

Роберт Тласс



ПРОГРАММИРОВАНИЕ И КОНФЛИКТЫ

ТЕОРИЯ И ПРАКТИКА ПРОГРАММНОЙ ИНЖЕНЕРИИ

2.0

software conflict 2.0

The art and science
of software engineering

Robert L. Glass



ПРОГРАММИРОВАНИЕ И КОНФЛИКТЫ 2.0

Теория и практика
программной инженерии

Роберт Гласс



Санкт-Петербург — Москва
2010

Серия «Профессионально»

Роберт Гласс

ПРОГРАММИРОВАНИЕ И КОНФЛИКТЫ 2.0

Перевод В. Овчинникова

Главный редактор	<i>А. Галунов</i>
Зав. редакцией	<i>Н. Макарова</i>
Выпускающий редактор	<i>П. Щеголев</i>
Редактор	<i>Е. Тульсанова</i>
Корректор	<i>Л. Уайатт</i>
Верстка	<i>К. Чубаров</i>

Гласс Р.

Программирование и конфликты 2.0. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 240 с., ил.

ISBN 978-5-93286-148-6

Сборник очерков, написанных ветераном и пионером индустрии разработки ПО Робертом Глассом, автором и редактором более чем двадцати пяти книг. Первое издание увидело свет еще в 1991 году и сразу же стало бестселлером.

Книга посвящена конфликтам и управлению конфликтами. В ней программисты и разработчики ПО противопоставлены менеджерам. Теория противопоставляется практике, стабильность – изменениям, разговоры – действиям и обещания – результатам. Предпринята попытка показать все стороны, участвующие в споре, заставить задуматься и сделать собственные выводы.

Ценность этой книги в том, что она выходит за круг обсуждения модных тенденций и сиюминутных взглядов и понятий. Откровения, содержащиеся в ней, не подвержены влиянию времени и сегодня могут служить источником информации и вдохновения для разработчиков и менеджеров, профессоров и предпринимателей, исследователей и студентов.

ISBN 978-5-93286-148-6

ISBN 0-9772133-0-7 (англ)

© Издательство Символ-Плюс, 2010

Authorized translation of the English edition © 2006 Developer.* books. This translation is published and sold by permission of Developer.* books, the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7, тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 29.10.2009. Формат 70×90 ¹/₁₆. Печать офсетная.

Объем 15 печ. л. Тираж 2000 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука» 199034, Санкт-Петербург, 9 линия, 12.

*Посвящается Айрис,
которая помогала мне совершенствоваться.*

Оглавление

Об авторе	11
Другие книги Роберта Л. Гласса.....	13
Вступительное слово ко второму изданию	15
Вступительное слово к первому изданию	17
Предисловие к первому изданию	19
Предисловие ко второму изданию	23
1. Обзор театра военных действий	25
Что было раньше: Теория или Практика?	26
«Опасно и обманчиво»: взгляд на исследования в области программной инженерии сквозь призму работ Дэвида Парнаса.....	31
«Серебряной пули нет»: взгляд на исследования в области программной инженерии сквозь призму работ Фредерика Брукса	34
Что думают умнейшие и лучшие	36
Некоторые выводы	38
Некоторые рекомендации.....	40
Заключение.....	43
Ретроспектива	43
2. Из окопов технологии	47
Когнитивный взгляд: проектирование ПО с другой точки зрения	48
Данные эмпирических исследований	48
Квинтэссенция проектирования.....	50
Некоторые другие выводы	52
Что же дальше?	53

Некоторые размышления об ошибках в ПО	55
Итоги размышлений	58
Экспериментальный аспект устранения программных ошибок	59
Выводы из экспериментов по тестированию ПО существуют	60
Данные, полученные в результате экспериментов	61
Точки над «и»	65
Прочие удивительные результаты	66
Многоликое тестирование	67
Качество ПО и сопровождение ПО. Какая связь?	71
Сопровождение ПО – это решение, а не проблема	73
Управление из одной точки	77
«Дружелюбный к пользователю» – модное выражение или прорыв?	80
Ретроспектива	83
3. Новейшие вооружения	85
МЕТОДОЛОГИИ	86
Повторное использование: готовые части ПО – ностальгия и дежавю	86
Автоматическое программирование – слухи с вечеринки?	93
Некоторые мысли о прототипировании	95
Стандарты и «блюстителю» стандартов: они действительно помогают?	99
ИНСТРУМЕНТЫ	102
Джентльменский набор разработчика	102
На всякий CASE: взгляд на последний «прорыв» в технологии программирования	107
Сравнение инструментов CASE и 4GL: что в сухом остатке?	110
Зачем писать компиляторы?	115
ЯЗЫКИ	118
Языки программирования высокого уровня: насколько высок их уровень?	118
Должны ли мы готовиться к доминированию 4GL?	121
Сомнения по поводу Кобола	125
Ретроспектива	128
4. Из штаба	131
МЕНЕДЖМЕНТ	132
Покорение вершин индустрии ПО	132
Новый взгляд на продуктивность ПО	135

Производительность и Теория G	138
Управление программными проектами – Теория W, принадлежащая Барри Ворму	140
Повышение производительности труда в индустрии ПО: кто чем занимается?	144
Метрики ПО: о громоотводах и накопленной напряженности	147
Как измерить качество: меньшее притворяется большим	148
Можно ли внедрить качество в программный проект	152
Легенда о плохом программном проекте	155
МАРКЕТИНГ	156
А вы купили бы у короля Людовика автомобиль с пробегом? ...	156
КОНСАЛТИНГ	159
Поднотная консалтинга	159
Какие прогнозы давали предсказатели раньше	162
Поддержка пользователей: все не так просто, как кажется	169
Ретроспектива	172
5. Из лабораторий	175
ИССЛЕДОВАНИЯ	176
Структурные исследования (немного ироничный взгляд)	176
Проблема информационного поиска	179
Неувязочка, или некоторые за и против (как будто больше не о чем!) ссылок	181
ПЕРЕДАЧА ТЕХНОЛОГИЙ	182
А что в следующем году? Как исследуют развитие технологий	182
Передача технологий ПО: процесс, обладающий многими недостатками, или неровная дорога к производительности	184
Мифология передачи технологий	188
ОБРАЗОВАНИЕ	190
Изучение ПО: новый источник информации	190
Открытое письмо профессору информатики	194
Ретроспектива	200
6. Арена после боя	203
Как компьютерная наука может стать настоящей наукой, а программная инженерия – настоящей инженерией	204
Моя блестящая (или нет?) идея, которую я назвал «решение задач»	206

Почему проваливаются программные проекты	209
Неспособность оценивать	210
Нестабильные требования	212
Капризы и заблуждения.....	214
Обзор.....	217
О том, насколько важны прикладные области	218
Не поможете ли вы мне найти это?.....	222
Ода вечно юному ПО	224
Ретроспектива	226
Эпилог	229
Подборка главных мыслей в этой книге.....	230
Приложение.....	234
Алфавитный указатель	238

Об авторе

Роберт Л. Гласс провел в залах вычислительных центров более 55 лет, начав с короткого трехлетнего периода работы в авиакосмической промышленности (в Северо-Американской авиационной компании) с 1954 по 1957 годы, что дает ему право называться одним из настоящих пионеров индустрии ПО.

После Северо-Американской он работал еще в нескольких авиакосмических компаниях (в Aerojet-General Corp., 1957–1965, и компании Boeing, 1965–1970 и 1972–1982). По большей части его работа заключалась в создании программных инструментальных средств, с которыми работали специалисты в прикладных областях. Участвовать в авиакосмическом бизнесе в то время было делом волнующим – это были пьянящие дни исследования космоса. Но работа в области вычислительной техники и программирования кружила головы еще больше. В обеих областях прогресс был стремительным, а перспективы – неземными!

Главный урок, усвоенный им за годы, проведенные в авиакосмической отрасли, состоял в том, что ему очень нравилась техническая сторона индустрии ПО, но быть менеджером он совсем не хотел. Он старательно вживался в роль технического специалиста, и это сильно повлияло на его карьеру двояким образом: а) его технические знания оставались актуальными и востребованными, но б) его компетентность как менеджера – и способность зарабатывать деньги (!) – соответственно уменьшилась.

Когда его способность продвигаться вверх достигла неизбежного предела, он предпринял фланговый маневр, перейдя на научную и преподавательскую работу. Он читал курс лекций по технике ПО аспирантам университета Сиэтла (1982–1987) и один год (1987–1988) проработал (занимаясь исключительно академической деятельностью) в Институ-

те программной инженерии (Software Engineering Institute – SEI). До этого, получив грант, он два года (1970–1972) занимался исследованиями инструментальных средств в Вашингтонском университете.

За годы научной и преподавательской работы он извлек еще один главный урок. Его разум с восторгом обратился к научной стороне техники программного обеспечения, но сердце так и осталось сердцем практика. Конечно, можно оторвать человека от его призвания, но нельзя вырвать призвание из его души. Вооружившись этой новой мудростью, он начал искать способ соединить академическую и практическую сферы вычислительной техники, перебросив мост через то, что он давно ощущал как «информационную пропасть».

И он нашел несколько способов. Многие из его книг (более 25) и статей (более 90) посвящены тому, как оценить открытия в вычислительной технике, сделанные учеными, и как внедрить в индустрию ПО те из них, которые имеют практическую ценность. (Это задача, бесспорно, нетривиальная, и именно она в значительной мере определяет уникальную и противоречивую природу его воззрений и печатных работ.) Читая лекции и проводя семинары, он сосредотачивается как на теоретических, так и на лучших практических достижениях, помогающих в реальной работе.

Этому же посвящен и его бюллетень *The Software Practitioner*, и более академический журнал *Journal of Systems and Software*, который он редактировал много лет для издательства Elsevier (сейчас он там почетный редактор). А также колонки, которые он ведет в таких изданиях, как *Communications of the ACM* и *IEEE Software*. Большинство его работ серьезны и уникальны, но изрядная их часть написана частично (а некоторые и полностью) в юмористическом ключе.

Так каковы же его наивысшие достижения в области вычислительной техники? В 1995 году шведский университет Линкопинга присвоил ему почетную степень доктора философии (Ph. D.). А в 1999 году он был избран членом Ассоциации вычислительной техники ACM (Association for Computing Machinery).

Роберт женат на Айрис Весси, которая занимается исследованием информационных систем, и у него четверо детей – двое родных и двое приемных.

С Робертом Л. Глассом можно связаться через редакцию *developer**.

Другие книги Роберта Л. Гласса

«Facts and Fallacies of Software Engineering»¹, 2003

«ComputingFailure.com: War Stories from the Electronic Revolution», 2001

«Computing Calamities» (редактор), 1999

«Software Runaways» (редактор), 1998

«In the Beginning: Personal Recollections of Software Pioneers» (редактор), 1998

«Software 2020», 1998

«Software Creativity»², 1995

«An ISO 9000 Approach to Building Quality Software» (в соавторстве с Осменом Оскарссоном (Osten Oskarsson)), 1995

«Measuring and Motivating Maintenance Programmers» (в соавторстве с Джеромом Б. Ландсбаумом (Jerome B. Landsbaum)), 1992

«Building Quality Software», 1992

«Software Folklore», 1991

«Software Conflict», 1991

«Measuring Software Design Quality» (в соавторстве с Дэвидом Н. Кардом (David N. Card)), 1990

«Software Communication Skills», 1988

¹ Роберт Гласс «Факты и заблуждения профессионального программирования». – Пер. с англ. – СПб.: Символ-Плюс, 2008.

² Роберт Гласс «Креативное программирование 2.0». – Пер. с англ. – СПб.: Символ-Плюс, 2009.

- «Computing Shakeout» (редактор), 1987*
- «Real-Time Software» (редактор), 1984*
- «Computing Catastrophes» (редактор), 1983*
- «Modern Programming Practices: A Report from Industry» (редактор), 1982*
- «Software Maintenance Guidebook» (в соавторстве с Рональдом А. Нойсеуком (Ronald A. Noiseux)), 1981*
- «Software Soliloquies», 1981*
- «The Second Coming: More Computing Projects Which Failed» (в соавторстве с Сью Деним (Sue DeNim)), 1980*
- «Software Reliability Guidebook», 1979*
- «The Power of Peonage», 1979*
- «Tales of Computing Folk: Hot Dogs and Mixed Nuts», 1978*
- «The Universal Elixir and Other Computing Projects Which Failed», 1977, 1979*

Вступительное слово ко второму изданию

Недавно я наводил порядок в одном из тех шкафов, куда редко заглядываю, и наткнулся на коробку с компьютерными журналами конца 80-х – начала 90-х годов XX века.

Как-то летом во всех журналах обсуждали одну и ту же актуальнейшую тему: какая платформа победит в войне графических пользовательских интерфейсов – Motif или Open Look. Споры относительно технических достоинств этих двух систем были увлекательными и обстоятельными.

Надо ли говорить, что они были абсолютно бессмысленными?

А в еще более старом журнале из доисторической эпохи CP/M и шины S-100 реклама новой СУБД обещала навсегда покончить с рутинной программирования. Отчеты и запросы к базам данных, говорилось в ней, можно будет формулировать на самом обычном английском языке, и программисты скоро станут не нужны.

Этого не случилось.

В таком же духе освещались и другие технические вопросы, жизненный цикл которых оказывался идентичным: официальное представление, лихорадочные ожидания, потрясающая реклама, продолжительные и обстоятельные дискуссии, за которыми следовало полное забвение, после чего в поле зрения возникала очередная новинка, идущая под флагом Next Big Thing¹.

¹ Next Big Thing – термин, означающий инновационный проект, реализация которого приводит к значительным изменениям в жизни общества. Например, таковыми были телефон и компьютер. – *Примеч. перев.*

На этом фоне, пестрившем новыми языками программирования, новыми методологиями, новыми платформами и т. д., большинство программных проектов, не укладываясь ни в сроки, ни в бюджеты, страдали функциональной недостаточностью. Ответы мы получали, но задавали неправильные вопросы.

Очевидно, что успехи в развитии технологий не помогали нам создать пристойный программный продукт в срок. Однако из журнала в журнал и из книги в книгу продолжаются разговоры, не поднимающиеся выше технологий низкого уровня.

К счастью, еще есть авторы, способные поддерживать более высокий уровень дискуссии. Роберт Гласс – один из них.

В центре внимания его очерков находятся реалии, с которыми приходится иметь дело практикам. И в них нет места ангелам, танцующим на остриях булавок. Большинство практических ситуаций бывают уникальны по крайней мере в одном аспекте, и Гласс понимает, насколько важен контекст. Вы не найдете в его книгах универсальных решений или советов на все случаи жизни.

Вместо всего этого вы отыщете в них информацию к размышлению.

Пьер Абеляр, философ, живший в XI веке, учил, что сомнение есть начало истинного знания. Сомнение ставит перед человеком вопросы, в поисках ответов на которые он и может обнаружить истину.

Перед вами очерки Роберта Гласса, посвященные конфликтам программного обеспечения. Читая их, вы найдете зерна сомнения, важные и не всегда очевидные вопросы и, кроме того, обретете собрата, как и вы находящегося в поисках.

Посулы новых технологий сомнительны. Найдите вопросы, которые кто-то обязательно должен задать. Подумайте над словами Роберта Гласса. И не останавливайтесь на этой книге. Прочитайте работы, которые цитирует Роберт. Прочитайте Фредерика Брукса и Дэвида Парнаса. И книги серии *Pragmatic Programmer*. Прочитайте другие книги и статьи Роберта Гласса.

И начинайте задавать правильные вопросы.

*Эндрю Хант (Andrew Hunt)
The Pragmatic Programmers, LLC
www.pragmaticprogrammer.com*

Октябрь 2005

Вступительное слово к первому изданию

Юмор – редкий гость в подавляющем большинстве технических книг. Но когда он в них появляется, то превращает чтение в удовольствие. Именно это нравится мне в произведениях Роберта Гласса. Его размышления о противоречиях, занимающих умы лидеров сообщества программистов и разработчиков, остроумны и полны сарказма. В своих суждениях, иногда весьма спорных, он проницателен и облекает их в занимательную форму. Он обладает чувством юмора и знанием человеческой природы. Он отличный рассказчик, обращающийся к анекдотам и притчам. Кроме того, Роберт Гласс – учитель, который дает уроки истории, этики и философии, увязывая их со своими суждениями и выводами.

По сути, эта книга посвящена конфликтам и управлению конфликтами. В ней программисты и разработчики ПО противопоставлены менеджерам. Теория противопоставляется практике, стабильность – изменениям, разговоры – действиям и обещания – результатам. В книге предпринята попытка показать нам все стороны, участвующие в споре, а также заставить нас задуматься и сделать собственные выводы.

Еще эта книга – об изменениях и об управлении изменениями. Она побуждает нас бросить вызов всеобщему убеждению в том, что изменения не могут происходить быстро. Она утверждает, что в ближайшем будущем никаких панацей не предвидится.

И кроме того, она призывает подумать, почему практика разработки ПО так сильно отличается от современной теории.

Главный тезис книги состоит в том, что конфликт полезен, если им правильно управлять, поскольку он заставляет профессионалов обдумыв-

вать свои идеи, прежде чем очертя голову бросаться внедрять их. Главный из описанных конфликтов – это конфликт между теорией и практикой. Объясняя, что ни одна из сторон этого конфликта не может добиться успеха, не сделав какой-либо вклад в пользу другой, Роберт принимает сторону практиков.

Я тоже практик и поэтому мне очень понятны и его взгляды, и примеры, которыми он свое мнение подкрепляет. Мои многочисленные баталии с преподавателями высших учебных заведений происходили на этой же арене. Я испытывал те же разочарования, о которых говорит Роберт. Он приводит те же аргументы, которые приводил я, чтобы выигрывать отдельные сражения, но мне так и не удалось победить в этой войне.

Эта книга дискуссионна, потому что она пытается развеять мифы, которыми пропитана компьютерная литература. По многим вопросам я согласен с Робертом, но между нами есть и серьезные разногласия, которые касаются технологии и ее развития. Я сторонник эволюционной теории развития, а Роберт предпочитает революции. Я стараюсь двигаться вперед постепенно, а Роберт ищет более быстрые решения. Ну что же, в этом и состоит цель книги – заставить людей задуматься и помочь им измениться.

Она очень нравится мне, хотя и не лишена недостатков. Во-первых, она слишком длинная. Роберт пытается так много сказать, что нередко чересчур долго формулирует главную мысль. Во-вторых, она слишком короткая. Стоит вам только поверить, что вот сейчас Роберт покажет решение, как он останавливается и заставляет вас думать самостоятельно.

Я еще не сказал, что советую прочитать эту книгу? Извлечь из нее пользу смогут все – и менеджеры программных проектов, и университетские преподаватели, и специалисты по теории вычислительной техники, и программисты, и специалисты по технологиям. Подумайте над словами Роберта Гласса и взвесьте их. Уверен, вы поймете их значение и оцените по достоинству. Не понимайте сказанное им буквально. Роберт мастерски умеет захватывать внимание читателя и заставлять его размышлять. А если вы с ним не согласны, то это даже к лучшему. По крайней мере, у вас появится собственное мнение по каким-то вопросам, важным в области программной инженерии, как мы вскоре увидим.

*Дональд Дж. Рейфер (Donald J. Reifer),
президент Reifer Consultants, Inc.*

Предисловие к первому изданию

Конфликты и противоречия в вычислительной технике и разработке ПО слишком долго находились на периферии внимания специалистов. Возраст этих областей пока не превышает продолжительности жизни одного человеческого поколения, поэтому им свойственны все несовершенства, сопутствующие юности. Однако в профессиональной литературе обнаруживается тенденция выдавать мнения за истину, а пропаганду – за факты; при этом ничто не указывает на умозрительную природу некоторых из этих «фактов» и значительной части «истин». Даже Дэвид Парнас, известный в отрасли специалист, окрестил многие истины компьютерной науки «фольклором», потому что они не были подтверждены экспериментально.

Нельзя сказать, что конфликтов и противоречий нет в нашей отрасли. Если N специалистов по вычислительной технике или разработчиков ПО обсуждают какой-либо вопрос, то нередко получается N правильных или лучших решений. (Об этом лучше всех сказал Билл Кертис в своем докладе на Международной конференции по программной инженерии в 1989 году: «Если двое из пятнадцати разработчиков, собравшихся вместе, пришли к согласию, то они образуют большинство!»)

Но по большей части эти конфликты происходят в залах конференций или заседаний и не переходят в полемику на страницах книг и в статьях. Замалчивание спорных вопросов иногда имеет нехорошие последствия, как это было с подспудным несогласием по поводу значимости формальной верификации, которое привело к некрасивым личным выпадам в журнале *Communications of the ACM* в 1989 году¹. (Предыдущий взрыв

¹ Этот эпизод спровоцировала статья «Program Verification: The Very Idea» (Верификация программ: идея, которую мы искали), написанная Джеймсом

отрицательных эмоций по этому поводу, который был точно таким же несдержанным, случился почти на десять лет раньше и нашел отражение во введении к моей книге «Software Soliloquies».)

Уход от разрешения спорных вопросов – это ошибка. Встречая конфликт должным образом и лицом к лицу, можно достичь прогресса. Не разрешенный конфликт способен привести к поляризации, застою и, как мы уже видели, к взрыву. Эта отрасль еще слишком молода, чтобы мы могли позволить себе полагаться на свои незрелые представления и взгляды. О главных противоречиях необходимо говорить публично и разрешать их, пока они не привели к открытым конфликтам.

Часто эти противоречия приобретают форму разногласий между теорией и практикой. На самом деле конфликт порождается распределением ролей теории и практики в прогрессе отрасли в целом.

Некоторые из этих конфликтов и являются предметом данной книги. И хотя слово дается обеим сторонам, предпочтение в ней отдается точке зрения практиков. Я практик, обладающий 30-летним опытом, и я достаточно долго вращался в преподавательской среде, чтобы иметь право сказать: «Умом я принадлежу к преподавателям вычислительной техники, но сердцем я практик программной инженерии».

Конфликты рассматриваются в серии очерков, сгруппированных по нескольким главным темам.

Генри Фетцером (James H. Fetzer), профессором философии и гуманитарных наук из университета Миннесоты в Делуте (Deluth). Статья, в которой формальная верификация была представлена в крайне невыгодном свете, была напечатана в *Communications* в сентябре 1988 года.

Самая сильная реакция на эту статью последовала от группы из десяти приверженцев формальных методов в мартовском номере этого же журнала за 1989 год, которые обвинили Фетцера в «искажении фактов», «вопиющем непонимании», «неправильном представлении», «дезинформации» и, наконец, в том, что он «не владеет информацией, безответствен и опасен». Фетцер ответил залпом из всех орудий, утверждая, что его оппоненты «абсолютно неспособны понять суть вопроса», «страдают умственной отсталостью», «непростительно нетерпимы, а их уверенность в собственной правоте невыносима» и что они ведут себя, как «религиозные фанатики и идеологические полицейские». Он даже предложил им «поучаствовать в полях... реактивных снарядов, чтобы продемонстрировать всем... что такое верификация в динамической среде».

Дополнительная порция огнестрельных любезностей содержалась в августовском выпуске *Communications* того же года. Не придется удивляться, если в будущем этот конфликт вспыхнет снова.

Обзор театра военных действий. Представлены очерки по концептуальным вопросам, например анализы работ Парнаса «Star Wars» (Звездные войны) и Брукса «No Silver Bullet» (Серебряной пули нет). Есть и не совсем обычный анализ исторических взаимосвязей между теорией и практикой.

Из окопов технологии. Очерки этой серии посвящены технологии. В одном из них, например, обсуждается проектирование ПО, в частности вопрос о том, почему мы фокусируем внимание на внешних атрибутах проектирования (методологиях и способах представления), а не на его сути (когнитивной деятельности).

Новейшие вооружения. В этих очерках рассказывается о методологиях, инструментах и языках. Так, один из них содержит, во-первых, обзор литературы по методам количественной оценки инструментария CASE, языков четвертого поколения, а также по другим подходам, нацеленным на повышение производительности, и, во-вторых, некоторые удивительные выводы.

С командного поста. Очерки о менеджменте, маркетинге и консалтинге. В четырех из них исследуется актуальная ныне тема повышения производительности и вопрос о корнях нашей отрасли и ее будущем.

Из лабораторий. Очерки посвящены внедрению открытий, сделанных в результате исследований. В одном из них, представляющем собой открытое письмо профессорам, которые читают курсы по вычислительной технике, обсуждается недостаточное освещение сопровождения ПО в учебных программах. В другом говорится об информационном разрыве между теорией и практикой.

Арена после боя. Подведение итогов. В одном из очерков указывается, чего именно не хватает теоретическим исследованиям в области вычислительной техники и программной инженерии, предлагаются способы улучшения ситуации. Другой посвящен юмору, шуткам и розыгрышам, изначально присущим нашей отрасли и, теперь это очевидно, покинувшим ее.

Книга завершается эпилогом, суммирующим ее главные идеи под заголовком «Квинтэссенция». (Не поддавайтесь соблазну заглянуть в него немедленно!)

Эта книга адресована тем, кто обеспокоен судьбой отрасли программной инженерии – и практикам, и ученым, и университетским преподавателям. Удовлетворить запросы такой аудитории непросто. В попытке сделать это я нередко прибегал к смешанному стилю изложения, выражаясь серьезно, но (я надеюсь на это) в то же время занимательно, пересыпая повествование метафорами и притчами. Иногда имена реальных людей и названия мест и событий заменялись вымышленными.

В таком же духе был выдержан и мой предыдущий сборник очерков «Software Soliloquies» (Монологи о ПО), выпущенный издательством Computing Trends в 1981 году.

Итак, с вводной частью покончено, и я приглашаю вас к чтению. Надеюсь, оно не оставит вас равнодушными. Станьте моими единомышленниками или оппонентами, но только не будьте безразличными. Конфликты и разногласия в нашей области заслуживают лучшей участи, чем игнорирование под каким-нибудь удобным предлогом.

Роберт Л. Гласс

1990

Предисловие ко второму изданию

Если вы читали предисловие к первому изданию, то знаете, о чем эта книга. В ней обозначаются глубинные конфликты программной инженерии и каждый подробно рассматривается с критической точки зрения, наиболее близкой разработчику ПО.

Однако в упомянутом предисловии плохо то, что оно было написано для книги, выпущенной более 15 лет тому назад. А почему, собственно, должна представлять хоть какой-то интерес книга пятнадцатилетней давности, посвященная разработке ПО, пусть даже и обновленная? В конце концов, эта область развивается настолько быстро и так технически насыщена, что имеет тенденцию предавать забвению образ мысли, рассуждения и выводы, которым больше 5 лет. Что заставляет меня, автора этой уже пятнадцатилетней книги, думать, что вы найдете ее заслуживающей хоть какого-то внимания?

Честно говоря, когда передо мной замаячила перспектива обновления старой версии «Software Conflict», я испытал именно эти сомнения. Ведь не только материалу исполнилось уже более 15 лет, я сам изменился и написал с тех пор больше полудюжины книг, причем в каждой мои взгляды и рассуждения были созвучны (а как еще?) своему времени. Стоит ли этот проект времени, которое я затратчу, излагая свои мысли, и времени, которое вы потратите на то, чтобы их прочитать?

Чтобы ответить на данный вопрос, я перечитал это безнадежно устаревшее издание с новой точки зрения, освеженной идеями XXI века. И был поражен тем, что большая часть материала сохранила свою новизну и актуальность.

Как это может быть? Как человек, изучающий программную инженерию и разработку ПО, я смотрю на большинство вопросов с концепту-

альной, философской точки зрения. И мне не очень интересны вопросы из серии «Как это сделать?». Я предпочитаю оставаться в стороне от современных войн языков и платформ.

Я считаю, что в программной инженерии мудрость достижима, если отступить на шаг, поднять голову над суетой детализированности и копнуть глубже. Именно это, как я обнаружил, я и сделал в данной книге.

О да, в языках программирования (например, в Ада), методологиях (в структурных методах) и инструментах (CASE-программах) есть некоторые средства низкого уровня, заставляющие меня краснеть из-за их устарелости. Но в то же время эти низкоуровневые средства сейчас, как и тогда, не представляют собой ничего существенного. Зато не потеряли своего значения концепции и философия, ну и сопутствующие им (куда ж без них) конфликты.

Таким образом, эта древняя книга все еще содержит очень актуальные мысли, и это сюрприз, который радует меня и послужит вам. К каждой серии очерков я добавил ретроспективное резюме, поясняющее, что именно в них, по-моему, устарело, а что выдержало испытание временем.

Я бы не предложил вашему вниманию это обновленное издание, если бы не думал, что оно будет интересно и полезно. Судить об этом вам. Надеюсь, книга «Программирование и конфликты 2.0» будет принята своими читателями так же, как «Software Conflict» образца 1991 года была принята своими.

Сказать по правде, за эти 15 лет появилось не так уж много литературы, помогающей идентифицировать и разрешить конфликты, о которых шла речь в первом издании. И в этом смысле книга «Программирование и конфликты 2.0» по-прежнему актуальна. К несчастью, так же актуальна, как и ее предшественница.

Роберт Л. Гласс

Осень 2005

Глава 1 | Обзор театра военных действий

- Что было раньше: Теория или Практика?
- «Опасно и обманчиво»: взгляд на исследования в области программной инженерии сквозь призму работ Дэвида Парнаса
- «Серебряной пули нет»: взгляд на исследования в области программной инженерии сквозь призму работ Фредерика Брукса
- Что думают умнейшие и лучшие
- Ретроспектива

Что было раньше: Теория или Практика?

*Теория: формулировка принципов,
лежащих в основе чего-либо.*

*Практика: в отличие от теории;
действия, которые следует предпринимать
в ходе профессиональной деятельности.*

Oxford American Dictionary, 1980

Значения слов «теория» и «практика» ясны и общеприняты – это позволяет нам достаточно хорошо понимать, что имеют в виду люди, употребляя их. Но как быть с временными соотношениями этих двух понятий, то есть с чего все начинается – с теории или с практики?

Для большинства из нас, прошедших лет десять, а кто и больше, внутри образовательной системы, ответ, скорее всего, очевиден. Теория предшествует практике и формирует ее. Но этот автоматический ответ может быть подвергнут суровой критике, и он может привести к глубокому непониманию.

Вот, например, что сказал Кристофер Александер (Christopher Alexander) в своем труде «Notes on the Synthesis of Form» (Заметки о синтезе формы), Harvard University Press, 1964:

Особый профиль крыла, благодаря которому могут летать самолеты, был изобретен как раз тогда, когда было «доказано», что никакой аппарат тяжелее воздуха летать не может. Его аэродинамические свойства не были поняты, пока оно не побывало некоторое время в эксплуатации. Действительно, скорее изобретение и применение профильного крыла внесло важный вклад в развитие аэродинамической теории, а не наоборот.

Таким образом, по Александеру, практика идет впереди теории. Есть ли еще доказательства истинности этого, может быть, удивительного тезиса?

Вот еще одно высказывание, поразительно похожее на слова Александера. Оно принадлежит Дерексу Прайсу (D. D. Price), который написал в статье «Sealing Wax and String: A Philosophy of the Experimenter's Craft and its Role in the Genesis of High Technology» (Сургуч и шпагат: философия экспериментаторства и его роль в возникновении теории), Proceedings of the American Association for the Advancement of Science Annual Meeting, 1983:

Термодинамика обязана паровой машине в значительно большей мере, чем паровая машина – термодинамике... Если взглянуть на обычный исторический ход событий,... то случаи, когда технология оказывается прикладной наукой, очень немногочисленны. Гораздо чаще наука оказывается прикладной технологией.

Два примера опережения теории практикой не могут доказать даже одну версию, не говоря уже о том, чтобы свидетельствовать о тенденции, однако работа Прайса наводит на мысль, что такая тенденция может существовать. А есть ли еще примеры?

Герберт А. Саймон (Herbert A. Simon) в своей книге «The Sciences of the Artificial», 2nd ed., The MIT Press, 1981,¹ сказал:

...главный способ разработки и усовершенствования систем с разделением времени состоял в том, чтобы построить их и посмотреть, как они себя поведут. Именно это и было проделано. Последовательно строили, модифицировали и усовершенствовали системы методом последовательных приближений. В принципе, теория могла бы предсказать результаты этих экспериментов, сделав их излишними. Но она этого не смогла, и я не знаю никого достаточно близко знакомого с этими исключительно сложными системами, кто обладал бы знанием, как это сделать. Чтобы понять эти системы, нужно было их построить и понаблюдать за их поведением.

Благодаря Саймону мысль о том, что практика предшествует теории, начинает становиться все более правильной и в нашей области – в вычислительной технике и программировании. Поговорим еще о теории и практике применительно к индустрии ПО.

Я вырос в окружении практиков этой отрасли и помню многие события, под влиянием которых она формировалась. Сначала практика, потом теория – эта точка зрения для меня естественна. Компьютер, конечно, уходит корнями в первые исследовательские лаборатории различных организаций, рассеянных по Северной Америке и Европе. И к середине 1950-х начался бурный расцвет вычислительной техники и программирования как профессиональных областей. Если практика и не была впереди теории на старте, то обогнала ее по мере развития отрасли. И академическая составляющая вычислительной техники возникла лишь в 1960-х. А теория, которую породили эти первые академические изыскания, не давала о себе знать вплоть до конца 1960-х – начала 1970-х годов.

Таким образом, личные воспоминания старожил отрасли делают безусловно правдоподобным тезис о первенстве практики перед теорией.

¹ Герберт Саймон «Науки об искусственном», 2-е издание. – Пер. с англ. – М.: Издательская группа УРСС, 2004..

Однако опыт, конечно же, может быть ненадежным учителем. Личный опыт одного человека (то, что он испытывает) может сильно отличаться от опыта (ощущений) другого. А есть ли какой-нибудь другой способ изучить этот вопрос?

До сих пор я говорил о вычислительной технике и программном обеспечении так, будто это единая отрасль знания, которую можно изучать как целое. А что если поступить наоборот, взглянуть на некоторые элементы, составляющие это целое? Не сможем ли мы увидеть благодаря такому подходу, как появляется знакомая нам пара – практика и теория?

А что можно сказать о стилях программирования? Что было сначала: практика или теория? То есть само собой, что раньше была написана уйма программ, а потом появились книги о стиле программирования. Отлично помню, что стиль некоторых программ был превосходен, тогда как другие этим не могли похвастаться. Я бы сказал, что стиль уже был очень хорошо отточен на практике, прежде чем появилась теория стиля. (Интересно отметить, что первые книги по стилю, «Proverbs...», написанные Генри Ледгардом (Henry Ledgard), по сути, представляли собой кодификацию «правил хорошего тона».)

А написание компиляторов? Компиляторы для таких языков, как FORTRAN, Commercial Translator, FACT, а позже и для языка Кобол, были написаны практиками задолго до того, как появилась солидная книжная теория создания компиляторов. Повторюсь, что написание компиляторов, сейчас являющееся центром академической науки о вычислительной технике, было очень неплохо разработано на практике, прежде чем начала появляться соответствующая теория.

И в наши дни не надо далеко ходить за примерами руководящей роли практики по отношению к теории. На системном уровне для выработки требований и поиска решения сложных проблем нередко применяется такой инструмент, как симуляция. Однако исследователи вычислительной техники редко обращаются к теме симуляции. Недавний интерес, который они проявляли к прототипированию, – это не совсем то же самое. (Заметьте, что симуляция, применяемая на практике, представляет собой выработку умозрительного практического решения с целью создать теорию проблемы. Даже здесь практика ведет теорию за собой!)

Проектирование пользовательских интерфейсов хоть и уходит теоретическими корнями в исследовательский центр Xerox PARC, к настоящему времени демонстрирует намного более быстрый прогресс в области компьютерной техники (особенно миниатюрной), чем теория.

И теория этой области до сих пор еще плохо изучена. Курсы по проектированию интерфейсов часто уделяют главное внимание методо-

логии и внешнему представлению, однако большинство специалистов понимают, что этот предмет намного сложнее, чем банальное определение последовательности определенных действий и способа ее записи. Практика проектирования интерфейсов намного опережает его теорию.

На самом деле вся система подходов к решению задач, чем, собственно, и занимается программная инженерия, все еще пребывает на ранних стадиях создания теории (см., например, уже цитированную книгу Саймона «The Sciences of the Artificial»), хотя практическим решением задач (подобных проектированию) человек занимается уже многие века.

Иначе говоря, подобных примеров хоть отбавляй. Вас это удивляет? А меня удивляет, и опыт не спасает меня. Весь мой академический опыт кричит, что это теория создает платформу, на которой практика может что-то строить. Если эта идея неправильна, то, может быть, надо проверить выводы.

Попробуем построить график взаимоотношений между теорией и практикой, которые подразумевает эта идея. На рис. 1.1 вы видите график, который получился у меня. Я считаю, что в любой конкретной области все начинается с практики, которая на первых порах развивается довольно быстро. Теория начинается тогда, когда уже накоплен практический опыт, который можно формализовать, и тоже быстро развивается. Как видно, в некоторой точке линии пересекаются, и теория начинает обгонять практику.

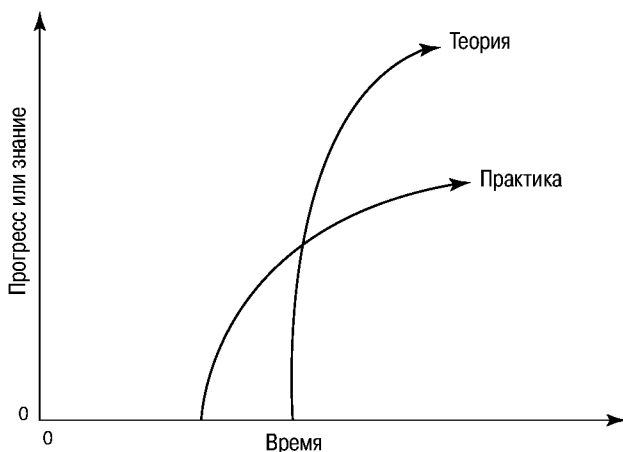


Рис. 1.1. График взаимоотношения между теорией и практикой

Допустим, что это так и есть. Какие выводы можно сделать из этого?

Прежде всего, из этого графика следует, что на ранних стадиях развития дисциплины теория быстрее всего прогрессирует благодаря изучению опыта практиков. Конечно, принятые в прошлом способы и методы не должны мешать теоретику генерировать и формулировать новые идеи, однако изучение практического опыта может научить очень многому, особенно если это лучший практический опыт. Это важная мысль. По большей части развитие теории вычислений и программного обеспечения не всегда соответствовало ее сути. Не многие теоретики в этой области раньше были практиками. В ней не нашел широкого распространения экспериментальный подход, подразумевающий исследование, имитирующие практическую деятельность. И уж совсем немногочисленны исследования практиков, занятых разработкой новых теорий (если не говорить о некоторых эмпирических исследованиях, принимаемых программистами).

Согласно этой мысли должен измениться по крайней мере ранний подход к теории. Сокровищница практических знаний слишком богата, чтобы теория ее игнорировала.

Из этого графика можно сделать и другой вывод: начиная с того момента, когда теория превосходит практику, последняя должна к ней прислушиваться. К примеру, теоретическая платформа в таких областях, как базы данных и структуры данных, намного превосходит знания большинства практиков. Состояние практики пока еще не достигло этой точки. И точно так же, как теория не занимается изучением практики, когда для этого наступает время, практика не слушает теорию вовремя.

Иначе говоря, для взаимодействия между теорией и практикой характерны фундаментальные проблемы, на которые этот график может пролить свет. А непонимание следствий из него имеет фундаментальное значение для современного положения дел и практики в вычислительной технике и программировании.

Возможно, этот график слишком упрощает картину. Если изобразить процесс как можно точнее, то развитие практики и теории будет больше напоминать переплетающиеся линии, показывающие, как практика и теория по очереди идут впереди. Но и на некоторых участках этой более сложной картины (что-то вроде увеличенных моментальных снимков) упрощенный график будет правомерным.

Предшествует ли практика теории? На некоторых уровнях и в некоторые моменты времени – да. Сейчас и практике, и теории пора усвоить выводы из этого факта.

Благодарности: Автор выражает признательность Айрис Вессе (Iris Vessey) и Дейлу Даусингу (Dale Dowsing) за помощь в развитии этих идей.

«Опасно и обманчиво»: взгляд на исследования в области программной инженерии сквозь призму работ Дэвида Парнаса

В течение некоторого времени бушевала полемика вокруг программы СОИ (стратегическая оборонная инициатива США), известной также под названием «звездные войны». Последующее обсуждение, однако, не имеет отношения именно к ней. Оно посвящается тому, как в результате этой полемики изменился взгляд на ПО и само ПО в целом.

Вам может быть известно, что профессор Дэвид Парнас (David Parnas), один из ведущих современных научных деятелей в области вычислительной техники (компьютерных наук), отказался от участия в реализации программы «звездных войн», потому что был уверен в непреодолимости проблем, связанных с ПО подобной системы.

Может быть, вы читали статьи Парнаса, опубликованные в журнале *American Scientist* (в сентябре–октябре 1985 года) и перепечатанные различными специальными компьютерными периодическими изданиями. В тех статьях он объяснял, почему не верит в возможность успешного создания компонентов программного обеспечения этой системы.

Однако я хочу взглянуть на слова Парнаса с другой точки зрения. Анализируя *причины*, по которым он считал невозможным создание этой системы, он высказал некоторые просто-таки уничтожительные соображения о современном состоянии ПО.

При этом он не говорил ничего плохого о состоянии *практики* создания ПО. На практику нападали, и, к несчастью, довольно часто, некоторые ученые-информатики, утверждавшие, что программное обеспечение «ненадежно, а его создатели никогда не укладываются ни в бюджет, ни в сроки». Ну, об этих словах у вас, наверное, есть свое мнение.

Он высказывался критически о состоянии *исследований* в области создания ПО. В своих статьях Парнас по пунктам объяснял, почему он считает, что ни одно из ведущих исследовательских направлений не поможет реализовать программу СОИ. И он сильно скомпрометировал это исследование. Рассмотрим некоторые из его высказываний.

Вам, вероятно, приходилось читать, что для резкого увеличения производительности ПО осталось создать всего лишь один новый язык или новый набор инструментальных средств. Парнас утверждает, что это не так: «Мы не можем думать, что [благодаря новым языкам программирования]... что-то сильно изменится» и «главным препятствием в нашей... работе не были проблемы, связанные со средой программирования».

Ну хорошо, а что же так называемые системы автоматического программирования, эти методологии, которые должны были через несколько лет сделать программистов отжившей категорией? «Я уверен, что заявления по поводу систем автоматического программирования неумеренно оптимистичны», – говорит Парнас. И продолжает: «если только входная спецификация не является описанием алгоритма, то результат просто никуда не годится... наши теперешние возможности не претерпят значительного изменения» благодаря непроцедурным, автоматизированным системам программирования.

Но тогда, может быть, резко улучшить ситуацию нам поможет искусственный интеллект? И здесь тоже Парнас не выражает энтузиазма. Известны, по словам Парнаса,

два совершенно разных современных определения искусственного интеллекта (ИИ)...

ИИ-1: Применение компьютеров для решения задач, которые до этого были под силу только человеческому разуму.

ИИ-2: Применение... программирования на основе правил... для решения задачи по алгоритму, которым руководствуется человек.

«Я видел выдающиеся примеры работы ИИ-1, – сказал Парнас, – но я не могу выделить группу методов... присущих только этой области». Иначе говоря, опыт обучения, полученный при решении одной задачи методами ИИ-1, не слишком хорошо распространяется на следующую задачу.

«Я нахожу, что подходы, принятые в ИИ-2, опасны, а сама работа по большей части дезориентирует... поведение программ недостаточно понято и его трудно предсказать... методики... не обобщаются». Здесь Парнас не оставляет камня на камне от объекта своей критики. По сути, он утверждает, что экспертные системы не заслуживают большого доверия.

Обеспокоен Парнас и надежностью программного обеспечения. Он говорит: «Мы не знаем, как обеспечить надежность ПО». А как же доказательство корректности ПО, действующее математический аппарат и призванное доказать соответствие программной системы ее спецификации? «Для меня непостижимо, как можно убедительно доказать корректность даже небольшого фрагмента [крупной] программной системы... и я ума не приложу, что означало бы такое доказательство, если бы я им располагал». Он говорит: «Мы не располагаем методами, которые позволили бы нам доказывать корректность программ при условии, что имеют место непредсказуемые отказы оборудования и ошибки во входных данных».

В общем, похоже, что Парнас в своих статьях резко критикует положение дел в исследованиях ПО. Обоснована ли эта критика?

«Добротная программная инженерия – это далеко не просто», – говорит он. «Те, кто думает, что проектирование ПО станет легким делом [благодаря новым технологиям] и что ошибки исчезнут, просто не сталкивались с настоящими задачами». «Я не думаю, что задача [построения крупных программных систем] будет решена в следующие 20 лет исследовательской работы». «Очень немногие работы [по исследованию ПО] приводят к полезным результатам. Многие полезные результаты остаются незамеченными, потому что хорошие работы тонут под массой остальных».

А что делать, если в каких-то сомнительных областях исследования сворачиваются? Парнас говорит и об этом.

Только те, кто очень хорошо знаком с прикладными аспектами вопроса, могут решать, имеют ли практическую пользу результаты исследовательского проекта... Судить о результатах должны группы специалистов, в том числе успешные исследователи и опытные системные инженеры.

Другими словами, если исследователь хочет получить полезные результаты, то при их оценке он должен прибегать к помощи практиков.

А теперь проанализируем слова Парнаса еще раз. Конечно же, их важно рассматривать в контексте крупных программных систем реального времени, которые должны были поддерживать глыбу «звездных войн» (стратегической оборонной инициативы США). Не менее важно, однако, иметь в виду, что многие из его возражений по поводу ценности текущей исследовательской работы относятся к использованию более широкого спектра крупных (и не только) программных систем.

Где, на каком направлении может произойти прорыв в увеличении производительности ПО – такой, который приведет к ее увеличению на порядки?

Не в языках программирования или в инструментальных средствах.

Не в технологиях автоматического программирования.

Не в формальной верификации программ.

Не в системах искусственного интеллекта.

Также не может привести к такому прорыву ни одно из популярных сейчас исследовательских начинаний в сфере ПО.

Для того, кто занимается в ней практической деятельностью и лелеял надежды на радикальные улучшения, это обескураживающее наблюдение.

Оно же дает богатую пищу для размышлений исследователю ПО.

«Серебряной пули нет»: взгляд на исследования в области программной инженерии сквозь призму работ Фредерика Брукса

В течение нескольких последних десятилетий исследования ПО обещают радикально повысить его производительность. А менеджеры ИТ-индустрии, отчаянно желая верить ученым, с готовностью хватались за каждую новую разработку.

Для большинства менеджеров результатом было разочарование. Каждая новая идея ученых, когда ее проверяли, конечно же, повышала производительность ПО. Неприятность заключалась в том, что улучшения носили настолько частный характер, что менеджеры начинали недоумевать: «По какому поводу, собственно, столько хлопот?»

Исследователи слышали об этом, но не очень внимательно слушали. Вместо того чтобы попытаться понять, почему их идеи не приносят ощутимых плодов, они, как правило, реагировали в том смысле, что, мол, практики не восприняли их идеи по-настоящему и что никто не удосуживается добросовестно проверять ход научных изысканий в сфере ПО.

Сейчас начинает формироваться новая точка зрения по этому вопросу. Причем ее создают лучшие и ярчайшие умы ИТ-отрасли.

Первым был Дэвид Парнас. В своих статьях, критикующих проект «звездных войн», Парнас также обрушился на исследования в сфере ПО, указывая, что они не привели – а на деле и не могли привести – к прорывам в практической деятельности, и называя некоторые из них «опасными и обманчивыми».

Данная точка зрения впоследствии была усилена. Это сделал Фредерик Брукс – менеджер, отвечавший за крупнейший проект OS/360. Он написал единственную в своем роде и почти наверняка самую значимую книгу о разработке ПО «The Mythical Man-Month»¹.

Свою позицию он изложил в статье, напечатанной в *IEEE Computer* в апреле 1987 года. Она быстро получила известность под названием «Silver Bullet», потому что в ней фундаментальные проблемы программной инженерии сравнивались с оборотнями, которые умирают только от серебряных пуль. Далее в этой статье Брукс утверждал, что в программной инженерии серебряных пуль нет.

¹ Фредерик Брукс «Мифический человек-месяц, или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

Что же сказал Брукс? Очень многое. К примеру, что процесс построения ПО труден и будет таким всегда. Он труден из-за органически присущей ему сложности: «вероятно, сущности, которыми оперирует программа, сложнее... чем любые другие конструкции, созданные человеком», «программные системы могут иметь на несколько порядков больше состояний, чем компьютеры», и «сложность – это неотъемлемое качество ПО», от которой нельзя избавиться методами других отраслей знания, например математическими, в которых упрощенные модели сложных систем являются полезными инструментами решения задач.

Сформулировав постулат о том, что ПО создавать трудно и что так будет всегда, Брукс проанализировал направления исследований теоретиков и, так же как и Парнас до него, обнаружил, что эти направления нуждаются в коррекции. По мнению Брукса, дело было не в том, что направления бесперспективны, а в том, что большой прогресс в повышении производительности уже был достигнут благодаря успехам первого языка высокого уровня, механизмам разделения времени и существованию унифицированных сред программирования, таких как UNIX и Interlisp. Успехи, которые могут быть достигнуты теперешними поисками серебряных пуль, окажутся куда менее значимыми. Брукс предполагает, что момент впечатляющего прогресса в программной инженерии уже позади.

Но на этом Брукс не останавливается. Если не получается существенно повысить производительность благодаря текущим исследованиям, то как это сделать? Он предлагает более практичные и более перспективные способы.

1. При любой возможности покупать ПО, а не разрабатывать его. Брукс считает, что в 1990-х годах пакеты ПО имели большую практическую ценность, чем в 1960-х, не потому, что за тридцать лет мы больше узнали о создании таких пакетов, а потому, что относительная стоимость ПО стала настолько высокой (по сравнению со стоимостью оборудования), что скорее фирмы подстраивают свои функции к пакетам ПО, а не наоборот.
2. Придерживаться инкрементного подхода при создании ПО. Уточнять требования к ПО при помощи прототипирования. Разрабатывать ПО инкрементно, чтобы иметь частичные решения задачи на ранних стадиях цикла разработки. Если процесс создания ПО действительно труден и всегда будет таким, то для него необходимо разрабатывать новые фундаментальные подходы.
3. Помнить, что выдающиеся проекты ПО создаются талантливыми проектировщиками. Индустрия ПО нуждается в творческих личностях, их надо нанимать на работу, возвращать и лелеять. Брукс счи-

тал, что такая поддержка позволит: (1) выявлять лучших проектировщиков как можно раньше («лучший» не всегда то же самое, что «самый опытный»); (2) прикрепить к каждому перспективному дарованию профессионального наставника; (3) продумать, составить и отслеживать выполнение плана карьерного роста каждого такого специалиста и (4) обеспечить им возможность общаться и помогать друг другу.

Так что же все-таки сказали нам Парнас и Брукс? Что разработка ПО – в принципе очень крепкий орешек и что волшебные палочки на дорожке не валяются. Что практикам пора заняться эволюционными подходами к улучшению дел и не ждать революций (прорывов) и не надеяться на них.

Некоторым в индустрии ПО эта картина кажется обескураживающей. Это именно те, кому казалось, что до победного рывка рукой подать.

Однако те из нас, у кого достаточно толстая шкура, чтобы считать себя реалистами, могут не терять надежды. В конце концов, мы способны сосредоточиться на чем-то более осязаемом, чем воздушные замки. Наверное, мы сумеем примириться с постепенными реальными улучшениями производительности ПО и не будем ждать мифических революционных достижений.

Скажем спасибо Дэвиду Парнасу и Фредерику Бруксу за то, что они проложили путь в этом направлении.

Что думают умнейшие и лучшие

*«Мы не думаем, что в ближайшие десять лет
появится хотя бы одна технологическая разработка,
сулящая десятикратный прирост производительности ПО,
повышение его надежности и своевременности».*

*«Мало в каких еще областях столь велик разрыв
между лучшими и средними образцами работы».*

*«Главные современные проблемы лежат
не в технической сфере... а в сфере менеджмента».*

Это цитаты из вступительной части дельного, отлично написанного отчета специальной научно-исследовательской группы Министерства обороны США по военному ПО «Defense Science Board Task Force on Military Software», опубликованного в конце 1980-х. В эту группу, которую возглавил знаменитый Фредерик Брукс, входили специалисты, которых Брукс назвал «высококвалифицированным и неутомимым

ядром» и которые избрали своей целью решение актуальных в то время проблем ПО.

Должен ли средний программист, пишущий приложения обработки данных, интересоваться отчетом, в котором обсуждаются проблемы *военного* ПО? Учитывая, что отчету уже не один год, можно ли считать его по-прежнему актуальным с этой точки зрения?

Принимая во внимание его охват, глубину и своевременность, я не колеблясь дам *утвердительный ответ*. Его авторы встретили и сразили целый выводок драконов, обитающих на территории ПО, и еще столько же они полностью разоблачили. Люди, подготовившие этот доклад, вероятно, относятся к лучшим и ярчайшим умам этой отрасли.

Над этим докладом и данными, которые в нем опубликованы, я размышлял с тех пор, как он был напечатан, то есть с сентября 1987 года. И стал считать его «последним словом», характеризующим состояние отрасли ПО в конце 1980-х. Я не могу согласиться со всеми его выводами, но, даже не соглашаясь, нахожу, что они дают очень сильный импульс к действию. Надеюсь, что вы найдете приведенное здесь краткое изложение этого доклада таким же стимулирующим.

Просто потрясает избранный в нем подход к рассмотрению как ведомственных проблем, над которыми на протяжении многих лет бились и военные, и их подрядчики, так и проблем, связанных с переходом на новые концепции управления и технологии. В докладе указаны главные направления, по которым Министерству обороны США рекомендовалось изменить схему создания и поставок ПО, особенно в таких старых традиционных областях, как ротация персонала, и вообще измениться, «строая карьеру отдельных должностных лиц так, чтобы они формировали костяк группы технических менеджеров, которые бы в совершенстве владели предметом и обладали широким оперативным кругозором». (Сейчас многие военные чиновники переходят на другую должность каждые два года и, конечно, не могут в совершенстве овладеть какой-либо технической областью.)

В этом отчете, призванном привлечь внимание военного ведомства, которое уже проигнорировало массу подобных исследований, нет никаких натяжек. Отчет написан ярко и убедительно, содержит короткие и содержательные высказывания, которые подтверждаются доказательствами и сопровождаются рекомендациями.

Вот что говорится в нем по поводу данных, полученных предыдущими исследователями: «Очень многие работы в избытке содержат ценные выводы и подробные рекомендации. Большинство из них так и не становятся реальностью. И если проблема военного ПО на самом деле существует, то она не воспринимается как срочная».

Некоторые выводы

Но какова значимость этого отчета для обычного программиста, который не имеет отношения к обработке военных данных? Посмотрим на эти выводы:

- Обсуждая системный анализ, авторы называют постановку задачи и формулирование требований «самой трудной частью» разработки ПО. Именно здесь они видят насущную и фундаментальную проблему и закладывают основу аргументов в пользу прототипирования как способа ее решения:

Самая трудная часть задачи создания ПО состоит в том, чтобы сформулировать точные требования. В общепринятой практике неизвестны хорошие приемлемые способы даже для того, чтобы сформулировать подробные требования и приоритеты компромиссов... Сплошь и рядом принимаются неправильные решения. Чаще всего это выражается в том, что на функции возлагается слишком много, а это отрицательно сказывается на размере и быстродействии кода и проявляется слишком поздно в жизненном цикле разработки. Другая распространенная ошибка заключается в неправильном представлении о том, как надлежит работать пользовательскому интерфейсу. По нашему мнению, это коренная проблема. Мы уверены, что пользователи не в состоянии точно описать рабочие требования к надежной программной системе, если ее тестирование не проводится реальными операторами в рабочем окружении и спецификация не уточняется методом последовательных приближений. Системы, которые создаются в наше время, слишком сложны, чтобы человек мог предвидеть все варианты в уме.

- В отчете констатируется полное непонимание роли языков четвертого поколения (4GL) и отмечается, что на практике методики 4GL и 3GL применяются вперемешку:

Термин «четвертое поколение» употребляется некорректно. Им обозначали самые разные языки, которые не происходили от языков третьего поколения, языков общего назначения. Этот термин относится к специализированным языкам, ориентированным на применение в конкретных прикладных областях, например к языкам баз данных и электронным таблицам, системам автоматического программирования (генераторам программ), не процедурным языкам и даже к языкам искусственного интеллекта, таким как Prolog. Каждый язык создавался в расчете на применение к задачам четко очерченной области. То есть языки чет-

вертого поколения не конкурируют с Адой (и другими языками общего назначения).

