Ольга Назина

Что такое ТЕСТИРОВАНИЕ

курс молодого бойца

- Изучите приложение
- Придумайте список тестов
- Найдите ошибки в программе
- Оформите тест-кейсы и чек-листы
- Опишите баги и улучшения
- Создайте свое портфолио

Ольга Назина

Что такое ТЕСТИРОВАНИЕ

курс молодого бойца

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.415.53
ББК 32.973.26-018.2
Н19

Назина О. Е.

Н19 Что такое тестирование. Курс молодого бойца. — СПб.: БХВ-Петербург, 2022. — 592 с.: ил.

ISBN 978-5-9775-6835-7

Уникальная книга-тренинг по тестированию программ, охватывающая весь необходимый тестировщику спектр знаний с азов до сложных концепций. Рассматриваются виды и методики тестирования, способы поиска ошибок в программах, оформления тест-кейсов и чек-листов, описания выявленных недостатков и предлагаемых улучшений. Книга содержит домашние задания, выполнив которые читатель освоит тестирование ПО на практике и соберет портфолио, необходимое для последующего трудоустройства.

Для начинающих тестировщиков ПО

УДК 004.415.53
ББК 32.973.26–018.2

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Корректор	*Анна Брезман*
Оформление обложки	*Зои Канторович*

Подписано в печать 02.12.21.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 47,73.
Тираж 1200 экз. Заказ № 2951.
«БХВ-Петербург», 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО «Принт-М», 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1.

ISBN 978-5-9775-6835-7

© Назина О. Е., 2021
© Оформление. ООО «БХВ-Петербург», ООО «БХВ», 2021

СОДЕРЖАНИЕ

Пара слов от автора 18

Введение в тестирование ПО 19

Что такое ПО? 20
Кто создает программы? 21
Один участник: автор = разработчик 21
Два участника: автор идеи + разработчик 22
Много участников 22
Что делают все эти люди? 25
Чем занимается тестировщик? 26
Почему тестирование так важно? 28
Вопросы для самопроверки 31
Ответы на вопросы для самопроверки 32

Глава 1. Исследование продукта 35

Что продукт умеет? 36
Зачем вообще нужны программы? 37
Как накидать тестов на что-нибудь? 40
Какие вопросы задавать? 41
Открытые и закрытые вопросы 42
Примеры 44
Какие вопросы НЕ задавать? 45
Сколько вопросов задавать? 46
Как ввести собеседника в контекст вопроса? 48
Как часто задавать вопросы? 53
Инструменты исследования 55
Блокнот 56
Интеллектуальная карта (mind map) 56

Домашнее задание ... 62
Вопросы для самопроверки ... 62
Магнитики на холодильник ... 62
Портфолио ... 63
Ответы ... 63
Магнитики на холодильник ... 63
Ответы на вопросы для самопроверки ... 64

Глава 2. Тест-кейсы и чек-листы ... 65

Проектирование тестов ... 66
Каким должен быть тест? ... 66
Приоритеты выполнения тестов ... 68
Позитивное тестирование ... 69
Негативное тестирование ... 71
Где граница «позитив-негатив»? ... 75
Повторим... ... 76
Так как оформлять-то? ... 77
Тест-кейсы ... 77
Название ... 79
Предварительные шаги ... 86
Шаги ... 96
Результат ... 99
Что еще? ... 115
Стандартные ошибки при оформлении тест-кейсов ... 117
Набор тест-кейсов ... 122
Особенности тест-кейсов ... 122
Когда применять тест-кейсы? ... 122
Инструменты для составления тест-кейсов ... 123
Чек-листы ... 124
Что такое чек-лист? ... 124
Как оформлять чек-лист? ... 126
Как составлять описание проверки? ... 126
Зачем в чек-листе нужны примеры? ... 128
Какой результат писать в чек-листе? ... 130
Особенности чек-листов ... 139
Плюсы и минусы ... 140

Когда применять чек-листы? ... 140
Инструменты для оформления чек-листов ... 140
Сравним тест-кейсы и чек-листы ... 141
Идея 1: дите или подросток ... 143
Идея 2: IKEA ... 143
Чит-листы ... 144
Вопросы для самопроверки ... 144
Портфолио ... 145
Ответы на вопросы «на подумать» ... 145
Ответы на вопросы для самопроверки ... 146

Глава 3. Классы эквивалентности и граничные значения ... 147

Несколько вступительных слов ... 148
Тест-дизайн ... 149
Тест-дизайн — это не наука ... 150
Классы эквивалентности ... 152
Классы эквивалентности — что это? ... 152
Классы эквивалентности через Золушку ... 154
Идеи для тестов ... 155
Когда остановиться? ... 156
Эффект пестицида ... 156
Граничные значения ... 158
Границы на числовой оси ... 158
Границы в нецелых числах ... 160
Границы там, где «нет числа» ... 160
Ноль ... 161
Типы границ ... 162
Инструменты ... 163
Поиск технологической границы ... 163
Снятие ограничений на клиенте веб-приложения ... 163
Автоматическое заполнение полей ... 164
Типичные ошибки ... 164
Классы: символы, циферки, перемешал ... 164
Ищем границу сверху — аж 1000 символов ... 165
Если границы нет, в чек-лист не пишем ... 166
Начинаем тестировать с длины строки ... 166

Не тестируем пограничные значения ... 166
Границы применяем лишь в длине строки ... 166
Не проверяем нечисловые значения ... 167
Вопросы для самопроверки ... 167
Домашнее задание ... 167
Банковская карта ... 167
Сортировка по строке ... 168
Портфолио ... 169
Ответы на вопросы для самопроверки ... 169

Глава 4. Анализ тестов ... 171

Разберемся с определениями ... 172
Анализ тестов ... 172
Что мы изучаем в книге? ... 172
Как выкидывать лишние тесты? ... 174
Объединить позитивные тесты ... 174
Выкинуть дубли ... 176
Не тестировать один функционал 10 раз ... 177
Типичные ошибки ... 178
Объединили негатив ... 178
Понапихали в один тест всего и сразу ... 179
Техника pairwise ... 181
Инструменты ... 181
Плюсы и минусы метода ... 182
Когда применять? ... 182
Портфолио ... 182
Ответы на вопросы для самопроверки ... 183

Глава 5. Баг-трекинг ... 185

Что такое баг? ... 186
Нашел баг, что дальше? ... 188
Процесс баг-трекинга ... 188
Инструменты ... 189
Workflow: жизненный цикл задач ... 190
Как заводить задачи в баг-трекер? ... 191
Локализуйте проблему ... 193

Придумайте короткий и емкий заголовок ... 193
Приложите скриншот ... 195
Опишите шаги воспроизведения и результат ... 196
Обоснуйте ожидаемый результат ... 197
Что еще? ... 198
Тотальная паранойя — друг тестировщика! ... 201
Сколько задач заводить? ... 202
Локализация ошибок ... 202
Что такое локализация? ... 202
Как локализовывать ошибки? ... 203
Минимальные данные для воспроизведения бага ... 207
Итого про локализацию ... 208
Оформление задач ... 208
Оформление названия ... 208
Оформление описания бага ... 210
Оформление описания улучшения ... 219
Как правильно вложить аттач? ... 220
Дополнительные поля ... 221
Пример оформления ... 222
Типовые ошибки ... 224
Кроссбраузерность ... 224
Concurrency (параллельная работа) ... 225
Валидация клиент-сервер ... 226
Буква «ё» ... 227
Как закрывать задачи? ... 227
Документация ... 228
Комментарий ... 229
Тестовые данные ... 229
Ретроспективный анализ ошибки, или Как анализировать пропущенные баги ... 230
Шпаргалки ... 231
От Павла ... 231
Плакат НЛО: Найти, Локализовать и Оформить ошибку ... 232
Ключевые моменты ... 233
Вопросы для самопроверки ... 233
Портфолио ... 234
Ответы на вопросы для самопроверки ... 234

