразные языковые и технологические средства формального описания процессов ЖЦ для последующего автоматизированного моделирования разных ПП.

В настоящее время имеются новые средства — язык BPMN для описания процессов ЖЦ и язык DSL для описания семантики доменов. В качестве примера реализации процессов ЖЦ избран онтологический подход. В нем ЖЦ представляется с помощью словарей понятий, концептов и отношений между ними в среде Protégé, DSL Tool VS.Net и др. В них онтологическое описание трансформируется к языку XML, который является языком реализации размеченных данных домена ЖЦ, установленных связей и обменов данными между процессами.

Домен ЖЦ занимает центральное место в ПИ, основным назначением которой являются методы и средства изготовления ПС.

9.3.6. Описание процессов ЖЦ средствами DSL и Protégé

Для описания онтологии домена ЖЦ взят Eclipse DSL. В нем имеются средства разработки графических моделей ЖЦ. Описание основных процессов домена ЖЦ с помощью инструментария DSL Tools VS приведено на рис. 9.6. Типы отношений в данном графическом представлении задают

основную логику процессов домена ЖЦ. В каждом классе заданы методы и поля, необходимые для функционирования.

В самом начале описания процессов ЖЦ в DSL разрабатывается визуальная модель домена ЖЦ. Описание классов, отношений между ними дается имеющимися методами в среде Eclipse DSL и DSL Tools VS.Net. Графическое представление процессов ЖЦ дано на рис. 9.8—9.9.

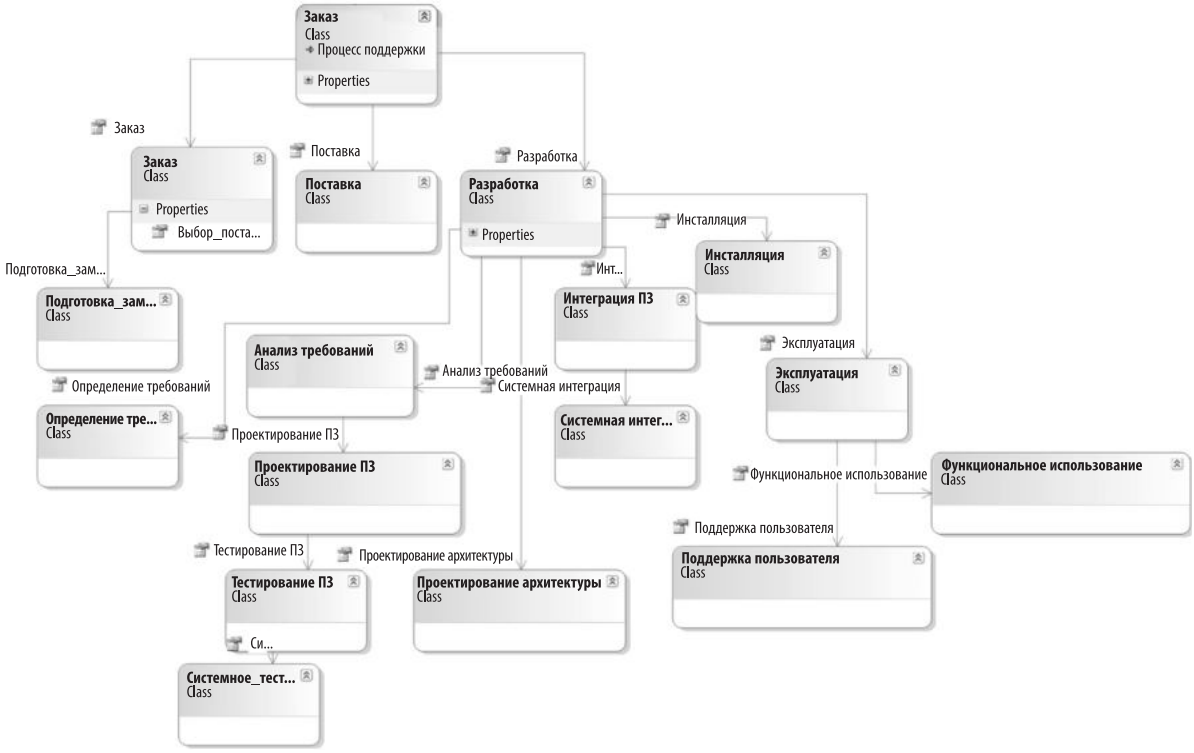


Рис. 9.8. **Графическое представление основных процессов ЖЦ**

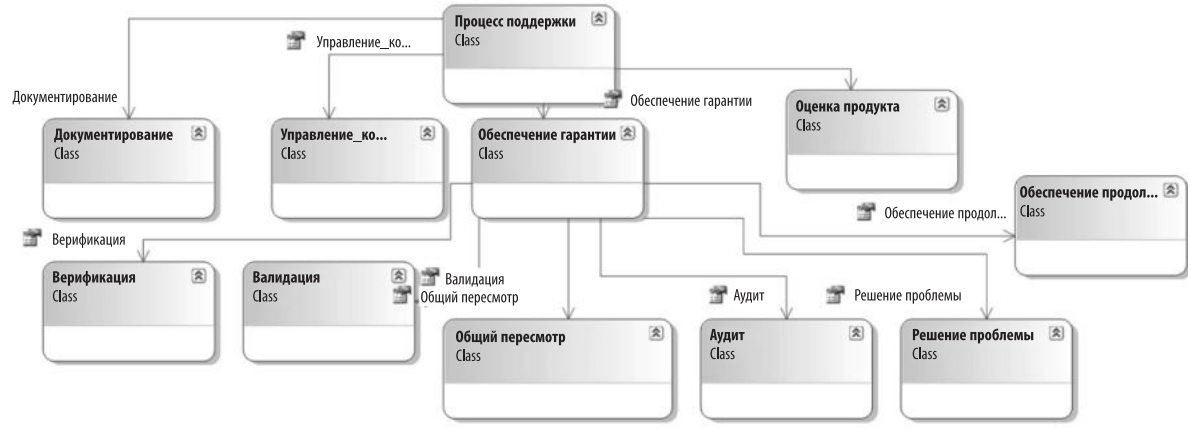


Рис. 9.9. **Графическое представление процессов поддержки ЖЦ**

Процессы поддержки включают в себя все процессы, которые выполняются после построения модели и проверки ее работоспособности. Онтологическая структура отражает структуру процессов ЖЦ и их представление в XML-языке средства DSL Tools VS.Net (для трех категорий процессов ЖЦ описание в языке XML занимает 15 страниц).

Пример фрагмента описания основных процессов ЖЦ в языке XML приведен ниже.

```
<?xml version="1.0" encoding="utf-8"?>
<AssociationLine Name ="Определение требований" Type="Main.Опре-
деление требований" ManuallyRouted
= "true" FixedFromPoint = "true" </AssociationLine>
<AssociationLine Name = "Интеграция_ПС" Type = "Main.Интегра-
ция_ПС"
<AssociationLine Name = "Инсталляция" Type = "Main.Инсталляция"
<AssociationLine Name = "Эксплуатация» Type = "Main.Эксплуатация"
<Property Name = "Интеграция_ПС" />
<Property Name = "Инсталляция" />
<Property Name = "Анализ_требований" />
<Property Name = "Эксплуатация" />…
```

Для процесса тестирования ЖЦ выполнено аннотирование его средствами Protégé. Такое описание отсутствует в практике программирования и ориентировано на проведение тестирования простых программ.

9.4. Средства Protégé для представления онтологии

Существует целый ряд инструментов поддержки онтологии. Наибольшую популярность приобрели Protégé и Ontolingva. Они относятся к бесплатным, расширяемым с помощью плагинов (plug-in) средствам и редакторам онтологии.

В редакторе Protégé используется фреймовая модель представления знаний, что позволяет приспосабливать его для редактирования моделей ПрО, описанных в разных форматах UML, XML, SHOE, DAML+OIL, RDF и RDFS и т.п.

Современные инструменты для разработки онтологий — Protégé, позволяют не только создавать онтологии доменов, но и находить в ней ошибки, подобно отладчикам в ЯП. Protégé может использоваться как среда быстрого прототипирования, в которой разработчики онтологии могут проверять семантические ограничения и генерировать интерфейсы пользователя.

Кроме средств поддержки разработки онтологии, Protégé имеет открытую архитектуру, что позволяет программистам встраивать в него программные компоненты с помощью плагинов. Эта возможность может использоваться как при разработке приложений для семантического веба, так и для их создания в парадигме генерирующего программирования.

Для Protégé существуют также готовые плагины. Например, Protégé OWL Plugin предоставляет отрытую среду, в которой можно выполнять код, подобный тестам JUnit. Еще одна возможность Protégé — генерация по онтологии программного кода классов для языка Java.

Преимущество Protégé состоит в простоте инструкций для описания онтологии и возможности преобразовывать описание в формат XML-схемы, который реализуется на платформе Eclipse в XML Schema Infoset Model (XSD). На этой платформе имеется библиотека функций и инструментов для манипулирования компонентами в XML (рис. 9.10).

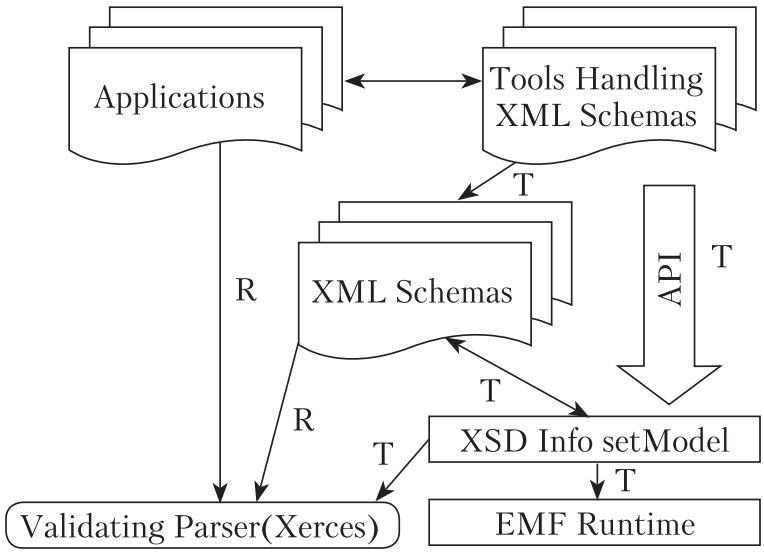


Рис. 9.10. **Структура XML Schema Infoset Model (XSD)**:
R — for application runtime; T — for execution of tool time

В качестве редактора онтологии используется Protégé 3.4. В нем онтология представляется классами, слотами, фасетами и аксиомами (см. параграф 9.2). Классы описывают понятие ПрО, а слоты — свойства. Фасеты описывают свойства слотов (конкретные типы и возможные диапазоны значений). Аксиомы определяют дополнительные ограничения (правила).

9.4.1. Разработка онтологии процесса тестирования

Предметом реализации онтологии является процесс тестирования. Рассматривается процесс тестирования с задачами и операциями. Главное назначение процесса тестирования ЖЦ — выполнение задач процесса на основе входов (входные данные для выполнения задач процесса) и выходов при завершении задач, а также ролей и действий исполнителей этих задач. Задачи тестирования рассмотрены и распределены по процессам ЖЦ ПС. Как результат, получен единый непрерывный процесс тестирования различных продуктов ПС, задачами которого являются подготовка, проведение и оценка результатов тестирования. Эти задачи распределились между 20 действиями (шагами) процесса разработки. Данный подход к углубленному тестированию ПС целесообразно применять, например, в системах реального времени.

На шаге подготовки осуществляется анализ рабочих продуктов процесса разработки ПС (входных для данного шага процесса тестирования) для определения целей, объектов, сценариев и ресурсов тестирования, адекватных тестированию. Проведение процесса тестирования должно фиксироваться в планах тестирования.

Концептуальная модель процесса тестирования ПС имеет вид:

$$SFT = \langle TM, TD, TA, Env \rangle,$$

где TM — подпроцесс управления тестированием; TD и TA — подпроцессы тестирования систем и приложений; Env — концептуальная и информационная среда процесса тестирования ПС.

Все подпроцессы имеют унифицированное формальное представление:

$$TM = \langle Task(TM, TD, TA), En(TM, TD, TA), CM(TM, TD, TA)\rangle,$$
$$Env(TM) \cup En(TD) \cup En(TA) = Env,$$

где *Task* — задачи соответствующего подпроцесса; *Env* — концептуальная и информационная среда соответствующего подпроцесса; *CM* — подмодель координации операций подпроцесса.

В состав среды *Env* входят следующие элементы:

$$Env = TG \cup SG \cup T \cup P \cup RG \cup RP,$$

где *TG* и *SG* — тесты и готовые программы; *T* и *P* — тесты и протестированное приложение; *RG* и *RP* — отчеты о выполнении тестовых программ и тестов.

Онтологическое описание процесса тесирования в системе Protégé включает классы, слоты, фасеты и аксиомы. Подобную возможность предоставляют и другие инструменты. Например, диаграммы классов UML системы Rational Rose отображаются в программный код на нескольких ЯП.

При представлении онтологии тестирования рассматривались простые и сложные понятия.

Простые понятия — понятия в ПрО, которые могут описываться в виде самостоятельного абстрактного (или конкретного) класса. В тестировании выделено шесть типов простых понятий:

- тестер (*Tester*);
- контекст (*Context*);
- действие (*Activity*);
- метод (*Method*);
- артефакт (*Artifact*);
- среда (*Environment*).

Простое понятие может также характеризоваться несколькими атрибутами. В качестве атрибутов выбираются подпонятия, которые характеризуют базовое (родительское) понятие и могут принимать конкретные значения.

Сложное понятие — понятие, которое является композицией простых понятий.

В онтологии тестирования такими сложными понятиями являются обязанности тестера и задача (*task*).

1. *Тестер*

Это понятие определяет субъекта, который выполняет тестирование. Тестером могут быть человек, ПС (инструменты тестирования), группа тестирования из тестеров. Для понятия «тестер» можно выделить следующие атрибуты: имя (TESTER NAME), тип (TESTER TYPE) и обязанности. Обязанность описывает смысл действий тестера.

Фрагмент XML-схемы понятия «тестер»:

```
<!-- TESTER -->
<xs:element name="TESTER">
<xs:complexType>
```

```
<xs:sequence>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="TESTER"/>
</xs:sequence>
<xs:attribute name="TESTER TYPE" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="HUMAN"/>
<xs:enumeration value="SOFTWARE"/>
<xs:enumeration value="GROUP"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="TESTER NAME" use="required"/>
<xs:attribute name="TESTER LEADER"/>
</xs:complexType>
</xs:element>
```

Пример. Фрагменты XML-документа, которые удовлетворяют схеме.

```
<TESTER TESTER TYPE=»HUMAN» TESTER NAME="Иванов" />
```

2. *Контекст*

Контекст тестирования определяет соответствующие уровни: методы тестирования, входы и выходы задач тестирования.

Для понятия контекста определяется один атрибут: Уровень_тестирования (CONTEXT TYPE).

Уровень_тестирования = {модульное, интеграционное, системное, регресионное}

Фрагмент XML-схемы этого понятия:

```
<!-- CONTEXT -->
<xs:element name="CONTEXT">
<xs:complexType>
<xs:attribute name="TEST_LEVEL " use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="UNIT_TEST"/>
<xs:enumeration value="INTEGRATION_TEST"/>
<xs:enumeration value="SYSTEM_TEST"/>
<xs:enumeration value="REGRESSION_TEST"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
```

3. *Действие*

Понятие «действие» может состоять из понятий: планирование тестирования, разработка (генерация) тестов, выполнение тестов, оценка результатов, измерение тестового покрытия, генерация отчетов и др.

Для него определяется один атрибут — Тип действия (ACTIVITY TYPE) с возможными значениями.

Тип_действия = {планирование, разработка тестов, выполнение тестов, проверка результатов, оценка покрытия, подготовка отчета}

Фрагмент XML-схемы этого понятия:

```
<!-- ACTIVITY -->
<xs:element name="ACTIVITY">
<xs:complexType>
<xs:attribute name="ACTIVITY_TYPE" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="TEST_PLANNING"/>
<xs:enumeration value="TEST_CASE_GENERATION"/>
<xs:enumeration value="TEST_CASE_EXECUTION"/>
<xs:enumeration value="TEST_RESULT_VERIFICATION"/>
<xs:enumeration value="TEST_COVERAGE_MEASUREMENT"/>
<xs:enumeration value="TEST_REPORT_GENERATION"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
```

4. *Метод*

В тестировании применяются методы структурного тестирования, основанные на ошибках, методы функционального тестирования, «белый ящик», основанный на коде тестирования (*program-based*), и «черный ящик», основанный на спецификации тестирования (*specification-based*). Методы, базирующиеся на коде, подразделяются на структурные, подсев ошибок, мутационные.

Структурные методы можно разделить на две группы: тестирование потока управления и тестирование потока данных. Методы тестирования потока управления (*control-flow methods*) включают покрытие операторов, покрытие ветвей и разные критерии покрытия. Для представления конкретного метода тестирования вводятся следующие атрибуты:

- имя (название метода), например «разбивка на категории»;
- тип метода;
- подход, базирующийся на коде и спецификациях.

Фрагмент XML-схемы этого понятия:

```
<!-- METHOD -->
<xs:element name="METHOD">
<xs:complexType>
<xs:attribute name="METHOD_NAME" use=«required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="CONTROL_FLOW_TESTING"/>
<xs:enumeration value="DATA_FLOW_TESTING"/>
<xs:enumeration value="STATEMENT_COVERAGE_TESTING"/>
<xs:enumeration value="BRANCH_COVERAGE_TESTING"/>
<xs:enumeration value="PATH_COVERAGE_TESTING"/>
<xs:enumeration value="NODE_COVERAGE_TESTING"/>
<xs:enumeration value="LINK_COVERAGE_TESTING"/>
```

```
<xs:enumeration value="LIP_COVERAGE_TESTING"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="METHOD_TECHNIQUE">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="STRUCTUAL_TESTING"/>
<xs:enumeration value="FAULT_BASED_TESTING"/>
<xs:enumeration value="ERROR_BASED_TESTING"/>
</xs:simpleType>
</xs:attribute> <xs:attribute name="METHOD_APPROACH">
```

5. *Артефакт*

Каждое действие из тестирования может включать несколько артефактов — результаты тестирования, планы, наборы тестов, скрипты и др. Эти артефакты часто называют тестовыми активами.

Объекты тестирования могут быть разных типов: начальный код, HTML- и XML-файлы, встроенные изображения, звук, видео, документы и др. Результаты тестирования включают: отчеты об ошибках, измерение тестового покрытия и др.

С этим понятием связаны следующие атрибуты: Тип, Формат, Размещение, Данные.

Фрагмент XML-схемы для описания этого понятия:

```
<!-- ARTEFACT -->
<xs:complexType name="ARTEFACT">
<xs:sequence>
<xs:element ref="ARTEFACT"/>
</xs:sequence>
</xs:complexType>
<xs:element name="ARTEFACT">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" ref="TESTER"/>
<xs:element minOccurs="0" ref="ARTEFACT_DATA"/>
<xs:element minOccurs="0" ref="ARTEFACT_LOCATION"/>
</xs:sequence>
<xs:attribute name="ARTEFACT_TYPE" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="OBJECT_UNDER_TEST"/>
<xs:enumeration value="TEST_PLAN"/>
<xs:enumeration value="TEST_SCRIPT"/>
<xs:enumeration value="TEST_RESULT"/>
<xs:enumeration value="TEST_SUITE"/>
<xs:enumeration value="TEST_COVERAGE"/>
<xs:enumeration value="ERROR_REPORT"/>
<xs:enumeration value="SPECIFICATION"/>
</xs:restriction>
</xs:simpleType>
```

```
</xs:attribute>
<xs:attribute name="ARTEFACT_FORMAT" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="PROGRAM"/>
<xs: enumeration value="HTML"/>
<xs:enumeration value="XML"/>
<xs:enumeration value="TEXT_FILE"/>
<xs:enumeration value="WORD_FILE"/>
<xs:enumeration value="PDF_FILE"/>
<xs:enumeration value="POSTSCRIPT_FILE"/>
<xs:enumeration value="BMP_IMAGE"/>
<xs:enumeration value="JPEG_IMAGE"/>
<xs:enumeration value="CD_SOUND"/>
<xs:enumeration value="MPEG_VIDEO"/>
<xs:enumeration value="VCD_VIDEO"/>
<xs:enumeration value="DVD_VIDEO"/>
<xs:enumeration value="JAVA_APPLET"/>
<xs:enumeration value="JAVA_SCRIPT"/>
<xs:enumeration value="NODE_SEQUENCES"/>
<xs:enumeration value="LINK_SEQUENCES"/>
<xs:enumeration value="LIP_SEQUENCES"/>
<xs:enumeration value="DATA"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="ARTEFACT_DATA" type="any"/>
<xs:element name="ARTEFACT_LOCATION" type="any"/>
```

6. *Среда*

Для программной среды можно указать: имя продукта, тип продукта и версию.

Понятие среды включает два понятия: аппаратное и ПО со своими атрибутами. Например, атрибуты для среды задаются так:

Среда = {ОС, БД, Компилятор, Веб-браузер}

Фрагмент XML-схемы для описания этого понятия:

```
<!-- ENVIROMENT -->
<xs:element name="ENVIRONMENT">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="HARDWARE"/>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="SOFTWARE"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="HARDWARE">
<xs:complexType>
```

```
<xs:attribute name="HARDWARE_DEVICE" use="required"/>
<xs:attribute name="HARDWARE_MANUFATURER" use="required"/>
<xs:attribute name="HARDWARE_MODEL" use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="SOFTWARE">
<xs:complexType>
<xs:attribute name="SOFTWARE_TYPE" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="OS"/>
<xs:enumeration value="DATABASE"/>
<xs:enumeration value="COMPILER"/>
<xs:enumeration value="WEB_SERVER"/>
<xs:enumeration value="WEB_BROWSER"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="SOFTWARE_PRODUCT" use="required"/>
<xs:attribute name="SOFTWARE_VERSION" use="required"/>
</xs:complexType>
```

9.4.2. Онтология тестирования в Protégé

Процесс тестирования задается последовательностью этапов.

Этап 1. Создание проекта и выбор формы. Выполняется из меню File командой New Project. После этого выбирается тип проекта и определяется формат сохранения файлов проекта (рис. 9.11) в формате Protégé.



Рис. 9.11. **Выбор типа проекта**

Этап 2. Разработка иерархии классов. В изображении иерархии классов используются слоты классов Tester и Envirinment, а классы наследуются от класса THING.

Этап 3. Описание слотов и фасетов. Слот задает атрибут класса, идентифицирует его имя и фасет (возможных типов значений). Окно создания слота и его описания изображены на рис. 9.12.

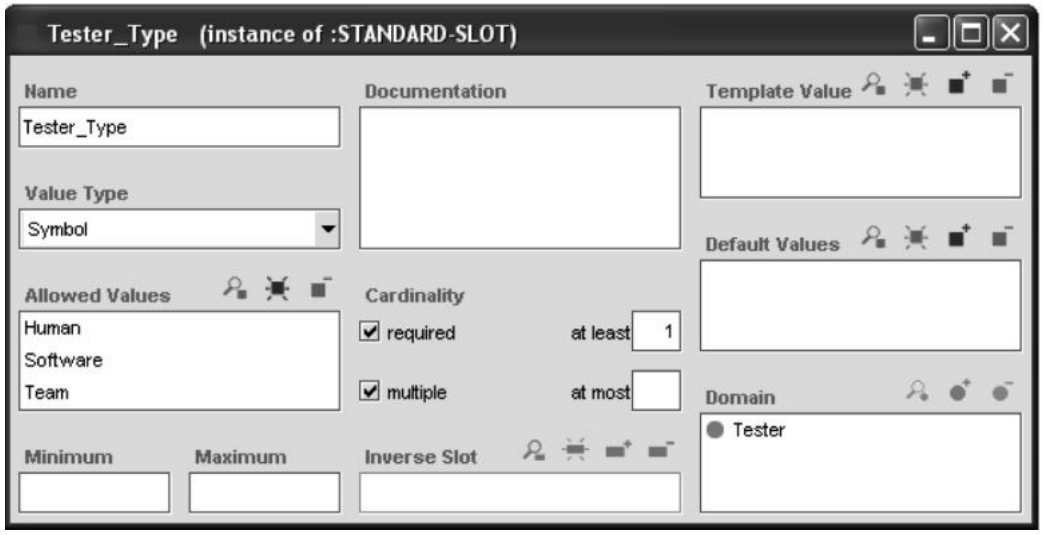


Рис. 9.12. **Создание нового слота**

Этап 4. Создание экземпляров классов. После создания онтологии в виде иерархии классов и их слотов можно создавать конкретные экземпляры классов. Для этого назначена вкладка Instance (рис. 9.13).

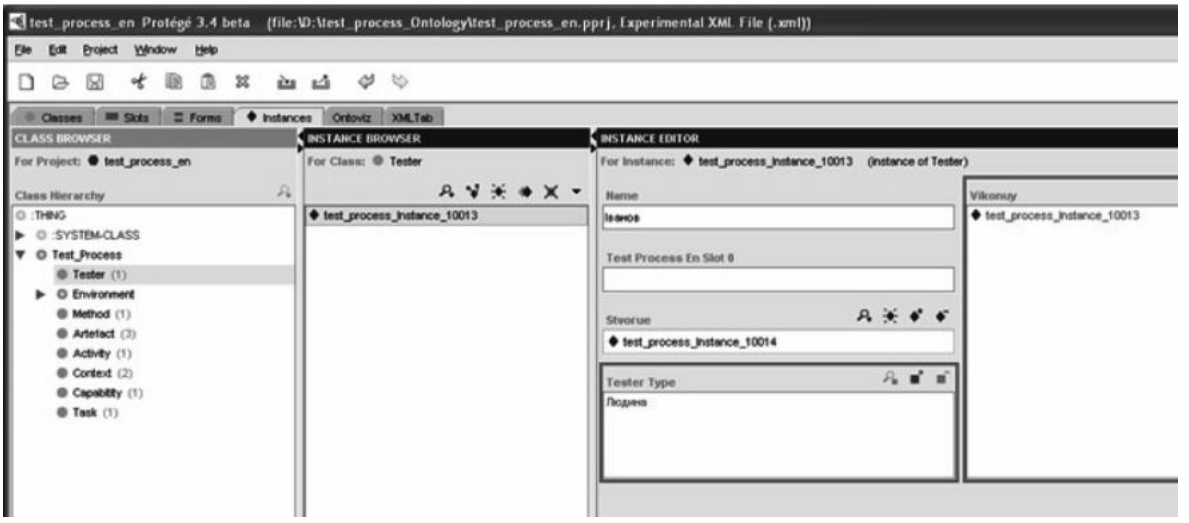


Рис. 9.13. **Создание экземпляра в редакторе экземпляров**

Таким образом, Protégé предоставляет средства описания структуры онтологии, ее редактирования и уточнения, наполнения значениями и сохранения в базе знаний (БЗ) ПрО онтологии и отдельных экземпляров классов со значениями слотов.

9.4.3. Плагины для графического представления онтологии

Для графического представления онтологии, разработанной в формате Protégé, может использоваться несколько плагинов для повторного использования отдельных понятий. При построении графа онтологии тестирования использовался плагин OntoViz.

На рис. 9.14 изображены классы онтологии тестирования (которые отображают простые и сложные понятия) и связи между ними. Этот плагин не поддерживает кириллицу, поэтому это понятие онтологии представляется латиницей.

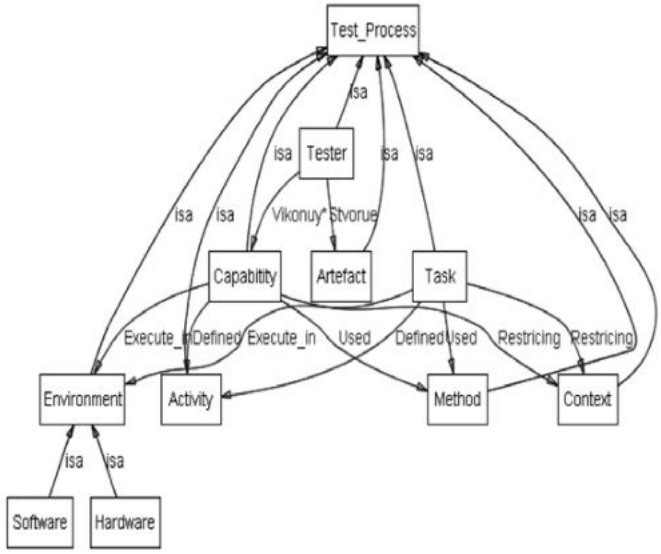


Рис. 9.14. **Графическое представление онтологии в плагине OntoViz**

Используя понятие онтологии, студенты КНУ на практических занятиях создали вариант онтологии процесса тестирования и реализации программы тестирования в языке Ruby (был также сделан доклад на конференции «Science and Information — 2015»). Он встроен в ИТК сайта http://sestudy.edu-ua.net, обращение к которому осуществляется нажатием на слово «Онтология» на главной панели данного сайта.

9.5. Онтология КПИ

КПИ и другие компоненты многоразового применения размещаются в разных хранилищах. Ими могут быть библиотеки, репозитарии компонентов и ресурсов в Интернете (например, GreedStone, Matlab, библиотека повторно используемых классов C++ и др.). Компонентами из этих хранилищ можно пользоваться многократно всем желающие для реализации своих целей, в том числе при построении программных систем различного назначения. Репозиторий типа библиотеки GreenStone и Matlab представляют огромное множество готовых программ научного и математического типов, т.е. они ориентированы на математиков, биологов, физиков и других специалистов ПрО.

В общем случае репозиторий — это система средств для хранения, пополнения наработанных КПИ, включает инфраструктуру разработки ПС из компонентов, организацию доступа к содержащимся в нем КПИ для последующего их применения в новых проектах.

С функциональной точки зрения репозиторий работает по принципу информационно-поисковой системы, объектами хранения которой являются разные типы документов, текстов и др. Система ставит в соответствие ключевым понятиям, словам, правилам доступа и др. запрос пользователя к требуемому ресурсу, который содержится в ее коллекции документов.

Репозиторий состоит из разделов, ориентированных на инженерию КПИ и на их сервисное обслуживание. В отличие от систем поиска информации в репозитории компонентов, кроме КПИ, размещается семантиче-

ская информация в виде поискового образа, созданного на основе описания информационной модели каждого компонента. Эта модель — средство построения поискового образа для каталога КПИ, ориентированного на осмысление человеком функций КПИ и возможности сопоставления с собственными потребностями.

Для отображения КПИ в репозитории проводятся их классификация и каталогизация, аннотирование и размещение.

Классификация компонентов ориентирована на унификацию представления информации о компонентах для последующего их поиска и отбора из среды репозитория. Она проводится с учетом следующих факторов:

- компоненты типа модуль, класс и т.п.;
- компоненты имеют интерфейс (входные и выходные параметры, условия выполнения), функциональность и реализацию на ЯП в виде шаблона развертывания;
- готовые к употреблению КПИ;
- сложные КПИ типа каркасов и паттернов, которые обеспечивают взаимодействие простых КПИ.

Каталогизация направлена на физическое размещение кодов КПИ в репозитарии для извлечения их при необходимости встраивания нового программного проекта ПС. Для выбранных компонентов выполняется их настройка на условия среды функционирования.

Построение онтологической системы «ОнтоКПИ»

Система обеспечивает информационную поддержку специфицированных образов КПИ в онтологической модели репозитария в целях их аттестации, ведения, поиска и принятия решения об использовании КПИ в новых ПС, а также практического конструирования онтологий различных доменов (например, графика, делопроизводство и др.)[1].

Система имеет три уровня:

1) определяет представление готовых КПИ и артефактов, полученных на соответствующих процессах ЖЦ разработки домена в HTML-формате;

2) отображает спецификации артефактов и КПИ первого уровня и задается поисковыми образами (ПОБ) в виде фасет;

3) база знаний, заданная совокупностью предметных поисковых образов знаний (ПОЗ) доменов средствами объектно ориентированных средств языка Java.

Основные функции системы «Онтоаспект» следующие:

- построение онтологий в виде графов с объектами в вершинах;
- аннотирование артефактов в терминах онтологий с помощью ключевых слов ПОБ, поисковых запросов к КПИ и артефактам;
- служба синонимии лексики домена, совместимой с лексическими понятиями других доменов;
- ведение словаря синонимов для концептов онтологии;

---

[1] *Бабенко Л. П., Поляничко С. Л.* Подходы к аттестации адаптивных возможностей программных компонентов // Проблемы программирования. 2001. № 1—2. С. 35—41.

• изменение или дополнение списка словаря онтологии с учетом синонимии;

• построение ПОБ и ПОЗ в соответствии с выбранной классификацией объектов онтологии (компоненты, КПИ, аспекты, фреймы и пр.).

Структура системы «ОнтоКПИ» представлена на рис. 9.15 и включает следующие компоненты:

• репозиторий КПИ;

• классификатор компонентов;

• база знаний фасет-аспектов (свойств) по каждому КПИ;

• словарь домена;

• хранилище кодового представления артефактов;

• панель управления и набор панелей для текущего выполнения операций системы, соответствующих функциям системы.

Взаимодействие пользователя с системой осуществляется через следующие операции:

• создание графического эскиза аспекта;

• сохранение файла аспекта в GML-нотации;

• отображение GML (*Graph Modelling Language*) нотации в графическом виде;

• сохранение нормированной лексики аспекта;

• редактирование аспекта в графическом режиме;

• удаление аспекта;

• копирование аспекта КПИ в другой домен.

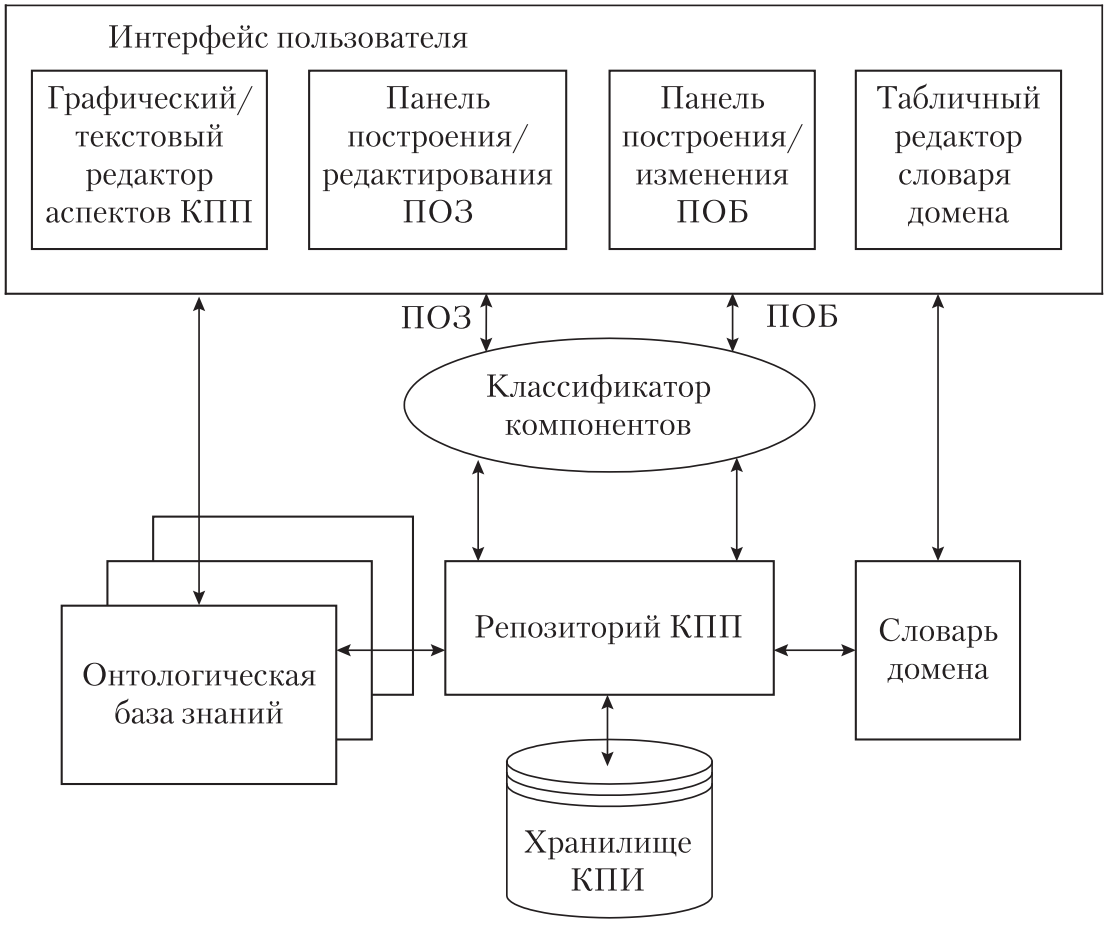


Рис. 9.15. **Архитектура системы «ОнтоКПИ»**

Среда реализации системы «ОнтоКПИ». Алгоритмы реализации функций системы выполнены в языке JAVA for Forte, Release 3.0 с использованием библиотеки визуализации графов Graphlet. Для моделирования онтологий используется язык GML, с помощью которого задается формат описания графов доменов в виде GML-файла, состоящего из списков пар «ключ — значение».

Общение пользователя с системой осуществляется через набор экранных форм — панелей, содержащих набор необходимых операций для выполнения заданной на панели функции.

Язык описания интерфейса компонентов

Для объединения компонентов в программную структуру необходимым условием является наличие для них формально определенных интерфейсов в языках IDL и APL, а также механизмов динамического контроля связей между компонентами.

Спецификация интерфейса в API и IDL включает описание функциональных свойств компонентов, их типов и порядка задания операций передачи аргументов и результатов для взаимодействия компонентов. Описание интерфейса представляет собой интерфейсный посредник между двумя объектами аналогично функции брокера ORB в системе CORBA.

ПС, построенная из компонентов КПИ, предназначена для функционирования в распределенной среде. Она состоит из двух частей, каждая из которых может выполняться на разных процессах и которые взаимодействуют друг с другом через вызов интерфейсных функций. Первая часть — серверная программа (сервер), а вторая — клиентская (клиент).

В функции интерфейсного модуля клиента входят:

- подготовка внешних данных клиента (параметров);
- набор вызовов процедур или обращение к сервису сервера;
- обработка разных ошибок, возврат данных от сервера к клиенту.

Общие функции интерфейсного модуля сервера содержат:

- ожидание сообщений клиента и их обработку; запуск удаленной процедуры и передачу ей параметров клиента;
- возврат результатов процедуры клиенту, уничтожение удаленной процедуры и др.;
- структура интерфейсного модуля не зависит от ЯП взаимодействующих объектов и в целом одинакова для всех. Это связано со стандартизированной его структурой и общим языком спецификации интерфейса IDL (*Interface Definitation Language*).

Готовые КПИ имеют сертификат (паспорт, описание функций, интерфейс и данные для обмена). У паспорта в языке WSDL вид стандартный.

Таким образом, механизмы и языки описания онтологии доменов ЖЦ, «Вычислительная геометрия» и КПИ, оказались понятными и доступными для работы со студентами. Требуются средства верификации полученных моделей ПрО, а также представление онтологий для оценки качества систем и продуктов.

Контрольные вопросы и задания

1. Расскажите про основы онтологии для описания доменов.
2. Назовите основные понятия онтологии для ПрО.
3. Какие существуют средства для проектирования доменов в DSL?
4. Какова структура описания ПрО в DSL?
5. Расскажите про трансформацию DSL описания моделей ПрО.
6. Назовите средства Protégé для представления онтологии.
7. В чем смысл разработки онтологии процесса тестирования?
8. Расскажите об онтология тестирования в Protégé.
9. Что такое плагины в онтологии?
10. Как работает онтология процессов ЖЦ?

Глава 10
МЕТОДЫ ЭКСПЕРТИРОВАНИЯ, ТЕСТИРОВАНИЯ И ОЦЕНКИ КАЧЕСТВА ПС

В процессе разработки нескольких прикладных систем специального назначения нами отработана технология проведения экспертизы отдельных КПИ, тестирования и оценки качества. После сборки КПИ в ПС проводилось комплексное тестирование собранной системы из КПИ и интерфейсов. Методы экспертирования, тестирования и оценки качества ПС защищены в трех кандидатских диссертациях и отдельные результаты представлены в монографии автора[1].

10.1. Экспертирование компонентов и систем

На основе всесторонней проверки разработки ПС сформировались методы экспертирования ПС и процессов их создания путем измерения и оценивания показателей качества компонентов и ПП в классе задач СОД:

1) оригинальная модель экспертизы ПС в процессах создания ПП;
2) тестирование ПС и сбор данных для метрического анализа;
3) модель качества с ориентацией на оценку надежности ПП;
4) модель распределения количественных требований по отдельным компонентам и с учетом показателей пользователя;
5) модель распределения надежности ПП из компонентов с функцией полезности ПП $Q_{\text{пс}} = \sum_{j=1}^{l} v_j^* \cdot q_j = \sum_{s=1}^{m} w_s^* \cdot r_s$ в зависимости от весомых коэффициентов w_s и надежности $q_j = \prod_{n \in E_j} r_n$ отдельных компонентов;
6) модель принятия решений по управлению качеством, включая методы контроля надежности, начиная с ранних стадий ЖЦ, измерения требований, тестирования ПС, оценки надежности с учетом интенсивности отказов и прогнозирования дефектов.