- Авторы отчета считают, что ожидания относительно Ады не оправдались. Может быть, Ада и хороший язык, и в отчете рекомендуется «развивать его всерьез и целеустремленно», однако там же указывается, что проблемы ПО намного превосходят способность языка решить их:

Ада слишком много обещала... Необходимо создавать инструментальные средства для следующих областей деятельности:

- Написание и форматирование программной документации.
- Управление версиями и конфигурациями и ПО, и документации.
- Сопровождение истории разработки, помогающее увязать требования с техническими условиями проекта, документацией, исходным кодом, компилированным кодом, программными отчетами, изменениями кода, тестами и результатами прогонов тестов.
- Отладка.
- Управление графиком выполнения проекта.

И поэтому результаты, которых ожидают менеджеры программных проектов, не имеющие солидной технической базы, не может обеспечить никакой язык программирования.

- Говоря о внедрении методом последовательных приближений, авторы отчета употребляют слова «профессиональная скромность» (professional humility), характеризуя с их помощью способность разработчиков ПО осознать пределы своих возможностей. Имеется в виду, что перед тем как начать полномасштабную разработку, выпускается ранний релиз минимально завершенных версий программного продукта:

Оказалось, что спокойное уточнение спецификаций и медленное построение [инкрементная разработка] программных гигантов представляет собою самый простой, безопасный и даже быстрый способ создания программной системы. Надо построить работающую систему, обладающую минимальной функциональностью, и наращивать ее, повышая быстродействие, уменьшая размер и т. д., соотносясь с приоритетами, которые будет диктовать реальность... Эволюционная разработка... разрушает привычные формы поставок ПО на конкурсной основе (впрочем, они делают с эволюционной

разработкой то же самое). Теперь необходимо привнести творческое начало в процесс разработки.

- Относительно готовых программных решений в отчете сказано, что лучший способ создать ПО – это совсем не браться за это дело:

Дешевле всего обходится ПО, покупаемое на рынке, а не создаваемое своими силами. Быстрее всего купить ПО на рынке, а не создавать его своими силами. Надежнее всего купить ПО на рынке, а не создавать его своими силами.

- Авторы затрагивают и программные метрики, отмечая сильнейшую потребность в измерении качества программного продукта и предлагая вполне осуществимое решение:

Не существует метрик качества программного кода, качества объектного кода, качества документации и т. д... Есть методы, позволяющие оценить общее качество по сложным характеристикам, выходящим за рамки индустрии ПО... О качестве современного ПО могли бы судить специальные судебские коллегии, аналогичные тем, которые судят на Олимпийских играх соревнования по прыжкам в воду, конькам и акробатике.

Некоторые рекомендации

Всего в докладе содержится 38 рекомендаций. Более половины из них относятся в основном к военному ПО и касаются организационных и политических изменений.

Остальные посвящены нескольким важным темам, представляющим интерес для более широкого круга разработчиков ПО обработки данных. Рекомендации касались областей, которые авторы отчета считали важными:

- Инкрементная разработка и прототипирование
- Повторное использование [кода]
- Язык Ада
- Языки четвертого поколения
- Управление рисками
- Метрики

Для каждой из этих тем дано несколько рекомендаций.

Инкрементная разработка и прототипирование. Сильнее всего в отчете защищается применение прототипирования в инкрементной разработке программных систем:

Рекомендация 12: «Придерживайтесь эволюционного подхода, действуйте симуляцию и прототипирование... чтобы снизить риск».

Рекомендация 23: «...дайте зеленый свет итеративному уточнению спецификаций, быстрому прототипированию систем, параметры которых определены, и инкрементной разработке».

Рекомендация 24: «...избавьтесь от любой... зависимости от допущений “каскадной” модели и... возведите быстрое прототипирование и инкрементную разработку в статус официальной нормы».

Рекомендация 26: «...обеспечьте... возможность совместного с пользователями быстрого прототипирования в... процессе разработки продукта».

Повторное использование. Почти так же активно в отчете поддерживается повторное использование уже созданных программных компонентов и систем:

Рекомендация 29: «...экономически стимулируйте... поставщиков и подрядчиков, предоставляя им возможность получать прибыль от поставки повторно используемых модулей, даже если те созданы на средства Министерства обороны».

Рекомендация 15: «...убеждайте менеджеров программных проектов, что программные системы с определенными характеристиками могут быть построены из готовых типовых подсистем и компонентов, кроме тех случаев, когда они уникальны».

Рекомендация 16: «Все методологические усилия... необходимо направлять на поиски способов отбора и стандартизации доступных на рынке программных инструментов и адаптации последних к потребностям Министерства обороны».

Рекомендация 30: «...экономически стимулируйте... поставщиков и подрядчиков, побуждая их покупать готовые программные модули, а не создавать новые».

Рекомендация 31: «...помогайте менеджерам распознавать в управляемых ими программных проектах такие подсистемы и, может быть, даже модули, покупка которых более предпочтительна, чем создание, и поощряйте такие приобретения...»

Ада. Отчет, что и неудивительно, твердо отстаивает применение этого языка как стандартного для военных приложений. Но формулирует некоторые предостережения:

Рекомендация 8: «...всячески препятствуйте дроблению Ады на подмножества».

Рекомендация 9: «...вкладывайте больше средств в обучение менеджеров и технического персонала практическому применению Ады».

Рекомендация 32: «...формируйте рынок модулей-прототипов, изначально ориентированных на модули, написанные на Аде, и на инструментальные средства для этого языка...»

Рекомендация 33: «...установите стандарты описания для модулей на Аде...»

Языки четвертого поколения. Поддержка языков четвертого поколения в этом отчете довольно сдержанная. Обратите внимание, как в нем подчеркивается «стоимость жизненного цикла», – знак, недвусмысленно указывающий на необходимость тщательно изучить как преимущества, связанные с производительностью и сопровождением кода, так и недостатки, связанные с эффективностью этих языков:

Рекомендация 10: «Языки 4GL имеет смысл применять там, где выгоды, связанные с полной стоимостью жизненного цикла, более чем в десять раз превышают эффект от применения языков общего назначения».

Управление рисками. В отчете подчеркивается важность повсеместного внедрения нового подхода к управлению. И новый менеджмент должен сосредоточить внимание на рисках, связанных с разработкой ПО:

Рекомендация 25: «...внедряйте методы управления рисками в приобретение ПО...»

В отчете есть пример плана управления рисками:

1. Определите 10 главных факторов риска в проекте.
2. Составьте план устранения каждого из этих факторов.
3. Список главных факторов риска, план и результаты его выполнения надо обновлять ежемесячно.
4. В ежемесячных обзорах проекта необходимо подчеркивать статус факторов риска.
5. Предпринимайте адекватные корректирующие действия.

Метрики. Необходимость иметь возможность количественно оценить и сам программный продукт, и процесс его создания, особенно качество ПО, подчеркивается особо:

Рекомендация 18: «...продумайте, как материально стимулировать рост качества ПО».

Рекомендация 19: «...разработайте метрики и способы измерения качества и степени завершенности ПО...»

Рекомендация 20: «...разработайте метрики, которые позволят количественно оценить процесс внедрения».

Заключение

Из рассеявшегося тумана рекомендаций отчетливо выступает главная мысль всего отчета: «Мы не призываем к новым технологическим начинаниям, предусматривающим небольшое смещение фокуса внимания по ходу дела, мы считаем, что необходимо коренным образом пересмотреть и изменить отношение и образ действия во всем, что касается приобретения ПО».

Другими словами, технологические прорывы не способны сыграть роль спасителя ПО (как, если уж на то пошло, и ничто другое). Но изменить подходы к управлению необходимо.

Может быть, те, кто создает ПО обработки данных, *хотели* услышать что-то другое, но нам *нужны* именно эти слова.

В эту комиссию входили такие известные в сообществе разработчиков ПО и вычислительной техники люди, как Вик Бэсили (Vic Basili) из Мэрилендского университета, Барри Боэм (Barry Boehm) из компании TRW, Элайна Бонд (Elaine Bond) из Chase Manhattan, Нейл Истмэн (Neil Eastman) из IBM, Дон Эванс (Don Evans) из Tartan Laboratories, Анита Джонс (Anita Jones) из Tartan Laboratories, Мэри Шой (Mary Shaw) из университета Карнеги-Меллона (Carnegie Mellon University), Чарльз Зракет (Charles Zraket) из MITRE, а также несколько представителей правительства и подрядчиков.

Ретроспектива

Когда я перечитывал очерки этой главы из первого издания, больше всего меня удивило, как сильно я обязан своим многочисленным известным коллегам тем, что они помогли мне понять все это.

Вот, к примеру, размышления самого, вероятно, авторитетного ученого в области программной инженерии, Дэвида Парнаса. В 1980-х годах Парнас находился в гуще полемики, подогреваемой нешуточными политическими страстями и посвященной теме, связанной с высокими технологиями, – так называемой программе «звездных войн». В основе этой программы лежала мысль о том, что можно создать противоракетный щит, способный нейтрализовать любую неприятельскую атаку, направленную против Соединенных Штатов (будь то обычные, ядерные или любые другие ракеты).

Полемика разгорелась вокруг вопросов «Можем ли мы?» и «Следует ли нам?». Парнас решительно выступил на стороне тех, кто отрицательно ответил на первый вопрос. Однако его аргументы, по-моему, намного больше относились к жизнеспособности новых подходов программной инженерии, чем к жизнеспособности программы «звездных войн».

Именно в таком духе был выдержан этот очерк, когда был напечатан впервые (как и сейчас, когда публикуется повторно). Поскольку современные политики реанимировали старые аргументы в пользу программы «звездных войн», сегодня вопрос «Можем ли мы?» так же актуален, как и тогда.

Не представляю себе, на кого еще я мог бы опереться в этом разделе, кроме Парнаса? Разве что на самого Брукса. Он-то мне и поможет в третьем очерке этого раздела. Если Дэвида Парнаса можно назвать лучшим ученым в области программной инженерии, то Брукс, вероятно, ведущий практик программной инженерии в академической сфере вычислительной техники.

«Мифический человеко-месяц»¹, – бесспорно, самая значимая, вечная книга «усвоенных уроков» о программной инженерии, а очерк «No Silver Bullet» (Серебряной пули нет), к которому я апеллирую в этой книге, – наверное, самая проникательная работа о программной инженерии (и тоже вечная).

В этом очерке Брукс сказал, что прорывы в сфере программной инженерии достигаются чрезвычайно тяжело и следует ожидать, что в ближайшие годы их будет очень немного. В некотором смысле и Брукс, и Парнас в своих очерках говорят очень похожие вещи, а именно, что на вопрос о возможности реализации программы «звездных войн», скорее всего, надо ответить «нет». А это, в свою очередь, очень многое говорит о громогласных заявлениях, потрясавших программную инженерию с тех пор, как первый восторженный поставщик и первый неумеренно оптимистичный ученый вообразили, будто совершили некий переворот в способах создания ПО.

Главная идея этого раздела, по мнению моих блестящих коллег, состоит в том, что такой прогресс вряд ли произойдет в нашей области. И те, кто заявляет о каких-то революционных достижениях, оказывают нам плохую услугу.

Самым очевидным примером устаревшей технологии в этом разделе можно считать язык Ада. Министерство обороны США отводило ему роль главного инструмента, который должен был устранить языко-

¹ Фредерик Брукс «Мифический человеко-месяц, или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

вой барьер «Вавилонской башни» их программного обеспечения. Однако за то время, которое разделяет «Software Conflict» и «Software Conflict 2.0», этот язык зачах, а потом и вовсе почти исчез.

Почему умер язык Ада?

Потому что для Министерства обороны выбор нового языка означал полный отказ от всех старых, а это породило организационный хаос.

Потому что в Министерстве обороны не нашлось такой силы, которая сделала бы Аду стандартом де-факто. (Правда, лет за десять-двадцать до этого такая сила нашлась и помогла Коболу, который без нее погиб бы той же смертью, что и Ада. На самом деле Министерство обороны не давало распоряжения использовать Кобол. Оно потребовало, чтобы производители обеспечили каждый компьютер, находившийся на балансе министерства, компилятором Кобола, и эффект получился тот же самый.)

Потому что шли споры о том, насколько хорошо этот язык приспособлен для решения задач в своей прикладной области, а именно в области систем реального времени.

И потому, наконец, что Министерство обороны попыталось спасти Аду, превратив эту изящную стрелу, способную точно поразить одну цель (системы реального времени), в бесформенную мягкую субстанцию, которая, хоть и заливает собою все вокруг, не поражает ничего. (В этой книге мы еще вернемся к теме технологий, ориентированных на конкретную область.)

Глава 2 | Из окопов технологии

- Когнитивный взгляд: проектирование ПО с другой точки зрения
- Некоторые размышления об ошибках в ПО
- Экспериментальный аспект устранения программных ошибок
- Многоликое тестирование
- Качество ПО и сопровождение ПО. Какая связь?
- Сопровождение ПО – это решение, а не проблема
- Управление из одной точки
- «Дружелюбный к пользователю» – модное выражение или прорыв?
- Ретроспектива

Когнитивный взгляд: проектирование ПО с другой точки зрения

Здесь я хочу обсудить вопрос «Что такое проектирование ПО?».

Он может показаться вам странным. В конце концов, люди занимаются этим уже больше 30 лет. И разве мы не располагаем всеми методологиями и языками проектирования, необходимость которых диктуется здравым смыслом? Так почему же до сих пор спрашивают: «Что такое проектирование ПО?»

Все это время, говоря о проектировании, мы ходили вокруг да около. Методологии – это не проектирование; они представляют собой платформу, на которой мы объединяем и организуем наши проектные усилия. И языки – это не проектирование; это способ записи готового, по сути, проекта. Проектирование есть нечто такое, что происходит внутри головы, в мозгу, причем происходит быстрее молнии. Проектирование, о котором я собираюсь говорить, концептуально *именно таково*.

За прошедшие годы я не понаслышке узнал, что такое проектирование ПО, и у меня богатый опыт преподавания этого предмета, но я должен сделать признание. У меня никогда не было ощущения, что я в самом деле понимаю, что такое проектирование, и что мой подход к проектированию правилен.

И в последние годы я обнаружил, что не одинок. Когда я читал курс программной инженерии в аспирантуре университета Сиэттла, некоторые из нас избегали темы проектирования, потому что мы чувствовали, что не знаем, что это такое на самом деле, и, следовательно, не знаем, как этому учить. А за тот год, что я провел в Институте программной инженерии в университете Карнеги-Меллона, мне встречались такие люди, которые чувствовали не только, что они сами не знают, как учить проектированию, но и что этого не знает вообще никто! В этот момент я уже не чувствовал себя так одиноко, но что такое проектирование, не понимал по-прежнему.

Данные эмпирических исследований

Все это начинает меняться. Вопросом о том, что же такое проектирование, занялись другие компьютерщики, которые тоже чувствовали, что оно является материей трудноосознаемой. И у них уже начали появляться ответы.

Может быть, для того чтобы лучше понять то, что я скажу дальше, вам потребуется немного отойти от некоторых представлений, успевших стать традиционными за короткую историю ПО. Перестаньте считать

внешние представления альфой и омегой проектирования и обратите главное внимание на мыслительный процесс. Секрет проектирования лежит внутри нашего разума.

Я до сих пор помню эпизод, благодаря которому это стало мне понятно. Дело было на занятиях по языку Ада в Военно-воздушной академии. Шел последний день недельного курса, руководители разбили нас на группы и выдали задания. Едва они закончили формулировать требования, как один из членов моей группы, молодой и очень толковый человек из компании «Боинг», сказал, что у него готов проект. Я сию и пытаюсь понять постановку задачи, а он уже все придумал! Тогда-то я и понял, что проектное решение с быстротой молнии было синтезировано в мозгу и что, между прочим, у некоторых людей эти «молнии» обладают намного большей скоростью, чем у других!

Но что же в действительности произошло в голове у этого блестящего молодого человека? Сейчас начинают выясняться ответы на этот вопрос, по крайней мере на главный, который за ним кроется.

Но прежде чем я начну говорить о том, что же это за ответы, разрешите мне немного рассказать об их происхождении. Исследования, посвященные ПО, переживают новый подъем, благодаря которому мы и получаем эти ответы. Некоторые исследователи, например Билл Кертис (Bill Curtis) из Корпорации микроэлектроники и компьютерных технологий (Microelectronics and Computing Consortium, MCC) и Эллиот Соловей (Elliot Soloway) из Мичиганского университета (бывшего Йельского), много времени посвятили доказательству тезиса о том, что изучение программного обеспечения по крайней мере в такой же степени означает изучение программистов, в какой оно означает изучение программ. Эту часть своей работы они называли «эмпирическими исследованиями программистов», и самый горячий интерес в этой области исследований вызывает проектирование ПО.

Столкнувшись с тем же вопросом, который я вынес в начало этой статьи («What is software design?»¹), эти ученые предложили свой план действий. Им было необходимо получить «мгновенный снимок» мыслительного процесса разработчиков в ходе проектирования, не вмешиваясь в этот процесс и, следовательно, не нарушая его. Однако это проще сказать, чем сделать.

Все-таки они смогли это сделать, обратившись к методам области знаний, известной под названием «анализ протоколов». Они сидели рядом

¹ Jack W. Reeves «What is software design?». *C++ Journal*, 1992 (Джек Ривз, «Как проектировать ПО?». «Компьютерра» № 17, май 2005).

с проектировщиками, попросив их думать вслух и записывая услышанное. Они записывали эти сессии на магнитофон и делали видеозаписи. Все записав, они погрузились в изучение результатов. И принялись описывать действия дизайнеров с точки зрения теории.

Первая группа обнаруженных ими фактов не так уж сильно помогла им. Оказалось, что проектирование включает:

- Осмысление задачи
- Декомпозицию задачи на цели и объекты
- Выбор и составление планов решения задачи
- Реализацию планов
- Раздумья над продуктом и процессом

Толку от всего этого было немного. Если слегка изменить слова, то окажется, что вышеприведенная последовательность не сильно отличается от жизненного цикла ПО, который известен нам и с которым мы боремся уже много лет.

И только углубившись в «выбор и составление планов» (см. вышеприведенный список), они наконец докопались до сути. Именно за этими общими словами скрывался простой набор действий, составляющий существо проектирования.

Квинтэссенция проектирования

Что это были за действия?

Мысленно, со скоростью молнии, проектировщики делали следующее:

1. Они строили умозрительную модель предлагаемого решения задачи.
2. Проверяли в уме работу модели – по сути, занимались симуляцией на модели, чтобы посмотреть, позволяет ли она решить задачу.
3. Обнаружив, что не позволяет (обычно из-за того, что она была слишком простой), они проверяли работу недостаточной модели при тех условиях задачи, с которыми она не могла справиться, находили, где именно она терпит неудачу, и корректировали модель для этих условий.
4. Повторяли шаги 1–3 до тех пор, пока не получали модель, которая, как им казалось, решала задачу.

А теперь, поскольку эти четыре шага являются ключом к ответу на наш исходный вопрос, задержимся на них еще немного, прежде чем двигаться дальше. Процесс, который мы здесь наблюдаем, протекает в мозгу итеративно и с большой скоростью; по сути, он представляет со-

бой быструю реализацию метода проб и ошибок. Мозг формирует решение задачи, зная, что оно будет недостаточным, поскольку он еще не в состоянии охватить все грани задачи. Разум осознает, что решение задачи должно быть выражено в форме модели, потому что придется, скорее всего, ввести в модель тестовые данные, выполнить (в уме) тестовый прогон модели с этими данными и получить выходные данные (опять в уме).

Суть проектирования, таким образом, состоит в быстром моделировании и симуляции выполнения (тестовом прогоне в уме). А ключевым фактором проектирования является способность предлагать решения, допуская при этом, что они окажутся неверными!

Интересно отметить, между прочим, что эти же самые ученые изучали трудности, которые испытывают те люди, которым не очень хорошо удается решать задачи проектирования. Как правило, они создают представления проектов, а не модели, и поэтому не могут выполнить тестовый прогон, а в результате у них получаются проектные решения, не соответствующие требованиям. С этими выводами связана одна из моих любимых мыслей: неудача – это неотъемлемая и важнейшая часть успешного проекта ПО! Каждая предыдущая итерация завершается созданием модели, которая не выдерживает испытаний, не позволяет решить поставленную задачу; а это, в свою очередь, наводит на мысль, что неотъемлемой составляющей успеха является способность терпеть неудачу и восстанавливаться после нее. Из этой мысли, на мой взгляд, можно сделать выводы, интересные с точки зрения обучения проектированию, да и любому другому предмету. Где-нибудь читают курс «Что такое провал и как его пережить»?!

Некоторые из этих же самых мыслей были сформулированы несколькими выдающимися разработчиками ПО. В книге «Programmers at Work» (Программисты за работой), выпущенной издательством Microsoft Press и составленной из интервью, взятых Сьюзен Ламмерз (Susan Lammers), они так высказывались о своих взглядах на процесс проектирования:

«Приступая к программированию, надо прежде всего представить себе решение в своем воображении. Просто надо добиться кристально ясного представления о том, что будет происходить дальше. На этом исходном этапе я прибегаю к помощи карандаша и бумаги. Но рисую машинально, что попало... потому что реальная картина находится у меня в голове». (Чарльз Шимоньи (Charles Simonyi), создатель Multiplan)

«В какой-то момент проект начинает напоминать заряд взрывчатки, и в это время он целиком находится в моем мозгу... И там происходят самые разные процессы, которые я не могу зафиксировать на бумаге, потому что все время что-то меняю в них». (Гэри Килдалл (Gary Kildall), создатель CP/M)

«Необходимо в голове воспроизвести работу программы... Когда вы что-то создаете... и ваша голова занята моделью, это занятие очень индивидуальное». (Билл Гейтс, глава компании Microsoft)

Интересно, что выводы исследователей во многом совпадают с этими неофициальными высказываниями хорошо известных разработчиков.

Некоторые другие выводы

Есть еще один очень важный вывод, относящийся к нашей теме. Другие исследователи, например Виллемин Виссер (Willemien Visser) из французского Национального исследовательского института информатики и автоматизации (Institut National de Recherche en Informatique et en Automatique), обнаружили, что «разработчики редко начинают работу с нуля». В качестве основы, позволяющей им начать процесс симуляции, они берут готовую модель, созданную в ходе решения предыдущей аналогичной задачи. Оглядываясь назад, я думаю, что на самом деле молодой человек из «Боинга» проделывал над Адой именно *такую* умственную работу. Ему уже приходилось решать подобные задачи, и предварительное решение уже таилось в его мозгу! (Не правда ли, удивительно, насколько мы преуспели в умении оправдывать свои личные недостатки?!)

Однако все это относится к отдельным разработчикам, занимающимся проектированием. А мы знаем, что в 1980-е проектирование стало командным процессом. Задачи стали слишком объемными для отдельных разработчиков, а кроме того, они начали возникать на стыках нескольких дисциплин, что потребовало от разработчиков разнообразных умений и навыков.

Те же самые исследователи интересовались и командным проектированием. И они обнаружили, что во многом командное проектирование представляет собой совместную форму индивидуального проектирования:

- Команды создают совместно используемую умозрительную модель.
- Члены команды выполняют тестовые прогоны на этой модели – иногда каждый у себя в уме, а иногда все вместе.
- Команды оценивают результаты тестовых прогонов и подготавливают следующий уровень модели.

Однако в некоторых отношениях их действия отличаются от действий отдельных разработчиков:

- Конфликт – это неотъемлемая часть процесса проектирования. Их необходимо разрешать, а не избегать.

- Жизненно важной частью процесса проектирования становятся приемы коммуникации.
- Некоторые вопросы остаются без внимания, потому что никто не берет их решение на себя.

Эти команды разработчиков обычно насчитывают от 3 до 6 человек. Однако иногда, если задача необыкновенно сложна, недостаточно и этого количества. И в этом случае проектирование становится организационной задачей. Как правило, в ходе коллективного проектирования возникает иерархия групп разработчиков, каждая из которых решает свою часть задачи, и дополнительно создается специальная группа главных архитекторов, работа которых состоит в том, чтобы объединять и координировать усилия всех разработчиков.

Однако командному проектированию и организационному подходу присущи свои проблемы. Проектирование легко может принять форму заседаний и совещаний со всеми их недостатками (помните: «Верблюд – это скаковая лошадь, спроектированная коллективно»?). Фредерик Брукс, написавший «Мифический человеко-месяц», а до этого статью «Серебряной пули нет», указал, что лучшие продукты, которые, по нашему общему мнению, обладают концептуальной целостностью (например, Pascal и UNIX), были спроектированы одиночками. Примеры удачных командных проектов известны – это Ада и Кобол, а также операционные системы для мэйнфреймов IBM, но на них в целом поглядывают, как замечает Брукс, снисходительно, считая их хотя и успешными, но нескладными.

Что же дальше?

Итак, мы достигли понимания по некоторым вопросам. Имеется некий четко определенный мыслительный процесс, происходящий во время проектирования. Он начинается с готовой или упрощенной новой модели, на которой затем выполняется в уме тестовый прогон (симуляция), и все это продолжается итеративно до тех пор, пока проектное решение не становится пригодным для поставленной задачи. Если задача достаточно велика либо сложна, то к проекту подключаются команды или даже организации. Они применяют многие из тех же методов, что и отдельные разработчики, но задействуют и групповой процесс, без которого, при всей его громоздкости, иногда не обойтись.

Что же нам делать с этим новым пониманием?

Мне представляется, что его можно применить в трех сферах: оно влияет на обучение проектированию, на собственно проектирование и на управление проектированием.

Обучая проектированию, уже недостаточно рассказать об одной, двух или трех методологиях и концепциях. Эти традиционные темы необходимо рассматривать в комплексе с новыми представлениями о проектировании как об умственном процессе.

Разработчику, который занимается проектированием, полезно понимать, что суть этого процесса не в том, что кажется таковым ему, и что тернистый путь проб и ошибок, которым, возможно, он следует, и есть правильный. И это может укрепить его в решимости следовать правильным путем проектирования без колебаний. (Хватит ли у нас смелости назвать этот подход, например, «беспечным»? Пожалуй, это было бы не слишком точно!)

Руководители проектных групп могут внести свою лепту, сосредоточив усилия на улучшении организационных связей и разрешении конфликтов. Ученые, занимающиеся эмпирическими исследованиями, говорят, что управление проектированием должно заключаться в разрешении главных вопросов, возникающих в процессе проектирования.

Стремясь к достижению этих целей, а именно к тому, чтобы усовершенствовать обучение, практические методы и управление в сфере проектирования ПО, эти ученые предложили несколько инструментальных подходов. Пока что мы умеем создавать не все эти инструменты, но если бы и умели, то до реального улучшения процесса проектирования было бы еще далеко.

1. Пакеты для моделирования и симуляции, способствующие умозрительному проектированию.
2. Пакеты, которые позволяют сохранять и извлекать идеи, препятствуют их утрате.
3. Корректировщики стратегических предположений (strategic assumption surfacers), которые следят за соблюдением главных требований и напоминают о них, когда проверяемый вариант решения грозит нарушить одно из них.
4. Поддержка тематического (issue-based) разрешения конфликтов.
5. Запись/отслеживание неразрешенных проблем.
6. Поддержка модулируемых обсуждений.
7. Сбор и координация идей в группах.

Однако, может быть, еще рано говорить об инструментальных средствах поддержки этого процесса. Мы только начинаем понимать, в чем же все-таки суть проектирования ПО (после того как мы больше 30 лет думали, что знаем это). Наверное, одного этого должно быть достаточ-

но, чтобы на некоторое время занять наше внимание. Решение вопроса о том, как распорядиться этим знанием, можно отложить на более позднее время.

Может быть, нам следует придумать какое-то вспомогательное средство, которое поможет нам с ним управиться. Упомянутые исследователи называют все это когнитивными процессами в проектировании. Мне это нравится. Они утверждают, что проектирование – это умственный процесс, и я считаю, что именно в таком ключе и надо о нем говорить.

Таким образом, в этой главе мы коснулись когнитивных аспектов проектирования ПО.

Некоторые размышления об ошибках в ПО

Эти мысли – результат моего более чем 35-летнего опыта в производственной и академической сферах программной инженерии. В этих размышлениях я пытаюсь суммировать свое понимание процессов и результатов этой отрасли, глядя на них сквозь ошибки в программном обеспечении, как сквозь увеличительное стекло, собирающее и фокусирующее взгляд.

Тезис 1. *Известные методики устранения ошибок не гарантируют стопроцентного обнаружения ошибок в ПО.*

Этот тезис имеет отношение к процессу. Известные процессы удаления ошибок в программах можно разбить на (1) анализ, или экспертизу (review); (2) тестирование (test) и (3) доказательство корректности (proof). О значимости этих процессов нам известно многое. Доказано, что из всех трех наименее затратным на всех этапах, от исходных требований до программного кода, является анализ. Доказано, что тестирование на всех уровнях, от программных модулей до систем, представляет собой абсолютно необходимое дополнение к анализу. Доказательству еще только предстоит показать на практике свою действенность как методики устранения ошибок, однако оно являет собой третью альтернативу двум другим методикам, причем отличается от них, как день от ночи.

Однако по-настоящему важная часть этого тезиса состоит в том, что ни один из этих процессов не является достаточным. Экстраполируя эту мысль немного дальше, можно с полным основанием сказать, что...

В настоящее время проблему ошибок в программном обеспечении нельзя разрешить при помощи процессов.

Тезис 2. Невозможно отыскать все ошибки в ПО.

Этот тезис имеет отношение к продукту. Многие уверены, что программные продукты – это самое сложное, что удалось создать человеку. Конечно, когда мы смотрим на логические блоки программы и пытаемся мысленно пройти лабиринтами путей выполнения этих блоков, мы сталкиваемся с комбинаторным взрывом и почти бесконечным разнообразием этих путей. Не приходится удивляться тому, как трудно получить ПО без ошибок.

Мысль о невозможности устранения всех ошибок имеет большое значение. Однако мы не можем, ссылаясь на нее, не прикладывать максимума усилий к тому, чтобы избавиться от всех ошибок, потому что важность этой мысли в другом. А именно в том, что...

ПО, предназначенное для работы в критически важных системах, нуждается в дополнительных мерах защиты от программных ошибок, помимо устранения последних.

На самом деле именно эта причина обусловила появление важной и развивающейся отрасли – отказоустойчивого ПО. Отказоустойчивость – это аварийное решение, которое задействуется там, где не удастся избежать ошибок.

Следует признать, что эта мысль полемична. Арлан Миллз (Harlan Mills) и другие утверждают, что способны создать ПО без ошибок, привлекая к работе программистов, обладающих должными психологическими характеристиками, и применяя определенные процедуры в строгом соответствии с установленным порядком. Может быть, им, в отличие от других, делавших ранее аналогичные заявления, удастся выполнить то, что они обещают. Однако можно с уверенностью сказать, что в 1988 году состояние как практики, так и теории требует считаться с вероятностью скрытых ошибок в критически важном ПО.

Тезис 3. Не все тестировщики ПО равны.

Этот тезис относится к людям, которые создают продукт, задействуя определенный процесс. Мы знаем массу примеров колоссальных различий между разработчиками ПО (превосходство лучших может быть 30-кратным). По данным Гленфорда Майерса (Glenford Myers), некоторые профессиональные разработчики обнаруживают в семь раз больше ошибок в ПО, чем остальные. Исследования Нэнси Ливсон (Nancy Leveson) показали, что лишь 9 из 24 участников обнаружили хоть какие-то ошибки в программе, заведомо содержавшей 60 ошибок. График на обложке книги «Software Engineering Economics» (Экономика программной инженерии), написанной Барри Боэмом (Barry Boehm), весьма убедительно показывает, что квалификация разработчиков намного сильнее влияет на производительность в индустрии ПО, чем любой другой фактор.

Вывод, который надлежит сделать из этого тезиса, таков...

Самый важный элемент системы устранения ошибок в ПО – это не характер продукта или процесса, а подбор специалистов, то есть человеческий фактор.

Увы, но данный вывод не учит нас, как распознавать таких людей. Поэтому мы до сих пор пытаемся сделать лучший выбор при помощи субъективных и произвольных методов.

Тезис 4. *Не все ошибки в ПО одинаково значимы.*

Этот тезис о том, как собрать воедино предыдущие тезисы. В первую очередь надо сказать, что не все программные ошибки плохи. Бывает и так, что выгода от устранения ошибки оказывается меньше, чем затраты на устранение. Более того, такие ученые, как Эллиот Соловей (Elliot Soloway), заинтересовавшись, как работают хорошие разработчики ПО, обнаружили, что они чаще начинают проект с неудачи, чем «плохие» разработчики. Другими словами, по крайней мере на ранних стадиях жизненного цикла путь к качественному ПО должен проходить через «ошибки», поскольку идеи рассматриваются, испытываются и отвергаются. Таким образом, мы ясно видим, что некоторые ошибки не так плохи, как другие.

Однако важнее то, что некоторые программные ошибки намного хуже других. К их числу относятся ошибки, которые (1) переживают все процедуры устранения ошибок и сохраняются в продукте до этапа эксплуатации, а позже (2) вызывают небезопасные или дорогостоящие последствия для системы, частью которой является данный продукт. В 1981 году я выполнил исследование, которое показало, что эти живучие ошибки чаще гнездятся там, где сложность программы меньше сложности задачи, которую программа решает. К примеру, в программе проверяются два условия в операторе IF, а третье условие игнорируется; или в одной из ветвей исполнения не происходит сброс значения переменной в начальное после того, как переменная отслужила свое в этой ветви и ей необходимо было вернуть значение, ожидаемое следующим блоком программы. Нэнси Ливсон связала программные ошибки и безопасность систем и показала, кроме того, что независимые разработчики ПО обнаруживают тенденцию совершать общие ошибки. Тим Грамз (Timm Gramms) называет их систематическими (biased errors) и говорит, что они обусловлены «ловушками мышления».

Значение этого тезиса состоит в том, что...

И устранение ошибок, и обеспечение отказоустойчивости должны быть сосредоточены на худшей разновидности ошибок, а именно на тех, которые могут сделать систему ненадежной.

Итоги размышлений

Увеличительное стекло программных ошибок позволяет нам увидеть и сфокусироваться на нескольких важных моментах, касающихся самого программного обеспечения:

1. В настоящее время проблему ошибок в ПО нельзя разрешить при помощи процессов. Дело не в том, что нам не хватает процессов. У нас есть:
 - a. Анализ, или экспертиза (review), тестирование (test) и доказательство корректности (proof).
 - b. Структурное тестирование и тестирование исходных требований.
 - c. Статистическое тестирование и тестирование безопасности продукта.
 - d. Модульное, интеграционное и системное тестирование.
 - e. Изобилие инструментальных средств и методик.
 - f. Просто-напросто оказалось, что проблема перекрывает все известные нам процессы.
2. ПО, предназначенное для работы в критически важных системах, нуждается в дополнительных мерах защиты от программных ошибок, помимо устранения последних. Несмотря на все наши усилия и желания, с программным продуктом необходимо обращаться так, будто он все еще содержит ошибки.
3. Самый важный элемент устранения программных ошибок (и, если уж на то пошло, создания ПО вообще) – это подбор исполнителей. Хорошие программисты обычно находят хорошие решения. Решения «плохих» программистов обычно НАМНОГО хуже.
4. И при устранении ошибок, и при обеспечении отказоустойчивости необходимо концентрировать усилия на самых тяжелых ошибках. Некоторые ошибки легко совершать, другие – трудно находить, а какие-то приводят к созданию ненадежных систем. Более половины из тех ошибок, которые легко совершить и трудно обнаружить, делают систему ненадежной.

Так что дела обстоят совсем не так хорошо, как нам хотелось бы. В устранении ошибок и обеспечении отказоустойчивости необходимо задействовать:

- Тщательно отобранных людей
- Разнообразные процессы, эффективность которых известна
- Понимание природы ошибок и надежности

Любой набор средств, в котором нет чего-либо из перечисленного, попросту недостаточно хорош.

Экспериментальный аспект устранения программных ошибок

Какой способ устранения ошибок ПО самый эффективный?

Какой способ устранения ошибок ПО наименее *затратный*?

Это масштабные вопросы. Анализ структуры расходов на разработку ПО показывает, что устранение ошибок – в целом самая дорогостоящая часть жизненного цикла ПО: она примерно в два раза дороже и анализа систем, и проектирования, и написания кода.

И, поскольку эти вопросы такие важные, остановимся подробнее на том, какие же ответы нам известны. Я хотел бы с уверенностью сказать, что нам известны факт А, факт В и факт С, и поэтому самое выгодное вложение средств, отведенных на устранение ошибок, состоит в том, чтобы применить стратегию Z. Однако все не так просто.

Почему не так просто? Прежде всего, по-настоящему полезных данных о действенности различных методик разработки ПО намного меньше, чем субъективных мнений. Может быть, еще хуже, чем преобладание субъективных мнений над фактами, то, что индустрия разработки ПО наводнена лоббистами.

Субъективные мнения и лоббирование не только не дают нам количественной информации, опираясь на которую можно было бы отвечать на масштабные вопросы, аналогичные приведенным выше, но и накаляют обстановку в целом. Лоббисты, как правило, так агрессивно проталкивают свою технологию, что люди либо переходят на их сторону, либо, наоборот, бегут от пропаганды. Лоббизм и субъективные мнения сильно способствуют потере объективности.

Преодолеть их действие можно при помощи объективных данных, верно? А есть ли такие данные, которые позволили бы нам ответить на эти масштабные вопросы? В подавляющем большинстве случаев ответ отрицательный. Для получения фактов и данных необходимы тщательные управляемые эксперименты, и едва ли хоть кто-нибудь их проводит. А почему? Если попробовать ответить на этот вопрос, то все запутывается еще сильнее.

Один из ответов состоит в том, что тщательные управляемые эксперименты влетают экспериментаторам в копеечку. Среднестатистический исследователь не располагает таким количеством денег. На самом деле денег требуется столько, что вряд ли кто-то ставит эксперименты, осо-

бенно в той сфере, где их значение наиболее велико – в сфере крупных и дорогостоящих программных продуктов, создаваемых профессиональными программистами. Эти эксперименты не проводят даже там, где их проводить следовало бы, например в субсидируемом правительством Институте программной инженерии (Software Engineering Institute) или в консорциумах, существующих на средства корпораций, таких как Консорциум продуктивности ПО (Software Productivity Consortium) и Корпорация микроэлектроники и компьютерных технологий (МСС).

Есть и еще один ответ: в вычислительной технике как области знаний и в разработке ПО сформировалась странная атмосфера, лишенная экспериментальной компоненты, имеющейся в большинстве других наук и инженерных дисциплин. Те ученые, квалификация и мотивация которых должна бы наилучшим образом соответствовать проведению экспериментов в сфере разработки ПО, попросту не проводят их.

Выводы из экспериментов по тестированию ПО существуют

Однако все это – лишь весьма унылое вступление к тому, что я хочу сказать *в действительности*. Ведь, несмотря на весь этот пессимизм, какие-то эксперименты в сфере ПО были осуществлены. А некоторые довольно важные эксперименты преследовали цель ответить на вопросы о тестировании. И кое-какие ответы у нас уже есть. Это не окончательные ответы, но они достаточно последовательны, чтобы программисты-практики и их менеджеры могли, опираясь на них, начать принимать решения. И в этом случае уже *можно*, опираясь на факты А, В и С, выбрать стратегию Z.

Этот очерк стоит читать (если вы дочитали до этого места), потому что выводы, которые следуют из данного эксперимента, не только относительно последовательны, но и показывают, что устранение ошибок в программном обеспечении надо вести не в том направлении, которое мы привыкли считать правильным.

Как это обычно делается сейчас? Протестируйте ПО жестко, потом еще немного, а потом еще.

Что говорят эксперименты по поводу того, что нам следует делать? Если считать, что наша цель состоит в том, чтобы отыскать как можно больше ошибок и/или затратить при этом как можно меньше денег в расчете на каждую найденную ошибку, то из экспериментов следует, что необходимо уделять больше внимания анализу, чем тестированию. То есть анализу результатов проектирования, написания кода и тестирования. Как мы скоро увидим, эти методы обычно позволяют обнару-

жить больше ошибок, чем тестирование, причем быстрее и с меньшими затратами.

Но подождите! Не торопитесь разрывать контракт с группой независимых тестеров и сокращать машинное время своих разработчиков во время тестовых испытаний. Данные, полученные в результате экспериментов, не говорят о том, что надо вообще отказаться от тестирования. Они лишь говорят, что одного тестирования недостаточно и что его необходимо дополнять анализом. И это очень важный вывод, потому что, насколько мне известно, не очень многие разработчики ПО уделяют достаточно внимания анализу на этапах проектирования и написания кода, прежде чем начать тестирование новой программы.

Итак, вступительных слов сказано достаточно. Какие исследователи занимаются экспериментальной работой и что им удалось узнать?

Данные, полученные в результате экспериментов

Я собираюсь рассказать вам о трех различных группах людей, которые занимались эмпирическими исследованиями в этой области, работая вполне независимо друг от друга (если не считать того, что самым последним были известны результаты, опубликованные их предшественниками). А именно я собираюсь рассказать о:

- Гленфорде Майерсе (Glenford Myers) из исследовательского института компании IBM, опубликовавшего результаты своей работы в статье «A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections» (Управляемый эксперимент в тестировании ПО, сквозной анализ/инспекции программного кода) в сентябрьском выпуске журнала *Communications of the Association for Computing Machinery* за 1978 год.
- Викторе Бэили (Victor Basili) из Мэрилендского университета и Ричарде Селби (Richard Selby) из Калифорнийского университета в Ирвине, которые работали совместно с Годдардовским центром космических полетов НАСА и опубликовали свои результаты в статье «Comparing the Effectiveness of Software Testing Strategies» (Сравнение эффективности методик тестирования ПО) в декабрьском номере журнала *IEEE Transactions on Software Engineering* в 1987 году.
- Джиме Коллофелло (Jim Collofello) из Аризонского государственного университета и Скотте Вудфилде (Scott Woodfield) из университета Бригема Янга, которые работали в неназванной компании и обнародовали результаты в статье «Evaluating the Effectiveness of Reliability Assurance Techniques» (Оценка эффективности методов обеспечения надежности) в мартовском номере журнала *Journal of Systems and Software* за 1989 год.

Это ответ на вопрос о том, какие ученые занимаются эмпирическими исследованиями в этой области. А теперь второй вопрос: «Что они узнали?»

Чтобы ответить на *него*, посмотрим сначала, что они *сделали*. В каждой из вышеупомянутых групп смысл вопроса понимали немного по-разному.

Майерс и Бэили/Селби сравнивали тестирование, которое называют «функциональным» (преследующим цель выяснить, все ли требования удовлетворены), со «структурным тестированием» (преследующим цель выяснить, все ли части программы были протестированы) и с анализом (экспертизой) программного кода. Свой эксперимент они ставили на одной или нескольких относительно небольших программах, привлекая к работе испытателей-практиков, обладающих обширным опытом, и поставив перед ними задачу искать ошибки, о наличии которых в ПО было известно.

Коллофелло/Вудфилд, с одной стороны, расширили базу исследования, а с другой – сузили ее; они взяли данные большого реального программного проекта, предпочтя не проводить управляемые эксперименты. И провели сравнительный анализ проектирования, написания кода и процесса, в целом характеризуемого как *тестирование*.

Исследователи из всех трех групп искали примерно одни и те же ответы, но немного по-разному их интерпретировали. Так, во всех трех группах хотели получить информацию о *количестве* устраненных ошибок и о соответствующих затратах, однако Коллофелло и Вудфилд ввели для этого новые метрики, назвав их *эффективностью обнаружения ошибок* (*error detection efficiency*) и *эффективностью затрат на обнаружение* (*error detection cost effectiveness*).

Я понимаю, что вам хочется узнать, что конкретно обнаружили исследователи из этих трех групп, однако прошу вас выслушать еще одно отступление. Оказывается, что способ проведения исследования может сильно влиять на то, что вы узнаете в результате (как бы нам ни хотелось обратного).

В известной степени в этих исследованиях некие яблоки сравниваются с некими апельсинами. Прежде всего, те, кто проводят анализ, обычно стремятся определить не только проблему, но и ее решение, тогда как в ходе тестирования (какое проводилось в этих экспериментах) независимые тестеры лишь констатируют, что ошибка существует, и не указывают, где именно эта ошибка может быть исправлена. Таким образом, и это здесь экспериментально измерено, эксперты, анализирувавшие код, выполняли больший объем работы, чем тестеры.

А кроме того, в исследовании Коллофелло/Вудфилда, где были задействованы проектные данные, тестирование выполнялось после того, как в результате анализа (экспертизы) была удалена изрядная доля ошибок, так что тестеры (1) искали ошибки не такого типа, как в двух других экспериментах, и (2) не исключено, что ошибки, которые уцелели в ходе экспертизы, распознавались с намного большим трудом, чем те, которые были устранены.

Эти виды различий могут оказывать на результаты исследований самое разное воздействие, от незначительного до очень глубокого. И я говорю о них здесь лишь для того, чтобы привести пример трудностей, с которыми связано проведение подобных экспериментов, и (что более уместно) чтобы показать, почему при выборе «стратегий» на основе «фактов» необходима осторожность.

А теперь (внимание!) вопрос: что же конкретно выяснили эти исследователи? Прежде всего, если говорить о сравнении анализа и тестирования, Бэили/Селби убедительно показали, что чтение кода было намного эффективнее, чем тестирование обоих рассмотренных ими видов, а по данным Коллофелло/Вудфилда, оба изученных ими вида анализа намного эффективнее, чем тестирование. Как ни странно, данные Коллофелло/Вудфилда были более убедительными в смысле анализа затрат, чем подсчета количества ошибок, тогда как в данных Бэили/Селби обнаружилась обратная тенденция. Как бы то ни было, эти две группы исследователей пришли к одному общему выводу – анализ (экспертиза) эффективнее тестирования.

Данные Майерса лишь частично согласуются в этом аспекте с данными других исследователей. Он пришел к выводу, что эффективность анализа и тестирования примерно одинакова, но анализ обходится дороже.

А что можно сказать о различных видах анализа и тестирования? Следует заметить, что больше данных собрано о разновидностях тестирования. Бэили/Селби выяснили, что функциональное тестирование позволяет обнаружить больше ошибок, чем структурное, а Майерс пришел к выводу, что данные виды тестирования в этом смысле примерно равнозначны. Разновидностями анализа интересовались только Коллофелло и Вудфилд. Они обнаружили, что устранение ошибок в процессе экспертизы проектирования обходится намного дешевле, чем экспертиза кода, однако в последнем случае несколько выше «КПД» устраненных ошибок.

В общем, все эти рассуждения, пожалуй, мешают понять, что же мы здесь пытаемся выяснить. Данные всех трех групп исследователей сведены в табл. 1.1. Применяв количественный критерий «более», то, что они, по-видимому, узнали, в упрощенном виде можно представить так:

1. Анализ более эффективен и обычно более выгоден, чем тестирование. Впрочем, собранные данные не позволяют утверждать это с полной уверенностью.
2. О разновидностях тестирования: функциональное тестирование обычно позволяет найти больше ошибок, чем структурное, и обходится дешевле.
3. О разновидностях анализа: анализ проектирования намного более выгоден экономически, чем анализ кода, но последний обычно позволяет найти больше ошибок.

Что все это означает применительно к методикам устранения ошибок? Надо полагать, приблизительно следующее:

- Анализ должен дополнять тестирование.
- В ходе тестирования необходимо придавать особое значение функциональности кода и не игнорировать его структуру.
- Анализ должен охватывать и проект, и программный код.

Таблица 1.1. Результаты исследования методик устранения ошибок (сравнение анализа и тестирования)

Исследователь	Майерс	Бэили/Селби	Коллофелло/ Вудфилд
Эффективность	Одинакова	1. Чтение программного кода 2. Функциональное тестирование 3. Структурное тестирование	1. Анализ программного кода 2. Анализ проектирования 3. Тестирование
Экономическая выгодность	Анализ программного кода дороже всего	Чтение программного кода дешевле всего	1. Анализ проектирования 2. Анализ программного кода 3. Тестирование (самое дешевое)

Точки над «и»

Однако остались некоторые неясности. Непонятно, например, что именно понимали исследователи под функциональным тестированием и структурным тестированием, чтением кода и анализом (экспертизой)? Было бы по меньшей мере неосторожно слишком полагаться на методики, опирающиеся на представленные выводы, не уяснив со всей определенностью, к чему именно эти выводы относятся.

При функциональном тестировании предмет тестирования определяется на основании спецификации требований к программному продукту. Майерс выдавал участвовавшим в эксперименте тестерам спецификацию и предоставлял им самим решать, как применять ее к тестированию. Участников экспериментов Бэсили/Селби просили, чтобы они, создавая сценарии (варианты) тестирования (test cases) на основе спецификаций, применяли эквивалентное разбиение (в одном тестовом сценарии представлен класс схожих тестовых сценариев) и граничный анализ (в сценариях тестирования особое внимание уделяется точкам, в которых происходят изменения в методах или алгоритмах).

При структурном тестировании создаются сценарии тестирования на основе внутренней логики функционирования программы. Участникам экспериментов Майерса предоставлялись и спецификации, и листинги программ, а также полная свобода в создании сценариев тестирования. От тех, кто участвовал в экспериментах Бэсили/Селби, требовалось убедиться, что каждый оператор программы протестирован хотя бы в одном сценарии.

Анализ программного кода – это статичный процесс (программа не запускается на исполнение), в ходе которого программный код анализируется на предмет обнаружения ошибок. В экспериментах Бэсили/Селби был задействован процесс, который сами участники называли *постепенным обобщением* (stepwise abstraction) и в ходе которого определялись главные подпрограммы тестируемого ПО и их назначение. Для анализируемой последовательности инструкций кода строилась функция, которую они вычисляют, и такие производные функции уже проверяли на соответствие спецификациям. Участники экспериментов Майерса избрали в качестве метода экспертизы *сквозной контроль* и *инспекции*, при этом экспертизе предшествовало чтение программного кода и мысленный прогон сценариев проверки программной логики.

В работе Коллофелло и Вудфилда не указано, какие процессы были задействованы в тестировании и экспертизах, однако описание рассматриваемого ими проекта (700 000 строк программного кода реального времени, который создавался на современном языке высокого уровня при участии 400 разработчиков; на каждой из основных фаз жизненного

цикла применялись процедуры обеспечения качества) позволяет предположить, что это был либо военный проект, подпадающий под стандарты экспертизы и тестирования, принятые Министерством обороны, либо какой-то другой, подчиняющийся аналогичным стандартам.

Прочие удивительные результаты

Тем, кто любит покопаться в «пыльных артефактах» старых экспериментов, могу предложить еще кое-какие интересные результаты, которые пока еще никто не обобщал.

Любопытные факты были найдены Бэзили и Селби:

1. «Количество ошибок, доля обнаруженных ошибок и суммарные усилия, затраченные на обнаружение, зависели от типа тестируемого ПО». Иначе говоря, выбор методов тестирования, вероятно, должен определяться типом тестируемого приложения.
2. «Чтение программного кода позволило обнаружить больше ошибок в интерфейсе, чем любые другие методы», «функциональное тестирование выявило больше ошибок, чем остальные методы». Другими словами, выбор методов проверки, наверное, должен зависеть от типов искомых ошибок.
3. «Когда экспертов, читавших программный код, просили оценить количественно, какую долю ошибок им удалось обнаружить, они давали самые точные оценки, а оценки тестировщиков, проводивших функциональное тестирование, были наименее точными». Вывод: чтение программного кода, по-видимому, позволяет человеку лучше понять проделанную работу.

А вот интересные находки Майерса:

1. «Результаты, получаемые конкретными специалистами, могут отличаться удивительным разнообразием». Из этого следует, что сам тестировщик может быть намного важнее, чем метод тестирования.
2. «Конечный результат в целом довольно унылый». Иначе говоря, ни один из методов устранения ошибок не показал себя достаточно эффективным в распознавании ошибок (и ни одно из сочетаний этих методов не проявило себя в таком качестве).
3. Наблюдалась «отрицательная корреляция между производительностью тестировщика и его опытом в проведении сквозного контроля и/или инспекций...» Другими словами, бывает и так, что эксперты, занимающиеся анализом программного кода, со временем «устают от своего опыта».

Коллофелло и Вудфилд тоже нашли дополнительную информацию:

1. «К сожалению, значительная часть полученных данных была противоречивой и ненадежной... Немногочисленность данных, по-видимому, отражает тот факт, что у разработчиков отсутствует интерес к их записи... Никто не проверял добротность данных по обеспечению качества». Иначе говоря, даже данные, взятые из реальных проектов, не гарантируют отсутствие затруднений в научном исследовании.
2. «Высокий уровень успеха, связанный с экспертизой проекта, вызвал удивление». То есть можно сказать, что очень много ошибок было найдено при экспертизе проекта ценой очень малых затрат.
3. «Интересно, что один из методов обеспечения надежности (reliability assurance), а именно тестирование, оказался экономически невыгодным». Это означает, что тестирование – это хотя и необходимый, но довольно дорогой способ получить требуемый результат.

Главный вывод этого очерка может быть заимствован из работы Коллофелло и Вудфилда. «Тестирование необходимо, но следует отдавать себе отчет, что традиционный подход, при котором особое внимание уделяют тестированию, а не экспертизам, не является экономически целесообразным».

Другими словами, это ответ на два вопроса, которые были поставлены в начале данной статьи:

- Какой способ устранения ошибок в ПО самый эффективный?
- Какой способ устранения ошибок в ПО наименее затратный?

Это довольно неожиданно. Нельзя сказать, что раньше мы недостаточно хорошо старались ответить на них, просто появились новые ответы, которые могут заставить нас пересмотреть наши методы.

Эти ответы мы слышим от людей, которые наконец начинают применять экспериментальные методы к научным аспектам компьютерных наук и к инженерии программной инженерии.

Многоликое тестирование

Имеются доказательства того, что при разработке ПО без тестирования по-прежнему не обойтись, и так, по-видимому, будет всегда. Экспертизы, согласно недавним исследованиям, могут быть менее затратными, доказательство корректности программы (если этот подход когда-нибудь достигнет до более крупных задач) может отличаться большей строгостью, но ни один из этих методов не в состоянии поместить программу в почти реальное окружение и испытать ее.

И если мы понимаем, что у тестирования богатые перспективы, значит, стоит потрудиться, чтобы узнать, что же оно означает на самом деле. Я бы сказал, что у тестирования есть несколько разновидностей и что слишком часто под тестированием мы понимаем ничтожно малую их долю.

С моей точки зрения, классификация видов тестирования такова:

1. Прежде всего, назову *тестирование на основе целей* (goal-driven testing). Именно в этом случае причина, по которой проводится тестирование, и заставляет запускать тесты. Грубо говоря, существует четыре цели тестирования:
 - a. *Тестирование на основе требований*. В данном случае создается столько сценариев тестирования, сколько требуется для того, чтобы показать, что все требования к продукту были протестированы хотя бы однажды. Типичная матрица тестовых сценариев на основе требований строится так, чтобы обеспечить как минимум один тест для каждого требования. Сейчас имеются инструменты, поддерживающие этот процесс, и полное тестирование на основе требований представляет собой неотъемлемый этап в создании любого программного продукта.
 - b. *Тестирование, основанное на структуре* (структурное). В данном случае тестовые сценарии строятся с таким расчетом, чтобы нагрузить как можно более обширную часть логической структуры программы – насколько позволяет здравый смысл. Структурное тестирование должно дополнять (но ни в коем случае не заменять) тестирование на основе требований¹, потому что этот, последний, вид тестирования попросту слишком груб, чтобы обеспечить выполнение достаточного количества тестов. «Правильное» тестирование обычно охватывает примерно 60–70% логической структуры программы; при тестировании ПО, предназначенного для работы в критически важных областях, этот показатель намного выше и должен приближаться к 95%. Полнота тестирования может быть измерена при помощи инструмента, называемого анализатором тестового покрытия. Такие инструменты доступны на рынке ПО.
 - c. *Статистическое тестирование*. В этом случае выполняется столько тестов, сколько требуется для того, чтобы убедить заказчика или пользователя в адекватности тестирования. Тестовые сценарии строятся на основе типичного профиля использования, что позволяет по окончании тестирования делать примерно такие заявления: «Можно ожидать, что программа будет выполняться

¹ Выше было упомянуто как функциональное. – *Примеч. перев.*

успешно в течение 96% времени из расчета типичного использования». Статистическое тестирование призвано дополнять (не заменять) структурное и функциональное (основанное на требованиях) в тех случаях, когда заказчики или пользователи требуют понятных для них гарантий, что ПО готово и будет работать надежно.