Глава 6. Исследовательское тестирование 239

Что такое исследовательское тестирование? 240
Виды тестирования: краткие определения 240
Эвристики и мнемоники 241
Эвристики 241
Мнемоники 242
Исследовательские туры Уиттакера 243
Мои любимые туры 255
Вопросы для самопроверки 255
Портфолио 255
Ответы на вопросы для самопроверки 256

Глава 7. Тестирование документации 257

Какая бывает документация? 258
ТЗ — требования 258
Пользовательская документация 263
Примеры 264
Письма от системы 267
Сообщения об ошибках 269
Поп-ап сообщения 277
Предупреждения «что вводить» 280
Инструкция по установке 281
Описание полей 285
Маркетинговые материалы 286
Обучение: FAQ, презентации 286
Поздравляшки 289
Кнопки 289
Остальное 290
Подведем итоги 290
Как тестировать документацию? 292
Чек-лист проверки 292
Мнемоника CIRCUSMATTA 300
Примеры багов из жизни 300
Где искать документацию? 301
Вопросы для самопроверки 302
Портфолио 302

Глава 8. Создание документации: тестовой и не только 303

Тестовая документация..... 304
Тест-план..... 304
Test-plan или test-suite?..... 305
Беседа у камина..... 306
Отчет о тестировании..... 309
Проектная документация (ТЗ)..... 312
Шаблон компании..... 314
Вариант использования..... 314
Decision Table (таблица решений)..... 315
State & Transition Diagramm (схема состояний и переходов)..... 318
Другие диаграммы, схемы, картинки..... 319
Подведем итоги..... 320
Домашнее задание..... 321
Вопросы для самопроверки..... 321
Магнитики на холодильнике..... 321
Портфолио..... 323
Ответы на вопросы для самопроверки..... 323

Глава 9. Классификация тестирования.....325

Классификация: что это и зачем?..... 326
Что такое классификация?..... 326
Минусы классификации..... 328
Плюсы классификации..... 328
По знанию системы..... 330
Черный ящик (black box testing)..... 330
Белый ящик (white box testing)..... 331
Серый ящик (gray box testing)..... 332
По позитивности..... 333
Позитивное тестирование..... 334
Негативное тестирование..... 335
По целям (по объекту)..... 335
Функциональное тестирование..... 336
Нефункциональное тестирование (НФТ)..... 337
По исполнителям (по субъекту)..... 360
Альфа-тестирование..... 361
Бета-тестирование..... 361

По затраченному времени (дымовое тестирование) ... 365
Различия санитарного тестирования (Sanity), дымового (Smoke) и приемочного (Acceptance) ... 366
Тестирование нового функционала ... 368
Регрессионное тестирование ... 369
По степени автоматизации ... 370
Ручное тестирование ... 370
Автоматизированное тестирование ... 373
ПолуАвтоматизированное тестирование ... 373
По состоянию системы ... 374
Статическое тестирование (static testing) ... 374
Динамическое тестирование (dynamic testing) ... 375
По формальности ... 376
Тестирование по готовым тестам ... 376
Исследовательское тестирование ... 376
Сравним подходы ... 378
Когда какое тестирование выбрать? ... 378
Подведем итоги ... 379
Домашнее задание ... 380
Вопросы для самопроверки ... 380
Портфолио ... 381
Ответы на вопросы для самопроверки ... 381

Глава 10. Автоматизация тестирования ... 383

Чего мы в этой главе делать НЕ будем ... 384
Что такое автоматизация? ... 385
Продумать тесты для автоматизации ... 386
Расписать тесты по шагам ... 387
Написать скрипт ... 388
Поддержка автотестов ... 389
Когда автотест начнет ловить баги? ... 391
Пирамида автоматизации ... 393
Unit-тесты ... 393
API-тесты ... 395

UI-тесты ... 400
Подведем итоги ... 405
Автоматизация рутины ... 409
Инструменты полуавтоматизации ... 412
Унылые задачи ... 413
Вопросы для самопроверки ... 416
Портфолио ... 417
Ответы на вопросы для самопроверки ... 417

Глава 11. Организация процесса ... 419

Стоит ли вмешиваться в процесс? ... 420
Процессы в компании-гиганте ... 422
Кратко ... 422
Подробнее ... 422
Плюсы ... 425
Минусы ... 426
Процессы в стартапе ... 426
Кратко ... 426
Подробнее ... 426
Плюсы ... 428
Минусы ... 429
Процессы на аутсорсе ... 429
Кратко ... 430
Подробнее ... 430
Минусы ... 432
Подведем итоги ... 432
Домашнее задание: работа мечты ... 433

Глава 12. Как составить резюме? ... 435

Составить самому
или заполнить форму на сайте? ... 436
Структура резюме ... 437
Навыки ... 437
Опыт ... 440
Образование ... 453

Ключевые слова 455
Портфолио 455
Остальное 456
Подведем промежуточные итоги 458
Подстройка резюме под вакансию 459
Шаблоны для резюме 462
Сопроводительное письмо 463
Главное правило сопроводительного письма 463
Стандартные ошибки сопроводительного письма 464
Как написать письмо? 467
Пример хорошего письма 467
Подведем итоги 468
Домашнее задание 470
Вопросы для самопроверки 470
Резюме 470
Ответы на вопросы для самопроверки 471

Глава 13. Собеседование 473

Стоит ли делать тестовое задание? 474
Компания эксплуатирует кандидатов 475
Я потрачу время впустую 477
Подведем итог 479
Как делать тестовое задание? 479
Как попросить фидбек на тестовое? 481
Как подготовиться к собеседованию? 487
Как проходит собеседование? 491
Рассказ о проекте и компании 492
Рассказ о себе 494
Почему ушли с предыдущего места? 495
Расскажите подробнее, чем вы занимались 497
Расскажите про самый интересный баг 498
Тестовое «на дому» 499
Тестовое «в офисе» 500
В чем прийти? 501
Как себя вести? 502

Что спросят вас? ... 503
Что спрашивать вам ... 504
Домашнее задание ... 505

Глава 14. Куда развиваться? ... 507

Несколько вступительных слов ... 508
Направления развития ... 508
Понимание цикла разработки ... 509
Что там внутри? ... 509
Общая компьютерная грамотность ... 510
Ручное тестирование ... 510
Автоматизация ... 511
НФТ ... 514
Аналитика ... 514
Тест-дизайнер ... 515
О профессии ... 515
Книги и ресурсы ... 517
Автоматизатор ... 519
О профессии ... 519
Книги ... 520
Нагрузочник ... 521
О профессии ... 521
Книги ... 523
Безопасник ... 523
О профессии ... 523
Книги ... 524
Тестировщик usability ... 524
О профессии ... 524
Книги ... 526
Тест-менеджер ... 527
О профессии ... 527
Книги ... 528
Аналитик ... 530
О профессии ... 530
Книги ... 531