10.1.1. Модель экспертизы процессов

Названные модели экспертизы разработаны в рамках фундаментального проекта и диссертационных исследований в ИПС НАНУ. Результаты исследований изложены в технических отчетах проекта, статьях и дис-

[1] *Лаврищева Е. М., Слабоспитская О. А.* Подход к экспертному оцениванию в программной инженерии // Кибернетика и системный анализ. 2008. № 4. С.151—168; *Слабоспитская О. А.* Модели и методы экспертного оценивания в ЖЦ программных систем : автореф. Киев : Изд-во ИК НАНУ, 2008. С. 19.

сертациях, защищенных под руководством автора[1], и в ряде монографий (доступны на сайте www.twirpx.com) по методам их программирования и оценке качества ПС[2]. Модели адаптированы для семейства систем ПС и описаны в коллективной *e*-монографии[3]. Модели качества и оценки стоимости ПП отражены и реализованы в инструментальном комплексе ИТК[4].

Идея экспертизы ПС, определение ее модели и процесса управления рисками, основанного на дереве ценностей, метод группового экспертного оценивания компонентов и ПП базируются на общей экспертной теории применительно к процессам стандарта ЖЦ 12207, которая приведена на рис. 10.1.

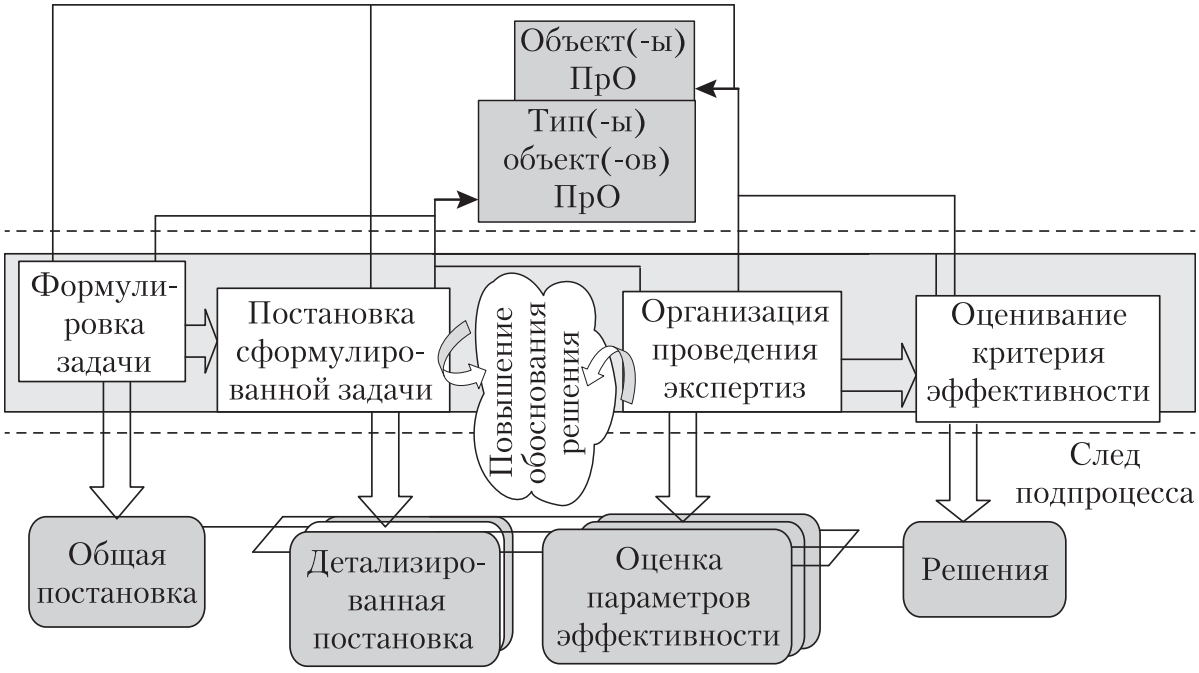


Рис. 10.1. **Схема процесса экспертизы ЖЦ**

Модель процесса экспертивно-аналитической оценки объектов ЖЦ имеет вид

$$PM = <EE, O, R, OM>,$$

где EE — информационная среда процесса; O — онтология знаний о предметной области; R — ретроспектива результатов работы процесса; OM — модель операций процесса.

---

[1] См., в частности: *Коваль Г. И.* Модели и методы инженерии качества программных систем на ранних стадиях ЖЦ. Киев : Изд-во ИК НАНУ, 2005. С. 20; *Коротун Т. М.* Модели и методы инженерии тестирования ПС в условиях ограниченных ресурсов. Киев : Изд-во ИК НАНУ, 2004. С. 21; *Слабоспитская О. А.* Модели и методы экспертного оценивания в ЖЦ программных систем. Киев : Изд-во ИК НАНУ, 2008. С. 19.

[2] См.: *Лаврищева Е. М., Коваль Г. И., Коротун Т. М.* Подход к управлению качеством программных систем // Кибернетика и системный анализ. 2008. № 5. С. 173—185; *Лаврищева Е. М., Коротун Т. М.* Построение процесса тестирования программных систем // Проблемы программирования. 2002. № 1—2. С. 272—281.

[3] *Лаврищева Е. М.* Новые теоретические основы технологии производства семейств программных систем в контексте генерирующего программирования : *e*-монография / Е. М. Лаврищева [и др.]. 2011. URL: http://www.nbuv.gov.ua/portal/natural/progr/lavrishcheva.pdf.

[4] *Коротун Т. М.* Модель определения критерия завершения тестирования на основе анализа риска отказа и ее реализация // Проблемы программирования. 2001. № 1—2. С. 63—68.

Подмодель *OM* неявно задает состав методов оценивания (*MF*) и актуализации (*MA*) для процедур технологии, обеспечивающей их полную поддержку, и модель интеграции методов.

Для использования опыта оценивания (механизм M_5) предложена специальная форма представления концептов онтологии *O*. Концепты онтологии *O* описываются в виде абстрактного класса фреймовой модели Н. Ноя посредством идентификации его типизированных связей с другими элементами ПрО (концептами и параметрами).

Экспертная оценка базируется на технологических нормативах, регламентах и результатах операций проектирования ПС в базовом процессе (*BP*), а также в процессе менеджмента *MP* (с участием заказчиков, менеджеров, оценщиков).

Производство ПС (*Production Action*, PA) можно представить кортежем:

$$PA = \langle P \in \{BP, MP\}\rangle;\ DA_{BP} = DC \cup DI \cup DE \cup DM;\\ DA_{MP} = DC \cup DM, \qquad (10.1)$$

где DA_{BP} — ПрО принятия решений по управлению *P*; *DC*, *DI*, *DE*, *DM* — подобласти ПрО *SE*, соответствующие ее выделенным дисциплинам: экономической (*Economics*), производственной (*Industry*), инженерной (*Engineering*) и дисциплине управления (*Management*).

Задача достижения качества ПС при производстве *PA* предполагает многократное оценивание в среде процессов *P* из выражения (10.1), т.е. делается установка текущих и прогнозных значений: критериев эффективности, уровней достижения для деловых процессов *P*.

Для критериев эффективности процесса *BP* сформированы следующие группы методов:

1) оценивания качества и трудоемкости проекта ПС;

2) специальные экспертно-аналитические для отдельных характеристик (функциональный объем и качество ПС, зрелость организации, мощность процесса ЖЦ и др.). Их оценка проводится экспертным или экспертно-аналитическим методом на процессах *P*;

3) специальный аналитический расчет показателей надежности и оптимальных параметров, полученных при тестировании ПС.

Кроме критериев эффективности, в процессах *P* возникает потребность в оценке двух групп характеристик. К первой группе относятся факторы влияния на критерии эффективности процессов $P \in \{BP, MP\}$. Вторая группа — характеристики, имеющие статус критериев эффективности в подходах к повышению качества ПС в методологиях разработки ПС, которые определяются требованиями к оценке объектов и характеристик.

10.1.2. Оценка процессов

Деятельность по оцениванию процессов *P* представлена системой унифицированных действий по оценке объектов, имеющих сходные характеристики и многократно контролируемые в ходе выполнения *P*. Каждое такое действие — установление значения характеристики управляемого (контролируемого) объекта *P*. Эта характеристика служит актуальным

критерием достижения целей усовершенствования и (или) управления P[1]. Она называется целевой, а объекты, для которых устанавливается ее значение, называются оцениваемыми объектами ПС (рис. 10.2).

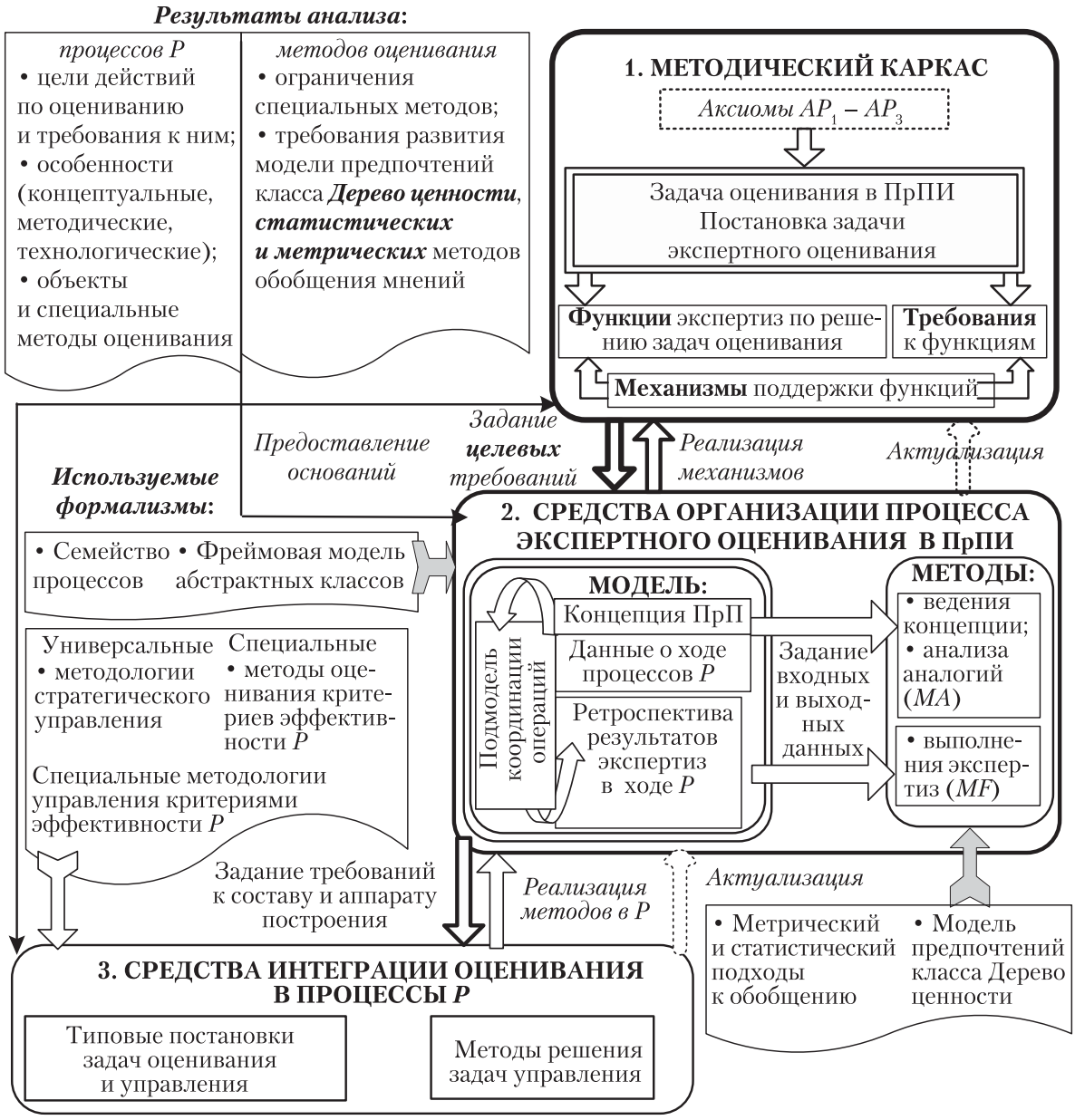


Рис. 10.2. **Методический комплекс экспертного оценивания**

К результатам анализа процессов P относятся: действия по оцениванию требований; операции (концептуальные, методические, технологические); объекты и специальные методы оценивания. К результатам анализа относятся: ограничения специальных методов; требования к модели предпочтений, дереву ценности, статистическим и метрическим методам обобщающих мнений экспертов.

---

[1] *Лаврищева Е. М., Слабоспитская О. А.* Подход к экспертному оцениванию в программной инженерии // Кибернетика и системный анализ. 2009. № 4. С.151–168; *Слобоспицкая О. А.* Модели и методы экспертного оценивания ПС. Киев : Изд-во ИК НАНУ, 2008; *Lavrischeva E. M., Slabospickaya O. A.* An approach to expert assessment in software engineering // Cybernetics and Systems Analyses. 2009. Vol. 45. № 4. P. 638–654.

Цели методик оценки процессов *P* следующие:

1) получение достоверной информации для выработки и обоснования решений по управлению ходом выполнения *P*;

2) усовершенствование процессов *P* в производственной деятельности *PA* для обоснования управленческих решений и условий эффективной деятельности участников;

3) предоставление формального аппарата оценивания с заданным качеством.

На основе специфики процессов *P* и среды оценивания определены многокритериальные древовидные модели, зависящие от характеристик моделей. Рассматриваются концептуальные и технологические особенности процессов *P*.

К *концептуальным особенностям* относятся потенциальные различия взглядов экспертов на характеристики и источники информации в *P*, а также на процессы разработки семейств на линиях продуктов, дублирования функций объектов *P* и др. К *технологическим особенностям* процессов *P* относятся повторяемость действий с одними и теми же объектами и их характеристиками; информационная преемственность и общие ресурсные ограничения действий; необходимость автономного и совместного оценивания объектов ПИ и др.

Результаты анализа использованы в составе математического аппарата экспертного оценивания задач в системе информатизации.

10.1.3. Задачи экспертной оценки объектов

Математический аппарат экспертного оценивания в процессах $P \in \{BP; MP\}$ предназначен для решения формализованных задач оценивания с помощью экспертиз объектов, взаимосвязанных в единую среду согласно потребностям *P*, их специфике и требованиям к действиям по оцениванию (см. рис. 10.2). Данный подход включает в себя:

1) определение задачи оценивания объектов ПС, конструктивное для достижения целей действий по оцениванию путем адекватной интеграции специальных и автоматических методов в экспертизах;

2) формирование структуры математического аппарата решения задач в виде системы моделей и методов их оценивания;

3) математическую и программную реализацию элементов;

4) введение разработанных моделей, методов и программных средств в процессы *P* и их апробацию в организациях.

В **задачу методического аппарата** входит сопоставление оценок аксиомы AP_1—AP_3 и конструктов, обеспечивающих удовлетворение потребностей процессов *P* после проведения экспертиз процесса оценивания. Они включают в себя: требования к функциям, агрегирующие потребности выбранных методов и требования методических источников; функции оценочных действий по решению задач в ходе *P*; механизмы реализации функций в экспертизах.

Средства организации процесса оценивания обеспечивают выполнение функций с помощью введенных механизмов. Модель процесса оценивания

задает направления развития выбранных многокритериальных методов для обеспечения механизмов и требования к входным и выходным данным.

Средства интеграции оценивания в процессы P — типовые постановки задач оценивания и управления, а также методы их решения и оценивания процессов позволяют использовать результаты оценивания для удовлетворения потребностей заказчика продукта.

Накопление опыта оценивания осуществляется за счет целевых требований к элементам ПС.

10.1.4. Технология экспертного оценивания объектов ПрО

Суть разработанной технологии (интегрирующего механизма M_1) — сопоставление задач оценивания множества унифицированных *подпроцессов оценивания*[1]. Такой подпроцесс реализует отдельное действие по оцениванию для фиксированных альтернатив A, формируя обоснованные оценки $\{ch(a), a \in A\}$ при фиксации остальных элементов постановки как начальных данных (см. рис. 10.1).

Процесс оценивания представлен пополняемой системой подпроцессов. Подпроцессы формируют систему экспертиз, информационно взаимосвязанных для действий по оцениванию: многократно контролируемые в процессах P; связанных аналогиями и (или) отношениями классификации. Первоначально решается одна и та же задача оценивания при разных начальных условиях, а затем другие с использованием целевых характеристик.

Для подпроцесса оценивания определены четыре этапа последовательного решения задачи экспертного оценивания, методы их выполнения и результаты. Для получения промежуточных результатов первых трех этапов (общей и детализированной постановки, а также оценки подхарактеристики) и окончательного результата четвертого этапа (для решения задачи предназначены процедуры (PG, PP, PE, PS) и методы *оценивания* ($MF = MG \cup MP \cup ME \cup MS$)). Для обеспечения целостности среды оценивания процедура *пересмотра* ПС предусмотрена вне подпроцессов на этапах пересмотра.

Описанная унифицированная структуризация результатов подпроцесса отображает все элементы постановки задачи экспертного оценивания. При этом результаты очередного этапа служат входными данными для следующего. Вследствие этого образуется иерархия результатов этапов, названная *следом подпроцесса*. Именно она обеспечивает возможность повторного использования результатов процесса оценивания всеми его участниками на всех этапах подпроцессов.

Таким образом, метод оценивания проводится на процессе ЖЦ с общей информационной средой, адекватной потребностям и специфике производственной деятельности, связанной с изготовлением ПП.

---

[1] См.: *Литвак Б. Г.* Экспертные оценки и принятие решений. М. : Патент, 1996; *Слабоспитская О. А.* Экспертиза процессов жизненного цикла программных систем: особенности организации и проведения // Проблемы программирования. 2006. № 2—3. Спец. выпуск. С. 341—348.

10.2. Подходы к процессу тестирования ПС и СПС

Тестирование ПС является важной задачей в процессе создания ПП[1]. В качестве ключевых вопросов тестирования рассматриваются те, которые изложены в SWEBOK.

1. *Критерии выбора тестов / Критерии адекватности тестов* (или правила прекращения тестирования). Эти критерии могут применяться как для создания набора тестов, так и для проверки, насколько выбранные тесты адекватны решаемым задачам (тестирования), а также помогают определить, когда можно или необходимо прекратить тестирование.

2. *Эффективность тестирования / Цели тестирования*. Тестирование — это наблюдение за поведением программы, выполняемой в целях тестирования с заданными параметрами, по заданному сценарию или с другими заданными начальными условиями или целями тестирования. Эффективность теста может быть определена только в контексте заданных условий.

3. *Тестирование для выявления дефектов*. В тестировании для выявления дефектов применяется деструктивный подход, а успешным считается тест, обнаруживающий дефект. Этот подход принципиально отличается от другого подхода, когда тесты запускаются для демонстрации того, что программа удовлетворяет предъявляемым к ней требованиями и, соответственно, тест считается успешным, если не найдено дефектов.

4. *Проблема оракула*. «Оракул» в тестировании — это любой агент (человек или программа), оценивающий поведение программы и формирующий заключение о результате теста (тест пройден или нет). Это заключение существенно зависит от трактовки понятий «отказ» и «дефект» в конкретном контексте.

5. *Теоретические и практические ограничения тестирования*. Ограничения обусловлены невозможностью исчерпывающего тестирования и его экономической нецелесообразностью для конкретных ПС. Тестирование должно проводиться с учетом риска отказа ПС и основываться на таких стратегиях тестирования, получивших распространение в современных методологиях разработки, как тестирование, основанное на риске (*risk-based testing*), и управляемое риском тестирование (*risk-driven testing*), которые кратко рассмотрены в этой главе.

6. *Проблема неосуществимых путей*. Неосуществимые пути — это пути потока управления программы, которые не могут быть выполнены ни при каких входных параметрах. Это сложнейшая проблема путевого тестирования и особенно его автоматизации. Основные методы путевого тестирования рассмотрены в параграфе 10.3.

7. *Тестопригодность*. Этот термин имеет два разных значения. Первое — это степень легкости описания критериев тестового покрытия для ПС, вто-

---

[1] См.: *Коротун Т. М.* Модели и методы инженерии тестирования пограммных систем в условиях ограниченных ресурсов. Киев : Изд-во ИК НАНУ, 2008; *Лаврищева Е. М., Коротун Т. М.* Построение процесса тестирования программных систем. 2002. № 1—2. С. 272—281.

рое — вероятность того, что выполнение программы при тестировании приведет к ее отказу, при наличии дефекта.

Тестирование ПС и СПС, составленное из модулей и компонентов, состоит в обеспечении следующих критериев: выполнение программы не менее одного раза на совокупности тестов, включающих в себя наборы входных и выходных данных; тестирование функций; тестирование межмодульных интерфейсов.

Вначале выполняется автономное тестирование отдельных модулей ПС нижнего уровня без вызова других модулей, затем выбирается очередной модуль, непосредственно вызывающий уже проверенные. Выполняется сборка модулей, а затем их тестирование в комплексе. Затем проводится тестирование интерфейсов, т.е. проверка правильности вызовов модулей и передачи экспортируемых и импортируемых значений параметров. Многие тесты, которые использовались при тестировании отдельных компонентов, могут использоваться при тестировании интерфейсов.

При тестировании объединенных модулей могут быть найдены ошибки в интерфейсных посредниках. Завершающим процессом тестирования является проверка функций и их комплексные испытания.

Методы тестирования построены по одной из четырех **стратегий моделирования процесса тестирования** СПС:

1) *тестирование продуктов СПС* на уровне инженерии приложений. Повышение качества достигается путем использования КПИ, количество дефектов уменьшается в результате многократного использования в разных операционных профилях;

2) *инкрементное тестирование СПС*. По этой стратегии продукт поддается тестированию по полному циклу тестирования *V*-модели. Все следующие продукты тестируются посредством регрессионных тестов;

3) интенсивная *разработка тестов* для КПИ в парадигме СПС. Вариантность в продуктах и потребности их тестирования закладывается в шаблоны артефактов и абстрактных тестовых ситуаций. На стадии инженерии приложений происходит процесс тестирования по *V*-модели. Конкретные тесты GoP используются как абстрактные — дополняются или уточняются для тестирования всего продукта;

4) распределение обязанностей *тестирования на уровнях V-модели* между разными организационными подгруппами. Одна подгруппа тестирует общие компоненты с помощью автономных тестов. Другая подгруппа выполняет интеграционное, системное и приемочное тестирование интегрированного продукта.

Тестирование «от требований» (*Requirements-based Testing*). Дж. МакГрегор (McGregor) разработал процесс тестирования ресурсов и приложений СПС, который базируется *на требованиях*. Этот процесс описывает действия по управлению тестированием, стратегией тестировании «от требований», и тесты спецификации интегрируются в процесс разработки СПС. Он также включает некоторые критерии качества тестов для СПС: трассировка, модифицируемость и профиль конфигурации.

Трассировка определяет, насколько легко трассировать тесты ресурсов в разработанном продукте и с учетом требований.

Модифицируемость показывает, насколько легко можно изменять тесты ГоР при выполнении итеративных изменений в СПС или продуктах, полученных в СПС.

Профиль конфигурации определяет, насколько легко тесты ГоР могут конфигурироваться для отображения разных комбинаций вариантов для разных приложений. Для расширения конфигурации Мак-Грегор рекомендовал проектирование процедуры тестирования, пригодной для использования на стадии инженерии ПрО и инженерии приложений, и использование одного и того механизма реализации вариабельности в продуктах и в тестах для улучшения трассировки между тестами и ПС. Т. Киши (Kishi) для систематического повторного использования тестов предложил использовать *диаграмму характеристик ПрО* и ассоциировать характеристики с необязательными переходами в машинах состояний, а затем создать повторно используемую модель верификации, которая может конфигурироваться для верификации проекта ПС, выведенной в СПС. Эта модель включает МХ, диаграмму компонента и машины состояний. Необязательные характеристики в МХ ассоциируются с необязательными компонентами, переходами и машинами состояний. Представление в виде машины состояний создается для каждого актора и компонента, а для верификации моделей может использоваться чекером (*model checker*), который работает с выражениями линейной темпоральной логики (*Linear Temporal Logic*, LTL) на этом представлении[1].

Тестирование «от сценариев использования» (*Use-case Based Testing*). Соответственно этому подходу тесты строятся на базе сценариев использования ПС, функциональных моделей, диаграмм активности и т.п., разработанных согласно описаниям этих сценариев. Подход предусматривает систематическое повторное использование спецификаций тестирования для всех продуктов СПС. Он определил отображение в методах тестирования, предложенных А. Бертолино (Bertolino) (метод Product Line Use case Test Optimization (PLUTO)), С. Нибутом (Nebut), Е. Кемстайсом (Kamsties), А. Реусом (Reuys).

Индустриальная технология надежного тестирования UniTESK (http://unitesk.ru) разработана в ИСП РАН и обеспечивает процесс тестирования ПО, начиная со сбора и анализа требований и до его сопровождения. Технология сформирована исходя из опыта тестирования реальных промышленных проектов. В ней красной нитью сквозь все элементы проходит прослеживаемость требований от этапа анализа потребностей пользователей до выпуска конечного продукта. Технология представлена на конференции Tenth Workshop on Model-Based Testing, ETAPS 2015. April 18, 2015. Сайт используется многими специалистами.

10.2.1. Тестирование видов систем в SWEBOK

Руководствуясь рекомендациями SWEBOK 2004 г., мы далее кратко охарактеризуем следующие методы:

---

[1] См.: *Андон Ф. И.* Основы инженерии качества программных систем / Ф. И. Андон [и др.]. Киев : Академпериодика, 2007.

- тестирование объектно ориентированных программ;
- компонентное тестирование;
- тестирование веб-приложений;
- тестирование графического интерфейса пользователя;
- тестирование протоколов на соответствие;
- тестирование систем реального времени;
- тестирование критических систем.

Особенности типов ПС обусловливают не столько применение специальных методов проектирования тестов, сколько выбор методов и видов тестирования, наиболее подходящих для определенных *объектов тестирования*.

Хотя каждый из перечисленных выше типов ПС заслуживает тщательного рассмотрения, здесь дадим им лишь краткую характеристику.

Тестирование объектно ориентированных программ (ООП). Специфика тестирования ООП связана в первую очередь:

- с итерационным инкрементным подходом к разработке, что приводит к усложнению интеграционного и регрессионного тестирования;
- событийно-управляемой природой ОПП, что требует тестирования не только структуры программы, но и ее поведения.

На модульном уровне выполняется автономное тестирование классов: их структуры и методов (функций-членов), — это традиционные методы, основанные на коде.

На уровне интеграции классов тестируются иерархия наследования и взаимодействие между объектами. Должны быть проверены все возможные внешние вызовы методов класса — как непосредственные обращения, так и вызовы, инициированные получением сообщений. В качестве формальных моделей выступают модель классов ПО и модели каждого класса.

На уровне системы применяется тестирование по сценариям использования (*use-cases*) — начиная с анализа требований и проектирования ПС.

Основные методы (подходы) тестирования ООП перечислены в табл. 10.1.

Таблица 10.1

Основные методы тестирования ООП

Задача тестирования	Методы/подходы
Разработка тестовых наборов (test-case)	Тестирование, основанное на ошибках (шаблонах ошибок) Тестирование, основанное на сценариях использования
Тестирование внутренней структуры класса	Стохастическое тестирование классов Эквивалентное разбиение на уровне класса Таблицы решений
Тестирование взаимодействия классов	Стохастическое тестирование классов Эквивалентное разбиение на уровне классов Тестирование переходов между состояниями

Парадигма **компонентного программирования** основана на концепциях повторного использования и распределенной разработки.

Компонентное тестирование проходит на нескольких уровнях:

- тестирование компонента в изоляции (автономное);
- тестирование в процессе сборки;
- системное тестирование.

Тестирование компонента в изоляции выполняется при создании нового компонента. Помимо традиционных методов тестирования корректности реализации, тестируются также возможности повторного использования компонента при разных конфигурациях сред.

В наибольшей мере специфика компонентного тестирования проявляется в процессе сборки компонентов. Например, возможны проблемы несовместимости разных компонентов (или их версий). Помимо этого, требуется тестирование связующего кода, который, собственно, и формирует ПС.

На стадии **системного тестирования** применяются обычные методы функционального тестирования. Специфика их применения для данного типа ПС проявляется в сложности поиска причин ошибок, особенно если ошибка, обнаруженная в одном компоненте, связана с другим компонентом. В этой связи повышается важность контрактных интерфейсов (*contract*), которые задают спецификацию входов и выходов компонента и дают возможность проверки корректности данных.

Особенности **тестирования веб-приложений** связаны в первую очередь с тремя основными факторами: моделью ЖЦ, спецификой объектов (архитектурой приложения) и важностью тестирования технических характеристик качества (безопасность, конфигурация, производительность и др.).

Тестирование протоколов. Бурное развитие сетевых технологий, усложнение их архитектуры, взаимодействие нескольких разнородных сетей многократно увеличивают число интерфейсов, подлежащих тестированию.

При **тестировании систем реального времени** система *всегда* представляется в виде формальной модели. Поэтому на модульном уровне применяются формальные методы, основанные на формальной спецификации. Одним из методов функционального тестирования, специально предназначенных для систем этого класса, является тестирование переходов между состояниями (*state transition testing*). Обязательно также применение методов статистического тестирования.

Тестирование критических систем включает не только динамические, но и статические методы верификации, такие как *анализ критичности*, *анализ деревьев событий* и *деревьев отказов* (кратко рассмотренные далее в этой главе), а также *инспекции* документов и исходного кода. Для минимизации стоимости тестирования и риска отказа в ходе анализа ПС выделяются критические модули, требующие углубленного тестирования. Для них применяются наиболее строгие критерии покрытия кода.

10.2.2. Задачи процесса управления тестированием

В процессе управления тестированием решаются следующие задачи.

1. *Определение процесса и его совершенствования.* Тщательное планирование с учетом потребностей распределенной работы команд с привлечением их участников и актуализация всех задач процесса тестирования.

2. *Анализ вариабельности СПС и распределение обязанностей по тестированию*. Распределение обязанностей по тестирования имеет целью избежать перекрытия и пробелов при охватывании тестами ресурсов и продуктов. Основаниями для распределения могут служить вариабельность в продуктах СПС, уровни *V*-модели и технические механизмы реализации вариабельности.

3. *Предвидение и распределение тестирования по функциональным приростам* (*инкрементам*). Эти приросты в функциональности могут быть тщательным образом протестированы, когда они используются впервые во время инженерии приложений или через использование эталонных продуктов в ходе инженерии ПС. План должен распределять тестирование приростов между циклами тестирования продукта и ПрО.

4. *Определение сферы применения КПИ тестовых ГоР*. Повторно используемые тесты ресурсов нуждаются в определенных границах для их ресурсно-эффективного сопровождения. Анализ установленной сферы применения СПС позволит создавать тесты ГоР в необходимом и достаточном количестве.

5. *Мониторинг прогресса*. Тестирование СПС является процессом, который охватывает много организационных подразделов и поэтому нуждается в постоянном наблюдении за соблюдением тестовых планов.

6. *Сопровождение КПИ тестовых ресурсов*. Это важная задача, поскольку любое тестирование, выполненное в ходе инженерии приложений, может быть значимым в масштабе всего СПС, но при этом является результатом инженерии ПрО.

10.2.3. Инженерия процесса тестирования ПрО

Инженерия процесса тестирования ПрО включает реализацию следующих задач (рис. 10.3)[1].

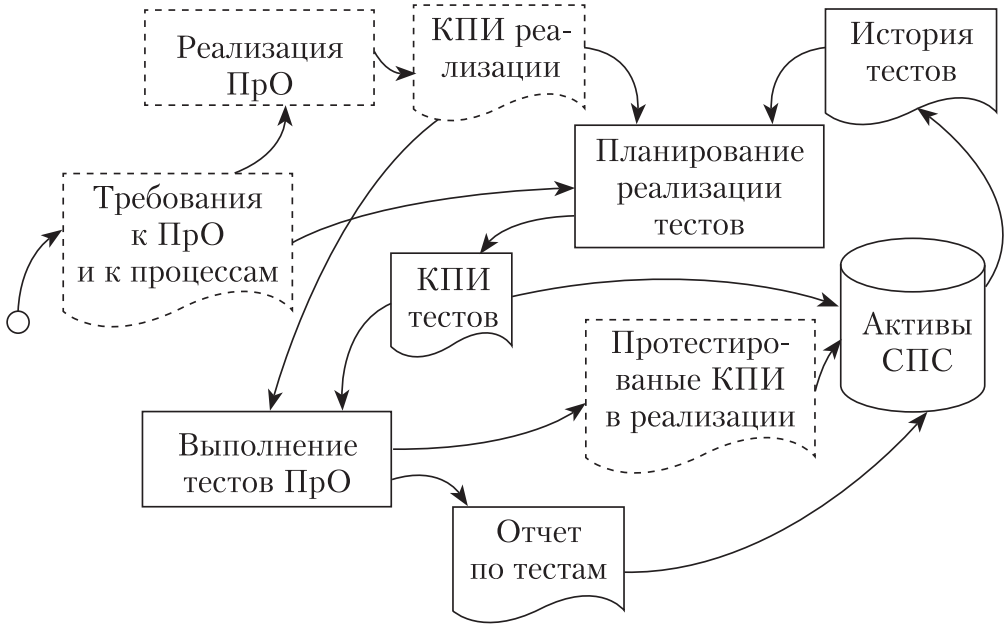


Рис. 10.3. **Структура инженерии процесса тестирования**

[1] *Коротун Т. М.* Модели и методы инженерии тестирования ПС : автореф. дис. Киев : Изд-во ИК НАНУ, 2004.

1. Проектирование и реализация тестовых ресурсов *для тестирования структуры и функциональности СПС*. В ходе инженерии ПрО создается структура для частичной реализации функциональности, которая входит в СПС. Проектирование и реализация тестовых ресурсов для тестирования общего ПО происходят до тех пор, пока остаются открытые точки вариантности для включения специфических для продукта тестовых ресурсов.

2. Выполнение теста с *использованием ресурсов* в наборе конфигураций СПС. Повторно используемые ресурсы тестируются отдельно, что минимизирует объем тестирования приложения. Необходимо выполнить имеющиеся тесты ситуации, созданные на минимизированном наборе конфигураций. Конфигурации должны покрывать предусмотренную структуру будущих продуктов, используя, например, комбинаторные подходы или операционные профили. Кроме того, здесь могут быть использованы прошлые конфигураций продукта как системы отладки для новых ресурсов.

3. Выполнение тестов произведенных приростов *в репрезентативном наборе эталонных продуктов СПС*. Новая функциональность может быть протестирована с использованием эталонных и рабочих продуктов. Нужно проанализировать результаты тестирования для решения того, какая новая функциональность нуждается в тестировании, и затем ее тестировать.

4. Анализ теста и *сопровождения GoP*. Качество повторно используемых тестовых GoP определяется при конкретном тестировании. Нужно проанализировать результаты тестирования, выполненного в ходе инженерии ПрО и инженерии приложений; проанализировать покрытие, полезность ресурсов и уровень многократного использования. Необходимо обслуживание ресурсов на протяжении всего времени для облегчения адекватного тестирования, управляемого моделью, и лучшей поддержки тестирования приложений.

Взаимосвязи между видами деятельности, выполняемыми в процессе тестирования ПрО, составляют базис построения модели координации операций этого процесса — *CM*(*TD*).

10.2.4. Модели тестирования СПС

К моделям тестирования относятся UML-профиль и *W*-модель тестирования[1].

UML-профиль тестирования

В основу построения модели тестирования СПС может быть положен *UML-профиль тестирования* (UML-ПТ), предложенный компанией OMG, который определяет язык для проектирования, визуализации, спецификации, анализа, конструирования и документирования артефактов тестовых систем. Профиль позволяет строить и повторно использовать проектные документы тестирования из самых ранних стадий ЖЦ системы и поддерживает основанное на UML тестирование, управляемое моделью (UML based model-driven testing).

---

[1] См.: *Андон Ф. И.* Основы инженерии качества программных систем. 2007.

Концепты UML-ПТ структурированы по четырем группам:

1) *архитектура теста* — определяет понятия, касающиеся структуры и конфигурации теста;

2) *поведение теста* — определяет понятия, связанные с динамическими аспектами тестовых процедур;

3) *данные теста* — определяет понятия, касающиеся тестовых данных в процедурах тестирования (тестовых процедурах);

4) *время тестирования* — определяет квантификацию времени при выполнении тестовых процедур.

Архитектура теста (*test architecture*). Для спецификации структурных аспектов системы тестирования введены такие понятия:

- *тестовый контекст* (*test context*), который позволяет пользователям группировать тестовые ситуации (*test cases*) для описания соответствующей тестовой конфигурации и определять диспетчера теста (*test control*) (для управления тестированием и контроля выполнения теста), т.е. необходимый порядок выполнения тестовых ситуаций;
- *тестированная система* (*System Under Test, SUT*) — один или больше объектов в пределах тестовой спецификации;
- *тест-компоненты* (*test components*) — объекты в системе тестирования (тестовой системы), которые могут «общаться» с SUT или другими компонентами для реализации работы теста (тестового поведения);
- *планировщик* (*scheduler*) — контролирует выполнение теста и тест-компоненты. Он отвечает за создание тестовых компонентов, синхронизированный запуск работы теста над разными тестовыми компонентами и выявление факта окончания тестовой ситуации;
- *средства* (*means*) *оценивания результатов тестирования*, полученных разными объектами в пределах системы тестирования, предназначенные для выноса окончательного вердикта относительно тестовой ситуации или тестового контекста. Этот процесс оценивания называют арбитражем (*arbitration*). Схема арбитража может формироваться с привлечением арбитра.

Поведение теста (события теста, *test behaviour*). Группа понятий для спецификации действий и актов оценивания, необходимых для определения задачи теста (*test objective*). Задача теста описывает цель его выполнения. Для описания тестовых стимулов, наблюдений отзывов от SUT, тестовых запусков, координации действий могут быть использованы диаграммы UML — диаграммы взаимодействия (*interaction diagrams*), машины состояний и диаграммы активности. Дальше определяются нормативы ожидаемого поведения (событий). UML-ПТ вводит также понятие для обработки непредвиденного поведения теста. Обработка непредвиденных сообщений достигается через спецификацию *типичных значений* (по умолчанию).

В добавление к таким понятиям, как задача теста, тестовая ситуация, вердикт, типичные значения, в профиле вводятся другие понятия, необходимые для спецификации теста, вроде:

- *действие с валидацией* (*validation action*) — может быть выполнено локальным тестовым компонентом;

• *журнал тестирования* (*test log*) — вместе с журналом действий (*log action*) является средством регистрации записей (*entry*) во время выполнения теста для последующего анализа;

• *действие по окончании* (*finish action*) — для определения завершения компонентом действия (поведения) в тестовой ситуации без его остановки;

• *оператор determAlt* — используется во взаимодействиях для спецификации детерминированного и последовательного оценивания предохранителей (*guards*) альтернативного поведения. Оператор determAlt всегда выбирает альтернативу, которая первой приобретает статус реализованной.

Данные теста (*test data*). UML-ПТ поддерживает заполнители (*wildcards*), динамические области данных (*data pools*), распределения данных (*data partitions*), селектора данных (*data selectors*) и правила кодировки (*coding rules*). Wildcards очень полезны для обработки непредвиденных событий или событий, связанных со многими разными значениями. Профиль вводит wildcards, которые позволяют проводить спецификацию любого значения (*any value*) за пределами набора возможных значений и любого или опущенного (попранного) значения (*any or omitted values*) для определения любого значения или отсутствия значения. Динамические области данных, распределения данных и селектора данных поддерживают повторное выполнение тестовых ситуаций с разными значениями данных для стимулирования SUT различными путями. Динамические области данных связываются с тестовыми контекстами и могут включать распределения (классы эквивалентности) данных и конкретные значения данных. Селекторы данных предоставляют разные стратегии для выделения и проверки данных. Профиль также поддерживает идею правил кодировки, что позволяет пользователю делать кодирование и декодирование тестовых данных при общении с SUT.

Время тестирования (*test time*). Новые понятия, которые пополняют простые понятия времени, определенные UML 2.0:

• *таймеры* (*Timers*) — для манипулирования и контроля тестового поведения и гарантирования окончания тестовых ситуаций;

• *зоны времени* (*Time zones*) — для группировки компонентов в пределах распределенной системы, позволяя таким образом сравнивать события в пределах той же зоны времени.

Введенные понятия предоставляют возможности для построения точных спецификаций тестов, используя UML 2.0. Профиль тестирования включает как структурные, так и динамические элементы и предоставляет ключевые понятия в области тестирования, которое делает тестовую спецификацию эффективной и действенной.