- d. *Тестирование, ориентированное на ошибки* (основанное на рисках). Выполняется столько тестов, сколько требуется, чтобы обеспечить уверенность в способности тестируемой программы успешно пройти через самые неблагоприятные сценарии отказов. Выполняется анализ ситуаций, связанных с высоким риском, после чего программа изучается и в ней определяются участки кода, которые могут спровоцировать эти риски. Затем проводится углубленное тестирование этих участков. Тестирование, основанное на рисках, обычно применяется только для критичного ПО. Повторюсь также, что оно должно дополнять, но не заменять функциональное (основанное на требованиях) и структурное тестирование.
- 2. В дополнение к тестированию на основе целей существует *тестирование, основанное на фазах*. Суть тестирования, основанного на фазах, меняется по ходу процесса разработки. Как правило, необходимо проводить тестирование как компонентов программы, так и всей системы в целом. В так называемом восходящем тестировании различают три вида тестирования, основанного на фазах, которое обсуждается далее. При нисходящем тестировании программа постепенно объединяется в растущее целое и от модульного тестирования (unit testing) отказываются в пользу повторяющегося и расширяющегося интеграционного тестирования (integration testing).
 - a. Модульное тестирование представляет собой процесс, в ходе которого проверяется работа мельчайших компонентов всей системы, прежде чем они объединяются, чтобы образовать целую программу.
 - b. Интеграционное тестирование – это процесс, в ходе которого проверяется функционирование объединенных модулей и который призван показать, что программа в целом работоспособна.
 - c. Системное тестирование – это процесс проверки программы как целого в глобальном контексте системы, которую она поддерживает.

Сложные переплетения процессов тестирования на основе целей и на основе фаз подвергают знания тестировщика и его здравый смысл серьезному испытанию.

Когда, например, следует проводить структурное тестирование: во время модульного тестирования, или, может быть, интеграционного, или

системного? В этом разделе я бы хотел поговорить о подходах к объединению многочисленных и разнообразных видов тестирования. Возьмем для начала тестирование, основанное на целях, и разложим его на фазы.

Тестирование на основе требований означает разное на различных фазах. На фазе модульного тестирования оно означает проверку тех требований, которые относятся к тестируемому модулю. Во время интеграционного тестирования под ним понимают проверку всех требований к программе на уровне спецификации требований. А на фазе системного тестирования оно означает повтор интеграционного теста в новом окружении.

Структурное тестирование на различных фазах также имеет разное содержание. Во время модульного тестирования оно означает проверку работы каждого из элементов структуры самого низкого уровня программы, обычно логических ветвей (по причинам, в тонкости которых мы здесь не будем вдаваться, тестирование всех ветвей исполнения – процесс более тщательный, чем тестирование всех операторов). На фазе интеграционного тестирования оно означает проверку всех модулей, а на этапе системного тестирования под ним понимают проверку всех компонентов системы, считая, что программа как целое представляет собой попросту один или несколько компонентов.

Статистическое тестирование имеет смысл только на уровне системного тестирования объединенного программного продукта. Выбор метода тестирования зависит от приложения; как правило, заказчик или пользователь считает тестирование на системном уровне более значимым.

Тестирование на основе рисков можно проводить на любом уровне в зависимости от уровня критичности системы, но по-видимому, оно наиболее значимо на системном уровне.

Есть и еще один фактор: *кто* проводит тестирование? Обычно модульное тестирование выполняет разработчик ПО, интеграционное – несколько разработчиков и независимых тестировщиков, а системное – независимые тестировщики и, может быть, системные инженеры. Заметьте, однако, что для статистического и функционального тестирования не требуется глубокого знания внутреннего устройства и функционирования тестируемой программы или системы, тогда как в случае структурного тестирования и тестирования на основе рисков такое знание обязательно. Таким образом, повсеместное и непрерывное участие разработчика в тестировании может оказаться необходимостью.

В определенных кругах очень часто говорят о том, что тестирование – это процесс вовсе не строгий, но такое представление в корне неверно. Я придерживаюсь абсолютно противоположного взгляда на тестирова-

ние. При условии надлежащего исполнения тестирование может быть и строгим, и тщательным. Вся штука в том, чтобы знать, как это делается, и воплощать это знание в жизнь. Я надеюсь, что это короткое обсуждение поможет вам увидеть основные принципы тестирования в правильном свете.

Качество ПО и сопровождение ПО. Какая связь?

Что такое качество программного обеспечения? Чаще всего о нем говорят как о совокупности признаков. Предполагается, что ПО качественное, если оно:

- Надежное
- Эффективное
- Эргономичное
- Понятное
- Модифицируемое
- Тестируемое
- Переносимое

Что такое сопровождение ПО? Это процесс поддержания способности ПО функционировать, удовлетворяя нужды пользователя.

Интересно отметить, что из этих семи признаков качества почти все прямо или косвенно относятся к ключевым понятиям сопровождения ПО. На самом деле два из них относятся конкретно к сопровождению, а большинство других критически важны для него. Другими словами, задача обеспечения высокого качества ПО, если рассматривать ее правильно, практически не отличается от задачи сделать ПО легким в сопровождении.

К сожалению, такой взгляд на качество ПО слишком необычен. В большинстве случаев мы представляем себе качество как результат деятельности разработчиков, который контролируют специалисты из службы обеспечения качества, относясь к нему как чему-то, что имеет собственную ценность. Вы создаете программное обеспечение? Благодаря этому оно, безусловно, обладает качеством.

Но это слишком близорукий взгляд на вещи, потому что вопросы, связанные с сопровождаемостью, решать труднее всего и поэтому о них проще всего забыть, если считать, что качество получается автоматически.

Посмотрим на приведенный выше список ключевых признаков и попытаемся исправить «близорукий» подход, включив в рассмотрение во-

просы сопровождения. Вот первые два признака, которые привлекают к себе внимание с точки зрения сопровождения:

- *Понятность* означает, что специалист, который читает программный код, может понять его назначение.
- *Модифицируемость* означает, что программный код, который читает этот специалист, может быть изменен.

Эти два признака качества практически целиком относятся к сопровождению. Едва ли кому-то, кроме специалиста по сопровождению, потребуется модифицировать программный код, а тех, кому надо понимать его, еще меньше! (И уж конечно, эти два признака качества труднее всего поддаются оценке.) Что можно сказать об остальных признаках? Перечислим их, расположив в порядке убывания их значимости с точки зрения сопровождения:

- *Надежность* – способность ПО безотказно выполнять возложенные на него функции. Это и есть главный элемент сопровождения! Если ПО ненадежно, то исправить этот недостаток должен именно специалист по сопровождению.
- *Эффективность* – это способность ПО функционировать, потребляя минимум ресурсов (времени и места). Повторюсь, что это свойство имеет не меньшее отношение к сопровождению, чем к любой деятельности разработчиков. ПО, которое работает слишком медленно или занимает слишком много места, должно быть ужато специалистом по сопровождению ПО.
- *Тестируемость* – это свойство, означающее, что ПО может быть протестировано без особого труда. Как правило, считается, что тестированием занимаются разработчики, однако бесспорно, что оно составляет значительную долю в труде специалиста по сопровождению, который тестирует каждое изменение, внесенное в программный продукт, и подвергает регрессионному тестированию те участки кода, которые не изменились.
- *Эргономичность* – это свойство, означающее, что программный продукт легко использовать. Кто принимает на себя всю силу удара, если работать с ПО трудно? Специалист по сопровождению.
- *Переносимость* – это свойство, характеризующее легкость, с которой ПО можно адаптировать к работе в другом окружении, например на иной аппаратной платформе или в другой ОС. И здесь то же самое: если ПО портируется, то весьма вероятно, что выполнять эту работу позовут специалиста по сопровождению.

Не только первые два признака качества имеют прямое отношение к сопровождению, остальные связаны с ним не менее тесно. Таким образом,

качество программного продукта связано с сопровождением не меньше, чем с чем бы то ни было еще.

Конечно, есть и другие свойства качества ПО, о которых упоминают те, кто занят обработкой информации. Они имеют отношение к сервисному обслуживанию, то есть наиболее значимы скорее с точки зрения пользователя, а не разработчика. Это своевременность, точность, надежность и цена. Непрерывное обеспечение этих дополнительных свойств полностью находится в сфере ответственности специалиста по сопровождению.

В чем суть всех этих рассуждений? Увы, почти в любом определении отрасли производства программных продуктов сопровождение присутствует лишь в форме запоздалых размышлений. Оно редко бывает предметом обсуждения в вычислительной технике, информатике и программной инженерии. Вплоть до последнего времени не было соответствующих инструментов и методов, и даже сейчас в научных исследованиях сопровождению уделяется весьма скудное внимание. Если мы обречем верное понимание того, насколько важно сопровождение для самой сути программной индустрии, то оно, может быть, наконец получит то внимание, которого заслуживает.

Барри Боэм высказал мнение, что главным человеком в команде обеспечения качества программного продукта должен быть тот, кто в конце концов будет сопровождать этот продукт. Это первый шаг в верном направлении. Он не только решает проблему, но и четко связан с этими двумя вопросами качества и сопровождения ПО, которые ныне редко увязываются друг с другом.

Сопровождение ПО – это решение, а не проблема

Сопровождение программного продукта – это проблема?

Сегодня стандартный ответ: «И еще какая».

Стандартное обоснование этого стандартного ответа: «Посмотрите только, какую часть нашего бюджета мы вкладываем в сопровождение. Если бы мы сразу разрабатывали ПО лучше, нам не пришлось бы тратить все эти деньги на сопровождение».

Я хочу сказать, что этот стандартный ответ неверен. Он неправильный, говорю я, потому что неправильно его стандартное обоснование.

Все дело в том, что сопровождение программного продукта – это не проблема, а решение!

При традиционном взгляде на ПО как на проблему мы теряем из виду особое значение двух фактов:

1. Программный продукт – более «податливый» (легко модифицируемый) материал по сравнению с другими, более «жесткими».
2. При сопровождении ПО на исправление ошибок в программном коде тратится намного меньше усилий (17%), чем на его усовершенствование (60%).

Иначе говоря, сопровождение ПО позволяет разрешить проблему, а не создает ее, потому что при сопровождении мы можем сделать нечто такое, что не способен выполнить никто другой, и потому что, когда мы это делаем, мы обычно создаем новые решения, а не просто заделываем старые прорехи. Позволяет ли такой взгляд на сопровождение по-новому понять его и сделать более качественным?

Я считаю, что да.

Согласно традиционному подходу к сопровождению, при котором оно считается проблемой, главной целью объявляется снижение затрат. Повторюсь, что считаю такую расстановку акцентов неправильной. Если сопровождение – это решение, а не проблема, то нетрудно понять, что *на самом деле* нам надо, чтобы его было больше, а не меньше. И главное внимание при этом следует уделить максимальному повышению эффективности, а не как можно большему сокращению расходов.

Такой образ мысли открывает перед нами новые перспективы. Переместив фокус интеллектуальных усилий на максимальное повышение эффективности, мы переходим на новый уровень понимания, позволяющий нам действовать по-новому. Действовать *как?*

Лучший способ добиться максимальной эффективности состоит в том, чтобы привлечь лучших специалистов. Этот вывод подтверждается массой фактов. Многие из них содержатся в литературе, посвященной индивидуальным отличиям, из которой можно узнать, в частности, что некоторые люди справляются с задачами программирования значительно лучше остальных:

1. Отладка: некоторые работники лучше других в 28 раз.
2. Обнаружение ошибок: превосходство одних над другими бывает семикратным.
3. Производительность: пятикратное превосходство одних над другими.
4. Эффективность: одни лучше других в 11 раз.

Главный вывод этого кратчайшего обзора индивидуальных отличий состоит в том, что разница между людьми бывает огромной и что самый надежный способ сделать работу как можно эффективнее – это привлечь к ее исполнению лучших людей.

Это логически приводит нас к двум следующим вопросам:

1. Так ли уж необходимо привлекать к сопровождению лучших специалистов?
2. Привлекаются ли ныне к сопровождению лучшие специалисты?

Наверное, на первый вопрос ответить труднее, чем на второй. Мой ответ на первый вопрос: «Да, сопровождение – это самый крепкий орешек во всей программистской отрасли». Объясню, почему я так думаю.

Несколько лет назад я участвовал в написании книги по сопровождению ПО. Вот что сказал по поводу сопровождения ПО один из ее анонимных рецензентов:

Задача сопровождения:

- Сложна интеллектуально (она требует новаторского подхода, накладывая на того, кто ее решает, жесткие ограничения)
- Трудна технически (специалист по сопровождению должен быть в состоянии оперировать концептуальными понятиями, решать вопросы проектирования и работать с программным кодом, причем все это он должен делать одновременно)
- Ставит специалиста по сопровождению в неблагоприятные условия (он никогда не получает то, что ему необходимо, например толковую документацию по сопровождению)
- Беспросветна (специалист по сопровождению имеет дело только с теми, кто отягощен проблемами)
- Обрекает специалиста по сопровождению на черную работу (он спускается на примитивный уровень написания программного кода)
- Помещает его в атмосферу прошлого (программный код вполне может принадлежать перу программиста, который еще не овладел профессией в полной мере)
- Консервативна (девиз сопровождения: «Не чини то, что еще не сломалось»)

Моя главная мысль, как и главная мысль этого рецензента, заключается в том, что задача сопровождения ПО очень сложна и требует колоссального напряжения.

Однако вернемся к вопросу о том, кто занимается сопровождением. В большинстве случаев им занимаются те, кто пришел в профессию недавно или не очень хорошо справляется с разработкой. На то есть причина. Большинство людей предпочитают разработку сопровождению, потому что последнее слишком тесно ограничивает творческую свободу, чтобы они могли получать от него удовольствие. Следовательно,

в сферу сопровождения чаще всего идут наименее способные и наименее востребованные.

Тем, кто следил за ходом моей мысли, должно быть очевидно, что это положение дел абсолютно неправильно. Сопровождение – это серьезный интеллектуальный вызов, – кроме того, что это решение, а не проблема. Если мы хотим добиться максимальной эффективности сопровождения, то должны совершенно иначе подойти к выбору исполнителей.

У меня есть конкретные предложения. Не теоретические рассуждения, которые сулят журавля в небе. Эти шаги вполне реальны, если, конечно, менеджмент решит, что хочет их сделать:

1. Сопровождение необходимо сделать привлекательным. Найдите способ привлечь специалистов к задаче сопровождения. Некоторые компании идут по пути выплаты премий тем, кто занимается сопровождением. В других необходимо проработать в этом качестве какое-то время, чтобы продвинуться по карьерной лестнице в руководители высшего звена. Где-то этого добиваются, давая сотрудникам понять, что лучший способ стать всесторонне развитым специалистом в индустрии программных продуктов – это в совершенстве изучить ПО компании.
2. Связать сопровождение с обеспечением качества. (Об этом говорилось в предыдущем очерке.)
3. Планировать улучшение технологии сопровождения. Сейчас создано множество инструментов и методов, позволяющих обеспечить более высокое качество сопровождения ПО. (За последние пару лет в этой области произошли разительные перемены.) Выбор и приобретение инструментальных средств, обучение персонала работе с ними должны иметь высокий приоритет в списке задач, решаемых менеджерами в сфере сопровождения ПО.
4. Усилить роль «добросовестного программирования». Как правило, специалист по сопровождению работает в одиночку. Лучший способ максимально повысить эффективность таких работников состоит в том, чтобы сообщить им чувство ответственности за качество работы. Заметьте, что эта точка зрения противоположна распространенной ныне моде на «обезличенное программирование», которое жертвует личным вкладом программиста в конечный продукт в пользу командного. Исключительно важно, чтобы вклад конкретного специалиста в качество сопровождаемого продукта был известен (если мы хотим, чтобы качество сопровождаемого продукта продолжало оставаться высоким).

Вот они, четыре простых шага, которые ведут к более качественному сопровождению ПО. Заметьте, однако, что для совершения каждого из

них требуется изменить привычный образ мысли. Технически такой переход несложен, несколько более трудными могут оказаться его социальный и политический аспекты. Большинство людей с огромным трудом отказываются от привычного для них взгляда на вещи.

Но если мы хотим получить результат, то совершенно необходимо сделать один жизненно важный шаг. Это шаг, упомянутый в самом начале данного очерка.

Мы должны понять, что сопровождение ПО – это решение, а не проблема. Если мы с этим согласны, то открываем путь к коренным изменениям в нашем подходе к программированию. Подумайте об этом.

Управление из одной точки

С точки зрения учебного процесса наука состоит из фундаментальных принципов, которые можно выделить и изучать и которые являются истинными.

Каковы основополагающие начала программной инженерии? Некоторые говорят, что программная инженерия – это «гуманитарная наука», что не так-то просто установить ее основополагающие начала.

Я хотел бы предложить свой вариант. Моим кандидатом на роль основополагающего начала программной инженерии выступает понятие, которое мой коллега Ли Макларен (Lee MacLaren) из компании Boeing Military Aircraft называет «одноточечным управлением» (single-point control).

При такой организации вычислений задача, которая должна решаться в нескольких местах, решается только в одном, а во всех остальных местах на это одно лишь ссылаются при необходимости.

В качестве самого распространенного примера такого управления можно привести программный *модуль*, например библиотечную подпрограмму. Если нам требуется извлечь квадратный корень или выполнить сортировку в нескольких местах в программе, то мы не пишем всякий раз один и тот же программный код, решающий эту задачу. Этот код мы располагаем в каком-то одном месте и *обращаемся* к нему (вызываем его) по мере необходимости.

Почему мы это делаем? Конечно, потому, что такой код занимает меньше места. Разумеется, потому, что это упрощает программную логику. И потому, что если необходимо изменить программный код, то это достаточно сделать один раз.

Модульная организация программы – это лишь один, хотя и самый важный, пример такого управления вычислениями. Есть ли еще примеры?

Именованные константы, к которым программист может обращаться по имени в ходе разработки и сопровождения кода. Допустим, к примеру, что в программе определен массив из 100 элементов. Кроме того, что этот массив надо объявить, вероятно, придется организовать цикл и присвоить его элементам значения. С точки зрения программиста, это может выглядеть примерно так:

```
Array TABLE (100);  
For I=1,100 Do...  
  TABLE[I] = <значение>;  
  I=I+1;  
If I>100 Then Do-<действия>;
```

Теперь посмотрим на этот код глазами специалиста по сопровождению. В один прекрасный день заказчик может потребовать, чтобы таблица содержала 150 строк, а не 100. Что придется сделать тому, кто будет сопровождать этот код? Изменить константу в объявлении массива (100). Изменить граничное значение счетчика цикла (100). Изменить значение условия окончания таблицы (100). Найти все остальные вхождения константы 100, имеющие этот смысл, и изменить их. При этом *нельзя* изменять другие вхождения этой константы, которые не имеют отношения к таблице.

Вот мы и получили ситуацию, чреватую ошибками. Это не предвещает ничего хорошего в смысле качества исправляемого программного продукта. А теперь предположим, что мы поступили так:

```
TABLE-SIZE Constant Integer = 100;  
Array TABLE (TABLE-SIZE);  
For I=1, TABLE-SIZE Do;  
  ...  
If I>TABLE-SIZE Then Do-<действия>;
```

В этом случае для изменения размера таблицы тому, кто поддерживает код, достаточно изменить объявление константы TABLE-SIZE и перекомпилировать программу; если разработчик применял константу последовательно, то все остальные изменения произойдут автоматически. Обратите внимание, что в этом варианте у специалиста, сопровождающего код, намного меньше возможностей совершить ошибку.

Объявления данных – это еще одна возможность реализации принципа управления «из одной точки». Посмотрите на такой код:

```
A = UNPACK(2,5,PACKED-DATA);
```

Во многих старых языках программирования подобная функция могла бы применяться для распаковки битов со 2 по 5 объекта PACKED-DATA.

Программисту, пишущему на таком языке, подобный способ работы с битовыми строками мог бы показаться весьма удобным.

Трудности начинают проявляться, если в программе много обработки битовых строк и определения этих строк могут со временем меняться. Допустим, к примеру, что в программе обрабатываются биты со 2 по 5 нескольких разных объектов, причем в каждом из них распаковке могут подлежать несколько ячеек. Предположим, что задача изменилась, и распаковываются биты со 2 по 7. Специалист по сопровождению сталкивается здесь с такими же трудностями, как и в случае с таблицей, только в другом контексте.

Лучшие средства разрешения таких трудностей предоставляют языки с богатыми средствами описания данных. На одном из таких языков можно было бы написать примерно следующее:

```
Record PACKED-DATA  
FIELD Bits 2 through 5  
End Record;
```

Наличие дополнительных подходящих языковых конструкций делает возможным обращение к указанному полю (FIELD) по имени, и если спецификация цепочки битов изменится, нам достаточно будет изменить объявление и не отыскивать каждое обращение к этому полю. Аналогия с выгодами *именованных констант* очевидна, хотя их применение и связано с некоторыми трудностями.

В *программном коде с табличным описанием данных* логика задачи видна в таблице объявлений, а не в последовательности процедурных операторов. Обычно таблица содержит значения, например Булевы, или целые, или строки символов, которые можно проверить, чтобы определить логику исполнения программы.

Такой подход позволяет отразить изменение логики программы в табличных данных, а не в процедурном коде, который может быть разбросан в теле программы. И этот способ тоже позволяет добиться того, что изменения, производимые в одной точке, приводят к автоматическим и последовательным исправлениям во многих местах.

Похожа на описанную концепция *программного кода, управляемого текстовыми файлами*. Разработка улучшенных и пользовательских интерфейсов (особенно в микропрограммных продуктах), вызвала потребность усовершенствовать управление данными этих интерфейсов. Один из подходов к решению этой задачи состоит в том, чтобы поместить в файл *все* текстовые константы пользовательского интерфейса и позволить программе читать их оттуда по мере необходимости.

Недостаток этого способа в том, что чтение текста из файла происходит медленно. А преимущество в том, что весь текст собран в одном месте, где его очень легко изменить, если потребуется. Пользовательский интерфейс программы можно изменить, отредактировав текстовый файл, не трогая (не перекомпилируя) программу.

Так можно создать программу с мультязычным пользовательским интерфейсом, например с возможностью замены англоязычного интерфейса на испаноязычный. Таким образом, для создания другой, иная версия ПО достаточно создать новый текстовый файл, а не новую ревизию.

Вышеупомянутый недостаток, связанный с производительностью, легко устраняется. Хотя данные пользовательского интерфейса и содержатся в текстовом файле, они могут быть считаны программой при запуске на исполнение. Текст может оставаться в памяти постоянно во время работы программы, что радикально решает проблему накладных расходов на считывание текстовых строк из файла.

Принцип управления из одной точки, конечно, не является чисто программистским изобретением. Составляя документацию, мы тоже часто ссылаемся на какой-либо источник информации, не воспроизводя саму эту информацию всякий раз, когда о ней идет речь. В базах данных наличие главного набора данных позволяет обойтись без многократного ненужного дублирования информации. Составляя программу учебного курса, мы стараемся сделать так, чтобы один и тот же предмет читался только в одном курсе. Найдя и обозначив единственный курс, в котором читается предмет, мы разрешаем эту проблему.

Управление из одной точки находит массу применений. Поэтому я и выдвинул его в качестве кандидата на роль основополагающего принципа программной инженерии. У вас тоже есть кандидаты?

«Дружелюбный к пользователю» – модное выражение или прорыв?

Это выражение, *дружелюбный к пользователю*, слышали все. Оно было одним из модных словечек в 1980-х годах, но только ли это можно о нем сказать?

Я считаю, что нет. Разработчики ПО стали мыслить, оперируя пользовательскими категориями. Результатом такого образа мысли стала, по моему мнению, подлинная революция в разработке пользовательских интерфейсов программного обеспечения. На этом направлении достиг-

нут самый значительный прогресс в компьютерной области за последние десять лет.

Что я имею в виду, говоря «мыслить пользовательскими категориями»? Разработчик ПО ставит себя на место пользователя и создает интерфейс именно с этой точки зрения. Некоторые даже считают, что разработчик должен начинать с интерфейса, предпочитая такой подход более традиционному, при котором сначала решается техническая часть задачи, а очередь косметической отделки – интерфейса пользователя – наступает позже.

Пользовательские интерфейсы (Graphical User Interface, GUI) стали по-настоящему новаторскими, начав свой путь с работы, проведенной в исследовательском центре компании Xerox в Пало-Альто («Xerox PARC»), и эволюционировав в коммерчески успешный Apple Macintosh. Усилиями многочисленных конкурирующих софтверных компаний для большинства компьютеров разработана так называемая «оконная» технология, в которой экран, отображающий пользовательскую информацию, трактуется как рабочий стол, а одну информацию можно «накладывать» поверх другой. Графические интерфейсы стали намного более реалистичными, и пользователь может взаимодействовать с изображениями так же, как со словами и диаграммами. От GUI не слишком сильно отстает анимированная графика. Уже можно получить звук высокого качества (при условии, что покупатель готов за это платить), и хотя возможности ввода звука еще довольно ограничены, эта технология пользуется успехом там, где с этими ограничениями можно мириться.

Важно отметить, что весь этот впечатляющий прогресс имеет свою цену. Разработчик ПО, желающий создать удачный пользовательский интерфейс, должен быть готов к полутора- или даже двукратному удлинению времени разработки. Разработка пользовательских интерфейсов начинает оформляться в самостоятельную профессию. Для проектирования, а иногда и для реализации интерфейса, может потребоваться участие специалиста по инженерной психологии, если программный продукт предназначается для широкой аудитории.

Как разработчик ПО может развить в себе умение мыслить с позиций пользователя? Пол Геккель (Paul Heckel) полагает, что проектировщик должен оперировать категориями искусства, а не технологии и рассматривать задачу как коммуникативную. Он предлагает изучать традиционные художественные дисциплины, например создание фильмов, чтобы нащупать аналогии, которые помогут проектировщику успешнее овладеть умением мыслить по-пользовательски.

К «пользовательскому образу мысли» могут вести разные пути:

- Рыночные исследования – необходимо изучить пользователей, их желания и потребности (заметьте, что желания не всегда совпадают с потребностями, но желания игнорировать опасно).
- Участие пользователей в экспертизе – надо наладить обратную связь с пользователями и получать их отклики по мере создания интерфейса.
- Прототипирование – если пользователи не имеют четкого представления о своих желаниях и потребностях, создайте прототип и разрешите им поэкспериментировать.
- Обучение пользователей – помогите пользователям почувствовать себя комфортно при работе с интерфейсом, который вы в результате выберете. Организуйте курсы по обучению работе с продуктом.
- Поддержка пользователей – тщательно продумайте обучающий интерфейс, что поможет пользователям самостоятельно освоить работу с продуктом. Предоставьте возможность доступа по выбору к файлам справки. Опытным пользователям предоставьте средства ускорения интерфейса. Обеспечьте доступность хорошей пользовательской документации, к которой можно будет прибегнуть, когда не помогут остальные средства. Позаботьтесь о том, чтобы программа генерировала осмысленные диагностические сообщения.
- Авторитетные руководства по созданию пользовательских интерфейсов – каждая публикация по этой теме (см. список дополнительной литературы ниже) содержит от 10 до 100 методических рекомендаций по созданию хороших интерфейсов. Позаботьтесь о простоте, последовательности, предсказуемости и т. д. Составьте набор требований, подходящих для вашего приложения, и следуйте им.

И для разработчика ПО, и для менеджера программных проектов важно быть в курсе последних практических и теоретических достижений в этой области. Она развивается с такой скоростью, что неосведомленность может легко обратиться старомодностью.

Дополнительная литература по этой теме:

- Издательство ABLEX печатает серию «Human/Computer Interaction» (Взаимодействие человека с компьютером), в которой уже вышли:
 - «Advances in Human-Computer Interaction» (Успехи взаимодействия человек-компьютер)
 - «Human-Computer Interface Design Guidelines» (Руководство по проектированию интерфейсов)

«Directions in Human-Computer Interaction» (Направления развития взаимодействия человек-компьютер)

- «Abstractions for User Interface Design» (Общие соображения о проектировании пользовательских интерфейсов), IEEE Computer, September 1985; Coutaz.

Обсуждение подходов для успешной *реализации* дружественных интерфейсов:

- «The Elements of Friendly Software Design» (Элементы разработки ПО, дружественного пользователю), Warner Books, 1984; Heckel. Утверждает, что проектирование хорошего интерфейса – это форма искусства, а не технология и что разработчики ПО должны изучать методы создания фильмов. Содержит советы по проектированию хороших интерфейсов.
- «Design Guidelines for the User Interface to Computer-Based Information Systems» (Руководство по разработке пользовательского интерфейса для компьютеризованных информационных систем), ESD-TR-83-122, 1983; MITRE Corp. Результат исследования, проведенного на средства ВВС США, содержит один из наиболее полных перечней принципов создания интерфейсов.
- «Designing the Star User Interface» (Разработка великолепного пользовательского интерфейса), Byte, April 1982; Smith, Irby, Kimball, Verplank, and Harslem. Список принципов, положенных в основу новаторского интерфейса компьютеров Xerox Star (который эволюционировал в интерфейс Apple Macintosh), написанного системными разработчиками Исследовательского центра Xerox в Пало-Альто.

Ретроспектива

Перелистывая текст первого издания, я удивляюсь, как быстро предметом книги стали технические материи. Это отнюдь не случайно.

На протяжении всей моей трудовой деятельности – а она началась в 1950-е годы, практически на заре программной инженерии – я сознательно избегал карьеры менеджера, руководящего программистами, и предпочел остаться техническим специалистом. Почему? Что за удовольствие руководить людьми, которым выпало делать интересную работу, когда ты мог бы делать ее сам?!

Итак, этот второй раздел посвящен техническим вопросам. Что может иметь более близкое отношение к технике, чем проектирование ПО? В очерке «Когнитивный взгляд: проектирование ПО с другой точки

зрения» я рассматриваю мир проектирования сквозь призму эмпирических работ нескольких известных исследователей и прихожу к важным для себя выводам о том, что значат для меня некоторые захватывающие откровения о существовании этой трудной для понимания темы. Эти мои коллеги прошлых лет сделали то, что я до сих пор считаю самыми решающими открытиями в области проектирования ПО, которые когда-либо были сделаны.

Меня интересуют еще две темы: тестирование и сопровождение. Долгое время эти направления программистской деятельности относились к числу самым непопулярным, и все-таки я испытываю тягу к ним, возможно, потому, что столь немногие уделяли им должное внимание. Это грустно, но вчерашние представления о тестировании и сопровождении, представленные в данном разделе, очень во многом справедливы и сегодня.

Что в этом разделе я нахожу устаревшим? Часть источников, на которые я здесь ссылаюсь, относятся к 80-м годам прошлого века. Было бы совершенно непростительно продолжать доверять такому древнему материалу – если бы не одно обстоятельство: никто не изучал эти темы, задаваясь теми же глубокими вопросами и достигая столь же глубокого понимания, как эти «ископаемые». Поэтому я приношу некоторые извинения за устаревшие на первый взгляд цитаты, но категорически отказываюсь делать это в отношении списка дополнительной литературы.

В этом и в последующих разделах встречаются также упоминания о некоторых организациях, игравших заглавные роли в индустрии ПО в то время, когда вышло первое издание этой книги, и затем постепенно пропавших из большинства сводок последних известий программной инженерии. В те далекие времена в сфере практических исследований программной инженерии ведущие позиции занимали: Институт программной инженерии (Software Engineering Institute, SEI), Консорциум продуктивности программного обеспечения (Software Productivity Consortium, SPC) и Корпорация микроэлектроники и компьютерных технологий (Microelectronics and Computing Consortium, MCC). Благодаря проводимым им работам по изучению процесса создания ПО Институт SEI, конечно, и сейчас находится на переднем крае отрасли. Организации SPC и MCC стали в основном реалиями прошлого. Очень жаль. Отрасли действительно нужны независимые организации, поставившие перед собой цель помочь всем нам продвигаться вперед в нашей работе.

Глава 3 | Новейшие вооружения

МЕТОДОЛОГИИ

- Повторное использование:
готовые части ПО – ностальгия и дежавю
- Автоматическое программирование – слухи с вечеринки?
- Некоторые мысли о прототипировании
- Стандарты и «блюстители» стандартов:
они действительно помогают?

ИНСТРУМЕНТЫ

- Джентльменский набор разработчика
- На всякий CASE: взгляд на последний «прорыв»
в технологии программирования
- Сравнение инструментов CASE и 4GL: что в сухом остатке?
- Зачем писать компиляторы?

ЯЗЫКИ

- Языки программирования высокого уровня:
насколько высок их уровень?
- Должны ли мы готовиться к доминированию 4GL?
- Сомнения по поводу Кобола

Ретроспектива

МЕТОДОЛОГИИ

Повторное использование: готовые части ПО – ностальгия и дежавю

Идея о том, что программное обеспечение должно строиться из готовых «деталей», очень подробно рассмотрена в литературе по программированию (см. список лит-ры на стр. 92 [BELA80], [KERN76], [DENN81]).

Это чрезвычайно заманчивая идея. Во-первых, изготовитель ПО может сократить и расходы, и сроки, потому что уже имеются части программы, написанные заранее. Во-вторых, изготовитель ПО одновременно может повысить качество продукта, потому что заранее протестированная программа, как правило, более надежна, во всяком случае, она надежнее только что написанной. А поскольку расходы, сроки и качество – это показатели, которые часто вступают в противоречие друг с другом, найти методику, способную улучшить их все одновременно, особенно приятно. Таким образом, идея о готовых частях ПО выглядит поистине волшебной. В эпоху, когда слово «производительность» стало самым модным, готовые части ПО должны находиться в зоне повышенного интереса.

Здесь кроются два парадокса. Один, который обсуждается в работе [GLAS81], состоит в том, что применение готовых частей для создания ПО – это концепция восходящей разработки, и она, таким образом, вступает в противоречие с моделями нисходящей разработки 1970-х годов. Второй, более интересный, заключается в том, что принцип готовых компонентов составлял существо индустрии разработки ПО более четверти века тому назад. За прошедшие годы в сфере готовых компонентов наша отрасль претерпела значительный регресс. Очень важно ответить на вопрос «Почему?».

Пожалуй, примерно 95% тех, кто разрабатывает ПО сейчас, в 1950-х годах не имели к этой отрасли никакого отношения. Поэтому имеет смысл ненадолго вернуться в то далекое прошлое. В раскрывшейся перед нами панораме мы увидим признаки, предсказывающие наступление эпохи готовых компонентов ПО. В таком случае, конечно, у прошлого можно поучиться.

Шаг в прошлое

Зайдем со мной в вычислительный центр 1950-х. О, на двери надпись «Вычислительная лаборатория». Войдем внутрь.

Наше внимание тут привлекают несколько обстоятельств. Короткие стрижки программистов-мужчин, пышные прически программистов-

женщин. Шум клавишных перфораторов. Необъятные просторы машинного зала – всю эту площадь занимает компьютер, по сегодняшним меркам просто крошечный.

Все эти детали могут быть важными (или неважными) с точки зрения ностальгии, но с технической позиции они бессодержательны. Посмотрим повнимательнее. Туда, на рабочие столы программистов. Что это за руководство?

Берем его с книжной полки, смотрим: на переплете написано «SHARE» (Совместно используемое). Открываем его, читаем. Это каталог программных компонентов, и *у каждого программиста есть либо свой экземпляр, либо доступ к общему.*

«А это откуда?» – спрашиваем у ближайшего молодого программиста. (Интересно, что все они молоды – это нетрудно заметить. Совершенно новая область, как магнит, одним своим полюсом притягивает молодость... а другим отталкивает опыт!)

«А, это руководство SHARE, оно общее, – говорит он небрежно. – SHARE – это пользовательская группа нашего поставщика. Мы все пишем программы и добавляем их в SHARE, и мы все берем отсюда то, что написано другими».

«А что на этой странице? Это генератор псевдослучайных чисел с равномерным распределением. Откуда он?»

«Его написал Фред Маснер из United Technologies. Он вообще очень много написал для SHARE».

«А эту подпрограмму чтения символьной строки?»

«Билл Клинджер из Northwest Industries. Он пишет отличный код. Его программы всегда работают корректно».

Прервем ненадолго наш визит в 1950-е. Важно кое-что понять. Прежде всего, вычислительной техники как научной дисциплины, достойной упоминания, в 1950-е еще не было. До начала этой стадии развития оставалось еще почти десять лет. В программирование приходили математики, бизнес-администраторы и даже дипломированные специалисты по английскому языку! А это, в свою очередь, означало практически полное отсутствие литературы по вычислительной технике. Выходил журнал *Communications of the ACM*, но этого было слишком мало. И более широко распространенный *Datamation*. И недолго просуществовавший *Software Age*. Публикации в компьютерной периодике того времени были неважным способом повышения престижа индустрии ПО.

Было и еще одно обстоятельство. Стандартное ПО, которое продавцы поставляли вместе с компьютерами, еще не было разобрано на состав-

ляющие. На самом деле оно еще не было собрано! Компьютеры нередко продавали вообще без программ. Тут-то на сцене и появились пользовательские группы, подобные SHARE.

И именно это на самом деле придает смысл названию SHARE. Назначение этой группы было в том, чтобы делиться программным обеспечением, которое попросту больше нигде было взять.

Второе приближение

Но вернемся в 1950-е. Теперь мы видим немного больше смысла в руководстве SHARE, представляющем собой коллекцию описаний готовых частей ПО. Посмотрим на него внимательнее.

Вот перед нами оглавление. Пробежав его быстро взглядом, мы видим, что части сгруппированы по функциональному назначению. Есть раздел с утилитами ввода/вывода, в другом разделе – работа с символьными строками, а вот группа математических функций... чего тут только нет. Откроем математический раздел, посмотрим, как он организован.

И в нем тоже ПО разбито на группы. Тригонометрические функции, подпрограммы для работы с матрицами и реализации численного интегрирования... есть и раздел с генераторами псевдослучайных чисел.

Ну что ж, углубимся еще немного. Что лежит в основе всей этой таксономии?

На этой странице нет ничего необычного. В первом абзаце описывается назначение компонента (функции, выполняемые программой). Далее мы видим имя автора и название фирмы, в которой он работает. Вот спецификация входных и выходных данных. И наконец описание ограничений и различные примечания. Мы видим, что одна часть обычно занимает одну страницу. Некоторые сложные программы, например утилиты ввода/вывода, занимают две-три страницы. Изредка, когда это важно, описывается алгоритм.

Но в любом случае в верхней части страницы приводятся имя автора и название фирмы. А внизу страницы, тоже всегда, отказ от гарантий. «Данное ПО протестировано, однако отсутствие ошибок в нем не гарантируется» или какая-нибудь аналогичная фраза.

«Программы-то хорошие?» – спрашиваем стоящего рядом программиста, озадаченные немного этими оговорками.

«Да почти всегда, – говорит он. – Почитайте код, и вы увидите, что обычно он очень хорош и – мне очень неприятно это признавать – лучше, чем самое гениальное, на что способен я. Как правило, в SHARE отдают только самое лучшее, потому что слишком многое при этом ставится на карту. Кроме того, если код плохой, это сразу видно».

«Слишком многое ставится на карту». «Если код плохой, это сразу видно». Все начинает представлять в другом свете. Производство готовых частей ПО в 1950-е было дорогой к престижу и славе в индустрии ПО, а в современную эпоху стимул к тому, чтобы «опубликовать код или погибнуть», выражен слабо. Успех программиста, отдающего код в общее пользование, достигался очень большими усилиями. Некомпетентность отсеивалась автоматически.

Полистаем SHARE еще немного. Конечно, некоторые имена и названия встречаются на каждой пятой или десятой странице. Вот почему уже знакомый нам программист мгновенно вспомнил Фреда Маснера и Билла Клинджера, а также United Technologies и Northwest Industries. Это очень интересно. И даже потрясающе. Пожалуй, пора оставить 1950-е и осмыслить полученные сведения.

Что мы узнали

Давайте вернемся, пройдем через дверь с надписью «Вычислительная лаборатория» обратно в 1980-е. Что мы узнали?

Во-первых, технология производства частей ПО в то время существовала и процветала. Каждый программист мог рассчитывать на то, что при необходимости найдет готовую программу.

Во-вторых, были созданы классификация готовых программ и толковая поисковая документация. Было очень просто узнать, какие готовые компоненты ПО имеются в наличии.

В-третьих, статус автора программы был почетным. Компоненты поступали в общее распоряжение, потому что к тому имелась сильная мотивация. Люди, создававшие ПО, получали поощрения.

В-четвертых, не было душной атмосферы внутренних противоречий коллектива. Продавцы компьютеров не поставляли программное обеспечение бесплатно или по низкой цене – можно было либо делиться программами, либо писать их самостоятельно.

И в таком свете 1950-е – это забытое время программистов с короткими стрижками – являют собой замечательный образец для современной эпохи. Парадокс и ирония в том, что мы идем туда, где мы уже были. А теперь пора вернуться к вопросу «Что было сделано неправильно?».

Я видел, как это произошло. Это грустная и неприятная история.

Главная неправильность заключалась в самой иронии по поводу происшедшего. По мере того как 1950-е годы плавно переходили в 1960-е, постепенно начало выясняться, что создавать программное обеспечение становилось все труднее. Скажем, пакетами утилит ввода/вывода можно делиться, выкладывая их в руководство вроде SHARE, а вот мож-

но ли так поступить с операционной системой? Участники встреч группы SHARE (и других сообществ пользователей) все сильнее и сильнее давили на продавцов компьютеров, чтобы те включали ПО в поставки. И продавцы в конце концов так и сделали. Так идея совместного использования ПО почилала... могли ли наши предки сделать это лучше и вернее? Встречи сообщества пользователей SHARE, открывающих свои решения для общего доступа, превратились в сборища пользователей, кричащих поставщикам «ДАЙ!».

Руководство SHARE вышло из употребления и в конце концов исчезло. Его место заняли километровые полки, занятые описаниями ПО от поставщиков компьютеров. В этих описаниях главное место отводилось системному ПО, много в них говорилось и об использовании различных инструментов внутри операционных систем, но понятие программных компонентов, за немногими исключениями вроде математических библиотек, попросту исчезло. В конце концов, изобилие таких компонентов заставило бы поставщиков намного активнее взаимодействовать с пользователями и сделало бы их более уязвимыми для критики. И разве могли бы они поместить отказ от гарантий под восхвалениями своих продуктов и уйти потом от юридической ответственности?

Произошло и еще кое-что, хотя роль этих событий в гибели сообщества пользователей программных компонентов была не так важна. Кафедры и факультеты вычислительной техники возникли в университетах по все стране, и постепенно стала появляться теория вычислительной техники. Энергия, которая тратилась на производство более качественных программных компонентов, теперь стала уходить на разработку более совершенных теорий их создания. Об этом лучше сказано в работе Белади (Belady) и Ливенворта (Leavenworth) [BELA80]:

«...разработка ПО поляризована, и в ней существуют две субкультуры – созидателей и умозрительных мыслителей. Первые наделены способностями к эффективной разработке продукта, но не к экспериментированию и должны прибегать в своей деятельности к надежным методам, какими бы несовременными они ни были. Вторые изобретают, но не идут дальше того, чтобы сделать новинку достоянием гласности, и поэтому ничего не могут сказать о практической полезности (или бесполезности) идеи».

Мы все осознаем важность и приветствуем развитие теории программирования, но думаю, что совершенно забыли о значении роли творцов, созидателей программного обеспечения и даже не жалеем об этом. В действительности творец очень часто становится объектом публичной критики теоретиков [DENN81].

И последним событием было появление концепции «обезличенного программирования». Для разработки ПО, сложность которого постоянно

возрастала, требовалось все больше программистов, и поэтому материализовалась идея командной работы, витавшая в воздухе. И человеческое Я программиста, кажется, встало на пути успеха таких команд. Да, так оно и было. Командный подход все-таки не учитывал, что человеческое Я – это основной движущий механизм, который нельзя подавить, не получив отрицательных побочных эффектов. Можете ли вы представить себе обезличенного менеджера? Или обезличенного теоретика, публикующего в профессиональных журналах статьи без подписи и, следовательно, без отзывов? Наше эго дает нам мощную мотивацию, и если его убрать, результатом будет чувство летаргической безответственности.

Не морочьте себе голову

А теперь, чтобы замкнуть цепь этого рассуждения, зададимся вопросом: разве не именно это было той самой неправильностью, которая произошла с SHARE – старым, подлинно общим фондом ПО? В конце концов появился сильный авторитет (поставщик), который сказал, что возьмет на себя заботу о программных компонентах и инструментальных средствах и что программистам не надо забивать себе этим головы. А поскольку ничье эго не было озабочено тем, чтобы сделать вклад в общий фонд программного обеспечения, поступление программных компонентов в него прекратилось.

Так что можно сделать, чтобы ускорить наступление эпохи готовых частей ПО в современный нам период? Учиться на опыте 1950-х, конечно. В своем вычислительном центре:

1. Создать классификатор частей (компонентов) ПО и документацию к ним.
2. Предложить программистам пополнять этот сборник своими разработками.
3. Разработать и установить систему поощрений для тех, кто откликнется на этот призыв.
4. Распространить каталог компонентов среди всех программистов.
5. Решить, что надо сделать:
 - a. сопровождать компоненты письменным отказом от ответственности, выбрав низкозатратный образ действия под девизом «Пользователь, берегись!»; или
 - b. учредить организацию, которая будет заниматься централизованной сертификацией программных компонентов, что, конечно, повысит затраты, но позволит получить более надежный результат.

К чему приведет выбранный путь, постепенно прояснится. В пределах вашей организации – а может быть, и нескольких организаций – ра-

зовьется процветающая субкультура готовых частей ПО. На этой субкультуре вырастет коллекция компонентов ПО, разработанных специалистами прикладного программирования, то есть теми, которые, скорее всего, лучше остальных понимают, какие именно компоненты нужны. А система вознаграждений позволит выявить группу лучших программистов, обладающих «неповрежденным» эго, у которых появится новая причина гордиться тем, что они делают, и осязаемое материальное подтверждение этой причины.

Пятидесятые годы XX века, дежавю. Мы знаем, что это может произойти, потому что это уже было раньше. Именно так.

Список литературы

BELA80 – L. A. Belady and B. Leavenworth «Program Modifiability», IBM Re-search Report RC8147 (#35397), 3/6/80.

Описывает экспериментальный подход к определению значения абстракций данных для улучшения модифицируемости программ; в качестве тестовой модели выступала операционная система с объемом кода 100 тыс. строк.

DENN81 – P. J. Denning «Throwaway Programs», *Communications of the ACM*, February 1981.

Рекомендует создавать компоненты программ, пригодные для использования, как способ решения проблемы одноразового ПО.

GLAS81 – R. L. Glass «No One Really Believes in Top-Down Design». *Software Soliloquies*, *Computing Trends*, 1981.

Обращает внимание на те факты, что проектирование зачастую итеративно и что в проектировании всегда задействуется восходящая модель.

KERN76 – B. W. Kernighan and P. J. Plauger «Software Tools» Addison-Wesley, 1976.

Описание набора инструментальных средств программирования, подчеркивающее роль каждой из составных частей.

WEIN71 – G. M. Weinberg «The Psychology of Computer Programming», Van Nostrand Reinhold, 1971.

Здесь обсуждаются обезличенное программирование и вопросы прав собственности на программы.

Автоматическое программирование – слухи с вечеринки?

Не так уж часто в серьезной компьютерной литературе появляется что-то, что оказывает сильное влияние на образ действия и мыслей профессионалов, программирующих вычисления и обработку данных.

Первым, кого я помню, наверное, был Бен Шнейдерман (Ben Shneiderman) из Мэрилендского университета, который в 1977 году раскритиковал применение блок-схем алгоритмов, подкрепив критику практическими результатами. После публикации его работы сообщество программистов очень скоро отказалось от блок-схем алгоритмов как необходимой компоненты проектирования и документирования ПО, признав их фактически бесполезными.

А несколько позже Дэвид Парнас (David Parnas) и Фредерик Брукс (Fred Brooks), каждый в своей работе, написали, что прорывов в технологиях создания ПО, на которые многие возлагали надежды, не предвидится. Для ученых и практиков программирования это был холодный душ. Эти две статьи до сих пор продолжают оказывать свое воздействие.

За этими работами последовала еще одна. Чарльз Рич (Charles Rich) и Ричард С. Вотерс (Richard C. Waters), опубликовавшие статью в августовском выпуске *IEEE Computer Magazine* за 1988 год, обратили пристальное внимание на сферу автоматического программирования и приуменьшили связанные с нею ожидания до уровня, обозначенного ими как «слухи с вечеринки».

Слухи с вечеринки? Что они имеют в виду?

Многие годы практики, теоретики и менеджеры индустрии ПО с нетерпением ждали того времени, когда программы будут создаваться другими программами, а не программистами. Эта самая автоматизация программирования была бы похоронным звоном по программистам – если, конечно, мы правильно понимаем их истинные цели. Кое-кто даже объявил, что эта эпоха уже наступила, поскольку применяются языки четвертого поколения.

Рич и Вотерс, чья работа «Programmer's Apprentice»¹ освещала вопросы, самые актуальные в этой сфере, еще сильнее остудили головы многих своих коллег, одержимых идеей автоматического программирования.

¹ Rich, C., Waters, R. C. «The Programmer's Apprentice: a research overview» (Подготовка программиста: обзор исследований). *IEEE Computer*, Т. 21, №11, Ноябрь 1988, с. 10–25.

По Ричу и Уотерсу, в понятии автоматизированного программирования заключено несколько мифов. Вот они:

- Системы автоматического программирования, ориентированные на конечного пользователя, не требуют знаний в специальных областях. Неправда, говорят Рич и Уотерс. Если система автоматического программирования вообще должна существовать, то она должна быть экспертом не только по части генерирования программного кода, но и в конкретной прикладной области.
- Универсальные, полностью автоматизированные, предназначенные для конечного пользователя средства программирования возможны. И это неправда, говорят Рич и Уотерс. Все известные работы по автоматизированному программированию пренебрегают хотя бы одним из этих факторов.
- Требования к программе могут быть исчерпывающими. Еще один миф. Требования для задачи любой сложности представляют собой незаконченные аппроксимации, которые превращаются в окончательные решения в результате последовательных приближений.
- Программирование – это последовательный процесс. Миф! Программирование – это итеративный процесс, в ходе которого между программистом и конечным пользователем происходит непрерывный диалог.
- Программирования больше не будет. И это тоже миф! Конечные пользователи станут программистами, соединив свои знания в прикладных областях и новую компьютерную специализацию.
- Программирование как явление крупного масштаба прекратит свое существование. Это последний миф. Мы никогда не примиримся с тем, что будут решаться только задачи для узких областей и только немногими людьми.

Иначе говоря, процесс написания программ достаточно сложен, и некоторые исследователи попытались этот факт проигнорировать. Вклад Рича и Уотерса состоит в том, что они указали, чем нельзя пренебрегать. Видят ли Рич и Уотерс, если учесть вышесказанное, хоть какие-то перспективы для автоматического программирования? Да, но только, по их собственному выражению, «будущие системы автоматизированного программирования будут больше похожи на пылесосы, чем на самоочищающиеся печи». Автоматизация поможет программисту, но не сможет заменить его.

Авторы проводят интересное противопоставление между академическими исследованиями, очень амбициозными, но не слишком успешными, и коммерческими продуктами, создатели которых ставили и ре-

шали более скромные задачи. Известны коммерческие системы для следующих категорий:

- Системы запросов к базам данных, обеспечивающие извлечение информации из БД.
- Языки четвертого поколения (4GLs), которые делают возможным создание быстрых программных решений для узкоспециальных задач, притом что эффективность конечного программного продукта оставляет желать много лучшего.
- Генераторы программного кода, которые концептуально (по способу реализации) похожи на языки четвертого поколения (они работают, скорее, не как интерпретаторы, а как трансляторы в код на компилируемом языке).
- Высокоуровневые средства проектирования, которые представляют собой инструменты, помогающие проектировщикам.
- Инструменты управления проектами, помогающие менеджерам отслеживать выполнение задач.

Рич и Уотерс считают, что на пути к автоматическому программированию «достигнут значительный прогресс». Однако, подобно своим предшественникам Парнасу и Бруксу, они не видят, чтобы в этом направлении были сделаны какие-то прорывы.

Некоторые мысли о прототипировании

Индустрия ПО породила массу областей с интересными противоречиями. Прототипирование – одна из них.

Все началось с того, что некоторые крупные ученые и профессионалы сферы ПО занялись «жизненным циклом ПО», а именно последовательностью этапов, через которые, говорили они, проходит все программное обеспечение, развиваясь от состояния идеи, существующей в чьем-то мозгу, до законченного решения, функционирующего в компьютере.

Некоторые говорили, что жизненный цикл представляет собой удобный способ описания процесса разработки ПО. К сожалению, за эту идею ухватились некоторые формалисты и объявили, что отныне так *должно* разрабатываться все ПО без исключения. Короткое слово «должно» не слишком удлинило формулировку, но основательно изменило ее смысл.

Естественно, такое искажение идеи жизненного цикла встретило сопротивление. Дэн Маккракен (Dan McCracken), автор многих книг по языкам программирования, и Майкл Джексон (Michael Jackson), из-

вестный специалист по проектированию структур данных, объединились и написали статью протеста под названием «Life Cycle Concept Considered Harmful» (О вреде идеи жизненного цикла). В конце концов началась открытая полемика.

Тем временем некоторые практики и теоретики вели работы в другом направлении, отличном от идеи жизненного цикла. Их увлекла идея прототипирования, то есть построения одноразовой версии программного продукта, предназначенной для того, чтобы проверить идеи, лежащие в основе разработки, прежде чем зафиксировать их. Со временем оказалось, что прототипирование конфликтует с концепцией жизненного цикла, и потому критики жизненного цикла включили идеи прототипирования в свой арсенал.

Разработчики ПО, однако, продолжали исследовать и развивать идею прототипирования независимо от ее полемичности. Одно из самых интересных исследований было предпринято Полом Геккелем (Paul Nessel, работавшим над созданием ПО для переносного электронного словаря, устройства, которое умело бы переводить слова по отдельности (чтобы, например, помочь туристу при необходимости узнать, как по-французски будет «гамбургер»)).

Никто до него не делал таких словарей, и задача Геккеля состояла не просто в том, чтобы написать программу, а в том, чтобы создать программное обеспечение, способствующее успеху продукта на рынке. Но благодаря чему не известный никому продукт может добиться успеха на рынке? Устройство должно иметь удобный размер, легко читаемый экран и дружелюбное ПО, которое помогает и которое любой новичок, не искушенный в электронных устройствах, мог бы применять, не впадая в ступор при каждой попытке что-нибудь перевести.

Оказалось, что этот проект как будто специально был придуман для прототипирования. Проблема, с которой столкнулся Геккель, коротко говоря, заключалась в том, что недостаточно четко были сформулированы исходные требования. Прототипирование помогло ему, позволив экспериментировать с различными вариантами продукта, который можно было подстроить для исследования меняющихся требований и найти такое их оптимальное сочетание, на котором можно было остановиться и создать программный продукт с четко определенными требованиями.

Конечно, прототипирование придумал не Геккель. Но он был одним из первопроходцев, и его статья «Designing Translator Software» (Проектирование ПО электронных словарей) в февральском выпуске *Datamation* 1980 года облекла всю концепцию прототипирования во вполне реальную и ясно различимую форму.

Остановимся и посмотрим на то, что сделал Геккель, и на противоречия между жизненным циклом и прототипированием. Вспомните, что при помощи прототипирования Геккель уточнял требования к продукту. Или, если смотреть на его работу с позиций жизненного цикла, он последовательно улучшал требования к продукту на соответствующем этапе жизненного цикла, пока не получал то, что его устраивало, после чего продолжал работу, переходя к проектированию и сборке окончательной версии продукта. Таким образом, жизненный цикл прототипирования, по Геккелю, выглядел так: требования – проектирование – реализация – фиксация требований и отбраковка продукта, а потом опять требования – проектирование – реализация.

И если вы считаете, что прототипирование и жизненный цикл противоречат друг с другом, то этот пример ясно показывает, в чем именно заключаются противоречия. Но работу Геккеля можно рассматривать как всего лишь небольшую вариацию на тему традиционного жизненного цикла. С той только разницей, что фаза определения требований у него была итеративной и включала проектирование и реализацию, предшествовавшие дальнейшему развитию остальной части жизненного цикла.

Однако такой взгляд на взаимодополняющую природу взаимосвязей между жизненным циклом и прототипированием не получил широкого распространения. Напротив, среди печатных работ преобладают посвященные теме «Прототипирование против точного описания...».