Оратор ... 531
О профессии ... 531
Книги ... 532
Мои бизнес-хаки ... 532
Швец, жнец, на дуде игрец ... 535
О профессии ... 535
Книги ... 537
«Мне некуда расти в компании!» ... 537
Применяйте техники тест-дизайна ... 538
Создавайте чит-листы ... 539
Пишите документацию ... 539
Улучшите процесс ... 540
Учитесь автоматизации ... 540
Организуйте внутреннее обучение ... 541
Соберите книжный клуб ... 544
Выступите на конференции ... 544
Подведем итоги ... 545
Домашнее задание ... 546

Глава 15. Всё обо всём ... 547

Несколько вступительных слов ... 548
Что такое API? ... 548
SOAP & REST API ... 550
Форматы передачи данных по API ... 550
Что такое клиент-серверная архитектура? ... 551
Цикл создания приложения ... 555
Что такое система контроля версий? ... 555
Что такое сборщик продукта? ... 557
Что такое сервер приложения? ... 558
Что такое CI (Continuous Integration)? ... 559
Что такое Docker? ... 561
Командная строка ... 562
Что такое Linux? ... 562
Где тренироваться? ... 563
Что такое bash / shell? ... 564

Что такое regexp? 564
Портфолио 565

Заключение 567

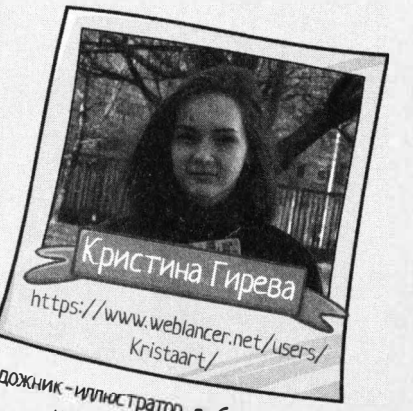
Когда есть требования 568
Изучить требования 568
Протестировать их! 568
Написать чек-лист проверок 569
Протестировать систему 570
Оформить результат 571
Когда требований нет 573
Изучить систему 574
Написать чек-лист проверок 574
Протестировать 575
Оформить результат 575
А что потом? 575
Несколько слов в завершение 576

Предметный указатель 579

Примечания 585

Наши художники

Художник-иллюстратор уже лет 10, люблю рисовать маскотов, зверьё и всё, что вызывает положительные эмоции



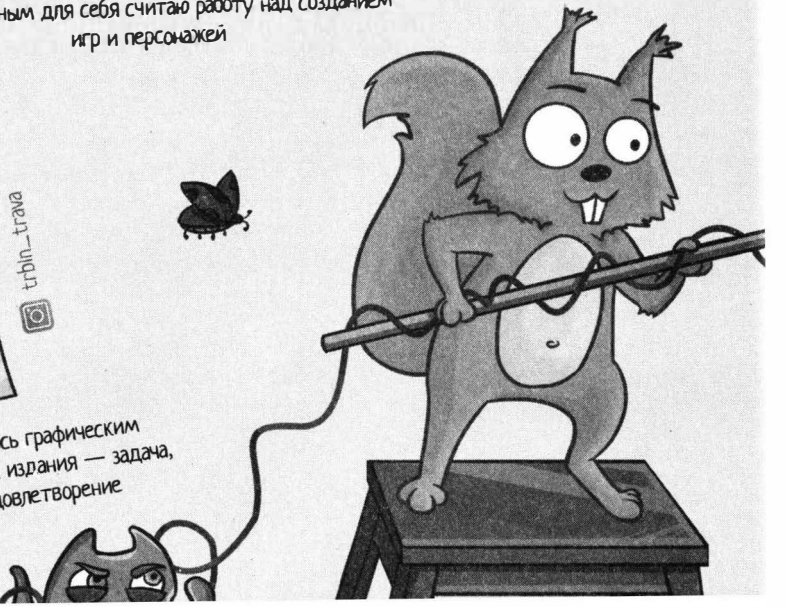
Я художник-иллюстратор, люблю развиваться и экспериментировать

gl_graphic_designer

Графический дизайнер/иллюстратор. Искусство — это моё хобби и смысл жизни. Работаю в разных стилях и направлениях. Приоритетным для себя считаю работу над созданием игр и персонажей

trbln_trava

Художник книги по образованию. Занимаюсь графическим и веб-дизайном. Создать жизнь на страницах издания — задача, решение которой приносит огромное удовлетворение



Привет, я Настасья — художник детской графики! Всем сердцем люблю создавать очаровательных бренд-персонажей, разрабатывать паттерны и, конечно же, иллюстрировать книги :)

С Викой я работала давно, а остальных нашла на Weblancer.ru

Автор книги

По образованию магистр экономики, но с детства любила рисовать и в долгих поисках себя всё-таки пришла к работе иллюстратора, чему безумно рада

dolce.png

Анастасия, 15 лет. Я — начинающий иллюстратор и полиглот. Также рисую иллюстрации к своему рассказу и набираюсь опыта

Иллюстратор, дизайнер

ПАРА СЛОВ ОТ АВТОРА

Привет!

Добро пожаловать в увлекательный мир тестирования. Раз вы держите эту книгу в руках, вам оно явно интересно! Я надеюсь, что моя книга поможет вам разобраться в новой области. А может, даже и работу получить. Если будете прилежно выполнять все домашние задания и создавать свое портфолио.

Хочу сразу предупредить вас о стиле книги. Я люблю писать просто о сложном. А еще больше я люблю картинки. Поэтому в моей книге их будет не просто много, а очень много! Они могут быть забавные, могут продолжать текст или оставлять запоминающуюся аналогию.

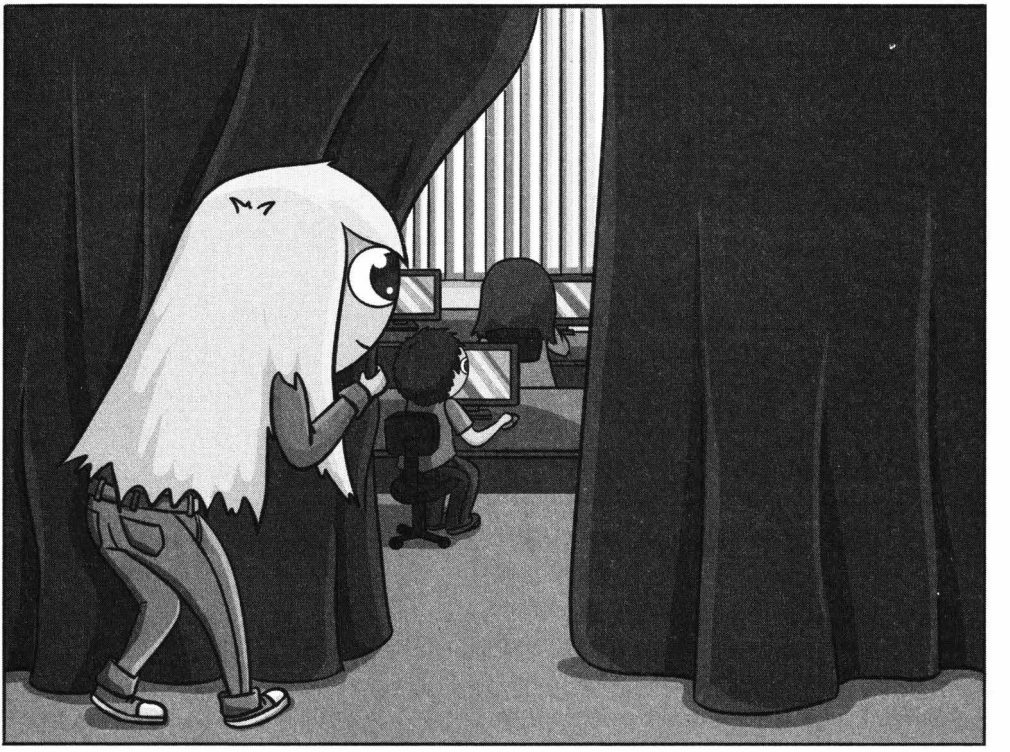
В любом случае они привлекают внимание. Вот что вы сделали, открыв эту страницу? Я думаю, в первую очередь, посмотрели на картинку ;-) Так оно и работает — картинки:

- привлекают внимание;
- помогают передохнуть от обилия текста.

Я сама обожаю читать (и предпочитаю бумажные книги, поэтому если вы держите в руках бумажный вариант, то мое вам почтение. А если еще и цветной, то двойное почтение!). Особенно нежно я люблю серию книг «Head First (O'Reilly)» об информационных технологиях (ИТ). Они написаны как раз в таком стиле — куча картинок, мало текста. Текст написан легко.

Именно их книги в свое время вдохновили меня писать. И я старалась делать так же. Надеюсь, вам понравится. Но чего вы точно не найдете у меня, так это академического языка. Оставим его для скучных лекций. А нас с вами ждет увлекательное путешествие в мир ИТ!

ВВЕДЕНИЕ В ТЕСТИРОВАНИЕ ПО



В этой главе мы приоткроем завесу мира ИТ:

- что такое ПО?
- кто такие тестировщики?
- чем они занимаются?

Что такое ПО?

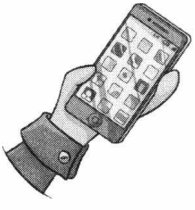
Сначала усвоим, что ПО — это программное обеспечение. Потому что в этой книге мы будем говорить именно про тестирование ПО. Хотя на самом деле тестировать можно всё что угодно:

- Еду — вкусно ли? Не отравлено? Почитайте историю Старбакс[1]* — как они выводили новый сорт кофе. Бариста создавал, тестировал, менял компоненты и так по кругу.
- Одежду — красиво? Стильно? Как в носке? А если вот тут прибавить, там убрать? А если пробегать весь день по городу, не появятся следы?
- Железяки — в конструкторских бюро тоже есть свои тестировщики. Нельзя просто создать подшипник скольжения, нужно проверить его на прочность и другие характеристики.
- ...

Представим ситуацию — вы встречаете на улице знакомого, и он спрашивает вас, сколько сейчас времени.

Куда вы посмотрите? Раньше я бы кинула взгляд на наручные часы, а сейчас потянусь за телефоном. Посмотрите на современный смартфон — сколько там разных программ! Часы, календарь, обменники сообщениями, фотокамера, галерея фотографий, яндекс- карты, музыка, блокнот, всевозможные игрушки...

Это всё — программы. Каждая из них была кем то придумана и создана, разработчик написал код — и вуаля, вот она уже в вашем телефоне!



И даже если вернуться к наручным часам, сейчас ведь и они бывают «умные». Знаете, такие, которые выглядят как черный браслет. И стоит помахать рукой, как на них загораются цифры. А еще они умеют измерять пульс и записывают данные во время пробежки. И это всё тоже программы.

Программы окружают нас повсюду — это всё приложения внутри вашего компьютера, это сканер в магазине, которым кассир пробивает товар, это банкомат, отсчитывающий вам денежку, это навигатор в автомобиле и многое-многое другое.

---

* Все сноски находятся в разделе «Примечания» в конце книги.

Кто создает программы?

Это всегда разное количество людей. Как в продуктовом магазинчике или кафе. Бывает, что есть только владелец: он и бариста, и повар, и продавец, и бухгалтер, и уборщица в одном лице. На найм сотрудников просто нет денег. Или, может, муж с женой исполнили мечту и открыли кафетерий? Тогда у нас уже два человека...

А когда бизнес пошел в гору, то и сотрудников прибавилось. Пригласили продавца, официанта, уборщицу. Открыли второй кафетерий, третий... Количество участников всегда меняется, и это нормально.

Также и в ПО — всё зависит, в первую очередь, от того, сколько у тебя есть денег ;-)

Автор, идеолог, разработчик, тестировщик

Один участник: автор = разработчик

Если у вас есть идея и вы можете сами ее реализовать — флаг в руки! Тогда вы, как тот начальник кофейни, сочетающий в себе всё и вся. Сами придумали идею, сами написали код. Иногда даже код писать не надо, если хотите сделать что-то простое. Например:

- **лендинг-страница** — это сайт-одностраничник, продающий ваш товар (супер-пупер расческу, курс по программированию, цемент для дачи и т. д.). Такую страничку[2] я сделала сама. Этот курс уже закрыт, но сама идея лендинга в Тильде[3] неплохая.
- **интернет-магазин** — сейчас и для них есть стандартные шаблоны. Сделал как нравится, и вперед!

Два участника: автор идеи + разработчик

Более частый сценарий: у меня есть идея, но я не программист. Не могу сама сделать сайт своей мечты. Тогда я обращаюсь к мужу / другу / разработчику по найму. Описываю ему свое ви́дение, а он уже делает систему.

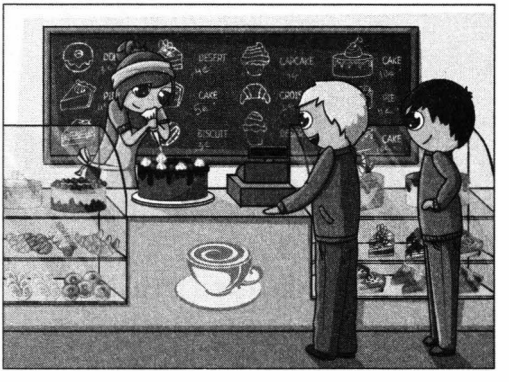
Пример — Testbase[4]. Всего два участника у сайта: я придумала, как должны выглядеть навыки, а мой друг-разработчик его сделал.

Много участников

Когда проект растет, в нем появляется больше людей. Это логично. Можно самому держать кафе, когда к тебе заглядывают по два человека в час. Но потом, когда посетителей становится больше, один человек просто не справляется.

Посмотрим на примерах.

Анечка-кондитер



Анечка — менеджер, но ее настоящее призвание — печь пирожки. Приходя вечером домой, она экспериментирует с кексиками и тортиками, добавляя новые вкусы или образы.

Как-то раз подруга попросила Анечку сделать торт на праздник. Торт так понравился гостям, что ей начали поступать новые заказы.

Сначала она успевала их сделать после основной работы в офисе. Но заказов становилось всё больше, и Анечка решила уйти с работы и заняться любимым делом.