W-модель тестирования

В инженерии СПС выделяется два трека тестирования (рис. 10.4). Первый представлен средней ветвью *W*-модели и содержит общие для всех продуктов СПС эталонные элементы моделей тестирования. Они являются абстрактными и преимущественно адресованы к тестированию платформы, если СПС работает на компонентной платформе. При этом на этой платформе могут содержаться скелетоны (*skeleton* — связник компоненов

на сервере по CORBA) для тестирования общих функций и интерфейсов всех продуктов СПС.

Правая ветвь *W*-модели отвечает второму треку тестирования, который включает тесты ситуации, необходимые всем продуктам СПС, приспосабливает их для применения в конкретном ПП, а также добавляет новые тесты при выполнении реального текущего продукта. На этой ветви содержатся детальные тесты и их результаты.

Принципы перехода от тестирования СПС к тестированию продукта на стадии инженерии приложений должны отвечать принципам инженерии, управляемой моделями (MDE) и архитектуры MDA.

Предложенная цель-модель тестирования является концептуальным фреймворком, способным решать проблемы вариабельности в тестировании и построенным на общих принципах профиля тестирования UML. *W*-модель включает некоторые элементы, уже доступные в профиле: *Test Case*, *Test Class* (который называется еще *Test Suite*) и понятие *Test Data*. Предлагается отображение *Test Class* на диаграммы перехода состояний (*State Transition Diagrams*), а также связь с моделью требований (*Requirements model*).

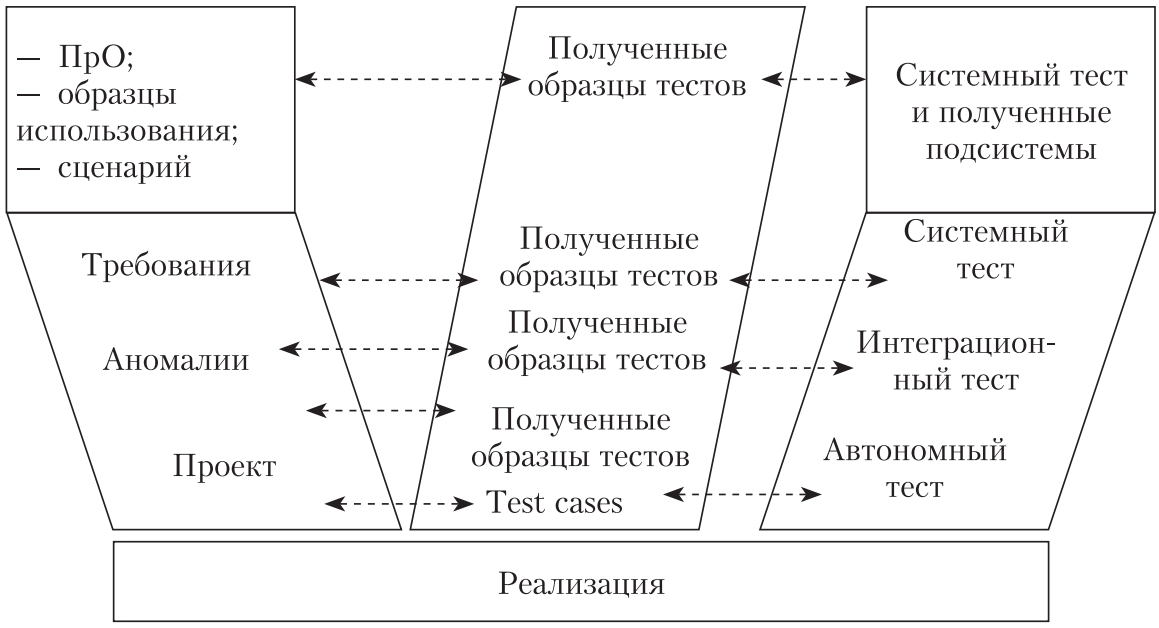


Рис. 10.4. **Виды тестов в *W*-модели**

10.2.5. Задача определения оптимального времени тестирования ПС

Эта задача выполняется с учетом типичных областей рисков и уровней угроз при отказах ПС, экономического взноса отказов модулей в убыток пользователей и стратегий тестирования.

Модель определения оптимального времени тестирования модулей основывается на исследовании областей рисков и уровней угроз, через отказы ПС, экономического взноса отказов модулей в убытки пользователей и стратегий тестирования. Под *угрозой* понимается событие при работе ПС, которое может привести к нанесению убытков пользователям (финан-

совых потерь или потерь времени), а под *риском* возникновения угрозы понимается величина убытков, обусловленных угрозами (аппаратные сбои, дефекты и т.п.) и теми из них, которые вызваны отказами ПС при недостаточном тестировании. Модель устанавливает критерий завершения тестирования до тех пор, пока разница между уменьшением риска и полной стоимостью тестирования не достигнет наибольшей величины.

Задачи оптимального времени тестирования определяются на следующих данных:

$H = \{H_i, i = 1, 2, ..., r\}$ — множество всех потенциальных угроз ПС, обусловленных отказами и дефектами в модулях;

$S = \{S_i, i = 1, 2, ..., n\}$ — множество всех возможных сценариев функционирования ПС;

t_e — время тестирования;

t_0 — время выполнения модуля в период эксплуатации ПС;

$C_м$ — взнос модуля в риск отказов ПС;

$R(t_0) = C_м\mu(t_0)$ — величина риска отказа модуля за время t_0;

$\mu(t)$ — функция роста надежности;

$\lambda(t) = d\mu(t)/dt$ — интенсивность отказов модуля;

$P(S_i)$ — вероятность того, что при реализации сценария S_i ($i = 1, 2, ..., n$) будет выполняться данный модуль;

$P(H_j | S_i)$ — условная вероятность того, что при реализации сценария S_i причиной возникновения угрозы H_j будет отказ именно данного модуля;

C_j — стоимость последствий реализации угрозы H_j ($j = 1, 2, ..., r$);

$C(t_e)$ — полная стоимость тестирования в течение времени t_e;

c_1 — стоимость единицы времени тестирования;

c_2 — стоимость устранения дефекта, который привел к отказу в процессе тестирования;

$\Delta R(t_0 | t_e)$ — функция снижения риска отказа модуля за время его выполнения t_0, при условии, что модуль тестировался время t_e;

$K(t_0 | t_e)$ — прибыль за время эксплуатации ПС, при условии, что модуль тестировался время t_e.

Необходимо найти такое время тестирования t_{e^*}, чтобы прибыль была максимальной:

$$t_{e^*} = \{t_e: \mathrm{K}(t_0 | t_{e^*}) \geq K(t_0 | t_e), 0 \leq t_e \leq t_{доп}\},$$

где $t_{доп}$ — максимально допустимое время тестирования.

Оптимальный час тестирования определяется по формуле

$$\Delta R(t_0 | t_e) = C_м[\mu(t_0) - \mu(t_0 + t_e) + \mu(t_e)],$$

где $C_м = \sum_{i=1}^{n}\left[P(S_i)\sum_{j=1}^{r}P(H_j | S_i)\cdot C_j\right]x_0$ — взнос модуля в риск ПС.

Прибыль от тестирования находится по формуле

$$K(t_0 | t_e) = \Delta R(t_0 | t_e) - C(t_e) = C_м[\mu(t_0) - \mu(t_0 + t_e) + \mu(t_e)] - c_1 t_e - c_2\mu(t_e).$$

Производная функции $K(t_0 | t_e)$ по t_e равняется

$$K'(t_0 | t_e) = C_м[\lambda(t_e) - \lambda(t_0 + t_e)] - c_1 - c_2\lambda(t_e).$$

Значение оптимального времени t_{e*} получается как решение уравнения $K'(t_0 | t_e) = 0$ в зависимости от конкретной функции $\lambda(t)$.

Метод оценивания риска отказов модулей заключается в последовательном решении следующих задач:

1) идентификации возможных угроз ПС и оценки стоимости последствий угроз C_j, $j = 1, 2, ..., r$;

2) взноса каждого модуля в угрозы для ПС путем связи возможных отказов модулей с внешними угрозами;

3) оценки ожидаемого количества отказов модуля за время t_0 посредством функции $\mu(t_0)$ модели надежности;

4) вычисления риска отказов для каждого модуля $R(t_0) = C_м\mu(t_0)$ с учетом неравнозначности компонентов и величины риска их отказов в ПС.

Реализация задачи оптимального времени проводится с помощью *базового процесса* тестирования, включающего в себя подготовку, проведение и оценивание результатов тестирования. Он базируется на стандартах разных документов, метрик процесса, а также методах для решения задач тестирования. Результаты тестирования документируются в специальных шаблонах документов.

10.3. Оценивание качества ПС и СПС

В условиях быстрых темпов развития индустрии программ и роста конкуренции, разработанных ПС имеется проблема оформления сертификата на ПП, включающего в себя оценки характеристик качества конечного продукта. Мониторинг качества по *количественным* показателям в ходе всего периода разработки ПП начинается с ранних этапов ЖЦ. Последняя версия международного стандарта «Процессы ЖЦ ПО» (ISO/IEC 12207:2007) включает в себя процесс «*управление качеством*» наряду с процессами верификации, тестирования и обеспечения гарантий качества. Для эффективного выполнения разработаны модели и методы инженерии качества, обеспечивающие поддержку принятия решений по управлению качеством на всех этапах ЖЦ ПС.

Существенный вклад в развитие этого направления внесли американские специалисты (О. Авен, Гантер, Б. Боем, Дж. Браун, Р. Гласс, Г. Майерс и др.) и советские ученые (В. В. Липаев, Е. С. Бутаков, А. Ф. Кулаков и др.).

В работах В. В. Липаева[1] предложена целостная *система управления качеством*, основанная на эталонной модели качества, совокупности организационных методов управления процессом проектирования комплексов программ (КП) на этапах ЖЦ и на методике учета технических показателей в ходе ЖЦ для последующего их использования при окончательной оценке полученных показателей качества в соответствии с требованиями к ним.

Качество КП определяется как совокупность технических, технологических и эксплуатационных характеристик КП, которые входят

[1] См., в частности: *Липаев В. В.* Методы обеспечения качества крупномасштабных программных систем. М. : СИНТЕГ, 2003.

в состав эталонной модели качества, представленной на верхнем уровне шестью базовыми характеристиками, значения которых могут быть определены количественно или оценены качественно. Этот подход применяется на этапах ЖЦ КП с контролем качества службой качества и устранением возникающих угроз качеству в связи с обнаруженными дефектами в КП. В задачи службы качества входят планирование и слежение за процессом достижения качества на этапах ЖЦ, квалификационное тестирование, испытание пробной версии КП и оценка базовых показателей качества этой версии на основе собранных технических данных КП.

В работах А. Ф. Кулакова предложен комплекс средств оценки качества больших программ ЭВМ, включающий в себя набор свойств и показателей качества программных изделий, а также методы их оценки. К ним относятся статистические методы анализа программ, основные функции тестирования и испытания программ. Основную роль сыграли введенные им функциональные и эксплуатационные показатели качества, определены задачи контроля и оценки качества при статистических испытаниях на основе моделирования, планирования и оценки уровня завершенности испытаний продукта или изделия. По его мнению, качество ПП находится в прямой зависимости от объема затрат на его разработку и сопровождение, т.е. 75% затрат на этапах ЖЦ идет на разработку продукта и его сопровождение. Под сопровождением понимается процесс модификации программ, обусловленных необходимостью устранения выявленных ошибок ПП и изменения его функциональности. Согласно положениям стандарта ГОСТ 15467—1989 уровень качества конкретной продукции включал в себя задачи выбора номенклатуры показателей качества, методов определения их значений и сопоставления полученных значений с базовыми. Он положил начало созданию научно обоснованных показателей качества и методов их определения, ориентированных на степень завершенности ПП.

Проект нового стандарта качества (ГОСТ 2844–1994) был разработан под руководством А. Ф. Кулакова и В. В. Липаева и специалистов из стран СЭВ. Проект обсуждался на рабочей группе СЭВ в марте 1987 г. в Дрездене. После развала СССР появился близкий к этому стандарт качества ISO/IEC 9126.

Анализ зарубежных стандартов качества серии 9000, 9126 проекта СЭВ отражен в коллективной монографии «Основы инженерии качества программных систем». По результатам многолетних исследований проблематики качества сформулирован комплекс задач инженерии управления качеством и защищен ряд диссертаций.

В результате был разработан комплекс базовых задач и инструментов инженерии качества (см. рис. 10.4), включающий в себя следующие из них:

1) *модели управления качеством*, начиная с ранних стадий ЖЦ ПС на основе модели процесса принятия решений по управлению качеством ПС, моделей задания требований к качеству компонентов ПС, прогнозирования плотности дефектов с помощью байесовских сетей и систематического количественного контроля качества ПС;

2) *модель процесса тестирования* ПС, которая способствует улучшению планирования тестирования, выделению необходимых ресурсов тестирова-

ния и обеспечивает управление тестированием с учетом рисков отказов ПС и рисков срыва выполнения проектов систем ПС в условиях ограниченных ресурсов;

3) *совершенствование моделей процессов* ЖЦ путем экспертно-аналитического оценивания процессов, выполняемых в каждом цикле управления, начиная с этапа прогнозирования целевой характеристики качества и ее достижения на процессах разработки ПС;

4) *модель интегрированной технологии* управления рисками программных проектов с использованием аппарата экспертного оценивания деревьями ценности и байесовскими сетями.

Новый подход к управлению и оценке качества базируется на процессах оценивания усовершенствованных процессов ЖЦ, ориентированных на создание моделей процессов ЖЦ с учетом требований к ПС и принятия решений по прогнозированию и управлению процессом тестирования для учета риска отказов ПС.

В основу моделей прогнозирования качества ПО и диагностики недостатков в применяемых процессах ЖЦ ПС положен теоретический аппарат — *байесовская сеть*. Этот подход к управлению качеством ПП ориентирован на достижение *завершенности* ПС.

Это соответствует получению характеристик качества и обеспечению надежной безотказной работы всех программных компонентов ПС. В нем акцент переносится на *процессы* ЖЦ, с помощью которых проводятся *выявление и устранение дефектов* в ПС.

В основе подхода лежит теория прогнозирования уровня завершенности ПС на процессах ЖЦ путем установления взаимосвязи внутренних и внешних метрик качества и совершенствования методов прогнозирования дефектов с учетом возникающих угроз и рисков по отношению к качеству продуктов.

10.3.1. Моделирование характеристик качества ПС

Качество, по определению стандарта ГОСТ 2844—1994, — это «совокупность свойств ПС, обеспечивающих способность удовлетворять *установленным* или *предполагаемым* требованиям в соответствии с назначением». Понятие «свойства продукта» ассоциируется с качеством ПС стандартной модели и особенностями ПС. В качестве ПС взята СОД, состоящая из нескольких компонентных приложений, работающих с большими объемами информации в БД, не критическими по отношению к безопасности функционирования[1].

Ключевыми характеристиками качества ПС выбраны надежность и завершенность как свойство системы исключать отказы в случае скрытых дефектов, и моделируются путем улучшения *управляемости* ПС исходя из критерия и *модели качества*, устанавливающей взаимосвязь мер и метрик внутреннего, внешнего и эксплуатационного типов (рис. 10.5).

---

[1] См.: *Коваль Г. М.* Модели и методы инженерии качества ПС : автореф. Киев : Изд-во ИК НАНУ, 2005; *Лаврищева Е. М.* Новые теоретические основы технологии производства семейств программных систем в контексте генерирующего программирования / Е. М. Лаврищева [и др.]. 2001. URL: http:// www. Nbou.ua/portal/ natural/progs/Lavrischeva.

На начальных этапах разработки ПС задаются спецификации значения (область значений) *внешних метрик*, которые служат критерием достижения установленного уровня качества при испытании ПС, определяют наиболее пригодные (с точки зрения прогноза значений внешних метрик) *внутренние метрики* и планируют поэтапное достижение внешних требований к качеству. Внешние требования к качеству ПС устанавливаются с позиций *эксплуатационного качества* ПП.

С позиций *завершенности* ПС главным показателем ее внутреннего качества являются дефекты, внешнего — отказы, а эксплуатационного — обобщенный взгляд пользователей на работу ПС. Это соответствует трехуровневой модели качества, устанавливающей взаимосвязь внутренних, внешних и эксплуатационных мер качества:

1) внутренняя мера D_0 — количество (плотность) дефектов в каждом компоненте ПС;

2) внешняя мера $R(t)$ — безотказность функционирования каждого компонента ПС в течение заданного времени t, т.е. вероятность того, что за время t эксплуатации компонента не возникнет набор входных данных в сценарии его использования, что приведет к отказу;

3) мера эксплуатационного качества $Q_{ПС}$ связана с безотказным функционированием ПС, соответствует характеристике качества при эксплуатации ПС и обозначает *удовлетворенность* («satisfaction») ПС.

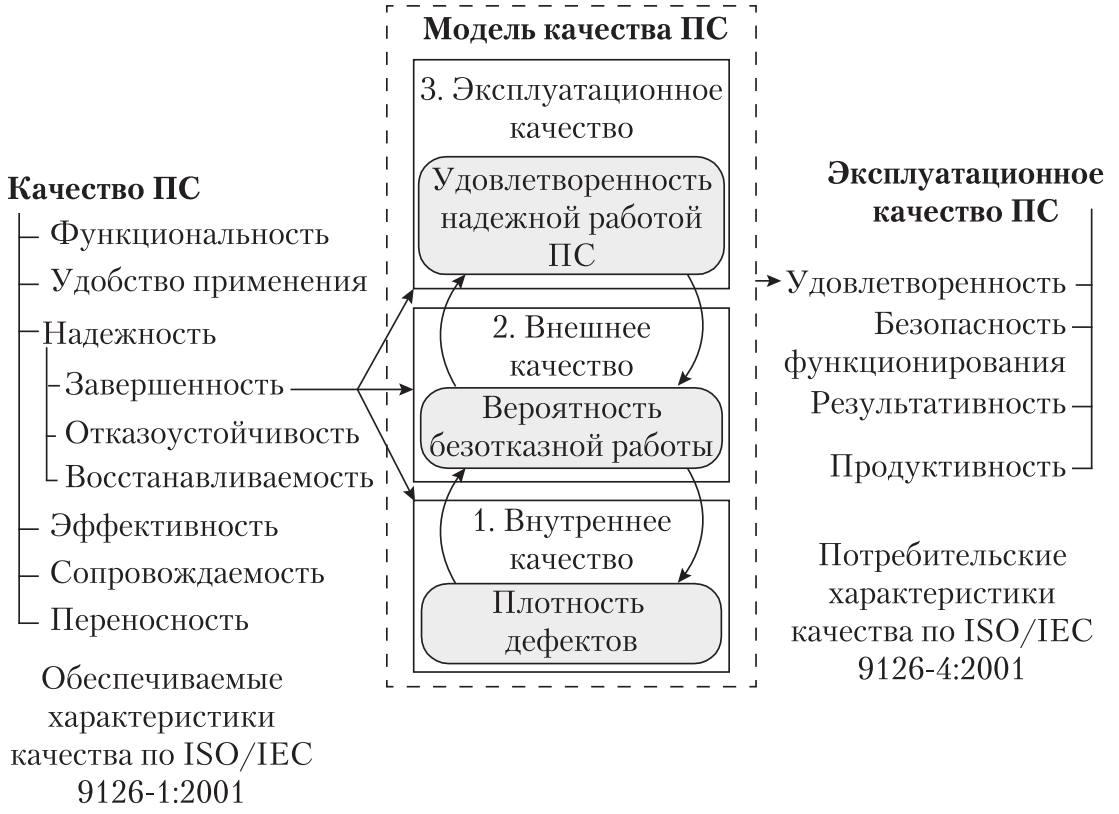


Рис. 10.5. **Трехуровневая модель качества ПС обработки данных**

В данной трехуровневой модели качества ПС главной характеристикой считается завершенность. Ей соответствуют показатели модели качества, которые определяют надежность ПС. К ним относятся: удовлетворенность надежной работой ПС, вероятность ее безотказной работы и плотность

дефектов, а также отказоустойчивость и восстанавливаемость ПС. Эти показатели заданы *метриками качества* на разных уровнях модели качества стандарта ISO/IEC 9126 (1—4):2001.

10.3.2. Задачи управления качеством ПС

Процесс принятия решений по управлению качеством включает в себя решение комплекса следующих задач:

1) определение значения выбранной характеристики качества ПС (целевого уровня качества) как критерия удовлетворения потребностей пользователей;

2) прогнозирование достижимости уровня качества на процессах ПС и условий устранения «слабых мест» в процессах ЖЦ;

3) выбор и принятие решений для предметных показателей (ПрП) в соответствии с заданным уровнем качества и методом оценки их реализации;

4) определение стратегии, методов и средств обеспечения качества компонентов ПС, а также методов проверки правильности рабочих продуктов ПС и сбора данных о ходе их выполнения;

5) анализ тех или иных факторов успешного достижения уровня качества и коррекция хода выполнения процессов проекта ПС.

Подход к управлению качеством упорядочивает решение указанных задач и может быть схематически представлен в виде концептуальной схемы принятия решений по управлению качеством по выбранной характеристике. Она включает в себя описание моделей и методов, разработанных в рамках данного подхода к управлению качеством.

10.3.3. Модель требований для обеспечения качества ПС

Модель требований к завершенности компонентов ПС построена с учетом дифференцированного подхода к ним и необходимости достижения установленных целевых значений завершенности компонентов ПС, адекватных потребностям заказчика ПС.

Мера эксплуатационного качества ПС определена как *функция полезности* вида

$$Q_{\text{ПС}} = \sum_{i=1}^{k} a_i \cdot R_i,$$

где a_i — мера важности i-й функции ПС для делового процесса; R_i — надежность (безотказность) выполнения функции в заданном периоде t эксплуатации системы.

Безотказность выполнения функций ПС оценивается во время системного тестирования и эксплуатации ПС. Поэтому проблема нахождения оптимального уровня завершенности декомпозируется по четырехуровневой иерархии (*ПС → функции ПС → программные приложения → программные модули*). Такая иерархия обеспечивает определение параметров модели требований для каждого компонента ПС.

Оценки важности отдельных функций в деловом процессе и оценки важности отдельных программных приложений для выполнения функций ПС выполняются в контексте риска отказов модулей при работе системы.

Рассмотрим задачу нахождения оптимального целевого уровня *завершенности* каждого компонента системы, который обеспечивает максимизацию функции полезности $Q_{ПС}$ с учетом технических и ресурсных ограничений проекта ПС. Пусть:

u_i — коэффициент относительного веса функции F_i при достижении эксплуатационного качества $Q_{ПС}$, $i = 1, ..., kv_{ij}$ — коэффициент относительного веса j-го ПС в обеспечении выполнения i-й функции, $u = 1, ..., k$; $j = 1, ..., l$;

w_{js} — коэффициент относительного веса s-го элемента ПС при выполнении j-го программного приложения, $s = 1, ..., m$; $j = 1, ..., l$;

r_s — безотказность модуля M_s в период эксплуатации t;

E_j — множество номеров всех модулей, необходимое для выполнения j-го компонента ПС;

α_s — нижняя граница безотказности модуля M_s;

β_s — верхняя граница безотказности модуля M_s;

G — общая цена ПС;

C — себестоимость создания ПС организацией-разработчиком;

c_s — накладные расходы, связанные с разработкой модуля M_s;

d_s — расходы, необходимые для достижения единичного уровня безотказности модуля M_s;

δ — доля прибыли в цене ПС.

Целевые значения безотказности модулей $r_1, ..., r_m$ определяются по функции максимума полезности $Q_{ПС}$:

$$Q_{ПС}(r_1, ..., r_m) = \sum_{j=1}^{l}\left(\sum_{i=1}^{k} u_i v_{ij} \cdot \prod_{n \in E_j} r_n\right) \to \max$$

при ограничениях

$$0 < \alpha_s \le r_s \le \beta_s \le 1, s = 1, ..., m; \tag{10.2}$$

$$c_s + d_s \cdot r_s \le (1-\delta) \cdot G \cdot \sum_{j=1}^{l}\sum_{i=1}^{k} u_i v_{ij} w_{js}; \tag{10.3}$$

$$\sum_{s=1}^{m}(c_s + d_s \cdot r_s) \le C. \tag{10.4}$$

Данная задача нелинейной оптимизации с линейными ограничениями (10.2)—(10.4) практически решается с помощью пакета MATLAB.

Параметры u_i, v_{ij}, w_{js} ($u = 1, ..., k$; $j = 1, ..., l$; $s = 1, ..., m$) находятся методом анализа иерархий путем парного сравнения и последовательного определения локальных приоритетов компонентов ПС в пределах каждого уровня иерархии по отношению к компонентам предыдущего (высшего) уровня.

Ограничения (10.2) задают допустимые нижние α_s и верхние β_s границы безотказности модулей, исходя из оценок важности каждого модуля.

Ограничения (10.3) устанавливают взаимосвязь общих расходов на разработку модуля для установления линейной зависимости между стоимостью модуля и уровнем его безотказности.

Ограничение (10.4) устанавливает взаимосвязь суммарных расходов на разработку всех модулей и себестоимости создания ПС.

Таким образом, данная модель устанавливает требования к завершенности каждого модуля (r_i), а затем и каждого приложения (q_i) с учетом независимости модулей в структуре ПрП. Так задаются целевые уровни завершенности всех компонентов и модулей ПС.

10.3.4. Система прогнозирования безотказной работы ПС

Контроль достижимости уровня завершенности q_i базируется на построении прогнозов двух видов: поискового прогноза с помощью методов и моделей прогнозной системы:

1) модель раннего *прогнозирования безотказности* ПрП (внешняя метрика качества ПС в контексте завершенности);

2) модель раннего *прогнозирования скрытых дефектов* в ПС (внутренняя метрика качества ПС в контексте завершенности);

3) метод *анализа альтернатив* достижения уровня качества ПС;

4) модель определения *оптимального времени тестирования* компонентов ПС с учетом риска их отказов.

Раннее прогнозирование безотказной работы ПС. Рассматриваемый подход включает раннее прогнозирование (до начала тестирования) и соответственно традиционное позднее прогнозирование.

Раннее прогнозирование безотказности ПС заключается в построении проекции значений мер безотказности, полученных по внутренним метрикам на определенном этапе ЖЦ, а также значений, вычисляемых по внешним метрикам на следующем этапе и в конце разработки ПС.

Цель раннего прогнозирования — определить, какие усовершенствования могут быть сделаны в применяемых методах и процессах разработки ПС с целью обеспечения минимальной плотности дефектов к моменту начала системного тестирования.

Позднее прогнозирование безотказности ПС связано с применением аналитических моделей роста надежности на этапе системного тестирования после сбора определенного количества данных об отказах ПС. При прогнозировании безотказности ПС процесс отказов целесообразно моделировать неоднородным пуассоновским процессом, в котором функция надежности $R(t|T)$ определяется по формуле

$$R(t|T) = \exp(-(m(T+t) - m(T))),$$

где $R(t|T)$ — условная вероятность того, что в течение заданного времени t эксплуатации ПС в определенных условиях среды функционирования не возникнет отказ при тестировании ПС в течение времени T; $m(T)$ — функция роста надежности, представляющей собой среднее количество дефектов в ПрП, выявленных во время его функционирования в течение времени t.

Для раннего прогнозирования безотказности целесообразно использовать экспоненциальную модель Дж. Мусы, которая не требует данных об отказах и особенно пригодна для раннего прогнозирования эксплуатационной надежности ПС в начале системного тестирования.

Функция роста надежности $m(T)$ определяется по формуле

$$m(T) = N_0\left(1-\exp\left(-\frac{\lambda_0}{N_0}\cdot t\right)\right),$$

где N_0 — количество скрытых дефектов в ПС в начале системного тестирования; λ_0 — интенсивность отказов ПС в начале системного тестирования, определяемая по формуле

$$\lambda_0 = N_0 \cdot \frac{\rho \cdot K}{I \cdot \varphi},$$

где ρ — интенсивность выполнения кода (скорость процессора); $K = 4 \cdot 10^{-7}$ — коэффициент выявления дефектов (постоянный для модели Дж. Мусы); I — количество инструкций исходного кода; φ — коэффициент расширения кода (число инструкций выполняемого кода, которое приходится на одну инструкцию исходного).

Условная функция надежности эксплуатации ПС имеет вид:

$$R(t \mid T) = \exp(-(m(T+t) - m(T))) =$$

$$= \exp\left[-N_0 \exp\left(-\frac{\lambda_0}{N_0}\cdot T\right)\left(1-\exp\left(\frac{\lambda_0}{N_0}\cdot t\right)\right)\right]. \qquad (10.5)$$

В частности, если $T = 0$, т.е. ПС вообще не будет пребывать на этапе системного тестирования,

$$R(t) = R(t \mid 0) = \exp\left[-D_0 \cdot I \cdot \left(1-\exp\left(\frac{\lambda_0}{D_0 \cdot I}\cdot t\right)\right)\right], \qquad (10.6)$$

где $D_0 = N_0 / I$ — плотность скрытых дефектов в начале тестирования.

Формулы (10.5) и (10.6) — это метрики раннего прогнозирования безотказности ПС с учетом (или без) времени его тестирования.

Размер (объем) ПС (I) можно определять одним из методов FPA (*Function Points Analysis*) в условных единицах функциональности и конвертировать полученное значение в единицы KSLOC.

Для прогнозирования *вероятной плотности дефектов* D_0 (или количества дефектов N_0) на момент начала системного тестирования используются существующие мультипликативные модели, например модель Rome Laboratory. Однако большинство из этих моделей ориентировано на стандартный каскадный ЖЦ и в то же время в коллективах накоплены собственные ретроспективные (исторические) данные для калибровки таких моделей. В этой ситуации для выполнения прогнозов дефектов предлагается использовать новый класс моделей — графические модели.

10.3.5. Анализ достижения уровня качества

Анализ альтернатив достижения целевого уровня завершенности i-го программного приложения базируется на сравнении прогнозируемого значения безотказности $R_i(t)$ с установленным значением q_i.

Для проведения анализа должны быть предварительно определены следующие **критерии для принятия решений** по управлению качеством i-го ПрП:

• q_i — установленный *целевой уровень безотказности* i-го ПрП;

• D_i^* — максимальное значение плотности дефектов в ПрП, которое не будет препятствием для достижения целевого уровня q_i (*целевой уровень дефектов* в ПрП), определяется из уравнения

$$q_i = \exp\left[-D_i^* \cdot I_i \cdot \left(1 - \exp\left(\frac{\lambda_{0i}}{D_i^* \cdot I_i} \cdot t\right)\right)\right];$$

• L_0 — минимально допустимая вероятность прогнозного значения плотности дефектов (*приемлемый уровень уверенности* менеджера проекта);

• максимально допустимые *отклонения*:

– Σ_r — прогнозного значения безотказности $R_i(t)$ от целевого значения q_i,

– Σ_d — прогнозного значения плотности дефектов D_{0i} от D_i^*,

– Σ_d — полученной наибольшей вероятности прогнозного значения плотности дефектов $L(D_{0i})$ от приемлемой L_0.

Исходя из предположения о том, что на ранних стадиях ЖЦ $q_i \geq R_i(t)$ и $D_{0i} \geq D_i$, существуют такие альтернативы относительно последующих действий по обеспечению качества i-го ПрП.

1. Если прогнозирование выполняется с приемлемым уровнем уверенности $L(D_{0i}) - L_0 \geq \sigma_p$, тогда:

а) если $|q_i - R_i(t)| \leq \sigma_r$, то продолжать выполнение проекта «как есть»;

б) если $|q_i - R_i(t)| > \sigma_r$, то принимать решение, учитывая оценки ресурсов проекта, в частности времени до завершения разработки, времени и трудоемкости системного тестирования, а также возможностей совершенствования процессов ЖЦ ПС.

2. Если уровень уверенности при прогнозировании $|L(D_{0i}) - L_0| < \sigma_p$, то это не позволяет принять однозначного решения о состоянии разработки, целесообразно вычислить $R_i(t)$ в диапазоне близких значений D_{0i} и оценить относительные отклонения $R_i(t)$ от q_i для каждого из них, имея в виду, что D_{0i} — это *плотность* дефектов в ПС, размер которой вычислен в условных единицах функциональности, т.е. фактический диапазон значений $R_i(t)$ может быть достаточно широким.

3. Если уровень уверенности при прогнозировании $L_0 - L(D_{0i}) > \sigma_p$, т.е. вероятности распределения значения D_{0i} ниже приемлемого уровня уверенности или с самого начала проекта нарушаются исходные предположения для ранних стадий ЖЦ ПС, это свидетельствует о несовершенстве применимой графической модели (малое количество учтенных факторов дефектов, некорректно определены априорные распределения переменных в вершинах без родителей и т.п.) и необходимости ее уточнения.

Если в ходе выполнения проекта после стабилизации кода i-го ПС (при приближении разработки к завершению) не наблюдается тенденция к снижению плотности дефектов (по результатам прогнозирований в контрольных точках проекта t_s, t_{s+1}, ..., получаются значения D_{0i}, для которых $D_{0i} - D_i^* > \sigma_d$) , то это свидетельствует о несовершенстве процессов разработки (проектирования, программирования, верификации). Одно из возможных решений — передача ПС в группу тестирования для выполнения процесса тестирования.

10.3.6. Задачи оценки показателей качества сложных систем

Сущность подхода к оценке качества с позиций завершенности состоит в усовершенствовании процесса конструирования качественных членов семейств ПС и генерации новых ПС с регулированием показателей качества сложных ПС из готовых КПИ и обязательной оценки показателей качества.

К новым задачам процесса проектирования ПС относятся:

1) моделирование качества семейства ПС и оценивание качества сгенерированных артефактов на каждом процессе инженерного проектирования ПрО;

2) подбор компонентов из репозитория, которые удовлетворяют требованиям ПС для их включения в состав ПС или СПС;

3) верификация КПИ, взятых из репозитория ИТК, которые включаются в состав ПС разрабатываемого СПС;

4) тестирование компонентов и интерфейсов их связи между собой в членах ПС или СПС;

5) проверка показателей качества членов ПС в СПС с точки зрения реализации функциональных требований к ПС или семейству.

Формулировка задач для их реализации:

1) обеспечение качества семейства ПС в процессах инженерного проектирования ПрО;

2) решение задачи о принятии решений по управлению вариантами ПС из КПИ путем конфигурации готовых ПП в среде ИТК.

Задача 1. Сущность задачи — моделирование набора совместных нефункциональных требований к семейству ПС и их спецификация для рассмотрения тех из них, которые могут конфликтовать с показателями качества:

1) высокая надежность → низкая эффективность;

2) высокая эффективность → низкая модификация;

3) высокое качество → высокие затраты и др.

Спецификация конфликтных требований к качеству отмечается в «матрице конфликтов», называемой «крышей дома качества», с указанием влияния отдельных характеристик друг на друга — «положительное, отрицательное или никакого влияния». Предлагается метод анализа иерархий для приведения в порядок спецификаций, взвешивания характеристик и оптимизации их целевых значений.

Задача 2. Цель задачи — проведение верификации описания компонентов ПрО, сбор данных и экспертная оценка полноты, точности, наличия спецификации КПИ и т.п. Верификация выполняется методом формальной инспекции, прототипирования и сценариев тестов и т.п. Результаты анализа выполняются с помощью данных, размещенных в репозитории. К ним отнесены XML-шаблоны спецификации требований, описания артефактов домена, зависимостей между ними, заданных средствами XML-link.

Задача 3. Разработаны новые методики количественного измерения и оценки показателей качества программ:

1) модель качества и оценка надежности ПП, включая КПИ, которые включены по своим функциям в состав ПС;

2) модель принятия решений при управлении качеством на основе байесовских методов и контроля надежности и дефектов ПС;

3) модель распределения надежности системы из компонентов на основе функции полезности продукта — $Q_{\text{ПС}} = \sum_{s=1}^{m} w_s^* \cdot r_s$ — в зависимости от приоритетов w_s^* и надежности $q_j = \prod_{n \in E_j} r_n$ компонентов.

Разработанные модели и методики их использования прошли испытания при создании специальных систем военного назначения. По этим направлениям (экспертирование, тестирование, оценка качества ПС) защищены три диссертации, опубликованы монографии[1] и статьи, которые выложены на сайте http://www.7dragons.ru/ru.

Контрольные вопросы и задания

1. Определите смысл экспертированого оценивания системы.
2. Охарактеризуйте модель экспертизы процессов.
3. Дайте перечень задач экспертной оценки объектов
4. Какие существуют подходы к построению процесса тестирования?
5. Перечислите задачи процесса управления тестированием.
6. Определите цель и задачи процесса тестирования ПрО.
7. Назовите модели тестирования СПС.
8. В чем смысл оптимального времени тестирования СПС?
9. Дайте оценку качества ПС и СПС.
10. Определите суть модели требований обеспечения качества.
11. Охарактеризуйте метод прогнозирования безотказной работы ПС.
12. Перечислите задачи оценки качества сложных систем.

---

[1] *Андон Ф. И.* Основы инженерии качества программных систем / Ф. И. Андон [и др.]. Киев : Академпериодика, 2007; *Лаврищева Е. М.* Новые теоретические основы технологии производства семейств программных систем в контексте генерирующего программирования. 2012; *Ее же.* Программная инженерия. Парадигмы, технологии, CASE-средства. 2016.

Глава 11
МЕТОДЫ РАЗРАБОТКИ ВЕБ-СИСТЕМ В SEMANTIC WEB

Semantic Web — эта новая информационная среда, которая изменяет суть, стиль, инструменты разработки программ, процессов путем применения разнообразных знаний. Понятие «семантическая сеть» возникло раньше, породив понятие Semantic Web (далее также — Семантический Веб, Веб), поэтому их следует разделять. Термин впервые ввел Тим Бернес-Ли в мае 2001 г. в журнале «Scientific American». Веб отображает международные усилия, направленные на представление данных (включая данные World Wide Web) в форматах, доступных автоматизированным обработке, интеграции и рассуждениям. Данные приобретает бóльшую значимость, когда связаны с несколькими источниками, образуя сеть связанных данных. Современные приложения включают интеграцию данных из коллажей (Mash-Ups) для предприятий, улучшенный поиск, композицию сервисов, интеллектуальных агентов, настольные и мобильные приложения и взаимодействие[1].

Развитию проблематики Semantic Web способствуют исследовательские программы США и Евросоюза, объединяющие множество больших и малых производителей, серии конференций по Семантическому Вебу и семантической технологии, а также большое количество разработчиков и проектов с открытым кодом. Онтологии Семантического Веба позволяют лучшее структурировать данные, чем теги Web 2.0, сохраняя при этом бóльшую гибкость.

Стандарты Семантического Веба, такие как RDF, OWL, SPARQL, продолжают развиваться под воздействием пользователей. Разработано много высококачественных инструментов для различных программных сред, в том числе с открытым кодом. Продолжает развиваться инициатива Linking Open Data (связывающие открытые данные), адресованная критической необходимости обеспечения многих приложений основополагающими данными. Многие инструменты и приложения масштабируются (Майк Дин Анн-Арбор, штат Мичиган, ноябрь 2008 г.).

Семантический Веб является стратегической технологией, которая действительно обеспечивает решение задач с существенным преимуществом. Он предлагает новый подход к крайне жестким, но прибыльным проблемам, которые используют большие объемы информации и услуг (сайт http://www.semwebprogramming.org).

---

[1] Semantic Web Represantation of Data on the World Wide Web, based on the Standarts. URL: http://www.w3.org/2001/sw; *Глибовец Н. Н.* Интеллектуальные сети : учебник / Н. Н. Глибовец [и др.]. Днепропетровск, 2014. С. 462.

Программирование в Семантическом Вебе — это новый подход к более эффективному использованию информации и доступных услуг. С ним приходят новые словари, новые инструменты и новые идеи о программировании приложений, а также возможность работы с информацией в процессе выполнения приложений и услуг.

11.1. Структура Semantic Web

Семантический Веб состоит из нескольких дискретных компонентов двух главных категорий: основные компоненты и ассоциированные инструменты. Рисунок 11.1 иллюстрирует основные компоненты и окружающие их инструменты.

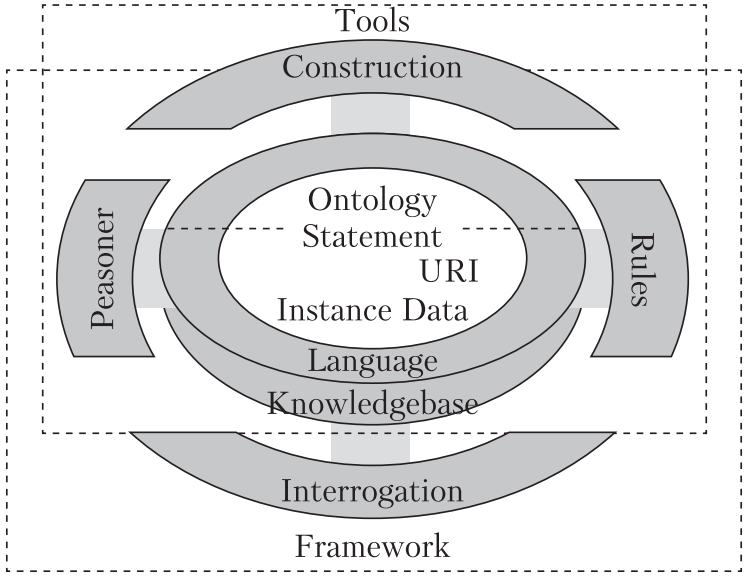


Рис. 11.1. **Основные компоненты Semantic Web**

Компоненты Веба имееют уникальные идентификаторы ресурса — Uniform Resource Identifier (URI), утверждения и включают язык семантического Веба[1] онтологии и данных, Web-Data Minning, MetaData, Data Interchange RDF и др. Остановимся на базовых определениях.