В середине 1984 года даже появилась пара статей, в которых различия между прототипированием и жизненным циклом исследовались экспериментально.

В настоящее время работы, посвященные экспериментальной оценке двух враждующих подходов, крайне немногочисленны, и мы должны приветствовать эти две статьи за предпринятую в них попытку рассмотреть противоречия объективно и беспристрастно. Статьи были интересными и хорошо написанными. Разве что противоречия были в них скорее обострены, чем приглушены.

В чем вообще была суть этих статей? В обеих документально фиксировался эксперимент, проведенный авторами и направленный на изучение преимуществ и недостатков двух «конкурирующих» методологий. Первая из этих статей называлась «Prototyping Vs. Specifying; A Multiproject Experiment» (Прототипирование или специфицирование; мультипроектный эксперимент) и была напечатана в журнале *IEEE Transactions on Software Engineering* в мае 1984 года. Исследование проводилось хорошо известным специалистом по разработке ПО Барри Боэмом (Barry Boehm) с коллегами.

Вторая статья, «An Assessment of the Prototyping Approach to Information Systems Development» (Оценка прототипированного подхода в разработке информационных систем), была опубликована на месяц позже в журнале *Communications of the ACM* и принадлежала перу Алави (Alavi) из Калифорнийского университета в Лос-Анджелесе (UCLA).

Базовая методика обоих экспериментов была одинаковой. Были задействованы несколько групп разработчиков, при этом некоторые создавали программный продукт, придерживаясь методологии жизненного цикла, а другие – прототипирования. По окончании разработки результаты оценивались в соответствии с хорошо продуманной системой критериев.

Что же показали результаты? Чтобы ответить на этот вопрос, были проведены интервью с пользователями и разработчиками ПО. Оказалось, что пользователи в целом оценивали продукты, созданные с применением прототипирования, как более удобные в работе, а отчеты, создаваемые этими продуктами, как более точные и полезные, однако (в эксперименте Боэма) функциональность ПО оставляла, по их мнению, желать лучшего.

Разработчики сообщали, что прототипированием труднее управлять, однако создаваемые проекты более логичны и последовательны, программный код легче писать и интегрировать. Таким образом, прототипирование в целом оказалось несколько более эффективным, но с определенными оговорками.

Так какое место занимает прототипирование? Центральное – в полемике. Известны интересные примеры его успешного применения, эксперименты, которые убедительно закрепляют этот успех, и некоторые важные концепции.

Как применять прототипирование на практике? Как оценить, сколько должен заплатить заказчик за программный продукт, созданный с применением прототипирования, если у вас нет четкого набора требований? Как избежать превращения первой версии продукта, сделанной наспех, в готовый продукт низкого качества?

На актуальность этих вопросов полемика не влияет. Уже больше десяти лет прошло с тех пор, как Фредерик Брукс сказал в своем знаменитом «Мифическом человеко-месяце», что первую версию продукта создают, чтобы выбросить ее, и мы до сих пор не знаем точно, как ответить на эти вопросы. Мы знаем только, что прототипирование – это перспективная технология, причем перспектива представляется вполне реальной.

Стандарты и «блюстители» стандартов: они действительно помогают?

О стандартах в индустрии ПО нередко говорят, что это последняя линия защиты качества программных продуктов. Но они не справляются с защитой. Современная практика применения стандартов показывает, что они больше похожи на линию Мажино, создавая иллюзию защиты качества и скрывая его серьезные нарушения.

Что мы понимаем под стандартами ПО? Это требования, которые предъявляют к работе программиста контролирующие структуры организации, менеджеры, заказчики и, может быть, коллеги. Они дополняют требования к продукту, указанные в проектной документации, и должны относиться не столько к функциональности конечного программного продукта, сколько к ремеслу программирования.

Говоря о стандартах ПО, важно упомянуть и их «блюстителей». Под «блюстителями» мы понимаем программы, проверяющие соблюдение стандартов. Обсуждать их надо совместно, потому что стандарты, которые устанавливаются, но не соблюдаются, – это все равно что полное отсутствие стандартов.

Введение стандартов преследовало цель повышения качества программных продуктов и производительности труда программистов. Например, стандарты могут предписывать правила именования переменных. Такие соглашения повышают производительность и качество, потому что снижают вероятность совершения ошибок именования переменных и улучшают читаемость кода, сберегая время и специалистов, занимающихся проверкой и сопровождением программ. Типичный стандарт может содержать:

- Требования к языку
- Ограничения на сложность программного кода
 - Ограничения структурного программирования
 - Ограничения на применение нерекомендуемых языковых конструкций
- Требования к модальности
- Требования к сопровождаемости
- Соглашения об именовании
 - Имена переменных должны отражать смысл данных ИЛИ
 - Имена переменных должны отражать структуру программы и данных ИЛИ
 - Любая комбинация обоих предыдущих требований

- Соглашения о комментировании
- Элементы программного кода, для которых прослеживается связь с дизайном продукта или с требованиями к продукту
- Интерфейсные структуры

Таким образом, задача программы-«блюстителя» состоит в том, чтобы проверить соблюдение возможно большего количества стандартов, определенных в документации по стандартам. Однако это проще сказать, чем сделать. Так, «блюститель» без затруднений проверит выполнение требований к сложности программного кода, но в проверке осмысленности имен переменных задействуется семантика, и эту проверку может выполнить только человек. Как правило, автоматизированный «блюститель» обнаруживает меньше половины нарушений стандартов, поэтому без участия человека не обойтись.

Откуда берутся стандарты? Существуют международные организации, задача которых состоит в том, чтобы разрабатывать стандарты общего назначения, однако большинство стандартов в настоящее время не являются официальными.

Что не так в этих рассуждениях

В современной практике применения стандартов и их «блюстителей» есть две серьезные проблемы. Первая заключается в том, что в большинстве организаций, связанных с разработкой ПО, пытаются соблюдать слишком много стандартов сразу. А вторая – что в большинстве таких организаций практически не применяются средства проверки соблюдения стандартов. Эти проблемы взаимосвязаны.

В большинстве программистских фирм документация по стандартам, если она вообще есть, насчитывает, вероятно, не меньше сотни страниц, до отказа заполненных замечательными правилами построения программных продуктов. Эта документация, скорее всего, написана людьми, которые либо (1) могли быть освобождены от написания программного кода, потому что их способности были не так активно востребованы, как способности их коллег, либо (2) отыскиали то, что показалось им «лучшим» способом написания программ, и захотели сделать так, чтобы все вокруг писали программы точно так же.

Все эти способы установления стандартов ПО неправильные.

Во-первых, стандарты должны быть короткими и дельными. Обязательные правила написания программ, независимо от того, в какой организации это происходит, необходимо собрать в кратком руководстве, которое должно быть легким в понимании и применении, а проверка соблюдения его требований должна быть удобной. Если имеют-

ся более подробные, предпочтительные, но не обязательные способы разработки ПО, то необходимо подготовить другой документ, содержащий соответствующие *рекомендации* (но не *стандарты*). Этот документ может быть настолько объемным, насколько это необходимо, потому что в нем приводятся полезные советы и указания, которые будут интересны любому программисту, и объем документа уже не будет таким препятствием, так как не возникнет никакой необходимости (или даже намерения) в жестком соблюдении приведенных в нем правил.

Во-вторых, написанием и анализом стандартов должны заниматься самые одаренные программисты, а не те, которые оказались свободными в данный момент. Правила, которые следует подчеркивать и соблюдать с особой тщательностью, заслуживают того, чтобы к их созданию привлекались лучшие доступные ресурсы.

В-третьих, проверка соблюдения стандартов должна выполняться обязательно, а не тогда, когда оказывается, что на это есть время. Стоит только принять решение о необходимости сократить стандарт, оставив в нем лишь то, что по-настоящему важно, как трудности, связанные с проверкой его соблюдения, быстро уменьшаются. К проверке соблюдения должны привлекаться – когда это возможно – автоматические «блустители» (их иногда называют *программными ревизорами* (code auditors)), а когда невозможно, необходимо проводить анализ (экспертизу) программного кода. Одним из важных способов проверки соблюдения стандартов является экспертная проверка кода, хотя при этом главное внимание должно уделяться качеству в целом, а не формальному соблюдению стандартов.

С проверкой соблюдения стандартов связана еще одна проблема. Нередко организации, занимающиеся обеспечением качества, проверяют соблюдение стандартов и делают вывод, что проверенное ПО имеет высокое качество.

Важно различать соответствие продукта стандартам и его качество. Собственно стандарты – это узкое подмножество всех вопросов обеспечения качества, и хотя очень заманчиво определить качество как соответствие достаточному набору стандартов, это попросту невозможно. Качество ПО определяется при помощи таких свойств, как переносимость, эффективность, соответствие требованиям инженерной психологии, понятность, тестируемость, модифицируемость и надежность. А если задуматься о том, как потребовать обеспечения этих свойств в программном продукте при помощи правил, быстро станет понятно, что качество ПО обеспечивается не законодательно, а только посредством совершенного исполнительного механизма. Попытки измерить

качество по шкале стандартов неизбежно приводят к тому, что плохое ПО проскальзывает незамеченным.

Что же в итоге дают нам стандарты и их «блюститители»? Они и вправду помогают повысить качество ПО и производительность труда программистов, но они требуют правильного применения. На этом пути много ловушек, и многие применяют стандарты *неправильно*.

А вы?

ИНСТРУМЕНТЫ

Джентльменский набор разработчика

В последние годы инструментальные средства развивались исключительно бурно. Рынок наводнен CASE-программами. Каталог инструментальных средств «The Guide to Software Productivity Aids» (Справочник по средствам повышения продуктивности ПО), выпускаемый Applied Computing Research, насчитывает более 300 страниц и содержит 25 *категорий* инструментов.

Но и в этом изобилии все равно чего-то не хватает. Того же, чего не хватало, когда я впервые написал статью с таким же названием (она была напечатана в *Software Engineering Notes* в октябре 1982 года).

Ни тогда, ни сейчас никто не определил минимального набора инструментов, который должны иметь все разработчики ПО независимо от того, какой методологии они придерживаются и в какой прикладной области заняты.

Я считаю, что программное обеспечение надо собирать при помощи инструментов (как при помощи молотка, отвертки и плоскогубцев мы собираем обычные детали). На первом месте в этом «джентльменском наборе» программиста – там, где лежат самые востребованные инструменты, – должно быть то, что есть в личном арсенале всех программистов. Самый сложный вопрос, насколько я понимаю, не в том, где взять побольше инструментов, а в том, какие инструменты необходимо приобрести.

Современное положение дел, при котором поставщики инструментальных средств имеют отличный объем продаж и при котором значительная часть инструментов отправляется прямиком «на полку», возникло именно потому, что стараемся ответить на первый из этих вопросов. («Полочное» ПО – это инструменты, которые были куплены, но оказались бесполезными и потому были поставлены на полку «собирать пыль»).

Я утверждаю и утверждал всегда (дольше, чем мне бы хотелось), что нужно обязательно определить этот необходимый (то есть «стандартный») набор инструментов.

Стандартизация? Это слово отпугивает? Пожалуй, да, в некотором смысле. В моем представлении, худшим следствием стандартизации могут быть колонны стандартизаторов, бездумно марширующих к никому не интересной цели. Однако стандартизация, если смотреть на нее с оптимистической точки зрения, может стать светом в конце тоннеля.

Проведем аналогию с обычной лампочкой, чтобы увидеть лучшие возможности, которые открывает стандартизация. Форма и размеры цоколя стандартизованы; форма и размер колбы – нет. Для того чтобы сделать производство лампочек *выгодным*, было необходимо стандартизировать цоколь. А от стандартизации колбы было необходимо отказаться, чтобы способствовать производству лампочек разных форм и размеров. Правильная стандартизация способна как стимулировать *творческие* способности, так и помогать в получении выгоды. Эта аналогия должна помочь нам понять, что именно надо стандартизировать в производстве инструментальных программных средств. Ясно, что «умеренная» стандартизация предпочтительнее «полной».

Чтобы показать значимость универсального минимального набора инструментов, расскажу одну историю. Недавно мне пришлось заняться одной программой, которая уже была создана. Программа не работала; то есть другой программист исправлял ее и каким-то образом привел в неработоспособное состояние. Моя задача состояла в том, чтобы устранить неисправность.

Инструмент, способный быстро решать подобные задачи, давно известен. Для обнаружения изменений, внесенных в последующие версии программы, применяются компараторы, которые сравнивают два файла и отыскивают отличия между ними. Программист, создавший некорректную версию, очевидно, не знал о существовании подобного инструмента. Я же, запустив имевшийся в моем распоряжении компаратор, быстро увидел, что, в дополнение к более осмысленным изменениям, мой предшественник случайно удалил ключевое слово `ELSE` из середины условного оператора `IF`. Проблема, поначалу казавшаяся серьезной и отнявшая у предыдущего программиста массу времени, была быстро разрешена.

Суть истории, конечно, в том, что если бы все программисты знали о существовании основных инструментов, включая компаратор, то они намного легче справлялись бы с некоторыми теперешними своими трудностями. Я бы взял на себя смелость утверждать, что компаратор – это один из главных претендентов на место в нашем стандартном наборе.

«Но разве у нас его еще нет? – можете спросить вы. – Разве, когда мы читаем книги или разговариваем с поставщиками CASE-программ, мы не говорим об одних и тех же по сути функциональных возможностях?»

Безусловно, нет. Первое подтверждение этому «нет» я получил, когда проводил поиск литературы во время написания упомянутой выше статьи. Я рассмотрел 16 наборов инструментальных средств и 38 функциональных возможностей. А когда я свел все наборы и функции в одну таблицу, то обнаружил, что у этих наборов нет общих главных функций, а наоборот, функции разбросаны по наборам инструментов.

Если бы вы провели такой анализ инструментальных CASE-средств от разных производителей сейчас, то обнаружили бы то же самое. Хорошо то, что сейчас появилась масса инструментов, которых не было десять лет тому назад. Однако никто не озаботился созданием минимального стандартного набора инструментов, и это плохо.

Большинство организаций, покупающих инструментальные средства, конечно, знают, каким арсеналом функциональных возможностей *они* хотели бы располагать. Однако в каждой из них свои предпочтения.

Приятная картина наблюдается в управлении по обслуживанию при правительстве США (U.S. Government Services Administration, GSA), выбравшем для всех правительственных агентств инструментальный набор под названием Programmer's Workbench (Автоматизированное рабочее место программиста). Этот набор инструментов подобран весьма тщательно, но функциональные возможности элементов перекрываются и в нем. Некоторые функции, в других организациях попавшие бы в разряд главных, здесь отсутствуют.

В состав APM от GSA входят следующие инструменты:

- Rand Development Center, объединяющий остальные инструменты
- ANALYZER, монитор тестового покрытия
- TRANSIT, транслятор
- DCD II, средство построения списков перекрестных ссылок
- RETROFIT, реструктуризатор Кобол-программ
- CSA, средство стандартизации именования данных
- HAWKEYE, форматировщик Кобол-программ
- DATA-XPRT, средство манипуляции файлами/базами данных
- PATHVU, анализатор метрик
- COMPAREX, компаратор файлов
- VIA/INSIGHT, анализатор исходного кода

За более подробной информацией об APM Programmer's Workbench обращайтесь по адресу:

Government Services Administration
(Управление по обслуживанию при правительстве США)

Office of Software Development and Information Technology (Служба разработки программного обеспечения и информационной технологии)
5203 Leesburg Pike, Suite 1100
Falls Church, VA 22041

Разработчики ПО испытывают острую необходимость в базовых инструментальных средствах. Как определить, что это за базовые средства? К сожалению, разные авторы до сих пор-разному понимают этот вопрос.

Далее я бы хотел предложить набор инструментальных средств, которыми каждая организация должна обеспечить своих разработчиков независимо от того, в какой операционной системе и на какой аппаратной платформе они работают, и увязать описание инструментов с фазами жизненного цикла разработки. Учитывая глобальность такого подхода, я, конечно, могу описать только функциональные возможности, а не конкретные инструменты. Имеется ли конкретный инструмент, соответствующий каждой функциональной возможности, зависит от состояния рынка в той конкретной области, в которой организация осуществляет свою деятельность. Однако, составляя свой список, я надеюсь:

- Помочь вам решить, отвечает ли предложенный набор потребностям *вашей* организации
- Предложить вам предварительный список покупок
- Стимулировать поставщиков инструментальных средств, чтобы они дописали в него функциональные возможности, которыми не обладают представленные в списке продукты

Итак, вот мой вариант минимального набора стандартных программных инструментальных средств.

Определение требований

Процессор представления требований (Requirements representation processor) – инструмент, автоматизирующий представление требований. Самым распространенным на сегодня способом представления являются диаграммы потоков и структурные схемы. Однако в данном случае выбор конкретного инструмента должен соответствовать методологическим предпочтениям, которые уже могли сформироваться в вашей организации.

Словари/репозитории данных (Data dictionary/repository) – средство, обеспечивающее централизованное хранение системных данных и работу с ними.

Проектирование

Процессор представления требований к проектированию (Design representation processor) – инструмент, аналогичный упомянутому выше про-

цессору требований. Из таких средств широкое распространение получил язык проектирования программ (Program Design Language, PDL), который поддерживается инструментами, не только создающими описание проекта, но и проверяющими его на действительность перекрестных ссылок, на логичность и последовательность. В этом случае тоже выбор инструмента определяется уже сформировавшимися предпочтениями.

Реализация

Условная компиляция – позволяет компилировать (или не компилировать) определенные участки исходного кода в зависимости от заданных условий. К примеру, один исходный файл может содержать несколько версий программы – производственную и отладочную или для PC и Макинтошей. Условная компиляция позволяет выбрать версию программы на этапе компиляции.

Отладка

Отладчик исходного кода – инструмент, позволяющий отлаживать программу на том же языке, на котором она была написана (выводит имена и форматированные значения переменных; обеспечивает трассировку переменных, позволяет устанавливать точки останова и т. д.).

Интерактивный отладчик – обладает такими же возможностями, что и отладчик исходного кода, и позволяет программисту, выполняющему отладку, взаимодействовать с программой.

Сопровождение

Глобальный список перекрестных ссылок – позволяет отыскивать все ссылки на любой именованный объект, например переменную или подпрограмму, в любом месте исходного кода программы.

Генератор структуры вызовов – показывает, какие программы вызывают друг друга.

Анализатор времени выполнения/производительности – позволяет выявить ресурсоемкие участки кода программы.

Управление

Менеджеры конфигураций – позволяют управлять синхронизацией и реструктурированием версий программного продукта и документации.

Документирование

Текстовый процессор – средство создания и сопровождения электронной документации.

Универсальные и прочие инструменты

Файловые менеджеры и менеджеры баз данных – обеспечивают создание больших объемов взаимосвязанных данных и доступ к ним.

Текстовые редакторы – средства создания и сопровождения электронной документации.

Компараторы файлов – многофункциональные инструменты, которые:

- Обнаруживают различия между версиями исходного текста (как документации, так и программного кода).
- Обнаруживают различия между результатами текущего и предыдущего, базового (baselined) и правильного (correct) сценариев тестирования.

Выше я дал определение своего варианта минимального набора важных инструментальных средств, которыми должна вооружить своих специалистов любая организация в индустрии ПО. Я хотел бы, чтобы этот минимальный набор был отправной точкой, а не конечной и чтобы организации приобретали больше инструментов, чем вошло в него (с учетом индивидуальных потребностей каждой из них, принятых в них методик, внедренных вычислительных систем и сформировавшейся среды).

Я даже надеюсь, что другие авторы отнесутся критически к моему списку и внесут в него свои ценные дополнения (помня, что минимальный стандартный перечень должен быть как можно короче).

Я глубоко убежден, что главную работу мы должны проделать именно при помощи *минимального* набора, чтобы удивительные результаты, которые удастся получить при помощи *всего* арсенала средств, не отвлекли нас от наших действительных нужд.

На всякий CASE: взгляд на последний «прорыв» в технологии программирования

Крупное достижение! Избавление! Чудо!

Если верить всем этим заявлениям, то можно подумать, что опять найдено средство от всех недугов индустрии ПО.

Какую панацею предлагают нам на этот раз? Это CASE-средства (CASE – Computer Aided Software Engineering), от которых ожидается почти такое же чудесное уменьшительное воздействие на время разработки программ, какое булавочный укол оказывает на объем воздушного шарика.

Неужели мы никогда ничему не научимся? Почему мы не оставляем поисков «эликсира жизни»? Почему по-прежнему не слышим таких экс-

пертов, как Дэвид Парнас и Фредерик Брукс (первый сказал, что в ближайшем будущем прорывов в индустрии ПО не предвидится, а второй – что нет никаких «серебряных пуль» для уничтожения «оборотней индустрии ПО»)?

Разработка программ – это самое суровое испытание интеллектуальных способностей человека, и ждать появления панацеи – все равно, что отрицать это.

Правда заключается в том, что разработка ПО может и должна ускориться благодаря людям, а не инструментам или технологиям. (Эту мысль наглядно иллюстрирует обложка книги Барри Боэма (Barry Boehm) «Software Engineering Economics» (Экономика программной инженерии).) Затянувшаяся эйфория по поводу инструментальных средств и технологий уводит нас в сторону от потенциально более перспективных направлений.

Дело в том, что мы пытаемся собрать полный комплект программных инструментов, не договорившись даже о том, что должна представлять собой его основа – «молоток, плоскогубцы и отвертка» ПО. У нас нет количественных данных о том, как строится ПО, и поэтому мы не сможем оценить революционность прорыва, даже если он произойдет у нас на глазах!

Иначе говоря, торговцы от программирования опять пытаются заморочить нам голову рекламным звоном, возводя в ранг идеала каждую СТРУКТУРНУЮ штукуну и ИСКУССТВЕННЫЙ интеллектус, которые при ближайшем рассмотрении оказываются мыльными пузырями и скоро лопаются. Я в ярости, и я больше не хочу с этим мириться!

Кто же эти люди, которые продают нам всяческие панацеи? Довольно странная компания, две отдельные группы людей, которые вряд ли понимают, что сидят в одной лодке:

1. *Бизнесмены.* Эти люди хорошо знают свое дело. Умным они продают знания, а простодушным – шарлатанские снадобья. Они превращают хороший товар в отличный независимо от того, заслуживает он этого или нет. Как правило, они достаточно умны, чтобы отличить хорошее от самого лучшего, но они также достаточно умны, чтобы понимать: просто хороший товар не покупают.
2. *Ученые-теоретики вычислительной техники и программирования.* Они знают свое дело ничуть не хуже. Они генерируют новые идеи и априори, без экспериментального подтверждения, считают их великими. Как правило, они искренне верят, что их хорошие идеи просто замечательны. Кроме того, они достаточно умны, чтобы понимать: просто хорошими идеями не добьешься грантов на исследования.

Нужен практический пример? Теоретики программирования утверждают, что до автоматической генерации кода на основе требований рукой подать. Они создают образцы инструментальных средств, чтобы доказать свою правоту и продемонстрировать реальность своих обещаний. При ближайшем рассмотрении можно увидеть, что спецификации требований очень похожи на проект, гранича в своей скупости с программным кодом. Ну, если *так*, то автоматическая генерация кода – из кодоподобных спецификаций – действительно у нас в кармане! Но это мало что нам даст.

В этот самый момент бизнесмены, указывая на открытия теоретиков, объявляют во всеуслышание, что *теперь* они могут предложить генерацию кода на основе спецификаций. Они даже встраивают эту функциональность в CASE-инструменты. Правильные требования. Правильные мотивы. Правильный продукт. Так у кого хватит глупости отказаться от гениального исследования и от гениального маркетинга?

CASE как он есть

Позор. Во всех этих заявлениях, и в самых громких, и в самых скромных, есть правда. И структурное программирование, и искусственный интеллект, и (а куда же без них) CASE-средства – не пустой звук. Беда в том, что мы не знаем, какова их настоящая ценность. Все, о чем мы можем догадываться, опираясь на технологический опыт отрасли, на литературные и экспериментальные данные, – это что каждый из таких «прорывов» способен улучшить нашу способность создавать качественное ПО (в относительных единицах – от 2 до 20%). (Как приблизиться к 20%? Для начала необходимы плохие разработчики ПО! Не то, что вы хотели услышать, правда?)

Исходя из того, что от CASE-инструментов есть польза, посмотрим на них внимательно. При всем обилии продуктов, претендующих на принадлежность к CASE-технологиям, они довольно неоднородны, по крайней мере, пока. Однако если проанализировать декларируемые ими цели, то можно заметить, что у них есть общие черты.

Придет время, когда инструменты CASE будут помогать:

- В системном анализе, принимая форму автоматических организмов требований (например, смогут строить диаграммы потоков данных и проверять их на точность и последовательность).
- В проверке требований и проектов (например, будут генерировать прототипы пользовательских интерфейсов или модели решений на основе элементарных требований).
- В поддержке проектных работ (например, вести словари данных или проверять правильность описания проекта на языке PDL).

- В сопровождении процесса кодирования (например, обеспечивать внутренний интерфейс программного кода с проблемно-ориентированным клиентским интерфейсом).
- В создании общего интерфейса между разработчиками и внутренними компонентами, который объединит все эти инструменты и превратит их в удобный и полезный набор.

Иначе говоря, технология CASE предстанет в образе интегрированного набора инструментов поддержки процесса создания ПО на протяжении всего жизненного цикла разработки. Складно получается?

Если мы оптимистично оценим положительный вклад каждого из этих пяти пунктов в улучшение качества и/или повышение производительности как равный 5%, то получим прирост в 25% при условии, что CASE-инструменты построены и подобраны тщательно. Это тоже звучит очень неплохо.

Но это не прорыв, не спасение от всех напастей и, уж *конечно*, не чудо.

Сравнение инструментов CASE и 4GL: что в сухом остатке?

Каких выгод мы можем ожидать от автоматизации разработки ПО?

Это важный вопрос. Мы тратим очень много денег и энергии, внедряя автоматизацию в индустрию ПО при помощи CASE-инструментов и языков четвертого поколения. Соизмеримы ли выгоды и затраты?

К сожалению, попыток ответить на этот вопрос ничтожно мало. Чтобы получить искомый ответ, необходимо как-то сравнить затраты, связанные с новой и старой технологиями, и, как бы мы это ни делали, получается дорого.

Тем не менее я приведу здесь результаты трех исследований, которые прольют некоторый свет на этот вопрос. Одно исследование основывается на обзоре, другое – на эксперименте и третье – на конкретном примере.

После этого мы еще раз проанализируем ситуацию, ознакомившись с некоторыми мнениями и данными опросов.

В исследованиях изучается ценность языков четвертого поколения и CASE-инструментов. Прежде чем ознакомить вас с результатами, должен сказать, что подобные исследования в нашей еще не сформировавшейся отрасли до сих пор не отличаются точностью. Для получения надежных ответов необходимо провести еще немало подобных исследований.

Высказав это предупреждение, представлю выводы:

1. Один из CASE-инструментов обеспечил уменьшение затрат на 9% и примерно такое же улучшение качества разработки ПО по сравнению с традиционными технологиями.
2. Программный код, сгенерированный при помощи двух языков четвертого поколения, был короче программы на Коболе на 29–39% и примерно в 50 раз медленнее.
3. Применение другого языка четвертого поколения позволило сократить трудозатраты в 4–5 раз по сравнению с предполагаемыми трудозатратами при разработке на Коболе.

Прежде чем мы узнаем, как были получены эти результаты и как их интерпретировать, поговорим немного о них самих.

Прежде всего, едва ли такие показатели позволяют говорить о каких-либо прорывах. Любые улучшения в диапазоне от 10 до 100% очень приятны, и за них можно заплатить, но они, конечно, не «улучшение на порядок», как заявляют некоторые исследователи и бойкие продавцы. Даже улучшение в 4–5 раз, что вовсе не на 29–39% лучше (если считать, что в обоих случаях речь идет о том, насколько быстрее можно написать программу на языке 4GL, чем на Коболе), – это все равно не «на порядок» лучше по сравнению с альтернативой. На сегодняшний день ситуация ясна: технологии автоматизированной разработки ПО заслуживают внимания, но они не являются определяющим фактором.

А теперь ознакомимся с упомянутыми выше исследованиями. В них больше информации, чем в этом коротком обзоре.

Результаты исследования CASE-инструментов

Результаты исследования CASE-инструментов представлены в до сих пор не опубликованной работе под названием «A Survey on Applications of a CASE Environment: Insights Gained» (Обзор применений CASE-средств: попытки осмысления), написанной Рудольфом Лаубером (Rudolf Lauber) из Штутгартского университета и Питером Лемппом (Peter Lempp) из SPS (Software Products and Services). Она посвящена CASE-инструменту EPOS, применяемому в Западной Германии в приложениях реального времени.

На момент написания этой книги EPOS поддерживал клиентские и серверные части приложения (front end, middle end) и управление жизненным циклом ПО, однако в нем не было некоторых важных функций, добавленных впоследствии, таких как графическое представление и автоматическая генерация кода. И все-таки он содержал функ-

ции организации и детализации требований, определения и уточнения концепций проекта, анализа осуществимости проекта, предварительного и подробного представления проекта, документирования и контроля качества.

Исследование достоинств EPOS представлено в форме обзора и в форме опроса. В нем приняли участие 22 менеджера проектов среднего масштаба из 14 компаний. Большинство этих проектов на тот момент еще не перешли из стадии разработки в стадию сопровождения. Обзор был всесторонним, а опросы – продолжительными (обычно они продолжались не менее половины дня).

И что же выяснилось?

1. Наибольшие выгоды от применения EPOS ощущались в области контроля и управления проектом.
2. Обнаружилась тенденция роста расходов на «внешнюю» часть жизненного цикла и их уменьшения на внутреннее обеспечение жизненного цикла (с запланированным значительным снижением расходов на сопровождение и документирование), при этом общая экономия составила 9%.
3. В 69% проектов было зафиксировано уменьшение количества ошибок в процессах предварительной обработки данных.
4. Основная масса проектной документации (75%) была сгенерирована автоматически при помощи инструментальных средств.
5. Способность к творчеству и мотивация технических специалистов не изменились.
6. Реакция на инструментальные средства менялась от энтузиазма вначале до разочарования во время обучения и опять до энтузиазма после некоторого опыта применения.

К каким выводам в целом пришли авторы? «Оценка чистой выгоды была далека от очень высокой: никто не наблюдал снижения расходов на порядок». Инструменты обеспечили «скорее эволюционное снижение расходов, чем революционное».

Эксперимент «Языки 3GL против языков 4GL»

А теперь посмотрим на результаты сравнения языков 3GL и 4GL. Сантош Мишра (Santosh Misra) и Пол Джэликс (Paul Jalics) из Государственного университета Кливленда опубликовали свою статью «Third-Generation vs. Fourth-Generation Software Development» (Разработка ПО: 3GL против 4GL) в июльском номере *IEEE Software* за 1988 год. Они создавали небольшие бизнес-приложения тремя способами: в сре-

де dBase III – этот инструмент они назвали «низкоуровневым 4GL», – на языке PC Focus, определенном ими как «4GL высокого уровня», и при помощи Кобола. СУБД dBase – это инструмент для работы с базами данных, обладающий и процедурными, и непроцедурными возможностями генерирования отчетов; PC Focus – это непроцедурный язык для миникомпьютеров.

Для языков 4GL результаты были неутешительными:

1. Исходные коды на 4GL были всего на 29 –39% короче, чем на Коболе.
2. Это преимущество языков 4GL могло бы быть более ощутимым, если бы в расчет принимались описания данных (для 4GL их либо требуется немного, либо без них можно обойтись), однако большинство разработчиков ПО полагают, что лучше иметь описания данных, чем не иметь.
3. Разработка с помощью dBase III оказалась на 15% быстрее, чем на Коболе, и на 90% медленнее, чем на PC Focus (даже если вычесть время на обучение).
4. Скорость исполнения программ на языках 4GL оказалась в 15–174 раз медленнее, чем программ на Коболе.

Авторы сделали выводы, что «...уменьшение... времени разработки не было существенным», «производительность... была, безусловно, намного хуже, ...чем для Кобола», и «...если задачу не удастся быстро решить непроцедурными средствами, ...то приходится писать программу на каком-то очень странном языке, который часто оказывается более несовершенным, чем языки 3GL». Иными словами, большие преимущества языков 4GL, если они вообще достижимы, проявляются, только если средства языка точно соответствуют приложению. Не очень оптимистичная картина.

Языки 3GL и 4GL – конкретный пример

Последняя группа данных была опубликована в том же журнале (*IEEE Software*, июль 1988 года). Джун Вернер (June Verner) и Грэм Тэйт (Graham Tate) из новозеландского университета Мэсси (Massey University) провели исследование под названием «*Estimating Size and Effort in Fourth-Generation Development*» (Оценка трудозатрат при разработке ПО на языках 4GL). Его основным предметом была полезность современных методов оценки приложений, написанных на языках 4GL. (Она была признана в целом недостаточной.) Однако вместе с тем они обнаружили, что в сфере разработки небольших бизнес-приложений преимущества 4GL намного заметнее.

Имея в виду конкретное приложение, авторы в своем исследовании остановились, после тщательного отбора, на языке ALL (Application Language Liberator от Microdata).

Их выводы были однозначными: трудозатраты, связанные с применением языков 4GL, на 400% меньше, чем при сравнимой разработке на Коболе. Если вынести за скобки анализ выполнимости и формулирование требований (потому что продолжительность этих этапов проекта не должна зависеть от выбора языка), то выигрыш составит 530%.

Эти выводы тем более впечатляют, что перед началом своего исследования авторы выразили скепсис по поводу заявлений производителя об улучшенной производительности этого языка. И хотя Вернер и Тэйт, как и предыдущие исследователи, обнаружили, что с помощью языков 4GL очень трудно решать сложные задачи, их главный вывод состоял в том, что они позволяют добиться значительного повышения производительности (но все-таки не на порядок).

Итак, имеются результаты, говорящие о полезности некоторых конкретных средств автоматизации разработки ПО. Они не слишком эффективны, но и ничтожными их назвать нельзя. Есть ли хоть какая-то надежда, что они нас куда-нибудь приведут?

В статье «Who's Winning the CASE-4GL Race?» (Кто ведет в гонке CASE-4GL?) (*System Builder*, март 1989 года) Джон Кэдор (John Kador) говорит, что существует, вероятно, две разновидности языков 4GL: для информационных центров (для пользователей) и для организаций-разработчиков (для разработчиков ПО). Как считает Стивен Бендер (Steven Bender) из Phoenix Software, который ознакомил Кэдора с этой классификацией, Focus относится к первой категории. Наверное, от пользовательских языков 4GL нельзя ожидать того же, на что способны более мощные 4GL для разработчиков (например Ideal и Cygnet).

В своем ежегодном обзоре «Annual CASE Survey 1988» компании QED Information Sciences и CASE Research Corporation сделали вывод, что, хотя CASE-инструменты и повышают производительность (почти 30% пользователей CASE-инструментов сообщили, что приобретают их именно поэтому), главная их выгода в том, что они повышают качество и слаженность работы. В качестве главного выигрыша от CASE-инструментов почти 25% пользователей упомянули «повышение качества проектирования», а еще 15% – «улучшение взаимодействия между пользователями» и «улучшение взаимодействия между разработчиками».

Другими словами, нам еще много предстоит узнать об автоматизации. Применение CASE-инструментов дает выигрыш, но не революционный, а эволюционный. Вероятно, у пользователей и у разработчиков разные взгляды на приемлемость инструментов. И уж конечно, если говорить о реальном выигрыше, то качество значит больше, чем производительность.

Зачем писать компиляторы?

Какой курс предлагает своим продвинутым студентам почти каждая считающаяся приличной программа обучения программированию и вычислительной технике?

Наверное, на этот вопрос есть несколько ответов, но один из них – это «написание компиляторов».

С этим фактом связано несколько любопытных обстоятельств. Например то, что специалистов по написанию компиляторов нужно не так уж много. Те из нас, кто участвовал в проекте по созданию компилятора, понимают, что им выпало редкое везение. Подавляющее большинство программистов решают прикладные задачи. Например, ведение учета, или управление процессами, или инженерные расчеты. Лишь немногие производители компьютерного оборудования и программистские организации берутся за написание компиляторов.

Даже в микропрограммировании, где практически все держится на Турбо-Сноболе или Компакт-Коболе, количество удачных продуктов весьма невелико. Остальной части несостоявшихся разработчиков компиляторов в конце концов приходится искать какие-то другие источники честного заработка вроде написания программ для бухгалтерии или торговли недвижимостью. Для большинства программистов разработка компиляторов – это чаще всего мечта, а не реальность.

Я полагаю, для включения разработки компиляторов в учебные курсы есть некоторые причины. Одна из них состоит в том, что обитатели университетских кафедр попросту не знакомы с реалиями софтверной жизни и не знают, что создание компиляторов – занятие не совсем обычное. Однако чаще студентов учат писать компиляторы, потому что навыки, которые при этом прививаются, такие как обработка текста и манипуляции с таблицами символов, укрепляют и усиливают умения, необходимые для создания других приложений. Если вы знаете, как разобрать текст программы и сохранить атрибуты данных в таблице символов, то вы уже почти умеете обрабатывать данные на сложных входных языках прикладных программных продук-

тов и манипулировать базами данных. Во всяком случае, так утверждает теория.

Но я собираюсь рассказать не об этом. С разработкой компиляторов связан еще один факт, весьма удивительный в свете того, о чем мы говорили только что.

Разработка компиляторов как род профессиональной деятельности безнадежно убыточна. Количество компаний, занятых разработкой компиляторов и не извлекающих из этого прибыли, поражает. Несмотря на все богатство технического арсенала, которым разработчиков компиляторов вооружают в колледжах, когда дело доходит до его практического применения, оказывается, что чего-то не хватает.

Хотите примеров?

- В 1960-х годах компания Computer Sciences Corporation не имела себе равных в создании компиляторов. Она не только выигрывала выгодные контракты, она еще и разработала два изощренных и важных инструмента, которые помогали в деле: компилятор компиляторов GENESYS и специализированный язык SYMPL (сейчас мы бы назвали его языком четвертого поколения), предназначенный для разработки компиляторов. А сейчас CSC практически ушла из этого бизнеса. Почему? Потому что они, прежде всего, не смогли достаточно разумно распорядиться выгодами своего положения.
- Примерно в то же время еще одним лидером компиляторных технологий была компания System Development. Один из ее сотрудников, Джул Шварц (Jules Schwartz), создал язык JOVIAL (Jules' Own Version of the International Algorithmic Language), который до недавнего времени был стандартным языком программирования военно-воздушного ведомства США. С этой компанией произошло то же самое, что и с CSC. Понеся убытки на достаточном количестве компиляторных контрактов, они решили найти применение своим талантам в какой-нибудь другой сфере.
- В 80-х годах XX века в области разработки компиляторов лидировали такие компании, как SofTech и Intermetrics, работавшие по очень выгодным контрактам, связанным с языком Ада. Однако и у них появились трудности. SofTech, и без того переживавшая тяжелые времена со своим самым последним контрактом на разработку компилятора JOVIAL, попала в еще больший переплет с проектом Ада, и в конце концов подрядная организация забрала у них незаконченный компилятор и отдала другой компании. У Intermetrics были примерно такие же трудности. Даже компании,

обладающие опытом создания компиляторов, не всегда преуспевают в этом сложном бизнесе.

- Возможно, самая грустная из всех этих историй связана с небольшой компанией Digitek, которая специализировалась на компиляторах в 1960-х годах. Digitek высоко подняла тонкое искусство компиляторостроения и, как многие другие компании, в то время занимавшиеся этим, располагала семейством инструментов, предназначенных для автоматизации данного процесса. Уверенная в своих силах после предыдущих успехов в создании компиляторов, Digitek заключила контракт на разработку компилятора PL/1 с крупным производителем компьютеров. И перегнула палку. Был нарушен график работ, бюджет раздулся и лопнул, и в конце концов компания обанкротилась. Это произошло всего лишь через год или два после того, как они занимали лидирующее положение в большой группе компаний, создававших компиляторы. (Положительным следствием этой истории было то, что двое из учредителей Digitek основали новую компанию, Ryan-McFarland, и продолжили – с успехом – заниматься компиляторами.)

Какой из этого следует вывод? Почему написание компиляторов, которое годами преподается такому количеству студентов компьютерных специальностей, так плохо ими усваивается? Моя теория состоит в следующем.

Здесь есть две проблемы.

Первая из них – это некоторая наивность академического сообщества. Создать компилятор, если говорить о профессиональном подходе к этому делу, означает намного больше, чем написать код для разбора выражений и работы с таблицами символов. Есть еще целый набор: последовательности объектного кода, регистровая оптимизация, анализ потоков выполнения и даже обыкновенная диагностика. Все это вместе взятое образует комплекс вопросов такой сложности, какая и не снилась студентам третьего года обучения.

Есть и вторая причина. Она связана с так называемой многослойностью прикладных задач. Чем больше слоев содержит задача, тем быстрее возрастает сложность ее решения. Учебная работа по созданию неоптимизированного компилятора для подмножества Паскаля очень сильно отличается от профессиональной задачи разработки полномасштабного компилятора и библиотек Кобола. Может быть еще уместнее сказать, что профессиональная работа по созданию компилятора Фортрана не дает разработчику подготовки, которая позволила бы ему справиться с компилятором языка Ада. Сложность и размеры компилятора растут экспоненциально вместе со сложностью языка. А с их ростом

труднее становится контролировать график и бюджет проекта. И следовательно, становится труднее создавать компиляторы и зарабатывать при этом деньги. Ну и в результате становится труднее удержаться на плаву в этом бизнесе.

Интерес к написанию компиляторов ограничен по тем же причинам, по которым сомнительна правильность подхода к обучению этой профессии. Есть слишком мало студентов с достаточным стимулом к получению этих знаний. Тем не менее имеются некоторые интересные наблюдения общего характера и в области обучения.

Одно из них состоит в том, что корреляция между тем, чему мы учим студентов, и тем, что им надо знать, в лучшем случае слабая. И это настолько же справедливо в отношении курсов по операционным системам и базам данных.

Другое – что в технологии необходимо уметь контролировать сложность (мы это и так знали), но это так же важно с точки зрения экономики. И эта мысль отрезвляет, поскольку мы входим в другую эпоху, которая поставит нас перед еще более сложными и масштабными задачами.

Нельзя сказать, что программисты не умеют справляться со сложностями. Однако в этой отрасли, где уровень «смертности» компаний вполне можно сравнить с катастрофой, постигшей автомобильную промышленность в 1920–1930-х годах в США, понимание роли, которую играет сложность, как раз и отличает победителей от проигравших.

Мы все, конечно, хотим быть в числе победителей и сейчас, и через десять лет!

ЯЗЫКИ

Языки программирования высокого уровня: насколько высок их уровень?

Программируя на языке высокого уровня, мы оперируем понятиями, которые ближе к прикладным областям, чем к «машинным». (Пожалуйста, не отвлекайтесь. Я знаю, что не сказал ничего нового, но хочу подготовить вас к некоторым свежим мыслям.)

Согласно этому определению язык ассемблера (и никого из нас это не удивляет) *не* является языком высокого уровня, потому что машина исполняет команды почти на таком же языке. Паскаль, Ада и Кобол – это языки высокого уровня, потому что без помощи специального компилятора компьютер не может исполнять программы, написанные на них.

Входной язык прикладной программы, посредством которого пользователь сообщает ей свои цели, – это язык самого высокого уровня.

Таким образом, языки программирования заполняют спектр, простирающийся от машинного кода на самом низком уровне до специализированных прикладных языков на самом высоком.

Мы, разработчики ПО, редко смотрим на языки программирования с этой точки зрения. Они для нас – это инструменты программистов, а данные, которые пользователи вводят в прикладную программу, – это нечто другое. Но это различие сейчас стирается, и так и должно быть. Пользовательские интерфейсы прикладных программ стали сложнее, чем когда бы то ни было, и у некоторых из них даже появились «языки», на которых пользователи могут «программировать». Некоторые языки 4GL проектировались с таким расчетом, чтобы пользователи могли писать на них программы. Значительная доля исследований в области вычислительной техники и серьезные усилия пользователей направлены на то, чтобы сделать пользовательской некоторую часть традиционно программистской сферы. Это означает, что появление языков, программирование на которых не требует владения профессией программиста, неизбежно.

Реакция разработчиков ПО на эту тенденцию может быть двоякой. Можно биться в отчаянии по поводу того, что у нас земля уходит из-под ног, и либо сдаться, либо начать войну, чтобы удержать то, что принадлежало нам «всегда».

Но есть и другой путь. Как технические специалисты, наделенные чувством ответственности, мы способны признать, что пользователи иногда лучше подготовлены к решению своих задач, и мы можем вооружить их еще более мощными средствами. Мы также способны признать, что с технической точки зрения задачу, как правило, лучше решать на языке как можно более высокого уровня, даже если при этом оказывается, что не надо писать новое ПО, а достаточно взять готовое.

Сравнительно недавно последнее утверждение могло показаться пустой болтовней. Однако уже в 1990-х годах, когда развивавшийся поразительными темпами рынок ПО был буквально наводнен компонентами, построенными на основе новых концепций, выбор готовых программных решений превосходил самые смелые ожидания. Задумывались ли вы, например, о том, какие возможности предлагали пользователям компьютеры Макинтош, для которых практически не потребовалось (чтобы не сказать «совсем не потребовалось») писать ПО? Здесь перед нами открывается новый мир.

Посмотрим немного пристальнее на основные принципы, стоящие за многообразием языков, доступных программистам и их партнерам –

пользователям, чтобы понять, какие решения могут принимать современные технологии.

В целом применение языков высокого уровня, если язык подходит для решения конкретной задачи, повышает производительность труда и качество программного кода. Если можно приобрести готовый прикладной продукт (обладающий достаточными показателями качества), то это лучшее решение задачи.

Если нам приходится выбирать язык более низкого уровня, потому что на более высоком подходящего не нашлось, и есть язык четвертого поколения, позволяющий решить задачу, то надо выбрать именно его. Это может означать, что, в зависимости от выбора языка 4GL и типа создаваемого приложения, программировать будет либо программист, либо пользователь. (Языки 4GL, как правило, ориентированы на создание приложений обработки бизнес-данных, особенно приложений, взаимодействующих с базами данных.)

Если этих возможностей мало, то вполне подойдет обычный язык программирования (например, Модула-2, или Фортран, или Кобол, или...).

И наконец, если не годится ни один из них, придется обратиться к языку ассемблера или к машинным инструкциям.

Какие обстоятельства заставляют нас проходить сверху вниз по этой цепочке выбора?

Одно из них – это возможности языка. Языки самого высокого уровня часто ориентированы на узкий класс очень конкретных задач и могут оказаться бесполезными, если задача нестандартная.

Еще одно – это эффективность. Как правило, чем выше уровень абстракции языка, тем больше вычислительных ресурсов компьютера он потребляет, и на самом высоком уровне подобное снижение эффективности может оказаться недопустимым.

Каждый из признаков качества (переносимость, надежность, эффективность, эргономическая проработка, тестируемость, понятность и модифицируемость) должен оцениваться с учетом уровня предлагаемого решения, чтобы более низкая цена оплаченного заранее программного продукта не превысила цену, которую придется со временем заплатить из-за менее качественного решения.

Выгоды от выбора языка как можно более высокого уровня немалые:

- Повышается производительность, причем нередко очень сильно.
- Языки высокого уровня обеспечивают изящество и продуманность решений, снижая вероятность ошибки. Так, программист, пишущий программу на ассемблере, должен оперировать аппаратно-зави-

симыми регистрами и может сделать массу ошибок, связанных с их неправильным распределением. Программируя на языке высокого уровня, мы застрахованы от таких ошибок, потому что не должны работать с регистрами.

- В программном коде высокого уровня меньше строк, поэтому ниже вероятность совершения ошибок.
- Языки высокого уровня помогают писать структурированный программный код. Код высокого уровня даже не обязательно структурировать, потому что он может быть непроцедурным (порядок исполнения кода определяет языковой процессор, а не программист).
- Языки высокого уровня облегчают написание переносимых программ.
- Применение этих языков облегчает сопровождение.
- Применение языков высокого уровня облегчает тестирование программ.

По сути, все вышесказанное означает, что решение задачи на языке высокого уровня позволяет улучшить большинство показателей качества программного продукта. Как было отмечено выше, это не относится к эффективности, и для задач, требующих исключительной скорости исполнения программного кода, приходится искать язык менее высокого уровня, «снижая» последний до тех пор, пока не будет достигнут приемлемый компромисс между искомыми показателями качества.

Производительность и качество нечасто идут рука об руку. Применение языков как можно более высокого уровня предоставляет неплохой шанс соединить их.

Должны ли мы готовиться к доминированию 4GL?

Одни говорят, что языки 4GL сверхъестественны и что они кардинально изменят способность человека создавать ПО. Другие – что их преимущества и выгоды сильно преувеличены и что они позволяют лишь повысить производительность (если подходят для решения текущих задач). И добавляют, что часто как раз для текущих задач они и не годятся.

Если правы первые, то каждый программист должен учиться программировать на языках 4GL. Если же правы вторые, то учиться программировать на этих языках должны только те программисты, которым языки 4GL способны помочь в создании ПО. Правда, в любом из этих случаев вопросы изучения и освоения языков 4GL имеют большое значение.

Я намеренно употребил два отдельных термина – *освоение* и *изучение*. Разница между ними по ходу повествования станет еще более важной. Освоение, в том смысле, в котором я его употребляю, означает познание с немедленным результатом. Под изучением же я понимаю познание, отдача от которого ощущается несколько позже. Компании предлагают курсы освоения конкретных программных продуктов (тренинги), например текстовых процессоров или новых сред программирования. Колледжи и университеты предлагают курсы по изучению (образовательные) общих дисциплин, таких как обработка текстов или языки программирования.

Если изучение языков 4GL действительно важно, то кто должен обеспечить этот образовательный процесс? Почти все согласны, что тренинги (практическое освоение) – это дело компаний. За обучение, по тем же причинам, должно взяться академическое сообщество. Обучают ли инструкторы и преподаватели именно так и именно тому, что действительно оправданно и необходимо? Ответить на этот вопрос не так просто.

Рынок предлагает массу курсов по освоению, обычно ориентированных на конкретные языки 4GL и организуемых производителями этих языков. То есть если говорить о начинающих, то их запросы, безусловно, полностью удовлетворяются.

Но образование – это, как говорится, совсем другая история. В большинстве колледжей и университетов о языках 4GL не рассказывают. Вопрос о том, почему это так, и даже вопрос о том, должны ли высшие учебные заведения заниматься этим, до некоторой степени спорный.

Проблема

Спор достиг своей высшей точки, когда журнал *Communications of the ACM* в марте 1985 года опубликовал интервью с Джеймсом Мартином (James Martin), главным апологетом языков 4GL. Научное сообщество, сказал Мартин, не выполняет свою просветительскую миссию и не раскрывает широким массам волшебных возможностей языков четвертого поколения. Однако несколько ученых-преподавателей, участвовавших в телеконференции с Мартином, решительно возразили ему.

Крайности

Разрешить споры о языках 4GL к всеобщему согласию мешали несколько факторов, из которых не последним по важности было эмоциональное напряжение. Заявления сторонников вполне можно было сравнить с заявлениями торговцев медицинскими препаратами. Противники,

по-видимому, впадшие в противоположную крайность, игнорировали и сами продукты, и их рекламу, своим поведением давая понять, что толку нет ни от того, ни от другого. В этой психологически напряженной обстановке нормой стала защита своих позиций при помощи крайностей, переходящих иногда в странности.

Так, в самом начале телеконференции с участием Мартина один из сторонников языков 4GL сказал, что «в большинстве компаний никто ничего не сделал» при помощи этих языков. Позже, в разгар полемики, тот же самый человек заявил, что языки 4GL «применяются в большинстве современных приложений обработки данных».

Противники языков 4GL ничуть не лучше. В ходе полемики один из представителей научного сообщества сказал, что «в некоторых университетах преподают самую суть» технологий 4GL. Однако он же, когда его спросили, в чем конкретно это выражается, признал, что языки 4GL – это одна из тех тем, которые могут «обсуждаться» на «семинарах для студентов пятого-шестого годов обучения» и едва ли лежат в главном русле компьютерного образования.

Позиции

Обозначить позицию сторонников 4GL по этому вопросу нетрудно: они утверждают, что языки 4GL слишком важны, чтобы их можно было исключить из образовательных программ. Колледжи и университеты игнорируют их, а значит не успевают за развитием технологий. И, поскольку эта технология медленно внедряется в отрасль, вина за то, что ее значение освещено недостаточно, частично ложится на научное сообщество.

Несколько труднее обозначить позиции противников 4GL или тех немногих, кто соблюдает нейтралитет: сначала они говорят, что языки 4GL не настолько важны, как это пытаются представить их защитники. Потом они заявляют, что необходимо соблюдать осторожность, чтобы не превратить образовательный процесс в тренинги, и что университетские курсы по 4GL становятся подозрительно похожими именно на них.

Есть и еще одна причина безразличия научного сообщества, и она совсем не такая философская, как две первые. Технологии 4GL – языковые процессоры и инструменты – очень дороги, и большинству научных и учебных учреждений нелегко найти средства на дорогостоящие расширения образовательных программ.

Однако в вопросе о роли образования в распространении технологий 4GL есть и еще один ключевой фактор, не затрагиваемый всеми этими дискуссиями. Современная система компьютерного образования старательно избегает контакта с сообществом практиков, программирую-

щих приложения обработки данных. А языки 4GL, насколько мы сейчас знаем, имеют максимальную ценность как раз для этого сообщества. «Выпускники лучших учебных заведений стремятся работать не столько в крупных банках и страховых компаниях, сколько в компаниях, занимающихся производством компьютеров», – сказал один из представителей научного сообщества. «Между отраслью обработки данных и факультетами информатики отношения напряженные, и практически самая главная проблема заключается в том, что вычислительная техника как наука испытывает нехватку ресурсов», – заявил другой.

Иными словами, в основе всех трудностей образования, связанных с языками 4GL, лежат две более крупные проблемы, а именно недостаток внимания вычислительной техники как науки к обработке данных и нехватка ресурсов для надлежащего преподавания вычислительной техники и информатики, и не похоже, что положение изменится в лучшую сторону, пока эти две проблемы не будут разрешены.

На пути к решению

Возможно ли подступиться к решению проблем обучения языкам 4GL, невзирая на эту дилемму?

Можно назвать некоторые обстоятельства, способные помочь в этом. В частности, сторонники 4GL должны увидеть эту дилемму и ее блокирующие факторы и начать изолировать те из них, которые относятся к образованию, а не к тренингам. Один из участников телеконференции (сторонник 4GL) назвал темы, которые должны затрагиваться при преподавании языков 4GL:

- Средства автоматической навигации в СУБД
- Способы генерации кода без традиционного программирования
- Непроцедурные конструкции
- Графические инструментальные средства оценки производительности для аналитиков

Эти вопросы, несомненно, выходят за рамки тренингов, и их решение является большим шагом к формированию образовательного фундамента при изучении 4GL. Но эти концепции все-таки нечеткие, и их необходимо облечь в форму ясных тем образовательного процесса.

Представителям научного и преподавательского сообществ, связанным (и тому есть веские причины) стандартными определениями учебных программ, также полезно осознавать, что ограничения этих программ могут помешать им адекватно воспринять действительно новую, многообещающую технологию.

Им и тем, кто составляет учебные планы, очень важно быть готовыми к быстрой смене технологий (точно так же при управлении конфигурациями стараются обеспечить стабильность, затрудняя перемены, и одновременно держат наготове аварийные процедуры, позволяющие преодолеть этот консерватизм).

Заключение

Трудно говорить о том, насколько важны языки 4GL и роль, которую играют в их распространении образование и тренинги. Однако, когда мы задумываемся об этом, перед нами возникают и другие проблемы, решение которых имеет по меньшей мере такое же значение.

Неужели мы собираемся отделить обработку данных и ее запросы от остальных областей вычислительной техники как нечто чуждое и недостойное внимания? Неужели предполагаем, что вычислительная техника и информатика настолько прочно застряли в границах традиционных представлений своих учебных программ, что неспособны воспринять новые и важные идеи, порожденные практическим применением компьютерных технологий?

Положительный ответ на эти вопросы, наверное, непосредственно связан с неспособностью компьютерной науки привлечь источники финансирования, необходимые для того, чтобы удержать ее на уровне достижений компьютерной профессиональной практики.

Сомнения по поводу Кобола

Назовите язык программирования, который нашел самое широкое применение. Что приходит вам в голову?