Ее день начинался в 5 утра: нужно было подготовить пекарню, протереть пыль, помыть полы, выкинуть мусор, отмыть туалет, заказать продукты, встретить продукты, подвести баланс в конце дня.

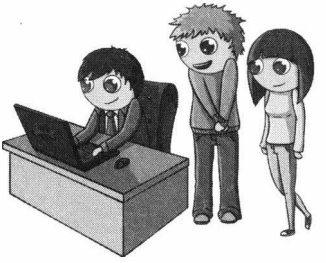
И всё это, не считая собственно выпечки и обслуживания клиентов за кассой, ей приходилось делать самой. Первое время было сложно, но весело. Потом энтузиазм поугас, и Анечка поняла, что проще нанять уборщицу и приходящего бухгалтера.

Теперь у Анечки появилось время на тортики! Она целый день занималась любимым делом. И всё было хорошо, и клиенты довольны. Они сарафанным радио пиарили ее кафе, и посетителей становилось всё больше, а очередь длиннее. Однако клиенты были уже не слишком счастливы. Ждать пироженку полчаса? Нет, спасибо, в соседнем кафе можно купить быстрее, пусть даже менее вкусную.

Чтобы не терять клиентов, Анечка наняла себе в пару другого кондитера. Сложно начать делегировать — ведь «это же мое детище, никто не сможет сделать лучше меня». А потом становится всё легче и легче.

Так что потихоньку количество персонала в кафе увеличивалось. Росли обороты, прибыль, Анечка открыла второе кафе, а потом еще парочку и сделала сеть кафе с вкусной домашней едой. Теперь у нее была сотня людей в подчинении и напарник-управленец, который оставлял ей время печь новые пирожки!

Сравним этот пример с разработкой ПО.



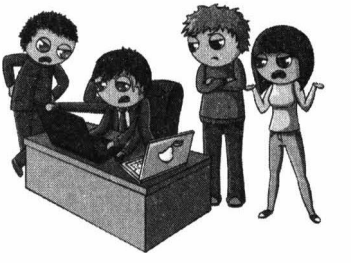
Ваня-программист

Ваня — продавец-консультант, но настоящее его призвание — писать код. Приходя вечером домой, он садится за компьютер и делает своего Марио с блэкджеком и единорогами.

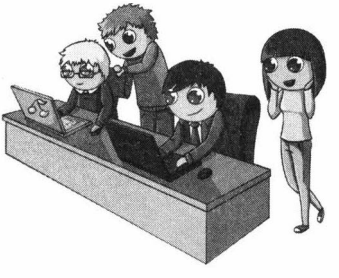
Как-то раз друг попросил Ваню сделать сайт для своего магазина. Его интернет-магазинчик имел такой успех, что Ване поступили новые заказы. Сначала он успевал их сделать после основной работы в офисе. Но заказов становилось всё больше, и Ваня решил уйти с работы и заняться любимым делом.

Его день начинался в 5 утра: нужно было написать требования, согласовать с заказчиком, потом переделать и еще раз согласовать. И еще, и еще, и еще. А потом проверить, что всё работает как надо. И всё это, не считая, собственно, программирования, Ване приходилось делать самому.

Первое время было сложно, но весело. Потом энтузиазм поугас, и Ваня понял, что проще нанять специалиста (менеджера-аналитика), который станет общаться с заказчиками и формулировать техническое задание (ТЗ), и еще одного, который будет за ним проверять код (тестировщика).

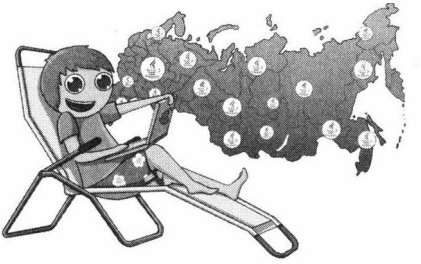


Теперь у Вани есть время на код! Он целыми днями разрабатывает новый функционал и дорабатывает старый. Клиенты довольны, но их становится больше. И вот уже Ваня просто не справляется. Ему приходится говорить клиентам, что он сможет сделать то, что они просят, только через месяц. Конечно, клиенты не будут столько ждать. Они уходят к конкурентам.



Чтобы не терять клиентов, Ваня нанял себе в пару другого программиста. Сложно начать делегировать — ведь «это же мое детище, никто не сможет сделать лучше меня». А потом становится всё легче и легче.

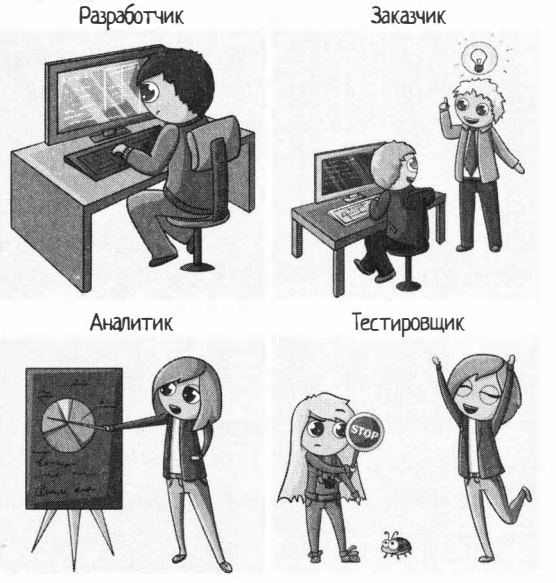
Так что потихоньку количество сотрудников увеличивалось. Росли обороты, прибыль, и вот уже Ваня — второй Билл Гейтс! Хотя нет, строгие костюмы не для него. Тогда Стив Джобс? Филиалы фирмы появились в самых разных городах, а Ваня получил возможность переехать в Таиланд и продолжать писать код, но уже нежась под солнышком и не переживая о том, как оплатить счет за квартиру.



Что делают все эти люди?

Если говорить именно про ИТ-специальности, то выделяют несколько основных. Вначале всё было просто:

- **Заказчик** — человек с деньгами и идеей, которую сам он не может реализовать.
- **Аналитик** — собирает требования. Расспрашивает заказчика о том, что тот хочет, затем обдумывает, как это лучше всего внедрить в продукт, и пишет техническое задание (ТЗ).
- **Разработчик** — пишет код, реализуя, как он думает, идею Заказчика, а на самом деле ТЗ.
- **Тестировщик** — проверяет корректность работы приложения, ищет баги, предоставляет информацию команде.
- **PM** — менеджер проекта (Project Manager). Именно к нему обращается Заказчик, а он уже распределяет задачи между исполнителями. Особенно важен, когда исполнителей много. ;-)



Но сейчас роли часто переплетены. Разработчик вполне может протестировать свой код, тестировщик — исправить баг, а заказчик — написать требования.

Кроме того, каждая команда может иметь свое видение того, что входит в обязанности каждой роли. Почитайте книгу «Как тестируют в Google»[5] — там и роли у них свои, и определения юнит-тестов (мы о них поговорим в разделе про автоматизацию).

И никто не отменял более общие роли. Бухгалтер, служба поддержки, уборщик, гендир и так далее. Они могут быть, а могут и не быть. Зависит от бюджета и размеров компании.

Чем занимается тестировщик?

Наверняка перед чтением этой книги вы уже успели что-то поискать. Бесплатные видеолекции, статьи, книгу Романа Савина[6]. Так как вы думаете, в чем основная задача тестировщика? Чем он занимается?