Утверждение. Каждое утверждение состоит из тройки элементов: субъекта, предиката и объекта. Их в Вебе миллиарды. Утверждения определяют информационную структуру компонента, конкретные его экземпляры и ограничения на эту структуру. Утверждения связываются друг с другом при формировании данных Веба.

URI (*Uniform Resource Identifier*) задает уникальные для всего Интернета имена элементов утверждений, каждый из которых (субъект, предикат и объект) содержит URI. Этим подтверждается его идентичность с WWW и исключаются конфликты имен, а также появляется возможность указывать путь к дополнительной информации. URI предоставляет расширяемое пространство имен — ключ к независимой адресации. URI может включать URL (*Uniform Resource Locator*), который может перенаправлять к дополнительной информации или задавать абстрактное Уникальное имя

[1] *Hebeler J.* Semantic Web Programming // Hebeler J. et al. Wiley Publishing Inc. 2008. 372 p.

ресурса (*Uniform Resource Name*, URN). Таким образом, URI может также давать доступ к памяти, адрес которой содержится в URL. Это относится и к идентификаторам интернационализированных ресурсов — IRIs (*Internationalized Resource Identifiers*).

Язык. Он включает набор ключевых слов, с помощью которых можно описывать инструкции и утверждения для различных инструментов Веба. Язык предоставляет пользователю Веба выбор из нескольких языков в соответствии с разнообразием и динамикой Интернета. Языки различной степени сложности и семантической выразительности могут сбалансировать требования к характеристикам исполнения компонентов и выразительности.

Онтология. Состоит из утверждений, которые определяют понятия, отношения и ограничения. Онтология аналогична схеме БД или объектно ориентированной диаграмме классов. Онтология используется для формирования информационной концептуальной модели домена. Готовые онтологии могут быть включены в приложения, использоваться непосредственно или адаптироваться к специфическим потребностям пользователя. Онтология может охватить такие области, как финансы, медицина, физика, биология и др. Онтология поддерживает коммуникации между приложениями. Использование существующей онтологии способствует получению домена с хорошо продуманной и оттестированной информацией с высоким качеством принятых решений.

Данные. Им соответствуют утверждения, содержащие сведения о конкретных экземплярах, а не общем понятии. Это аналогично объектам/экземплярам в ООП. Данные экземпляры не связываются с онтологией, они образуют основной объем Семантического Веба. Онтология, содержащая понятие «личность», может быть использована для миллионов экземпляров личности. Веб-данные отражают их смысловое значение вместе с используемыми данными. Интеграция данных / совместное использование должны давать доступ к информационным ресурсам всей Всемирной сети и обмен ими, включая использование многих существующих источников данных. Динамические данные Веба могут изменять структуры и содержание информации.

Инструменты. Их четыре типа: инструменты конструирования и развития приложений Семантического Веба, справочники для изучения Веба, инструменты рассуждений (резонеры), добавляющие механизмы вывода, а также машины правил для расширения его элементов. Рассмотрим их:

- *инструменты конструирования* позволяют конструировать или интегрировать в Семантический Веб созданные или импортированные утверждения для отдельных элементов онтологии или экземпляров. Некоторые GUI-инструменты предоставляют просмотр и исследование данных Веба и образуют редакторы Семантического Веба. Некоторые программные комплекты позволяют интегрировать программный интерфейс (*Application-Programming Interface*, API) в приложения;
- *справочные инструменты* дают ответ на вопрос о навигации по Семантическому Вебу, начиная от простой навигации по графу и до полного языка запросов;

• *инструменты рассуждений* (*резонеры*) добавляют утверждения к Семантическому Вебу пользователя. Утверждения создают логические дополнения в виде классификаторов. Классификация заполняет структуру класса, позволяя соотносить понятия и отношения с другими. Имеется несколько типов резонеров, предлагающих различные уровни рассуждений. Резонеры включаются в другие инструменты и каркасы. Они служат рычагами создания логически правильных вспомогательных утверждений;

• *машины на правилах* вывода дескрипторной логики. Они добавляют измерения в конструкцию знаний. Правила позволяют слияние онтологий и других логических задач, в том числе таких методов программирования, как поиск с подсчетом по строке (*count and string searches*). Машины на правилах управляют правилами, которые могут рассматриваться как часть общего представления знаний. Каждая машина правил поддерживается языком представления правил.

Семантический каркас. Это пакет, объединяющий перечисленные выше инструменты и позволяющий им работать как единое целое. Есть также несколько отличных коммерческих семантических каркасов. Утверждения, URI, языки, онтологии и данные экземпляра задают связанную семантическую информацию. Семантический Веб инструментов создает, спрашивает, обогащает Семантический Веб, манипулирует им.

11.2. Задание информации в Semantic Web

В Семантическом Вебе может быть размещена информация о книгах, статьях, билетах и др. Она сохраняется в RDF-файлах и используется веб-приложениями для сбора и обработки информации из разнообразных источников. Обрабатывая и комбинируя эту информацию, приложение может представить ее конечному пользователю (например, расписание поездов, цены на автомобили, тематика книг, статей и др.).

Одним из наиболее распространенных примеров Семантического Веба является RSS (*Really Simple Syndication*) — средство для публикации определенной информации типа ленты новостей. Сама информация представляется в языке RSS, основанном на языке XML, размещается в Вебе и используется поисковыми агентами в веб-приложениях.

Семантический Веб предлагает практический подход к проработке и использованию большого количества информации и информационных услуг. Семантика дает способы и методы для получения содержания большего количества информации. Она дает подход для представления данных, включая знания о новых конструкциях и инструментах программирования, которые можно использовать в программах пользователей.

Функции современных компьютеров в основном сводятся к хранению, отображению и поиску информации. Создание информации, ее оценку, классификацию и актуализацию — все это выполняет человек. Современный компьютер пока не понимает человеческого языка, и поэтому используются специальные языки. То есть в идеальном варианте вся информация в Интернете должна размещаться на двух языках: человече-

ском — для человека и компьютерном, можно сказать, математическом, языке, который будет понятен компьютеру. В 2010 г. Б. Гейтс предсказал появление таких разумных машин к 2025 г. По постановке математической задачи они определяют путь ее решения, выполняют и выдают требуемые результаты. На данный момент Веб приближает нас к такому решению.

11.2.1. Базовые элементы Semantic Web

На Вебе имеются языки идентификации различных выражений для их размещения. Как было сказано выше, выражение включает тройку: субъект, предикат и объект. Она формирует основу Семантического Веба и состоит из большого количества таких выражений, образующих **большую базу знаний**.

Для примера, выражение «машина белого цвета» может использоваться для сервиса продажи автомобилей. В данном случае субъект — «машина», предикат — «цвет», объект — «белый». Такое выражение описывается языком RDF, который понятен для машины и может использоваться поисковыми агентами. Для описания переменных выражения используется URI (*Uniform Resource Identifier*), являясь уникальным идентификатором в выражении при размещении в Вебе. Для примера, им может быть URL (*Uniform Resource Locator*), который задает адрес расположения данных.

Язык задания выражений Семантического Веба состоит из набора ключевых слов, которые используются в выражении.

Фреймворки Семантического Веба включают набор программных средств и программных библиотек для создания, обогащения Семантического Веба и манипулирования им. Они помогают создавать выражения, онтологии, веб-приложения и публиковать их в Вебе. Например, в них содержатся среда программирования Eclipse IDE, ЯП Java, набор библиотек для работы с Семантическим Вебом Apache Jena и средство для создания онтологии в Protégé.

Описания любого ресурса в Семантическом Вебе выполняется в формате RDF, который основан на синтаксисе XML и использует идентификаторы URI для обозначения ресурсов.

RDF был утвержден как стандарт комитета W3C в феврале 2004 г. Он представляет систему описания сетевых ресурсов и метаданных (данные о данных). Каждое описание в формате RDF должно прикрепляться к сетевому ресурсу. Документы RDF должны обрабатываться компьютером автоматически. Он не предназначен для чтения и использования человеком. Формат RDF сформировался и получил широкое распространение как каркас для создания Семантического Веба.

RDFS (*RDF Schema*) — это важная надстройка над RDF, которая позволяет создавать классы и свойства (как в ООП).

OWL, язык описания онтологии, вышел в феврале 2004 г. Он построен на форматах RDF, RDFS и предназначен для представления и обработки информации в сети. Имеет три степени детализации, легко масштабируется и согласовывается с современными сетевыми стандартами. В 2008 г. был принят новый стандарт OWL2, включающий описание логики.

SPARQL (*SPARQL Protocol and RDF Query Language*) — новый язык запросов для быстрого доступа к данным RDF. Описание в языке SPARQL позволяет анализировать RDF-описания ресурсов и получать из сети необходимую информацию.

RIF — формат обмена правилами (*Rule Interchange Format*).

11.2.2. Представление онтологии в Semantic Web

Онтология является инструментом для последующей стандартизации дерева знаний, в том числе программной и системной инженерии. Основные направления онтологии:

- разработка онтологических моделей, методов и средств доступа к мировым информационным ресурсам;
- создание моделей описания ГоР разработки как повторно используемых знаний;
- создание библиотек онтологии с профилем знаний;
- разработка методов и средств создания комплекса инструментов по ведению онтологии как рабочего места онтолога;
- разработка методических и учебных материалов по применению Семантического Веба в программировании.

Онтологию можно рассматривать как аппарат построения концептуальной модели домена, которая включает понятия, отношения и ограничения на элементы домена. Онтологии формирует доменную модель ПС, аналогично диаграмме классов в UML. Кроме того, существует много готовых онтологий для разных доменных моделей (биология, медицина, физика и др.), которые можно найти на *e*-сайтах доменов и использовать их как готовые решения для специалистов. Набор средств веб-онтологии представлен на рис. 11.2.

Онтология на Вебе содержит аппарат описания (языки URI, XVL, RDF, OWL и др.), правила построения синтакисиса и семантики ПрО и формализмы задания концептуальной схемы (модели), содержащей структуры данных, релевантные классы объектов, их связи и теоремы, утверждения, ограничения и др.

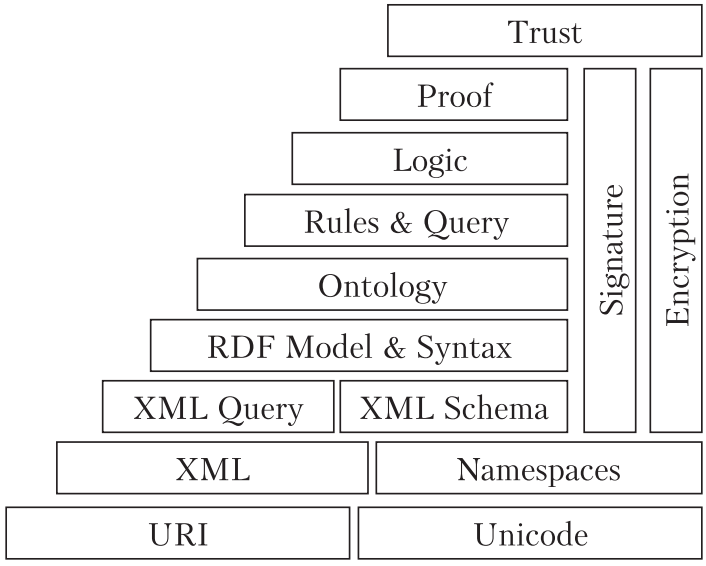


Рис. 11.2. **Стек понятий Semantic Web**

Онтология используется как форма представления знаний о реальном мире или о его отдельных частях. Основные области применения — моделирование бизнес-процессов, Семантический Веб, искусственный интеллект и *e-science* (физика, биология, медицина и др.).

Современная онтология строится для доменов одинаково, независимо от языка написания. Обычно она состоит из индивидов, понятий, атрибутов и отношений, смысл которых дан ниже.

Индивиды — это основные, низкоуровневые компоненты онтологии, представляющие собой как физические объекты (люди, дома, планеты), так и абстрактные (числа, слова). Строго говоря, онтология может обойтись и без конкретных объектов. Однако одной из главных ее целей является классификация таких объектов.

Понятие — абстрактные группы, коллекции или наборы объектов. Они могут включать в себя индивиды, другие классы или же сочетание того и другого.

Объекты в онтологии могут иметь атрибуты. Каждый атрибут имеет по крайней мере имя и значение и используется для сохранения информации, специфической для объекта и привязанной к нему. Значением атрибута может быть сложным ТД.

Важная роль атрибутов в том, чтобы определить отношения между объектами онтологии. Обычно обращением является атрибут, значением которого является другой объект.

К языкам описания онтологии относятся:

- OWL — стандарт W3C, язык для задания семантических утверждений, являющихся расширением RDF и RDFS;
- KIF (*Knowledge Interchange Format*), основанный на *S*-выражениях синтаксиса логики;
- Common logic — преемник KIF, стандартизован ISO / IEC 24707:2007;
- Cycl — язык проекта Cyc, основанный на исчислении предикатов с некоторыми расширениями более высокого порядка DAML + OIL (FIPA).

Для работы с языками онтологии существует несколько видов технологий: редакторы онтологии (для создания онтологии), DBMS онтологии (для сохранения и обращения к онтологии) и хранилища онтологии (для работы с несколькими онтологиями).

В мировой инфраструктуре идет процесс глобализации терминологии, накопления профильной онтологии путем типизации (стандартизации) для актуальных проблемных областей с помощью базовых понятий, отношений и свойств. Основные направления применения и развития идей Семантического Веба:

- создание моделей описания ГоР знаний;
- разработка онтологических моделей, методов и средств доступа к мировым информационным ресурсам;
- создание и стандартизация онтологии верхнего уровня (универсальных доменов) для базовых процессов SE (например, ЖЦ);
- создание библиотек онтологических знаний и методов создания и ведения онтологии как рабочего места онтолога;
- организация создания и использование реестров ГоР;

• разработка методических и учебных материалов по программированию Семантического Веба.

11.2.3. Стандарты консорциума W3C

Основу веб-семантик составляют стандарты языков определения внешнего представления данных и гиперсвязей между ними. Более специализированные языки определяют форматы передачи специфических данных:

- математические формулы;
- векторная и растровая графика (png);
- мультимедиа SMIL (*Synchronized Multimedia Integration Language*).

Стандарты формируются рядом международных организаций: W3C, ISOC, OASIS, GGF (рис. 11.3).

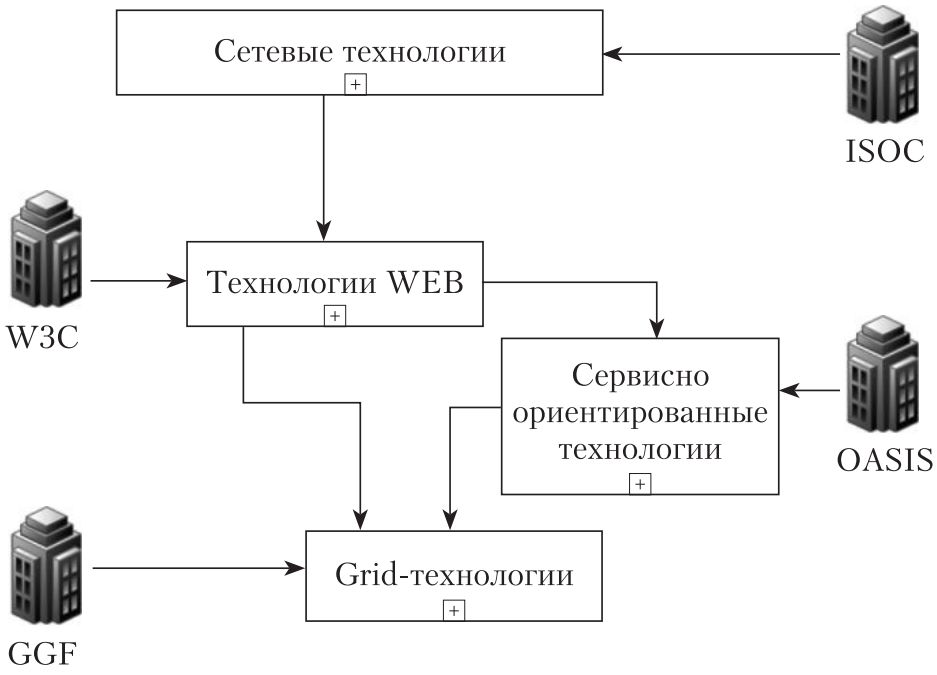


Рис. 11.3. **Схема стандартных Интернет технологий**

Эти организации определяет методы кодирования и интерпретации структурированных данных. Их основу составляет язык XML, который предоставляет средства синтаксического описания структурированных данных. Они задают сетевые и веб-технологии, сервисные и Grid-технологии. Они предоставляет синтаксические и семантические структуры данных ПрО и механизмы для проверки соответствия конкретных документов семантике, определенной в схеме.

11.3. Веб-технологии

Глобальная информационная сеть (веб-среда) объединяет десятки миллионов документов по всему миру благодаря стандартам консорциума W3C, который объединяет наиболее крупных производителей ПО для веб-технологий (веб-серверов и веб-браузеров) и дает механизм взаимодействия в сети («сетевая интероперабельность»). Стандарты глобальной информационной сети можно подразделить на четыре группы (рис. 11.4).

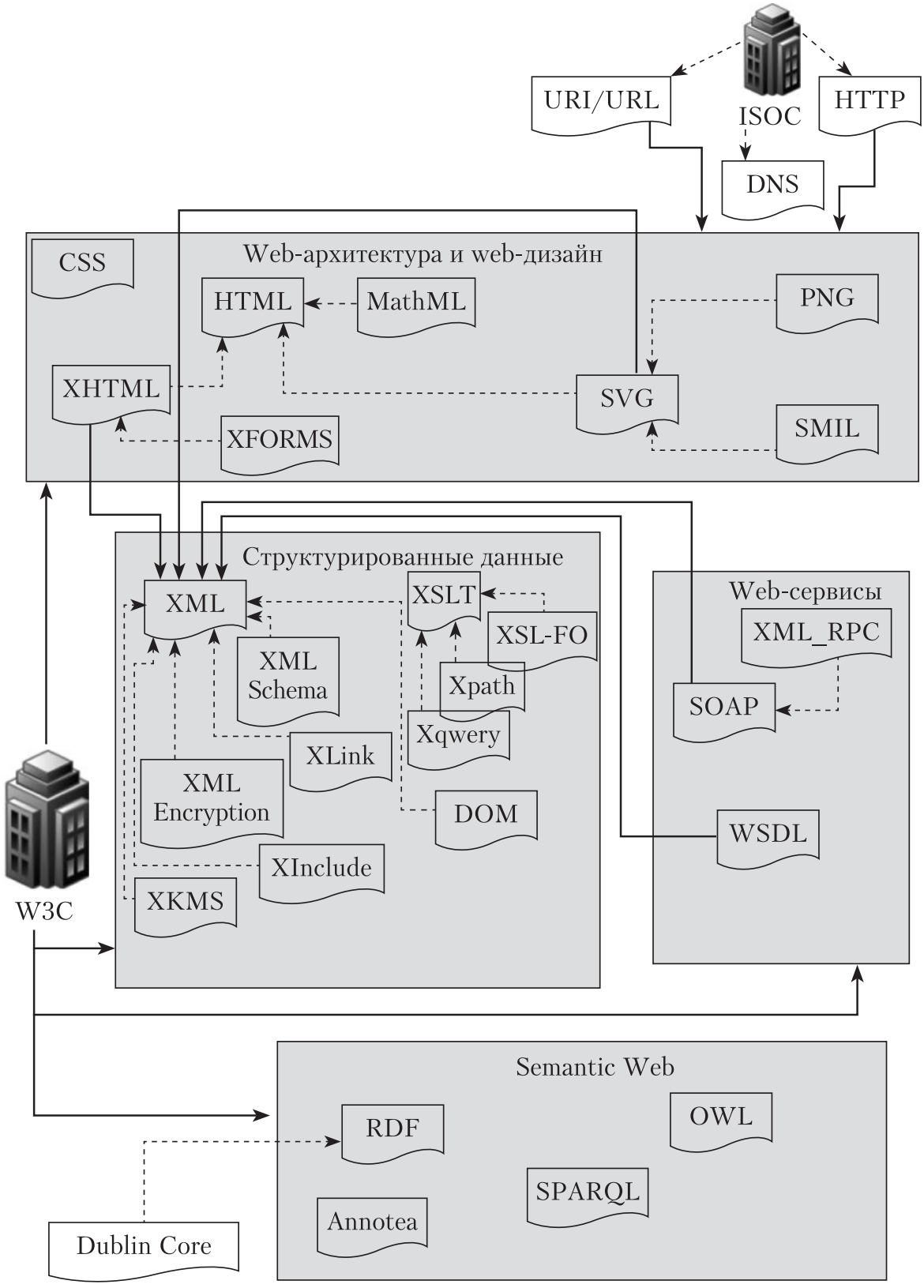


Рис. 11.4. **Технологии Web**

Представление форматных данных пользователя

HTML — язык разметки гипертекста и документов в виде ASCII-текста, фрагменты которого облагаются специальными пометками (тегами).

XHTML — расширяемый язык разметки гипертекста (*Extensible Hypertext Markup Language*) на основе принципов и синтаксиса XML.

CSS (*Cascading Style Sheets*) — каскадные таблицы стилей для описания внешнего вида документа (страницы).

MathML (*Mathematical Markup Language*) — язык математических формул, который применяет формат XML для отображения математических формул. Используется для отображения формул в веб-браузерах MathML.

SVG (*Scalable Vector Graphics*) — язык масштабируемой двумерной векторной графики, которая использует формат XML.

Представление структурированных данных

XML Schema — описание схемы документа в языке XML, состоит из набора тегов и их атрибутов и устанавливает соответствиие документа его предметной отрасли.

XLink (*XML Linking Language*) — XML-схема, которая рекомендована W3C для организации ссылок между ресурсами.

XInclude (*XML Inclusions*) — XML-схема, которая рекомендована W3C и предоставляет механизм включения в XML-документы текстовых файлов и др.

XSL (*eXtensible Stylesheet Language*) — семейство рекомендаций (стандартов), которое определяет методологию превращения XML-документов для визуализации или другой обработки соответствующими программными средствами.

XSLT — декларативный язык, который интерпретирует правила и применяет их к входным документам.

DOM (*Document Object Model*) — независимая от платформы и языка программирования модель, которая позволяет создавать динамические веб-страницы посредством скриптовых языков (JScipt, JavaScript).

XML Encryption — определяет порядок шифровки и дешифровки содержания элементов XML-документа.

XKMS (*XML Key Management Specification*) — определяет порядок безопасной передачи и регистрации открытых ключей для шифровки и дешифровки XML-документов за протоколом XML Encryption.

PNG (*Portable Network Graphics*) — формат для сохранения растровой графики.

SMIL (*Synchronized Multimedia Integration Language*) — язык мультимедийной интеграции на синтаксической основе формата XML.

Протоколы удаленного выполнения сервисов

SOAP (*Simple Object Access Protocol*) — определяет порядок обмена сообщениями (данными) в веб-среде.

WSDL (*Web Services Description Language*) — язык для описания вебсервисов и доступа к ним на основе XML.

Представление семантических данных

RDF (*Resource Description Framework*) — ряд стандартов, которые определяют базовые методы формального представления знаний для машинной обработки.

OWL (*Web Ontology Language*) — язык, который определяет правила описания онтологии предметных отраслей и предназначен для обеспече-

ния одинаковой и однозначной интерпретации документов разными агентами в распределенной среде.

Наиболее распространен протокол SOAP. При описании сервиса указывается адрес URI транспортного протокола (например, HTTP). Средством описания функциональности сервиса является язык WSDL (*Web-service Description Language*). Для представления данных, в особенности метаданных, используется модель RDF. Описание процессов представления и обработки запросов на сервисы в графическом виде осуществляется языками:

- WSCI (*Web Services Choreography Interface*);
- WSCL (*Web Services Conversation Language*);
- BPMN (*Business Process and Model And Notation*);
- BPEL (*Business Process Execution Language for Web Services*) и др.

11.3.1. Язык BPMN для спецификации бизнес-процессов

Business Process Modeling Notation (BPMN) создан группой BPMI, целями которой являются:

- создание графических диаграмм бизнес-процессов и автоматический их перевод в представление процессов на языке WS BPEL (*Web Services Business Process Execution Language*);
- визуализация XML-языков при выполнении бизнес-процессов, таких как BPEL4WS, в обозначениях бизнеса;
- создание регистров для повторного использования спецификаций моделей бизнес-процессов или их частей, которые понятны создателям и пользователям таких моделей.

Главное назначение BPMN — создание стандартизированного моста для преодоления водораздела между разработчиками бизнес-процессов и реализаторами процессов, а также обеспечение автоматического перевода графических диаграмм процессов BPMN в представление процессов на языке BPEL.

BPMN является графовым языком и его отображение на BPEL с блочной структурой — сложная задача. В нем можно моделировать плохо сформированные процессы, которые не могут быть переведены в исполняемый набор инструкций BPEL.

Для преодоления этих трудностей предложено семантическое расширение BPMN — Semantic Business Process Modeling Notation (sBPMN). Его назначение следующее:

1) добавлять смысл каждому из элементов процесса и сделать их машиночитаемыми;

2) сделать возможными рассуждения над описаниями процессов;

3) усилить расширения для семантических веб-сервисов (SWS), в том числе:

- предоставить возможность автоматически привязываться к каждой задаче реализации веб-сервиса (или их композиции),
- генерировать представление процессов в BPEL, развернуть механизм исполнения и затем выполнить процесс.

11.3.2. Процессы и операции бизнес-процессов

Впервые BPMI Notation Working Group был выпущен в мае 2004 г. Последняя версия: OMG Document Number: formal/2011—01-03 Standard document (URL: http://www.omg.org/spec/BPMN/2.0).

В языке BPMN проводится спецификация требований на разработку ПС, в том числе коммерческих[1]. В нем по сравнению с языком UML акцент сделан на взаимодействии акторов системы.

Процесс — это любая деятельность, осуществляемая внутри компании или организации. Любой процесс в нотации BPMN представляет собой последовательное или параллельное выполнение различных **действий (операций)** с указанием определенных бизнес-правил.

Внутри процесса могут выделяться **подпроцессы**. Действия могут состоять, в свою очередь, из более мелких действий. Элементарное действие называется **задачей**, имеющей одного исполнителя (человека или программу).

В процессе участвует круг лиц, называемых участниками. Каждый из них играет свою роль или имеет зону ответственности — пул. Пул ограничивает потоки работ, выполняемые конкретным участником.

Очередность действий в процессе или *поток управления* задается в виде сети *объектов потока*, которыми являются:

- определенные *действия* или последовательность действий;
- *события*, прерывающие последовательное течение потока управления. События могут генерироваться внешними факторами или внутренними действиями процесса. Различаются стартовые, завершающие и промежуточные события;
- *логические операторы* или *шлюзы* (*gateway*), управляющие (подобно ЯП) передачей управления от действия к действию по условию, в цикле, разделяющие поток управления на параллельные ветви или сводящие их в единое русло и т.п. Это точки принятия решений в процессе;
- *операции ассоциации* используются для установления связей между артефактами, данными или текстовыми аннотациями объектов потока управления.

11.3.3. Типы объектов, событий и задач в BPMN

BPMN описывает бизнес-процессы в виде диаграмм бизнес-процессов с небольшим числом графических элементов четырех категорий:

1) **объекты потока управления** — события, действия и логические операторы, задаваемые специальными знаками;

2) **соединяющие объекты** — поток управления, поток сообщений и ассоциации, задаваемые специальными линиями;

3) **участники процесса** (конкретные или их роли). Представлены на диаграммах вертикальными или горизонтальными прямоугольниками — пулами, внутри которых могут детализироваться потоки управле-

---

[1] См.: *Федоров И. Г.* Моделирование бизнес-процессов электронной коммерции в нотации BPMN 2.0. Стандарт ISO/IEC 195110—2013. М. : Tempus, 2015.

ния — дорожки (*line*), за которые отвечают участники (каждый имеет свою зону ответственности);

4) **артефакты** — данные, группы и текстовые аннотации.

Элементы этих четырех категорий позволяют строить простейшие диаграммы бизнес-процессов. Для повышения выразительности модели спецификация разрешает создавать новые типы объектов, потоков управления и артефактов.

События-таймеры (*timer events*) моделируют события, регулярно происходящие во времени (моменты времени, периоды и таймауты).

События-условия (*conditional events*) позволяют интегрировать бизнес-правила в процесс.

События-сигналы (*signal events*) рассылают и принимают сигналы между несколькими процессами. Один сигнал может обрабатываться несколькими получателями. Таким образом, события-сигналы позволяют реализовать широкую рассылку сообщений.

Задание или **задача** (*task*) — это единица работы, элементарное действие в процессе:

- циклическое действие (*loop activity*) выполняется, пока условие цикла верно. Условие цикла может проверяться до или после выполнения действия;
- множественные экземпляры (*multiple instances*) действия показывают, что одно действие выполняется многократно, по одному разу для каждого объекта. Например, для каждого объекта в заказе клиента выполняется один экземпляр действия. Экземпляры действия могут выполняться параллельно или последовательно.

Ad-hoc-подпроцесс (*ad-hoc subprocess*) содержит задания, которые выполняются до тех пор, пока не выполнено условие завершения подпроцесса.

11.3.4. Моделирование семантики бизнес-процессов

Моделирование семантики бизнес-процессов включает следующие операции:

- семантическое аннотирование существующих моделей процессов, т.е. добавление ссылок на элементы онтологии, цели (WSMO) и семантические ограничения. Использование BPMN способствует сохранению существующей модели предприятия и не вводит новые графические нотации;
- хранение семантических моделей в библиотеке семантических бизнес-процессов и обслуживание запросов к библиотеке для поиска существующих семантических моделей процесса или их фрагментов с целью их повторного использования и моделирования новых процессов;
- перевод представленных семантических моделей бизнес-процессов в исполняемые модели процессов, т.е. перевод sBPMN в sBPEL. Среда моделирования должна включать в себя сервис трансформации, который будет порождать исполняемые модели sBPEL.

Инструменты семантического аннотирования моделей процессов могут способствовать решению ряда взаимосвязанных задач:

- повторно использовать части модели процессов в создаваемых новых моделях;

- сделать модели процессов исполняемыми;
- обнаруживать отношения между процессами;
- поддерживать управление изменениями;
- предоставлять структурную основу для передачи знаний.

Информация должна быть понятна бизнес-экспертам во время моделирования. Пользователи должны быть в состоянии понять семантическую информацию, с которой они имеют дело.

Дополнительная информация должна способствовать реализации процессов, помогая находить веб-сервисы и образовывать их композиции для действий (*activities*) процессов модели. Для поддержки семантического улучшения моделей процессов информация должна быть получена из соответствующих онтологий домена.

Для представления смысла BPMN в ряде SUPER-проектов предлагается понятие WSMO: *goal — цель*, которая формулирует *capability — возможность*. Эти конструкции являются прямыми аналогами вариантов использования UML (Use case). Модель ПрО обогащается знаниями о целях действий, характерных для нее. Такие же знания включаются в семантические аннотации процессов, подпроцессов, действий и задач в модели бизнес-процессов.

Фактически приемы семантического обогащения онтологий ПрО и моделей бизнес-процессов включением в них характерных состояний объектов, действий, меняющих указанные состояния указанных объектов, целей сближают три типа объектов моделирования — домена или ПрО, бизнес-процессов и семантических веб-сервисов. Это позволяет связать элементы семантических аннотаций бизнес-процессов с реализующими их веб-сервисами или веб-сервисами, входящими в требуемую реализацию. Кроме того, улучшенная модель бизнес-процессов способствует обнаружению подходящих веб-сервисов. Семантические аннотации используются при запросах к репозиториям моделей процесса в масштабах ПрО.

11.3.5. ЖЦ бизнес-процессов

ЖЦ управления бизнес-процессами включает следующие фазы.

Фаза 1. Составление модели процесса в виде порядка задач (*tasks*) бизнес-процесса с использованием средств BPMN и комментариев на естественном языке. Для исполнения такой модели устанавливается связь с IT-сервисами и форматами данных.

Фаза 2. Преобразование модели процесса в модель *workflow* (потока работ) и трансляция этого описания в язык BPEL для исполнения. Если семантические аннотации отсутствуют, то требуемые веб-сервисы присоединяются вручную.

Фаза 3. Исполнение механизмов делегирования автоматизированного задания веб-сервисов. Процесс сам рассматривается как веб-сервис и может быть использован другими.

Фаза 4. Анализ качества процесса — мониторинг исполнения процесса состоит в определении ветвей процесса, ошибок и текущих вариатов экземпляров процесса и др. Одна из главных проблем — трансляция модели бизнес-процесса аналитика в модель *workflow* IT-инженера с помощью технологий Семантического Веба.

Применение семантических технологий не влияет на главные фазы ЖЦ, но повышает степень автоматизации при переходах от фазы к фазе и расширяет функциональности sBPM. ЖЦ для sBPM тот же, а функциональность другая.

В ходе моделирования подбираются повторно используемые фрагменты процесса с помощью семантических средств обнаружения. Он описывает требуемую функциональность с указанием нужных онтологий, как на шаге аннотирования. Получив альтернативы, выбирает одну из них и вставляет в свою модель процесса. Для незавершенной части модели процесса может быть выполнено автоматическое дополнение (*Auto-Completion*), если система может найти недостающие фрагменты, которые запомнила ранее.

Семантика элементов процесса таким образом явно указывается в машиночитаемой форме с помощью ссылок на сущности онтологий. Сам процесс определяется на основе онтологии процесса (построенной на базе BPMN).

Онтологии создаются экспертами домена, онтологическими инженерами и бизнес-аналитиками. Могут делаться ссылки на другие онтологии, полезные в контексте предприятия.

При этом семантические онтологии позволяют или облегчают дополнительные функции, а именно обнаружение и автодополнение повторно используемых фрагментов процесса. Имеются в виду части процесса, которые определены как потенциально повторно используемые и помещены в репозиторий семантических бизнес-процессов.

На фазе 2 реализуется трансляция в BPEL4WS — семантические онтологии формулируют запросы к регистрам веб-сервисов в терминах упомянутых онтологий, чтобы найти подходящие для реализации бизнес-процессов семантические веб-сервисы. То есть те сервисы, которые имеют аннотации (или спецификации) в терминах соответствующих онтологий, облегчают их поиск и композицию.

Пути построения онтологии sBPMN. Выделяются следующие типы сущности — процесс, задача, шлюз, класс. Основные элементы диаграмм бизнес-процесса моделируются как классы, имеющие соответствующие атрибуты, определенные в спецификации BPMN. Аннотация процессов в sBPMN означает создание экземпляров ее понятий, например, *задача* (*task*) *получения лицензии* (*Obtain License*) будет экземпляром класса *задача*, а не его подклассом. Чтобы убедиться в том, что все элементы модели процесса относятся к нему, вводится специальное свойство *hasProcess* для явной или неявной (через рекурсию) ссылки.

Поток управления (*Sequence Flow*) моделируется с помощью правил соединения свойств *источник* (*Source*) и *цель* (*Target*) понятия *Поток управления*, определяющих, какие объекты потока (*Flow Objects*), например, *Tasks*, *Activities*, *Events*, *Gateways* (*задачи*, *действия*, *события*, *шлюзы*), могут быть связаны друг с другом (в соответствии с правилами соединения *потока управления* BPMN (*Sequence Flow Connection Rules*)). Правила соединения для потока сообщений реализуются аналогично.

Основным элементом онтологии sBPMN является диаграмма бизнес-процесса, представляющая модель процесса. Согласно спецификации BPMN имеются четыре основных категории элементов: Flow Objects,

Connecting Objects, Swimlanes и Artefacts. Для ясности и совместимости с другими онтологиями проекта SUPER понятие *Процесс* вводится на том же уровне. В остальном основные концепции онтологии sBPMN совпадают с концепциями BPMN.

Текущая онтология sBPMN имеет 95 понятий, более 50 аксиом. Она доступна на веб-сайте проекта SUPER (http://www.ipsuper.org).

BPMO Based Semantic Business Process Modelling Environment. В рамках проекта SUPER определяется набор онтологий для моделирования бизнес-процессов. Ядро стека онтологий SUPER состоит из следующих элементов онтологии:

- Upper Process Ontology (UPO), определяющая понятия верхнего уровня, такие как задание (*task*), цель (*goal*) и условие (*condition*);
- Business Process Modelling Ontology — BPMO (онтология моделирования бизнес-процессов), расширяющая UPO в полную онтологию процессов, абстрагируя различные нотации бизнес-процессов BPMN;
- sBPMN и sBPEL — онтологизированные версии подмножеств BPMN и WS-BPEL соответственно. Кроме того, sBPEL дополнительно обогащается элементами Web Services Modelling Ontology (WSMO), ориентированными на цели обнаружения, посредничества и исполнения сервисов.

Пользовательский интерфейс основан на графической нотации BPMN, расширенной специфическими примитивами моделирования BPMO (например, паттерны блока), и интергируется с существующими средствами WSMO Studio. Таким образом, для создания семантической аннотации модели процесса в BPMO можно перетащить и опустить (*drag & drop*) существующие семантические элементы (например, WSMO-цели, семантические ограничения, концепции и др.) в релевантные элементы модели процесса (элементы процесса, действий, поток данных).

11.4. Веб-сервисы

История развития программирования с самого начала связана с использованием сервисов и услуг. Сервисы реализуют некоторые дополнительные возможности, которые необходимы многим программистам и потенциальным пользователям. Как правило, описания сервисов содержат в себе информацию о назначении и форме представления сервиса, которое включает три вида сервисов:

1) общие системные, которые есть в каждой общесистемной среде для поддержки процессов проектирования и реализации ПС;

2) объектные сервисы, которые поддерживают объекты и классы, операции ЖЦ, услуги, необходимые для разработки ОС;

3) веб-сервисы, которые являются информационными ресурсами Интернета и предоставляют бизнес-услуги для решения задач во Всемирной Паутине.

Каждый сервис имеет: имя (*Name*), способствующее поиску в распределенной среде пространства имен; связь (*Binding*) для задания соответствия «имя — объект»; транзакцию (*Transaction*) для организации и управления

отдельными компонентами в Интернете; сообщение (*Messaging*) для общения с компонентами.

Четыре вида сервисов обязательны для любой модели ПС и ее реализации. С их помощью осуществляются: поиск компонентов; доступ к их ресурсам; организация обмена информацией между компонентами; динамическое управление функциями, обусловленными некоторыми готовыми КПИ в интернет-системе.

Веб-сервис имеет интерфейс, описанный на языке SGL (*Services Description Language*). Система взаимодействуют с веб-сервисом в порядке, установленном в его описании через SOAP-сообщения, которые передаются HTTP с XML в сочетании с другими веб-стандартами. Прикладные программные веб-интерфейсы, позволяющие комбинировать несколько веб-сервисов в новых приложениях, называются **mush-up**.

Можно выделить два основных класса веб-сервисов: REST-совместимые веб-сервисы, в которых основной целью сервиса является работа с XML, и произвольные веб-сервисы, в которых сервис может выполнить произвольный набор операций.

Сервисы REST не требуют использования XML, SOAP и WDSL. Семантика таких сервисов базируется на совокупности HTTP-команд, которые выглядят следующим образом (далее приведены основные команды и названия соответствующих методов на примере JAX-RC — реализации REST-средствами платформы Java):

- createResource (PUT) — создать новый ресурс по соответствующим идентификаторам;
- getResourceRepresentation (GET) — отыскать представление ресурса;
- deleteResource (DELETE) — удалить заданный ресурс;
- modifyResource (POST) — модифицировать ресурс;
- getMetaInformation (HEAD) — получить метаинформацию о ресурсе.

Понятие сервиса является логическим развитием термина «программный компонент», который определяется как открытый компонент, пригодный к использованию в распределенных прикладных системах. Провайдерами сервисов обеспечиваются их программная реализация, описание и возможность коммерческой либо иной поддержки. Благодаря описаниям сервисов, в которых определяются их возможности, интерфейсы, поведение и т.п., пользователи имеют возможность найти наиболее подходящие модули для интеграции в собственную архитектуру. Подобные возможности используются в SOC (*Service-Oriented Computing*), SOA (*Service-Oriented Architecture*) и SCA (*Service-Componented Architecture*).