Согласно последним известным мне данным первое место занимает Кобол, который лидировал с большим отрывом от своих конкурентов. На некотором отдалении от него следуют либо генераторы программных отчетов (RPG), либо Фортран в зависимости от того, на данные каких исследователей вы смотрите. Затем идет Паскаль, а распространность остальных языков довольно быстро стремится к нулю.

А теперь сравните этот список со списком языков, которые у компьютерных исследователей имеют репутацию хороших. Какой бы язык вы ни поставили на первое место, он, я думаю, вряд ли занимает сколько-нибудь заметную позицию в списке популярности. Модула-2? Ада? ЛИСП? Пролог? Не много найдется подтверждений практической значимости этих языков.

Задумавшись немного над этим фактом. И у Кобола, и у RPG, и у Фор-трана, конечно же, есть недостатки. И конечно же, как Модула-2, так

и Паскаль и Ада, сделали серьезные заявки на исправление этих недостатков. Почему же на практике применяют совсем не то, что так хвалят?

Ничего удивительного – людям свойственно защищать то, к чему они привыкли. Старые языки со всеми их изъянами применялись так долго, что люди попросту не могут отказаться от них. Эта мысль во многом верна.

Но есть и еще кое-что. Почему хотя бы некоторые компании не предпочли что-то получше Кобол, который так интенсивно задействуется в бизнес-приложениях, ведь в бизнесе для того, чтобы получить прибыль, надо сокращать расходы везде, где это возможно? Нет, я имею в виду не языки 4GL. Я имею в виду языки 3GL, которые дают те же возможности, что и Кобол. Почему компании, находящиеся на переднем крае технического прогресса, не перешли на более совершенный Кобол?

А теперь я выскажу главную мысль этого очерка. От Кобол по-прежнему не отказываются, потому что на самом деле ему нет адекватной замены.

Ересь! Невежество! Глупость! Я слышу крики читателей. В моем безумном утверждении есть смысл, и я хотел бы его раскрыть.

Прежде всего, рассмотрим этот вопрос с другой точки зрения. Язык программирования – это инструмент, и его существование оправдывается только наличием в нем средств, позволяющих записать решение задачи, которое можно найти, выполнив программу на компьютере.

Этим утверждением я хочу опровергнуть тех, кто считает лучшим языком тот, который обладает самыми новыми возможностями. Если новые возможности не помогают решить конкретную задачу, то я отнес бы их к интересным нововведениям, но не к новым полезным инструментам.

Запомним эту мысль и посмотрим на прикладную область, для которой Кобол был создан. Это область, в которой на первом месте находятся обработка данных, манипуляции с файлами; для нее характерны данные, хранимые в виде записей, перемещение символьных строк фиксированной длины и генерация редактируемых отчетов. Конечно, это слишком сильное упрощение, но так проще подойти к рассмотрению языков программирования как инструментов. Какие еще инструменты были созданы для этой прикладной области?

Если хотите, можете назвать Паскаль, или Модулу-2, или Аду, но если вы не будете себя обманывать, то найдете попытки приспособить эти

языки к разработке бизнес-приложений довольно искусственными и поймете, что должен быть более удобный способ.

Этот более удобный способ и есть Кобол. Это квинтэссенция моей ереси.

Означает ли это, что я считаю Кобол хорошим языком? Боже сохрани.

Программистам нужен более совершенный Кобол. Он нужен им уже больше четверти века. Кобол многословен, он затрудняет структурирование кода и написание модульных программ, он... список можно продолжить.

Но именно Кобол полностью соответствует требованиям своей прикладной области. Он обладает специализированными средствами, которые упрощают решение каждой из основных задач, характеризующих эту область. Никакой другой современный язык не справляется с этими задачами так хорошо.

Почему? Почему этот древний и неуклюжий язык не заменяют каким-то другим, который справится с этой работой лучше? А потому, я думаю, что сообщество исследователей, которые могли бы создать более совершенный язык, с таким высокомерием относятся к бизнес-приложениям вообще и к языку Кобол в частности, что мысль о целесообразности создания улучшенного Кобола просто не приходит им в голову.

Я бы предложил тем компьютерным исследователям, которые ищут новых путей к более производительному и совершенному ПО, обратить на Кобол и его прикладную область пристальное внимание. Я бы также посоветовал им собрать требования, предъявляемые к языку данной прикладной области, и разработать такой язык, который удовлетворял бы этим требованиям, и реализовать при этом новые, более прогрессивные идеи, которые появились после 1950-х годов, когда был создан Кобол. И наконец, тому (или тем), кто сделает это, надо будет опубликовать свои находки, чтобы они могли быть оценены по достоинству рынком новых идей.

В 1970-х некоторые программисты говорили, что Кобол безнадежен и что уже в 1980-х он не найдет себе применения. Сейчас очевидно, что Кобол опроверг предсказания скептиков. Но не потому, что он этого заслуживал. Перефразируя Черчилля, можно сказать, что Кобол – это очень плохой язык, но остальные намного хуже.

Тот, кто не побоится рутины обработки бизнес-данных, *имеет шанс* сделать реальностью ранние предсказания о судьбе Кобола. И в то же время – выполнить интереснейшую исследовательскую работу.

Знаете кого-нибудь, кто хотел бы попробовать?

Ретроспектива

Мой интерес к инструментам и техническим приемам, применяемым в ремесле разработки программ, которые находятся в центре внимания этой главы, обусловлен моей склонностью к чисто техническим аспектам данной профессии. В контексте книги, посвященной «конфликту», я называю их «вооружениями».

Возможно, именно в этой главе возраст материала является фактором наибольшего риска. В конце концов, инструменты и технические приемы XXI века должны разительно отличаться от своих теперь уже древних аналогов из прошлого столетия, а на самом деле и превосходить их.

До некоторой степени эта мысль подтверждается предыдущими очерками. Концептуально CASE-инструменты – это уже вчерашний день, хотя мы до сих пор признаем значимость их специфических возможностей. Языки четвертого поколения также широко применяются сейчас и уже стали стандартным инструментом в арсенале программиста. Такие языки, как Фортран и Кобол, тоже все еще в ходу, но главные конкуренты теперь уже не Паскаль и Ада, а С и Java.

Однако меня больше интересует, насколько современным будет то, что останется, если абстрагироваться от этих конкретных моментов. Материал очерка «Повторное использование: готовые части ПО – ностальгия и дежавю» находит отражение как в современных тенденциях повторного использования кода и применения методологии COTS¹, так и в совместном использовании кода, характерном для сообщества Open Source и для методологии шаблонов проектирования.

Что касается очерка «Автоматическое программирование – слухи с вечеринки?», то мысль о несерьезности надежд на автоматическое генерирование кода кому-то может не показаться новой, однако вы бы удивились, узнав, как часто вновь и вновь предпринимаются попытки автоматизировать программирование.

Основная мысль очерка «Джентльменский набор разработчика» уходит так далеко в историю разработки ПО, что мне, пожалуй, даже неловко вспоминать здесь о ней. Однако посмотрите вокруг: до сих пор никто не знает, что такое стандартный набор инструментов программиста и где его взять.

В очерке «Сравнение инструментов CASE и 4GL: что в сухом остатке» говорится, что у нас никогда не было методики, которая позволила бы определить ценность CASE-инструментов и языков 4GL. Эту мысль

¹ COTS (Components Off The Shelf) – согласно этой методологии ПО следует создавать из готовых компонентов сторонних разработчиков. – *Прим. перев.*

я развивал, не останавливаясь, и написал более полную статью («The Realities of Software Technology Payoffs» (Реальные выгоды технологий создания ПО), опубликованную в *Communications of the ACM* в феврале 1999 года). В ней я пришел к выводу, что ценность почти любой технологии (или инструмента) – а не только CASE или языков четвертого поколения – нельзя определить, глядя на результаты исследований.

Поэтому сегодня я по-прежнему горжусь этими своими мыслями, которые приходили мне в голову много лет назад. Надеюсь, что вы найдете их такими же полезными, если отбросите второстепенные, неуместные сейчас соображения.

Глава 4 | Из штаба

МЕНЕДЖМЕНТ

- Покорение вершин индустрии ПО
- Новый взгляд на продуктивность ПО
- Производительность и Теория G
- Управление программными проектами – Теория W, принадлежащая Барри Боэму
- Повышение производительности труда в индустрии ПО: кто чем занимается?
- Метрики ПО: о громоздках и накопленной напряженности
- Как измерить качество: меньшее притворяется бóльшим
- Можно ли внедрить качество в программный проект
- Легенда о плохом программном проекте

МАРКЕТИНГ

- А вы купили бы у короля Людовика автомобиль с пробегом?

КОНСАЛТИНГ

- Подготовная консалтинга
- Какие прогнозы давали предсказатели раньше
- Поддержка пользователей: все не так просто, как кажется

Ретроспектива

МЕНЕДЖМЕНТ

Покорение вершин индустрии ПО

Величие – штука эфемерная, ускользающая и труднодостижимая.

Отчасти потому, что немногие обладают качествами, необходимыми для его достижения. Но также и потому, что для достижения величия надо не только обладать определенными качествами, но и соответствовать духу времени.

Возьмем, к примеру, индустрию ПО. На заре развития отрасли для покорения ее вершин требовались совсем другие личные качества, чем нужны теперь. Я попробую проиллюстрировать это утверждение тремя частично придуманными историями о трех великих представителях индустрии программирования, каждый из которых отличался от других и был порождением своего времени.

Давным-давно, в 1958 году, каждому программисту было известно имя Вуди Программиста (Woody Codalot). Каким путем Вуди пришел к славе? Он занимался созданием лучшей из известных человечеству разновидностей ПО общего назначения – сейчас мы назвали бы это программными компонентами. Это такие программы, которые встраивались другими программистами в свой код, чтобы «не изобретать велосипед».

Вуди написал одну из самых первых операционных систем, по современным меркам примитивный, но элегантный (для того времени) пакет ПО, который освободил программистов от необходимости погружаться в дебри тонкостей работы компьютерного «железа», понимание которых требовалось от тех, кто разрабатывал операционные системы. Еще Вуди написал массу вспомогательных программ – генераторов случайных чисел, тригонометрических функций и других хорошо известных аналогичных программ, до сих пор имеющих ценность.

Разбогател ли Вуди на этих программах? Нет. Может быть, вам трудно в это поверить, но в те далекие дни программное обеспечение было бесплатным! Никому и в голову не приходило, что программы пригодны на какую-то другую роль, кроме роли приложения к компьютеру, которое нужно было только для того, чтобы продать компьютер. Вуди достиг величия, жертвуя свои программы в пользу сообщества Share, группы пользователей IBM, чья миссия заключалась в том, чтобы сделать программное обеспечение, написанное в одной компании, доступным для других. (Те, кому довелось бывать на последних встречах Share, могли заметить, как сильно изменилась эта организация!) Вуди стал великим, потому что (1) его именем были подписаны многие программы, перечисленные в руководстве Share (сейчас его называли бы ка-

талоном программных компонентов), и (2) все знали: если программу написал Вуди, то это хорошая программа. Тот, кто не знал, кто такой Вуди Программист, не мог назвать себя программистом.

Через пятнадцать лет, в 1973 году, Вуди Программиста никто не помнил. В индустрии ПО действовали другие механизмы достижения известности и появились новые имена. Одним из самых известных было имя Вольфганга Писателя (Wolfgang Writalot). Как добился славы Писатель? Проводя исследования и публикуя их результаты. Что же это были за исследования и в каких работах публиковались результаты? Наверное, вы помните первые годы революции, которую произвело структурное программирование. Вольфганг был одним из тех, кто в результате своих исследований стал приверженцем продуманной структуры программ и эвентуального изгнания из них операторов GOTO. Результаты его работ были опубликованы в журнале Ассоциации профессиональных программистов *Transactions and Communications* и со временем были воспроизведены в нескольких книгах по структурному программированию. Кроме того, он часто читал лекции на эту тему и много выступал с докладами на компьютерных конференциях (и даже как основной докладчик). Если вы хоть что-то знали о структурном программировании, то вы слышали о Вольфганге.

Вуди Программист, как мы видели, был действующим профессиональным программистом. Работал в компании Wings Aloft Aviation. Вольфганг Писатель, напротив, был ученым и занимал пост профессора в одном из ведущих высших учебных заведений, где обучали информатике и вычислительной технике, в Колледже-под-Плющом (Ivyclad College). За десять лет, которые их отделяли, способ достижения известности разительно изменился: надо было уже не программировать, а думать о том, как это делать.

Эти две истории плавно подводят нас к третьему великому персонажу, Гейтсу Продавцу (Gates Sellalot). В 1984 году в индустрии программирования закончился великий сдвиг, начавшийся на много лет раньше. Помните, что во времена Вуди Программиста программы были бесплатными? Так вот, в эпоху Гейтса Продавца об этом времени и думать забыли. Мало того, что программы перестали быть бесплатными, – выяснилось, что на них можно делать большие деньги. Гейтс Продавец оказался тут как тут, и на руках у него были лучшие программы.

За несколько лет до этого, в дни его безоблачной хакерской юности, Гейтса посетила новая идея, относившаяся к программным генераторам отчетов и впоследствии получившая известность в образе электронных таблиц. Он создал свою электронную таблицу, улучшил ее, а потом улучшил еще. Однако через некоторое время технические усовершенствования ему наскучили, и он решил попробовать себя в роли про-

давца программ – посмотреть, сколько человек захотят купить его программу. Результат, как сейчас всем известно, был потрясающим. Он измерялся в сотнях тысяч, если не в миллионах. Внезапно программирование и ПО превратились в большой бизнес. Гейтс Продавец, очень похожий на Горацио Элджера (Horatio Alger), появился на обложке глянцевого журнала, где, кроме того, объявлялось, что человеком года была признана компьютерная программа – точно так же, как компьютер за два года до этого.

Гейтс стал миллионером. Он стал президентом собственной компании, признанным успешным менеджером и техническим специалистом, а его имя прозвучало по всему миру. Вот *это* успех!

Задумаемся над этими историями еще раз. Вуди Программист был известен, потому что в совершенстве владел технологией. Вольфганг Писатель стал знаменит, потому что был блестящим теоретиком. А фундаментом известности и славы Гейтса Продавца была гениальность маркетинголога. Идею необходимо сгенерировать, подвергнуть тщательному исследованию и наконец продать. Наверное, эти три человека имели не слишком много общего, и может быть, ход этих событий никак не был связан с программами и программированием, и возможно, все произошедшее составляет суть капиталистической системы.

Я хочу сказать об известности и величии кое-что еще. Вспомните известных великих людей. Возьмите наудачу любое имя. Авраам Линкольн, например. Что сделало его великим?

Он сохранил Соединенные Штаты Америки и освободил рабов. Но как он это сделал? Вернитесь мысленно в начало 1860-х и представьте, как вы пытаетесь принять решение, которое должен был принять Авраам Линкольн. Объявите ли вы войну, чтобы сохранить США, или позвольте им распасться? (Помните, что об освобождении рабов задумались чуть позже.)

Подозреваю, что, учитывая антивоенный дух нашего времени, принятие этого решения будет для вас делом нелегким, если, конечно, вы не станете себя обманывать. Задача не станет намного легче, если вы посмотрите на размеры потерь, понесенных во время Гражданской войны. За сохранение США пришлось заплатить чудовищную цену человеческими жизнями. Выбор, который сделал Линкольн, был трудным и, может быть, даже потребовал от него качеств деспота.

Однако я думаю, что не это решение сделало Линкольна великим. Он стал великим потому, что, приняв его, он сделал все, чтобы оно было правильным в глазах истории. Для этого он приложил все свои силы и энергию к тому, чтобы победить в войне. Был бы Линкольн великим в наших глазах, если бы Север был разбит? Нет, я думаю, что история

низвергла бы его, как Джефферсона Дэвиса, и он вошел бы в ее анналы как неудачник, может быть, даже достойный презрения.

Но это было философское отступление. Вернемся к нашему предмету – величию в мире ПО и программирования.

Хотите принять участие в игре? Подумайте.

Что потребуется для достижения величия в индустрии ПО в следующем десятилетии? Гениальность в технологии? В теории? В маркетинге? Или в чем-то совершенно новом? Если вы достаточно хорошо сыграете в эту игру, то сможете стать следующим великим человеком в сфере ПО и программирования!

Новый взгляд на продуктивность ПО

Как добиться существенного (на порядок) увеличения продуктивности ПО?

Именно этот вопрос интересовал несколько лет назад Министерство обороны. Если уж на то пошло, этот вопрос уже многие годы интересует всех профессиональных программистов.

Люди, отвечающие на этот вопрос, делятся на две категории. К первой принадлежат люди, уверенные, что знают, как сделать ПО в 10–20 раз эффективнее, и предлагают добиться этого, выделив много денег на научные исследования и разработав революционные методологии. Они говорят, что им требуется столько-то тысяч долларов, чтобы показать всем, как каракули, положенные рукой мастера на салфетку, превращаются в автоматическую генерацию программного кода.

Ко второй принадлежат те, кто думает, что никто не знает, как улучшить ПО в 10–100 раз. Они заламывают руки и говорят, что бессмысленно тратить деньги на эту неразрешимую задачу.

И, как часто бывает, когда друг другу противостоят две крайности, никто и не думает занять промежуточную позицию. Можно ли добиться заметного улучшения продуктивности ПО, объединив известные технологии, методики и приемы каким-то новым способом? На этот вопрос пытаются ответить лишь немногие (если не считать тех, кто отвечает «да» чисто умозрительно).

Конечно, всегда найдутся торговцы методологиями, которые пообещают, что как раз то, что они сейчас продают (обычно это какая-нибудь методика, в названии которой есть слово «структурная»), позволит добиться именно таких улучшений, какие нужны. Разумные разговоры с ними вести трудно: они слишком заняты продажами, чтобы тратить время на доказательства в поддержку своих обещаний.

Известны, однако, некоторые тревожные данные по этому вопросу. Их графическое представление можно увидеть на обложке книги Барри Боэма (Barry Boehm) «Software Engineering Economics»¹. Данные Боэма показывают, что вклад методологий, языков и других технологических подходов в повышение эффективности ПО не превышает уровня «шума».

Несколько лет назад в одном из профессиональных журналов появилась статья с анализом данных эксперимента по изучению положительного эффекта улучшения методологий, таких как структурное программирование. Анализ завершался выводом, что ни один из экспериментов не дал сколько-нибудь вразумительного ответа о наличии означенного эффекта. Это был просто холодный душ!

Несколько раньше состоялась компьютерная конференция, на которой сообщалось об эксперименте в некоей узкой области, посвященном преимуществам структурированного кода перед неструктурированным. Согласно данным этого эксперимента преимущества были очевидны (что было хорошо), но незначительны (и это было плохо) – не больше 5%.

Получалось, что для повышения эффективности ПО на порядок необходим целый легион технологий. Трезвый взгляд на эти данные заставлял понять, что таким способом желаемого эффекта не получить.

Однако посмотрим на обложку книги Боэма еще раз. Один из факторов, влиявших на эффективность, разительно отличался от остальных. Речь идет о *качестве программистов*, выполнявших работу. Возможно, что ответ на вопрос Министерства обороны лежит именно здесь, в человеческом факторе технологии производства ПО.

А во сколько раз хорошие программисты лучше, чем все остальные, – на порядок? За последние 20 лет получены эмпирические и экспериментальные данные, подтверждающие эту мысль. Они убедительно показывают, что хорошие программисты справляются с различными задачами лучше, чем остальные, превосходя их *в 7–28 раз*.

Какую практическую выгоду можно извлечь из того, что программисты – это один из ключевых факторов эффективности ПО? Нельзя ли, например, выяснить, как именно они работают? И нельзя ли, выяснив это, передать секрет этой технологии другим?

Некоторые исследователи занимаются этими вопросами сейчас и занимались ими раньше. Они называют эту область «эмпирическими исследованиями программирования».

¹ B. W. Boehm. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall PTR, 1981 (Б. У. Боэм «Инженерное проектирование программного обеспечения». – Пер. с англ. – М.: Радио и связь, 1985).

Главными мотиваторами этого движения выступают такие исследователи, как Эллиотт Соловей (Elliot Soloway) из Мичиганского университета, Билл Кертис (Bill Curtis) из MCC (Microelectronics and Computing Consortium) и Бен Шнейдерман (Ben Shneiderman) из Мэрилендского университета. Эти первопроходцы весьма оптимистичны, начали появляться и некоторые надежные и полезные результаты.

Какие результаты? Проанализируем три работы, опубликованные в разных изданиях, в которых сообщалось об экспериментах в этом направлении. Одна из них, «Problem Solving for Effective Systems Analysis: An Experimental Exploration» (Решение задач эффективного анализа систем: Экспериментальное исследование), написанная Виталари и Диксоном (Vitalari, and Dickson), появилась в *Communications of the ACM* в ноябре 1983 года и была посвящена сравнению качества генерации требований «хорошими» и «менее хорошими» профессиональными системными аналитиками.

В другой работе, «You Can Observe a Lot Just Watching How Designers Design» (Можно многое узнать, наблюдая за тем, как работают проектировщики), которую написали Литтман, Эрлих, Соловей и Блэк (Littman, Ehrlich, Soloway, and Black) и которая была напечатана в *Proceedings of the Eighth Annual NASA Goddard Software Workshop* в ноябре 1983 года, сравнивалась работа «опытных» и «неопытных» проектировщиков.

В третьей, которую написал Соловей и которая называлась «Why Software Maintenance is Hard» (О причинах трудности сопровождения ПО), была опубликована в разных изданиях, в том числе в *Systems Development* в августе 1985 года, и в ней сравнивались «эксперты» и «начинающие» специалисты по сопровождению ПО.

Удивительно, но в этих работах были общие мысли. Итак, внимание. Хорошие/опытные/искусные специалисты делали кое-что такое, чего не делали их менее одаренные коллеги. Что же это было?

1. Они ставили больше конечных и промежуточных целей и двигались к этим целям последовательно, сверху вниз.
2. Они искали аналогии с задачами, решенными ранее, и строили модели на основе этих аналогий.
3. Они были более методичны. Они фиксировали в словах и на бумаге больше стратегий, записывая допущения, ограничения и предположения. Их описания конечного продукта были менее расплывчатыми (системные аналитики указывали больше требований).
4. Они проверяли и отклоняли больше гипотез. Они чаще меняли стратегии.

5. В некотором смысле их работа была больше ориентирована на людей. Системные аналитики лучше продумывали пользовательский интерфейс. Специалисты по сопровождению анализировали стиль программного кода, оценивая его по «человеческим» меркам.

Очень интересный список. А если учесть, что в сравнении участвовали такие разные дисциплины, как анализ, проектирование и сопровождение, то он еще и довольно удивительный. Ну и что?

Что если мы сможем превратить это в технологию? Что если мы сможем научить программистов ставить цели и работать с аналогиями? Что если мы вдохнем в разработчиков ПО желание проверять и отбрасывать гипотезы? (Мы должны понять, что достижение блестящего совершенства – скорее не мгновенное событие, а постепенный процесс, сопровождаемый неудачами.) Что, если мы проанализируем другие ответы в этом коротком списке, а также данные, которые уже предоставлены нам другими эмпирическими исследованиями?

Сможем ли мы тогда на порядок повысить эффективность ПО?

Имеет смысл попробовать.

Производительность и Теория G

Все вокруг говорят о производительности. Почему бы и мне не заняться тем же самым?

В конце концов, это была животрепещущая тема 1980-х. Когда с календарей исчез 1979 год, произошло нечто мистическое, чего я до сих пор не понимаю. По всей стране тут же заговорили: «Производительность – вот в чем все дело».

Неужели это было напечатано в календарях? А к 1984 году мы, сами этого не заметив, были в массовом порядке загипнотизированы? Как иначе объяснить, что все в один голос практически одновременно заговорили об одном и том же, откуда взялись все эти экспертные мнения? Единственное объяснение, которое приходит мне в голову, – это что мы все проходим через некую информационную систему. Может быть, вы еще не осознали этот феномен. Вот что я имею в виду.

Некий информационный поток существует на уровне третьего класса, и информация передается от прошлогоднего третьеклассника к третьекласснику этого года, причем ни учителя, ни родители в этом не участвуют. Например, каждый третьеклассник знает, что страна, в которой нет никаких машин, кроме розовых, – розовая. Мы сказали им это? Может быть, учителя? Нет, конечно. Подобную чушь может сказать только ребенок.

Возьмем другой пример. Те из нас, чей возраст приближается к среднему (кстати, никто никогда не достигает среднего возраста, мы только приближаемся к нему), помнят старую считалку «Catch a <обидное слово> by his toe» (Поймай ... за носок). Можете ли вы поверить, что современные дети не знают этой считалки? В той, которую они знают, <обидное слово> заменено словом «тигр». Как это произошло? Как исчез обидный вариант? Сработал тот же механизм обмена информацией, который действовал на уровне третьеклассников. Я его не понимаю, но он действует. Он действует и на уровне других возрастов. И на уровне профессионалов, наверное, тоже (возвращаемся к нашей теме). Он действует как канал связи, по которому нам без конца повторяют слово «производительность».

Как бы то ни было, сейчас уже 1990-е, и я чувствую себя вынужденным говорить о производительности. Тема разговора «Производительность – Теория G». Теория G – это моя собственная теория производительности. (Интересное совпадение: с буквы G начинается моя фамилия!). Теория G, если говорить коротко, состоит из следующих компонентов:

Подтеория G11: Система, построенная на враждебных отношениях, редко предполагает возможность существенных компромиссов.

Подтеория G12: Американская система наемного труда построена на враждебных отношениях между работниками и менеджерами.

Подтеория G13: Значительное повышение производительности требует существенных компромиссов.

Из них следуют:

Теория G1: В американской системе наемного труда, в ее теперешнем состоянии, значительное повышение производительности практически невозможно.

Доказательство теории G1: Посмотрите на британскую систему. Работники и менеджмент находятся в патовой ситуации. Производительность в британской системе стагнирует. Что и требовалось доказать.

Подтеория G21: Пока враждующие стороны в системе работники–менеджмент не определены четко, повышение производительности еще возможно.

Подтеория G22: Компьютерная отрасль еще слишком молода, чтобы в ней успели полностью сформироваться враждебные отношения между работниками и менеджерами.

Теория G2: Повышение производительности, связанное с развитием и, может быть, с применением компьютеров, еще возможно.

Доказательство теории G2: Я не член профсоюза. Возможно, вы тоже не член профсоюза. Мои менеджеры до сих пор иногда ко мне прислушиваются. Может быть, ваши тоже к вам прислушиваются. Там, где мы работаем, сотрудничество между работниками и менеджерами встречается чаще, чем конфронтация.

Подтеория G31: Важным фактором повышения производительности является мотивация.

Подтеория G32: Компьютерный менеджмент чаще прибегает к контролю, чем к мотивации.

Теория G3: Путь, выбранный компьютерным менеджментом, ведет к уничтожению тех шансов на повышение производительности, которые пока еще не исчезли.

Доказательство теории G3: Вам так же нравится ходить на работу, как и пять лет назад? Подозреваю, что нет.

Общая теория G: Может быть, американской системе уже слишком поздно пытаться достичь существенного повышения производительности. Компьютерная отрасль все еще не теряет надежды. Компьютерный менеджмент может убить эту надежду.

Ну вот, это была моя Теория G. Прошу простить мне излишнюю серьезность. Внутренний голос, побудивший меня говорить о производительности, сообщил также, что легкомыслие тут неуместно. Но я, по крайней мере, подсластил пилюлю.

Управление программными проектами – Теория W, принадлежащая Барри Боэму

Что могут сделать менеджеры программных проектов, чтобы повысить производительность?

Есть масса ответов на этот вопрос. Однако доктор Барри У. Боэм из компании TRW нашел еще один. Он называет его «Управление программными проектами – Теория W».

Почему «Теория W»? Известны Теория X, Теория Y и Теория Z (см. далее). Так почему не дать этой теории имя W (в соответствии с алфавитным порядком)?!

Что означает это W? W – первая буква слова Winner (победитель), то есть эта теория сделает каждого, кто ей следует, победителем. Иначе говоря, вы должны сделать все возможное, чтобы реализация программного проекта обернулась выигрышем для всех его участников (менеджеров, заказчиков, пользователей, разработчиков и специалистов по сопрово-

ждению). Под выигрышем Боэм понимает достижение каждым из участников большинства прямых и косвенных целей, связанных с проектом.

Вот какие шаги Боэм считает способствующими реализации Теории W:

1. Определить набор беспроигрышных предпосылок. Понять, как каждый участник проекта формулирует условия выигрыша. Сформулировать цели, которые включают обеспечение возможности выигрыша для каждого участника. Создать среду, способствующую принятию беспроигрышного решения всеми участниками.
2. Структурировать беспроигрышный процесс реализации программного проекта. Составить реалистичный план и обозначить в нем ситуации, в которых выигрывают не все и в которых никто не выигрывает, как рискованные. Обеспечить обратную связь, которая позволит сохранить заинтересованность участников в продолжающихся переговорах и достижении компромиссов.
3. Определить структуру беспроигрышного программного продукта. Определить конечный программный продукт, удовлетворяющий всем условиям выигрыша, особенно условиям пользователей и специалистов по сопровождению.

Типичные условия выигрыша для участников программного проекта:

- Менеджеры хотят, чтобы продукт был создан без нарушений графика и без неожиданностей.
- Заказчикам надо, чтобы проект был реализован как можно быстрее и не вышел из бюджета.
- Пользователи хотят, чтобы продукт имел как можно больше функций, был быстрым и надежным, обладал дружественным интерфейсом.
- Разработчикам проект интересен с точки зрения карьерного и профессионального роста, они стремятся свести к минимуму написание документации и получить цельный продукт.
- Специалисты по сопровождению хотят получить продукт без дефектов, хорошо документированный и легко модифицируемый.

Достижение этих иногда противоречащих друг другу целей не всегда оказывается простым делом. Кроме того, у каждого из участников могут быть собственные цели, которые помогают понять, что означает «выигрыш» для них. При выработке беспроигрышного решения, как указывает Боэм, необходимо учитывать и функциональные (см. предшествующий список), и личностные факторы.

Противоположность беспроигрышным продуктам, согласно Боэму, представляют собой: а) быстрые, дешевые и сырые продукты, которые

дают выигрыш разработчику и заказчику, но означают проигрыш для пользователей; б) продукты, увешанные «фенечками» и «рюшечками», дающие выигрыш разработчикам и пользователям, но проигрыш – заказчику; и в) продукты, являющиеся результатом длительных мучений разработчиков и невыгодные им, но дающие выигрыш заказчику и пользователям. Боэм указывает, что в этих ситуациях ни одна из участвующих в проекте сторон не получает настоящего выигрыша.

Применить Теорию W на практике с учетом разнообразия и противоречивости целей труднее, чем кажется на первый взгляд. Боэм указывает, что специалисты по обработке данных, чьи социальные запросы намного ниже, чем их ожидания профессионального роста, по сравнению с большинством остальных специалистов испытывают трудности с поиском решений в терминах человеческого фактора. Эту проблему Боэм предлагает разрешить с помощью метода, подразумевающего выполнение четырех шагов:

1. Отделить людей от задачи.
2. Сосредоточиться на интересах, а не на положениях.
3. Сгенерировать взаимовыгодные варианты.
4. Следить, чтобы для анализа результатов выбирались объективные критерии.

Боэм утверждает, что теория управления программным проектом, претендующая на эффективность, должна удовлетворять нескольким критериям. Она должна быть простой, универсальной и конкретной.

Простота означает, что теория может быть сформулирована кратко и при этом из главной концепции должны без труда выводиться конкретные практические указания. Теория W, утверждает Боэм, именно такова, поскольку ее можно выразить при помощи четырех слов и трех шагов.

Универсальность означает, что теория должна быть применимой ко всем классам продуктов. Теория W не привязана ни к одной прикладной области.

Конкретность означает, что теория может быть применена непосредственно к конкретному проекту в соответствии с ясно очерченным подходом и критериями проверки результатов. Боэм применил свою теорию в нескольких проектах.

Любая теория управления программными проектами также должна предоставлять основу для системы управленческой деятельности. Боэм упоминает управленческую систему Кунца-О’Доннелла (Koontz-

О'Donnell), в которой реализовано пять процессов: планирование, организация, кадровое обеспечение, отдавание распоряжений и контроль. Боэм отмечает, что теория W может быть полезной в каждой из этих областей.

Другие теории управления проектами и их применение в сфере ПО

Какие еще теории управления получили широкое распространение? Доктор Барри Боэм выделил три: Теорию X, Теорию Y и Теорию Z.

Теория X:

- Люди не любят работать – это их органическое свойство.
- Их приходится принуждать к работе.
- Люди предпочитают, чтобы им говорили, что надо делать.

Теория Y:

- Нелюбовь к работе не является органическим свойством человека.
- Люди способны к самоуправлению.
- Приверженность поставленным целям зависит от итогового вознаграждения.
- Люди способны учиться стремлению к ответственности.
- Творческий подход к работе – это не исключение, а норма.
- Люди реализуют свой потенциал лишь частично.

Теория Z:

- Лучше всего люди работают, когда стремятся достичь целей, в постановке которых они принимали непосредственное участие.
- Если работник принимает цели как свои собственные, то ему можно доверить их реализацию.
- Если люди разделяют общую систему ценностей, то они способны поставить достижимые цели проекта.

Каждую из этих трех теорий, согласно Боэму, не так-то просто применить к управлению программными проектами. Теория X основывается на допущении, что людей необходимо понуждать, чтобы они работали. А это способствует формированию антагонистических отношений между разработчиками и менеджерами, и теория на самом деле не соответствует природе людей, занятых в программных проектах. Теория Y изначально предполагает, что люди хорошо работают, если по-

лучают достойное вознаграждение; это ведет к формированию эгоистических тенденций, и вся система рухнет при возникновении конфликта. Теория Z, известная еще как «японский стиль управления», хорошо себя зарекомендовала в однородной среде сотрудников, но перестает работать, когда над выполнением проекта работают несколько организаций и разные типы работников, менеджеры программных проектов, заказчики, пользователи, разработчики и специалисты по сопровождению ПО. Интересно, что Боэм также указывает на некоторые трудности в применении к программным проектам другой известной максимы, золотого правила: «Поступай с другими так же, как ты хочешь, чтобы другие поступали с тобой». Боэм отмечает, что в индустрии ПО ее дополнили: «Поступай с другими так же, как ты хочешь, чтобы другие поступали с тобой, – если ты похож на них». Однако у тех, кто работает в сфере ПО не такие же цели, как, например, у заказчиков и пользователей, и золотое правило в своей изначальной формулировке не действует.

Повышение производительности труда в индустрии ПО: кто чем занимается?

Представьте, что у вас есть миллион долларов и жгучее желание заняться проблемой эффективности отрасли ПО. Что бы вы сделали?

Оказывается, на этот вопрос уже есть три ответа. За последнее десятилетие были созданы три организации, призванные попытаться улучшить положение дел в индустрии ПО.

Первой, в 1983 году, при участии 12 компьютерных и IT-компаний в Остине (штат Техас) была основана Корпорация микроэлектроники и компьютерных технологий (Microelectronics and Computer Technology Consortium, MCC). Затем, в 1984 году, в Питтсбурге Министерство обороны создало Институт программной инженерии (Software Engineering Institute, SEI). Последним, в 1985 году, при участии 14 авиакосмических компаний, базировавшихся в Рестоне (штат Вирджиния) был создан Консорциум продуктивности ПО (SPC – Software Productivity Consortium), задачей которого было изучение средств, повышающих эффективность труда программистов.

Каждая из этих организаций была создана для того, чтобы улучшить состояние компьютерных технологий. И каждая делает это по-своему!

Самая большая из трех, корпорация MCC, также является самой многогранной. В ней работают примерно 400 человек, из которых менее 100 заняты исследованием программных технологий. Какова цель MCC? Проводить исследования, направленные на создание новых технологий, но не создавать программные продукты.

Институт SEI, появившийся вторым, занимает второе место и по размерам. В нем работают около 150 человек, и все они занимаются программным обеспечением, но в разных областях. Например, одно из самых первых успешных подразделений SEI разработало материалы программы по программной инженерии, пригодные для изучения этой дисциплины и в высших учебных заведениях, и в компаниях отрасли. Остальные подразделения разрабатывают стандарты качества, выступая в качестве информационных технологических центров, создавая образцовые примеры рабочей среды, анализируя и улучшая процесс разработки ПО.

Консорциум SPC избрал третье направление. В нем работает немногим более 100 человек, занятых исключительно разработкой программных пакетов и готовых продуктов для компаний-заказчиков.

Итак, есть три группы: одна занимается исследованиями и не имеет отношения к продуктам, вторая – развитием системы образования и распространением знаний, а третья разрабатывает ПО и может не иметь никакого отношения к исследованиям! Три совершенно разных направления!

Две из этих трех организаций столкнулись с интересной проблемой. Отношение к подобным организациям, созданным несколькими компаниями, раньше было настороженным, так как их подозревали в стремлении к монополизму. И для того чтобы расчистить дорогу новому подходу, было принято новое антимонопольное законодательство.

Как функционируют эти консорциумы? Каждая из компаний, входящих как в MCC, так и в SPC, имеет представителей в советах директоров, и кроме того, каждая предоставляет в распоряжение консорциума технических специалистов.

Институт SEI, созданный правительством, не испытывал подобных трудностей. Однако в смысле обеспечения кадрами SEI не сильно отличается от двух других организаций, потому что дополняет свой постоянный штат специалистами из академических учреждений и компаний отрасли, которые вместе с Министерством обороны приняли участие в деятельности SEI.

А теперь скажите: подумали вы хоть об одной из этих организаций как об объекте, в который можно инвестировать несколько миллионов долларов, предназначавшихся вами на решение вашей задачи? Если нет, то надо, наверное, обратить внимание на еще одну возможную модель.

Она пока находится в стадии обдумывания, поэтому еще не имеет названия и не связана ни с каким конкретным местом. Но она реализует

совершенно иной подход, потому что имеет чисто академический фундамент.

Обычно академические организации включают образовательные подразделения, возможно – исследовательские и иногда подразделения, создаваемые в целях содействия сообществу в целом. Именно на эти, последние, компоненты и обращает внимание новый подход.

Что же представляют собой подразделения содействия сообществу (Community Service Components)? Вспомните, например, что университеты проявляют инициативу, разрабатывая курсы и организуя чтение лекций и проведение семинаров в интересах сообщества. Скажем, консультант по вопросам сельского хозяйства может делиться с фермерами своими знаниями, которые он получил в университетских лабораториях.

Отличный пример того, как академическое учреждение помогает повысить эффективность труда. Почему бы не сделать то же самое в индустрии ПО? Почему бы научному учреждению не предложить в рамках оказания содействия местному сообществу (или его организации) варианты передачи программистам-практикам знаний, добытых исследователями?

Как это сделать практически? Консультанты по разработке ПО могли бы трудиться вместе с программистами, изучая варианты практического внедрения результатов исследований. Другой способ мог бы включать разработку образовательных программ для студентов, изучающих программирование, обращая особое внимание *на связь* научных исследований и практики, а не ограничиваясь только чисто научными или лишь практическими вопросами, как это обычно бывает сейчас.

Такую программу проще всего можно было бы создать в каком-нибудь научном центре, активно задействованном в индустрии ПО. Для этого понадобится площадь, персонал, специально набранный для этой цели, заинтересованность и прямое участие ученых и практиков, а также насыщенная, разнообразная и легко усваиваемая традиционная учебная программа.

Возможно ли это? Есть люди, работающие над тем, чтобы претворить все это в жизнь. Многие из идей, представленных здесь, принадлежат этим людям.

Итак, я описал четыре модели повышения эффективности выполнения программных проектов. Две из них базируются на ресурсах отрасли. Одна создана правительством. И одна – научным сообществом. Похоже на то, что многие в этой стране принимают заботу о повышении производительности в индустрии ПО близко к сердцу.

Метрики ПО: о громоотводах и накопленной напряженности

Закончив презентацию по метрикам ПО, докладчик спросил слушателей, есть ли у кого-нибудь вопросы.

В глубине аудитории поднялась рука. В тоне, которым был задан вопрос, отчетливо звучала враждебность.

– Зачем вы вообще говорите о метриках Холстеда? Они совершенно несостоятельны и не подтверждаются ни одним разумным способом.

– И вообще, – продолжил слушатель, как бы раздумывая, – говорить сейчас о теории Холстеда как о теории программного обеспечения – все равно, что включить алхимию в университетский курс химии.

Наступило гробовое молчание. Такие прямые столкновения в научных кругах все-таки редкость. Пока докладчик приходил в себя и собирался с мыслями для ответа, я стал думать о том, что же собственно происходит. Метрики программного обеспечения, как известно, – это количественные параметры программ. Лучше всего известны количественные параметры, позволяющие оценить сложность ПО. Есть метрики, относящиеся к производительности, качеству, оценке и массе других характеристик, о которых мы хотели бы знать больше. Метрики программного обеспечения – это область теории ПО, которая отыскивает количественные ответы на вопросы, до сих пор считавшиеся качественными.

Существует масса школ определения метрик ПО. Одна из них – это школа Холстеда, которая называется «теорией ПО» и в которой основные метрики определяются для того, чтобы сформировать единый теоретический подход к измерению объектов и явлений программного обеспечения.

Когда Холстед выступил с этой идеей, она казалась плодотворной. Однако с течением времени теоретики и практики разработки ПО, пытавшиеся подтвердить справедливость этих метрик, стали все чаще испытывать разочарования. Иногда полученные метрики коррелируют с измеряемыми параметрами, но многие разуверились в осмысленности этой работы.

По ходу дела метриками заинтересовались бизнесмены. Сейчас на рынке ПО имеются инструменты, которые позволяют рассчитывать различные метрики, в том числе и метрики Холстеда, подкрепляя заявления менеджеров о том, что они знают процесс разработки ПО и управляют им. Эти заявления усилили накал страстей, который уже и без того нарастал в отношении метрик.

Резкая сцена, в которой докладчик был атакован недружелюбными вопросами, иллюстрирует типичную реакцию людей на программные метрики в то время. Докладчик, собравшись с мыслями, начал отвечать, а я продолжил размышления о том, что же *на самом деле* произошло.

Отношения между теоретиками информатики и вычислительной техники и практиками, которые считают эту дисциплину инженерной, довольно напряженные. И тут я начал понимать, что метрики ПО – это своего рода громоотвод, через который напряжение разряжается. Резкий вопрос сыграл роль удара молнии – неизбежного следствия этого напряжения.

Некоторые метрики получить легко, другие, наоборот, трудно. Однако если рассматривать метрики в контексте других сфер интереса индустрии ПО, то окажется, что те из них, получить которые труднее всего, легко поддаются изучению и оценке. Иначе говоря, намного легче подтвердить действительность метрики, чем, например, количественной оценки принципов сокрытия информации, предложенных Парнасом. Никто не подвергает сомнению ценность работы Парнаса и лежащей модуляризации, но никто не представляет себе, насколько именно ПО, отвечающее этим принципам, лучше, чем ПО, им не отвечающее.

Поэтому, говоря о метриках ПО, мы можем не только судить об их теоретическом значении, но и указать (с гордостью или неприятием) на результаты измерения их ценности, основываясь на легко воспроизводимых результатах.

И, что очень важно, это слишком плохо. Мы должны помнить важную мысль о метриках ПО, которая запуталась в паутине протекционизма теоретиков и рекламной шумихи. Мы пока не научились работать с ними правильно, но не должны оставлять попытки научиться. Если неправильна теория Холстеда, то что правильно? Сможем ли мы узнать в будущем что-либо полезное о разработке ПО, опираясь на цифры, собранные в пыли работ прошлого времени?

Ответ, безусловно, утвердительный.

И когда это произойдет, молния больше не ударит в того, кто станет рассуждать о метриках ПО.

Как измерить качество: меньшее притворяется бóльшим

Пытались ли вы недавно купить какие-нибудь CASE-инструменты? Вы искали программы для измерения качества вашего ПО? Если да, то вы

знаете, что сейчас, если не жалеть денег, можно купить программу для измерения качества, способную удовлетворить любые запросы.

Но сколько таких инструментов вам требуется? Вот вопрос. Можно его переформулировать: действительно ли имеет смысл покупать все эти инструменты измерения качества?

В поисках ответа на *этот* внешне невинный главный вопрос мы сталкиваемся с массой побочных моментов: «А что, есть разные способы измерения качества?» «И если да, то какой из них лучше остальных?» И вообще: «Разве для измерения качества нужны инструменты?».

Если вам нужны быстрые ответы, то вот они: «Да», «Мы об этом еще поговорим» и «Вероятно, нет». Посмотрим, что скрывается за этими быстрыми ответами.

А что, разве есть разные способы измерения качества?

Да, качество можно измерять разными способами. По сути существует два подхода к измерению качества, об одном из которых вы, наверное, слышали, а о другом, может быть, и нет.

Более широкую известность получил подход к измерению качества, опирающийся на научную теорию, согласно которой качество ПО может быть измерено на основании его сложности. Сторонники этого подхода утверждают, что сложное ПО трудно понять, трудно модифицировать, оно ненадежно и не так эффективно, как могло бы быть. Между прочим, отличный список параметров качества. И на первый взгляд сложность как мера качества ПО выглядит весьма обнадеживающе.

Не все так просто, однако, с единицами измерения сложности. Прежде всего, разные теоретики отстаивают разные единицы измерения. Одни считают, что сложность программы надо измерять количеством операторов и операндов в ней, другие – что количеством точек принятия решения, третьи предлагают подсчитывать типы управляющих конструкций, или исполняемых операторов, или связей между элементами данных, или связей между логическими точками, или ошибки, или тестируемость, или... Назовите какой-нибудь параметр, и наверняка окажется, что кто-то уже использует его как меру сложности ПО.

Все это оборачивается разнообразием измеряемых параметров и, что интересно, обуславливает значительные затруднения в количественном определении сложности ПО. Дело в том, что, по данным теоретиков, проводивших сравнительные измерения, большинство измеряемых параметров согласуются между собой. И если взять программный код реального приложения, то окажется, что большинство способов измерения сложности ПО дадут приблизительно один и тот же ответ. Но,

и это большое НО, тот же ответ можно получить простым подсчетом строк исходного кода. А это очень просто, даже банально!

Самая главная неприятность, связанная с наиболее популярными способами измерения сложности, следовательно, заключается в том, что для любого из них нетрудно найти эквивалент. А если это так, то зачем нужна морока с покупкой специального инструмента?

С измерением сложности ПО связана и еще одна серьезная проблема. Вспомним, что в основе всего лежит тезис об изначально неясной природе сложности и, как следствие, о том, что сложное ПО – это ПО низкого качества. А что если ПО создано для решения очень сложной задачи? Известно, что сложность задачи, как правило, гарантирует сложность решений. Следовательно, если программе присваивается высокий рейтинг сложности, это может означать всего лишь то, что ее разработчикам надо было решить исключительно трудную задачу.

Эту трудность, конечно, обойти довольно просто. Программа с высоким рейтингом сложности (по количеству строк кода или по каким-то другим, более сложным критериям) является лишь *кандидатом* на дальнейший анализ. Может быть, программа неоправданно сложна, и тогда ее надо упростить. Но может быть, разработчики просто не могли избежать создания сложного продукта. Чтобы понять это, нужен хороший анализ.

Голос из другого сообщества

Оказалось, что о качестве ПО можно судить не только по его сложности. Помните, что единицы измерения сложности были придуманы теоретиками из университетской среды? Однако интересную работу по измерению качества проводят и в другом сообществе. Из уголка исследовательского мира, финансируемого военными, до нас доходят намного более скудные новости. Работы, которые проводятся здесь, более всесторонние и во многом значительно интереснее, чем в том сообществе, о котором мы только что говорили.

Здесь предлагают измерять качество при помощи более полного набора параметров, не ограничиваясь одной только сложностью. Качество разбивается на составляющие элементы, каждый из которых подвергается измерению. Вы должны были слышать об этих элементах, и большинство исследователей включают их в свои определения качества ПО – это переносимость, надежность, результативность, эргономичность, понятность, тестируемость и модифицируемость. Вот элементы, по которым в этом сообществе измеряют качество. (На самом деле в исследованиях военно-воздушного ведомства задействовано 13 таких элементов.)

Идея состоит в том, чтобы измерить каждый из них (если это возможно), присвоить им значения относительного веса для данного конкретного программного продукта и, просуммировав эти значения, вычислить качество этого продукта:

Качество = SUM(Числовое значение элемента * относительный вес элемента)

Возникает следующий вопрос: насколько точно мы в состоянии измерить эти элементы? Над этим вопросом работают на базе ВВС США в северной части штата Нью-Йорк.

Растущий интерес к качеству ПО привел к тому, что уже более десяти лет по их заказам проводят исследования различные промышленные организации. На первых порах очень большую работу вели в «Дженерал Электрик». Позже большую активность в этих исследованиях проявила компания «Боинг».

Что же они сделали? Они разработали методику измерения элементов качества, основанную на применении рабочих таблиц. Для каждого элемента создавалась таблица с длинным перечнем информации, которую надлежало зафиксировать, чтобы получить измерение. В таблицу записывались такие параметры, как количество главных функций, или количество логических путей исполнения, или количество оверлеев. Информация обычно собирается вручную, и происходит это медленно. В контрольных исследованиях пользователи тратили на заполнение рабочих таблиц метриками от 4 до 25 ч.

Который же из этих подходов самый лучший?

Мы уже обсудили значение методики, основанной на метриках сложности. При всем многообразии способов подсчета сложности самый простой, основанный на подсчете строк исходного кода, представляется ничуть не менее правильным, чем остальные. Как эта метрика учитывается в исследованиях ВВС?

Если коротко, то есть сложности. Как говорится, есть две новости: хорошая и плохая. Хорошая новость: исполнители, которым ВВС заплатила, чтобы они оценили своевременность выполнения работ, нашли результаты «надежными» и «разумными» и сказали, что выводы «совпадают с интуитивными ожиданиями разработчиков относительно качества продукта». Плохая новость: они также пришли к выводу, что способы измерения «случайны» и им не хватает «реальных основ, необходимых эффективной методологии измерений». Более того, практики, похоже, нигде не применяли этот метод к измерению качества. Конечно, это могло объясняться тем, что идеи данного сообщества не находят широкого отклика за пределами военного ведомства. Однако скорее дело

не в этом, а в том, что повсеместное распространение данного метода измерения сдерживается его недостаточной экономической выгодностью.

Действительно ли нужен инструмент, чтобы измерять качество?

Какая мысль здесь главная? Существует масса критериев качества. Есть даже два не очень похожих сообщества, разрабатывающих два довольно разных подхода. Однако дело в том, что ни один из них явно не дает хороших практических результатов.

Поэтому, если надо измерить качество ПО, собственного или приобретенного у другой компании, сейчас не удастся избежать глубокого и трудоемкого анализа, выполняемого вручную высококласными экспертами.

Конечно, при желании в помощь им можно купить инструмент. Но этот инструмент даст вам не намного больше информации, чем количество строк исходного кода, которое можно найти в конце листинга программы.

За более подробной информацией о проекте измерения качества ПО, реализуемом в лабораториях BBC, можно обратиться по адресу:

Роджер Б. Панара (Roger B. Panara)
Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base NY 13441

Можно ли ВНЕДРИТЬ качество в программный проект

К какой сфере относится качество ПО: к менеджменту или к техническому исполнению?

Очень многие думают, что к менеджменту. В книгах или на занятиях, посвященных качеству ПО, подразумевается, что в непосредственной близости от слов «качество ПО», должно располагаться слово «обеспечение». С учетом этого дополнения вопрос качества, безусловно, можно отнести к сфере менеджмента.

Однако мне это дополнение не кажется необходимым. Я уверен, что главная трудность, связанная с качеством ПО, техническая, а не административная, и я хочу расцепить слова «качество» и «обеспечение».

Такая оценка, ставящая технологию на первое место, требует объяснения. Посмотрим на технический и административный аспекты качества программных продуктов.

Я думаю, что «внедрить» качество в программный продукт не более возможно, чем «втестировать» его, независимо от того, насколько тщательно менеджеры планируют и обеспечивают внедрение качества в разработку ПО. Никто не спорит, что качество нельзя «встроить» в ПО при помощи тестирования (потому что тестирование проверяет только надежность, одну из граней качества, и кроме того, тестирование проводится на слишком поздней стадии жизненного цикла, чтобы оно могло предотвратить плохое качество), но довольно многие уверены, что качество можно внедрить административно. Радикальная точка зрения.

Я думаю, что качество нельзя обеспечить *при помощи мер менеджмента*, потому что качество – это глубинное свойство программного продукта. Если мы проанализируем, что же такое качество *на самом деле*, то увидим, что и его «внедрение», и его детекция происходят далеко за фасадом процесса разработки. Как «вложить» в программу понятность и модифицируемость (два признака ее качественности)? Как узнать, было ли это сделано? Легких ответов на эти вопросы нет, но суть в том, что сделать это может только технический специалист. Понятность и модифицируемость программы имеют сугубо техническую природу, и случайный наблюдатель попросту не в состоянии оценить их. А менеджеры, хорошо это или плохо, в данном смысле лишь случайные наблюдатели. Менеджеры в целом не настолько знакомы (да и не должны) с технической стороной, чтобы обеспечивать или оценивать понятность и модифицируемость.

И это не единственный пример ключевой роли технологии в обеспечении качества. Программное обеспечение должно быть надежным почти всегда. Программное обеспечение должно быть переносимым, по крайней мере иногда. И переносимость, и надежность – это атрибуты качества. Кто, кроме технического специалиста, может обеспечить надежность ПО или отличить переносимое ПО от непереносимого? Способности большинства менеджеров в этих областях либо мизерны, либо вообще отсутствуют.

Конечно, нельзя сказать, что менеджеры никак не участвуют в создании качественного ПО. Напротив, им отводится очень важная роль. Они должны создать благоприятную среду, в которой качество возвращается и взлелеивается.

Что я понимаю под средой качества? Такую среду, в которой обеспечиваются и соблюдаются процессы, облегчающие создание качественного ПО. Такую, в которой приобретаются и применяются инструменты, помогающие обеспечить качество. Такую, в которой нанимаются и поощряются специалисты, которые заботятся о качестве. Такую, в которой качество продукта отстаивается наравне с необходимостью соблюдать сроки выполнения работ и укладываться в бюджет.

Все это проще сказать, чем сделать. Мы живем в эпоху, когда сроки и бюджет играют доминирующую роль в оценке менеджмента и разработки ПО. Для этого есть веские причины. Нередко при выполнении программных проектов сроки и бюджеты нарушались самым вопиющим образом. Столкнувшись с проблемами, доминирующими в этой области, менеджмент совершенно правильно сосредоточил свои усилия на их разрешении.

Здесь кроется опасность. Возрастающее давление сроков и бюджетных ограничений неизбежно ведет к ухудшению именно качества. Как можно ускорить затянувшуюся сдачу продукта? Обнаружив проблему, сэкономить на чем угодно (обычно это бывают верификация и тестирование). В результате ухудшается качество продукта.

Менеджер программных проектов XXI века должен найти новое решение. Старое уравнение

программный продукт = в срок + в рамках бюджета

должно быть модифицировано:

программный продукт = качество + в срок + в рамках бюджета

Для достижения хороших результатов это более сложное уравнение требует, чтобы менеджеры считали качество такой же важной конечной целью, как соблюдение сроков и бюджета.

К счастью, от менеджеров не требуется больших усилий, чтобы облегчить создание качественного ПО. Большинство технических специалистов в общем и целом стремятся создавать хорошее, качественное ПО. На самом деле бывает, что в поисках оптимального соотношения между качеством, стоимостью и сроками технические специалисты слишком большое внимание уделяют качеству! (В 1989 году в одном из интервью Стив Джобс сказал: «Наша задача как топ-менеджеров состоит в том, чтобы помогать специалистам как можно яснее увидеть цель ... и как можно быстрее освободить им дорогу».) Как правило, находятся люди, которые с большой охотой помогают менеджерам внедрять качественное ПО.

Такие дела. Технические задачи, связанные с обеспечением качества ПО, намного сложнее и важнее, чем задачи менеджмента. Обеспечение качества важно, но это только одна дорога к хорошему ПО.

Попробуем выполнить маленькое упражнение. Я сейчас напишу слово «качество». Посмотрим, сможете ли вы удержаться и не добавить торопливо не написанное мною «обеспечение».

Качество.

Ну вот, это оказалось не так уж трудно, правда?

Легенда о плохом программном проекте

Жил-был когда-то программист, и случились у него БОЛЬШИЕ неприятности. Неприятности этого программиста состояли в том, что его программный проект не уложился в бюджет, сроки полетели ко всем чертям, а программа была ненадежной.

Те, кто читал о так называемом кризисе программирования, может быть, подумали сейчас, что ничего такого особенного в этом нет.