Типичный ответ — ищет баги... «Крушить, ломать, ПО побеждать». Поймал ошибку, сообщил разработчику, всё, твоя работа на этом закончилась.

Другой популярный ответ — помогает улучшать качество продукта. А как помогает? Ну, ищет баги и сообщает о них.

Да, мы ищем баги и сообщаем о них. Но главная задача тестировщика — предоставить информацию о том, как работает приложение.

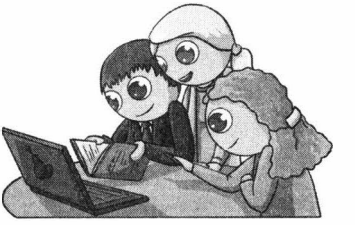
Мы исследуем продукт и рассказываем команде, как он работает. Или не работает ☹ На основе полученной информации команда решает, можно ли выпускать релиз (отдавать новую версию приложения пользователям), или стоит сначала исправить баги. Какие баги важно исправить сейчас, а какие можно будет поправить позже. Что вообще никогда чиниться не будет.

В чем же разница между этими двумя понятиями?

○ Тестировщик ищет баги.

○ Тестировщик предоставляет информацию о продукте.

В том, что мы можем не найти багов, но это не значит, что тестирование плохое или его не было вообще. Мы можем дать отмашку «новых багов нет, старые незначительны, давайте выпускать продукт!» — и это тоже тестирование.



Вот мы, допустим, решили разработать онлайн-банк. И всё — уже через неделю у нас реальные посетители начнут им пользоваться. Но если наша задача — просто поискать баги, то мы будем вводить всякую фигню во все поля, будем просто биться головой о клавиатуру в надежде на то, что хоть что-нибудь где-нибудь сломается.

Мы заведем кучу ошибок вида «при вводе миллиарда символов в поле „имя„ система рушится» или «если открыть 10 копий интернет-банка, нигде не сможешь залогиниться», или «если нажать вперед, потом назад, потом снова вперед, и так 35 раз — система повиснет»...

И что-то из этого списка мы даже исправим, и вроде как не зря тестировали, столько ошибок нашли!

Но потом придет реальный пользователь и, не пытаясь сломать систему, попробует зарегистрироваться... а регистрация сломана. Или он войдет в систему, но не сможет пополнить счет.

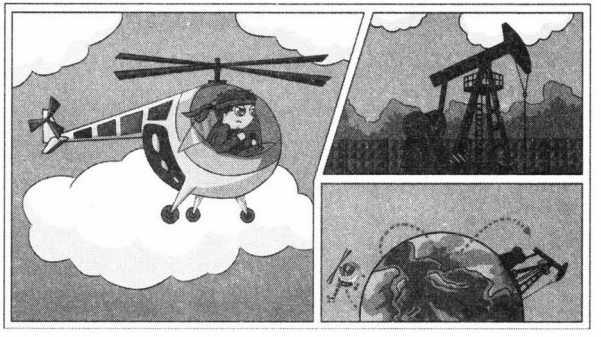
То есть мы проверили всякие хитрые комбинации «а выдаст ли система ошибку, если мы вводим отрицательную сумму, если мы вводим буквы вместо цифр и всё такое».

А про основные позитивные сценарии забыли — это же не столь интересно... Не делайте так! ВСЕГДА, всегда тестировщик начинает с позитивного тестирования. Сломать еще успеете, проверить работу — нет.

Задача тестировщика — рассказать, что сейчас в продукте работает, а что нет. И поэтому сначала мы проводим более приоритетные проверки, то есть изучаем наш продукт, выясняем, какой в нем функционал. И, в первую очередь, проверяем то, что будут делать пользователи, — не пытаясь сломать систему и найти как можно больше багов. А уже потом, когда остается время, мы проверяем всякие хитрые сценарии.

По предоставленной нами информации наш менеджер уже может сделать вывод, готов продукт к выпуску или стоит его еще немного доработать. Поэтому всегда стремитесь предоставить грамотную информацию, а не просто найти как можно больше багов.

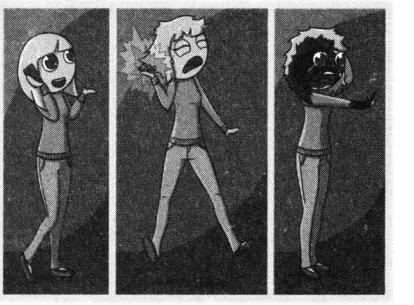
Почему тестирование так важно?



В конце концов, можно же просто выпустить продукт, а потом исправить баги, на которые наткнутся пользователи. Да?

А теперь представьте, что вы делаете приложение для нефтяной вышки — как выкачивать нефть, распределять по бакам и т. д. И если обнаружится неисправность, придется брать вертолет и лететь на место происшествия искать и чинить ошибку. Никаких тебе «прислал фикс[7] по e-mail» — там даже Wi-Fi нет.

Это, между прочим, реальная ситуация. Просто вряд ли вы будете тестировать именно такой софт[8], и поэтому ситуация кажется надуманной. Хорошо, помните, на каком примере мы обсуждали, что такое программы? На примере смартфона и всего того хлама, что в нем установлен! А теперь попробуйте погуглить историю «Samsung Galaxy Note 7», если вы о ней раньше не слышали. В поисковую строку можно добавить слово `самолет`.



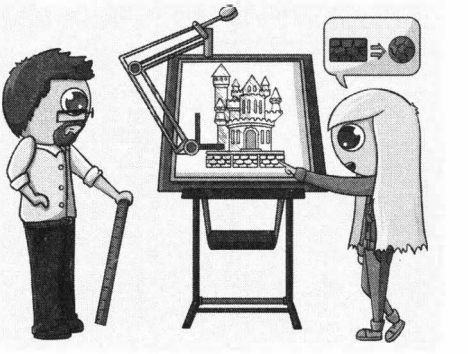
История Samsung Galaxy Note 7[9]

Компания Samsung выпустила очередной «новый клевый телефон» — Galaxy Note 7, который на тот момент был самым крутым в цепочке. Пользователи радостно побежали его покупать, а потом... Телефоны начали взрываться! Всё происходит по однотипному сценарию: телефон перестает работать, не включается и — взрыв.

В итоге Samsung пришлось отзывать все телефоны назад. Это убытки на возврате денег и транспортировке (они отправляли пользователям огнеупорную коробку) + потери на репутации и цене акций.

По официальным данным, их подвел производитель аккумулятора: на тестирование прислал один, а в общей поставке оказался другой. В любом случае это был провал.

На этапе написания требований внесение изменений — дело 1 минуты, поправили текст, и всё



А представьте, если в машине обнаружится массовый брак? Телефон-то хоть стоит тысяч 20–30, ну даже если 100. А машина? Представляете, какие убытки несет производитель, если надо отозвать модель?