Под **веб-сервисом** понимается программная единица, которая выполняется на некотором узле, доступна по определенному адресу и имеет строго определенный **интерфейс**.

Порядок взаимодействия с веб-сервисом определяет протокол **SOAP**, который позволяет выполнять как простой удаленный вызов процедур, так и выполнение взаимодействия в виде запрос/ответ, основанный на обмене сообщениями. Для описания интерфейса могут использоваться языки **WSDL** и **IDL**.

Каждый веб-сервис представляет собой объект со свойствами и методами и идентифицируется строкой URI (уникальный идентификатор ресурса). Обмен данными осуществляется с помощью протокола SOAP, построенного на основе интернет-протокола высокого уровня (HTTP), сформулированных в XML-запросах. Описание доступных методов и свойств, которые позволяют связывать веб-сервис с конкретными программными платформами, также даются в XML-форме, регламентированной стандартом WSDL (*Web Service Definition Language*). В качестве примера веб-сервиса в ИТК приведен сервис, реализующий арифметические операции, построенные на платформе Java EE. Доступ осуществляется через веб-интерфейс (в виде веб-страницы с вызовом отдельных методов сервиса), являющийся сервлетом Java, написанным для клиента на платформе MS.NET.

Модель сервиса имеет типизированную структуру и интерфейс. Для представления разных аспектов описания и функционирования сервисов используется язык XML. К этим аспектам относятся описание структур и семантики данных, механизмов взаимодействия с сервисами, функциональных услуг и механизм поиска необходимых сервисов.

Средством описания механизма взаимодействия с сервисами является SOAP, а описания функциональности сервисов — язык WSDL. Описание функциональности сервисов выполняется унифицированными языками XML, WSDL; описание структуры и семантики данных — RDF; описание процессов представления и обработки сервисов — языком BPMN и взаимодействия с сервисами и поиска необходимых сервисов — язык SOAP.

Основными формами реализации веб-сервисов являются идентификация URL-адресами и взаимодействие сообщениями через Интернет. Веб-сервис имеет URL-адрес, интерфейс и механизм взаимодействия с другим сервисом через протоколы Интернета. Обмен данными между веб-сервисом и программой осуществляется с помощью XML-документов, оформленных в виде сообщений. Веб-сервисы могут способствовать решению бизнес-задач разной природы через интернет-провайдеров[1].

Основные средства описания и разработки новых систем средствами веб-сервисов:

- язык XML для описания и построения SOA-архитектуры;
- язык WSDL для описания веб-сервисов и их интерфейсов в XML, что касается ТД и сообщений, а также моделей взаимодействия и протоколов связи сервисов между собой;
- SOAP (*Simple Object Access Protocol*) для определения форматов запросов к веб-сервисам;
- SCA (*Servise-Component Architecture*) для создания более сложной системы на основе компонентов и сервисов;
- UDDI (*Universal Description, Discovery and Integration*) для универсального описания, выявления и интеграции сервисов, обеспечения их

[1] *Лаврищева Е. М., Карпов Л. Е., Томилин Н.* Сервисные средства интернет для решения бизнес-задач // Труды ИСП РАН. 2015. Т. 1.

хранения, упорядочения деловой сервисной информации в специальном реестре с указателями на конкретные интерфейсы веб-сервисов.

Чтобы найти нужный веб-сервис, достаточно обратиться на сайт http://www.uddi.org/, запустить поиск и получить описание сервисов. Определения форматов веб-сервисов представлены на языке WSDL (*Web Services Description Language*). Если необходимый веб-сервис найден, то нужно написать JSP-страницу с обращением к найденному веб-сервису.

Для создания веб-сервисов могут использоваться такие инструменты, как *Open Net* (компании Sun), *.NET* (Microsoft), *e-services* (HP) и *Web Services* (IBM). Существуют также инструменты с открытыми исходными кодами (*open source frameworks*), например проект *Mono Project* (www.go-mono.com), предоставляющий систему компиляции, исполнения кода и библиотек для работы одних и тех же веб-сервисов на всех платформах, включая Unix. Для моделирования систем с помощью сервисов используются модели SOA и SCA.

11.4.1. Базовые модели сервисов SOA и SCA

Модель SOA (*Servise Oriented Architecture*). Объектом сервисно ориентированной архитектуры является веб-сервис, который обеспечивает функциональность (*Functions*) и качество сервисов (*Quality service*). Эти технологии вынесены на уровень IT-стандартов комитета W3C и имеют следующие уровни.

Технология функциональности веб-сервисов включает:

- транспортный уровень (*transport layer*) для обмена данными;
- коммуникационный уровень (*service communication layer*) для определения протоколов;
- уровень описания сервиса (*service description layer*) и связанных с ним интерфейсов;
- уровень бизнес-процессов (*business process layer*) для реализации бизнес-процессов и потоков работ через механизмы веб-сервисов;
- уровень реестра сервисов (*service registry layer*), который обеспечивает организацию библиотек веб-сервисов для их публикации, поиска и вызова для их WSDL-описаний интерфейсов.

Технология обеспечения качества веб-сервисов имеет следующие уровни:

- политики (*policy layer*) для описания правил и условий применения веб-сервисов;
- безопасности (*security layer*) для описания вопросов безопасности веб-сервисов и функционирования (авторизация, аутентификация и распределение доступа);
- транзакций (*transaction layer*) для установления параметров обращения к веб-сервисам и обеспечения надежности их функционирования;
- управление (*management layer*) веб-сервисами.

Технологический фундамент веб-сервисов составляют: XML, SOAP, UDDI, WSDL. С их помощью осуществляется реализация базовых свойств веб-сервиса и механизма взаимодействия между собой веб-сервисов в среде SOA;

• провайдер сервиса, который осуществляет реализацию сервиса в виде веб-сервиса, прием и выполнение запросов пользователей сервиса, а также публикацию сервиса, отмеченного в реестре;

• реестр сервисов, который содержит в себе библиотеку сервисов для пользователей сервиса и средства поиска и вызова необходимого сервиса для запросов, которые поступили от провайдеров сервисов на предоставление сервисов;

• пользователь сервиса (приложение, программный модуль или сервис), который осуществляет поиск и вызов необходимого сервиса из реестра описания сервисов, а также использует сервис, предоставленный провайдером в соответствии с его интерфейсом.

Модель SCA (*Servise-Component Architecture*) обеспечивает доступ к сервисным компонентам и определение зависимости между ними через аппарат ссылок. Компоненты SCA системы IBM ® WebSphere ® Integration Developer могут быть упакованы в модуль для выполнения сервисного модуля с WebSphere Process Server — эквивалентного EAR-файлу J2EE и некоторым другим. Подмодули J2EE и артефакты упаковываются с модулем SCA. Это позволяет запустить сервис через модель SCA и передавать данные при их интеграции. Механизмы, которые используются для вызова внешнего сервиса, названы *импортом* и *экспортом*. Они связаны с другими технологиями, такими как JMS, Enterprise JavaBeans или веб-сервисы. SCA-модуль позволяет обратиться к существующему Enterprise JavaBean, используя модель SCA. Элементы SCA могут компоноваться и обмениваться данными друг с другом, пересылая SDO, подготовленные в необходимом виде. Этот интерфейс включает определения метода получения и устанавления свойства данных. Инструментарий WebSphere Integration Developer обеспечивает разработку приложения на платформе Eclipse 3.0.

11.4.2. Вычисление задач в Cloud computing

Cloud computing — это парадигма распределенного компьютинга. В нем по запросу удаленного пользователя через Интернет предоставляется набор абстрактных виртуализированных, дистанционно управляющих машин, приборов памяти, платформ и сервисов. Основная идея Cloud computing заключается в том, что клиент не должен создавать свой вычислительный центр и обучать персонал установке приложений и обслуживанию системы[1].

С помощью облачных средств (WCloud, Azure, Amazon, Mech, Wapps) фирм IBM-VSphere и Microsoft могут объединяться виртуальные компьютеры для организации вычислений задач. На рис. 11.5 приведена технологическая схема распределенной обработки разнородных программ в сетевой среде программ и систем сервера «Системы» и хранилища данных сервера «Хранилища», которые занимают центральное место в сетевой среде.

---

[1] *Лаврищева К. М.* Базовые основы индустрии программ, вычислений и данных // Проблемы программирования (Спецвыпуск конференции УкрПРОГ—2012). 22—24 мая 2012 г. С. 57—68.

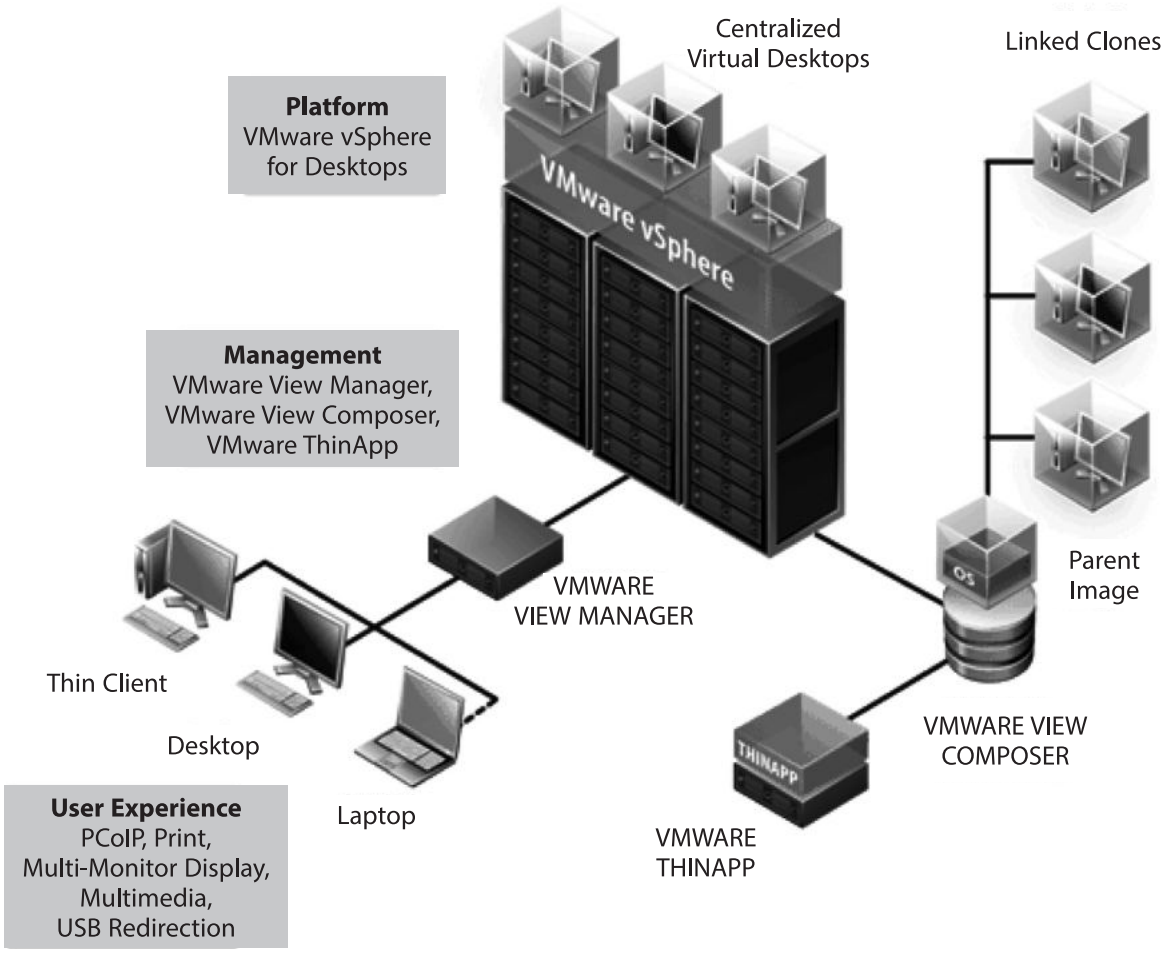


Рис. 11.5. **Схема вычисления программ в среде VSphere**

Исходя из того, что эти серверы поддерживают вычисления программ по данным, которые находятся в хранилищах данных, подобная среда вносит новую виртуальную прослойку между Интернетом и средствами Cloud computing. Этот слой предоставляет сервис по обработке данных больших размеров в режиме онлайн с виртуальных серверов Web-services, Sky-driven (www.cmswire.com), которые обеспечивают будущую виртуальную обработку данных.

Такое направление получило название облачных вычислений задач глобального типа (http://lenta.ru/articles/2010). Эти средства ориентированы на сохранение, синхронизацию данных больших размеров и доступ к глобальным хранилищам данных онлайн, расположенных на удаленных серверах.

Организация облачных вычислений требует реализации новых методов взаимодействия готовых ресурсов и сервисов (например, через конфигурационный файл для решения научных задач).

Одним из практических решений обеспечения вычислений является появление системы Grid в 2005 г. в рамках Европейского проекта. Эта система была разработана как инфраструктура для глобальных решений суперсложных задач из области *e-science*. Подсистема Etics в Grid поддерживает репозитории программ и данных, выполняет сборку готовых КПИ, программ, модулей, систем в конфигурационный файл для выполнения вычислений.

В рамках Grid проведено описание спецификации КПИ для дальнейшего их представления в глобальном репозитории. В ней разработана общая модель вычислений как развитие модели взаимодействия программ для обеспечения интероперабельности в конфигурационном файле разнородных программ и использования данных из on-line Cloud. Для вычисления таких задач применяются вновь разработанные КПИ и системные ресурсы вида: системы хранилищ данных; средства коммуникации; фонды reuses; общие инструменты (конверторы, генераторы, трансформаторы, верификаторы и т.п.); модели и схемы взаимодействия программ в среде гетерогенных платформ, географически расположенных в удаленных точках.

К **системам облачных вычислений** относятся системы Google Apps, IBM-VSphere, Microsoft — WCloud, Azure, Amazon, Mech, WApps, SkyDriven и др.

Функции некоторых систем рассматриваются ниже. Google Apps обеспечивает вычисление веб-приложений с хранилищами данных через интерфейс API в языках Java и Python с помощью стандартных протоколов. Amazon Machine Image (AMI) поддерживает применение библиотеки с инструментами хранения AMI и образов в хранилище, выбор ОС для запуска и контроля нескольких AMI с использованием веб-сервиса, инструментов управления и платой за предоставленные для употребления ресурсы, такие как время и количество переданных данных. Azure (Windows Cloud, 2008) используется для создания распределенных облачных веб-приложений и интернет-сервисов в технологии .Net.

Amazon Web Services — это инфраструктура для оказания следующих услуг: хранение данных (файловый хостинг, распределенные хранилища данных), предоставление вычислительных мощностей и т.п. Причем время вычисления стоит 12 центов в час, гигабайт данных на сервере — 15 центов, каждые 10 тыс. транзакций — 10 центов. Эти системы ориентированы на хранение данных, доступ к глобальным хранилищам данных онлайн, синхронизацию данных больших размеров и передачу их на удаленные серверы. Здесь главная проблема организации вычислений требует определения новых методов, таких как координация, кооперация и взаимодействие различных сервисов и других ГоР через конфигурационный файл для выполнения вычислений соответствующих задач. Накопления ГоР в различных глобальных хранилищах для доступа к ним различных научных пользователей нуждаются в развитии индустриальных методов и моделей с учетом особенностей и специфики научных задач. Эта индустрия тоже базируется на готовых обычных компонентах (артефактах, reuses, assets и данных) многоразового использования, находящихся в различных библиотеках, репозиториях и новых «облачных» хранилищах Интернета.

Главным методом индустрии научных продуктов будет усовершенствованный метод сборки разнородных научных ресурсов, что относится к направлениям *e-science* в структуры ПП, после изготовления они подаются на разные глобальные серверы для их вычисления. Для облачных вычислений основной проблемой является неравномерность запроса ресурсов со стороны клиента(-ов). Для сглаживания этой неравномерности в плане предоставления сервиса между собственно железом и промежуточ-

ным слоем middleware помещается еще один слой — виртуальный сервер, который выполняет функции виртуализации и балансировки нагрузок ПО и распределения виртуальных серверов по реальным платформам и компьютерам.

Сервисы SkyDriven и **Mech** взаимодействуют с облаком в целях выбора необходимого сервиса. Облачное хранилище является базисом при работе сервисов. Пользователи смогут иметь различные объемы данных из хранилищ, а также создавать резервные копии данных. Наряду с таким механизмом доступа к облачным данным появляются новые беспроводные технологии типа WiMAX подобно беспроводному Интернету. Веб-технологии дают значительные возможности для хранения данных и доступа к ним (Amazon, Flash, HTMLS и т.п.).

Сервис приведенных данных онлайн-хранилищ включает:

- огромное количество накопленных данных с новым видом доступа к ним средствами Amazon, Flash, HTML5 из любой точки на планете;
- синхронизацию данных средствами Windows Live Mech, SkyDriven и др.;
- передачу данных на удаленные серверы и резервирование различных копий данных (Office WebApps);
- использование новых веб-технологий (Flash, Silverlight, Amazon);
- композитор заданий, работу с БД (Windows Azure, MS SQL Services, Live Services).

Таким образом, облачные средства дают пользователю разнообразные возможности по накоплению данных и программ, а также решению разнообразных задач.

11.4.3. Язык WSDL

Для описания общедоступных веб-ресурсов используется язык WSDL на основе XML. Среда программирования Eclipse позволяет автоматически создавать описания на основе классов Java. В языке определены следующие основные ТД:

- строки (xsd:string);
- целые числа (xsd:int, xsd:long, xsd:short, xsd:integer, xsd:decimal), числа с плавающей запятой (xsd:float, xsd:double);
- логический тип (xsd:boolean);
- последовательности байтов (xsd:base64Binary, xsd:hexBinary);
- дата и время (xsd:time, xsd:date, xsd:g);
- объекты (xsd:anySimpleType).

В качестве переменных для сообщений могут использоваться последовательности, созданные из фиксированного количества переменных простых типов.

Типичный WSDL-файл имеет следующую структуру.

```
<wsdl:definitions [.]>
<!-- Декларация типов, используемых в сервисе →
<wsdl:types>
<element name="someMethod">
<complexType>
```

```
<sequence>
<element name="arg0" type="xsd:double"/>
<element name="arg1" type="xsd:boolean"/>
</sequence>
</complexType>
</element>
<element name="someMethodResponse">
<complexType>
</wsdl: types> …
```

Приведенное WSDL-описание определяет веб-сервис MyService с единственным методом String someMethod(double arg0, boolean arg1). На его основе можно сгенерировать два ТД, которые отвечают входным и выходным аргументам метода. Эти типы применяются в описаниях someMethodRequest и someMethodResponse — входного и выходного сообщений для операции someMethod.

Операции декларируются в описании интерфейса сервиса (декларация wsdl:portType) и дальше в описании привязки сервиса к SOAP (декларация wsdl:binding), причем во втором случае также оговаривается способ вызова (<wsdlsoap:body use = “literal”/>). За счет этого при вызове операции используются те же названия параметров, что и в методе класса. В конце WSDL-файла находится декларация веб-сервиса (<wsdl:service>), в которой содержится информация относительно его расположения (параметр *location*).

11.4.4. Языки RDF и OWL

RDF — язык представления метаданных о веб-ресурсах в WWW, их сущностях, идентифицированных средствами *веб-идентификаторов* (URI), о данных приложений, которые обмениваются данными в интернет-среде. Каждый ресурс описывается в терминах *свойств* и их *значений*. Простые утверждения относительно ресурсов задаются в графическом виде с помощью вершин и дуг. Для представления графической информации в RDF задается синтаксис описания с помощью XML (RDF/XML) и URI, который связывает любые части информации в Веб.

Вершинами графа могут быть субъекты и объекты. В качестве субъекта может выступать предикат, содержащий свойства. Объект может выступать в роли субъекта в другой тройке графа RDF. Объединение троек (субъект, объект, свойство) в графе означает их конъюнкцию. Квалифицированное имя Qname в XML предназначено для записи идентификаторов в тройках и RDF/XML через *префикс*, которому присваивается URI пространства имен и *локальное имя*. Иными словами, URI образует словарь терминов, а совокупность взаимосвязанных терминов образует URIref как общие префиксы URI.

URIref дает возможность отличать свойства, определяемые одним человеком, от свойств, определяемых другими (или в другом контексте). Например, name может означать Ф. И. О. человека, или название фильма, или имя переменной в некоторой программе. Так как свойства являются ресурсами, то о них можно записать дополнительную информацию (напри-

мер, описание на русском языке того, что именно подразумевается под свойством «name») — для этого добавляется предложение RDF, в котором свойство в URIref выступает в качестве субъекта. Значения свойств могут быть структурированными и содержать информацию типа: город, штат, улица, почтовый код.

RDF — язык описания данных Интернета (а не документов), обеспечивает уникальную иденитификацию всех сущностей языка в Интернете. Ресурс описывается в виде субъекта, предиката и объекта, задается тремя типами вершин — URIref, литералы, пустые вершины. Субъект — это URIref или пустая вершина. Предикат — URIref. Объект — пустая вершина или литерал. RDF задает бинарные связи. Для задания *n*-арных связей вводится дополнительная сущность, которая разрешает встроенные ТД (исключая URIref rdf:XMLLiteral) для представления литерального значения XML-контента.

Концепция ТД RDF базируется на концепции ТД XML Schema, включает лексическое пространство, пространство значений и отображение «лексика — значение».

Пример в XML Schema типа данных *xsd:boolean*:

Value Space {T, F}
Lexical Space{“0”, “1”, “true”, “false”}
Lexical-to-Value Mapping {<“true”, T>, <“1”, T>, <“0”, F>, <“false”, F>}.

Данные, представленные с помощью литерала, могут быть представлены и с помощью URI. Литералом могут быть только объект и предикат. Простой литерал — это строка с факультативной меткой языка, а типизированный литерал — это строка с URI ТД со значением из пространства значений ТД.

RDF предоставляет методологию и схему для определения модели данных объектов и отношений на базе XML синтаксиса и механизма XML-схем (http://www.w3.org/TR/2002/WD-rdf-schema-20021112).

RDF Schema на базе принципов RDF определяет модель для построения простых справочников и тезаурусов (иерархических справочников).

OWL добавляет еще больше конструкций для определения модели предметной отрасли, в том числе — множество классов сущностей и их свойств, правила отношения сущностей к классам, определения порожденных классов и свойств на этих множествах. Стандартом предусматривается три версии (уровня) языка. Соответственно, ПО, которое поддерживает более низкие уровни, может быть легче и быстрее реализовано и быть менее ресурсоемким. OWL Light берет за основу конструкции RDF schema и расширяет их для детального определения онтологии в справочниках и тезаурусах предметных отраслей[1].

---

[1] *Lavrischeva E.* Ontological Approach to the Formal Specification of the Standard Life Cycle // Science and Information Conference — 2015, Jule 28—30, London, UK. P. 965—972; URL: www.conference.thesai.org; *Lavrischeva E. M.* Ontology of Domains. Ontological Description Sofware Engineering Domain — The Standard Life Cycle // Journal of Software Engineering and Applications. 2015. July 24. URL: http://www.scirp.org/journal/jsea.

OWL DL расширяет Light конструкциями для определения порожденных классов, свойств на базе разных операций над ними в соответствии с положениями дескриптивной логики, которая является формальной основой OWL DL.

OWL Full использует те же самые конструкции, что и DL, снимает некоторые ограничения, которые налагаются в DL на ТД, отношения и свойства, что значительно расширяет языковые возможности OWL.

11.5. Языки описания систем и протоколов Интернета

Под **ресурсом** понимается некий **информационный объект** (БД, прибор, который регистрирует данные) в некотором состоянии, доступ к объекту возможен с помощью одного или нескольких веб-сервисов.

Объединение ресурса сохраняет состояние и веб-сервис, через который к нему имеется доступ, и составляет понятие WS-ресурса, который имеет определенный ЖЦ и характеризуется набором свойств. WS-ресурс похож на удаленный объект в CORBA.

Объект в ООП

WSRF сделан поверх SOAP и WSDL и представляет собой XML-схему для определения WS-ресурсов, их свойств и протокола ЖЦ WS-ресурса.

Информационные системы компонуются из большого количества взаимодействующих WS-ресурсов. WSRF определяет механизмы согласованного выполнения ресурсов с помощью специализированных веб-сервисов, обеспечивающих поддержку и передачу контекста выполнения (WS_CONTEXT), механизма транзакций (WS-TX) уведомлений о событиях (WSNotification), обмена сообщениями (AMPQ advanced message protocol Que) и т.п.

Для выполнения прикладных задач нужно выполнять отдельные операции в определенной последовательности, причем эта последовательность может зависеть от хода выполнения предыдущих шагов процесса (*workflow*). Единственное (стандартное) представление схемы выполнения процесса из отдельных операций (вызовов сервисов) на всех этапах ЖЦ дает возможность взаимодействовать разным специалистам, используя соответствующие инструменты, которые поддерживают эти стандарты, независимо от платформы и производителя ПС. Стандарт OASIS WS — BPEL определяет спецификацию процессов и их выполнение. Он использует XML и расширяет пространство имен WSDL для описания поведения процессов в распределенной среде посредством веб-сервисов. WS-BPEL добавляет механизмы и XML-конструкции для определения участников, условий и порядка вызова сервисов.

Определение переменных для сохранения сообщений и выполнения операций над элементами сообщений задает:

- вызов сервисов;
- ожидание и получение результатов от условного выполнения ветвей процесса;
- циклическое выполнение ветвей процесса.

Стандарт предусматривает два языка (соответственно, два пространства имен) — абстрактный и тот, что выполняется.

Абстрактный уровень имеет конструкции, посредством которых можно определить принципиальную схему выполнения процесса без ухода в детали реализации отдельных операций (вызовов сервисов) с точки зрения аналитика. Абстрактный WS-BPEL не может быть выполнен, поскольку не содержит детальной спецификации сервисов.

Уровень, который выполняется, содержит конструкции, позволяющие добавить детальные спецификации вызовов сервисов, которые уже могут быть определены и выполнены.

WS-BPEL не предусматривает взаимодействия с человеком, потому что все равно нужно реализовывать интерфейс пользователя в HTML. Спецификация WSRP предлагает перенести принципы SOA и на разработку интерфейса пользователя. Так же как BPEL компонует сложный процесс из отдельных строительных блоков (веб-сервисов), так и WSRP обеспечивает компоновку страницы представления результатов и взаимодействия с пользователем.

Интернет-протоколы

Взаимодействие в сети Интернет и в большинстве локальных сетей построено на основе протоколов TCP/IP, которым соответствует 4-уровневая модель семейства протоколов Министерства обороны США. В рамках ISO разрабатана 7-уровневая модель, которая претендует на роль стандарта OSI.

Протоколы TCP/IP поддерживаются комитетом ISOC и получили всеобщее признание в модели DOD, в которой прикладной уровень объединяет сеансовый, прикладной и уровень представления OSI. Каждый уровень протоколов соответствует определенной фазе передачи данных.

Протоколы физического уровня определяют уровни сигналов в передающих средах, а также их соответствие логическим единицам данных. Примеры — bluetooth, Wi-Fi, rs232 arinc и др.

Канальный уровень определяет порядок упаковки данных в пакеты формирования и последующую проверку контрольных сумм восстановления сбоев в процессе передачи кадров.

Сетевой уровень определяет порядок нахождения физических адресов по логическим именам узлов и кратчайших маршрутов и соответствует аппаратным или программным маршрутизаторам сети.

Транспортный уровень определяет порядок склейки целостного масива данных из отдельных кадров, в том числе и повторный запрос сбойных кадров. С помощью этих протоколов происходит общение пользователя с Семантическим Вебом.

Контрольные вопросы и задания

1. Назовите основные программные компоненты Веба.
2. Приведите короткое назначение Семантического Веба.

3. Назовите языковые элементы семантического Веба.
4. Как представляется онтология в Вебе?
5. Приведите стандарты консорциума W3C.
6. Определите основные элементы веб-технологий.
7. Приведите основное назначение языка процессов BPMN.
8. Перечислите средства моделирования бизнес-процессов.
9. Определите типы объектов, событий и задач.
10. В чем сущность семантического моделирования процессов?
11. Приведите процессы ЖЦ процессов Веба.
12. Определите назначение моделей сервисов SOA и SCA.
13. Определите цель языков WSDL, RDF и OWL.
14. Назовите протоколы Интернета.

Глава 12
ОСНОВЫ SOFTWARE ENGINEERING METHOD AND THEORY

Созданное ACM (*Association for Computing Machinery*) и институтом инженеров по электронике и электротехнике IEEE Computer Society ядро знаний SWEBOK содержит разнородные знания в области инженерии программ, планирования и управления процессами. Эти знания представлены в 10 специализированных областях SWEBOK, которые соответствуют процессам проектирования ПО и методам их поддержки. ПИ рассматривалась как инженерная дисциплина, в которой инженеры и программисты применяют теоретические идеи, методы и средства при разработке ПО, создают продукты в соответствии со стандартами, регламентирующими процессы их проектирования, разработки и оценки качества продуктов (www.swebok.com).

Как инженерная дисциплина она охватывает все аспекты создания ПО, начиная от формирования требований и до создания, сопровождения и снятия с эксплуатации ПО, а также включает инженерные методы оценки трудозатрат, стоимости, производительности и качества. Инженеры в ПИ — это специалисты, выполняющие практические работы по реализации программ и систем.

Между тем компьютерная наука (*CS*), в состав которой входит SE, охватывает теорию и методы построения вычислительных и программных систем, тогда как ПИ рассматривает вопросы практического построения ПС по разработанным стандартам (ISO/IEC 12207, 15504, ISO/IEC 9126 и др.). Знание компьютерной науки необходимо специалистам в области ПО так же, как знание физики — инженерам-электронщикам. Если для решения конкретных задач программирования не существует подходящих методов или теории, инженеры применяют свои знания, накопленные ими в процессе разработок конкретных ПО, а также используют опыт применения соответствующих инструментальных программных средств. Инженеры, как правило, работают в условиях заключенных контрактов и выполняют задачи проекта с учетом этих условий и ограничений по срокам, времени, стоимости и др.

Кроме инженеров разработкой систем занимаются:

- аналитики, постановщики задач и менеджеры, которые планируют проект и руководят им, отслеживают сроки и затраты;
- инженеры службы аннотации и хранения готовых компонентов в библиотеках и репозитариях;

• технологи, которые определяют необходимые инженерные методы и стандарты, создают соответствующую для данного проекта модель ЖЦ, удовлетворяющую целям и задачам проекта;

• тестировщики (контролеры), которые проверяют правильность выполнения процесса проектирования и продуктов процессов, на основе собранных данных проводят измерения разных характеристик качества, включая оценку надежности ПО;

• верификаторы, проверяющие правильность реализации функций в проекте;

• валидаторы, проверяющие ПО на соответствие заданным требованиям;

• специалисты, сертифицирующие готовые системы, и др.

Современные подходы к внесению изменений в функциональность сложной готовой или разрабатываемой системы, как правило, являются интуитивными, ручными и выполняются системными специалистами или аналитиками. Любое внесенное изменение требует уточнения архитектуры системы или, соответственно, конфигурационного файла, а также проведения тестирования на множестве тестов. На поиск и исправления тратится много времени. Бессистемные, хаотически вносимые изменения усложняют систему и требуют документирования возникших ошибок, а также времени их исправления.

Как было сказано выше, в SEI USA разработан индустриальный метод производства программных продуктов — Product Line / Product Family путем сборки готовых продуктов (программ, компонентов, reuses, assets и др.) по заказу потребителей. Именно в нем создана новая концепция понятия изменяемости и управления различными их вариантами на основе FM.

Для поддержки новых методов SE разработано огромное количество разнообразных инструментальных средств (UML, Rational Rose, Com, CORBA и др.), а также стандарты (ISO/IEC 12207, 15504, ISO/IEC 9126 и др.) разработки ПС на основе ЖЦ ПО и оценки качества и стоимости разработки ПП. Все это способствует повышению эффективности проектирования, тестирования, прогнозирования надежности и оценки качества ПО. Основные из них рассмотрены в гл. 11.

Начиная с 2005 г. шли обсуждения теоретических вопросов разработки ПП и систем[1]. В указанных статьях говорилось не только о необходимости теории, которая давала бы толкование разным практическим аспектам инженерии ПО. В результате появилось новое направление — SE Methods and Theory (SEMAT).

Основным теоретическим наработкам в SE, а также целям и задачам этого нового направления SEMAT посвящена данная глава.

---

[1] См.: *Лавріщева Е. М.* Классификация дисциплин программной инженерии // Кибернетика и системный анализ. 2008. № 6. С. 3–9; *Lavrischeva E. M.* Software engineering. New disciplines and E-learning them for development of applied systems // Progressive Academic Publishing. European Journal of Engineering and Technology. 2015. Vol. 3 (3). P. 36—63. URL: http://www.idpublications.org/ejet–vol–3–no–3–2015; *Lavrischeva E. M.* Classification of software engineering disciplines // Springer. 2008. Vol. 44. № 6. P. 791—796.

12.1. Основные теоретические достижения SE

В работе М. Броя «Теория и практика программной инженерии» (Открытые системы. 2011. № 2) написано: «...теория помогает инженерам создавать и изучать методы, оценивать результаты и оптимизировать процессы... Исследователи должны создавать, адаптировать и развивать научные методы программной инженерии». Он считает, что главная задача исследователей в SE заключается в формализации методов разработки систем из разных элементов и процессов их реализации, как это сложилось в инженерии компьютеров. В области формальних методик SE могут произойти открытия в области формализации, верификации и доказательства систем.

Начиная с 2004 г. и программы «Curricula Software Engineering — 2004», созданной комитетом ACM, началось обучение техническим, инженерным вопросам SE. Студенты получали навыки применения технических методов разработки систем, а теперь наступило время, когда в обучение включаются существующие теории, и студенты должны получать теоретические знания в математической логике, а также в новых формализованных теориях с учетом инженерных методов.

К основным направлениям теории SE автора в период 2003—2015 гг. относятся: новые дисциплины (теория, инженерия, управление, экономика, индустрия) SE, онтологическая теория представления ЖЦ на примере домена ISO/IEC 12207, объектно-компонентный метод моделирования ПС с помощью логико-математического апарата и методы прогнозирования и оценки качества ПС, а также метатехнология изготовления линий производства программ в рамках СОД и работ USA. Далее даются основные положения названных теорий[1].

12.1.1. Определение новых дисциплин в SE

За период с 1968 по 2008 г. автором настоящего учебника сформирована классификация общих формальных дисциплин SE, которые определяют разные аспекты теоретической деятельности по производству ПП, включая научные, инженерные, управленческие, экономические вопросы. В каждой дисциплине определены базовые теория и подходы к их использованию при проектировании ПС, а также их преподавания в составе курсов SE (рис. 12.1).

Наука программирования

Основу этой дисциплины составляют классические науки (теория алгоритмов, теория множеств, теория доказательств, математическая логика и т.п.), теория программирования, теория проектирования на уровне абстрактных моделей и архитектур целевых программных объектов. Она содержит базовые понятия и объекты, формальные теории, методы программирования и методы управления изготовлением вариантов ПП.

---

[1] *Лаврищева Е. М.* Классификация дисциплин программной инженерии // Кибернетика и системный анализ. 2008. № 6. С. 3–9; *Ее же*. Программная инженерия — научная и инженерная дисциплина // Кибернетика и системный анализ. 2008. № 3. С. 19–28; *Ее же*. Software engineering as a scientific and engineering discipline // Springer. 2008. Vol. 44. № 3. P. 324—332; *Ее же*. Classification of software engineering disciplines // Software–Hardware Systems. Springer. 2008. Vol. 44. № 6. P. 791—796.

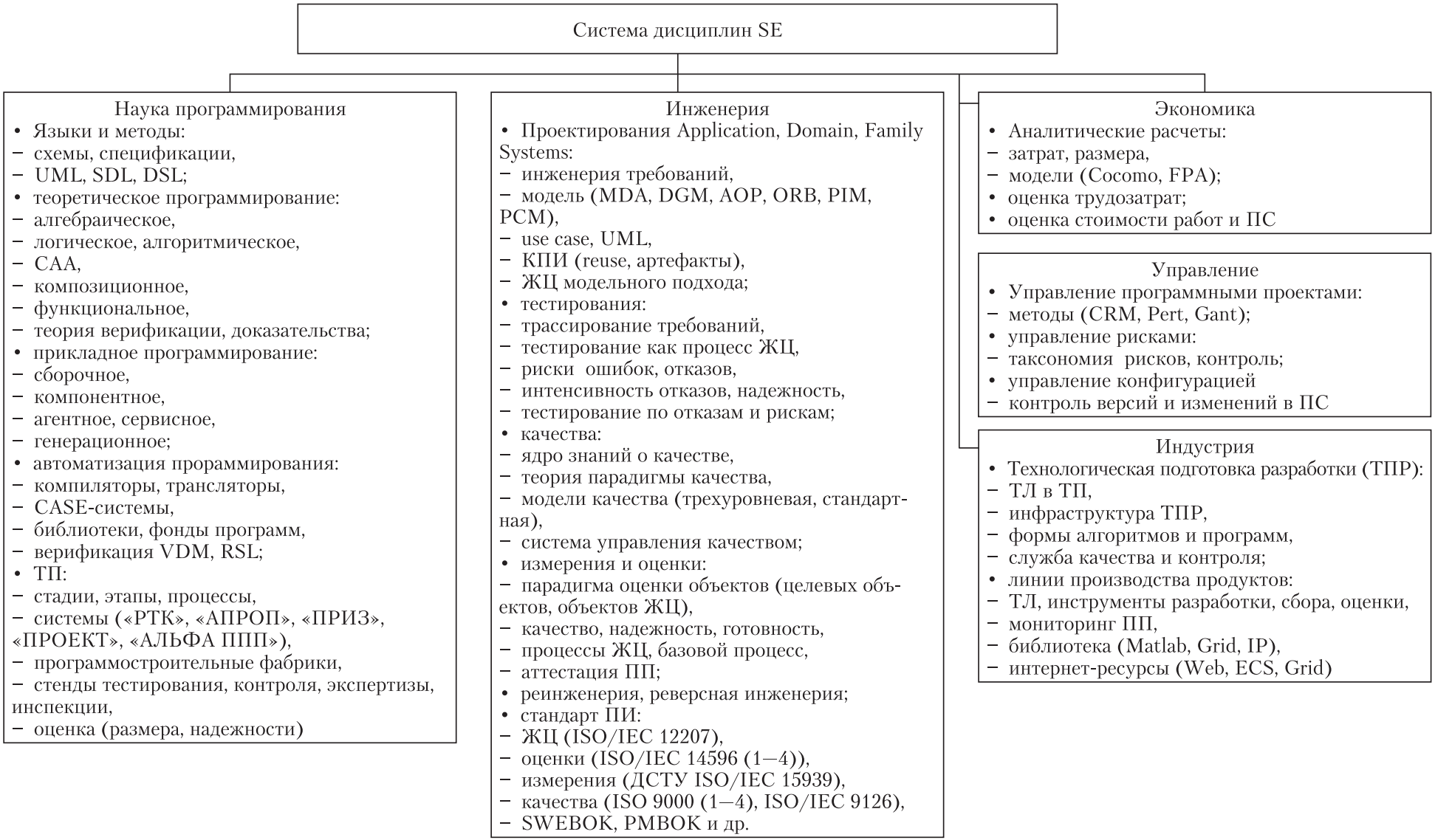


Рис. 12.1. **Новые дисциплины SE и их структура**

Основные понятия SE составляют типы и структуры данных, функции и композиции, простые и сложные целевые объекты. Разработка простых объектов выполняется посредством формальной их спецификации и инженерии реализации с применением инженерных методов проектирования и организации управления процессами их изготовления.

Теория программирования является фундаментом научной дисциплины, который включает:

- теорию разработки схем программ, доказательства их эквивалентности, формальные языки спецификации и проектирования целевых объектов, методы доказательства их правильности (верификация, валидации и тестирование);
- формальные методы управления проектом и отдельными его характеристиками;
- методы оценивания заданных показателей качества ПП (надежность, корректность и т.п.);
- теорию изменения вариантов программ и их семейств.

Таким образом, научная дисциплина является теоретическим фундаментом SE и учить ей необходимо не только для повышения уровня квалификации и развития новых возможностей теории программирования в индустрии ПИ. Одна из важных научных проблем — это интеграция (композиция, синтез, сборка) составных элементов будущего продукта, основанная на совместимости их интерфейсов. Эта теория опирается на фундаментальную теорию синтеза как одну из ветвей научной дисциплины SE и индустриальную сборку ПП.

Главным курсом обучения в вузах должна стать именно научная дисциплина, которую необходимо представить общим теоретическим курсом программирования (объектно ориентированного, компонентного, агентного и т.п.), отдельными действующими классическими курсами и дополнительными курсами из программы Curricula-2008, например «Инженерия качества и безопасность ПП».