Однако для нашего героя такая ситуация *была* необычной. Причем я бы сказал, что намного более необычной, чем вы, может быть, подумали. Дело в том, что этот программист закончил одно из лучших учебных заведений в стране, в котором готовят программистов. И после этого у него было несколько лет хорошего боевого опыта программирования. Он даже отыскал способ объединить лучшее из того, чему его научили в вузе, с лучшим из того, чему он научился за эти несколько лет практической работы.

Другими словами, этот программист был настолько хорошим программистом, насколько программисту следует быть. Почему же все пошло не так?

Все началось с того момента, когда была сформулирована задача, которую должна было решить его программа. Это была актуальнейшая задача, и компания могла заработать на ней кучу денег. Так ему сказали менеджеры. Так ему сказали специалисты по маркетингу. Казалось, мало кто сомневался, что именно к этой задаче нужен особый подход.

Первая особенность подхода состояла в том, что требовалось сделать все к определенному сроку. Так сказали менеджеры. Так сказали специалисты по маркетингу. Наш программист не думал, что к этому сроку можно успеть, но это никого не интересовало. Надо было успеть.

Наш программист был сознательным членом команды и начал работать над задачей, несмотря на свои сомнения. Надо сказать, что менеджеры прислушивались к его словам. Программист писал свою программу, а они испытывали инструменты моделирования стоимостных ожиданий, пока не нашли тот, который дал нужный им ответ. «Смотри, – радостно сказали ему менеджеры, – и ты тоже можешь написать свою программу за то время, которое у тебя есть».

А время шло, приближался срок сдачи, и наш программист нервничал все сильнее. Сначала он старался придерживаться своих собственных стандартов качества, тщательно соблюдая требования, основательно подходя к проектированию, вдумчиво проверяя каждую строчку программного кода.

Однако когда срок сдачи приблизился вплотную, программист начал понемногу отодвигать свои проверенные надежные методы. Тестирование выполнялось по сокращенной программе, наскоро. Тестируемые модули попали в сборку вместе с модулями, которые не тестировались. На бета-тестирование попал продукт, который не прошел бы нормальное внутреннее тестирование. Срок сдачи получил статус священной коровы, а все остальные соображения – под давлением, конечно, – были сочтены менее важными и принесены в жертву.

В конце концов программист провалил срок сдачи примерно на столько, на сколько предполагал с самого начала. Расходы, конечно, превысили ожидаемые, потому что оценивались по оптимистическому сценарию своевременной сдачи продукта. А надежность? Она тоже пострадала, потому что программист, пытаясь не отстать от графика, сократил слишком многое. Последствия вполне соответствовали духу кризиса программирования – еще один программный проект отстал от графика, не уложился в бюджет, а получившийся продукт был ненадежным.

В общем, жил-был когда-то хороший программист, который провалил программный проект. В глубине души программист знал, что не должен был так снижать требования к качеству. Но еще он знал, что в процессе разработки была допущена всего одна настоящая ошибка.

Она состояла вовсе не в том, что он как-то не так разрабатывал продукт. А в том, что график выполнения был неправильным изначально.

Прочитав отзыв менеджеров, программист узнал, что его рейтинг понижен, потому что он плохо работал и провалил проект.

И он задумался. Сколько людей разделили его судьбу? До какой степени кризис программирования на самом деле вызван неправильными или дутыми оценками?

Он до сих пор думает.

МАРКЕТИНГ

А вы купили бы у короля Людовика автомобиль с пробегом?

Полностраничная реклама в любом компьютерном издании стоит недорого. А цветная полностраничная реклама стоит еще дороже. Непомерно дорого.

Но мы живем в эпоху компьютерных звездных войн. Эти войны разгораются на страницах газет, журналов и даже в телевизионном эфире.

Попытка продвинуть на рынок программный продукт, адресованный широкой аудитории, совершенно безнадежна, если она не сопровождается покупкой места для рекламы в одном из этих пространств. Пожалуй, можно даже сказать, что этого места должно быть много.

Вам, наверное, доводилось видеть примеры. Помните, как Ring-a-ding Tel and Tel впервые вывела на рынок новую линейку компьютерных продуктов? Они купили не одну страницу рекламы. И не одну страницу цветной рекламы. Они купили сразу несколько цветных страниц. Призыв «Покупайте наш продукт» занимал столько места на страницах того издания, где была напечатана эта реклама, что не увидеть его было невозможно.

Или как одна компания, разрабатывавшая СУБД, вступила в борьбу со своим конкурентом? Наш продукт R, заявляли они, ловко скрывшись за загадочной аббревиатурой, превосходит продукцию конкурентов. И на нескольких дорогостоящих страницах поясняли, в чем именно. При том количестве денег, которые крутятся в рекламе, вполне понятно, что эти компании вкладывают сравнимые суммы в рекламную полиграфию. По крайней мере, большинство. В традиционных рекламных агентствах развивается новая область специализации, а именно умение громко и ярко рекламировать программное и аппаратное обеспечение. Компьютерный рынок стал массовым, и товары на нем, нравятся нам это или нет, продаются по законам массового рынка.

Это была вводная часть к моему следующему рассказу. Это рассказ о компании-разработчике ПО, которая прошла весь путь. Полностраничная реклама. Цветная реклама. Рекламное агентство. Они прошли все этапы, о которых мы сейчас говорили. Он все сделали как положено.

За исключением одной мелочи.

Вы когда-нибудь видели изображение замка Нойшванштайн? Готов поспорить, что да (даже если вы не знали, что это был он). Это такое средневековое, похожее на крепость здание, расположенное высоко в Баварских Альпах, изображение которого украшает большинство европейских туристических брошюр. Башни и башенки, растущие вокруг высокие деревья, и все это на фоне горных вершин во всем их невообразимом величии.

Но есть в этом замке кое-что странное. Он как будто ненастоящий. Конечно, я не хочу сказать, что его там нет или что-то в этом роде. Я хочу сказать, что он не средневековый. И никогда не был крепостью. Он был построен королем Баварии Людовиком в XIX веке. Да-да, в девятнадцатом. То есть не в Средние века. И в эту эпоху крепости уже были не нуж-

ны. Поэтому замок Нойшванштайн при всем его великолепии является не совсем тем, чем кажется на первый взгляд.

С замком короля Людовика связана еще одна странность. Это сам король Людовик. История утверждает, и это почти бесспорно, что король Людовик был своеобразным анахронизмом. Он построил замок потому, что хотел жить в эпоху Средних веков. Он был равнодушен к истории французской короны, а именно к тому ее отрезку, на котором существование французской короны было прекращено при помощи гильотины. Некоторые историки даже утверждают (но не все с ними согласны), что старый Людовик совершенно выжил из ума. Известно, что он пустил по ветру все свое королевство и был отстранен от власти, прежде чем закончилось строительство Нойшванштайна.

А теперь уточним кое-что. Представьте, что вы президент Security Software и хотите вывести на рынок новейший продукт, который защищает данные, сочетая применение блокировок, паролей и шифрования. Все это вместе называется сверхнадежной защитой данных. И вы хотите, чтобы дорогостоящая реклама, которую вам придется купить, вызвала ассоциации с несокрушимой силой. Замок Нойшванштайн? Завораживающая красота, и кроме того, картинка подходящая. Отличный рекламный ход!

Компания Security Software существует и занимается как раз защитой данных (хотя и называется иначе), и она выбрала именно этот путь. И реклама получилась отличная. За исключением одной мелочи. Работники рекламного агентства, очевидно, не знали о замке Нойшванштайн столько же, сколько знаем мы.

Рекламный текст начинался словами «Секрет замка короля Людовика». И продолжался так: «Высоко в Баварских Альпах стоит грандиозный замок Нойшванштайн». Но затем его авторы вступили на зыбкую почву. «Величественная средневековая твердыня преподает нам полезный урок в проектировании безопасности данных. Именно приверженность короля Людовика к тщательному планированию сделала замок Нойшванштайн такой неприступной крепостью». Далее в рекламе рассказывалось, что именно делает компания Security Software, чтобы обеспечить такую же глубоко продуманную, непроницаемую защиту данных от злоумышленников.

Насколько я понимаю, новые возможности и большие деньги индустрии ПО помогли нам изобрести новые способы провалить дело. Винной всему может быть не только плохой программный продукт. Рекламные трюки тоже таят в себе неприятные сюрпризы.

Однако интересно немного продолжить размышления о рекламной игре. Если программный продукт сделан неудачно, то это просто пло-

хой продукт, который незаметно исчезнет с рынка ПО. Плохая реклама способна обернуться другими последствиями. «Слава» о вашей нелепой ошибке может намного пережить «славу» о самом продукте, как это было в случае с подвесным мостом Тэкома-Нэрроуз, который сдуло ветром. Вот, собственно, та мысль, которую я хотел донести до вас, когда писал эту статью.

А теперь представьте себе эту роскошную рекламу с замком Нойшванштайн. Вы видите сам замок, стоящего перед ним короля Людовика и флаги, реюющие над крепостными башнями. «Продукты компании “Security Software” – это как раз то, что вам надо», – гласит рекламный текст. Вы читаете это, а внутренний голос спрашивает: «А вы купили бы у этого человека автомобиль с пробегом?»

КОНСАЛТИНГ

Подноготная консалтинга

Николас Звегинцов (Nicholas Zvegintzov)

Примечание автора из первого издания:

Задумывались ли вы о том, какова реальная жизнь консультантов экстра-класса? Почет и уважение, 5000 долларов в день на расходы, полеты на «Конкордах», выступления с основными докладами, роскошная еда и толпы поклонниц? В таком иронически-насмешливом тоне Николас Звегинцов, эксперт по управлению программными системами, редактор Software Maintenance News, отзывается о работе Джерри Байнберга «The Secrets of Consulting—A Guide to Giving and Getting Advice Successfully» (Секреты консалтинга – как надо давать и получать советы), выпущенной издательством Dorset House Publishing в 1985 году, раскрывая тайны консалтинга (или, по крайней мере, тайны с 1-й по 4-ю!).

Однажды январским воскресным утром я заснул на пляже в Хермоса-Бич (штат Калифорния). Проснувшись, я увидел троих малолетних серферов и спасателя, сидевших в бежевом джипе, на крыше которого красовалась доска для серфинга, и сверливших меня взглядами. Мальчишки смотрели сурово, как ангелы мщения.

Один из них сказал: «Он не бегал, не катался на велосипеде или скейте и не занимался серфингом. Он храпел. Это было симфонично.»

Второй добавил: «Я как-то видел хиппи-наркомана, жившего на пляже. Мой дед знал его».

Третий: «Интересно, если мы его сдадим, нас покажут по телевизору? А вознаграждение заплатят?»

Полицейский погладил акулий электрошокер.

Я ответил: «Я консультант с Восточного побережья. Прилетел на самолете, и у меня нарушены суточные ритмы». Показал им книгу в красной обложке: «Читал “Секреты консалтинга” Вайнберга и заснул. *Datamation* заказала мне рецензию. Я обещал прочитать книгу в самолете. Если верить Джеймсу Мартину, то это лучшее чтиво для консультанта, который летит в самолете».

«Я однажды видел самолет очень близко, – сказал один из мальчишек. – Целый день катил на скейте и докатил до здорового бетонного поля. Это было *авионично*».

«Консультанты появляются из самолетов, – сказал другой мальчишка. – Нам говорили на уроках полового воспитания».

«Дай-ка я с ним поговорю, сынок, – сказал полицейский. – Джеймс Мартин, это не его ли называют главным IT-гуру всех времен? Хотел бы я знать все секреты консалтинга и быть таким IT-гуру, к которому все прислушиваются! Как это только вы заснули, читая “Секреты консалтинга”?»

«Слишком уж там все лучезарно», – ответил я.

Полицейский издал низкое недружелюбное ворчание и ткнул пальцем в медведя на рукаве формы калифорнийских спасателей. «Как это “слишком лучезарно”?» – спросил он.

Я понял свою ошибку. «Ну, пожалуй, лучше сказать, что все это слишком хорошо, чтобы быть правдой. Вайнберг говорит, что секреты консалтинга в том, чтобы тонко чувствовать, быть скромным, готовым оказать поддержку, надежным, не требовать слишком много и не ожидать слишком мало».

Полицейский зевнул: «А-а, понимаю. Это *ненастоящие* секреты».

«Конечно. Вот Первый Настоящий Секрет: у каждого консультанта есть лишь одна главная идея. Джеймс Мартин говорит, что программисты – это главная причина несвоевременного выполнения работ. Ричарду Нолану принадлежит мысль, что IT-группы в своем развитии проходят несколько стадий. Т. Каперсу Джонсу – что IBM располагает данными, раскрывающими секреты повышения производительности. Согласно Харлану Миллзу, путь к безошибочным программным системам лежит через математику, а по Э. Ф. Кодду, данные имеют реляционную природу. Том Гилб говорит, что необходимо сводить проектирование к измеряемым показателям.

Вейнбергу принадлежит мысль, что работоспособность команды и успех проекта необходимо обеспечивать при помощи технического руководства, а не политических игр».

Я остановился.

«Ну тогда не скромничай, – сказал спасатель. – Выскажи *свою* главную идею».

«Главное в разработке систем – это сопровождение».

«Эти великие откровения, – гордо сказал один из мальчишек, – я узнал, когда смотрел видео Western Civ. Это было *эклектично*».

«Да, – сказал я, – но Второй Большой Секрет заключается в том, что главная идея вовсе не должна быть правильной. Если уж на то пошло, то некоторые из этих главных идей (не будем указывать пальцем, какие) попросту ложные».

«Но как же можно заставить поверить в идею, если она неправильная?» – спросил спасатель.

«А она и не должна быть правильной, она должна быть привлекательной. Посмотрите вокруг». Я показал на скейтбордистов, бегунов и велосипедистов, заполнявших прибрежную полосу.

Спасатель опять начал издавать медвежье ворчание, и я поспешил перейти к Третьему Большому Секрету. «Консультанты “стригут” своих клиентов понемногу, снимая лишь тонкий слой. Обработав одну группу, консультант переходит к следующей. Вот скажите, что находится за Калифорнией?»

«За Калифорнией ничего нет», – ответил один из мальчишек.

«*Северная* Калифорния», – сказал другой.

Третий мальчишка: «Мои предки как-то взяли меня с собой в Вегас, но оставили в гостинице в детской комнате. Это было *герметично*».

«В общем, ты все правильно понял, – сказал я. – За Калифорнией Невада, за Невадой Юта, за Ютой Колорадо, а за Колорадо Небраска. Вайнберг живет в Небраске».

«*Мезозойно*», – отреагировал мальчишка, которого возили в Вегас.

«Минуточку, – сказал спасатель. – А что бывает, когда консультант находит клиентов, которых недавно стригли?»

«Для этого существует Четвертый Большой Секрет. Нет двух консультантов, которые питались бы одним кормом. Если они постоянно меняются, то всегда есть что стричь».

«Как это может быть?» – спросил спасатель.

«Природа – волшебная штука, – сказал я. – Знаете, что такое партеногенез?»

Один из мальчишек сказал: «Это примерно как когда мама куда-нибудь уезжает, а папин приятель дядя Брюс приходит и начинает готовить еду, которая у него всегда подгорает?»

«Почти, но не совсем. Это больше похоже на то, как посетители бара знакомств расходятся по домам поодиночке – и считают, что им повезло. С консультантами примерно то же самое».

«Однако твои Большие Секреты как-то не греют, – сказал спасатель. – То есть каждый консультант вечно одинок и обречен на добывание хлеба насущного, вечно повторяя одну и ту же “главную мысль”? Да это просто ад. Я, конечно, очень привязан к спасательскому образу жизни, но если большие секреты консалтинга заключаются в этом, то мне пора сдать свисток и забраться в мешок для трупа. Как будто солнце скрылось за темным облаком, вода замерзла и превратилась в снег, деревья потеряли листву, и выдыхаемый воздух, дымясь, превратился в мокрый сгусток, который зазвенел бы, если бы ударился о железную землю».

«*Арктично*», – всхлипнул самый маленький серфер. По его прорезиненному костюму покатились слезы.

«Смотри, что ты наделал, – проговорил спасатель. – И тебе не стыдно? Что мне сделать, чтобы в их глазах опять засиял свет?»

Я встал, отряхнул песок со складок моих консультантских одежд и протянул ему книгу Вайнберга «The Secrets of Consulting».

«Прочитайте им это», – сказал я.

Уходя с пляжа, чтобы позавтракать в заведении «У Джерри на берегу», я видел три детских фигурки вокруг спасателя и слышал, как три голоса восклицали: «*Эпично! Сказочно! Восторг!*»

Так уж получилось, что я знал, какую фразу из этой рецензии издатель поместит на обложку книги.

Какие прогнозы давали предсказатели раньше

Предсказания – дело увлекательное.

Перевернуть лист календаря с важной датой – это как повернуть ручку шкатулки с сюрпризом. Предсказатель, предрекающий будущее, появляется так же, как из шкатулки раздается мелодия.

Ничего не имею против предсказателей. Во-первых, заглядывать в будущее необходимо. Все мы жаждем подсказок о том, что нас ждет, что-

бы лучше знать, что делать с тем, что уже произошло. Во-вторых, предсказатели делают жизнь разнообразнее. Где бы мы брали пищу для своих фантазий, если бы думали только о настоящем? На самом деле все мы тайные предсказатели. Все дело в том, что некоторые из нас предсказывают не так хорошо, как другие.

Это плавно подводит меня к теме данного очерка. Злонамеренно-ехидная часть моей натуры говорит: «Сейчас 90-е годы XX столетия, посмотрим, насколько точны были предсказатели». Имеет ли смысл прислушиваться к ним? Очень полезно (и честно) задать этот вопрос. Сделаем это.

Пройдемте со мной к пыльным библиотечным полкам. Тсс! Предсказатели не должны знать, что мы собираемся сделать. Спокойно и бесшумно, пройдем поскорее к полкам с компьютерной периодикой прошлых лет.

Предсказатели прошлых лет находятся здесь, среди страниц, заполненных развевающимися локонами юных секс-символов, рекламирующих компьютерные товары, и строгими ежиками корпоративных менеджеров. Послушаем. Что они говорят?

Вот, пожалуйста. В январе 1970 года один из предсказателей отвечал на актуальный в то время вопрос. «Третье поколение, этот младенец, встал на ноги и пошел, а сейчас он уже бегаёт... За горами ли следующее поколение?» В ответ звучит уверенное «нет», подтверждаемое всеми последующими событиями [AMDA70].

А теперь мы вернулись на много лет вперед. Малыш третьего поколения по-прежнему бегаёт, хотя и выглядит немного иначе благодаря многочисленным косметическим подтяжкам. Но по своей сути это все то же дитя, только постаревшее.

«Но, – шепчете вы тихонько, чтобы не потревожить других читателей, стоящих у соседних библиотечных полок, или чтобы не спугнуть подслушивающих предсказателей, – возможно, это был взгляд одного из предсказателей. Конечно, остальные знали, что чудеса и превратности смены поколений уже были готовы раскрутиться во всю мощь?»

А вот и еще один прогноз. В январе все того же 1970 года другой предсказатель написал: «Следующее поколение придет скоро, оно будет несовместимо с теперешним, пользователи ухватятся за него, ошибки, сделанные в 1965 году, будут повторены и усилены, и обо всем этом вы раньше всего прочитаете здесь». [DORN70] Не имеет значения, что в этом предсказании есть доля иронии, ситуация не улучшается, а ухудшается. Сейчас предсказатели борются за первое место в очереди на право ошибиться! Во всяком случае по крупным вопросам вроде

поколений вычислительных систем, предсказатели несли практически чистый бред.

«Ну хорошо, а что думали в прошлом о революции, связанной с микрокомпьютерами и распределенными вычислениями? – тихо спрашиваете вы. – Конечно, пророки знали о ее наступлении». Хороший вопрос. Проверим.

Одно из предсказаний мы найдем здесь, в периодике, настоящей на библиотечном воздухе. «В конечном счете такое ощущение, что для крупных корпоративных пользователей имеет место тенденция к централизации...» [AMDA70]. Вы, наверное, скажете, что формулировки «такое ощущение» и «крупные корпоративные пользователи» немного уклончивы? Хорошо, а как вам вот это: «...если говорить об общих вычислениях, то экономия за счет масштабирования чаще проявляется на больших машинах». [SOLO66]

Тут все совершенно ясно. Эти предсказатели совершенно неправильно оценили потенциал распределенных вычислений и микропроцессорной техники.

Пройдем дальше и посмотрим, что говорили о программном обеспечении. Еще не кончились 1970-е, мы только начинаем приходить в себя от смены третьего поколения вычислительной техники и «развязывания» цен на ПО. Помните развязывание? На заре развития вычислительной техники все ПО поступало вместе с аппаратным обеспечением «бесплатно». (Слово «бесплатно» взято в кавычки, потому что нередко более высокая цена на компьютер обуславливалась лишь тем, что вместе с ним поставлялось программное обеспечение.) Затем, в конце 1960-х, некоторые из компаний, производящих компьютерное оборудование, пришли к разумному выводу, что программное обеспечение стоит слишком дорого, чтобы этим не воспользоваться, и стали брать за него деньги, отделяя его стоимость от общей цены на компьютер. Как раз примерно в это время у бизнесменов, занимавшихся производством и продажей программного обеспечения, разыгрались волчьи аппетиты, стимулируемые фантазиями о потенциальных прибылях, которые сулило развязывание цен. В самый разгар предвкушения, в 1970-х, было предсказано, что «вскоре не меньше 1000 компаний станут продавать компьютеры и ПО отдельно, предложив покупателям целое море программных продуктов» [BROM70].

Увы, как и остальные предсказания в этом очерке, данное было результатом «прозрения» сквозь затуманенный магический шар гадателя. Производители компьютерного железа, которые умели читать прогнозы предсказателей не хуже любых других игроков компьютерного рынка, установили свои цены на ПО аккуратно – достаточно высокими, что-

бы получить прибыль, и достаточно низкими, чтобы помешать бизнесу компаний-хищников, – и бум бизнеса на программном обеспечении провалился. Не совсем, конечно. Известны успешные фирмы, занимающиеся производством ПО, и большинство из них, конечно, выросли благодаря перевороту, произведенному появлением персональных компьютеров, а не развязыванию цен. Но все равно, речь не идет о сотнях компаний, каждая из которых предлагает море программных продуктов.

Теперь попробуем найти здесь что-нибудь еще... видите? Это настоящая сокровищница давних предсказаний. Это записи интервью, взятых в эпоху 1970-х у тех, кто принимал решения в корпорациях. Каждый из проинтервьюированных, похоже, руководствовался личными мотивами и давал прогноз, который соответствовал его собственным желаниям. Здесь мы имеем дело с новым типом предсказателей, которые убеждены, что предсказание может сбыться уже потому, что оно сделано. Что же предсказывалось в этих попытках самосбывающихся пророчеств? Попробуем-ка вот эти:

- По поводу малых вычислительных машин... «Мелкие компании не собираются приобретать и использовать малые компьютеры. Для них этот путь нецелесообразен. Предприятия малого бизнеса будут обращаться к ресурсам больших вычислительных систем» [BENN70].
- О человеческом факторе... «По мере развития и становления отрасли, увеличения количества компетентных, квалифицированных специалистов... работодатели смогут выбирать работников из большего числа кандидатов...» [PARR70].
- О языках программирования... «Уверен, что главные перемены выразятся в отказе пользователей и от Кобола и от ПЛ/1... В компьютерном сообществе продолжают говорить о Коболе так, как будто они знают, о чем говорят. Однако в конечном счете Кобол не работает... он безнадежен» [HARR70].
- О вычислительных центрах... «Вычислительные центры стоят на пороге больших перемен... Самая главная из них: многие компании, вероятно, откажутся от собственных специализированных вычислительных мощностей» [YOUN70].

Чистый бред! Эти предсказатели вообще ничего не поняли (по крайней мере, главного) также и по таким вопросам, как переворот в сфере микрокомпьютеров, дефицит программистов (иногда весьма тревожный), вездесущность Кобола (он ПО-ПРЕЖНЕМУ занимает первую строчку в программистском хит-параде) и значение собственных компьютеров для организаций. Просто фестиваль несбывшихся пророчеств!

Может быть, я допустил какую-то несправедливость в своем анализе? Конечно допустил. Прежде всего, эпоха 1970-х была особенной. Не успела отрасль пережить свой подъем (в 1970 году один из ведущих компьютерных журналов, выходивший раз в месяц, стал выходить в два раза чаще, потому что располневшее содержимое просто перестало помещаться в переплет), как наступил глобальный спад (и к началу 1972 года тот же самый журнал опять выходил раз в месяц, но уже в значительно истончавшей ипостаси). Даже те, кто печатал предсказания, ошиблись в своих предположениях!

Однако, может быть, несправедливо сосредоточиваться на прогнозах популярных периодических изданий. Нет ли более точных прогнозов на страницах серьезных научных изданий?

Вот на этих полках лежат престижные журналы. Посмотрим, что печатали в них.

Прежде всего надо сказать, что вульгарные предсказания в них отсутствуют. Заглянем в этот журнал, вышедший в конце 1960-х. Есть статьи о сортировке, о наборах символов, и ни слова о том, что они будут представлять собой в будущем.

Но с другой стороны, каждая статья в серьезном журнале – это завуалированное предсказание. Вы, наверное, знаете, что ученые называют сегодняшние исследования практикой завтрашнего дня? Под «завтра», конечно, надо понимать несколько лет, однако чаще всего мы подразумеваем десятилетие. Заглянем же снова в эти серьезные статьи и посмотрим, что в них предсказывается.

Как и прежде, нас интересуют 1970-е. Популярные журналы говорили о наступлении эпохи вычислительной техники четвертого поколения, централизованных вычислительных систем, повсеместного распространения пакетов ПО. А что же их более серьезные собратья?

Вот, например, статья о переводе с естественных языков. А-а, как же, не за горами то время, когда слова, фразы и суждения будут автоматически переводиться с русского (для примера) на английский. Все очень хорошо, но есть одно маленькое «но». «Но» заключается в том, что пословный перевод, за исключением простейших случаев, оказался неосуществимым – благодаря идиомам, сленгу и многозначности слов. Попытки перевести выражение *Time flies like an arrow* терпели крах из-за двусмысленности слов *time* и *flies*, употребляемых в качестве и существительных и глаголов, и машинный перевод с естественных языков, уподобившись полету тяжелой мухи, которой обкорнали крылья и которая в конце концов шлепнулась на землю, впоследствии возродился уже в качестве вспомогательного средства переводчиков.

А вот еще одна научная статья 1970-х, на этот раз об информационных системах управления (MIS). Да, загадки эпохи MIS. К сожалению, они во многих отношениях оказались ошибками. Сейчас мы видим, что знали совсем не так много об информационных аспектах управления организациями. Ожидания 1970-х растаяли и стали отчасти успешными исканиями 1980-х. Успешная реализация крупномасштабных интегрированных информационных систем управления так и осталась мечтой. Были ли предсказания ученых 1960-х годов сколько-нибудь точнее? Перейдем на десять лет назад и посмотрим более старые статьи о вычислительной технике. Вот, например, статья о доказательстве теорем. Автор статьи обещает, что очень скоро компьютеры будут применяться для доказательства математических теорем и разрешения трудных философских вопросов. Это обещание так и не было исполнено. Как и обещание машинного перевода с естественных языков, которое было дано позднее, это было реализовано лишь в уточнении мелких деталей проблемной области.

Итак, где же компьютерному профессионалу взять надежный прогноз, если их не дают ни популярные, ни серьезные журналы?

Один из ответов, не очень приятный, состоит в том, что надо прочитать все предсказания, вооружиться изрядной долей сомнения и отложить это все, чтобы прочитать еще раз потом, – именно так, как мы это сделали только что. А до тех пор относиться к ним как к смутным видениям в магическом шаре, по которым образованный человек может сделать квалифицированную догадку о будущем, которое отнюдь не всегда соответствует нашим просвещенным представлениям.

Но есть и другой ответ. Он состоит в малодушном подходе к предсказанию будущего. Он порождает серию прогнозов, сделанных лишь ради повышения рейтинга предсказателя. Они не дают пищи для богатой фантазии, но зато опираются на знание незыблемого скучного прошлого.

Это прогнозы, сделанные Незадачливым пророком, который рисует для нас яркую картину далекого будущего, избегая малейшего риска:

1. Люди продолжают делать предсказания о компьютерах следующих поколений. Технические журналы продолжают публиковать все более точные обзоры компьютерного железа, которое ждет нас в будущем. И когда-нибудь эти предсказания сбудутся.
2. Производители компьютерного оборудования по-прежнему будут получать поздравления по поводу захватывающих дух повышений цен на свои продукты. А люди, как и раньше, будут предсказывать, что *оборудование* скоро будут давать бесплатно в нагрузку к *про-*

граммному обеспечению, а вслед за эпохой развязывания цен наступит эпоха их связывания.

3. Усовершенствования аппаратного обеспечения останутся практически незамеченными производителями ПО. Не будет заметных улучшений и в наборах команд и соглашениях об использовании регистров. «Ада-машины» и «Модула-машины» предстоящего десятилетия будут такими же разъевшимися на продажах бумажными тиграми, как «Фортран-машины» и «Кобол-машины» двадцатилетней давности.
4. Героем предсказаний все так же будет грядущее в ближайшем будущем автоматическое порождение программ мощными утилитами. Предсказатели будут так же ошибаться, как ошибались их весьма незаурядные коллеги и десять, и двадцать лет назад.
5. Некоторые представители научного мира по-прежнему будут строить и теорию и практику, основываясь на убеждении, что кто-нибудь когда-нибудь научится доказывать корректность программного кода. Некоторые специалисты-практики будут игнорировать доказательства корректности на том основании, что они слишком трудоемки, дорого обходятся и дают слишком неоднозначные результаты. Время докажет правоту практиков.
6. Менеджеры программных проектов будут настаивать на все большем увеличении объема документации, описывающей создаваемые программы на английском языке. Разработчики ПО будут составлять эту документацию, скрипя зубами и зная, что никто, кроме них, так и не узнает, соответствует ли она программному коду.
7. Для лекторов будут устраиваться роскошные презентации по языку Ада. Теоретики и превозносили этот сложный язык, и осмеивали его. Основные методики обучения испытают сильнейшее давление прагматических тенденций. Фортран и Кобол уцелеют и на этот раз.
8. Прилагательное «структурированный» все так же будет применяться к любому существительному, имеющему смысл в отрасли. И оно будет означать ровно то, что имеет в виду автор в тот конкретный момент. Презентации по структурным методологиям, устраиваемые для университетских преподавателей, будут уже не такими богатыми, как десять лет назад.
9. Некоторые из этих «безопасных» прогнозов, как и многие из более интересных и рискованных, окажутся неправильными. Однако едва ли кто-нибудь окажется настолько злонамеренным, чтобы хранить этот очерк на протяжении времени, достаточного для проявления истинности его тезисов!

Ах да, еще одно, последнее, предсказание, с которым никто не будет спорить:

10. В начале следующего года предсказатели, как по команде, опять представят нам букет своих пророчеств. А мы накинемся на эти пророчества так же жадно, как сейчас!

Список литературы

AMDA70 – Lowell D. Amdahl «Architectoral Questions of the Seventies». *Datamation*, январь 1970.

BENN70 – William Bennett «Decision Makers». *Computer Decisions*, январь 1970.

BROM70 – Howard Bromberg «Revolution 1970». *Datamation*, январь 1970.

DORN70 – Philip A. Dorn «The Onslaught of the Next Generation». *Datamation*, январь 1970.

HARR70 – Peter Harris «Decision Makers». *Computer Decisions*, март 1970.

PARR70 – Myron E. Parr «Decision Makers», март 1970.

SOL066 – Martin B. Solomon, Jr. «Are Small Free-Standing Computers Really Here to Stay?» *Datamation*, июль 1966.

YOUN70 – Chester Young «Decision Makers». *Computer Decisions*, март 1970.

Поддержка пользователей: все не так просто, как кажется

Звонит телефон. Опять звонит. Это еще один пользователь с еще одним вопросом о Продукте. У телефона сидит ответственный за поддержку пользователей Продукта – это вы, и предел ваших мечтаний, когда вы поднимаете трубку, – это что пользователь прочитал руководство. Однако у вас есть все основания полагать, что он этого не сделал!

Вы знаете, что работаете в крупной отрасли. У Ashton-Tate 42 представителя службы по работе с покупателями, и количество звонков, принимаемых за один день, составляет 1100. Майкрософт идет впереди: у нее 50 представителей и 1800 звонков. У WordPerfect 31 телефонная линия, по которым ежедневно приходит от 2500 до 3500 вызовов. Когда ваша компания только намеревалась создать Продукт и обдумывала способы вывода его на рынок, вряд ли она до конца понимала, насколько серьезным делом окажется поддержка пользователей. Оказа-

лось, что сопровождение ПО – это намного больше, чем просто сопровождение.

Возглавив службу поддержки пользователей в рамках Проекта, вы проделали собственное небольшое исследование этого вопроса. Теперь вы знаете, что за обслуживание звонков Living Videotext выкладывает 30–50 долларов в час. Вы знаете, что во многих компаниях масса звонков остается необработанной (хотя приведенная статистика впечатляет). А еще вы узнали, хоть эта новость вас и огорчила, что 70% звонящих не удосужились прочитать руководство!

Вы даже знаете, какое место в структуре компании занимает служба поддержки пользователей (и это довольно высокое место). В Lotus есть вице-президент службы информации и вице-президент издательской службы. В Ashton-Tate есть вице-президент по связи и составлению документации и вице-президент по информации и системным службам. Этот персонал, обеспечивающий службу поддержки пользователей, очень важен и стоит больших денег.

Вы ознакомились с вариантами поддержки пользователей. Вы узнали, что есть и другие варианты, кроме консультирования по телефону и руководств пользователей:

1. Справочная система (HELP). Программное обеспечение оснащается собственными механизмами поддержки пользователей. Этот вариант очень популярен во многих компаниях, особенно известны этим Lotus и Microsoft.
2. Форумы и доски объявлений. На доске объявлений пользователь может разместить пост с вопросом или идеей, а другие пользователи могут ему ответить. Многие компании, например Borland, или Software Publishing, или какая-нибудь другая, организуют чтение досок объявлений с определенной периодичностью, скажем каждые 36 ч, чтобы ответить на те вопросы, на которые не успел ответить никто другой.
3. Дилерская поддержка. Некоторые компании, в частности Software Publishing, перенесли основную часть поддержки пользователей на дилеров. Пользователи, обращающиеся за поддержкой в такие компании, просто перенаправляются к дилерам.
4. Пользовательские группы. Пользователи совместно обсуждают возникающие у них трудности и способы их разрешения. В некоторых компаниях считают, что этот способ сопряжен с некоторым риском. Запрос пользователя, расширившего свою компетенцию, может выйти за рамки информации, которую готова предоставить компания.

5. Консультативные советы. Люди, хорошо знающие отрасль, в которой специализируется заказчик, регулярно встречаются и вырабатывают рекомендации для пользователей по новым направлениям развития и использования продукта.
6. Информационные бюллетени. Они позволяют с определенной периодичностью распространять информацию о продукте.
7. Опросы пользователей. Результаты таких опросов могут быть как использованы внутри компании, так и доведены до сведения пользователей.
8. Справочники. Предоставляют пользователям (потребителям) информацию друг о друге.

И вы даже узнали о некоторых любопытных источниках информации, которые предназначены для углубленного изучения этого вопроса (вами или вашими консультантами). Институт корпоративного обучения в университете Вандербильта предлагает курс Consultation Skills (Мастерство консультации) продолжительностью в одну неделю, включающий практические ролевые занятия по обучению поддержке пользователей. Институт международных исследований размещает в *Software Maintenance News* рекламу трехдневного курса Software Support (Сопровождение ПО), посвященного консультированию по вопросам программного обеспечения. В литературе активно обсуждалось понятие информационных центров, которые, кроме всего прочего, представляют собой средства поддержки пользователей, работающих в штате организации, причем статьи на эту тему печатались и в периодических изданиях. Даже компании, работающие в сфере социальных услуг, печатают книги и проводят семинары на темы «Staying on Track Under Pressure» (Под давлением: как не потерять нить) и «Dealing Effectively With Difficult People» (Трудные люди – как правильно общаться с ними). (Вы для себя выяснили, что последний особенно полезен!)

Другими словами, как только вы стали отвечать за поддержку пользователей, вы стали относиться к ней, как к любой другой задаче, которую необходимо решить. Вы узнали, каков необходимый запас знаний в этой области, и поглотили столько этих знаний, сколько смогли. Что вас действительно удивило, так это огромный объем материала, обнаруженного после обескуражившего поначалу его отсутствия.

Однако все это сейчас уже неважно. Пока мы об этом рассуждали, телефон звонил непрерывно. Вы поднимаете трубку, и конечно же, в ней раздается горестный вопль пользователя.

«Как мне заставить работать функцию А?» – в голосе слышны нотки отчаяния.

«Об этом рассказано на странице 6 руководства пользователя», – отвечаете вы. «Вы его еще не прочитали?» – спрашиваете вы, стараясь не ругаться и не показывать, что, по вашему мнению, только полный болван мог не прочитать страницу 6.

«Да как-то еще нет, – говорит пользователь. – Но мне эта функция нужна прямо сейчас». Впечатление такое, что он сейчас заплачет.

Вы с пониманием улыбаетесь. Говоря следующие слова, вы изображаете терпение. «Ну что же, это делается так...», – начинаете вы.

По крайней мере, вы знаете, что покупатель заплатит компании 45 долларов в час за эту помощь.

Ретроспектива

Я с удовольствием рассказал вам в паре последних разделов о своих любимых аспектах программной инженерии: о технологии, инструментах и методиках. Однако в книге с названием «Программирование и конфликты» рано или поздно придется перейти к суровым реалиям управления программными проектами. В конце концов, большинство неприятностей, связанных с программными проектами, происходят именно в области менеджмента. То есть менеджмент гораздо чаще является причиной неудач программных проектов, чем технология. Анализ большинства проектов, вышедших из-под контроля, показывает, что раньше других происходят и имеют наибольшее значение те неприятности, которые обусловлены управлением.

Должен признать, конечно, что мой собственный вклад в развитие этой области надо считать до некоторой степени сомнительным. Я уже говорил, что на протяжении всей моей карьеры старательно избегал работы, связанной с управлением, и держался поближе к практическим вопросам технологии. Так что же я могу сказать по поводу роли плохого управления в неудачах программных проектов?

Вероятно, немного. И все-таки, как вы, безусловно, можете себе представить, во всех проектах, в которых я благополучно создавал программные продукты, менеджеры, оставаясь на втором плане, не переставали делать свое дело. В общем я, хотя и не руководил программными проектами, но имел массу возможностей наблюдать непосредственно и испытывать на себе подобное руководство!

Вам судить о полезности моих заметок о менеджменте и конфликтах. Читайте дальше.

Однако сначала я скажу пару слов о том, что в этих очерках о менеджменте актуально, а что нет. Предметом моей гордости по-прежнему яв-

ляется очерк «Покорение вершин индустрии ПО». Он охватывает вопросы, которые я считаю важными и по поводу которых никто в отрасли еще не высказывался.

Если анализировать ситуацию в отрасли без оглядки на время и стремиться к адекватным выводам, то в контексте современных представлений о вершинах и их покорении имеет смысл рассмотреть движение Open Source (Открытые исходные тексты). Может быть, именно из программистов, входящих в это движение, вырастет следующее поколение великих разработчиков ПО.

Немногим более десяти лет тому назад тема производительности безраздельно владела умами в индустрии ПО. Главный вопрос звучал так: «Как улучшить (может быть радикально) способность разработчиков генерировать решения?» Сейчас производительность, наверное, уже не так актуальна, как раньше, но я думаю, что она никогда не перестанет интересовать разработчиков. Я опасаясь, что, стараясь не отстать от времени, я мог немного переусердствовать, но ничто из сказанного здесь не устарело полностью.

Полагаю, что один из очерков этой главы вполне заслуживает того, чтобы его удалить. Лет десять тому назад я считал тему «Поддержка пользователей: все не так просто, как кажется» захватывающей, но сейчас она устарела так, что дальше некуда. В конце концов, центры обработки вызовов, где пользователи и получают поддержку, сейчас полностью переведены на аутсорсинг, за тридевять земель. Поэтому, если вы вообще читали этот очерк, считайте его срезом времени, запечатлевшим то, что должно было измениться до неузнаваемости!

Остается очерк «Какие прогнозы давали предсказатели раньше». Я отлично провел массу времени, иронизируя над прогнозами предсказателей судеб индустрии ПО. Большинство предсказаний так устарели, что стали смехотворными, и, конечно, именно поэтому я решил написать о них! Надо также сказать, что в свое время точность большинства из них была сравнима с попаданием пальцем в небо.

Надеюсь, что мне удалось удержаться в своих работах от соблазна предсказать будущее индустрии ПО – ведь в ней это самый быстрый способ сделаться посмешищем!

Глава 5 | Из лабораторий

ИССЛЕДОВАНИЯ

- Структурные исследования (немного ироничный взгляд)
- Проблема информационного поиска
- Неувязочка, или некоторые за и против (как будто больше не о чем!) ссылок

ПЕРЕДАЧА ТЕХНОЛОГИЙ

- А что в следующем году? Как исследуют развитие технологий
- Передача технологий ПО: процесс, обладающий многими недостатками, или неровная дорога к производительности
- Мифология передачи технологий

ОБРАЗОВАНИЕ

- Изучение ПО: новый источник информации
- Открытое письмо профессору информатики

Ретроспектива

ИССЛЕДОВАНИЯ

Структурные исследования (немного ироничный взгляд)

Есть одна тема, которую практически никто не хочет затрагивать, но я думаю, что пришла пора это сделать. Я назвал бы ее «кризисом исследований ПО».

Кризис исследований? Что это за кризис? Я слышу, как вы задаете эти вопросы.

Исследовательские проекты, как правило, имеют свойство не укладываться в бюджет, в сроки и давать ненадежные результаты. В этом и заключается кризис. Когда вы последний раз слышали об исследовательском проекте, бюджет которого строго контролировался, который шел по предсказуемому расписанию и дал результаты, достаточно надежные для того, чтобы можно было немедленно приступить к их практическому применению?

Моя задача отнюдь не в том, чтобы стенать по поводу этого кризиса. Напротив, у меня есть конкретная программа конструктивных действий.

Прежде всего, я думаю, что исследования пора структурировать. На самом деле я хотел бы провести структурную революцию в исследованиях.

Что я понимаю под структурированным исследованием? Для проведения исследований прежде всего нужен упорядоченный, строгий, прозрачный процесс. И никаких анархических, безответственных, «безусловных переходов». Никаких пережитков прошлого: неряшливости, бесконтрольности, вседозволенности и надежд на авось. Мы создадим жизненный цикл исследований, установим тщательно контролируемые вехи для отслеживания их результатов и определим комплект документации, которая обеспечит прозрачность хода исследований для руководителей.

И метрики исследований. Нам нужны способы измерения и результативности, и успешности исследовательских проектов. (Может быть, количество человеко-часов можно измерить с помощью количества строк опубликованных статей (Source Line Of published research Paper – SLOP).) Для того чтобы сравнивать исследования, которые мы будем проводить в рамках этой новой парадигмы, с неупорядоченными мероприятиями прошлого, начнем собирать эти метрики. Возможно, набор актуальных метрик – это самая насущная потребность в условиях кризиса исследований.

Ах да, еще нам надо дать определение исследовательскому процессу. Может быть, мы могли бы даже создать модель исследовательского процесса и разработать язык, который позволил бы нам описать его процедуры и отслеживать выполнение конкретного проекта, сравнивая его с этой моделью.

Что надо включить в эту модель? Прежде всего, элементы жизненного цикла исследования. Требования, предъявляемые к исследованию, мы определили бы с помощью формального, строгого, математического языка, чтобы облегчить понимание этих требований тем людям, которые открывают источники финансирования. Достаточно строгий язык может позволить нам заглянуть в будущее автоматического генерирования результатов исследований, опираясь на строгие описания требований.

И тогда мы сможем проектировать исследования. Может быть, у нас даже появится методология проектирования – так сказать, нотации Гейна–Сарсона (Gane–Sarson) или Йордона (Yourdon) для исследований – упорядоченная последовательность процедур проектирования. А разобравшись с проектированием, мы сможем представить его в виде набора структурных языков: диаграмм потоков идей (Idea Flow Diagrams – IFDs), показывающих потоки идей и процессы, которые задействуют эти идеи; структурных диаграмм исследований (Research Structure Charts – RSCs), представляющих иерархию исследовательских процессов; языков проектирования исследований (Research Design Languages – RDLs), которые помогли бы зафиксировать многочисленные мельчайшие подробности исследовательского проекта. Строгое и подробное представление исследовательских проектов может позволить нам даже без автоматического генерирования, опирающегося на спецификации требований, провести исследования силами лаборантов.

И наконец, практическое проведение исследований. Перечисленные выше меры по формализации процессов обеспечивают прозрачные и предсказуемые исследования. Мы создадим своего рода репозиторий – папки исследовательских проектов (Research Design Folders – RDFs), в которые будет помещаться все, связанное с проведением исследований и предназначенное для последующего использования всеми, кто заглянет в эти папки. Кроме того, мы организуем структурированный сквозной контроль исследований (Research Structured Walkthroughs – RSWs), процедуру, при помощи которой исследователи-коллеги смогут ознакомиться с методикой проведения исследований и высказать свои замечания. Мы подготовимся к тестированию исследований.

Возможны два подхода к тестированию структурных исследований. Традиционный подход, конечно, основан на применении тестовых данных, которые могут быть как структурированными, так и произволь-

ными (статистическими); при этом тестовые сценарии применяются к выполненным исследованиям и преследуют цель найти изъяны. Применяется также формальная верификация исследований, в ходе которой для отыскания ошибок и доказательства корректности результатов исследований задействуются математические методы. В любом случае тестирование проводится в соответствии со структурированными планами, а его результаты помещаются в отчеты о тестировании.

А сопровождение? Задача сопровождения при проведении исследований, конечно, неактуальна (а если ее и приходится решать, то она, по сути, такая же, как в процессе разработки исследования), и поэтому мы не будем определять самостоятельный процесс сопровождения исследований. (Заметьте, что только за счет этого мы уже сэкономили от 40 до 80% бюджета исследования.)

Итак, применив строгий подход к проведению исследований, мы наконец сможем взять исследователей под контроль. Мы сможем примерно определить время, необходимое для проведения исследования, опираясь на оценочное значение количества строк опубликованных статей, которые будут написаны по его результатам. Опираясь на соответствующие оценки, полученные в результате применения этих строгих структурированных подходов, мы сможем плотнее контролировать и пристальнее наблюдать за исследованиями и в конце концов разрешим кризис исследований.

Необходимо учесть еще одну грань современных подходов к исследованиям. В современных исследованиях очень важны личностные мотивы, при этом и с самой работой, и с публикациями о ней связаны интересы как организации, так и исследователя. Необходимо освободить исследования от личностных мотивов. Для того чтобы это сделать, будут проводиться регулярные обзоры фаз жизненного цикла исследования, которые позволят отслеживать его ход и менеджерам, и заказчикам, и коллегам-исследователям. Также будут проводиться эпизодические аудиторские проверки, которые помогут обнаружить недостатки, органически присущие скомпрометированным проектам. Результатом этих мероприятий будет командная методика проведения исследований, свободная от вредных проявлений эгоизма.

В общем, примерно так. Четко определенная программа исследований, должная дисциплинированность исследователей, жизненный цикл исследований с контрольными точками, позволяющими оценивать выполнение, обзоры и аудиторские проверки, обеспечивающие прозрачность процесса, и метрики, предназначенные для оценки конечных результатов, позволят нам взять под контроль эту трудноуправляемую сферу.

Да возрадуются исследователи всего мира! До структурной революции рукой подать, на повестке дня внедрение дисциплинирующей силы строгих формализованных методик, и кризис исследований скоро будет разрешен!

(Данное исследование финансировалось Международным теологическим обществом изучения исследований и других неуправляемых процессов, которому автор признателен за то, что эта работа стала возможной. Особенно важно, чтобы спонсоры не настаивали на применении представленных выше предложений к *данному* исследованию. Эти идеи, конечно же, предназначены для испытания на *других* исследователях и разработчиках ПО, а не на представителях элиты, таких как я.)

Проблема информационного поиска

Терпеть не могу усложнять простые вещи. Но иногда приходится.

Нехитрый процесс, который я хочу здесь обсудить, – это поиск информации. Как вы обычно поступаете, когда надо раздобыть дополнительные сведения по какой-либо конкретной теме? Допустим, вы занимаетесь изучением CASE-инструментов, вам известно, что за последние пять лет о них было много написано, и вы хотели бы прочитать какие-нибудь из этих публикаций.

По сути, это простая задача. Если у вас есть доступ к достаточно современной библиотечной системе, то вам достаточно запустить в ней функцию информационного поиска, чтобы найти ссылки в литературе на CASE-инструменты за последние пять лет. Современные библиотечные технологии довольно хорошо справляются с этой задачей. Вы получаете список аннотаций релевантных статей и, просмотрев их, выбираете, что будете читать. Куда уж проще.

Разве что есть одна неприятность. Из поля зрения средств поиска библиотечных систем выпадают два мощных информационных ресурса. Один из них – это правительственные отчеты. Второй – материалы, принадлежащие вендорам. В результате от вас ускользает половина той информации, которую вы должны были бы получить в идеальном случае по любой конкретной теме.

Если это правда, то почему до сих пор никто об этом не заявил? Ведь история применения информационно-поисковых систем насчитывает десятки лет.

Кто-то же должен был заметить эту зияющую дыру?

Это случай массовой близорукости. Если все ищут информацию одинаковым способом, то как кто-то сможет заметить, что чего-то не хватает?

Проблема станет очевидной, только если кто-то обнаружит источники, не попавшие в отчет о результатах поиска.

Приведу пару примеров.

Просмотрев литературу, посвященную программным метрикам, можно найти очень много материалов. Замечательные, захватывающие книги. Добротные, только что проведенные исследования. Новые прогрессивные идеи. Однако вы не найдете ни одной ссылки на отличную работу по метрикам, выполненную в Римском центре развития авиации (Rome Air Development Center). В течение 10 лет эта организация проводила исследования в области измерения качества программных продуктов. Они выпустили массу отчетов, многие из которых были ничуть не хуже тех, что печатались в более традиционных изданиях по информатике и вычислительной технике. Но мне не известна ни одна работа ни в одном ведущем журнале, автор которой ссылался бы на эти материалы. Все дело в том, что эти правительственные отчеты, как и многие другие, подобные им, просто не попадают в периодическую печать и не могут быть охвачены стандартными средствами информационного поиска.

Еще один пример. Несколько лет назад я написал обзорную статью, посвященную инструментальным программным средствам, в которой привел сводную таблицу этих инструментов и их функциональных возможностей. Меня тогда не покидало тревожное чувство, что упущено нечто важное. И постепенно я понял, в чем дело. Я собрал функциональные возможности инструментов, упомянутых в литературе, но не затронул наиболее распространенные инструментальные средства – те, которые распространяются производителями компьютерного «железа». Данные из этих источников попросту не попали в результаты ни одного из предпринятых мною информационных поисков.

Другими словами, в любой области исследователи, полагающие, что современные средства информационного поиска обеспечивают им полноценный доступ к релевантной информации, получают от этих систем совсем не то, что им кажется.

Вот в чем проблема. Сбор информации намного сложнее тех примитивных решений, которые доступны нам в настоящее время.

Какие же решения у нас есть? Во-первых, для получения сведений о соответствующих отчетах исследователю имеет смысл связаться с Национальной службой технической информации (NTIS – National Technical Information Service).

Но и этого недостаточно. Оказывается, некоторые правительственные документы, имеющие большое значение для нашей отрасли, не попадают в списки этой организации.

Во-вторых, исследователь должен знать о соответствующих продуктах и связаться с производителями продуктов. Сведения, которые предоставляют производители, содержат изрядное количество рекламной шелухи, которую необходимо отсеять, но если искать достаточно усердно, то всегда можно найти то, что хочешь узнать. Сложность не в отсеивании, а в том, чтобы количество затронутых источников позволило охватить исследуемый вопрос.

Что еще мы можем сделать? Мой ответ мне не нравится. Внимательно следите, кто ведет работу в интересующей вас области, и установите с ними контакт. Просматривайте списки правительственных документов. Не теряйте связи с производителями и поставщиками продуктов. Но вот досада: это трудные ответы, а вопрос казался таким простым. Установление всех этих связей и контактов увеличивает и без того большой объем трудной работы.

У меня нет простых ответов. Если они есть у вас, буду рад их услышать. Как ВЫ убеждаетесь в том, что ваш информационный поиск дал исчерпывающие результаты?

Неувязочка, или некоторые за и против (как будто больше не о чем!) ссылок

Нужны ли в статьях для профессиональных журналов ссылки на литературу? Казалось бы, ну о чем тут спорить?

Лично я думаю, что есть о чем. Попробую объяснить почему, рассмотрев для примера цитаты из двух книжных обзоров в мартовском выпуске *IEEE Computer* за 1989 год.

Автор одного из них обсуждает «утверждение, которое... необходимо подкрепить ссылкой на книгу или статью». В другом сделан вывод о том, что «автор предпринимает попытку скрыть дискуссию при помощи плотной научной завесы ссылок на другие публикации». Как говорится, куда ни кинь – всюду клин.

Так когда все-таки ссылки нужны, а когда – нет? В связи с этим у меня есть одна оригинальная мысль. Если вы пишете для профессиональных или отраслевых изданий, то не надо скупиться на ссылки; в популярных изданиях ссылки неуместны. Нет ни абсолютно правильного, ни абсолютно неправильного способа, все диктуется лишь принятыми нормами.

Оригинальность моей позиции в том, что, хоть я и убежден в уместности правильно данных ссылок (особенно для подкрепления точек зрения и выводов), я также уверен, что мы слишком уж активно пропагандируем их применение.

Возьмем для примера первую из приведенных цитат. Автор книги сформулировал вывод, а рецензент сказал, что этот вывод необходимо подкрепить. Однако в данном случае в качестве первичного источника информации выступает сам автор книги и он имеет полное право высказывать мысли, которые до него никому не приходили в голову. Вероятно, поэтому в книге нет ссылок. Если так, то с какой стати мы ждем, что автор будет на кого-то ссылаться? Есть и другой вариант рассуждений: не следует ли автору воздержаться от вывода, если последний нельзя подкрепить?

Буду краток. Ссылки важны, если обсуждение основано на других работах. Однако требовать, чтобы ссылки были там, где излагается новый материал, несправедливо, и, что еще хуже, это может погубить плодотворную идею.

А в нашей молодой отрасли плодотворные идеи ценятся на вес золота, мы не можем себе позволить потерять ни одну из них.

ПЕРЕДАЧА ТЕХНОЛОГИЙ

А что в следующем году?

Как исследуют развитие технологий

Вы когда-нибудь задумывались, сколько времени требуется новым идеям в индустрии ПО, чтобы реализоваться на практике?

Несколько лет назад этим вопросом задались два известных в нашей отрасли ученых, Сэм Редвайн (Sam Redwine) и Билл Риддл (Bill Riddle), которые работали в правительственном Институте оборонного анализа (Institute for Defense Analysis) и в частной компании под названием Software Design and Analysis соответственно.

Для удовлетворения своего любопытства Редвайн и Риддл избрали довольно строгий подход. Они выбрали примерно десяток технологий и определили, какие этапы проходит технология в процессе формирования. Они изучили официальные и неофициальные источники, чтобы узнать, когда именно каждая из этих технологий достигла каждого из этих этапов.

Для того чтобы рассматривать технологии на равных основаниях, ученые аккуратно подсчитали и проследили этапы развития каждой технологии вплоть до ее истоков. Вот эти этапы:

1. Рождение главной идеи.
2. Формулирование подхода к решению в основополагающей статье или в рамках демонстрационной системы.

3. Создание работоспособных функций.
4. Переход к применению за пределами группы разработки.
5. Надежные свидетельства значимости и пригодности.

Другими словами, отрезок времени, который изучали Редвайн и Риддл, должен был простирается от первых проблесков главной идеи до интенсивного практического применения.

К каким областям относились выбранные технологии? К разным: языки высокого уровня, структурное программирование, программная инженерия, UNIX. Всего их было 14. Многие из них еще не прошли все пять (0–4) этапов. Например, согласно Редвайну и Риддлу, искусственный интеллект, основанный на знаниях (Knowledge-based AI) достиг в своем развитии лишь этапа 2, а программные модели оценки затрат – этапа 3.