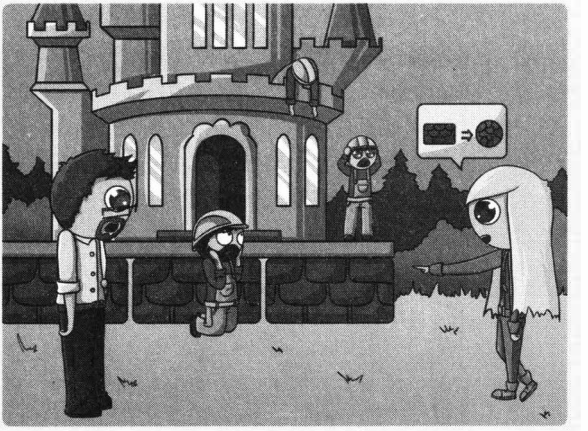
А ведь сейчас вовсю разрабатывают машины, которые катаются без водителей. Что если в их систему закрадется баг? Одно дело — когда ты сам виноват в аварии, совсем другое, когда это сделал робот.

Ну хорошо, хорошо, вернемся к тем реалиям, в которые вы попадете. В ближайшие лет пять, по крайней мере. Скорее всего, это будет разработка ПО, сайта в Интернете, игрушки для мобильного телефона, десктопного[10] приложения типа ворда...

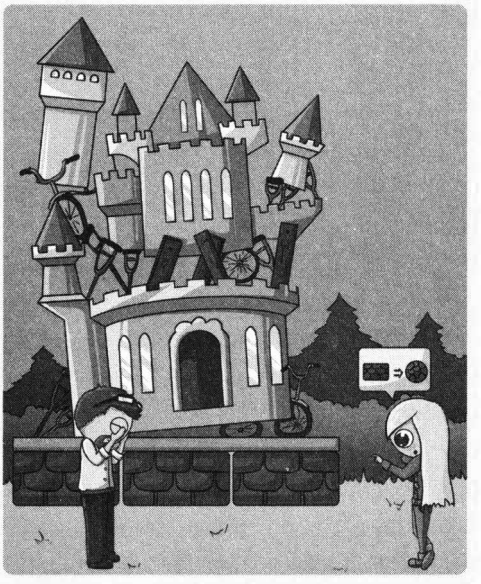
Если мы заметили ошибку на этапе написания требований, то исправить ее — дело одной минуты, просто скорректировали предложение в тексте, и всё.

На этапе продумывания архитектуры будущего кода стоимость ошибки уже дороже. Представьте, что архитектор уже придумал, как всё будет выглядеть, а вы хотите изменить фундамент...

Когда код готов, внести исправления бывает очень сложно. Вдруг вы хотите изменить фундамент?



Когда код давно готов и выпущен, сверху понастроили кучу всего, внести исправления почти невозможно. Так и живем с критичным багом...



Но это еще реально. А вот если мы уже все построили приложение (написали код), то некоторые изменения просто нельзя внести, и приходится мириться с багом. А даже если можно, то стоить это будет сильно дороже:

- аналитику — поправить ТЗ;
- архитектору — придумать, как поправить минимальными усилиями;
- разработчикам — внести правки.

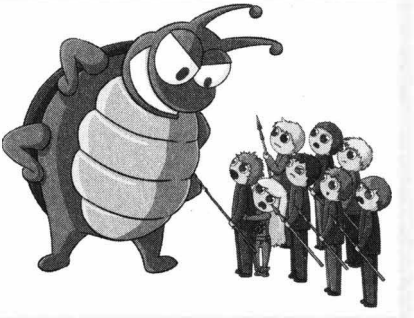
Да, найденная на этапе тестирования проблема обходится дорого. Приходится исправлять много кода, но всё же этот код написан в последний месяц (или сколько времени у нас занимает выпуск одной версии продукта).

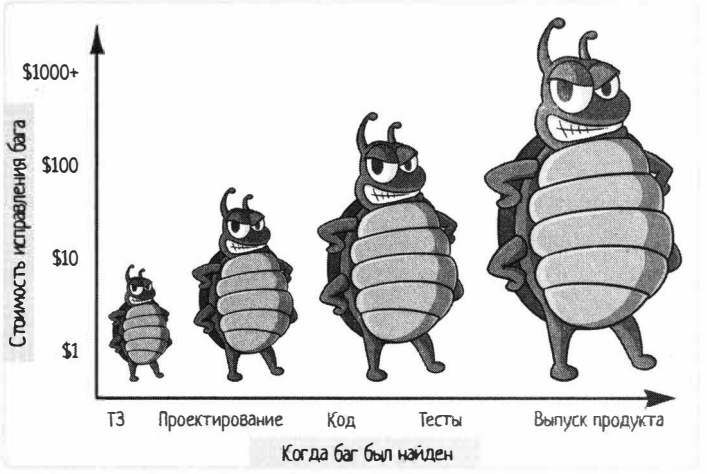
А вот если мы выпустили версию и уже сделали другую, третью пятую, десятую... А потом нашли баг в самой первой — той, на коде которой уже столько всего понастроено, в том числе и костылей... В такой ситуации исправить вашу находку будет особенно сложно.

Тем не менее такой подход до сих пор используют в больших организациях на сотни человек. Там с требованиями работает отдел аналитики, потом их передают дальше и так по каскаду всё приходит в тестирование. С проблемами ошибки в требованиях приходиться мириться...

Баг на стадии требований

Баг в готовом продукте





Так что запоминаем: чем раньше найдена ошибка, тем проще ее исправить!

Можно представить это в виде графика, как сделал еще Ли Копленд (Lee Copeland) в своей книге «A Practitioner's Guide to Software Test Design»[11].

Поэтому тестировщики так важны. Чем раньше они заметят проблемы, тем проще будет их исправить!

Да, конечно, можно обойтись вообще без тестировщиков. Сам придумал, сам сделал, сам проверил. Но ведь намного проще заметить соринку в чужом глазу, чем бревно в своем, не так ли?

Ясно, что маляр покрасит стену лучше, чем разнорабочий. Тяжело быть суперкрутым сразу во всем — ведь мастерство требует времени. Поэтому мы будем учиться быть хорошими тестировщиками. Которые придут на проект и сэкономят кучу времени остальной команде, пытавшейся до этого тестировать свой продукт самостоятельно, не особо понимая, как это делается.

Вопросы для самопроверки

1. Что такое программа?
2. Сколько человек нужно, чтобы ~~вкрутить лампочку~~ создать программу? Кто? Есть ли минимум? А максимум?
3. Зачем нужны тестировщики?
4. «Основная миссия тестировщика — найти баги!» Насколько верно это утверждение?
5. На какой стадии разработки нужно подключать тестировщиков?

Ответы на вопросы для самопроверки

1. **Что такое программа?**
   Программа — это любое приложение, которое есть у вас в телефоне, компьютере, телевизоре, браузере...
2. **Сколько человек нужно, чтобы ~~вкрутить лампочку~~ создать программу? Кто? Есть ли минимум? А максимум?**
   Зависит от программы и умений ее создателя. Он может быть вообще один. Сам придумал, сам сделал и проверил. Может быть много людей, максимума нет. А минимум — один.
3. **Зачем нужны тестировщики?**
   Чтобы предоставлять информацию о том, как работает программа. И сколько в ней на текущий момент ошибок (тех, что удалось найти)
4. **«Основная миссия тестировщика — найти баги!». Насколько верно это утверждение?**
   Нет, неверно. Искать баги — одна из задач тестировщика. Но если он багов не нашел, это не значит, что он не работал ;-)
5. **На какой стадии разработки нужно подключать тестировщиков?**
   Чем раньше, тем лучше, потому что исправить ошибку дешевле всего на ранней стадии разработки

Часть I

ОСНОВЫ ОСНОВ: О ТОМ, ЧТО ЕЩЕ ОБЯЗАТЕЛЬНО ДОЛЖЕН ЗНАТЬ ЛЮБОЙ ТЕСТИРОВЩИК

Глава 1
ИССЛЕДОВАНИЕ ПРОДУКТА

Переходим к делу! Мы устроились на работу, и нам дали задачу протестировать уже готовый продукт. Что делать? В этой главе научимся:

- исследовать продукт — что он умеет;
- правильно задавать вопросы;
- использовать инструменты для записи результатов.