Инженерная дисциплина

Инженерная дисциплина определяет совокупность инженерных приемов, средств и стандартов, ориентированных на изготовление целевых объектов ПП с применением научной дисциплины. Сущность этой дисциплины составляют:

- ядро знаний SWEBOK как набор методов и средств разработки ПП и управления проектами;
- базовый процесс ПИ как стержень процессной деятельности разработки ПП;
- стандарты как набор регламентированных правил конструирования артефактов на процессах ЖЦ;
- инфраструктура — условия среды, методическое и организационное обеспечение базового процесса ПИ и поддержки деятельности исполнителей ПП;
- общие системные средства и инструментальные среды поддержки процессов изготовления ПП.

Технология инженерного изготовления ПП базируется на КПИ, средствах и инструментах их построения. К таким технологиям относятся: инженерия КПИ (*Reuse Engineering*), инженерия приложений (*Application Engineering*), доменов (*Domain Engineering*) и семейство систем (*Family of Systems Engineering*).

Инженерия ПИК сформировалась как систематическая и целенаправленная деятельность по поиску и подбору готовых ПИК, которые размещаются в современных хранилищах (репозиториях или библиотеках). Базис изготовления из них ПП — каркас, набор вновь реализованных и готовых компонентов, функциональных ПИК. В новых ПП могут использоваться готовые прикладные системы (например, бизнес, коммерция, экономика и т.п.), а также системы общего назначения (трансляторы, редакторы, ОС, СУБД, системы интеграции, генерации и т.п.).

Инженерия приложений и *инженерия доменов* также основываются на многократном использовании разных КПИ и других программных элементов. Основная задача этих видов инженерной деятельности — построение прикладных систем или семейств систем, которые реализуют фунции приложения или домена с учетом общих и изменяемых характеристик составляющих их элементов (членов семейства). Технология изготовления ПС вплотную подошла к современным принципам конвейерного производства продуктов из готовых «деталей» типа КПИ по модели домена в DSL (*Domain Specific Language*) и спецификациям каждого члена семейства (см. гл. 3). Основная суть этой технологии — управление изготовлением ПП, основанное на план-графиках работ, контроле результатов работ и оценивании степени применимости ГоР в процессе реализации специфических задач домена.

Базовые компоненты данной инженерной дисциплины должны непрерывно совершенствоваться и адаптироваться к новому типу целевых объектов и условиям производственной среды (что в духе концепций совершенствования, заложенных в моделях CMM, SPICE, Trillium и др.).

Значение инженерной дисциплины в производстве первостепенно. Без инженерии не мыслится ни один промышленный продукт. Здесь нужно тщательно исследовать все наработки (как научные, так и инженерные) в области компьютерных наук и на их основе создать математический фундамент инженерии ПП, которая включает стандартные принципы инженерии и базовых компонентов, а также языков спецификации доменов, членов семейств и процессов производства ПП средствами и инструментами инженерных технологий.

Дисциплина управления

Базис этой дисциплины — классическая теория управления, менеджмент производства проектов и стандарт IEEE Std.1490 PMBOK (*Project Management Body of Knowledge*). Теория управления, в частности теория организационного управления, разработана академиком В. М. Глушковым (1970). Эта теория проверена практикой при построении технологических процессов в металлургической, судостроительной и химической промышленности, а также при внедрении для целей массового производства (например, АСУ «Львов»).

К теории управления сложными системами относятся: метод CRM (*Critical Path Method*), базис которого — графическое представление работ, соответствующих видов операций и времени их выполнения; метод сетевого планирование PERT (*Program Evaluation and Review Technique*) был апробирован при реализации проекта ракетной системы «Polaris». Каждый из этих методов возник в недрах промышленного производства. Они адаптированы к среде программирования и стали базовыми в индустрии программных продуктов.

Элементы теории управления и планирования нашли отображение в стандарте PMBOK. В нем определены процессы ЖЦ проекта и главные области знаний, сгруппированные по таким задачам, как инициация, планирование, мониторинг, управление и завершение. Основная область знаний этого ядра — интеграция — включает концепцию управления организационной деятельностью коллектива исполнителей проекта. Она базируется на методах принятия решений, касающихся ресурсов, общих задач проектирования, служб контроля правильности проекта и вкладывания в заданную заказчиком стоимость проекта.

Эти базовые наработанные теории управления и планирования, стандартные положения PMBOK, серии стандартов ISO 9001, регламентирующих управление качеством, и соответствующее методическое обеспечение проекта должны стать базисом дисциплины управления в ПИ. Сформированный курс обучения этой дисциплине будет обеспечивать подготовку в вузах будущих высококвалифицированных менеджеров проектов и других специалистов в области организационного управления выпуском программной продукции.

Экономическая дисциплина

Экономика ПП является самостоятельной дисциплиной и связана с экономическими аспектами индустрии ПП. В ее основе лежит метод проведения экономических расчетов разных сторон деятельности на программном проекте с учетом знаний специалистами всех факторов и затрат в проекте. Эта дисциплина имеет свою теорию и практику решения задач по проведению экспертизы проекта, оценке стоимости, сроков и экономических показателей, устанавливаемых в требованиях к ПП при заключении контракта на его разработку. В рамках этой дисциплины проводятся оценка требований, проектных решений, архитектуры, рисков разработки, связанных с имеющимися материальными и человеческими ресурсами, показателями качества ПП, а также финансовые расчеты на всех этапах договора с каждым исполнителем.

Эта дисциплина наиболее развита с точки зрения методов экономических расчетов в ПИ, а именно: наличия методологий прогнозирования размера ПП (FPA — Function Points Analyses, Feature Points, Mark-II Function Points, 3D Function Points и др.), оценки трудозатрат на разработку ПП с помощью семейства моделей COCOMO, ряда других математических моделей оценки трудозатрат на разработку ПП (Angel, Slim, Seer-SEM и др.), а также моделей, связывающих экономические показатели ПП с характеристиками качества.

В состав этой дисциплины должны входить фундаментальные экономические методы, связанные с принципами распределения и экспертизы работ в сложных системах, методы расчета стоимости отдельных частей систем в зависимости от размера ее составляющих и системы в целом, существующие стандарты, обеспечивающие оценку и сертификацию готового продукта, и т.п. Систематизированный и научно обоснованный курс экономической дисциплины закроет тот пробел, который имеется в SE из-за отсутствия соответствующих пособий и учебников для обучения специалистов, которые будут заниматься экономическими задачами в индустриальном цикле производства ПП.

Производственная дисциплина SE

Главным вопросом индустрии, как таковой, является не только выпуск программной продукции, но и получение прибыли. В области SE продукты массового производства, создаваемые известными фирмами Microsoft, IBM и др., а также результаты аутсорсинга (обновление устаревшего, унаследованного ПО), приносят владельцам большие *прибыли*. Этим подтверждается (в соответствии с толкованием понятия «производство»), что виды ПП этих фирм выпускаются на *индустриальной* основе. Производство ПП базируется на технологических процессах изготовления определенных видов продуктов с применением теории проектирования и инструментальных сред поддержки выпуска ПП.

Первыми попытками индустриального производства являются технологическая подготовка разработки ПП (ТПР), линия продукта (*Product line*) института SEI США, обеспечивающая удовлетворение рыночных потребностей пользователей на некоторые виды программной продукции. Более продвинутые инженерные технологии производства ПП — это инженерия приложений, доменов, семейств систем, а также средства поддержки их производства (ОС, общесистемные средства, новые языки, интегральные среды и т.п.).

ТПР была применена при разработке АІС «Юпитер» для производства программ обработки данных на нескольких объектах Военно-морского флота СССР. ТПР не развивалась последние годы из-за отсутствия такого рода систем. Производство ПП на упомянутой линии продуктов осуществляется из готовых программ, информационных ресурсов, КПИ и инструментов ТЛ, в которую включаются необходимые методы разработки, тестирования и оценивания конечного результата. Технология конструирования на такой линии выполняется с помощью каркаса ПП и применения подобранных КПИ. Инструментальная среда их разработки содержит необходимые средства и инструменты производства, а также механизмы отслеживания хода построения продукта в соответствии с установленным заказчиком планом.

На сегодня в индустрии ПП не решены проблемы, касающиеся сложности объектов, процессов изготовления вариабельных ПП. Мало работ, в которых предлагаются пути преодоления сложности, в особенности при интеграции больших проектов из разных простых и готовых ресурсов, а также при их сопровождении и внесении разного рода изменений.

Данная дисциплина как предмет обучения должна включать классические методы и технологии производства разных видов продуктов, методы анализа сложности структуры, средства описания специфических особенностей целевых объектов, оценки ГоР и особенностей инструментальных сред, языков спецификации этих объектов, стандартных положений по производству и документированию готового продукта.

12.1.2. Теоретические аспекты онтологии ПП

При проектировании систем используются модели ЖЦ и стандарт ISO/IEC 12207, являющийся общим средством регламентированного построения разных ПС.

Для описания МХ процессов ЖЦ существуют методологии доменного анализа. Среди них наиболее известны ODM (*Organization Domain Modeling*) и FODA (*Feature-Oriented Domain Analysis*). Для анализа некоторых доменов используется модель DSSA (*Domain-Specific Software Architectures*), в которой задается *МХ*, содержащая общие характеристики программных компонентов и процессов ПС. Отличительной особенностью представления процессов ЖЦ является *диаграмма* зависимостей между характеристиками. Концепция таких диаграмм унаследована из метода FODA. В ней описываются все возможные конфигурации процессов из разных категорий стандарта ЖЦ. Каждый из них рассматривается как *экземпляр* или *образец* (*instances*). Нотация диаграмм характеристик процессов выполнятся языком FDL (*Feature Definition Language*), позволяющим описывать:

- атомарные и композиционные характеристики, имена которых определяются и используются в разных процессах;
- необязательные (*optional*) и обязательные характеристики (*mandatory*) выражений, которые относятся к замкнутым конструкциям *all*();
- альтернативные характеристики (*exclusive* — выбор) через *one-of* ();
- характеристики по умолчанию (*default*).

Результат трансляции характеристик в FDL может быть выполнен XMI для обмена информацией Metadata (XML Metadata Information Exchange format) и импортирован в систему моделирования UML при генерации классов. Подход к описанию МХ ПрО использован при разработке вариантов ЖЦ ПС и конфигурировании разных процессов. С помощью данной модели генерируются необходимые варианты ЖЦ для реализации определенного класса ПС. Конкретная онтологическая модель ЖЦ описана в гл. 9.

12.1.3. Подход к обучению предмету SE

Все рассмотренные выше дисциплины SE и их базовые теории, по мнению автора, должны стать самостоятельными предметами в вузах, специализирующихся на создании разного рода ПП[1].

---

[1] *Lavrischeva E. M.* Software engineering. New disciplines and E-learning them for development of applied systems // European Journal of Engineering and Technology. 2015. Vol. 3. № 3. P. 36—63. URL: http://www.idpublications.org/ejet-vol-3-no-3-2015.

В плане обучения SE нами было разработано учебное пособие «Методы и средства инженерии программного обеспечения» (2007, МФТИ). В этом пособии, согласно стандарту Curricula (2001, 2004, 2007, 2014) представлены базовые элементы, средства и инструменты инженерии разработки разных ПП, а также процессы ЖЦ, методы проектирования и управления исполнителями, качеством, сроками и стоимостью. Описаны базовое ядро SWEBOK, PMBOK и ряд стандартов, которые используются в инфраструктуре менеджмента проекта.

Ранее были описаны теории, методы и парадигмы разработки изменяемых систем.

К ним относятся оригинальные теоретические разработки автора:

- теория сборочного программирования систем из разнородных языковых модулей, интегрируемых вместе с помощью интерфейса;
- формальная классификация дисциплин ПИ;
- онтологическая теория представления научных и инженерных доменов (ЖЦ стандарта ISO/IEC 12207, домен КПИ и вычислительная геометрия, гл. 9 настоящего учебника);
- теория формальных и общих ТД (FDT, GDT) и некоторые новые механизмы генерации неструктурированных больших данных типа Big Data структур FDT, GDT (гл. 8);
- теория объектно-компонентного моделирования предметных областей с применением математических и логических операций (гл. 7);
- теоретические основы и методы проектирования вариабельных систем на основе МХ, моделей систем и модели конфигурации (гл. 4, 5);
- анализ зарубежных методов формальнай спецификации объектов ПрО, верификации и доказательства правильности с использованием метода Флойда, Дейкстры, Турского и др. (гл. 6);
- парадигмы программирования (модулей, компонентов, сервисов, объектов, аспектов и др.) в виде отдельных самостоятельных элементов повторного использования КПИ, взаимодействующих между собой через интерфейс, определенный автором в 1970-х гг. и широко используемый во всех общесистемных инструментах (IBM, Microsoft, Intel, Apple и др.) и сегодня в фабриках программ.

Приведенные теории составляют базовую основу нового варианта SEMAT.

12.2. Основные сущности SEMAT. Практика, теория, образование

SEMAT является инициативным направлением по изменению ПИ, так чтобы разработка ПО квалифицировалась как строгая математическая дисциплина. Инициатива была предложена в декабре 2009 г. И. Якобсоном, Б. Мейером и P. Soley на основе ряда стандартов SE, например ISO / IEC 14756: 2008. Информационные технологии. Измерение и рейтинговое оценивание производительности программных систем (ISO / IEC 14756: 1999, IDT), ISO / IEC TR 15271: 2010. Информационные технологии. Руководство по применению ISO / IEC 12207 (Процессы ЖЦ ПО) (ISO / IEC TR

15271: 1998, IDT), ISO / МЭК 15939: 2008Инженерия систем и программных средств. Процесс измерения (ISO / МЭК 15939: 2007, IDT).

В начале инициативы учредители написали призыв к действию и продолжили многолетнюю работу по SE, чтобы преодолеть разрыв между сообществом разработчиков и академическим сообществом.

Работа в настоящее время структурирована в четырех смежных областях: практика, сфера образования, теория и сообщество. Практика обращается к практике. Образование затрагивает все вопросы, связанные с обучением. Теория прежде всего занимается поиском общей теории для разработки ПО. В сообществе работают юридические лица, которые создают веб-сайты для вовлечения новых специалистов. В будущем практика, образование и теория будут интегрироваться и теория должна направлять исследования к научным результатам для сообщества теоретиков и практиков. Образованию должны соответствовать новые теории подготовки студентов к специальности инженерии разработки прграммных и прикладных систем.

Область практики

Первый шаг состоит в описании основных понятий сущности программирования — вещи, которые мы всегда имеем, делаем, производим и развиваем ПО. Второй шаг предполагает увеличение стоимости этого ядра, содержащего библиотеки методов, которые будут разработаны, а затем станут методами для всех видов продуктов и ПО. Результат — ядро, включающее универсальные элементы для разработки ПО, называется ядром сущности с языком описания ее элементов (методы, подходы и др.). Сущность и ядро обеспечиваются языком, близким стандарту OMG Beta статуса 2013 г., формально принятым стандартом в начале 2014 г.

Практика разделена на отдельные, но связанные элементы: библиотеки, инструменты, практики. Они помогают студентам и уже работающим, авторам метода и производителям инструментов при использовании. Путеводитель содержит сценарии, демонстрирующие ядро сущности, чтобы оценить выбранные методы и ЖЦ разработки ПП. Сценарии также развиваются, чтобы показать команде, как на практике создаются и обновляются методы разработки продуктов.

Область образования

Область сосредоточивается на обучении программированию. Она способствует глобальному образованию, основанному на соответствующих сущностях. Целевая аудитория — преподаватели, профессора университетов и промышленные тренеры, а также их студенты, изучащие эти сущности.

Цель области состоит в том, чтобы создать образовательные курсы и материалы курса, которые жизнеспособны и определяют педагогические подходы, которые являются эффективными для определенных целевых аудиторий и распространяли опыт и извлеченные уроки.

Область включает участников из многих университетов — Университет Карнеги — Меллон Силиконовая Долина, Свободный университет Больцано Bozen, Королевский технологический институт, Национальный уни-

верситет Колумбии, Университет Дуйсбурга — Эссена, Университет Осло, Университет Васэда, Национальный институт информатики.

Большинство участников уже были вовлечены в разработку аспектов SEMAT в контексте курсов программирования. Они собирают свои ресурсы и начинают общее предприятие для определения нового поколения SEMAT, приведенных в действие учебных планов программирования.

Область теории

Важную часть SEMAT составляет общая теория программирования.

Серия семинаров, проведенных под заголовком «Семинар SEMAT по общей теории программирования (GTSE)», является ключевым моментом этой теории. Она должна быть основана на языке SEMAT и должна поддержать целенаправленное принятие решений для практики программирования. Такая поддержка основана на прогнозирующих возможностях теории. Таким образом, сущность теории SEMAT должна быть расширена, чтобы позволять предсказывать критические события в программировании. Типовые вопросы, на которые теория должна отвечать, включают изменение способа работы, структуру команды, содержание и требования к ПО.

Ряд семинаров GTSE помогают развитию общей теории программирования SEMAT, вовлекая более многочисленное сообщество специалистов в плане поиска, развития и оценки многих теорий, которые могут использоваться в качестве основы для теории SEMAT.

Область сообщества

Российское отделение SEMAT включает около 20 членов, в частности несколько университетов. Проводятся семинары по SEMAT.

Российское отделение занимается вопросами инженерии систем, ядра ЖЦ ПО и др. Создан новый стандарт, принятый компанией OMG. Сущность представляет инновационный и новый опыт пользователей, основанный на оценке прогрессивных идей в разработке ПО. То есть сущность — суперпростая основа ПО, инженерия операций, которые выполняются при выпуске продукции. В результате теория разработки ПО как инженерная дисциплина больше чем ремесло. Требуется работать на новых инструментальных средствах, создавать практики, библиотеки и использовать их в практической жизни разработчиков.

12.3. Задачи повышения компетенции инженеров ПС

Стандарт «The Software Assurance Competency Model» («Модель компетенций уровня доверия ПО», 2014) определяет компетенцию разработчиков, основанную на наличии навыков, необходимых для деятельности при разработке ПС. *Знания* в этой модели компетенции отличаются от *навыков*: знания — это то, что человек знает, а навыки — то, что он может делать.

Модели компетенции служат для управления разработкой ПО, ее руководителями и персоналом, а также проектировщиков учебных программ. Человек, который развивает или поддерживает ПО, должен использовать

эту модель компетенции для оценки своих текущих уровней компетентности для различных видов деятельности SE (таких как выявление требований, проектирование синтеза, конструирование ПО, планирование тестирования и др.).

12.3.1. Назначение модели компетенции специалистов

Менеджер может использовать эту модель, чтобы уточнить квалификацию персонала, проконсультировать отдельных сотрудников, а также определить направления обучения студентов по разным областям знаний SWEBOK и в зависимости от их освоения осуществлять соответствующую деятельность в разработке разного вида систем. Каждая деятельность в этой модели компетенции описывается на одном из пяти уровней компетенции.

Область навыков в данной модели компетенции включает в себя навыки, которые распадаются на деятельность, а не на рабочие роли, потому что рабочие роли, как правило, зависят от организационной среды, в которой осуществляется трудовая деятельность.

Данная модель компетенции называется моделью компетенций SE (SWECOM). Модель была ратифицирована. Она включает в себя варианты использования для обозначения того, как менеджеры, сотрудники и новые сотрудники могли бы получить пользу от SWECOM. Глоссарий содержит определение терминов, которые используются в SWECOM и могут отличаться от обычных значений. SWECOM охватывает навыки и мероприятия для лиц, которые разрабатывают и поддерживают ПО. SWECOM основана на следующих основных источниках:

- The Software Assurance Competency Model (модель компетенций уровня доверия ПО);
- GswE2009 (методические рекомендации для учебных программ SE);
- SE2004 (методические рекомендации для студентов учебных программ SE).

Данная модель компетенции добавляет к растущей совокупности знаний SWEBOK то, что характеризует профессию программного инженера и специалистов SE. Это не директивная модель профессии SE или характеристика профессионала SE. SWECOM охватывает технические, а также лидерские навыки, необходимые для технических руководителей программного проекта в различных областях квалификации.

Определенная роль отводится подготовке менеджеров программных проектов. Они должны, прежде всего, знать следующий стандарт: Software Extension to the PMBOK Guide — Fifth Edition. Он относится к разделу компетенции SWX 2013.

Технические навыки являются основным направлением стандарта SWECOM и аналогом общеотраслевых технических компетенций.

12.3.2. Понятие навыков в стандарте SWECOM

На рис. 12.2 представлены технические навыки, включающие когнитивные навыки, поведенческие атрибуты и необходимые знания для осуществления технических видов деятельности SE (www.swebok.org, Приложение C).

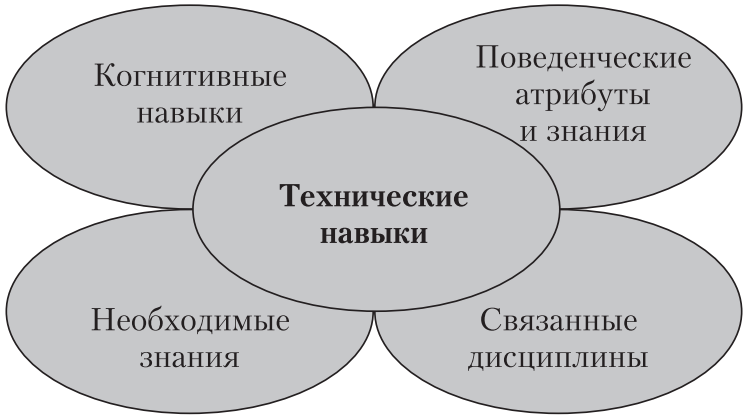


Рис. 12.2. **Элементы SWECOM**

Эти навыки приобретаются в SWECOM с помощью следующих научных дисциплин:

- компьютерная инженерия;
- информационная наука;
- общее управление;
- математика;
- управление проектами;
- управление качеством;
- системная инженерия.

Когнитивные навыки применяются ко всем областям навыков, умениям и деятельности SWECOM. Они показывают способности применения знаний и рассуждений при выполнении деятельности SWECOM в областях технической квалификации. Поведенческие атрибуты и навыки применимы ко всем элементам и на всех уровнях областей технических навыков, умений и деятельностей. Технические навыки и связанные с ними виды деятельности SWECOM сгруппированы вокруг области знания ЖЦ ПО SE и включает навыки, необходимые для выполнения различных рабочих действий при разработке ПО.

Разделы этого документа определяют ЖЦ и пересекающиеся области квалификаций для изготовления продуктов на его основе.

Контрольные вопросы и задания

1. Определите классификацию дисциплин SE.
2. Какой существует подход к обучению предмету ПИ?
3. Представьте теоретические аспекты онтологии ЖЦ.
4. Представьте основные направления SEMAT.
5. Дайте краткую характеристику направления образования в CS.
6. Какие направления образованы в SEMAT?
7. Какие задачи поставлены перед сообществом SEMAT?
8. В чем смысл новых областей SEMAT?

Заключение

В настоящее время программные и операционные исследования и разработки проводятся в отделе технологии программирования в ИСП РАН в рамках проекта РФФИ «Методы и средства разработки вариабельных операционных систем и веб-систем». Основываясь на работах автора и других специалистов по вариабельности программных систем, изложенных в данном учебнике, в рамках проекта ставятся задачи:

- определение модели вариабельности FM (Feature Model) для класса операционных и веб-систем;
- исследование работ по проблеме изменяемости ОС (конференции VAMOC 2008—2015) Linux, Unix OS — 370 и др. с целью проведения новых задач вариабельности ОС, относящихся к обеспечению проверки правильности моделей FM и систем ОС, Веб;
- разработка метода верификации рассматриваемых система, а также применение современных отработанных методов доказательства спецификации моделей вариабельности и операционных и веб-систем, а также вариантов сконфигурированных систем;
- выбор методов тестирования и их отработка применительно к отдельным артефактам и КПИ, а также сконфигурированных из них вариантов систем и их интерфейсов, включающих внешние характеристики систем и их ограничения;
- разработка новой методологии создания систем из готовых сервисов Интернета, их композирование на основе модели FM для ПС в разные варианты бизнес-задач, а также проведение верификации и тестирования как отдельных элементов, так и всей веб-системы.

Будет сформирована методология создания вариабельных веб-систем, которая использует механизмы *взаимодействия* разнородных программ и систем и возможность их переноса в другую операционную среду; обеспечивает работу с данными, которые передаются через интерфейсы или выбираются из БД, онлайн-хранилищ данных в Cloud Computing, Azure, Big Data и др.; реализуют средства хранения отдельных элементов систем, представленных в стандарте WSDL в репозиториях или *e*-бибиотеках, которые можно выбирать для последующего применения в веб-системах разного назначения; будут отработаны процессы стандарта ISO/IEC 12207 ЖЦ применительно к веб-системам, включая процесс тестирования таких систем.

В ближайшей перспективе предложенные и реализованные средства обеспечения сложных систем будут развиваться по пути инженерии к виду наноэлементов в таких областях науки, как биология, химия, физика, генетика, медицина и др. Нанотехнологии займут ведущее место в информационном мировом сообществе (от технологии программирования к нанотехнологиям, — Метод. конгресс «Инф. общество в Украине», 2013).

Словарь терминов ТП и SE

Абстрактная архитектура (Abstract architecture) — декомпозиция решения для выделенного спектра задач домена на подсистемы или иерархию подсистем, на каждом уровне которой фиксируются возможные варианты выделенных параметров и ограничений, которые определяют соответствующие вариации состава выделенных компонентов.

Абстракция (Abstaction) — это логическое представление приложения без деталей реализации, для того чтобы сконцентрироваться на его существенных свойствах.

Агрегация (Agregation) — объединение ряда понятий в новое понятие, существенные признаки нового понятия при этом могут быть или суммой признаков компонентов, или существенно новыми (или отношение «часть — целое»).

Адаптивность (Adaptive) — атрибут, который характеризует усилия, необходимые для настройки программ к разным средам без выполнения особенных действий или средств.

Анализ требований (Analysis of requirements) — отображение функций системы и ее ограничений в модели требований домена.

Артефакт (Artifacts) — любой продукт деятельности специалистов при разработке ПО.

Архитектура системы (Systems arhitecture) — определение системы в терминах вычисляемых составляющих (подсистем) и интерфейсов между ними, которое отображает правила декомпозиции проблемы на составляющие.

Аспект (Aspect) — точка зрения, с которой можно рассмотреть какое-либо понятие, процесс, перспективу.

Аспектно-ориентированное программирование (Aspect-oriented programming) — разновидность сборочного программирования, основанная на сборке полнофункциональных приложений из многоаспектных компонентов, инкапсулирующих различные варианты реализации.

Ассоциация (Association) — общее отношение, которое утверждает наличие связи между понятиями, не уточняя зависимости их содержания и объемов.

Безотказность (Faultless) — атрибут надежности, который определяет отсутствие или частоту отказов через ошибки в программном обеспечении.

Валидация (Validation) — обеспечение соответствия разработки продукта требованиям ее заказчиков.

Вариабельность (Variability) — способность продукта (системы) к расширению, изменению, приспособлению или конфигурированию с целью использования в определенном контексте и обеспечения его эволюции.

Верификация (Verification) — проверка правильности проекта в ходе разработки.

Взаимодействие (Co-operation) — обмен сообщениями между объектами разных сред.

Водопадная (или каскадная) модель (Cascade model) — схема работ, в которой каждая из работ выполняется один раз и в том порядке, в котором работы перечислены в модели ЖЦ.

Восстанавливаемость (Restored) — атрибут надежности, который указывает на возможность к перезапуску программы для повторного выполнения и возобновления данных после отказов.

Восходящее программирование («сверху вниз») разлагает проблему на отдельные функции, каждая из которых детализуется до команд машины.

Дефект (Fault) — результат ошибок разработчика во входных или проектных спецификациях, текстах кодов программ, эксплатационной документации и т.п.

Диаграмма UML (Diagram) — графическое представление элементов моделирования системы (классов, сценариев, поведения, последовательности, сотрудничества, активности, состояний, реализации, компонента и т.п.).

Домен (Domain) — спектр задач ПрО, которые допускают похожие способы их решения.

Динамический метод тестирования (Dynamic method of testing) — выполнение программ с целью установления причин ошибок с ожидаемыми реакциями на эти ошибки.

Защищенность (Protected) — атрибут функциональности, который указывает на возможность предотвращать несанкционированный доступ (случайный или преднамеренный) к программам и данным.

Изменяемость (Changeable) — показатель сопровождения, который определяет усилия, которые расходуются на модификацию, удаление ошибок или внесение изменений в связи с ошибками или новыми условиями среды функционирования.

Императивное программирование (Imperative programming) основано на идее абстрактного вычислителя, выполняющего последовательность команд программы, образуя некоторое состояние, которое изменяется при переходе к другой последовательности команд.

Индустрия (Industry) — производство разных видов продуктов массового применения и средств их изготовления для получения прибыли.

Инженерия (Engineering) — применение научных результатов, которые позволяют извлекать пользу от свойств материалов и источников энергии.

Инженерия качества (Engineering quality) — процесс предоставления ПП заданных характеристик качества.

Инженерия требований (Engineering requirements) — сбор, анализ требований заказчика и представления их в нотации, которая является понятной и согласованной как заказчиком, так и исполнителем.

Инструмент (Instrument) — программное, языковое или методическое средство, применяемое для получения состояния объекта в некотором законченном виде. В зависимости от этапа жизненного цикла и состояния объекта, его могут обрабатывать языки, трансляторы, генераторы и т.п.

Интероперабельность (Variabilities) — атрибут функциональности, который обеспечивает возможность взаимозамены совместно действующих компонентов в ПС с необходимой инсталляцией или адаптацией.

Информационная система (Infofmation systems) — система, которая проводит сбор, обработку, сохранение и выработку информации людьми с использованием автоматических процессоров.

Качество программного обеспечения (Software quality) — совокупность свойств, которые определяют возможность программного обеспечения удовлетворить заказчика, который сформулировал эти свойства как требования к разработке.

Компонент (Component) — тип, класс, проектное решение, документация или другой продукт ПИ, специально приспособленный для повторного использования.

Компонентная программа (Component program) — совокупность компонентов, которые необходимы для обеспечения функциональных и нефункциональных требований, и которая построена и функционирует в соответствии с правилами создания компонентных конфигураций и взаимодействия КПИ.

Компонентная разработка (Component development) — конструирование ПО из готовых конструкций путем композиции готовых КПИ из репозиториев.

Компонентное программирование (Component programming) — это подход к созданию компонентных программ из готовых КПИ путем сборки КПИ на всех этапах ЖЦ.

Компонентное сборочное программирование (Component assembling programming) — компонентное программирование, основанное на распространении классов в бинарном виде и предоставлении доступа к методам класса через строго определенные интерфейсы. Это программирование поддерживают технологические подходы COM, CORBA, .Net.

Компьютерная инженерия (Computer engineering) — теория и принципы построения компьютеров, фреймворков, суперкомпьютеров, вычислительных кластеров и их системного ПО.

Конкретизация (Concretization) — добавление существенных признаков, благодаря которых содержание понятия расширяется, а объем понятия сужается.

Контейнер (Container) — оболочка, внутри которой реализованы функции в виде экземпляров компонентов и которая обеспечивает взаимодействие с сервером через стандартные интерфейсы (функция Home интерфейс). Экземпляры обращаются друг к другу через системные сервисы данного или другого контейнера.

Конфигурация (Configuration) — структура ПС из компонентов и набора вариантов модулей.

Концептуальное моделирование (Conceptual design) — процесс построения модели проблемы, ориентированной на понимание ее человеком.

Логическое программирование (Logical programming) — программирование в терминах фактов и правил вывода, с использованием языков формальных исчислений.

Метод программный (Programs method) — способ или планомерный подход к достижению той цели, которая ставится перед объектом разработки (методы — нисходящий, восходящий, модульный, сборочный).

Метод сборки (Method of assembling) — способ соединения разноязычных объектов в языках программирования (ЯП), основанный на теории спецификации и отображения типов и структур данных ЯП, представленной алгебраическими системами.

Модель жизненного цикла (Life cycle model) — типичная схема последовательности работ на этапах разработки ПП.

Модель качества (Quality model) — четырехуровневая модель, которая отображает характеристики, атрибуты (показатели), метрики и оценочные элементы ПО.

Модель процессов (Process model) — действия, которые сопровождают изменения состояний объектов.

Модуль (Module) — независимая функциональная часть программы, к которой можно обращаться как к самостоятельной единице через внешний интерфейс.

Модульное программирование (Modular programming) — метод разработки программ, основанный на разбиении проблемы на независимые части — модули, которые реализуют отдельные функции и их интерфейсы (входные и выходные данные) для взаимодействия с другими модулями, собранными в большую программу.

Надежность (Reliability) — набор атрибутов, которые указывают на возможность ПО превращать входные данные в исходные результаты при условиях, которые зависят от периода времени (износ и старение не учитываются) и наличия отказов в результате внутренних дефектов или нарушения условий его применения.

Нефункциональные требования (Unfunctional requirements) — требования, которые характеризуют организационные, исполнительные, операционные и безопасные аспекты работы ПС.

Нисходящее (top down) **программирование (программирование «сверху вниз»)** — методика разработки программ, при которой разработка начинается с определения целей проблемы, после чего идет их детализация и описание в языках программирования.

Объект (Object) — базовое понятие в объектно ориентированном программировании, которое имеет свойства наследования, инкапсуляции и полиморфизма. Объекты с общими свойствами и методами образуют класс экземпляров. Они взаимодействуют между собой через сообщения и используют библиотеки классов общего применения C++.

Объектно ориентированная модель (Object-oriented model) — представление программ системы как совокупности объектов, которые взаимодействуют между собой и имеют определенные свойства и поведение.

Объектно ориентированное программирование (Object-oriented programming) — подход, при котором ПрО рассматривается как набор самостоятельных объектов, заданных в виде ориентированного графа, в вершинах которого находятся объекты, а дуги задают связи между ними. Является разновидностью сборочного программирования.

Онтология (Ontology) — совокупность примитивных понятий, терминологии и парадигмы их представления в коцептуальной модели прикладной области.

Оценка качества (Quality estimation) — действия, направленные на определение степени удовлетворения ПО потребностей соответственно их назначению.

Ошибка (Mistake) — недостатки в операторах программы или в технологическом процессе разработки, что приводят к неправильной интепретации входной информации, а следовательно, и к неправильному решению.

Парадигма АОП фиксирует модель встраивания аспекта в систему (JPM, от Join Point Model), которая включает такие важные элементы парадигмы как срезы, фрагменты, точки присоединения, аспекты и др.

Переносимость (Bearable) — группа свойств, которая обеспечивает приспособляемость к переносу из одной среды функционирования в другие, а также усилие для перенесения ПО в новую среду или сеть.

План тестирования (Plan of testing) — описание стратегии, ресурсов и графика тестирования отдельных компонентов и системы в целом.

Повторно используемый компонент (Reuses) — КПИ — фрагмент знаний о прошедшем опыте разработки систем программирования, который можно адаптировать при создании новых систем.

Правильность (Rightness), точность (Exactness) — атрибут функциональности, который показывает, как обеспечивается достижение теоретически правильных и согласованных результатов.

Программа (Program) — это объект разработки, который не является осязаемым (нельзя пощупать, взвесить и т.п.) человеком, а доступен пониманию ЭВМ, для которой написан. Готовая программа — это программный продукт, реализующий определенные функции (задачи) ПрО, процесс проектирования и разработки которого осуществлялся соответствующими программными методами, средствами и инструментами технологии программирования.

Прикладная система (Application system) — конкретный продукт программной инженерии, предназначенный для выполнения конкретных сценариев конечного пользователя.

Программирование[1] (Programming) — см. **Реализация программной системы.**

Программная инженерия (Software engineering) — система методов, средств и дисциплин планирования, разработки, эксплуатации и сопровождения ПО, готового к внедрению. Основные принципы — продуктивность, индустрия и качество (определено в ядре знаний SWEBOK (Software Engineering Body of Knowledge, URL: www.swebok.com, 2001 г.).

Продуктовая линия (Product line / Product family) — группа продуктов или услуг, которые имеют общее управляемое множество свойств, удовлетворяющие потребностям определенного сегмента рынка или вида деятельности.

Проектирование (Planning) — превращение требований в последовательность проектных решений относительно системы.

---

[1] *Лаврищева Е. М.* Методы программирования. Теория, инженерия, практика. Киев. : Наукова думка, 2006. С. 451.

Проектирование архитектуры (Planning architecture) — определение главных структурных элементов системы, которую строят.

Проектирование концептуальное (Planning conceptual) — уточнение понимания функций системы и согласование требований.

Производство продуктов (Production of products) — процессы ТЛ, посредством которых реализуется ПП определенного назначения для массового применения.

Пространство проблемы (Space of problem) — понятия ПрО ПС, характеристики объектов, новых компонентов-функций или готовых КПИ. Они связываются МХ с изменяемыми параметрами разных членов семейства и операциями взаимодействия членов между собой.

Пространство решений (Space of task) — знание о действующих компонентах и их конфигурациях, средствах их конструирования, соединения или встраивания в соответствующий член семейства с оценкой их избыточности. Элементы этого пространства фактически решают функциональные задачи определенной ПрО.

Процесс поставки (Delivery process) — действия по передаче разработанного продукта покупателю.

Процесс приобретения (Acquisition process) — действия, которые инициируют ЖЦ системы и определяют организацию-покупателя автоматизированной системы, ПС или сервиса.

Процесс разработки (Development process) — действия организации — разработчика программного продукта, которые включают инженерию требований к системе, проектированию, кодировке и тестированию.

Процесс сопровождения (Accompaniment process) — действия по управлению модификациями, поддержкой актуального состояния и функциональной пригодности, инсталляции или изъятия версии системы у пользователя.

Процесс эксплуатации (Exploitation process) — действия по обслуживанию системы во время ее использования — консультации пользователей, изучение их пожеланий и др.

Реализация программной системы (System realization) — превращение проектных решений в работоспособную ПС, которая реализует заданные функции решения (кодировка, конструирование, программирование).

Риск (Risk) — совокупность угроз, которые могут привести к неблагоприятным последствиям и убыткам, а также к изменениям состояния системы или среды.

Сборочное программирование (Assmbling programming) — метод сборки разнородных готовых программных ресурсов (модулей, объектов, компонентов, reuses, asets, services и др.) многократного применения через их интерфейсные данные, которыми обмениваются модули между собой.

Семейства систем (Families systems) — множество прикладных систем с общими функциональными свойствами.

Сервис (Service) — системный ресурс, который реализует некоторую функцию, в том числе и бизнес-функцию. Содержит независимый интерфейс с другими сервисами и ресурсами. Обеспечивает реализацию задачи интеграции программ и сервисов разной природы и используется как провайдер.

Сервисно ориентированное программирование (Service oriented programming) — совокупность взаимодействующих сервисов, веб-сервиса и их интерфейсов — это сервисно ориентированная архитектура (Service Oriented Architecture), доступ к которой происходит через веб-языки и протоколы.

Сертификация продукта (Certification of product) — процесс, который подтверждает соответствие идентифицированной программной продукции (процесса или услуг) конкретному стандарту или техническим условиям с оценкой специальным знаком или свидетельством.

Синтезирующее программирование (Synthesizing programming) — постановка задачи в виде спецификации модели вычислений, по которой производится синтез программы.

Системная инженерия (System engineering) — теория, методы и принципы построения информационных, компьютерных, АС и систем управления ими.

Событие (Event) — явление, которое провоцирует изменение определенного состояния и переход к другому состоянию в домене или системе.

Совместимость (Compatibility) — показатель переносности, который определяет возможность использования новых программ в среде действующего ПО.

Сопровождение (Accompaniment) — любые работы по внесению изменений в ПС после того, как она была передана пользователю для эксплуатации.

Спецификация (Specification) — представление правил, стандартов, критериев качества, ограничений пользования объектами.

Структурное программирование (Structured programming) — методология и технология разработки по методу «сверху вниз» отдельных блоков системы, структура которых задается графическими элементами: последовательность, ветвление и цикл.

Схема сборки (Chart of assembling) — граф, в вершинах которого находятся объекты сборки, а на дугах — их интерфейсы для взаимодействия друг с другом и обмена данными.

Сценарий (Scenario) — один из возможных путей использования системы.

Тест (Test) — некоторая программа, предназначенная для проверки работоспособности другой программы и выявления в ней ошибочных ситуаций.

Тест данные (Data of testing) — данные, которые готовятся определенными генераторами на основе документов программ или пользователем для спецификации требований и которые используются для проверки работы ПС.

Тестирование (Testing) — средство семантической наладки (проверки) программы, которая заключается в проработке программой последовательности разных контрольных наборов тестов, для которых известен результат.

Техника функционального моделирования SADT (Structured analysis and design technique) основана на спецификации функций системы в виде диаграмм, задающих фрагменты описания программ и их интерфейсов (входные и выходные данные) с другими функциональными программами.

Технология (Technology) — совокупность методов, способов, приемов, средств автоматизации и порядка их использования при производстве некоторого продукта.

Технологическая линия (Technological line) — набор процессов разработки функций объекта, представленных совокупностью автоматизированных операций, которые последовательно и систематически преобразуют состояния объекта, включая заключительное его состояние — готовый программный продукт с заданными показателями качества.

Функциональные требования (Functional requirements) — требования, которые определяют цели и функции системы.