Но это была, так сказать, присказка, а где же сказка? Честно говоря, она грустная. Тем технологиям, которым удастся дойти до этапа 4, обычно требуется на это от 15 до 20 лет. Это означает, что технологии, которые сейчас начинают внедряться в практику, были задуманы на заре 1970-х. А идеи, роящиеся в головах специалистов сегодня, не станут применяться на практике, пока не будет перейден рубеж веков.

Впрочем, Редвайн и Риддл не были удовлетворены таким выводом. Выяснив, сколько времени все это занимает, они принялись придумывать и анализировать способы, которые позволили бы улучшить ситуацию. И придумали следующие потенциальные ускорители:

- *Концептуальная целостность.* Если идея хорошо продумана в самом начале, то ей удастся избежать противоречий, которые в противном случае замедлили бы ее развитие.
- *Четкое понимание потребности.* Потребность, наличием которой вызвано появление идеи, должна быть хорошо понята и четко обозначена.
- *Адаптируемость.* Идея должна предусматривать возможность модификации в соответствии с другими потребностями, которые связаны с исходной.
- *Предшествующий положительный опыт.* Очень помогают свидетельства положительного опыта применения смежных технологий.
- *Поддержка менеджмента.* Руководители, отвечающие за реализацию идеи, должны способствовать ее развитию.
- *Обучение.* Необходимо обеспечить своевременное обучение пользователей.

И даже если выполняются все эти благоприятствующие условия, срок практической реализации идеи всего лишь смещается к нижней границе этого 15–20-летнего интервала. Заметное ускорение, обусловленное какими-то другими факторами, возможно лишь в чрезвычайных обстоятельствах.

Между прочим, за подтверждениями справедливости этих выводов далеко ходить не надо. Цифры такого же порядка были получены и при исследованиях для других дисциплин.

Так что если вам не дает покоя новая идея (по-настоящему новая) и вы подумываете о том, чтобы в следующем году вложить в нее серьезные деньги, то смысл всего сказанного ясен. Не торопитесь задерживать дыхание. Согласно Редвайну и Риддлу, созревание технологии обычно требует намного больше времени, чем принято думать. (Те, кто заинтересуется полным текстом статьи, найдут ее в материалах проходившей в Лондоне с 28 по 30 августа 8-й международной конференции по программной инженерии (Proceedings of the 8th International Conference on Software Engineering, held in London August 28–30, 1985). Проведение конференции и издание материалов спонсировал институт IEEE.)

Передача технологий ПО: процесс, обладающий многими недостатками, или неровная дорога к производительности

Любой специалист, занятый в индустрии ПО, хорошо знает, насколько чудовищны темпы ее развития и что это оборачивается нехваткой квалифицированного персонала.

Благодаря этому знанию любой, кто работает в индустрии ПО, хорошо понимает значение передачи технологий. Нехватку квалифицированных кадров можно компенсировать, совершенствуя их навыки, создавая более мощные инструменты поддержки и повышая производительность труда.

Исходя из этого передача технологий, по моему мнению, превращается в обширное поле боя, на котором очень трудно или даже невозможно добиться успеха, и это положение не изменится.

Почему я так считаю? Почему я представляю технологию ПО как поле боя, а не как совместные поиски путей к прогрессу, как это должно быть?

Чтобы ответить на этот вопрос, я должен нарисовать идеализированную картину передачи технологий (рис. 5.1):

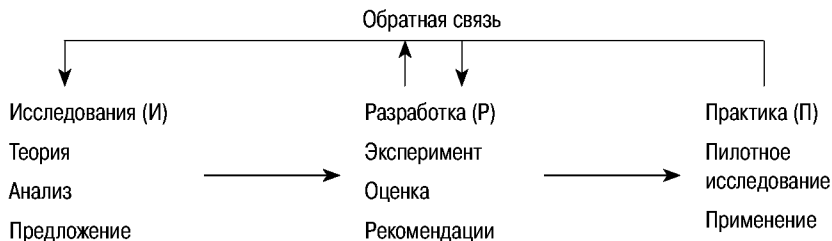


Рис. 5.1. Идеализированная картина передачи технологий

В этой идеализированной картине действуют три посредника: исследователь И, разработчик Р и специалист-практик П. Исследователь создает теорию, анализирует результаты и предлагает новые концепции. Разработчик экспериментирует с этими концепциями, помещая их в окружение, которое занимает промежуточное положение между своим окружением и средой специалистов-практиков, оценивает результаты эксперимента и рекомендует технологии, которые заслуживают того, чтобы организовывать их передачу. Практик проводит дальнейшую проверку пригодности технологии, запуская пилотные исследования, и находит практическое применение тем технологиям, которые преодолели все барьеры.

Пока что на приведенной выше диаграмме информационные потоки идут слева направо. Необходимо замкнуть цикл передачи информации. Исследователь должен узнать у практика и разработчика, какие теории удалось реализовать, какие нет и почему. Кроме того, общаясь с практиками, исследователь должен почерпнуть у них идеи для новых теорий и исследований.

Я уже сказал, что это идеализированная картина передачи технологий. Какие препятствия мешают реализации идеальной картины в современном процессе передачи технологий? Я бы сказал, что на практике реализуется весьма незначительная часть представленной схемы.

Вот что можно сказать, по-моему, об отдельных компонентах:

1. *Исследовательская работа.* Этот компонент процесса передачи технологий в полном порядке. Создаваемые теории жизнеспособны, проводимые анализы тонки и глубоки, предлагаемые концепции многочисленны. Правда, исследователи склонны поддаваться прихотям и фантазиям, поэтому расходуется слишком много сил, а спектр направлений, в которых проводится исследования, слишком узок. Однако в целом исследования как самостоятельный компонент справляются со своей задачей.

2. *Разработка.* Это первый серьезный изъян в процессе передачи технологий. В нашем техническом социуме очень немногие находят интерес в том, чтобы заниматься разработкой. В результате этот важнейший мост между исследованиями и практикой не функционирует, и теории не удается привести его в работоспособное состояние. (Интересно отметить, что в японском техническом социуме значительно меньший упор делается на исследования и намного больше внимания уделяется разработке. Возьму на себя смелость утверждать, что именно этому Япония обязана таким внушительным техническим прогрессом в прошлом веке.)
3. *Практика.* Этот компонент процесса передачи технологий также в полном порядке. Спорная точка зрения. Дискуссии о состоянии дел в практике фокусируются на длинном заунывном повторении: «Программные проекты никогда не выполняются в срок, всегда выходят из бюджета, программное обеспечение ненадежно». Однако на самом деле программисты-практики решали задачи исключительной сложности при помощи технологий, которые признавались большинством как наисложнейшие из всех, которые когда-либо реализовывались человечеством, и таково было положение дел в практике через каких-то 30 лет после ее начала! Действительно, практики избегают риска и поэтому не очень охотно применяют новые технологии, но если новая технология неотразима (а это случается постоянно), то она легко преодолевает этот барьер.

При почти полной атрофии функции Р канал связи, направленный на диаграмме слева направо, безнадежно испорчен. Однако на самом деле все еще хуже. Линия обратной связи (справа налево) в верхней части диаграммы также практически отсутствует. По разным причинам очень немного информации просачивается от практиков (П) обратно к исследователям (И). Одна из этих причин состоит в том, что первых раздражает практическая наивность вторых и они считают, что разговаривать с ними бесполезно. Вторые же считают первых отсталыми, и потому им неинтересно разговаривать с ними. (Выбор слов «бесполезно» и «неинтересно» неслучаен. Практик исповедует принципы, в соответствии с которыми он работает только с тем, что приносит пользу. Согласно принципам исследователя работать можно только с тем, что интересно. Там, где сталкиваются жизненные принципы, взаимопонимание и взаимосвязь, как правило, нарушаются).

Таким образом, если неучастие разработчика (Р) наносит очевидный и сильный урон процессу передачи технологии, то отсутствие взаимосвязей между практиком (П) и исследователем (И) оказывает на этот процесс еще более фатальное действие. Попытки осуществить передачу технологий невзирая на эти изъяны вылились в баталии, описанные мною выше.

Я бы сказал, что передача программных технологий будет по большей части терпеть неудачи до тех пор, пока звено Р на этой диаграмме будет отсутствовать. Но даже если решить проблему звена Р, взаимосвязь между П и И все равно необходимо наладить, иначе процесс передачи технологий обречен. (Отмечу, что включение звена Р в цепочку способно помочь наведению моста П → И.)

Как же разрешить это затруднение? В первом приближении решение очевидно. Мы должны создать звено Р (разработчиков). Мы должны создавать взаимосвязь П → И, и некоторую роль в этом процессе должно играть устранение барьеров, обусловленных различием образа мышления.

Эти решения легко пропагандировать, но трудно осуществить. Что мы должны сделать, чтобы общество было заинтересовано в росте звена Р? Что надо сделать, чтобы наладить взаимосвязь между двумя вооруженными и конфликтующими группами? Эти проблемы имеют не только чисто техническую природу, но и (в не меньшей степени) социальную, и они очень трудны.

Я нашел свое собственное решение. Проведя в отрасли почти 30 лет (сначала в качестве представителя звена П, затем – во все большей степени – звена Р), я несколько лет тому назад начал смещаться в сторону университетской науки. Очевидно, что образование и обучение составляют суть любых реализуемых решений, и я горжусь, что в качестве преподавателя принял участие в образовательной программе «Эксперт в области разработки программного обеспечения» (Master of Software Engineering – MSE) в университете Сиэттла.

Номинальная цель программы MSE состояла в том, чтобы помочь действующим практикам (звено П) усовершенствовать свои умения и навыки. Однако лично я преследовал цель помочь тем практикам (представителям звена П), которые этого хотели, стать разработчиками (перейти в звено Р). Образование, которое дает университет Сиэттла, содержит ядро кристаллизации знаний, необходимых разработчикам. Я горячо верю в мощный потенциал этой системы.

Однако недостаточно организовать переход звена П в звено Р. Некоторым представителям звена И также придется совершить этот переход. Я полагаю, что в нашей стране слишком многие хотят заниматься исследованиями, и надеюсь, что в университетском научном сообществе найдутся люди, которые смогут правильно оценить то, что я здесь говорю, и они помогут студентам, нацеленным на исследования, но интересующимся разработкой и делающим в ней успехи.

Иначе говоря, для успешной передачи технологий должно выполняться несколько условий:

1. Необходимо создать звено Р, чтобы навести мост через пропасть, которая разделяет теорию и практику. Это звено должно пополниться людьми из мира как науки, так и практики.
2. Исследователи должны прислушиваться к практикам. Если какая-то теория хороша на бумаге, это еще не значит, что она будет работать на практике. Опыт практиков, который они вкладывают в решение задачи, имеет ценность.
3. Практики должны прислушиваться к разработчикам. Если какая-то теория хороша на бумаге, то это еще не значит, что она *не будет работать* на практике! Знания исследователей и разработчиков, которые они вкладывают в решение задачи, имеют ценность.

Только согласованные действия *всех* участников процесса передачи технологии дают этому процессу и сопутствующему приращению производительности шанс на успех.

Мифология передачи технологий

Не так давно в Санта-Фе прошел семинар по передаче программных технологий. На конференции страсти разгорелись вокруг вопроса «Как взять лучшее от компьютерных исследований и найти их результатам практическое применение?»

Это, безусловно, один из главных факторов производительности ПО. Но это очень сложный вопрос. Прежде всего, что значит «лучшее» применительно к компьютерным исследованиям? Как отсеять «худшую часть» компьютерных исследований и не потерять время на попытки претворить это худшее в практику? Как побудить практиков испробовать что-то новое, имея в виду реальность таких проблем, как риск, расходы и необходимость все делать точно в срок?

Один из докладчиков, Патрик О'Брайен (Patrick O'Brien) из Digital Equipment, показал сложность вопроса, подойдя к его рассмотрению с другой стороны. Он говорил о «мифологии передачи технологий» (его собственная формулировка) и представил на обсуждение семь мифов. Вот они (мифы О'Брайена, формулировка обоснований принадлежит мне):

- *Миф 1.* Группа, ведущая исследования в корпорации, несет полную ответственность за инновации.

Неправильно! Если прикладной практик не будет постоянно искать новые и более совершенные способы делать свою работу, то он упустит важнейшие перспективные идеи. Ответственность за инновации нельзя делегировать.

- *Миф 2.* Поскольку исследовательская группа и группа разработки так много должны выиграть от сотрудничества, то они будут с радостью действовать совместно, чтобы процесс передачи технологии прошел гладко.

Тоже неправильно! Это не политика, это синдром «изобретено не здесь» (Not Invented Here – NIH). Есть такие люди, которые ни за что не будут сотрудничать, если только не запереть их вместе и не выпускать, пока они не начнут взаимодействовать.

- *Миф 3.* Великие инновации (которые действительно влияют на компанию) являются результатом настоящих технологических прорывов.

Может быть, да, а может быть, нет. Однако маркетинг может породить блестящую новую концепцию продукта, а производство – отыскать меры экономии, которые сделают воздушный замок вполне реализуемым продуктом. Инновация может произойти в любом месте.

- *Миф 4.* Техническое превосходство и стратегическое значение инновации гарантируют, что она будет принята.

Интересно отметить, что это просто неправда. Своевременность – совпадение готовности полезного продукта и готовности рынка к его принятию – это, пожалуй, единственный значимый фактор, который определяет коммерческий успех или провал продукта.

- *Миф 5.* Лучший способ известить других о значении своего исследования состоит в том, чтобы рассказать о его результатах в печатной работе. Железная логика ваших аргументов убедит всех принять новую технологию.

Как же далеко вы можете зайти в своем заблуждении?! Некоторые люди читают специальную литературу, некоторые – нет. Те, кто более всего способны помогать вашему успеху, возможно, ее не читают. Писать статью о том, что вы сделали, может быть очень приятно, и это может укрепить ваше реноме как автора, но для того чтобы сделать исследование значимым, одной только печатной работы недостаточно.

- *Миф 6.* Лучший способ создать эффективную исследовательскую лабораторию состоит в том, чтобы нанять самых лучших и талантливых ученых.

Ну да, это так. Но ученые должны понимать принципы, определяющие важность исследования. И они должны понимать, как на-

водятся мосты через пропасть, которая отделяет науку от практики. Эти качества не всегда входят в традиционное понятие «лучший и талантливый».

- *Миф 7.* Если построить сахарный дом, то от гостей отбоя не будет.

Точно, но только предполагаемые гости должны знать, что дом сахарный и где его найти.

Какой вывод делает О'Брайен? Исследование должно быть дополнено анализом рынка. Исследователям необходимо:

- Установить связи с внешним миром
- Установить в качестве цели и основы своего исследования удовлетворение некоей потребности
- Превратить в товар идеи, которые появятся в результате исследования
- Передать результаты своих исследований тем, кто в них нуждается
- Встроить процесс передачи технологии в структуру своей организации

Может быть, у них и не получится кристально «чистое» исследование. Но зато намного возрастет вероятность того, что оно даст практические результаты. А в вашей компании исследователи тоже так думают?

ОБРАЗОВАНИЕ

Изучение ПО: новый источник информации

Что надо читать, если вы хотите больше узнать о программном обеспечении? На этот вопрос есть несколько хороших ответов.

Во-первых, можно заглянуть в учебники, которых по этой теме предостаточно. Есть отличные книги по программной инженерии, написанные такими авторами, как Роджер Прессман (Roger Pressman) и Ричард Фэйрли (Richard Fairley). Есть превосходные книги, представляющие собой сборники рассказов или очерков, например «Мифический человеко-месяц» Фредерика Брукса¹ и «Человеческий фактор» Демар-

¹ Brooks, Frederick P., Jr. «The Mythical Man-Month». Reading, MA: Addison-Wesley, 1975 (Фредерик Брукс «Мифический человеко-месяц, или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000).

ко и Листера¹. Есть мастерски написанные специальные книги, например «Искусство тестирования ПО» Гленфорда Майерса². В конце концов хорошие книги становятся доступными.

Во-вторых, есть журналы. Много полезной информации для руководителей информационных отделов содержится в журнале *System Development*. Есть журнал *Software – Practice and Experience*, который издается в Англии и очень полезен и для теоретиков, и для практиков. То же самое можно сказать о журнале *Journal of Systems and Software*, который редактирую я и который издается в Нью-Йорке издательством Эльзевир (Elsevier). Еще есть журналы, выходящие под эгидой ACM и IEEE, например *Communications of the ACM* и *IEEE Transactions on Software Engineering* и *IEEE Software*. В ACM даже образовалась группа, которая объединила профессионалов из сферы программной инженерии и которая издает небольшой и едкий нерецензируемый журнал под названием *Software Engineering Notes* – можно сказать, лучший в своем роде.

И еще, конечно же, есть курсы. Если мы с вами похожи, то вы не успеваете прочитать всю приходящую на ваше имя почту, в которой рекламируются курсы и семинары.

Было бы здорово, если бы у нас был журнал, аналогичный *Consumer Reports*, в котором бы приводился сравнительный анализ стоящих курсов, но такого журнала, к сожалению, нет. Отдавая за курсы деньги, вы покупаете кота в мешке. Некоторые преподаватели курсов располагают превосходным материалом, отлично им владеют и умеют его подать, а некоторые не располагают, не владеют и не умеют.

Однако появился новый источник информации обо всем, что связано с программированием и программами. Я говорю об учебных модулях, которые предлагает питсбургский Институт программной инженерии (Software Engineering Institute – SEI). Каждый из этих модулей содержит полную дозу информации о какой-либо из граней мира программных продуктов, снабженную ссылками на дополнительные материалы, и они представляют собой неплохую отправную точку для умеренно искушенного читателя, желающего начать изучение какой-либо конкретной темы. А что лучше всего – они бесплатные!

¹ DeMarco, Tom, and Timothy Lister «Peopleware: Productive Projects and Teams». 2d Ed., New York: Dorset House, 1999 (Том Демарко и Тимоти Листер «Человеческий фактор: успешные проекты и команды», 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2005).

² Myers, Glenford J. «The Art of Software Testing». John Wiley & Sons, Inc. 1979 (Гленфорд Майерс «Искусство тестирования программ». – Пер. с англ. – М.: Финансы и Статистика, 1982).

Что такое учебный модуль? Институт программной инженерии (SEI), кроме других своих задач, старается предоставить материал в помощь преподавателям-методистам. И учебные модули – это система, которая позволяет им решить данную задачу. В них на нескольких страницах, от 15 до 45, описываются актуальные вопросы по конкретной предметной области и говорится, где можно найти дополнительные материалы.

А-а, так они для преподавателей? А вы не преподаватель?

Да, эти модули предназначаются для университетских профессоров или инструкторов, работающих в отрасли, но вы можете совершенно свободно получить копию такого модуля для себя лично, чтобы глубже изучить какую-либо конкретную тему, связанную с программами и программированием. Допустим, вас интересуют требования к программам и системный анализ. Институт SEI предлагает соответствующие учебные модули:

- *Обзор спецификаций требований.* Охватывает вопросы, связанные с пользователями и их потребностями. А также вопросы, связанные с прикладными областями, разбитыми на категории. В нем представлены вычислительные модели, такие как конечные автоматы и вычислительные графы.
- *Формальная спецификация ПО.* Здесь рассказывается, что такое формальные спецификации и каковы принципы, лежащие в основе этого понятия. Обсуждаются примеры и процесс создания формальных спецификаций.
- *Спецификации ПО: структура.* Раскрывает неоднозначность термина «спецификация». Представляет разновидности спецификаций, которые может составить программист. Описывает способы оценки спецификаций.

Кроме того, в процессе подготовки находится другой соответствующий учебный модуль:

- *Анализ требований.* В нем описываются практические аспекты задач, связанных с формулированием требований к программным системам.

И еще один, авторы для которого пока не найдены:

- *Языки спецификаций.* Обзор конкретных языков формальных спецификаций, их сильных сторон и способов применения.

Из вышесказанного вы, наверное, поняли, что учебный модуль не дает такого широкого охвата темы, какой можно было бы ожидать от курса в университетском колледже или на профессиональном семинаре. Он позволяет глубже заглянуть в более узкую область и в комбинации

с другими учебными модулями может послужить для построения отдельного курса или семинара. Так что модуль позволяет рассмотреть конкретную узкую тему, а для того чтобы увидеть более широкую картину, надо объединить несколько модулей.

Итак, вы получили представление об учебных модулях, а вот их список, постоянно пополняемый институтом SEI: «Обзор спецификаций требований» (Requirements Specification Overview), «Введение в проектирование ПО» (Introduction to Software Design), «Процесс технической экспертизы ПО» (The Software Technical Review Process), «Управление конфигурацией ПО» (Software Configuration Management), «Защита информации» (Information Protection), «Безопасность ПО» (Software Safety), «Обеспечение качества ПО» (Assurance of Software Quality), «Формальная спецификация ПО» (Formal Specification of Software), «Модульное тестирование и анализ» (Unit Testing and Analysis), «Модели эволюции ПО: жизненный цикл и процесс» (Models of Software Evolution: Life Cycle and Process), «Спецификация ПО: структура» (Software Specification: A Framework), «Метрики ПО» (Software Metrics), «Введение в верификацию и валидацию ПО» (Introduction to Software Verification and Validation) и «Защита интеллектуальной собственности применительно к ПО» (Intellectual Property Protection for Software).

Ширина охвата тем ограничивается лишь рамками профессиональной сферы.

Кто же пишет эти модули? Обычно этим занимаются профессора из университетских колледжей или эксперты в конкретных областях. Иногда это делают профессионалы высочайшей квалификации. Например, профессор Нэнси Ливсон (Nancy Leveson) из университета Калифорнии в Ирвине написала модуль по безопасности ПО. Вероятно, она знает о безопасности ПО больше, чем кто-либо другой. Профессор Памела Сэмюэльсон (Pamela Samuelson) из университета Питсбурга — юрист, и она написала модуль по защите интеллектуальной собственности применительно к ПО. Одна из ее работ на стыке права и ПО привела к недавним важным изменениям в правительственной политике закупок и поставки ПО. Очень может быть, что модуль, которым вы интересуетесь, будет написан кем-то, о ком вы слышали.

Может быть, вы думаете, что тут кроется какой-то подвох? Что, можно просто попросить и получить бесплатно модули, написанные лучшими профессионалами? Никакого подвоха. Институт SEI финансируется правительством, поэтому его материалы должны становиться достоянием гласности. Конечно, модули, пожалуй, пересыщены компьютерной наукой, и потому иногда в них больше теории и всяческих непонятностей, чем хотелось бы практикам. Однако, если вы преодолеете

это препятствие, то найдете новый, причем хороший, источник знаний о программах и программировании.

Для того чтобы получить более подробную информацию, например полный список модулей, доступных в настоящее время, обратитесь к Эллисон Брюнванд (Allison Brunvand) из института SEI (Software Engineering Institute, Carnegie-Mellon University, Pittsburgh PA 15213).

Открытое письмо профессору информатики

Этот очерк написан в форме открытого письма профессору информатики от специалиста по сопровождению ПО (и тот и другой представляют собой собирательные образы). Я предназначал его двум совершенно различным типам людей: тем практикам, которые хотят узнать немного больше о месте сопровождения ПО в общей панораме индустрии ПО, и тем компьютерным ученым-преподавателям, которые не понимают, какую важную роль играет сопровождение ПО в панораме компьютерной науки.

Я надеюсь, что независимо от того, к какой группе вы относитесь, вам понравится мое письмо, написанное от всей души.

Дорогой профессор информатики!

Трудно переоценить те усилия, которые вы затратили, обучая меня всему, что вы знаете о компьютерных науках. Об особенностях языков программирования и о выборе структур данных, об алгоритмах и о нотации big O, об операционных системах и компиляторах, и даже немного о программной инженерии...

Ну что ж, у меня для вас сюрприз! Я, квалифицированный специалист по разработке программного обеспечения, занимаюсь сопровождением. И я жалею, что не подготовлен лучше...

Я понимаю, дорогой профессор, вы думаете, что в сопровождении нет ничего особенного... что, обучая нас разработке, вы научили нас всему, что стоит знать. Я говорил с другими профессорами информатики, и они думают так же!

Но вы ошибаетесь. В сопровождении ПО еще много неизвестного, и мне до сих пор приходится много учиться. Я расскажу вам о том, чему я успел научиться.

Проектирование для сопровождения

Оказывается, программы надо писать так, чтобы их было легче сопровождать, — это разработчики делают в интересах специалистов по со-

провождению, и вы просто должны были мне рассказать об этом на своих занятиях.

Есть, к примеру, пара ключевых принципов, которым должны следовать все разработчики, – это принцип *управления из одной точки* и принцип *оборонительного проектирования*.

О принципе *управления из одной точки* вы упоминали, хотя и не называя его. Идея управления из одной точки состоит в том, что любые действия, выполняемые в программе (или с документацией, или с данными) в нескольких местах, реально должны выполняться в каком-то одном месте, на которое в остальных точках размещаются ссылки. Примеры реализации принципа управления из одной точки:

- *Модульность*. Все основания для модульного программирования, о которых вы нам рассказывали, – это частные случаи общего принципа – управления из одной точки.
- *Абстракция данных*. То же самое: все основания, по которым описания данных надо отделить от выполняемых над ними действий (и этому вы тоже нас учили), чтобы легче было ссылаться на данные и проверять их, входят в принцип управления из одной точки.
- *Объектное ориентирование*. И опять то же самое. Этому вы нас тоже учили, принципу абстракции данных с наследованием характеристик, но вы не связали этого ни с принципом управления из одной точки, ни с сопровождением.
- *Проектирование, основанное на применении таблиц и файлов*. Если данные, с которыми работает программа, хранятся централизованно, то их неизбежные изменения затрагивают только описания данных, но не их обработку в процедурах программы.
- Как говорят продавцы и маркетологи, «и многое, многое другое». Принцип управления из одной точки весьма и весьма многообразен. В качестве примера можно привести именованные константы и нормализованные базы данных, как, впрочем, и продуманную структуру документации.

А принцип *оборонительного проектирования* вы едва затронули. Оборонительное проектирование – это совокупность мер, которые принимает разработчик (не перекладывая эту задачу на специалиста по сопровождению), чтобы сделать программу способной восстанавливаться в случае нештатных ситуаций и сбоев. Примеры оборонительного проектирования:

- *Обработка исключений*. Я узнал, что во многих программах (более чем в 50%) предусматривается обработка сбоев, которые не должны случаться при их штатной работе, но время от времени все-таки

происходят. Эти обработчики исключений, которые представляют собой блоки кода, реагирующие на ошибочные данные или другие предвосхищаемые разработчиком обстоятельства, составляют значительную часть его труда.

- *Утверждения.* Программа может проверять саму себя во время работы на предмет возникновения не только предвидимых исключений, но и программных или аппаратных сбоев. Утверждения – это встроенные обработчики ошибок, содержащие код, который корректирует обнаруженные сбои.
- *Переусложненное проектирование (overengineering).* Сопровождение ПО, втиснутого в маломощные, медленные компьютеры, может потребовать в два раза больше времени на более совершенных и продуманных программно-аппаратных системах. Много неприятных сюрпризов для специалиста по сопровождению таит в себе программа, которая аварийно завершается после попытки ввода 15 позиций данных в таблицу, рассчитанную на 14 позиций. Разработчик способен предвидеть проблемы и планировать рост сопровождения.
- *Отказоустойчивость.* Периодические отказы ПО неизбежны и случаются независимо от того, насколько тщательно мы удаляем ошибки. Существует целый арсенал средств, которые могут применяться, если ПО должно работать безотказно. Этот арсенал носит название *отказоустойчивости* и включает очень специальные применения диверсификации и избыточности.

Сопровождение

Однако специальное изучение сопровождения ПО, дорогой профессор информатики, не ограничивается тем, что могут делать *разработчики*. *Специалисты по сопровождению* должны также выполнять и совершенно уникальные функции.

К примеру, среди программ, относящихся к инструментальным средствам, есть целые комплекты инструментов, предназначенных исключительно в помощь специалисту по сопровождению ПО. Их так много, что их начали классифицировать, и Администрация общих служб США (the United States General Services Administration) отобрала комплект готовых инструментов, назвав его «The Programmers Workbench». Эти инструменты позиционируются как средство «повышения производительности в управлении программными проектами, особенно в сфере сопровождения ПО».

Инструменты разбиты на следующие категории:

- Анализаторы зависимостей. Средства составления таблиц перекрестных ссылок, структурные анализаторы вызовов, стандартизаторы имен наборов данных и детекторы волнового эффекта (Ripple Effect Detectors), которые автоматически генерируют таблицы, содержащие информацию, призванную помочь специалистам по сопровождению понять работу сопровождаемого ПО.
- Средства обратной разработки. Такие инструменты, как «антипроектировщики», анализаторы метрик и производительности, помогают понять суть и назначение функций ПО, скрытые между строк стандартной документации.
- Анализаторы изменений ПО. Это компараторы, средства управления конфигурациями и составления отчетов об изменениях, помогающие специалистам по сопровождению управлять изменениями сопровождаемого ПО.
- Реконструкторы. Структуризаторы кода, переформатировщики и условные компиляторы помогают специалистам по сопровождению собрать воедино измененный программный продукт по окончании сопровождения.

Управление сопровождением

Все эти вопросы, дорогой профессор, охватывают лишь крохотную часть *технологии* сопровождения ПО. Но кроме этого есть совершенно уникальные знания, которыми обязан владеть специалист по сопровождению ПО.

Как, к примеру, надлежит *организовать* управление сопровождением? Дело не сводится только к вопросу «Должны ли разработка и сопровождение находиться в разных ведомствах?» (между прочим, на этот вопрос нет общего правильного ответа). Вот что еще необходимо знать:

- Кто принимает решения о том, какие изменения и когда необходимо внести? (Ответ: эти решения должен принимать *совет по управлению изменениями* в проекте (Change Board).)
- Кто отвечает за принятие решений по тестированию продукта и выпуск релизов? (Ответ: нередко это делает орган, называемый *прямочным контролем* (Product Test).)
- Кто гарантирует, что продукт не подвергнется непоследовательным изменениям или что он не будет безвозвратно утерян? (Ответ: *орган по управлению конфигурациями* (Configuration Management).)

Понимаете, дорогой профессор? Существуют целые классы организаций, необходимые для сопровождения.

Есть и еще кое-что. Я знаю, дорогой профессор, что в курс нашего обучения практически не попали вопросы, связанные с ролью человека в компьютерных науках, но человеческий фактор важен в сопровождении ПО. Так, не все люди одинаково хорошо справляются с сопровождением ПО. Некоторые не способны реализовывать новшества в рамках уже готового программного продукта (или просто предпочитают этого не делать). Они пытаются исправлять код, не понимая программу как единое целое. Они недостаточно терпеливы, гибки, пытливы и вдумчивы, и у них не слишком хорошо получается думать, приняв точку зрения кого-нибудь другого (например, разработчика). Поэтому нельзя поручать сопровождение неопытным и посредственным – эта задача сама по себе особенная и сложная.

Билл Гейтс (основатель и глава Microsoft) – человек, известный в компьютерном мире, – считает овладение технологией сопровождения лучшим способом постижения компьютерной науки. Он цитируется в книге Сьюзен Ламмерс «Programmers at Work» (Как работают программисты): «...[самый] лучший способ подготовиться [к работе программистом] состоит в том, чтобы писать программы и изучать хорошие программы, написанные другими... Я выуживал листинги операционных систем из мусорных корзин вычислительного центра», – говорит он, вспоминая, как изучал программирование в колледже.

Защитники программ всего мира

Дорогой профессор, я участвовал в конференции, которую ежегодно устраивает Ассоциация специалистов по сопровождению ПО (Software Maintenance Association), и в других аналогичных мероприятиях, где пообщался с лучшими из представителей в этой области (их иногда называют «защитниками программ всего мира»). (На них собираются люди, которые живут и дышат проблемами сопровождения ПО!) И я узнал, что в компаниях, где действительно понимают важность сопровождения ПО, для этих специалистов создаются самые благоприятные условия. В этих компаниях специалисты по сопровождению обычно делают следующее:

- Они ясно понимают свои функции и значение. Специалист по сопровождению не только (и не столько) исправляет ошибки, львиную долю своего времени он посвящает улучшению и усовершенствованию ПО.
- Они понимают важность своей роли. Они поддерживают основное ПО своей компании в работоспособном состоянии. Зачастую это ПО представляет собой краеугольный камень, скрытый механизм, работа которого определяет будущее компании, и уж конечно, по нему заказчики судят обо всех программных продуктах компании.

- Их труд, если он успешен, хорошо вознаграждается. В некоторых компаниях лучшие специалисты по сопровождению награждаются ценными подарками и сувенирами (подарочными сертификатами, почетными значками, бесплатными обедами), получают карьерные привилегии (особые наименования должностей, например applications support team leader (руководитель группы сопровождения прикладного ПО), или бюрократическую поддержку при выполнении рутинных заданий), или корпоративный авторитет («лучшие и самые талантливые работники компании занимаются сопровождением ПО»).
- Они применяют новейшие инструменты и технологии, о которых я говорил раньше.
- Они имеют налаженный контакт со своими пользователями (задействуя для этого пользовательские сообщества, пользовательские информационные бюллетени, консультативные советы пользователей или опросы пользователей).

Важнее всего для них только одно – качество программного продукта, а не расходы на обслуживание. Эти специалисты по сопровождению видят свою задачу в том, чтобы продлить жизнь системы, и они как профессионалы посвящают этому все свои усилия.

Просвещение просветителей

Поэтому, дорогой профессор информатики, я решил, что пришло время просвещать просветителей. По-моему, Вам полезно было бы знать, что специалисты по сопровождению:

- Расходуют от 40 до 80% средств, необходимых для обеспечения жизненного цикла ПО.
- Тратят 75% своего времени на улучшение программных продуктов и лишь 15% – на исправление ошибок (Ричард К. Болл (Richard K. Ball), профессионал сопровождения и большой шутник, вел колонку в *Software Maintenance News* и в 1988 году написал: «Предназначение специалиста по сопровождению не в том, чтобы исправлять чужие ошибки. Мы делаем это лишь в порядке оказания любезности.»).
- Помогают людям, управляющим нашей экономикой, создающим продукцию таких высоких технологий, о которых наши родители и не мечтали, и даже запускающим космические корабли на другие планеты.

И я *горжусь*, дорогой профессор, что принадлежу к числу специалистов по сопровождению. Дело не только в том, что мне приятно заниматься своей работой. Мир в целом тоже начинает понимать роль, которую играют специалисты по сопровождению ПО.

На первой полосе *The Wall Street Journal* в 1988 году было сказано, что специалисты по сопровождению «играют жизненно важную роль» и что их задача в том, чтобы «привести нас к спасению».

В исследовательском отчете IBM за 1980 год: «Проблема... сопровождения в настоящее время стала центральной проблемой программной инженерии».

А Барри Боэм, один из главных выразителей интересов программной инженерии, сказал в своей программной речи в 1987 году: «Необходимо поднять сопровождение ПО на более престижную позицию».

Специалисты по сопровождению решают задачи, требующие немалых знаний, мастерства и информации. И они занимают *видное* место в общей схеме.

Дорогой профессор информатики, можете ли Вы работать совместно с нами, чтобы способствовать появлению специалистов, обладающих знаниями и мастерством, которые необходимы в такой ответственной области, как наша?

Я думаю, что можете. Я надеюсь, что можете. Но я знаю, что Вы этого не сделаете, если не измените свой образ мышления и образ действия.

Согласно все тому же отчету IBM (1980 год), «программная инженерия поляризована вокруг двух субкультур: тех, кто мыслит и наблюдает, и тех, кто выполняет реальную работу. Первые изобретают, но не идут дальше того, что публикуют работы о новых достижениях, и поэтому никогда не узнают, насколько полезной (или бесполезной) была их идея. Последние, располагая средствами лишь для рациональной разработки продукта (но не для экспериментирования), должны прибегать лишь к проверенным, хотя и устаревшим, методикам».

Дорогой профессор информатики, мы – практики – нуждаемся в помощи мыслителей и наблюдателей. Я надеюсь, что Вы поймете необходимость помочь нам.

Ретроспектива

Я люблю говорить, что мой разум принадлежит теории программной инженерии, а мое сердце – ее практике. Речь в этом разделе пойдет именно о теории. Как вы увидите, я смотрю на теорию скорее критически, чем одобрительно. Я полагаю, что за прошедшие годы теория программной инженерии/компьютерных наук подвела практику под монастырь многократно и разными способами, причем иногда как следует.

Этот постулат становится очевидным в первом же очерке данной главы, «Структурные исследования? (немного ироничный взгляд)», в ко-

тором я, по выражению одного обозревателя, «жестoko высмеиваю» академические исследования как источник практических знаний. Кое-какие частности – вроде забытых теперь методологий – уже устарели (то же самое можно сказать и о некоторых других мыслях, высказанных в этой книге), но суть, которую я тогда пытался донести до читателей, совершенно не потеряла своего значения, что очень жаль.

Устарелость становится намного более серьезной проблемой во втором очерке этой главы «Проблема информационного поиска», который я написал в то далекое время, когда Google еще не стал лучшим другом всех тех, кому надо что-нибудь найти. Все дело заключалось в том, что источники, которые указывались тогда в ссылках, были недостаточно содержательными.

Размышляя недавно, так ли это и по сей день, я понял, что вышел за пределы предметной области, которая изначально пробудила мой интерес. В те далекие дни я работал в компании аэрокосмической отрасли и очень хорошо понимал роль, которую играли MIL-SPEC и другие спонсируемые правительством исследования в моей способности в полной мере охватить открытия, сделанные в ходе исследований ПО.

Поскольку сам я больше не работаю на конторе .gov, я связался с некоторыми из тех, кто работает, и постарался узнать, что могут для нас сделать в этой области современные поисковые машины. Вот что я узнал: Google действительно собирает все, что может быть обнаружено во Всемирной Сети, в том числе, к примеру, отчеты Национального агентства по авионавигации и исследованию космического пространства (NASA), Национального института стандартов и технологий (NIST), Национального научного фонда (NSF) и даже ФБР (FBI). И если сфера ваших интересов включает именно правительственные отчеты, то вы вполне можете «гуглить», вооружившись поисковым словом *unclesam*. (За помощь в этом вопросе я должен поблагодарить Виктора Бэсили (Vic Basili) и Марвина Зелковитца (Marvin Zelkowitz)!)

Возможно, сильнее всего устарел (и стал интересен исторически) материал очерка «Изучение ПО: новый источник информации». В то время, когда я писал этот очерк, Институт программной инженерии поставлял некоторые ценные и уникальные материалы по информационным ресурсам, связанным с программной инженерией. Я работал над проектом, который SEI называл своим «учебным модулем» (Curriculum Module), и испытывал сильнейший энтузиазм по поводу этого проекта. К тому времени уже образовался представительный корпус удачной литературы, и институт SEI планировал сделать его полным и всеобъемлющим.

Почему из этого ничего не вышло? По двум причинам.

Приблизительно в это же время SEI начал свой проект, ставший известным под названием «Модель развития функциональных возможностей» (Capability Maturity Model – CMM), и ресурсы, изначально предназначенные для учебного модуля, плавно переключались в сторону CMM.

Кроме того, сам проект учебного модуля испытывал серьезные трудности управления. Чрезмерное увлечение научными аспектами в этом подразделении SEI неизменно приводило к отвержению результатов работы над учебным модулем, выполняемой умелыми и знающими практиками, и из-за вздорных представлений о качестве, основанных на научных представлениях, была забракована почти вся работа, которую сделали и ученые, и практики.

По сути, кислород был перекрыт для всех новых учебных модулей, проект прикрыли, и он постепенно зачах и умер.

Значит ли это, что университетской науке и ее исследовательским лабораториям нечего сказать практикам? Отнюдь нет. С годами некоторые университетские ученые поняли, что лучший способ создания теории программной инженерии заключается в том, чтобы формализовать лучшие достижения практики. Те, кто поступит именно так и найдет время поразмышлять о способах улучшения проверенных и действующих практических технологий, получают в результате превосходный коктейль из практики и теории, который практики не могут получить сами и живительное действие которого они, вероятно, почувствуют незамедлительно.

Однако в лабораториях все еще обитают индивидуумы, которые, к сожалению, относятся к практике с пренебрежением и ведут свои частные исследования, не связанные с нуждами практиков, или (что хуже всего) исповедуют и отстаивают понятия и идеи, которые никогда не были – и наверное, не будут – испытаны на практике. Это прискорбное положение, характерное для прошлого, сейчас отчасти исправлено. Но, к сожалению, кое-где такие взгляды еще живы.

Глава 6 | Арена после боя

- Как компьютерная наука может стать настоящей наукой, а программная инженерия – настоящей инженерией
- Моя блестящая (или нет?) идея, которую я назвал «решение задач»
- Почему проваливаются программные проекты
- О том, насколько важны прикладные области
- Не поможете ли вы мне найти это?
- Ода вечно юному ПО
- Ретроспектива

Как компьютерная наука может стать настоящей наукой, а программная инженерия – настоящей инженерией

В этой статье я хочу порассуждать с вами о паре терминов, относящихся к программному обеспечению. Это слова *«формальный»* и *«структурный»*.

Эти прилагательные часто употребляются в сочетании с такими понятиями, как *«языки спецификаций»* и *«документация»*. Как правило, им придается положительный смысл. То есть считается, что нечто формальное или структурное лучше, чем то же самое, но лишенное структуры или неформализованное.

Эти слова интересны тем, что за каждым из них стоит свой легион компьютерщиков. За словом *«формальный»* имеют обыкновение объединяться представители сообщества университетских компьютерных теоретиков. Слово *«структурный»* обычно сплавивает представителей сообщества теоретиков, получивших образование «с предпринимательским уклоном». В этих сообществах означенные слова приобретают особенно глубокое значение.

Вопрос, который я хочу задать в этом очерке, крамольный: *«Должны ли эти слова иметь глубокое значение?»* Как мы себе представляем, что такое формальные сущности и что такое структурные сущности?

Ну, к примеру, мы знаем, что и те и другие активно отстаиваются многочисленными сторонниками. Другими словами, если кто-то где-то написал о «Формальной Штуковине» или «Структурном Причиндале», то можно заранее и мгновенно сказать, что статья выдержана в положительном, а может быть, даже в хвалебном духе. «Все компьютерщики-практики обязательно должны изучать формальную/структурную Абракадабру» – такой вывод не редкость для подобных статей. Как и заявление «Все наши сотрудники окончили курсы по внедрению формализованных/структурных Барабанов, и производительность возросла на 116%».

А теперь я хочу задать вопрос. Вот он: «Откуда мы знаем, что формальное/структурное что бы то ни было лучше, чем то же самое неформальное/бесструктурное?»

Вопрос кажется дурацким, да? Я хочу сказать, что мы интуитивно наделяем сущности, характеризующиеся этими положительными прилагательными, более высокими качествами, чем сущности, этими прилагательными не характеризующиеся. Так?

Полагаю, что правильный ответ «да». «Да, интуитивно кажется, что они лучше». Весь фокус в слове *«интуитивно»*. Разве интуитивные решения помогают выбирать лучшие ходы в поисках путей повышения производительности ПО?

Итак, вопрос задан, ответ сформулирован, и я выскажу свою мысль.

Компьютерная наука как наука и программная инженерия как инженерия не поддерживаны крепкой экспериментальной составляющей.

Иначе говоря, мы *не должны* полагаться на интуицию, решая, лучше ли формальные/структурные подходы, чем их зеркальные собратья с частицей «не». Мы должны дать научное определение и поставить эксперименты, которые предоставили бы нам количественные ответы на эти вопросы. И тогда мы могли бы заявлять: «Было показано, что структурное проектирование на X% эффективнее, чем неструктурное проектирование» или «Было показано, что ПО, прошедшее формальную верификацию, на Y% надежнее, чем ПО, которое ее не прошло».

А на основании подобных заявлений мы могли бы проанализировать затраты и выгоды и понять, стоило ли вбивать Z000 долларов в переподготовку сотрудников, чтобы заменить их старые знания новыми. В конце концов, новые знания не всегда лучше старых. Как отличить их друг от друга?

Так почему же наука и инженерия нашей отрасли до сих пор не имеют крепкой экспериментальной поддержки? Я вижу две причины:

1. Эксперименты, управляемые и проводимые должным образом, нелегки и обходятся недешево. Недостаточно посадить трех недоучившихся студентов за написание 50 строк кода на Бэйсике и сравнить их результаты. В представительном эксперименте необходимо задействовать настоящих разработчиков ПО, решающих реальные задачи в тщательно продуманном и спланированном окружении.
2. В нашей отрасли ни у инженерно-технических специалистов, ни у ученых нет ни мотивации, ни подготовки для проведения таких экспериментов. Мы так долго привыкали к пропаганде, что ни до кого, кажется, не доходит: наши исследования лишены важной составляющей. А из-за отсутствия мотивации к восполнению недостающей части ни у кого нет соответствующих интеллектуальных орудий, позволяющих узнать, как вести экспериментальную работу.

(Возможно, слова *«ни у кого»* в предшествующем абзаце являют собой пример излишней категоричности суждений. К примеру, специалисты, работающие на стыке ПО и психологии, которые занимаются «эмпири-

ческими исследованиями», ведут весьма интересную экспериментальную работу.)

Как же нам решить проблему экспериментов? К сожалению, решение не обещает быть ни легким, ни дешевым, поэтому необходимы масштабные действия. Нам нужен национальный институт, финансируемый крупной промышленностью или правительством и располагающий достаточными ресурсами. Или частная компания со стартовыми средствами, достаточными для проведения экспериментов и последующего рыночного внедрения результатов в промышленных приложениях.

Сейчас вы, наверное, подумали: «Так ведь такая организация уже есть». Разве не этим должны заниматься Консорциум продуктивности ПО (Software Productivity Consortium) и Институт программной инженерии (Software Engineering Institute)? Ну да, они, по-моему, должны, но на самом деле они этим не занимаются. Ставить эксперименты – это дорого и трудоемко, и поэтому большинство людей предпочитают реальным экспериментам «мысленные». Организации, которые финансируются промышленностью и правительством и которые могли бы проводить экспериментальные исследования, попросту этого не делают и вряд ли когда-нибудь начнут.

Но разве не хорошо было бы, если бы они это делали? Разве не было бы прекрасно, если бы кто-то, располагающий необходимыми средствами, начал выпускать что-нибудь вроде «Отчетов для потребителей ПО» (Software Consumer Reports)? Тогда мы могли бы приступить к трансформации того, что Дэвид Парнас (David Parnas) назвал «народным компьютерным знанием», в нечто более научное. И тогда, вооружившись экспериментальными данными, мы больше не рассуждали бы с удивлением о положительных свойствах прилагательных *«формальный»* и *«структурный»*. Мы бы знали не только, что они означают, но и каково их значение для нас.

Не правда ли, это было бы прекрасно? И даже важно?

Моя блестящая (или нет?) идея, которую я назвал «решение задач»

Этот очерк посвящен «решению задач». В нем я собираюсь высказать кое-какие соображения, и, честно говоря, не знаю, каковы они: банальные или гениальные. Вы поможете мне понять это?

Дело, которым мы так давно и плотно занимаемся, – это *не* обработка данных, не вычисления и вообще не любое из этих приложений, тесно

связанных с технологией, которые мы обычно называем сферой нашей деятельности.

Дело, которым мы занимаемся, – это решение задач.

Но эта мысль точно банальная, потому что решением задач занимаются практически все. Если вы занимаетесь персоналом, то решаете задачи, связанные с людьми. Если вы занимаетесь изготовлением чего-либо, то решаете задачи, связанные со сборкой. Если вы занимаетесь бухгалтерским учетом, то решаете задачи, связанные с финансами. Решение задач – это универсальное занятие. Мы отличаемся только тем, что решаем наши задачи при помощи компьютеров.

Так где же блеск в этой банальной мысли?

Посмотрим, какие рекомендации мы даем тем, кому приходится решать задачи. Мы говорим, что задачу надо сформулировать. Что надо составить план ее решения. Затем построить решение. Мы говорим, что решение надо проверить. И еще мы говорим, что работающее решение надо поддерживать в работоспособном состоянии, если это необходимо.

Эти наши рекомендации одинаковы и для тех, кто учится техническому писательству и набрасывает небольшую статью, и для тех, кто изучает математику и штурмует текстовую задачу, и для тех, кто изучает архитектуру и проектирует здание. Кажется, эти рекомендации мы даем независимо от того, какую задачу они решают!

В связи с этим я думаю, что существует общий подход к решению задач, не зависящий от того, к какой прикладной области они относятся.

Высказанная ранее мысль становится интересной и, возможно, даже блестящей благодаря тому, что было упомянуто парой абзацев выше и что мы, как мы думаем, придумали десятки лет тому назад. В конце 1960-х годов мы изобрели «жизненный цикл ПО». В жизненном цикле мы определяем, как мы выражаемся, требования к нашей вычислительной задаче, проектируем решение, воплощаем его в программном коде, тестируем код, а потом, получив готовый программный продукт, сопровождаем его. А теперь скажите, как соотносятся между собой фазы жизненного цикла и фазы решения задачи? Они практически идентичны, пункт в пункт.

Иначе говоря, то, что мы называем жизненным циклом, – идея совсем не новая. Также нельзя сказать, что она родилась в сфере разработки ПО. Она уходит корнями в дисциплину, намного более обширную и важную, чем наша.

К последствиям этого мы вскоре вернемся. Сначала, однако, я хочу остановиться на паре забавных моментов.

Первый заключается в том, что разработчики ПО тратят массу времени, споря, какую модель жизненного цикла следует выбрать. Кое-кто вообще отвергает идею жизненного цикла, называя ее «вредной». Некоторые считают, что в связи с появлением новых идей, например прототипирования, подходы, связанные с жизненным циклом, перестали быть эффективными. Были предложены новые модели жизненных циклов. Институт SEI, где я работал, опубликовал доклад, в котором было определено полдесятка таких моделей.

Что тут забавного? Дело в том, что мы упустили главное. Не нам отмечать идею жизненного цикла, в котором воплощен обобщенный подход к решению задач. Он общий для нескольких дисциплин, и мы можем уточнить его для наших программистских целей. Но, отказываясь от него или изменяя его радикально, мы отказываемся от бесценного опыта, накопленного другими отраслями знаний, намного более старыми, чем наша.

Второй забавный момент: некоторые разработчики ПО никогда не понимали по-настоящему, что такое жизненный цикл. Когда он был «открыт», кое-кто воспринял его как незыблемый свод правил, процедуру, которой надлежало следовать при разработке ПО неукоснительно, проходя ее фазы в раз и навсегда заданном порядке правильным образом. Нельзя начинать проектирование, не закончив формулировать требования. Нельзя приступать к написанию кода, если не закончено проектирование. Нельзя тестировать программный код, если не закончено его написание.

Эти жесткие представления о жизненном цикле должны были стать очевидными для нашего здравого смысла. Например, вышеупомянутый технический писатель, который учится сначала составлять план статьи, а потом писать ее, тем не менее знает, что написание небольшого пробного фрагмента имеет большое значение для формирования плана и что скомканная страница рукописи, не прошедшая проверку, может означать всего лишь, что задача еще не была понята или что план еще не был составлен до конца. Архитектор, упомянутый ранее, который проектировал здания, прежде чем строить их, на самом деле знает, что, прежде чем заканчивать план и заливать бетон, сначала строят модель готового здания и экспериментируют с ней, чтобы понять задачу и предполагаемый путь ее решения. Никто, никто никогда не утверждал, что решать задачу можно, только проходя перечисленные выше этапы в строго определенной правильной последовательности. С чего вдруг нам, разработчикам ПО, взбрела в голову эта мысль?

Однако оставим иронию и продолжим...

Я все больше убеждаюсь, что общая дисциплина решения задач существует и что мы, разработчики ПО, усердно и настойчиво открываем ее заново, делая один неловкий шаг за другим.

Если это так, то, наверное, есть и еще какие-то знания, которые мы пока не «открыли», но которые мы откроем, затратив массу сил, и будем взирать на них с огромным удивлением и пытаться объяснить их, демонстрируя при этом свое вопиющее невежество. И какой это будет стыд, потому что мы вполне могли бы сорвать плоды этих знаний, если бы только знали, на каком дереве они растут.

Я не исследовал эту мысль с учебно-научных позиций и поэтому не знаю, существует ли учебная дисциплина, посвященная теме решения задач. Я точно не изучал ничего подобного в университете. Но если такой дисциплины нет, то она, конечно, должна быть. Но не внутри учебных планов по инженерно-научным дисциплинам, по архитектуре или математике, где к ней могли бы получить доступ лишь немногие избранные. Она должна находиться в открытом пространстве, в междисциплинарном курсе на стыке разных отраслей, где к ней мог бы приблизиться каждый. Решение задач, как мы уже видели, входит или должно входить в сферу интересов каждого.

Поэтому разрешите мне напоследок задать два вопроса. Так ли эта моя мысль банальна, как мне кажется иногда, или она так гениальна, как мне кажется все остальное время?

И второй вопрос, может быть, более важный: есть ли где-нибудь в университетском мире, где сейчас так или иначе учат решать задачи, место, в котором можно было бы ожидать следующего великого открытия в сфере знаний о программной инженерии?

Если у вас есть ответы на любой из этих вопросов, я бы выслушал их с очень большим удовольствием.

Почему проваливаются программные проекты

С годами я стал специалистом по провалам программных проектов. В Ассоциации по вычислительной технике (Association for Computing Machinery – ACM) я известен как национальный лектор по провалам. Я написал серию книг по этой теме, например «The Universal Elixir and Other Computing Projects Which Failed, and Computing Catastrophes». Я настолько погряз в отказах и провалах, что до сих пор езжу на «Студебекере», а еще я уверен, что Джимми Картер был одним из лучших президентов США!

Однако я предпочитаю смотреть на отказы и провалы с юмористической точки зрения. Компьютерщики обожают обмениваться историями

о провалах. Я убежден, что способность посмеяться над провалом – сначала с кем-то, кто испытал его на себе, а потом и над собственной неудачей – открывает нам доступ к жизненно важной стороне человеческой природы нашей личности и к скромности.

По прошествии некоторого времени в моем маленьком мире, где над провалами можно было смеяться, некоторые из моих слушателей и читателей стали побуждать меня к исследованию более серьезных, зримых компьютерных провалов и к извлечению из них ценного опыта. В конце концов, самый убедительный опыт ребенок получает, когда прикасается к горячей печке. А теперь я постараюсь Говорить О Провалах Seriously.

Глядя на останки разбитых проектов, о которых я говорил и писал для удовольствия, я вижу, как они складываются в узоры, из которых вырисовываются некоторые общие причины случившихся неудач. Именно об этих узорах я и хочу поговорить.

Неспособность оценивать

Первая и самая распространенная, по-моему, причина провалов в компьютерной отрасли – это наша неспособность оценивать. Мы можем прикладывать какие угодно титанические усилия, применять какие угодно замечательные инструменты и алгоритмы, но наши предположения о сроках выполнения проекта и затратах не выдерживают никакого сравнения с действительностью, которая предстает перед нами, когда продукт наконец готов. Вот один из самых недавних и ярких примеров: Microsoft неправильно оценила срок выхода новой версии своего текстового процессора, вывела с рынка предыдущую версию, прежде чем была готова новая, и за один день акции компании упали на 15%. Некоторые фирмы, специализирующиеся на разработке ПО, начинают расписывать свое ПО настолько задолго до его фактической готовности, что ко всей индустрии ПО приклеилось словечко *varoware*, означающее «ПО-фантом» – продукт, осязаемость которого не больше, чем у облачка дыма.

Почему нам так плохо удаются оценки? Может быть, потому, что мы до сих пор не понимаем нашу отрасль, еще не вышедшую из младенческого возраста. Мы только начинаем осознавать тот факт, что написание ПО – это намного больше, чем написание программного кода. Мы все еще только начинаем осознавать, что проверка занимает намного больше времени, чем мы могли себе представить, и что этот процесс протекает непредсказуемо. Мы до сих пор потрясены тем, что сопровождение ПО поглощает более половины средств, которые люди тратят на ПО. Беда не в том, что мы не знаем, как создавать программные продук-

ты, – я думаю, что мы знаем. Беда в том, что мы до сих пор не до конца понимаем процесс.

Почему имеет значение, что наши оценки так плохи? Потому что мы пытаемся управлять, опираясь на них. Мы строим бюджетные схемы и сложные графики, основанные на наших оценках, а потом пытаемся заставить разработчиков закончить продукт к тому сроку, который определен схемами и графиками. А когда оказывается, что продукт к этому сроку *не* готов, менеджмент начинает давить на разработчиков, чтобы уменьшить разрыв между датой окончания по расписанию и постоянно отодвигающейся *реальной* датой. Это в свою очередь обостряет проблему. Постепенно проект из запаздывающего и перерасходующего бюджет превращается в запаздывающий, перерасходующий бюджет и ненадежный, потому что приходится экономить на тестировании, необходимом для того, чтобы избавиться от ошибок. Лично я уверен, что неправильная оценка – это практически единственная причина «кризиса ПО» и утверждений о том, что программные проекты «никогда не укладываются ни в бюджет, ни в сроки и порождают ненадежное ПО». Если я прав, то эта проблема самая важная в программной инженерии.