Функциональность (Functionality) — совокупность свойств, которые определяют возможность ПО выполнять перечень функций в заданной среде и отвечают требованиям.

Характеристики качества (Quality descriptions) — функциональность (*functionality*), надежность (*reliability*), удобство (*usability*), эффективность (*efficiency*), сопровожденность (*maintainability*), переносимость (*portability*) и т.п.

Целостность (Integrity) — возможность системы сохранять стабильность в работе и не иметь риска.

«Черный ящик» метод (Black body) — тестирование реализованных функций путем проверки несоответствия между реальным поведением функций и ожидаемым поведением согласно с входными спецификациями требований.

Экземпляризация (instantiation) — зависимость между параметризуемым абстрактным классом —шаблоном (*template*) и реальным классом, который инициирован путем определения параметров шаблона.

Экстремальное программирование (Extreme programming, XP) — технология программирования, которая многократно проходит цикл создания системы от анализа потребностей заказчика до комплексного тестирования и изготовления готовой системы. В функционирующую систему последовательно добавляются различные протестированные «возможности» (*facilities*), согласованные с заказчиком, и цикл повторяется.

UML (Unified Modeling Language) — язык для спецификации, визуализации, конструирования и документирования артефактов программных систем, а также для моделирования бизнеса.

Список рекомендуемой литературы

1. *Авдошин, С. М.* Фабрики программного обеспечения / С. М. Авдошин. — 2011. — URL: http://www.ewwek.com/on.

2. *Андон, Ф. И.* Методы инженерии распределенных компьютерных приложений / Ф. И. Андон, Е. М. Лаврищева. — Киев : Наук. думка, 1997.

3. *Андон, Ф. И.* Основы инженерии качества программных систем / Ф. И. Андон [и др.]. — 2–е изд. — Киев : Академпериодика, 2007.

4. *Бабенко, Л. П.* Основы программной инженерии : учебник / Л. П. Бабенко, Е. М. Лаврищева. — Киев : Знание, 2001.

5. *Бей, И.* Взаимодействие разноязыковых программ. Диалектика / И. Бей. — М. : Вильямс, 2005.

6. *Боггс, У.* UML Rational Rose / У. Боггс, М. Боггс. — М. : Лори, 2000.

7. *Боем, Б. У.* Инженерное проектирование программного обеспечения / Б. У. Боем. — М. : Радио и связь, 1986.

8. *Буч, Г.* Объектно-ориентированный анализ / Г. Буч. — М. : БИНОМ. Лаборатория знаний, 1998.

9. *Буч, Г.* Язык UML. Руководство пользователя : пер. с англ. / Г. Буч, Д. Рамбо, А. Джекобсон. — М. : ДМК, 2000.

10. Быстродействующая вычислительная машины М-20 / под ред. И. С. Брука. М., 1957.

11. *Вельбицкий, И. В.* Технологический комплекс автоматизации программ на машинах «ЕС ЭВМ» и «БЭСМ-6» / И. В. Вельбицкий, В. Н. Ходаковский, Л. И. Шолмов. — М. : Статистика, 1980.

12. *Вельбицкий, И. В.* Технология программирования / И. В. Вельбицкий. — Киев : Техника, 1984.

13. *Волховер, В. Г.* Производственные методы разработки программ / В. Г. Волховер, Л. А. Иванов. — М. : Финансы и статистика, 1983.

14. *Вудкок, Д.* Первые шаги к решению проблемы верификации программ / Д. Вудкок // Открытые системы. — 2006. — № 8.— С. 36—43.

15. *Глушков, В. М.* Алгебра. Языки. Программирование / В. М. Глушков, Г. Е. Цейтлин, Е. Л. Ющенко. — Киев : Наук. думка, 1974.

16. *Глушков, В. М.* Вычислительная машина «Киев» / В. М. Глушков, Е. Л. Ющенко. — Киев : Изд-во ГИТЛ, 1962.

17. *Глушков, В. М.* Кибернетика, ВТ, информатика (АСУ) / В. М. Глушков // Избр. труды : в 3 т. — Киев : Наук. думка, 1990.

18. Глушков, В. М. Основы безбумажной информатики / В. М. Глушков. — М. : Наука, 1982.

19. *Глушкова, В. М.* Прошлое, устремленное в будущее / В. М. Глушкова. — Киев : Академпериодика, 2013.

20. *Глушков, В. М.* Система автоматизации производства программ — АПРОП / В. М. Глушков [и др.]. — Киев : Изд-во Ин-та кибернетики АН УССР, 1976.

21. *Гринфильд, Дж.* Фабрики разработки программ. Диалектика / Дж. Гринфильд. М. : Вильямс, 2007.

22. *Ершов, А. П.* Введение в теорию программирования / А. П. Ершов. — М. : Наука, 1977.

23. *Ершов, А. П.* Научные основы доказательного программирования : научное сообщение в президиуме АН СССР / А. П. Ершов. — М. : Наука, 1984.

24. *Жоголев, Е. А.* Система стандартных подпрограмм / Е. А. Жоголев [и др.]. — М. : Изд-во ГИФМЛ, 1958.

25. *Задорожная, Н. Т.* Менеджмент документооборота в информационных системах образования / Н. Т. Задорожная, Е. М. Лаврищева. — Киев : Педагог. думка, 2007.

26. *Иванников, В. П.* Распределенные объектно-ориентированные среды / В. П. Иванников [и др.] // Труды ИСП. — М. : Изд-во РАН, 2000. — С. 84—100.

27. *Кахро, М. И.* Инструментальная система программирования на ЕС ЭВМ (ПРИЗ) / М. И. Кахро, Э. Х. Тыугу. — М. : Финансы и статистика, 1981.

28. *Коваль, Г. И.* Программирование в системе виртуальных машин ЕС ЭВМ / Г. И. Коваль, Т. М. Коротун, Е. М. Лаврищева. — М. : Финансы и статистика, 1990.

29. *Кулаков, А. Ф.* Оценка качества программ ЭВМ / А. Ф. Кулаков. — Киев : Техника, 1984.

30. *Лаврищева, Е. М.* Методы и средства программной инженерии : учеб.-метод. пособие / Е. М. Лаврищева. — М. : Изд-во МФТИ, 2007.

31. *Лаврищева, Е. М.* Методы и средства инженерии ПО : учеб. пособие / Е. М. Лаврищева, В. А. Петрухин. — М. : Изд-во МФТИ, 2007.

32. *Лаврищева, Е. М.* Методы программирования. Теория, инженерия, практика / Е. М. Ларищева. — Киев : Наук. думка, 2006.

33. *Лаврищева, Е. М.* Основы технологической разработки прикладных программ СОД / Е. М. Лаврищева. — Киев, 1987. — (Препр. Ин-та кибернетики им. В. М. Глушкова).

34. *Лаврищева Е. М.* Проблематика программной инженерии / Е. М. Лаврищева. — Киев : Знание : Изд-во РДНТП, 1991.

35. *Лаврищева, Е. М.* Программная инженерия. Теория программирования : учеб.-метод. пособие / Е. М. Лаврищева. — М. : Изд-во МФТИ, 2016.

36. *Лаврищева, Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. — М. : Издательство Юрайт, 2016

37. *Лаврищева, Е. М.* Сборочное программирование. Основы индустрии программных продуктов / Е. М. Лаврищева, В. Н. Грищенко. — Киев : Наук. думка, 2009.

38. *Лаврищева, Е. М.* Связь разноязыковых модулей в ОС ЕС / Е. М. Лаврищева, В. Н. Грищенко. — М. : Финансы и статистика, 1982.

39. *Лаврищева, Е. М.* Программная инженерия. Технология программирования : учеб.-метод. пособие / Е. М. Лаврищева. — М. : Изд-во МФТИ, 2016

40. *Лаврищева, Е. М.* Транслятор «Д-АЛГАМС» для УВК «Днепр-2» / Е. М. Лаврищева [и др.]. — Киев : Изд-во Ин-та кибернетики АН УССР, 1981.

41. *Липаев, В. В.* Качество АСУ / В. В. Липаев. — М. : Радио и связь, 1983.

42. *Липаев, В. В.* Оценка затрат на разработку ПС / В. В. Липаев. — М. : Финансы и статистика, 1988.

43. *Липаев, В. В.* Технология проектирования комплексов программ / В. В. Липаев. — М. : Радио и связь, 1982.

44. *Липаев, В. В.* Технология сборочного программирования / В. В. Липаев, Б. А. Позин, А. А. Штрик. — М. : Радио и связь, 1992.

45. *Иванников, В. П.* Распределенные объектно-ориентированные среды / В. П. Иванников [и др.] // Труды Института системного программирования РАН. — 2000. — С. 84—100.

46. *Морган, М.* JAVA2. Руководство разработчика : пер. с англ. / М. Морган. — М. : Вильямс, 2000.

47. *Непомнящий, В. А.* Спецификация и верификация распределенных систем средствами языка Elementary-REAL / В. А. Непомнящий, Н. В. Шилов, Е. В. Бодин // Программирование. — 1999. — № 4. — С. 54—67.

48. *Орлов, С. А.* Технология разработки ПО / С. А. Орлов. — СПб. : Питер, 2002.

49. *Орлик, С.* Программная инженерия / С. Орлик. — URL: http://www.almportal.ru/public/so/book/3-soft ware_engineering.pdf.

50. Рекомендации программной инженерии и информатики в университетах. Software Engineering 2004, Computing Curricula 2001: Computer Science / ИНТУИТ 2007.

51. *Роджерсон, Д.* Основы COM / Д. Роджерсон. — М. : Изд.-торг. дом «Русская редакция» : Microsoft Press, 1997.

52. Системы компьютерной алгебры семейства АНАЛИТИК. Теория, Реализация, Применение / НАНУ Украины. Институт математических машин и систем. — Киев, 2010.

53. *Соммервил, И.* Инженерия программного обеспечения : пер. с англ. / И. Соммервил. — 6-е изд. — М. : Вильямс, 2002.

54. *Тыугу, Э. Х.* Концептуальное программирование / Э. Х. Тыугу. — М. : Наука, 1984.

55. *Уилсон, С. Ф.* Принципы разработки ПО : учеб. курс : пер. с англ. / С. Ф. Уилсон, Б. Мэйплс, Т. Лэндгрейв. — М. : Изд.-торг. дом «Русская редакция», 2000.

56. Управляющая вычислительная система «Днепр-2» / А. Г. Кухарчук, В. М. Египко и др. — Киев : Наук. думка, 1972.

57. *Фаулер, М.* UML в кратком изложении. Применение стандартного языка объектного моделирования : пер. с англ. / М. Фаулер, К. Скотт. — М. : Мир, 1999.

58. *Фреге, Г.* Логика и логическая семантика / Г. Фреге. — М. : Аспект-пресс, 2000.

59. *Фуксман, А. Л.* Технологические основы создания программных систем / А. Л. Фуксман. — М. : Статистика, 1979.

60. *Чернецки, К.* Порождающее программирование. Методы, инструменты, применение / К. Чернецки, У. Айзенкер. — СПб. : Питер, 2005.

61. *Эммерих, В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и JAVA RMI / В. Эммерих. — М. : Мир, 2002.

62. *Ющенко, Е. Л.* Адресное программирование / Е. Л. Ющенко. — Киев : Знание, 1963.

63. Agile — манифест разработки ПО. — URL: www.agilemanifesto/org/iso.

64. *Bachmann, F.* Variability in software product lines / F. Bachmann, P. Clements. CMU/SEI Technical Report CMU/SEI-2005-TR-012, 2005.

65. *Barker, R.* CASE-method. Entity Relationship Modelling / R. Baker. — N. Y. : Publ., 1990.

66. *Bettin, J.* Model-Driven Software Development / Jorn Bettin // MDA Journal. — 2004, April. — URL: http://www.bptrends.com/publicationfiles/04-04%20COL%20MDSD%20Frankel%20-%20Bettin%20-%20Cook.pdf.

67. *Duval, P.* Continuous integration Împroving Software Quality and Reducing Risk / P. Duvual, Matyas, Grover A. — Addison Wesley, 2009.

68. *Fowler, M.* Language Workbenches and MDA / M. Fowler. — URL: http:// martinfowler.com.

69. *Guckenheimer, S.* Software Engineering with Microsoft Studio Team System / S. Guckenheimer, J. I. Perez. — Crawfordsville, USA : Adison–Wesley, 2006.

70. *Hoare, T.* Verified software: Theories, Tools, Experiments. Vision of Grant Challenge project / T. Hoare, J. Misra. — Microsoft Research Ltd and the University of Texas at Austin, 2005.

71. ISO/IEC DTR 19759 Software Engineering — Guide to the Software Engineering Body of Knowledge — SWEBOK. — URL: http://www.swebok.org/ironman/pdf/ SWEBOK_Guide_2004.pdf.

72. *Jacobson, I.* Object-oriented Software Engineering. A use case driven Approacth / I. Jacobson. — N. Y. : Wesley Addison Publ. Co., 1994.

73. *Jackson, M.* Software requirement & specifications / M. Jackson. — Wokingham, England : Addison–Wesley, ACM Press Books, 1995.

74. *Jacobson, I.* Software Reuse: Architecture, Process and organization for Business Success / I. Jacobson, M. Griss, P. Johnson. — Addison Wesley, Reading, MA, May 1997.

75. *Lenz, G.* Practical software fabrics in Net. From theory in practice — a primer referense and case study / G. Lenz, C. Wienands. — Apress, 2007.

76. *Mernik, M.* When and How to Develop Domain-Specific Languages / M. Mernik, J. Heering, A. M. Sloane // ACM Computing Surveys. — 2006. — Vol. 37. — № 4. — P. 316—344.

77. Model Driven Development and Software Product Lines. BigLever Software Inc. — 2007. — URL: http://www.biglever.com/technotes/mdd_spl.html?source=mdd.

78. MSF — Microsoft Solution Framework, 2005. — URL: http://msdn.microsoft.com.

79. *Northrop, L. M.* SEI's Software Product Line Tenets / L. M. Northrop // IEEE Software. — 2002. — Vol. 19, № 4. — P. 32—39.

80. OMG Unified Modeling Language Specification. — Version 1.3, June 1999. — URL: http://folk.uio.no/inods/doc/UML1.3.pdf.

81. *Pohl, K.* Software Product Line Engineering: Foundions, Principles and Techniques / K. Pohl, G. Böckle, F. J. Linden. — N. Y. : Springer–Verlag, 2005.

82. *Siegel, J.* CORBA Fundamentals and Programming / J. Siegel. — N. Y. : Wiley Co. Publ. Group, John Wiley & Sons, Inc., 1996.

83. *Skidmore, S.* SSADM: Models and Mehtods / S. Skidmore, G. Mills, R. Farmer. — Prentice-Hall, Englewood Cliffs, 1994.

84. URL: http://computer.org/education/cc2001.

85. URL: http://se.math.spbu.ru/cc2001.

86. *Гринфильд, Дж.* Фабрики разработки программ / Дж. Гринфильд. — М. ; СПб. ; Киев : Диалектика ; Изд. дом «Вильямс», 2007.

87. *Морган, М.* JAVA2. Руководство разработчика / М. Морган : пер. с англ. — М. : Изд. дом «Вильямс», 2000.

Новые издания по дисциплине «Программирование» и смежным дисциплинам

1. *Зыков, С. В.* Программирование : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2016.
2. *Зыков, С. В.* Программирование. Объектно-ориентированный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2016.
3. *Зыков, С. В.* Программирование. Функциональный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2016.
4. *Тузовский, А. Ф.* Объектно-ориентированное программирование : учеб. пособие для прикладного бакалавриата / А. Ф. Тузовский. — М. : Издательство Юрайт, 2016.
5. *Черпаков, И. В.* Основы программирования : учебник и практикум для прикладного бакалавриата / И. В. Черпаков. — М. : Издательство Юрайт, 2016.

Моделирование систем диаграммами UML

Основная цель языка UML — визуализация на разных уровнях абстракции от наиболее абстрактной общей модели к более наглядной логической и далее к физической модели платформы ООП. В UML нашли отражение методы объектно ориентированного моделирования, разработанные Г. Бучом, Дж. Рамбо (метод Object Modeling Technique, OMT), И. Якобсоном (метод Object-Oriented Software Engineering, OOSE). Все указанные методы известны как средства визуального моделирования, зарекомендовавшие себя на практике.

Главным инструментом UML являются графические диаграммы, с помощью которых описываются три основные объектные модели:

1) *структурная* (статическая) служит для описания структуры сущностей системы, включая классы, интерфейсы, отношения, атрибуты;

2) *модель поведения* (динамическая) описывает поведение объектов системы, включая методы, взаимодействие, процесс изменения состояний отдельных компонентов или всей системы;

3) *модель функционирования* описывает модель функционирования системы.

Словарь UML оперирует такими понятиями, как сущности, отношения и диаграммы. Первые являются первичными элементами модели, вторые связывают различные сущности, а третьи группируют совокупности сущностей. В UML существует ряд правил, позволяющих корректно и однозначно определять:

- имена, которые можно давать сущностям, отношениям и диаграммам;
- область действия имен (контекст, в котором имя имеет некоторое значение);
- видимость (когда имена видимы и могут использоваться другими элементами);
- целостность (согласованное соотношение элементов системы);
- исполнение модели.

Концептуальная модель

В UML имеется четыре типа сущностей: структурные, поведенческие, группирующие и аннотационные.

Структурные сущности обычно являются статичными элементами системы, соответствующими концептуальным или физическим ее элементам. Выделяют следующие типы структурных сущностей: класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел. Основные сущности: актор, сигнал, утилита (утилиты являются видами

классов), процесс и нить (нити являются видами активных классов). К остальным относятся: применение, документ, файл, библиотека, страница и таблица.

Поведенческие сущности являются динамическими составляющими модели UML. Выделяют следующие типы поведенческих сущностей: взаимодействие и автомат.

Группирующие сущности являются организующими частями модели UML. На данный момент реализована только одна групповая сущность — пакет.

Аннотационные сущности — разъяснительные части модели UML, изображаются примечаниями (комментариями).

В UML существует четыре типа отношений: зависимость, ассоциация, обобщение, реализация.

Есть также их вариации, например уточнения, трассировки, включения и расширения (для зависимостей), агрегация и композиция.

Виды диаграммы UML

Диаграммы — это основные объекты UML. Существует несколько типов диаграмм, представляющих различные аспекты систем. Наборы диаграмм позволяют получить достаточно полное представление обо всей проектируемой системе и ее отдельных компонентах. Диаграмма в UML — это графическое представление набора элементов, изображаемое обычно в виде связного графа с вершинами (сущностями) и ребрами (отношениями).

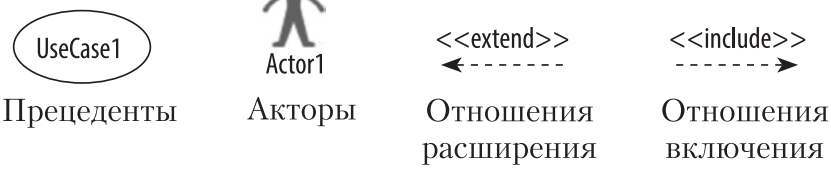
В UML определено восемь видов диаграмм, каждая из которых может содержать элементы определенного типа. Типы возможных элементов и отношений между ними зависят от вида диаграммы. Виды диаграмм:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграммы поведения, к которым относятся диаграммы состояний, деятельности, взаимодействия; в свою очередь, к диаграммам взаимодействия относятся диаграммы последовательности и кооперации;
- диаграммы реализации, к которым относятся диаграммы компонентов и развертывания.

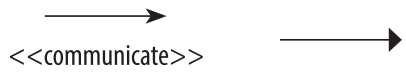
На всех диаграммах могут присутствовать комментарии.

Диаграммы задают концептуальную модель системы:

- определяются общие границы и контекст систем;
- уточняется их внешнее функциональное поведение, появляется первичная документация, которая может использоваться для предметного обсуждения будущей системы разработчиками, заказчиками, пользователями и другими заинтересованными сторонами. Диаграммы *use case* называются диаграммами прецедентов и являются основой для дальнейшей детализации системы в форме логических и физических моделей. Приведем их.



Прецеденты | Акторы | Отношения расширения | Отношения включения



Отношения ассоциации

Отношения обобщения

Отношение расширения (*extend*) является отношением зависимости между базовым и другим прецедентом, функциональное поведение которого используется базовым не всегда, а только при выполнении некоторых условий. Стрелка направлена к базовому прецеденту (во всех других отношениях — от прецедента).

Прецедент, который включается, не зависит от базового в том смысле, что он предоставляет последнему инкапсулирующее поведение, детали реализации которого скрыты. Таким образом, базовый прецедент зависит только от результатов выполнения прецедента, который в него включается, а не от его структуры.

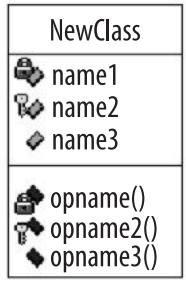
Потомок наследует все свойства родительского поведения прецедента (актора) и может иметь дополнительные.

Диаграммы классов

Описывают статическую структуру классов. Разрешается на концептуальном уровне формировать словарь ПрО и на уровне спецификаций и реализаций определять структуру классов в программной реализации системы.

Диаграммы классов могут использоваться для генерации каркасного кода (в реальном языке программирования).

Класс изображается прямоугольником из трех частей.



В верхней секции размещены имя, в средней — атрибуты, в нижней — методы класса. Атрибуты и методы класса описываются с указанием спецификаторов доступа. В UML они обозначаются +, # и – и означают:

+ — публичные (*public*) атрибуты/методы;

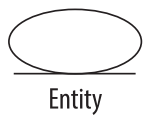
— защищенные (*protected*) атрибуты/методы;

– — частные (*private*) атрибуты/методы.

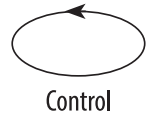
Выделены следующие типы классов (класификация по назначению).



Приграничные (*boundary*), или интерфейсные



Классы-сущности (*entity*)

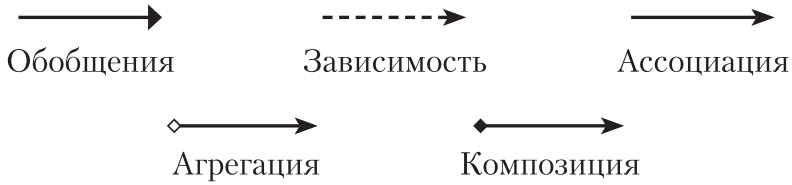


Управления (*control*) (классы-менеджеры)

Интерфейсы обозначаются так:



Классы могут быть абстрактными или конкретными. Типы отношений между классами обозначаются следующим образом:



Отношение обобщения отражает иерархическое строение классов и наследование свойств и поведения.

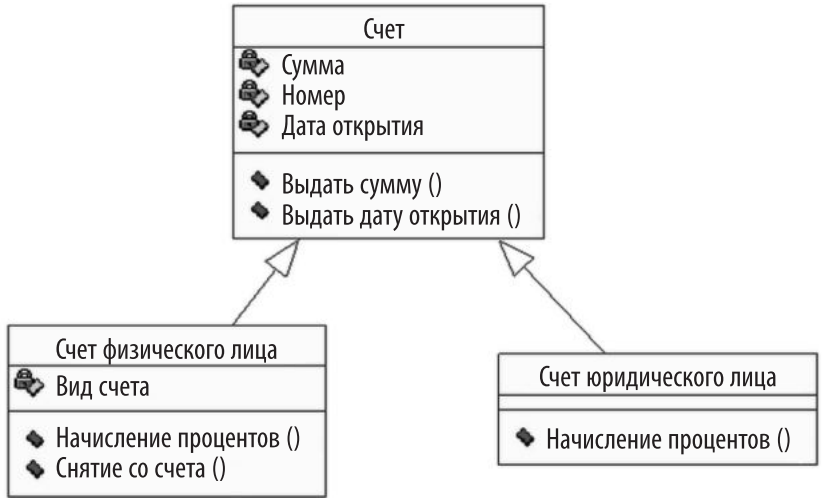
В качестве ограничений могут быть использованы следующие ключевые слова UML:

{*complete*} — в данном отношении обобщения специфицированы все классы-потомки — других потомков у данного класса быть не может;

{*incomplete*} — на этой диаграмме изображены не все потомки, добавление новых потомков не повлияет на построенную диаграмму;

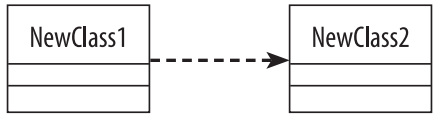
{*disjoint*} — классы-потомки не могут содержать объекты, одновременно являющиеся экземплярами двух или более классов;

{*overlapping*} — отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам.



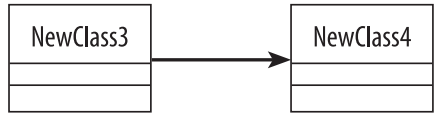
Отношение зависимости отражает связь между классами, когда зависимый класс зависит от определений, сделанных другим классом.

Зависимость — однонаправленная связь от зависимого класса к классу, от которого он зависит:



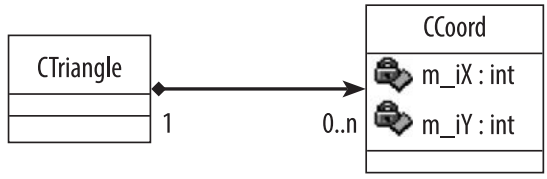
При генерации кода для зависимых классов к ним не добавляются новые атрибуты, но создаются необходимые команды (в код вставляется директива препроцессору #include).

Отношение ассоциации между классами — это семантическая связь между ними:



При генерации кода отношение агрегации добавляет соответствующий атрибут в код класса «целое».

Сильная разновидность агрегации — композиция, когда объект может принадлежать только единственному целому, а при удалении целого уничтожается и часть:



Диаграммы поведения (*Diagram behavior*) делятся на диаграммы состояний и диаграммы деятельности.

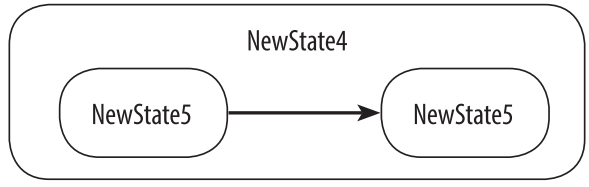
Диаграммы состояний (*Statechart diagram*) определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате некоторых событий.

Диаграмма состояний состоит из следующих элементов:



Состояние позволяет отразить ситуацию в жизни объекта.

С состоянием ассоциируется некоторая деятельность, которая обозначается меткой и изображается как <выполнить>/<действие>. Состояния могут образовывать группы, как в приведенном примере диаграммы состояний:



Между состояниями возможны следующие отношения:

- переход для выхода из одного состояния в другое при наступлении определенного события или изменения условий;
- переход на себя (рефлексивный).

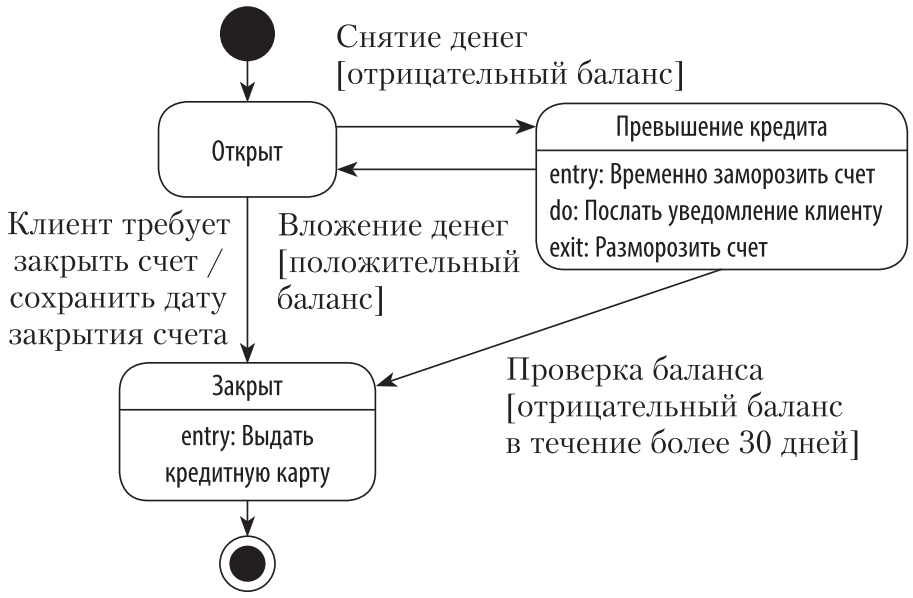
Переход может иметь метку, синтаксис которой состоит из трех необязательных частей: <Событие>[<Условие перехода>]/<Действие>. Если метка перехода не содержит никакого события, то это означает, что переход состоится при условии, что завершится деятельность, ассоциированная с данным состоянием.

Действие — это набор простых операций:

а) вход в состояние — на схеме это обозначается надписью entry/ <действие>;

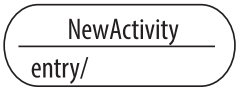
б) выход из состояния — на схеме это обозначается надписью exit/<действие>;

в) при переходе выполняется действие, которое на схеме обозначается меткой перехода/<действие>.



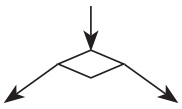
Действия отражают динамику системы, задавая схемы потоков управления, в том числе параллельные и альтернативные потоки (а также разветвления).

Деятельность (активность) на диаграмме изображается так:

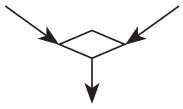


Между видами деятельности возможны отношения перехода: ⟶

Условное поведение изображается таким образом:

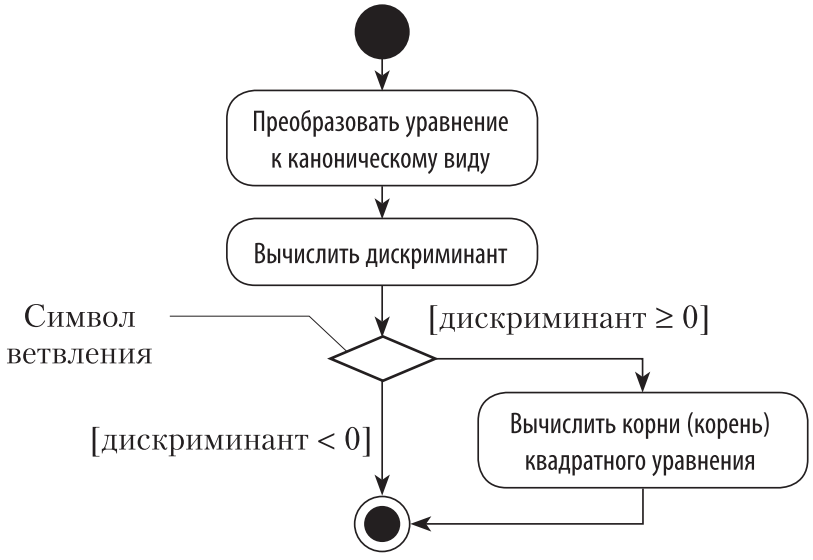


Ветвления (начало условного поведения)



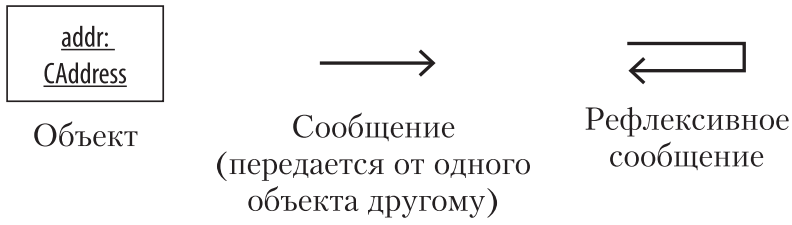
Соединения (завершение условного поведения)

Деятельность складывается из поддеяльностей.



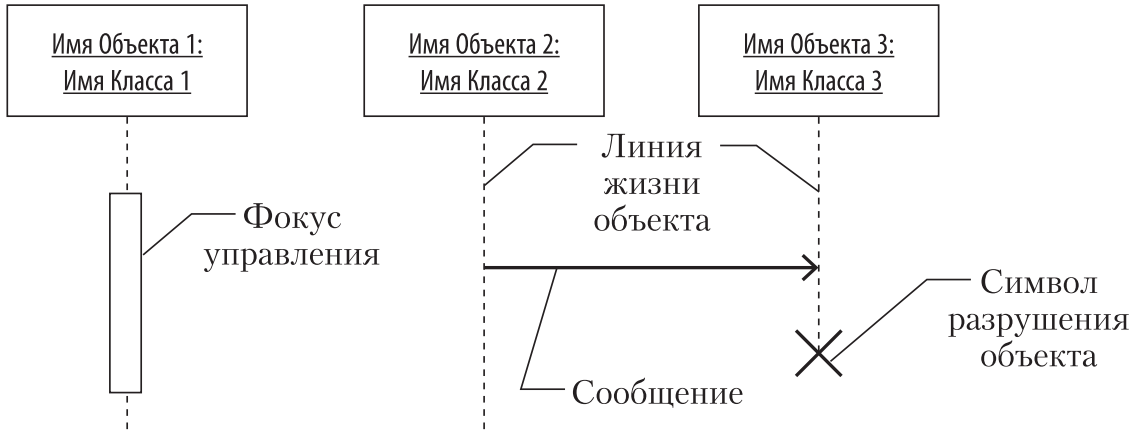
Диаграммы взаимодействия (*Interaction diagrams*) делятся на диаграммы последовательности (*Sequence diagram*) и кооперации (*Collaboration diagram*). Каждый сценарий реализуется совокупностью объектов, взаимодействующих между собой. Такая совокупность называется кооперацией.

Диаграмма последовательности позволяет изобразить сценарий прецедента графически, путем отображения последовательности сообщений, которыми обмениваются объекты. Состоит из следующих элементов:



Каждое сообщение содержит метку (обязательно указывается имя сообщения, к тому же можно указать аргументы и некоторую управляющую информацию).

Вертикальной линией под обозначением объекта на диаграмме *последовательности* отражена его линия жизни — ЖЦ объекта в процессе взаимодействия. Для отображения периода активности объекта используется прямоугольник активности.



Предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме последовательности в форме стереотипа рядом с сообщением, к которому они относятся. В этом случае они записываются в кавычках. Используются следующие обозначения для моделирования действий:

- *call* (вызвать) — сообщение, требующее вызова операции или процедуры принимающего объекта. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у того, кто прислал это сообщение объекта;
- *return* (возвратить) — сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;
- *create* (создать) — сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может получить фокус управления, а может и не получить его;

• *destroy* (уничтожить) — сообщение с явным требованием уничтожить соответствующий объект. Ссылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;

• *send* (послать) — обозначает посылку другому объекту.

На **диаграмме кооперации** (*Collaboration diagram*) экземпляры объектов и сообщений между ними отражаются так же, как и на диаграмме последовательности. Однако их временна́я последовательность указывается посредством нумерации сообщений. Эти диаграммы отражают конкретный сценарий варианта использования.

Диаграммы реализации (*Diagram Implementation*) разделяются на диаграммы компонентов и развертки.

Диаграммы компонентов (*Component diagram*) изображают компоненты системы и связи между ними. Они задают порядок компилирования и исполнения компонентов. Применяются теми участниками проекта, которые отвечают за компиляцию системы. Из этих диаграмм очевидно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут при этом созданы.

Группы логически связанных компонентов позволяют структурировать модель системы:

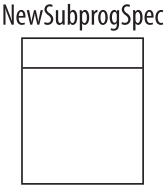


Подпрограмма (*subprogram*) используется для программ, а также для необъектно ориентированных компонентов.

Тело подпрограммы (*subprogram body*):



Спецификация подпрограммы (*subprogram specification*):



Родовая спецификация подпрограммы (*generic subprogram specification*):



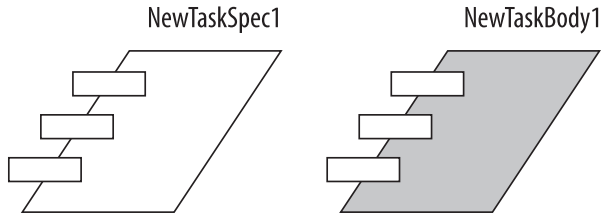
Для представления exe- и .dll-файлов используются компоненты типов EXE и DLL:



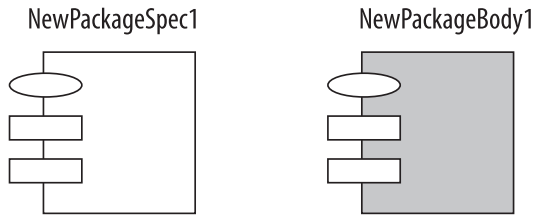
Для задания БД используют компонент



Компоненты *Task specification* и *Task body* позволяют отобразить независимые потоки в многопотоковой системе:



Компоненты *Package specification* и *Package body* имеют вид



В языках C, C++ данные компоненты используются таким образом: *Package specification* — для .h-файлов, *Package body* — для файлов с расширениями .c и .cpp.

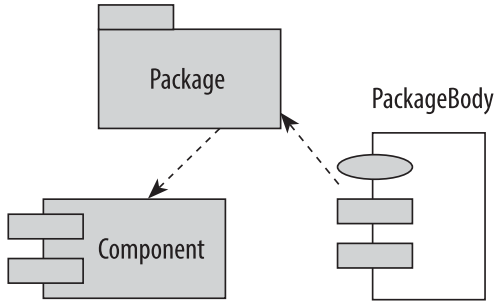


Диаграмма развертки (*Diagram Deployment*) используется менеджером проекта, пользователями, архитектором системы для понимания физического размещения системы и ее отдельных подсистем.

Сборка систем в .NET

Впервые сборка программ представлена в монографии «Связь разноязыковых модулей в ОС ЕС» (1982). В ней содержится теория построения разноязыковых отдельных модулей/КПИ, их объединение в ПС и трансформация несовместимых ТД объединяемых разноязыковых КПИ через интерфейсного посредника (рис. П2.1) и разработанной библиотеки интерфейсных функций (из 64) в системе АПРОП. Развитию теории сборки посвящена монография «Сборочное программирование» (1991).

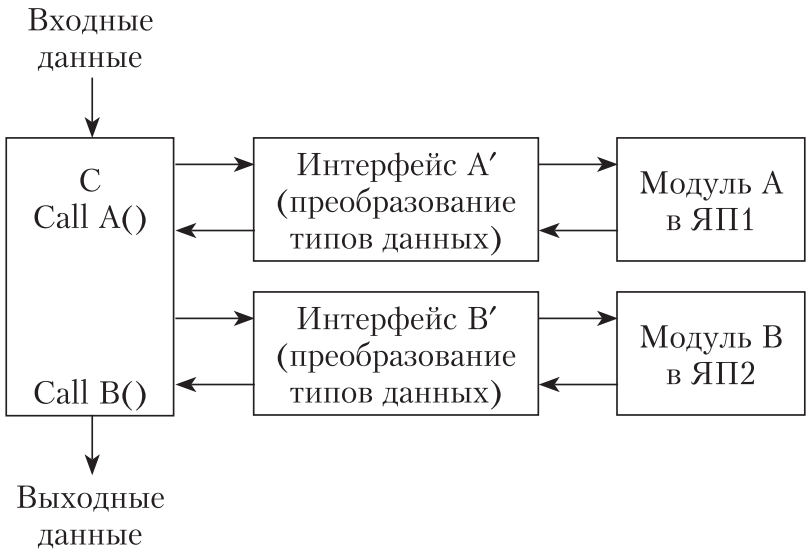


Рис. П2.1. **Схема сборки разноязыковых программ**

Сборка систем из КПИ[1]

Сборка КПИ от разнородных программ в ЯП (C++, C, Fortran 77, Pyton, Java и т.п.) осуществлялась операторами CALL, RPC, RMI, передаваемыми через интерфейс или перекидной мостик (stub, skeleton) между объектами, данные которых описываются в языках IDL, API, Icontract и др., а также в языке SIDL (Scientific IDL).

Корень проблемы связи разнородных программ — ТД и средства их обработки между разнородными программами, расположенными на разных компьютерах в Интернете. Все зависит от ЯП и ТД, которые существуют для описания параметров передачи данных другим программам. Каждая инструментальная среда реализует взаимодействие таких программ

[1] *Радецкий И. О.* Один из подходов к взаимодействию программ в средах Java и Eclipce // Проблемы программирования. 2011. № 2. С. 44—54.

по своему принципу. Примерами являются известные системы поддержки коллективного создания программных продуктов — VS.Net, CORBA, IBM, Sun Microsystems, Java и др. (1994—2004) со сборкой:

- одноязычных программ в среде VS.Net с использованием библиотек CRL, CTS и CLS;
- разноязычных программ из готовых КПИ через посредник ORB системы CORBA;
- взаимодействия программ на языках Java, C#, Basic и C++ с интерфейсом IDL в ИТК (http://sestudy.edu-ua.net).

Сборка, взаимодействие программ в разных средах

Сборка (коммуникация) программ в разных ЯП далее представлена новым механизмом задания интерфейса передачи данных между разноязычными программами (рис. П2.2).

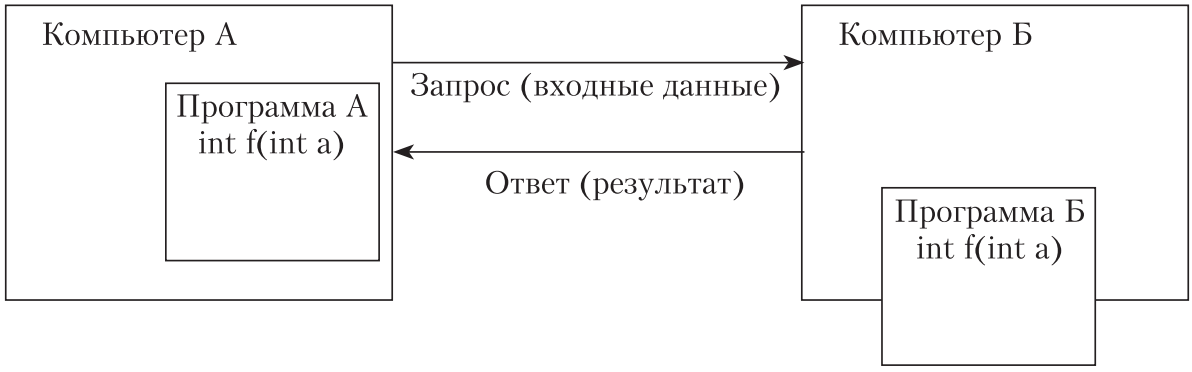


Рис. П2.2. **Схема связи разноязычных программ**

Пусть некая программа А находится в сети на компьютере А. В этой программе реализован некоторый интерфейс, содержащий int f (int a). Но конкретно функцию *f* программа А не реализует, т.е. не реализует то, что хотел бы пользователь. Например, функция int f (int a) использует параметр *a*, выполняет с его помощью определенные действия и возвращает результат вычислений. Но в программе А в функции *f* эта логика не реализована. Она реализована в программе Б, которая находится на компьютере Б, находящимся в той же сети, что и компьютер А.

Программа Б содержит функцию int f (int a) и служит сервером, который дает доступ другим программам к этой функции. То есть конкретная логика определенного действия реализована в программе Б, а другие программы, которым нужны такие вычисления, обращаются к ней (программы Б).

Кроме того, программы могут быть написаны на разных ЯП, они могут размещаться на разных сетевых компьютерах с разными архитектурами (i# 16 and 64 и др.). На этих компьютерах также могут быть установлены разные ОС (Windows, Linux, Mac OS и др.), а программы могут выполняться на разных платформах (Java, .NET, Python и др.).

В качестве основы для реализации способа передачи данных между компьютерами была выбрана технология сокетов (*Socket*). Эта технология удовлетворяет всем вышеперечисленным требованиям (разноязычность, разноплатформность, разные ОС, разные компьютерные архитектуры),

а также как отдельный случай позволяет реализовать объединение программ на одной машине.

Для реализации создания интерфейсов была взята идея из технологии CORBA, в которой содержится механизм описания интерфейса на языке спецификации интерфейсов IDL. А потом с помощью некоторых программных средств можно на основе этого интерфейса генерировать файлы уже для конкретных ЯП, которые поддерживаются данной технологией (например, Java и C#). Например, на вход программы для генерации конкретных классов поступают следующие параметры: ЯП, тип класса (сервер или клиент) и задекларированный интерфейс. Сгенерированные классы могут быть двух видов: клиент; сервер.

Если сгенерированный класс — *клиент*, то в него помещаются только механизмы для наладки коммуникации между программами, которые генерируются генератором. Пользователь этого класса ничего не должен в нем менять, а использование выглядит так:

```
…
int param;
…
ClientInterface ci = new ClientInterface();
int result = ci.func(param);
…
```

Таким образом, ClientInterface — это сгенерированный класс, который содержит механизмы, необходимые для передачи данных на сервер, который предоставляет сервис int func (int param). Необходимо только создать экземпляр данного класса и вызвать необходимую функцию.

Если сгенерированный класс — сервер, то кроме механизмов наладки связи с клиентами в него помещают пустые функции. Они отвечают задекларированным методам, которые описаны в интерфейсе. Для использования сервера пользователю необходимо реализовать конкретную логику данных методов.

Это выглядит следующим образом:

```
…
class ServerInterface {
…
public int func (int param){
…//реализация методу
}
…
}
…
void main(){
…
ServerInterface si = new ServerInterface();
si.work();
…
}
```

Эти примеры демонстрируют, что главной целью данной технологии является обеспечение функциональности в связи разноязычных программ,

простоты и прозрачности их использования. Это и есть главное преимущество данной технологии над технологией CORBA, для которой только главная функция клиентской и серверной программы выглядит следующим образом:

Сервер:

```
public static void main(String args[]) {
try {
// create and initialize the ORB
ORB orb = ORB.init(args, null);
// get reference to rootpoa & activate the POAManager
POA rootpoa = POAHelper.narrow(
orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();
// create servant and register it with the ORB
HelloImpl helloImpl = new HelloImpl();
helloImpl.setORB(orb);
// get object reference from the servant
org.omg.CORBA.Object ref = rootpoa.servant_to_
reference(helloImpl);
Hello href = HelloHelper.narrow(ref);
// get the root naming context
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
// Use NamingContextExt which is part of the Interoperable
// Naming Service (INS) specification.
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
 // bind the Object Reference in Naming
String name = "Hello";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);
System.out.println("HelloServer ready and waiting ...");
// wait for invocations from clients
orb.run();
} catch (Exception e) {
System.err.println("ERROR: " + e);
e.printStackTrace(System.out);
}
System.out.println("HelloServer Exiting ...");
}
```

Клиент:

```
public static void main(String args[]){

try{
// create and initialize the ORB
ORB orb = ORB.init(args, null);
// get the root naming context
org.omg.CORBA.Object objRef = orb.resolve_initial_references(
"NameService");
// Use NamingContextExt instead of NamingContext. This is
// part of the Interoperable naming Service.
```

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
// resolve the Object Reference in Naming
String name = "Hello";
helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
System.out.println("Obtained a handle on server object: " +
helloImpl);
System.out.println(helloImpl.sayHello());
helloImpl.shutdown();
} catch (Exception e) {
System.out.println("ERROR : "e) ;   e.printStackTrace(System.
out);
}
}
```

Механизмы передачи данных

Механизм передачи данных между программами реализован с помощью технологии сокетов. Сокет — это название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут выполняться как на одной, так и на разных машинах, связанных между собой сетью. Сокет содержит конечную точку соединения, через которую обеспечивается коммуникация программ, которые написаны на разных ЯП, находятся на разных платформах, компьютерах в сети с разными архитектурами.

Данные передаются по сети в виде потоков байтов. Но многие ТД у ЯП имеют размер больше 1 байта. В таблице приведены размеры ТД, которые используются в таких ЯП, как Java и C#.

ТД и их размер

ТД	Размер, байты
Bool	1
Byte	1
Char	1
Short	2
Int	4
Long	8
Double	8
Float	8

Тип byte в ЯП Java имеет диапазон от 1 до 127, в то же время как у ЯП C# — от 0 до 255.

Эти факты накладывают определенные ограничения на механизм передачи данных. В его реализации ключевым моментом является механизм превращения параметра типа int в массив байтов.

```
public static byte[] intToByteArray(int value) {
byte[] b = new byte[4];
```

```
for (int i = 0; i < 4; i++) {
int offset = (b.length — 1 — i) * 8;
b[i] = (byte) ((value >>> offset) & 0xFF);
}
return b;
}
```

В данном примере результатом будет массив байтов, который имеет размер 4 (так как тип int имеет размер 4 байта). Каждый раз биты параметра сдвигаются вправо на восемь позиций. Потому делается явное превращение типов из int в byte, после которого от данного параметра остается его поточный байт.

Для реализации методов превращения таких типов, как *long* и *short*, необходимо изменить размер результирующего массива на 8 и 2 соответственно.

Типы с фиксированной и плавающей точкой (*double* и *float*) необходимо сначала привести к параметру *long*, но не при помощи явного превращения типов, а при помощи стандартной функции getRowLongBits(). Потом необходимо произвести такие же действия, как и с параметрами типа *long*.

Когда какая-нибудь программа получает данные, они находятся в виде массива байтов. Необходимо из этого массива взять данные, т.е. превратить определенный массив байтов в тип int, double или какой-то другой. Ниже приведен фрагмент кода, который содержит функцию для представления массива байтов в число типа int:

```
public static int byteArrayToInt(byte[] b, int offset) {
int value = 0;
for (int i = 0; i < 4; i++) {
int shift = (4 — 1 — i) * 8;
value += (b[i + offset] & 0x000000FF) << shift;
}
return value;
}
```

В этой функции также используется побитовая операция сдвига. Сдвиг проходит влево на восемь позиций, при этом происходит превращение типа byte в тип int. В результат записывается сумма предыдущего значения и число со сдвинутым влево байтом.

Аналогичные функции есть и других типов. Основное различие между ними — размер входного массива байтов. Для типов с плавающей и фиксированной точкой после работы соответствующей функции производится превращение стандартной функцией longRawBitsToDouble() или longRawBitsToFloat() из числа типа long в число типа double или float соответственно.

При передаче данных клиент посылает серверу имя функции, которую необходимо выполнить. Потом — количество аргументов и сами аргументы в следующей последовательности: тип аргумента и значение аргумента. Сервер, получив данные и совершив вычисления, высылает клиенту следующее сообщение: название функции и результат. Результат состоит из двух частей — тип результата и значения.

Спецификация интерфейсов

Для задания спецификации интерфейсов выбран механизм системы CORBA. Но именно спецификация интерфейса с помощью определенного языка спецификации интерфейсов IDL. Спецификация начинается с ключевого слова interface, после которого указывается имя интерфейса, а потом в фигурных скобках — тело интерфейса. Выглядит это следующим образом:

```
interface Test{…}
```

Тело интерфейса состоит из объявлений методов, которые перечисляются через точку с запятой. Для объявления методов необходимо указать тип и имя параметра. Ниже приведен пример интерфейса:

```
//Test.if
interface Test{
int func1(int p1, double p2);
float func2(float p1);
}
```

Это описание похоже на язык Java и C#, поэтому пользователям будет легко его освоить. Входные и выходные параметры могут быть только простого ТД.

После спецификации интерфейса нужно сгенерировать клиентский и серверный классы. Для этого используется программа classgen:

classgen [classtype] [language] interfacefile

classtype — тип класса (клиент или сервер);

language — ЯП генерированного класса;

interfacefile — путь к файлу, в котором описан интерфейс.

Тестовый пример

В качестве тестового примера реализована следующая задача: дана некоторая программа на ЯП Java, содержащая метод int times(int a, int b); эту программу необходимо использовать в другой, написанной на ЯП C#.

Java-программа:

```
package javatest;
public class Test{
public int times(int a, int b){return a*b;}
}
```

Интерфейс к java-программе:

```
//testj.if
interface TestJ{
int times(int a, int b);
}
```

С помощью генератора получены следующие классы.

Сервер (Java):

```
class TestJServer{
private int port;
```

```
private String addr;
private ServerSocket ss;
private Socket cl;
public TestJ(String addr, int port){
this.addr = addr;
this.port = port;
try {
ss = new ServerSocket(port);
System.out.println("Created server socket " + ss);
cl = ss.accept();
System.out.println("Accepted new client " + cl);
} catch(Exception e) {e.printStackTrace();}
}
public void work(){
try {
InputStream is = cl.getInputStream();
OutputStream os = cl.getOutputStream();
byte[] size = new byte[1];
is.read(size);
byte[] methodName = new byte[size[0]];
is.read(methodName);
byte[] a = new byte[4];
byte[] b = new byte[4];
is.read(a);
is.read(b);
if (ByteArrayConverter.ByteArrayToString(methodName).equals
("times")){
byte[] rc = ByteArrayConverter.IntToByteArray(
this.times(ByteArrayConverter.ByteArrayToInt(a),
ByteArrayConverter.ByteArrayToInt(b)
)
);
os.write(rc);
}
} catch (IOException ex) {
Logger.getLogger(Server.class.getName()).log(Level.SEVERE,
null, ex);
}
}
public int times(int a, int b){
return (new Test()).times(a, b);
}
}
```

Клиент (C#):

```
class TestJClient
{
private int port;
private IPAddress addr;
private IPEndPoint ep;
private Socket s;
```

```
public TestJ(byte[] addr, int port)
{
this.addr = new IPAddress(addr);
this.port = port;
this.ep = new IPEndPoint(this.addr, this.port);
s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
0);
s.Connect(this.ep);
Console.WriteLine("Connected to server " + addr +
    " on port " + port);
}
public int times(int a, int b)
{
byte[] byta = BitConverter.GetBytes(a);
byte[] bytb = BitConverter.GetBytes(b);
s.Send(new byte[] { 5 });
s.Send(new byte[] { (byte)'T', (byte)'i', (byte)'m',
(byte)'e', (byte) 's' });
s.Send(byta);
s.Send(bytb);
Console.WriteLine("Sent arguments: int " + a + ", int " + b);
byte[] rc = new byte[4];
s.Receive(rc);
Console.WriteLine("Received result: int " +
BitConverter.ToInt32(rc, 0));
return BitConverter.ToInt32(rc, 0);
}
}
```

Объединенная с Java-программой C#-программа выглядит следующим образом:

```
class Program
{
static void Main(string[] args)
{
TestJClient c = new TestJClient(new byte[] { 127, 0, 0, 1},
1050);
Console.WriteLine(c.times(1000, 3));
}
}.
```

В результате исполнения программы получено решение:

```
Connected to server 127.0.0.1 on port 1050
Sent arguments: int 1000, int 3
Received result: int 3000
```

Данная технология базируется на клиент-серверной архитектуре, где сервер обеспечивает определенные сервисы, а клиент делает запросы к серверу для выполнения функций. Клиент передает входные данные и имя функции, которую надо выполнить, а сервер, исполнив запрошенную функцию, генерирует ответ, содержащий результат исполнения для передачи

его клиенту. С помощью IDL можно специфицировать интерфейс между разноязычными компонентами, методы которого будут выступать в качестве сервисов на сервере, предоставляющем доступ к ним клиентам; также разработан механизм генерации клиентских и серверных классов. В серверном классе необходимо реализовать конкретную логику предоставляющихся функций, а для клиентского класса нужна программа-генератор, которая сгенерирует нужные механизмы для обеспечения взаимодействия программы с сервером.

Взаимодействие систем в Интернете

Проблема взаимодействия компонентов и систем, созданных в разных средах, носит нетривиальный характер. Далее рассматриваются основные принципы построения модели взаимодействия систем в разнородных средах на примере алгоритма шифрования с открытым ключом RSA, отдельные компоненты которого располагаются в средах Microsoft .NET и Java. При этом используются преимущества каждой из сред: поддержка вычислений с произвольно большими целыми числами в Java и простота написания пользовательского интерфейса в .NET[1].

1. Принципы взаимодействия программ в распределенных системах

Модель взаимодействия КПИ входит в состав модели разработки ПО и ПС [1—3]. К настоящему времени сформировались следующие модели:

- генерирующая модель GDM (*Generative Domain Model*);
- сервисно ориентированная модель SOA (*Service-Oriented Architecture*);
- архитектурно ориентированная модель MDA (*Model Driven Architecture*);
- модель OSI (*Open Systems Interconnection*) и др.

В рамках генерирующей модели ПС приложение порождается из отдельных элементов — компонентов, сервисов, каркасов и т.п. В основе этой модели лежит ООП, дополненный механизмами порождения КПИ и средствами их взаимодействия. К модели взаимодействия (*interconnection*) в GDM предъявляется требование обеспечения взаимодействия разнородных компонентов, работающих в разных операционных средах.

Определяющим аспектом модели взаимодействия является понятие распределенной системы, включающей следующие вспомогательные определения.

Определение 1. Хост — компьютер, на котором взаимодействуют компоненты, входящие в единую систему.

Для обеспечения взаимодействия между компонентами, размещенными на разных хостах, теоретически можно пользоваться примитивами, предоставляемыми сетевой ОС (т.е. используя транспортные каналы), либо промежуточным слоем.

Определение 2. Промежуточный слой (*middleware*) — слой между сетевой операционной системой и приложениями, предназначенный для решения проблем неоднородности (различия между ОС и средами выполнения по отношению к форматам данных хоста) и распределения (размещение компонентов на разных хостах).

---

[1] *Островский А. В.* Подход к обеспечению взаимодействия программ в середах Java и MS.Net // Проблемы программирования. 2011. № 2. С. 37—44.

В разных распределенных системах используются подходы к построению промежуточного слоя. В системах распределенных баз данных, таких как IBM CICS и Tuxedo, используется транзакционно-ориентированный промежуточный слой, который обеспечивает одинаковый механизм обработки транзакций для разных БД. Промежуточный слой, ориентированный на сообщения, применяется для обеспечения надежного обмена сообщениями между компонентами (примером может служить Java Message Queue). Удаленный вызов процедур (Remote Procedure Call, RPC), применяемый в операционных системах MS Windows, позволяет вызывать процедуры за пределами хостов. Такие технологии, как CORBA компании OMG, Microsoft COM, удаленный вызов методов (Remote Method Invocation, RMI) в системе Java, используют объектно ориентированный промежуточный слой, считающийся более перспективным.

Определение 3. Распределенная система — система, состоящая из компонентов, которые расположены на объединенных в сеть хостах, взаимодействуют через промежуточный слой, обладают свойством прозрачности (*transparency*) и внешне их механизмы взаимодействия не отличаются от таковых между компонентами в централизованной системе.

Общая модель взаимодействия компонентов в распределенной системе показана на рис. П3.1.

При реализации распределенной системы применяется стандартная семиуровневая модель ISO/OSI. В этой модели нижние уровни взаимодействия (физический, канальный, сетевой и транспортный) обеспечиваются сетевой ОС, а самый верхний уровень — прикладной — средствами прикладных программистов. Промежуточный слой реализует уровень представления и уровень сессии.

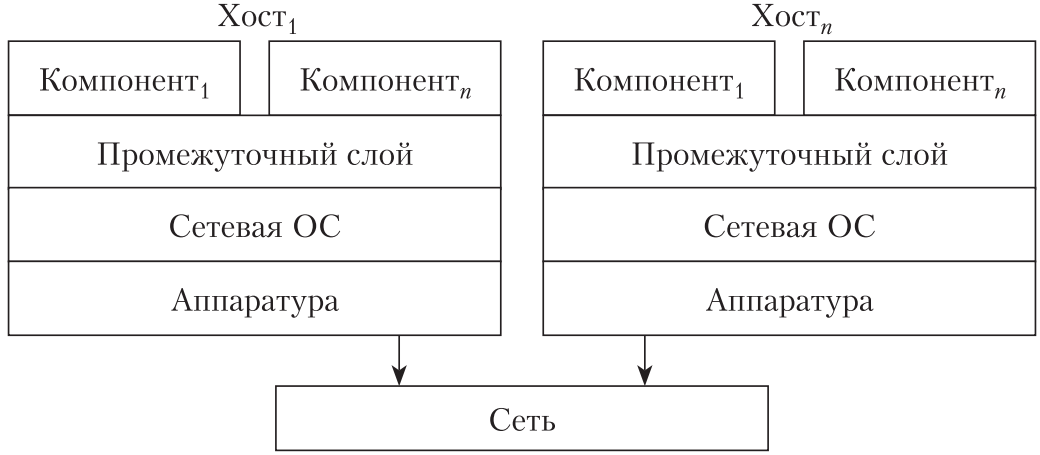


Рис. П3.1. **Модель работы распределенной системы**

Уровень представления отображает структуры данных приложения в однородной форме (не зависящей от среды выполнения). Преобразование данных от прикладного представления к транспортному называется маршалингом, а обратное преобразование — демаршалингом. Для того чтобы маршалинг и демаршалинг данных происходили автоматически и учитывались на этапе компиляции приложения, промежуточный слой применяет клиентские и серверные стабы (*stub*) — специальные интерфейсы, используемые соответственно компонентом, который осуществляет удаленный

вызов, и компонентом, который этот вызов обрабатывает. Что касается уровня сессии, он обеспечивает связь между компонентами за счет одного или нескольких транспортных каналов, отображая ссылки на объекты в хосты, которые эти объекты содержат. В большинстве распределенных систем с объектно ориентированным промежуточным слоем уровень сессии имеет достаточно сложную структуру и не будет рассматривается [4].

2. Распределенная система CORBA

Технология CORBA (Common Request Broker Architecture — общая архитектура брокера объектных запросов) позволяет организовать взаимодействие между компонентами с произвольными средами разработки и выполнения, а также предоставляет возможность удаленного взаимодействия. Объектно ориентированный промежуточный слой, используемый в CORBA, предусматривает объединение программного кода в объекты, интерфейс доступа к которым описывается на языке описания интерфейсов IDL (*Interface Definition Language*) [5].

В нем определены атомарные типы: boolean (логическая переменная), char (символ), short (целое число, 2 байта), long (целое число, 4 байта), float (число с плавающей запятой, 4 байта), string (символьная строка) и др.

С помощью атомарных типов строятся более сложные: списки (*sequence*), структуры (*struct*), массивы (*array*), объединения (*union*). КПИ определяются с использованием ключевого слова interface; объектная модель IDL поддерживает ограничение доступа к полям объектов и обработку исключительных ситуаций.

ТД, поддерживаемые ЯП, определенным образом отображаются в типы IDL и обратно. Стандартом установлены отображения для языков Ada, C++, C, Lisp, Smalltalk, Java, Кобол, Object Pascal, PL/1, Python; созданы отображения и для других языков и сред разработки, в том числе для языков среды .NET [6].

Промежуточный слой в технологии CORBA обеспечивается брокером объектных запросов (ORB). ORB производит компиляцию IDL-описаний компонентов, создание клиентских и серверных стабов (и скелетонов в терминологии CORBA). Существует большое количество реализаций брокеров, такие как Borland Enterprise Broker, MICO, omniORB, JacORB; ORB входит в пакет Java Development Kit. Помимо этого, существуют пакеты, которые расширяют функции брокера, позволяя компилировать IDL-описания и стабы для дополнительных ЯП (например, для языков платформы Microsoft .NET с помощью пакета IIOPNet). Схема работы распределенной системы CORBA изображена на рис. П3.2.

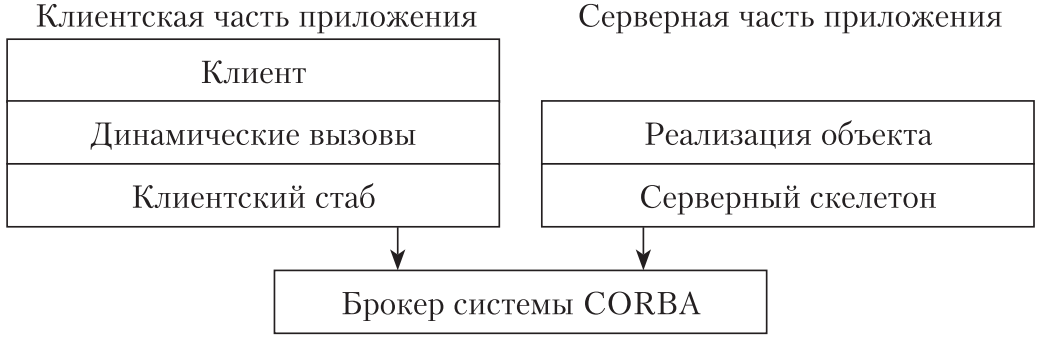


Рис. П3.2. **Схема работы распределенной системы CORBA**

3. Реализация взаимодействия компонентов в распределенной среде

Фрагменты кода на языках Java и C# для платформы .NET, приведенные ниже, используются для демонстрации основных принципов программирования распределенной системы.

3.1. Описание практической задачи

В качестве теста для проверки взаимодействия сред используется программа шифрования с открытым ключом RSA. Согласно алгоритму шифрования случайным образом выбираются два достаточно больших простых числа p и q. Затем вычисляется их произведение $N = pq$ и функция Эйлера от него $\varphi(N) = (p-1)(q-1)$. В качестве открытого ключа используется пара (e, N), где e — некоторое небольшое число (одно из чисел Ферма 17, 257, 65 537); в качестве секретного — пара (d, N), где число d взаимно обратно к e по модулю $\varphi(N)$. Кодирование числа M состоит в вычислении $X = M^e \pmod N$, раскодирование — в вычислении $M = X^d \pmod N$. Надежность шифрования для алгоритма зависит от величины числа N — надежным считается 1024-битный ключ. Для эффективной реализации RSA нужно, чтобы среда программирования поддерживала операции над большими числами. Одной из таких сред является Java, в ней большие числа реализованы в виде класса BigInteger из модуля java.math, который предоставляет всю необходимую функциональность: арифметические операции, генерацию случайных простых чисел, модульную арифметику. Таким образом, в данном случае использование Java для серверной части приложения вполне естественно.

CORBA реализует объектный подход к проблеме взаимодействия, поэтому один из главных вопросов при работе с этой технологией состоит в определении структуры объектов. Объект RSAService, имплементирующий алгоритм шифрования RSA, содержит следующие методы:

- **newKey()** — генерирует открытый и секретный ключи и возвращает открытый ключ;
- **encode(key, message)** — кодирует текстовое сообщение message открытым ключом key, алгоритм которого предназначен для кодирования чисел, а сообщение разбивается на фрагменты, каждый из которых определенным образом переводится в число, не превосходящее N, и затем кодируется согласно вышеописанному методу;
- **decode(key, code)** — декодирует закодированное сообщение секретным ключом, соответствующим открытому ключу *key*;
- **checkKey(key)** — проверяет открытый ключ *key* на подлинность. Фактически, этот метод проверяет, существует ли секретный ключ с числом N таким же, как и в открытом ключе. Методы *encode* и *decode* будут выполнять эту проверку перед началом вычислений.

При такой архитектуре объекта клиентам передаются только открытые ключи, а соответствующие им секретные хранятся на сервере. Это обеспечивает сокращение длины открытого ключа (вместо пары чисел (e, N) достаточно хранить только N, поскольку e можно оставлять фиксированным), следовательно, уменьшается и количество данных, которыми обмениваются клиент и сервер.

Так как в клиентской среде .NET отсутствует аналог больших чисел, в приложении используется текстовое представление ключа и закодиро-

ванных сообщений в системе исчисления с большим основанием (чтобы получившиеся строки имели меньшую длину).

Таким образом, все данные, которыми обмениваются клиент и сервер, имеют строковый тип. Как в ЯП Java, так и среде .NET строки являются неизменяемыми (*immutable*) экземплярами класса String, причем поля и методы класса определяются по-разному. Из-за этого прямое преобразование строк при переходе от одной среды к другой затруднено. При этом в обеих средах программирования строки хранятся внутри объектов String в виде последовательности символов Unicode фиксированной длины и поэтому соответствуют стандартному ТД CORBA wstring, т.е. передача данных через CORBA не вызывает проблем.

3.2. Описание серверной части приложения

Технология CORBA интегрирована в среду программирования Java. Регистрация объекта в этой распределенной системе подразумевает создание трех базовых частей приложения:

1) интерфейс объекта, содержащий описание удаленно доступных полей и методов объекта. Этот интерфейс наследуется от стандартного интерфейса Remote из модуля java.rmi; каждый его метод декларируется как потенциально вызывающий исключение RemoteException, поскольку это исключение возбуждается серверным скелетоном при возникновении каких-либо неполадок. С учетом описанной в разд. 3.1 архитектуры объекта, декларация интерфейса будет выглядеть так, как показано на рис. П3.3. Помимо четырех функций, упомянутых ранее, в описании интерфейса также объявляется функция getVersion, возвращающая версию объекта;

```
public interface RSAService extends Remote {
   public abstract String getVersion() throws RemoteException;
   public abstract String newKey() throws RemoteException;
   public abstract boolean checkKey(String key) throws
         RemoteException;
   public abstract String encode(String key, String text) throws
         RemoteException;
   public abstract String decode(String key, String code) throws
         RemoteException;
}
```

Рис. П3.3. **Описание интерфейса доступа к объекту в среде Java**

2) класс, являющийся потомком класса PortableRemoteObject из модуля javax.rmi, который реализует описанный интерфейс. Этот класс определяется с публичным конструктором без аргументов (у PortableRemoteObject конструктор имеет область видимости protected), причем в простейшем случае никаких действий, помимо вызова родительского конструктора, внутри него не производится;

3) сервер, публикующий ссылку на класс, делая его доступным для других программ с помощью метода rebind сервиса именования Context. Этот метод связывает имплементацию объекта с его именем в виде строки; другие приложения, использующие технологию CORBA, после этого могут обращаться к этому объекту по его имени.

Как уже упоминалось в разд. 2, в комплект разработчика Java Development Kit встроен ORB. С помощью приложения rmic из этого пакета опи-

сание интерфейса на языке Java преобразуется в IDL-описание (рис. П3.4). Метод getVersion, исходя из его имени, при этом автоматически превращается в поле строкового типа с доступом только для чтения (аналогично преобразуются методы set). Это же приложение генерирует стабы стандартным образом согласно имени интерфейса. В данном случае клиентский стаб и серверный скелетон будут называться _RSAService_Stub и _RSAServiceImpl_Tie соответственно, при этом фактически приложение будет использовать только скелетон, так как клиентская часть приложения написана не на Java.

```
interface RSAService {
   readonly attribute ::CORBA::WStringValue version;
   ::CORBA::WStringValue newKey( );
   boolean checkKey( in ::CORBA::WStringValue arg0 );
   ::CORBA::WStringValue encode(
      in ::CORBA::WStringValue arg0,
      in ::CORBA::WStringValue arg1 );
   ::CORBA::WStringValue decode(
      in ::CORBA::WStringValue arg0,
      in ::CORBA::WStringValue arg1 );
};
```

Рис. П3.4. **Описание объекта на языке IDL**

3.3. Описание клиентской части приложения

В отличие от Java платформа .NET не располагает встроенными средствами работы с технологией CORBA. Для их взаимодействия используются пакеты сторонних разработчиков, одним из которых является IIOPNet с открытым исходным кодом. С помощью утилиты из этого пакета IDL-описание объекта преобразуется в CLS-совместимую библиотеку, которая затем присоединяется к приложению. Для рассматриваемого приложения библиотека будет содержать описание интерфейса RSAService.

Для связи с брокером ORB пакет IIOPNet предоставляет канал связи IiopClientChannel и сервис именования NamingContext, позволяющий получить ссылку на объект по его имени. Канал фактически играет роль клиентского стаба, выполняя маршалинг и демаршалинг передаваемых через него данных.

Метод resolve сервиса именования возвращает ссылку на объект по его имени, которое было задано вызовом метода rebind. В платформе Microsoft .NET возвращаемое значение имеет тип MarshalByRefObject, что позволяет трактовать все обращения к полям и методам объекта как удаленные вызовы. Использование канала и сервиса именования в клиентской части приложения показано на рис. П3.5.

Общая последовательность шагов построения приложения:

1) скомпилировать серверную часть приложения;

2) с помощью утилиты rmic создать серверный скелетон и описание объекта на языке IDL;

3) пользуясь пакетом IIOPNet, преобразовать полученное IDL-описание в CLS-совместимую библиотеку;

4) присоединить библиотеку к клиентской части приложения для использования сервиса.

```
//зарегистрировать канал связи с CORBA
channel = new IiopClientChannel();
ChannelServices.RegisterChannel(channel, false);
// получить сервис через его имя
CorbaInit init = CorbaInit.GetInit();
NamingContext nameService = init.GetNameService("localhost",
1050);
NameComponent[] name = new NameComponent[] {
      new NameComponent("RSAService", "")
};
service = (RSAService)nameService.resolve(name);
//узнать версию сервиса
string result = service.version;
```

Рис. П3.5. **Использование объекта в клиентской части приложения**

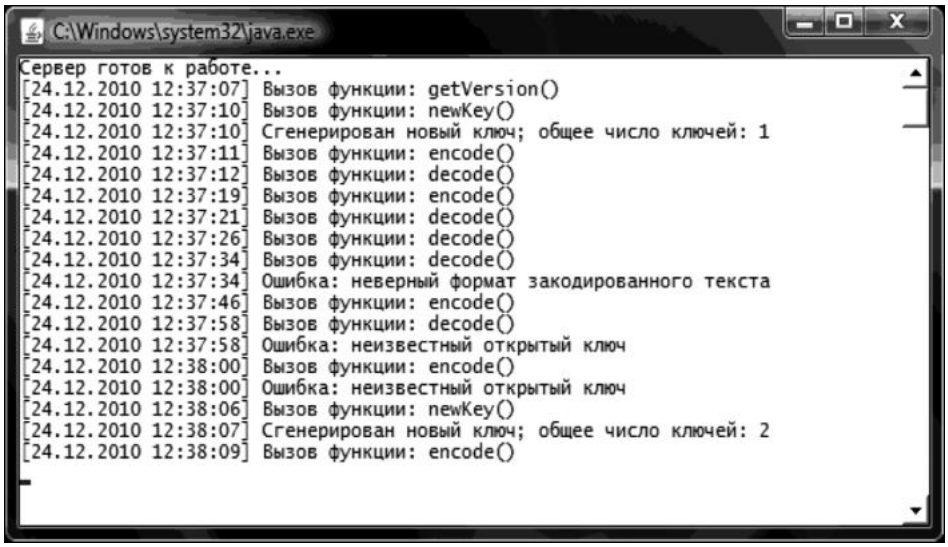


Рис. П3.6. **Интерфейс серверной части приложения**

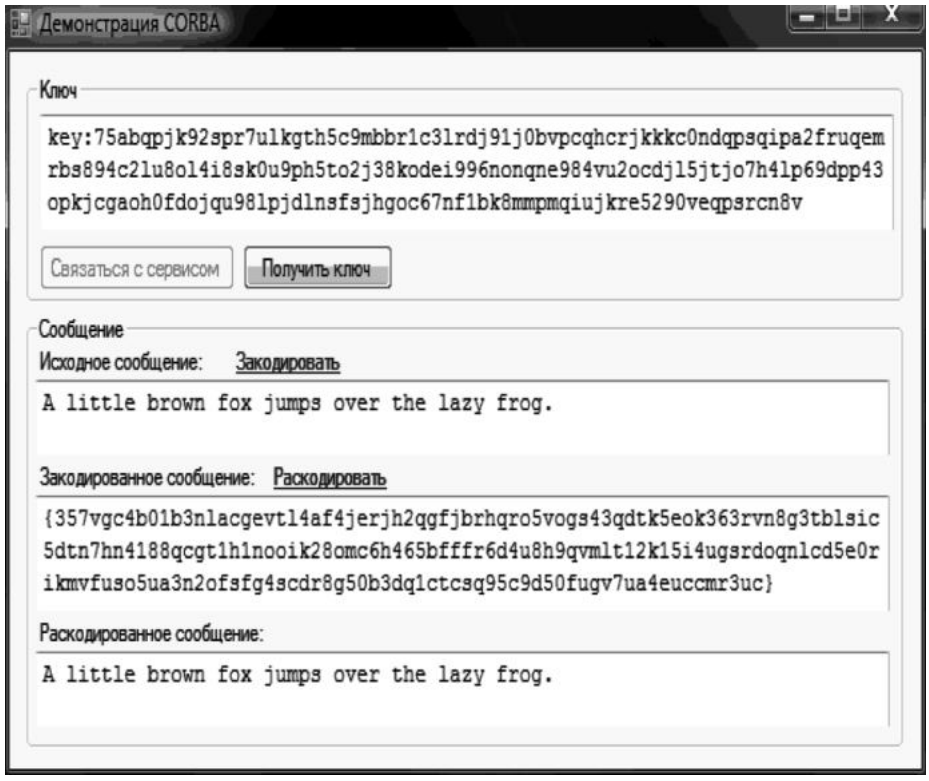


Рис. П3.7. **Интерфейс клиентской части приложения**

3.4. Обработка исключительных ситуаций

Часто необходимо, чтобы сервер передавал клиенту отчеты об исключениях, возникающих при обработке запросов. В случае рассматрения сервиса шифрования с открытым ключом исключительные ситуации могут возникать в нескольких ситуациях:

- используется некорректное строковое представление чисел (т.е. открытого ключа или закодированного сообщения) при вызове функций *encode* и *decode*;
- предоставленный для кодирования или декодирования открытый ключ не является подлинным.

Спецификация CORBA включает в себя исключения, порождаемые методами объектов, однако использование стандартных исключений Java для этих целей невозможно, так как они не поддерживают маршалинг. Чтобы обойти эту проблему, в серверной части приложения определяется пользовательский тип исключения RSAException, имплементирующий интерфейс java.io.Serializable, так как в Java все объекты, в том числе и исключения, поддерживающие маршалинг, реализуют этот интерфейс. Способ преобразования данных к транспортному представлению и обратно при этом выбирается платформой Java автоматически. После изменения соответствующим образом деклараций методов encode и decode в интерфейсе RSAService при компиляции IDL-описания объекта будут автоматически сгенерированы описания объектного ТД (ключевое слово языка IDL valuetype) RSAException, соответствующего исключению, и его обертки — CORBA-совместимого исключения (exception) RSAEx.

Помимо этого, будут сгенерированы описания некоторых стандартных классов Java, связанных с пользовательским исключением, например Exception и Throwable из модуля java.lang, являющиеся его предками в иерархии классов. IDL-файлы в процессе компиляции преобразуются в интерфейсы и помещаются в CLS-совместимую библиотеку, так что их имплементация обеспечивается клиентской частью приложения. Исключение RSAEx после компиляции станет потомком класса пользовательского исключения AbstractUserException, определенного в пакете IIOPNet; аналогично IDL-описанию, оно будет содержать в себе экземпляр объекта RSAException.

Итак, проверка на возникновение исключительных ситуаций в клиентской части приложения будет осуществляться так, как показано на рис. П3.8.

```
try {
   string decMessage = service.decode(key, code);
} catch (RSAEx rsaE) { //Пользовательское исключение
   MessageBox.Show(rsaE.value.message);
   //Действия по обработке исключения
} catch (Exception exc) { //Какое-то другое исключение
   MessageBox.Show(exc.Message);
   //Действия по обработке исключения
}
```

Рис. П3.8. **Обработка исключительных ситуаций клиентом**

Описание реализаций интерфейсов является задачей не программиста, пишущего клиентскую часть приложения, а ответственного за серверную часть. Поэтому в реальных приложениях имплементации пользовательских исключений и классов Java компонуются в отдельную CLS-совместимую библиотеку, которая поставляется вместе с интерфейсом CORBA-объекта.

Вывод. Таким образом, была представлена реализация задачи взаимодействия прикладных компонентов, разработанных в программных средах Microsoft .NET (клиентская часть приложения) и Java (серверная часть) через CORBA. Приведены общие принципы взаимодействия в распределенных системах, в частности в технологии CORBA, а также принципы написания КПИ в программных средах MS.NET и Java. Разработана структура распределенного приложения, реализующего алгоритм шифрования с открытым ключом с возможностью обработки исключительных ситуаций. Полученный результат взаимодействия демонстрирует эффективность использования каждой из названных сред программирования.

Литература

1. *Лавріщева, К. М.* Генерувальне програмування програмних систем і сімейств / К. М. Лавріщева // Проблеми програмування. — 2009. — № 1. — С. 3—16.

2. *Андон, П. І.* Розвиток фабрик програм в інформаційному світі / П. І. Андон, К. М. Лавріщева // Вісник НАН України. — 2010. — № 10. — С. 15—41.

3. *Лаврищева, Е. М.* Проблемы интероперабельности разнородных объектов, компонентов и систем. Подходы к ее решению. Спец. выпуск международной конференции «УКРПрО — 2010» / Е. М. Лаврищева // Проблемы программирования. — 2010. — № 2—3. — С. 28—41.

4. *Эммерих, В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft/COM и Java/RMI. : пер. с англ. / В. Эммерих — М. : Мир, 2002.

5. *Лаврищева, Е. М.* Методы и средства инженерии программного обеспечения : учеб. пособие / Е. М. Лаврищева, В. А. Петрухин. — М. : Изд-во МФТИ, 2007.

6. *Островский, А. В.* Подход к обеспечению взаимодействия программных сред Java и Ms.Net / А. В. Островский // Проблеми програмування. — 2011. — № 2. — С. 37—44.

7. *Радецький, І. О.* Один з підходів до забезпечення взаємодії середовищ MS.Net і Eclipse / І. О. Радецький // Проблеми програмування. — 2011. — № 2. — С. 45—52.

8. *Lavrischeva, E.* General Disciplines and Tools for E-learning Software Engineering / E. Lavrischeva, A. Ostrovski // 8th International Conf. ICTERI–2013 «ICT in Education, Research and Industrial Applications», Ukraine, June, 2012, Springer.com, Communication in Computer and Information Sciences. — P. 212—229.

9. *Бабенко, Л. П.* Онтологический подход к спецификации свойств программных систем и их компонент / Л. П. Бабенко // Кибернетика и системный анализ. — 2009. — № 1. — С. 30—37.