И очень многие стараются разрешить эту проблему. Алгоритмы оценки имеются в изобилии. Некоторые из них воплощены в программных средствах. Однако результаты алгоритмических оценок настолько же ошибочны, насколько оптимистичны оценки, которые дает человек. Различные исследования показали, что алгоритмические оценки одного и того же гипотетического программного проекта расходятся аж на 800%. Наверное, когда-нибудь с помощью алгоритмов можно будет давать правильные оценки, может быть, вам это удастся даже сейчас, если вы настроите модель под свои задачи и проверите ее корректность в ваших конкретных условиях, но пока что ни о каком массовом надежном применении алгоритмических оценок речь не идет.

Поэтому оценкой, несмотря на всю свою предвзятость и весь свой оптимизм, по-прежнему занимается человек, поскольку пока что квалифицированный разработчик ПО – это самый достоверный источник предсказаний. Однако в целом мы пока не делаем то, что можем сделать сейчас, чтобы начать избавляться от собственного незнания или отказаться от привычного нам процесса. Мы не собираем ретроспективные данные, которые воссоздали бы картину прошлых реалий построения ПО и помогли бы нам точнее оценивать наше будущее. Каждая софтверная организация, которая тщательно подберет такие данные, связывающие процессы со штатом сотрудников, а сроки исполнения – с прикладной принадлежностью и сложностью ПО, сделает огромный шаг к уве-

ренным и надежным предсказаниям. Однако это делают лишь очень немногие компании.

С оценками ПО связан еще один интересный нюанс. Те, кто интересовался, как обстоят дела в смежных дисциплинах, говорят, что никто, ну *никто* не пытается давать окончательные оценки на такой ранней стадии процесса разработки, как это делаем мы в индустрии ПО. На каком этапе мы обычно делаем оценки? Как правило, перед сбором требований или непосредственно во время этого процесса. Откуда нам знать, сколько времени потребуется, чтобы создать продукт, если мы еще не измерили всю глубину и сложность задачи? Может быть, все наши успехи и страдания зависят от оценок, с которыми мы вообще не должны иметь ничего общего. То есть ощущение буквально такое, что большие дяди, заправляющие в смежных областях, задирают новичков из индустрии ПО и склоняют их к тому, чтобы они давали обещания, которые никогда не смогут выполнить, а они слишком неопытны, несведущи и бесхарактерны, чтобы дать отпор. Не исключено, что задача оценки ПО может быть *окончательно* решена методом последовательных приближений, в ходе которых на каждом из нескольких четко различимых основных этапов вырабатывается новая оценка, а принимаемые решения основываются на *самой свежей* оценке, а не на *самой первой*. Если воплотить этот способ в жизнь, то менеджеры, которые привыкли ощущать, что они «держат руку на пульсе», могут лишиться этого ощущения, но в теперешней ситуации с оценкой оно, так или иначе, все равно является иллюзией.

Однако может возникнуть вопрос: не станет ли изменчивость оценок предпосылкой к злоупотреблениям со стороны технических специалистов? Известны некоторые интересные данные, имеющие отношение к этому вопросу. В ходе одного из исследований было установлено, что производительность труда программистов, которые разрабатывают ПО, не страдая от ограничений графика работ, выше, чем у тех, кто таким ограничениям подчиняется, независимо от того, кто составлял график (даже если это были сами программисты). Может сложиться впечатление, что люди работают лучше всего тогда, когда они *сами* контролируют свою работу (и некоторые в индустрии ПО считают именно так). Тут есть о чем подумать.

Нестабильные требования

Нестабильность требований, над выполнением которых мы работаем, является, по моему, второй самой распространенной причиной провалов. Слишком часто оказывается, что цель, которую мы стараемся поразить, не стоит на месте. Стараясь ни с кем не испортить отношений, мы соглашаемся со всеми изменениями, которые предлагают заказчик

или пользователи, в результате чего процесс проектирования переворачивается с ног на голову, а графики работ летят ко всем чертям.

Никто не способен решить задачу, постановка которой все время меняется. Ни программисты, ни кто-либо еще. Однако ПО недаром называют словом *software* – оно мягкое и потому податливое, и все вокруг уверены, а мы стараемся доказать, что способны создать настолько гибкое ПО, что изменения никак на нем не скажутся.

И у нас ничего не получается. Раз за разом мы демонстрируем, что программное обеспечение, несмотря на всю свою мягкость, не может быть бесконечно гибким. Лучший способ убедиться в этом – посмотреть на сопровождение ПО. Что такое сопровождение ПО? Главным образом, это внесение изменений в ПО, преследующее цель привести его в соответствие с новыми требованиями. На это уходит от 40 до 80% средств, затрачиваемых на ПО. Программное обеспечение податливо – мы и вправду способны создать решение, превосходящее своей гибкостью решения почти в любой другой области. Однако оно не настолько податливо, чтобы его можно было изменять как угодно. Фактически дело обстоит совершенно иначе. Изменения – это главный генератор прибыли в индустрии ПО!

Таким образом, секрет нестабильности требований заключается всего лишь в осознании реальности. Требования надо заморозить в начале проекта, сразу после принятия спецификации требований. Сделав это, не будьте слишком упрямы. Если стало понятно, что решение неэффективно, то от него надо отказаться. А когда столкнетесь с неизбежным требованием внести изменения, оцените их воздействие с хорошим запасом (см. первую причину провалов, упомянутую выше!) и выставьте заказчику счет за их реализацию. Если у заказчика есть достаточно веские основания хотеть изменений, то он заплатит. А если нет, то разве вы не обрадуетесь, что вам не пришлось вносить изменения бесплатно?!

Каких-то лет пять назад я думал, что нестабильность требований – это самая главная проблема индустрии ПО. Сейчас она переместилась (по крайней мере, в моем представлении) на вторую позицию, и это свидетельствует о росте нашей зрелости как в политическом смысле, так и в техническом. Большинство в индустрии ПО понимают суть этой проблемы. И она по-прежнему занимает второе место – не потому, что мы ее понимаем, а потому, что действуем в соответствии с нашим пониманием.

Интересно отметить, однако, что принятые в высокотехнологичных областях современные подходы к требованиям (формальные методы и формально определенные языки требований) ориентированы на совершенно другие аспекты проблемы требований. Сообщив строгость не-

стабильным требованиям, мы получаем строгие и нестабильные требования, а они ничуть не лучше, чем нестрогие и нестабильные. Нестабильность требований не удастся преодолеть путем исследований или с помощью технологий. Эту проблему можно разрешить только средствами менеджмента.

Капризы и заблуждения

Мы плавно подошли к третьей, самой важной причине провалов. Эта проблема, которую я называю «капризы и заблуждения», недоступна моему пониманию. То есть я понимаю, что это за проблема; я только не понимаю, почему она по-прежнему существует.

Если говорить прямо, все дело в том, что в индустрии ПО до сих пор жива вера в серебряную пулю. Дам короткое пояснение для тех, кто не читал работу Брукса, посвященную серебряным пулям. Проблемы, связанные с ПО, представляются неким аналогом оборотня, причем враг так опасен, что избавиться от него можно, только сразив насмерть. Легенда гласит, что оборотни уязвимы лишь для серебряных пуль. Поскольку оборотень мифический (умозрительный), то и серебряная пуля для его уничтожения должна быть умозрительной.

Кстати, что бы вы ни думали о кризисе софтверной отрасли, он, конечно, намного реальнее мифического оборотня. Если ближе к делу, то нет никакого смысла искать воображаемое средство разрешения проблемы независимо от того, насколько она реальна (или воображаема).

Что скрывается за абстрактной серебряной пулей в этой аналогии? Некий прорыв, который каким-то образом позволит создавать ПО недорого и безболезненно. А какие трудности связаны с серебряными пулями? Я бы, как и Фредерик Брукс, предположил, что их попросту нет, так что и искать нечего.

Почему тогда стремление найти серебряную пулю является причиной провалов программных проектов? Потому что в погоне за чудом мы забываем о более важных поисках обычных решений. Это еще ничего, если поисками серебряной пули занимаются ученые в увитых плющом стенах университетов (все-таки было бы неплохо, если бы кто-то из них также вспоминал о более перспективных исследованиях). А вот если реальные решения отвергаются менеджерами реальных программных проектов и надежды возлагаются на волшебство, это уже плохо.

Есть ли у меня свидетельства в пользу моих слов? Приведу несколько примеров.

Во-первых, возьмем так называемую «революцию структурного программирования». Она разразилась в 1970-х годах, и, прежде чем мы

осознали, что происходит, «тридцатью двенадцать» консультантов ринулись делать деньги, обучая структурным премудростям программистов всего мира. Какая разница, что не было экспериментальных данных, которые бы показали, каковы выгоды структурного программирования? И какая разница, что трезвые головы призывали вспомнить: хорошие программисты уже давно занимались именно структурным программированием? И неважно, что, когда дым рассеялся, оказалось, что прирост производительности труда и качества ПО, хоть и несомненный, составил каких-то 5–10% (в зависимости от того, насколько ужасен был исходный «макаронный код»). Попытки сделать всех структурными программистами поглотили уймы усилий, времени и денег. Нельзя ли было найти всем этим ресурсам лучшее применение?

Чтобы ответить на этот вопрос, вспомним отладчик, работающий в терминах языка исходного кода. Этот инструмент позволяет программистам отлаживать код на том же языке, на котором он был написан. Языки высокого уровня и связанный с ними рост качества ПО и производительности труда были в распоряжении программистов еще в далеких 1950-х годах. А в далеких 1960-х создатели инструментальных средств явили миру отладчики исходного кода, которые выводили из игры отладчики машинного кода, все еще преобладавшие на рынке.

Но когда началось интенсивное применение отладчиков исходного кода? Не раньше, чем в конце 1970-х, через целых десять лет, в течение которых о них все знали и каждый мог получить их в свое распоряжение. Революция структурного программирования и прозябание готовых к делу отладчиков исходного кода совпали во времени, и я думаю, что это не просто совпадение. Мы погнались за серебряной пулей и упустили не такую волнующую, но более полезную возможность.

А-а, вы говорите, что это все было в прошлом? Мы не повторим этих ошибок, ведь мы живем в свободные от предрассудков 1990-е? *А вот и нет!* В дело рвется другая технология – совсем как отладчики исходного кода двадцать лет тому назад. Те, кто принимает решения, старательно ее игнорируют. Что это за технология? Анализаторы тестового покрытия.

Зачем они нужны? Они позволяют измерить, насколько основательно ваши тестовые наборы охватывают структуру тестируемой программы. А зачем нам это знать? Дело в том, что мало протестировать ПО на предмет выполнения всех требований. Необходимо не только убедиться в соблюдении требований. По причинам, которые слишком сложны, чтобы в них сейчас вдаваться, надо также протестировать структурные компоненты ПО.

Почему анализаторы тестового покрытия не нашли широкого применения? Потому же, почему это случилось и с отладчиками исходного кода. Эта технология известна уже лет десять, и созданы продукты, уверенно работающие в различных окружениях (в том числе и на микрокомпьютерах), однако места, где они применяются, можно пересчитать по пальцам. И вот почему:

1. В инструментах, обеспечивающих «внутренний интерфейс» жизненного цикла, не так уж много внешнего блеска. Менеджмент, пребывая в эйфории по поводу выгод, связанных с активным развитием инструментов «внешнего интерфейса» (как ему и полагалось), потерял интерес к лишенным блеска и трудным внутренним процессам, таким как отладка.
2. Исследователи и продавцы ПО продолжают говорить менеджменту то, что тот хочет слышать, а именно что революционные решения – серебряные пули – рядом и только того и ждут, чтобы их взяли на вооружение.

Я сказал, что этот феномен мне непонятен. Почему менеджеры, компетенция которых прямо соотносится с их способностью выпускать реальное ПО в срок и укладываться при этом в бюджет, продолжают гоняться за серебряными пулями и не обращают внимания на более действенные и доступные решения? Я не знаю. Мне это попросту непонятно. Единственное объяснение, которое приходит мне в голову, – это что в нашей быстро меняющейся отрасли новейшие технологии (насколько угодно фантастические) помогают стяжать своим пропагандистам намного более пышные лавры, чем обыденная реальность.

Вся эта ситуация, которую я характеризую как сочетание капризов и заблуждений (потому что людям свойственно слепо следовать за новыми веяниями, которые ведут их в тупик), породила странные союзы. На стороне новейших технологий исследователи, продавцы-коммерсанты и менеджеры, идущие на поводу у собственных капризов. У этих людей очень мало общего, разве что интерес ко всему новому!

С областью капризов и заблуждений связаны очень сильные подавляемые эмоции. Вот, например, доказательство корректности программы, подразумевающее строгое математическое доказательство. За последние годы сторонники и оппоненты этой серебряной пули дважды устраивали злобную полемику на страницах главных профессиональных журналов. Последний их конфликт начался в марте 1989 года на страницах флагманского издания Ассоциации по вычислительной технике *Communications of the ACM*. Ничего похожего на спокойные и аргументированные профессиональные разногласия. Оппоненты переходили на личности и ставили под сомнение поря-

дочность друг друга. Чем вызван такой эмоциональный всплеск? Полагаю, что отчасти он вызван разочарованием тех, кто наблюдал отказ от готовых решений в пользу обещанных серебряных пуль. Опасность острых эмоциональных конфликтов будет сохраняться до тех пор, пока капризы и заблуждения будут преобладать над чувством реальности.

И еще я уверен, что нерешенной останется, а это еще важнее, одна из самых серьезных проблем индустрии ПО.

Обзор

Итак, где мы побывали в ходе нашего путешествия по стране провалов программных проектов? Мы увидели три самых серьезных причины провалов программных проектов: неспособность к оценке, нестабильные требования, капризы и заблуждения.

Можно ли устранить эти причины? Вероятно, хотя от столь крупных виновников провалов запросто не избавишься (если бы это было возможно, то это уже было бы сделано!).

Рецепты от неспособности оценивать:

- Вести сбор данных и, опираясь на них, улучшать оценки в будущем.
- Анализировать процесс оценки, чтобы выяснить, не поможет ли метод последовательных приближений лучше соотнести оценку с реальностью, с методами, принятыми в других дисциплинах.

Рекомендации по борьбе с нестабильными требованиями:

- Заказчики должны адекватно оплачивать любые выдвигаемые ими изменения требований.

А для того чтобы победить капризы и заблуждения, надо:

- Убеждать менеджеров, успевших обжечься на посулах продавцов и исследователей, что есть источники информации, более достойные внимания.
- Найти кого-то, *кто основал бы* Национальный центр экспериментальной деятельности в области ПО (National Software Experiment Center), где капризы и заблуждения подверглись бы экспериментальной проверке, прежде чем практики всей страны успели бы пойти за ними в очередной тупик.

Как видите, простых решений нет, как, впрочем, и серебряных пуль. Может быть, для того чтобы мы все начали искать способы разрешения самых серьезных проблем, достаточно, чтобы мы знали, что это за проблемы.

О том, насколько важны прикладные области

Попробую предсказать будущее программной инженерии.

На рубеже веков произойдет ее раскол на несколько дисциплин. Точно так же, как существуют электротехника (электрическая и электронная инженерии), машиностроение (механическая инженерия) и гражданское строительство (civil engineering), образуются научно-техническая программная инженерия, инженерия бизнес-ПО, инженерия ПО реального времени и инженерия системного ПО.

Почему я так считаю? Просто я думаю, что именно сейчас мы начинаем осознавать влияние прикладных областей на методы, которыми мы создаем ПО. И еще я думаю, что по мере понимания последствий этого влияния мы начнем осознавать, что нельзя говорить о программной инженерии вообще, не адресуясь к ее конкретным компонентам, областям прикладных задач, которые представляют собой актуальные сферы, характеризующиеся общими параметрами аналогичных задач.

Я бы назвал период времени, через который мы проходим, эпохой универсальных решений. Приверженцы различных языков, методологий и инструментов заявляют, что лучшим средством решения любых задач является именно то, что предлагают они. Их заявления почти всегда ошибочны. Опираясь на оптимизм, порожденный неосведомленностью, специалисты в одной прикладной области приходят к выводу и заявляют во всеуслышанье, что их знания можно успешно распространить на другие прикладные области. Как правило и к большому сожалению, этого сделать нельзя.

Какие свидетельства подкрепляют мою точку зрения?

1. Виктор Высоцкий (Victor Vyssotsky), директор Кембриджской исследовательской лаборатории корпорации DEC, избрал темой почетной лекции, которую он читал в институте SEI в 1988 году, эту самую мысль, сказав, что трудные, запутанные вопросы прикладных областей представляют собой такой же насущный компонент арсенала специалиста по разработке ПО, как и его сугубо специальные знания.
2. На Международной конференции по программной инженерии в 1987 году аудитория не баловала выступавших аплодисментами, но прервала ими доклад Джона Келли (John Kelly), когда он, говоря о методологиях проектирования ПО реального времени, сказал, что нет самого лучшего способа проектирования для всех приложений и что поиски такого способа могут затормозить более важный прогресс в отрасли.

3. Японцы, формируя гипотетический государственный стандартный набор инструментальных средств для разработки ПО, включили в этот набор Кобол и Фортран и дали этим языкам совершенно разное функциональное назначение.
4. Несмотря на первую эйфорию по поводу языков 4GL и их воздействия на развитие индустрии ПО, область их применения не вышла за пределы немногих тесно связанных прикладных областей.
5. Эксперименты по представлению логики и данных, которые проводила Айрис Весси (Iris Vessey) из Пенсильванского государственного университета, показали, что при выборе схемы представления следует больше руководствоваться соображениями «когнитивной подгонки» (cognitive fit) инструментов, то есть их пригодностью для решения конкретной задачи, чем какими-либо другими факторами.
6. В мартовском выпуске *IEEE Transactions on Software Engineering* за 1989 год была опубликована статья Бо Сэндена (Bo Sanden) «The Case for Electric [sic] Design of Real-Time Software» (Аргументы в пользу эклектического (!) проектирования ПО реального времени), где он отстаивает подход к проектированию, который он сам назвал «эклектическим», и говорит, что «вместо того, чтобы спорить о том, какой метод проектирования лучший, надо применять... любую комбинацию методов, которая дает значимые результаты в конкретной ситуации».
7. В обзоре, содержащем анализ факторов производительности в проектах ПО обработки бизнес-данных и выполненном для журнала *System Development*, сказано, что «компетенция в прикладных областях» переместилась в списке важнейших факторов с четвертой позиции, которую она занимала в 1984 году, на вторую в 1987 году (сразу за «компетентностью персонала/команды»).
8. Одной из ложных идей, разоблаченных в работе Рича и Уотерса (Rich, Waters), посвященной «коктейльным» мифам автоматического программирования, была та, что при автоматическом программировании особенности прикладной области не имеют никакого значения.
9. Бэсили и Селби (Basili, Selby) в своей работе о сравнении анализа и тестирования обнаружили, что прикладная область ПО определяет некоторые виды ошибок и методики их удаления.

Я изложил свое представление о перспективах программной инженерии и объяснил, почему я вижу их именно так. Должен признаться, что полной уверенности в собственной правоте у меня нет. По-моему, жизненность традиционных подходов до сих пор подкрепляется многочисленными свидетельствами. Сторонники новых высокоразвитых

языков рекомендуют применять их в бизнес-приложениях, не осознавая, что многие функции, необходимые таким приложениям, практически не поддерживаются этими новыми языками. Утверждения об универсальности технологий и инструментов CASE до сих пор сопровождаются пустозвонством о независимости этих средств от особенностей прикладных областей. Диаграммы потоков данных, которые до сих пор преподносятся как пример структурного подхода к ПО, несомненно, полезны для представления процессов в приложениях, ориентированных на данные, однако их ценность для других приложений сомнительна. Надежды на автоматическую разработку ПО не увязываются с необходимостью дополнения мастерского владения специализированным инструментарием глубокими знаниями в прикладной области. Чтобы мое предсказание сбылось, необходимо низвергнуть существующую укоренившуюся систему взглядов.

Для этого, по-моему, в первую очередь необходимо начать исследования, сосредоточенные на данной области. Каковы важнейшие прикладные области? Чем они характеризуются? На какие функциональные потребности должны быть ориентированы прикладные языки программирования? При помощи каких средств и методов можно удовлетворить эти функциональные потребности? Как лучше всего рассказать о них? Где проходит разграничительная линия между подходами, специфическими для прикладной области, и универсальными? Какова доля программной инженерии (да и вычислительной техники вообще), которую можно преподавать без оглядки на прикладные вопросы?

На все эти вопросы должны отвечать те, кто понимает все многообразие прикладных областей и их запросы. Иначе говоря, в этой области практика ведет за собой теорию, и построение последней должно основываться на изучении практического опыта. К примеру, отличной отправной точкой дальнейших исследований мог бы стать сделанный практиками обзор прикладных областей и примитивов, необходимых для конкретных областей.

Будущую систему обучения программной инженерии, где могли бы реализоваться некоторые из моих мыслей, представленных здесь, я бы структурировал так, как показано на рис. 6.1:

Универсальный базовый курс:

- Р = решение задач (problem solving) – не связано с прикладными областями

Базовый курс вычислительной техники:

- С = вычислительная техника (computer science) – не связана с прикладными областями

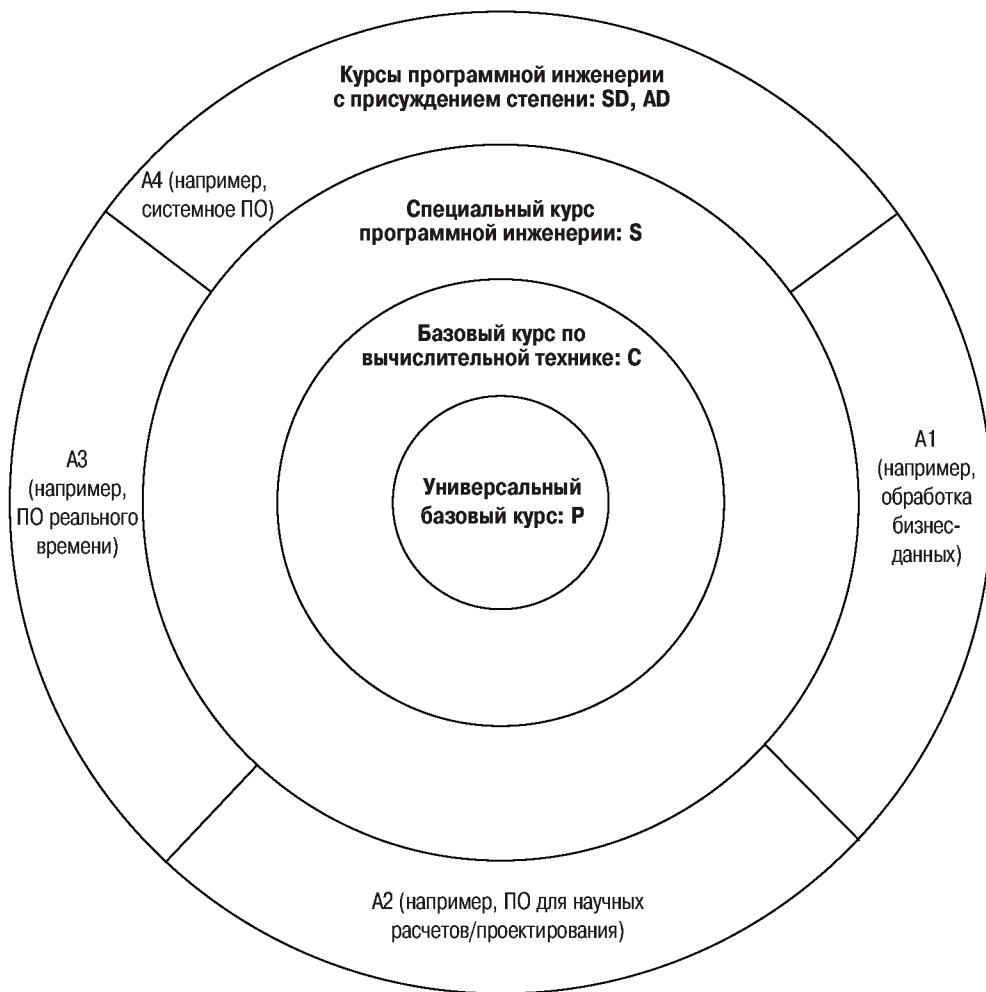


Рис. 6.1. Структура предлагаемой системы обучения программной инженерии

Специальный курс программной инженерии:

- S = программная инженерия (software engineering) – не связана с прикладными областями

Курс обучения программной инженерии с присуждением степени:

- SD = программная инженерия (software engineering) – с прикладным уклоном

- AD = прикладное программирование с ориентацией на конкретные области (Application Domain Specific Topics)

Такая система образования могла бы дать следующих специалистов:

- Исследователей в области вычислительной техники (курсы P + C + соответствующая программа AD)
- Исследователей в области программной инженерии (курсы P + C + S + соответствующие программы SD и AD)
- Специалистов по системному ПО (курсы P + C + S + соответствующая программа SD)
- Разработчиков прикладного ПО (курсы P + C + S + программы SD + AD)
- Специалистов по работе с прикладным ПО (курс P + программа AD)

Да уж, предсказания – дело загадочное. Как указывает Эд Йордон (Ed Yourdon), предсказания, имеющие отношение к технологии, всегда слишком консервативны (скорость развития технологии всегда выше, а направления – разнообразнее, чем мы предсказываем), а социологические прогнозы всегда слишком радикальны (реальная скорость развития общества всегда ниже). Однако некоторые важные факторы, способные изменить будущее отрасли ПО, мною были в этом очерке указаны. Интересно посмотреть, каковы будут *реальные* изменения!

Не поможете ли вы мне найти это?

Удовольствие. Ведь именно в нем заключается весь смысл программирования? Мы отдаем компьютерам и программированию все свои силы и время, потому что это интересно и приятно.

Да, конечно. Еще мы делаем это ради денег, социального статуса, профессионального роста и прочих прозаических вещей, о которых люди пишут книги. Однако, по сути, мы занимаемся этим ради удовольствия. Именно поэтому никто еще не написал книгу под названием «Программист на минуту»... и вовсе не потому, что программирование отнимает больше времени, – дело в том, что невозможно оторваться от этого занятия через одну минуту.

После того как я сказал это, я должен сделать одно небольшое признание. Работа в индустрии ПО *была* для меня удовольствием. По прошествии 30 лет удовольствия стало намного меньше. Я переживаю третий кризис середины жизни и пытаюсь понять почему. Я пишу этот очерк, чтобы посмотреть, не найдется ли среди читателей кого-то, кто мне поможет.

И вот чего я достиг к настоящему моменту. У меня получился небольшой список возможных объяснений – так, на пробу.

1. *Работодатели думают, что сотрудники, достигшие тридцатилетнего возраста, слишком ценны, чтобы позволить им заниматься программированием.*

Последний раз меня попросили написать программу за деньги так давно, что я уже и не вспомню, когда это было. А как же – ведь я должен изучать требования, или составлять планы, или заниматься исследованиями, или писать статьи. Но это все не то. Удовольствие от программирования – в самом программировании. И больше ни в чем (покашливание). И работодатели уверены, что юный новоиспеченный программист ничуть не хуже, чем программист постарше и с опытом.

2. *Программисты горят на работе.*

Однако удовольствие от программирования имеет свою оборотную, темную сторону. Из тысяч тончайших, сложнейших узоров программист создает живое полотно, исполняемое произведение искусства. Через некоторое время возня с этими мелкими деталями доставляет все меньше удовольствия и все сильнее становится похожа на работу. А почему еще такое количество программистов стремятся стать менеджерами... какое удовольствие можно найти в этом?

3. *Чем вы старше, тем больше в вас накапливается напряжения.*

А теперь действительно неприятная мысль! Неужели, приобретая опыт жизни, мы так мало узнали о ней, что жить нам становится чем дальше, тем тяжелее, а не легче? С возрастом способность получать удовольствие должна, в каком-то смысле, совершенствоваться. Что-то я не вижу, глядя на своих ровесников, чтобы это было так.

Список можно продолжить. В этом я рассчитываю на вас, дорогие читатели. А сейчас я закончу этот очерк, рассказав, что заставило меня сосредоточиться на этом вопросе.

Как бы это лучше сказать: у меня вырос программист – второе поколение. Тридцать лет тому назад у меня появился сын, и черт меня побери, если он не решил пойти по стопам отца, чтобы копаться в программном коде, разгребать завалы багов и все такое. Итак, вот программист нового поколения, и он носит мое имя. Я наблюдаю развитие его карьеры и вижу, как он получает удовольствие от работы. Совсем как я когда-то... ностальгия и дежавю.

Его рассказ о кольце защиты от записи – вот что меня зацепило. Как они сидели однажды в пятницу в конце рабочего дня в своем програм-

мистском офисе, отдыхая после недели напряженной работы. Можете не сомневаться, что кто-то из них бросил кому-то кольцо защиты от записи (ведь вы знаете, что такое кольцо защиты от записи? Пластмассовая штучковина с неудачным названием, которая, когда она установлена на место, разрешает запись на ленту и не разрешает, когда не установлена). И как, что неудивительно в такой компании, бросок был возвращен. Кольца защиты от записи летали по воздуху, как баги в неотлаженной программе. В игру вошел даже вице-президент компании (вы знаете, как невелики бывают современные программистские компании и как молоды бывают их руководители!). Мой сын, широко улыбаясь, безмятежно рассказывает мне эту историю. А я слушаю его, анализирую свою реакцию и понимаю, что испытываю *зависть*. Передо мной проносятся счастливые, безмятежные дни начала *моей* карьеры, когда мы травили байки, стреляли резинками, играли блиц-партии в шахматы, как нам и было положено самим богом. Так что же случилось?

Нет, правда, что произошло? Я стал более опытным. Я стал больше зарабатывать. Я стал чувствовать себя более ответственным. Я написал несколько книг, вел одну или две колонки. Я занимался исследованиями, написал одну-две работы. Я добился *успеха*, совсем как это должно быть в мечтах мальчишек и девчонок всей Америки.

Но я думаю, что по дороге я что-то потерял. Не поможете ли вы мне найти это?

Ода вечно юному ПО

Довольно давно я написал очерк («Не поможете ли вы мне найти это?»), в котором сокрушался по поводу неприятностей, связанных со старением в индустрии ПО. Нет, речь шла не о биологическом возрасте, а о том, что наше общество ожидало от нас, что с возрастом мы научимся мыслить глубже и станем более умелыми организаторами и руководителями.

В качестве иллюстрации я рассказал историю о том, как мой сын, работающий в индустрии программирования, сидел с коллегами в пятницу в конце рабочего дня в офисе, и как они отдыхали, перебрасываясь кольцами защиты от записи. «Куда ушло то время, которое мы проводили так беззаботно?» – сокрушался я. И просил читателей высказаться по этому поводу.

Что они и сделали. Проблема старения в обществе и в системе, которые ожидают определенного поведения от людей старшего возраста, задела чувствительную струну. Здесь я хочу привести выдержки из лучших комментариев. Когда будете их читать, представьте себе, что игра-

ет ненавязчивая музыка, какая-нибудь «Ода вечно юному ПО». Музыка оживленная, тоску не наводит, но в ней есть нотка грусти. Представьте себе, например, песню Барбары Стрейзанд «Happy Days Are Here Again», где вполне позитивный текст поется в ритме похоронного марша.

Может быть, лучше других высказался Крис Торкильдсон (Chris Torkildson) из Миннесоты:

Первые десять лет жена обвиняла меня в том, что я люблю свою работу. Она говорила, что никто не должен любить работу, если ему платят. Я всегда отвечал, что если не любить работу, то от тоски с ума сойдешь. Сейчас я думаю, что больше не люблю свою работу.

Я проанализировал события, которые происходили в те дни, когда я приходил домой в отличном настроении, и в те, когда бывало наоборот, и пришел к следующим выводам:

Вы чувствуете подъем, когда удастся решить трудную задачу. Сложная работа программиста связана с необходимостью решать тысячи задач, то есть с массой возможностей поднять настроение. У менеджера программных проектов задач намного меньше, просто они гораздо крупнее. И совсем не такая высокая вероятность, что они будут решены.

Мы получаем удовольствие от движения. Детям нравятся скоростные велосипеды и американские горки. Программистам нравится движение вперед, эволюция, связанная с их работой. Вам приятно, если кто-то просит вас изменить что-нибудь в программе и вы можете сделать это за какую-то пару часов. В крупных проектах добиться движения значительно труднее. Все эти руководящие комитеты, аналитические группы, согласования бюджета и т. д. сильно замедляют ход событий.

Вам хорошо и комфортно, когда хорошо и комфортно людям, с которыми вы работаете. Быстрая работа вместе с первоклассными программистами, которые получают удовольствие от того, что они делают, увлекает и захватывает. А когда вам достается роль менеджера проекта, вы получаете «синдром полицейского», потому что вас окружают исключительно несчастные люди...

Откликнулся также Бенджамин Дорз (Benjamin Doors) из города Вила-Парк (штат Иллинойс). Как и Крис, он подошел к проблеме аналитически:

К сожалению, тому, кто играет роль шестеренки в большом механизме, трудно поддерживать в себе приятные ощущения. Бюрократы почему-то не понимают, что некоторые люди вроде вас считают, что должны получать от работы положительные эмоции. Положительные эмоции — это личное удовлетворение от выполненной работы; товарищеский дух, свойственный командной работе; радость случайного творческого открытия и его применение к решению практических задач...

Я надеюсь, что вы готовы поступиться некоторой долей успеха ради удовольствия.

Но Уиллард (Билл) Холден (Willard (Bill) Holden) из Сан-Хосе (штат Калифорния) (бывший региональный представитель ACM) заглянул в самую суть:

Даже не пытайтесь; как говорится, «нельзя войти в одну и ту же реку дважды». Смотрите на вещи реально – отрасли не нужны старперы вроде нас, мы обходимся слишком дорого. А если мы соглашаемся на зарплату поменьше, то они думают, что мы, может быть, не так уж хорошо работаем. Мы находимся в безвыходной ситуации и должны принять неизбежное.

Кстати, Боб, это неизбежное не так уж плохо. Вот ты, например, ведешь единственную интересную колонку в ежемесячном издании...

К сказанному Билл присовокупил копию сообщения – первоапрельского розыгрыша, которое он разослал некоторым из своих друзей и в котором сообщалось о вымышленном событии, якобы радикально изменившем его жизнь:

...Ей 22 года, она умна и прекрасна. Мы должны пожениться в июне. В качестве свадебного подарка мы получим виллу на частном греческом острове. А еще у нас будет большая квартира с кондиционером в Афинах...

Этот человек умеет мечтать! (Я почти слышу, как набирает силу ритм «Happy Days Are Here Again»!)

Спасибо всем, кто откликнулся. Может быть, я и не смогу разрешить эту проблему (разве что исчезну в фантазиях Билла Холдена!), но теперь я понимаю ее намного лучше.

А вы? Вы получаете от программирования столько же удовольствия, сколько и раньше? И если нет, то что вы делаете? Я бы очень хотел узнать.

Ретроспектива

Вы читаете ретроспективу, посвященную ретроспективе! Дело в том, что в первом издании глава «После боя» как бы суммировала материал всей книги. А сейчас я анализирую ту, давнишнюю ретроспективу.

В этой главе есть один очерк («О том, насколько важны прикладные области»), который заставляет меня буквально раздуваться от гордости. Давно, в начале 90-х, когда вышло первое издание книги, многие – особенно университетские преподаватели и ученые (и даже некоторые практики) – верили, что существуют универсальные методики реше-

ния задач, которые (если мы сможем их определить) помогут нам создавать программное обеспечение для любых прикладных областей.

Структурные методики не были ориентированы на конкретные прикладные области. Как и все остальные, которые с тех пор были изобретены в помощь практикам. Сейчас почти любой язык программирования не должен быть предназначен ни для одной конкретной прикладной области. И вы не удивитесь этому, пока не вспомните, что в свое время Фортран и Кобол были сугубо специализированными языками, предназначенными для научных/инженерных расчетов и для обработки бизнес-данных соответственно.

Мне приятно сознавать, что очень многие с тех пор приняли доктрину прикладной специализации ПО, которой эта книга так активно пробивала дорогу в то далекое время. Конечно, я бы с удовольствием приписал авторство этой доктрины себе, но это было бы неправильно. Уже давно люди начали понимать, что разные прикладные области требуют разных подходов, и очень многие начали говорить то же самое, что я говорю здесь. Из высказываний, иллюстрирующих зарождающуюся поддержку прикладной специализации ПО, мне больше других нравится принадлежащее П. Дж. Плогеру (P. J. Plaugher): «Любой, кто верит в “гуттаперчевое ПО”, должен рекламировать колготки».

Конечно, сделанное мной в этом очерке предсказание, что программная инженерия распадется на несколько специализированных областей, совершенно ошибочно. Существует очень устойчивое ядро универсального программного обеспечения, а количество инструментов и технологий, ориентированных на конкретные прикладные области, весьма невелико. (Разве я не сказал раньше, что самый верный способ выставить себя на посмешище состоит в том, чтобы предсказать что-либо?)

Конечно, этот вечный разговор об универсальных подходах в программировании ведется неспроста – для этого есть причина. Если мы вместо доктрины универсальности ПО примем доктрину его прикладной специализации, то жизнь теоретиков и практиков программной инженерии чрезвычайно осложнится.

Только вообразите! Нам придется изучить каждую прикладную область и выявить ее специфические особенности и потребности. Многие ученые стараются не вступать в плотное взаимодействие с практикой в любом обличье, и от мысли, что им придется взаимодействовать не просто с практикой, а с ее конкретными гранями, в их глазах может промелькнуть тень ужаса!

Я заканчиваю эту часть книги очерком «Ода вечно юному ПО», содержащим грустную мысль о том, что чем дольше вы трудитесь в индуст-

рии ПО, тем меньше радости и удовольствия остается в ней для вас. Хотел бы я сказать, что эта мысль, как и многие другие, высказанные мной в этих с годами устаревших ретроспективных заметках, тоже ошибочна или хотя бы уже не актуальна. Но не могу!

Я не сомневаюсь, что те, кто приходит в такую отрасль, как программная инженерия, и кто понимает, что находится у самого основания чего-то, что очень важно для человечества, способны получать от решения своих задач больше удовольствия, чем старые консерваторы вроде меня. Старые консерваторы – как мне не хочется это говорить, – которые со времени выхода первого издания «Программирования и конфликтов» постарели еще на 15 лет и стали еще меньше заботиться о получении удовольствия!

Докажите мне, что я ошибаюсь. Ветераны, которые перечитывают этот материал, и новички, которые видят его впервые, помните, что пункт «получение удовольствия» должен занять высокое место в списке приоритетов ваших программистских задач. Мы хорошо знаем, что «подслащенную пилюлю проще проглотить». Добавьте сладости в свою работу, пусть вам будет интересно жить. Скажите, что это рецепт Роберта Л. Гласса!

Эпилог

Написание сборника очерков может оказаться небезопасным делом.

Во-первых, многим не нравится слово «*очерк*». В лабиринтах нашего эмоционального восприятия оно, как правило, приобретает значение «сухой и скучный». Надеюсь, что эти очерки не сухие и не скучные; во всяком случае, не для меня (да и какой автор способен беспристрастно судить о своих работах?!).

Однако есть и более важный аспект, касающийся их содержательной ценности. Сборник очерков очень приятно почитать немного и отложить, а потом возобновить чтение в какой-нибудь дождливый день, когда в камине горит огонь, а под рукой любимый напиток. (Так я читал «Мифический человеко-месяц» Брукса, «Человеческий фактор» Демарко и Листера и книгу Джерри Вейнберга «Understanding the Professional Programmer» (Профессиональный программист), и каждый раз я получал массу удовольствия.)

Однако окончательная оценка складывается, когда книга в конце концов прочитана: имело ли употребленное литературное блюдо какую-то настоящую питательную ценность? Сборник очерков опасен тем, что может оставить после себя пресловутое чувство мимолетного насыщения: какая-то пара часов – и как будто ничего и не было.

Что же я сказал (если я вообще что-то сказал) в этой книге? Перед этим вопросом меня поставил рецензент. Вот какой ответ пришел мне в голову. Книга представляет собой квинтэссенцию мыслей, вложенных в эти очерки, организованную в виде коллекции коротких цитат, которые я адресовал разнородной целевой аудитории в соответствии с моими представлениями о ее запросах. Здесь есть одна опасность. Вы можете запомнить вкус этих концентрированных мыслей, пода-

ваемых на десерт, а не сами очерки, тщательно продуманные и подготовленные.

Ну что же, так тому и быть. Однако, для того чтобы зафиксировать главное и учитывая приверженность нашего общества ценностям фаст-фуда, я подаю на стол экстракт из блюд, которые мы с вами только что разделили.

И пусть главное блюдо – как и эти «мини-десерты» – не вызовут у вас несварения!

Роберт Л. Гласс

Подборка главных мыслей в этой книге

Техническому специалисту-практику

Иногда мудрость практики больше, чем мудрость теории. Поэтому время от времени теория предлагает решения, не пригодные для практических задач. Но когда теория оказывается мудрее – а это неизбежно наступает рано или поздно, – практике следует прислушаться к ней.

К технологиям, которые сулят индустрии программирования очень скромные, но все равно важные преимущества, относятся повторное использование кода, прототипирование, инкрементная разработка, языки 4GL, CASE-инструменты и адекватно выбранные метрики. О том, что выгоды скромные (хотя обещания утверждают обратное), мы знаем, потому что (1) есть экспертные оценки тех, кто реально работал с этими технологиями, и (2) появились экспериментальные данные.

Отсутствует связующее звено между методическими подходами к проектированию ПО и методами представления, предназначенными для фиксирования проекта в графическом виде. Это связующее звено – когнитивное, или творческое, проектирование, которое представляет собой самую важную часть процесса проектирования.

Удаление ошибок из ПО сопряжено с трудностями и редко бывает абсолютно успешным. Однако тщательное тестирование и расчетливый подход к проведению экспертиз помогут нам создавать качественное ПО, которое можно будет считать надежным.

Сопровождение ПО, несмотря на все приложенные усилия, по-видимому, до сих пор недопонято и недооценено как представителями менеджмента, так и научным сообществом. Его считают проблемой, тогда как на самом деле оно представляет собой ее решение; это вид обслуживания, которое предоставляется только в индустрии ПО. Сопровождение и качество ПО изолируются друг от друга, тогда как на самом деле они переплетаются друг с другом теснейшим образом. Не признается один

из немногих правильных принципов программной инженерии – управление из одной точки.

Качество ПО, несмотря на все наше пустословие, так и не обеспечивается по-настоящему. Представители менеджмента и научного сообщества считают, что обеспечить качество ПО должны менеджеры, признавая в то же время, что только технические специалисты знают, как это сделать. Усилия концентрируются на измерении того, что можно измерить, а не на самом качестве. Малоизвестными остаются лучшие исследования по метрикам качества, и в результате нам приходится иметь дело с метриками качества, значимость которых сомнительна.

Одним из немногих настоящих успехов за последнее десятилетие стали пользовательские интерфейсы.

Мы уделяем много внимания более крупным и совершенным наборам инструментальных средств, но равнодушны к насущной потребности разработчиков ПО в минимальном стандартном наборе – своего рода молотках, пилах и плоскогубцах ПО.

Возможно, самая злая напасть технологий создания ПО – это изменчивость требований к нему. Трудно попасть в движущуюся мишень.

Что происходит с удовольствием от программирования? Почему оно исчезает: мы стареем, устаем от дисциплины, его душит бюрократия или почему-то еще? Сколько людей предпочитают лишиться какой-то доли успеха, чтобы получать удовольствие от работы?

Менеджеру-практику

Надежды на революционное повышение качества ПО и производительности труда ухудшают способность по достоинству оценивать и принимать более скромные методологические достижения. Более того, почти всем подобным надеждам суждено разбиться вдребезги.

Самые серьезные проблемы индустрии ПО заключаются в скверном менеджменте, а не в плохих технологиях.

Стандарты помогают нам создавать более качественное ПО, но это как раз тот самый случай, когда «меньше», возможно, означает «лучше». Если стандартов слишком много, то их труднее претворять в жизнь и, следовательно, ими мало кто руководствуется.

Величие оказалось недолговечным. Великими в истории индустрии ПО сначала были программисты, потом исследователи, а вслед за ними предприниматели. Сейчас нельзя предсказать, какими будут истоки величия следующего поколения.

О производительности не говорил только ленивый, но очень немногие действительно что-то сделали для ее повышения.

Неправильные оценки в большей степени отвечают за так называемый «кризис ПО», чем любой другой фактор. Мы пытаемся разрешать административные проблемы при помощи технологий.

Общепризнано, что ключом к повышению качества ПО и производительности труда в нашей отрасли являются отличия между индивидуумами. Беда в том, что мы не знаем, что с этими отличиями делать.

Исследователю

На ранних стадиях развития новой отрасли знания один из лучших способов создания теории состоит в изучении практического опыта лучших специалистов.

Заявки на исследовательские гранты и программы предпринимательских тренингов обещают намного больше революционных прорывов, чем их происходит на самом деле. Эти обещания порождены либо невежеством, либо непорядочностью, и в них обычно тонут более реальные, но менее захватывающие работы.

Может быть, Кобол действительно очень плохой язык, но все остальные, созданные для этой прикладной области, намного хуже. Проектирование улучшенного Кобола может оказаться одним из самых перспективных исследовательских проектов современности.

В большинстве средств информационного поиска зияют прорехи, через которые улечивается значительный объем информации. В результате исследователи ничего не знают о важных работах, выполненных по государственным контрактам и в промышленности (например, поставщиками).

Наша озабоченность подкреплением своих позиций ссылками на прошлые работы может помешать опубликованию плодотворных идей.

Сообщества исследователей и практиков практически прекратили сколько-нибудь значительное общение друг с другом. Существует отчаянная обоюдная потребность в людях, которые наведут мосты через эту пропасть.

Эта пропасть отчасти обусловлена недостатком эмпирических исследований. Будучи лишенной экспериментальной составляющей, информатика не является настоящей наукой, а программная инженерия – настоящей инженерной дисциплиной.

Преподавателю

Недостаточно преподавать проектирование ПО как методологию и способ представления. Мы в достаточной степени понимаем творческие аспекты проектирования, чтобы преподавать их тоже.

Наука индустрии ПО упрекает практику в том, что та тормозит теорию, а потом, когда дело доходит до обучения новым понятиям, таким как языки 4GL и CASE-инструменты, говорит, что не может себе позволить такую роскошь.

Институт программной инженерии (SEI) владеет постоянно растущим объемом учебного материала, очень важного для образования в сфере программной инженерии. И эти материалы по большей части бесплатны!

В большинстве образовательных учреждений попросту не преподается сопровождение ПО. Преподаватели не понимают, что эта деятельность требует солидной технической подготовки.

Программная инженерия оказалась областью, в которой мы преподаем решение задач – намного более универсальную тему. К примеру, так называемый «жизненный цикл ПО» есть не что иное как последовательность действий, выполняемых в ходе решения *любой* задачи. Почему решение задач не входит в базовую академическую программу?

Понимание специфики прикладной области имеет намного большее значение, чем мы думали. Адекватное понимание различий между прикладными областями может логически привести нас к специализированным прикладным методологиям, инструментам, языкам и... возможно, даже образованию.

Приложение

Многие очерки, вошедшие в первое издание «Программирования и конфликтов», уже были опубликованы в других изданиях. Некоторые из них публиковались в моих колонках в *System Development* (Software Reflections) или в *The Journal of Systems and Software* (Editor's Corner), другие – в *Software Magazine* (Software Folklore), *Computerworld* или *Datamation*.

Ниже приведена история публикации этих статей. В список не включены новые очерки из серии «Ретроспектива», добавленные в сборник «Программирование и конфликты 2.0» и впервые опубликованные в нем.

Обозначения:

CUPS	Confessions of a Used Program Salesman, Will Tracz (Признания Уилла Трэкса, торговца поддержанным ПО) (готовится к печати)
CSM	Труды конференции по сопровождению ПО (Conference on Software Maintenance), 1989 г.
CW	<i>Computerworld</i>
D	<i>Datamation</i>
GCS	Заметки технической группы Германского компьютерного общества по отказоустойчивым вычислительным системам (German Computer Society Technical Interest Group on Fault-Tolerant Computing Systems)
JSS	<i>Journal of Systems and Software</i>
SD	<i>System Development</i>

SESY	Труды итальянского симпозиума по программной инженерии (Italian Software Engineering Symposium), 1989 г.	
SM	<i>Software Magazine</i> (ранее назывался <i>Software News</i>)	
SMN	<i>Software Maintenance News</i>	
TT	Материалы семинара по передаче программных технологий (Workshop on Software Technology Transfer), 1987 г.	

Обзор театра военных действий

Что первично: теория или практика	JSS, SD
«Опасно и обманчиво»: взгляд на исследования в области программной инженерии сквозь призму работ Дэвида Парнаса	JSS, SD
«Серебряной пули нет»: взгляд на исследования в области программной инженерии сквозь призму работ Фредерика Брукса	JSS, SD
Что думают умнейшие и лучшие	—

Из окопов технологии

Когнитивный взгляд:	
проектирование ПО с другой точки зрения	CW
Некоторые размышления об ошибках в ПО	GCS, SD, JSS
Экспериментальный аспект устранения программных ошибок	—
Многоликое тестирование	—
Качество ПО и сопровождение ПО. Какая связь?	SD, JSS
Сопровождение ПО –	
это решение, а не проблема	SD, JSS, CSM, SESY
Управление из одной точки	SD
«Дружелюбный к пользователю» –	
модное выражение или прорыв?	SD

Новейшие вооружения

Повторное использование: готовые части ПО –	
ностальгия и дежавю	D, CUPS
Автоматическое программирование – слухи с вечеринки?	JSS, SD
Некоторые мысли о прототипировании	SD
Стандарты и «блюстители» стандартов:	
они действительно помогают?	JSS, SD

Джентльменский набор разработчика	—
На всякий CASE: Взгляд на последний «прорыв» в технологии программирования	SD
Сравнение инструментов CASE и 4GL: что в сухом остатке	—
Зачем писать компиляторы	SM
Языки программирования высокого уровня. Насколько высок их уровень	JSS, SD
Должны ли мы готовиться к доминированию 4GL	JSS, SD
Сомнения по поводу Кобола	JSS, SD

Из штаба

Покорение вершин индустрии ПО	SD
Новый взгляд на продуктивность ПО	JSS, SD
Производительность и теория G	SM
Управление программными проектами – теория W, принадлежащая Барри Боэму	SD
Повышение производительности труда в индустрии ПО: кто чем занимается	SD
Метрики ПО: о громоотводах и накопленной напряженности	JSS, SD
Как измерить качество: меньшее притворяется бóльшим	—
Можно ли ВНЕДРИТЬ качество в программный проект	CW
Легенда о плохом программном проекте	SD, JSS
А вы бы купили у короля Людовика автомобиль с пробегом?	—
Подготовная консалтинга	—
Какие прогнозы давали предсказатели раньше	CW
Поддержка пользователей: все не так просто, как кажется	SMN

Из лабораторий

Структурные исследования (немного ироничный взгляд)	JSS, SD
Проблема информационного поиска	JSS
Неувязочка: некоторые «за и против (как будто больше не о чем!) ссылок	JSS
А что в следующем году?	
Как исследуют развитие технологий	JSS, SD

Передача технологий ПО: процесс, обладающий многими недостатками, или неровная дорога к производительности SD, JSS, TT

Мифология передачи технологий SD, JSS

Изучение ПО: новый источник информации SD

Открытое письмо профессору компьютерных наук SM, SESY, SD

Арена после боя

Как компьютерная наука может стать настоящей наукой, а программная инженерия – настоящей инженерией JSS, SD

Моя блестящая (или нет?) идея,
которую я назвал «решение задач» JSS, SD

Почему проваливаются программные проекты —

О том, насколько важны прикладные области —

Не поможете ли вы мне найти это? SM

Ода вечно юному ПО SD

Алфавитный указатель

С, S, T

CMM (Capability Maturity Model), Модель развития функциональных возможностей, 202
SEI (Software Engineering Institute), Институт программной инженерии, 194, 202, 206, 208
Software Maintenance Association, Ассоциация специалистов по сопровождению ПО, 198
SPC (Software Productivity Consortium), Консорциум продуктивности ПО, 206
The Programmers Workbench, инструментальное средство, 196

А

абстракции данных, 195
автоматическое программирование, 219
анализ зависимостей, 197

Б

Болл, Ричард (Ball, Richard), 199
Бозм, Барри (Boehm, Barry), 200
Брукс, Фредерик (Brooks, Frederick), 190, 214, 229
Бэсили, Виктор (Basili, Victor), 201

В

Вейнберг, Джерри (Weinberg, Jerry), 229
Весси, Айрис (Vessey, Iris), 219
выгорание, 223
Высоцкий, Виктор (Vyssotsky, Victor), 218

Г

Гейтс, Билл (Gates, Bill), 198

Д

Демарко, Том (DeMarco, Tom), 191, 229
Дорз, Бенджамин (Doors, Benjamin), 225

З

Зелковитц, Марвин (Zelkowitz, Marvin), 201

И

индивидуальные отличия, возможности отдельных работников и команд, 219
инструментальные средства, 180
информатика, вычислительная техника, 200, 206, 222, 232
исследования, 188, 201, 202, 216, 220, 232
исследовательский процесс, 177

Й

Йордон, Эд (Yourdon, Ed), 222

К

капризы и заблуждения, 217
качество ПО, 232
Келли, Джон (Kelly, John), 218
Кобол, язык программирования, 219, 227
когнитивная подгонка, 219

Л

Ламмерс, Сьюзен (Lammers, Susan), 198
Листер, Тимоти (Lister, Timothy), 191, 229

М

Майерс, Гленфорд (Myers, Glenford), 191
маркетинг, рыночные исследования, 189
менеджмент, 211, 217
методологии, 201, 226
метрики ПО, 176, 180
Модель развития функциональных возможностей, проект SEI, 202

О

обезличенное программирование, 178
оборонительное проектирование, 197
оборотни, 214
обработка исключений, 195
О'Брайен, Патрик (O'Brien, Patrick), 190
обратная разработка, 197
объектное ориентирование, 195
отказоустойчивое ПО, 196
оценка, 212

П

Парнас, Дэвид (Parnas, David), 206
передача технологий, 188, 190
переусложненное проектирование, 196
Плогер, П. Д. (Plaugher, P. J.), 227
ПО реального времени, 218
практики, 188, 220, 227, 232
Прессман, Роджер (Pressman, Roger), 190
провалы, 217
программный код
 с табличным описанием данных, 195
 управляемый текстовыми файлами, 195
проектирование ПО, 198, 219
производительность, 188
пропаганда, 204
прорывы, 189, 217

Р

Редвайн, Сэм (Redwine, Sam), 184
реконструкторы, 197
решение задач, 209, 218
Риддл, Билл (Riddle, Bill), 184
Римский центр развития авиации, 180

С

серебряная пуля, 217
совет по управлению изменениями, 197
сопровождение ПО, 200, 233
специализированные языки, 222, 227
спецификации ПО, 194
ссылки, 181, 182
Сэмюэлсон, Памела (Samuelson, Pamela), 193
Сэнден, Бо (Sanden, Bo), 219

Т

теория, 200, 202, 220
тестирование, 197
 анализаторы тестового покрытия, 215
Торкильдсон, Крис (Torkildson, Chris), 225
требования к ПО, 192, 217

У

универсальные решения, 218
университет Сиэттла, 187
управление из одной точки, 195
управление конфигурациями, контроль версий, 197
утверждения, 196
учебный модуль, 193, 201

Ф

фантомное ПО, 210
Фортран, язык программирования, 219, 227
Фэйрли, Ричард (Fairley, Richard), 190

Х

Холден, Уиллард (Holden, Willard), 226

Э

эксперименты, 206
эмпирические исследования программистов, 206

Я

языки четвертого поколения (4GL), 219

Основано в 1995 году

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.



Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

главный офис издательства –

В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

московский филиал издательства –
ул. Беговая, д. 13 (м. Динамо),
тел. (495) 638-5305

через Интернет <http://www.symbol.ru>

Бесплатный каталог книг высылается по запросу.

GIMBO®
www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-148-6, название «Программирование и конфликты 2.0» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.