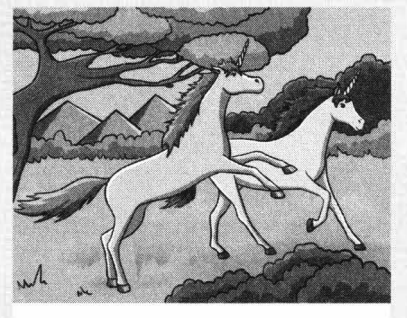
Что продукт умеет?

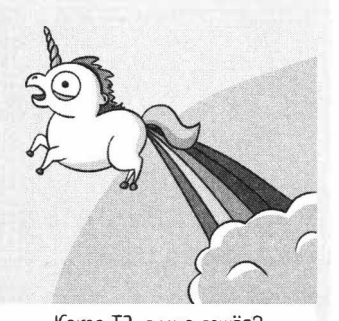
Чтобы протестировать продукт, надо сначала с ним познакомиться. Что он умеет? Для чего был создан? Какие основные задачи решает?

Помните, задача тестировщика — это не просто пойти и все сломать[12]. Лучше провести три теста, но полезных, чем десять бесполезных. Кому какая разница, как именно округляется пятнадцатая девятка в пятом подменю десятого пункта меню, если в системе не работает регистрация? Сперва проверяем важное. Но вначале нужно понять, что именно важно.

А как узнать, что важно? Конечно, мы уже прочитали Савина, увидели диаграмму стоимости исправления дефектов, а потому ожидаем, что на работе нам сразу выдадут подробное и понятное техническое задание (ТЗ). И мы — такие классные — просто проверим систему на соответствие ТЗ, и всё!

На самом деле нет. ТЗ на проектах бывает не всегда. А даже если оно есть, там не будет всех- всех- всех подробностей. А чем больше подробностей, тем больше шансов, что оно уже устарело, и всё работает совсем не так.


Вот держи ТЗ, тестируй!
Повникай получше, времени неделя.


Какое ТЗ, с ума сошёл?
Быстренько проверь, что всё работает, через час выпускаемся!

Итак, вы устроились на работу. Теперь вы тестируете какой-то продукт. Ваша первая задача — изучить его. Если документация есть, читаем ее. Если нет, то просто исследуем и задаем вопросы аналитику про непонятные моменты.

Зачем вообще нужны программы?

Программы нужны, чтобы:

- помочь в достижении цели;
- помочь решить проблему;
- развлечься.

Иначе они никому не сдались ¯_(ツ)_/¯.

Я не стану открывать Word, если хочу посмотреть фоточки, — я открою средство просмотра фотографий. Потому что Word не решает моей задачи.

Я не буду открывать сайт налоговой, если хочу отдохнуть и развлечься.

А если я выбираю программу для покупки, то смотрю, решает ли она мою проблему. Потому что сейчас так много программ, которые выполняют одно и то же. Я выберу ту, что решает мою проблему. А если несколько программ ее решают — то самую симпатичную ;-)

Когда мы выясним, для чего нужна программа, то придет понимание, что именно нам тестировать в первую очередь. Посмотрим на примерах.

Бухгалтерский софт

Даже если вы никогда не работали с ним, то наверняка слышали о программах типа 1С. А ведь раньше этих программ не было. Бухгалтеры вели какие-то особые тетрадки, куда вносили суммы. И не дай бог

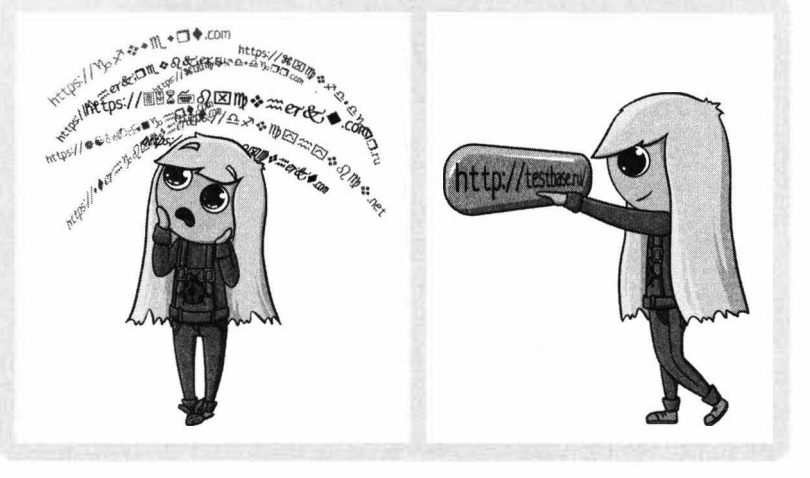
опечататься — исправлять эту книгу нельзя! Нужно начинать все сначала!! Вот где реальная проблема: и так волнуешься перед сдачей отчетности, а еще ошибиться нельзя.

А сейчас есть удобная программа. Хотя я видела скриншоты из 1С и меня они пугают, но кому какая разница, я же не бухгалтер. А бухгалтеров программа избавляет от целой кучи проблем, так что им там все нравится.

Если мы тестируем такую программу, то узнаем, как чаще всего ее применяют бухгалтеры, и проверяем, что это все работает. При этом полезно также узнать, как именно это работало раньше. Потому что если мы, например, изменили «горячие клавиши», то нам-то удобнее стало, а вот тетушкам очень плохо, они привыкли уже к старым комбинациям!

Testbase

Для чего нужен Testbase[13]? Фактически, это набор закладок по определенной тематике (тестированию). Просто закладки красиво расположены



и сгруппированы, и, вообще, теперь можно в закладках держать только Testbase, а не кучу вложенных ссылок.

Исходно этот сайт я делала для себя. Я веду курсы для начинающих тестировщиков, и у нас есть отдельный чатик для выпускников. Там периодически кто-то писал: «Ой, а помните ту статью про классы эквивалентности?». Или: «А что мне почитать про SQL или автоматизацию?».

И вот ты такой идешь, роешься в своих закладках, находишь ссылку и кидаешь ее в чат. Или в свой блог идешь, зная, что речь о твоей статье, и пытаешься вспомнить тег или название... Это самое обидное, кстати, когда речь о твоей статье и ты ее помнишь, но не можешь найти.

Так что я стала ссылочки сохранять. Но делать кучу закладок не хочется — в итоге будешь все равно долго искать, но уже среди них. Вот и пришла идея создать портал, где ссылочки полезные будут сгруппированы, чтобы можно было легко и просто найти нужную.

А еще такой портал могу использовать не только я, что приносит сразу две плюшки-возможности:

- поделиться знаниями;
- пропиариться.

Создал мне этот сайт мой знакомый разработчик. Он это сделал один раз. И теперь с точки зрения разработки нужно только бэкапы[14] периодически снимать. Я захожу в админку, и там есть разделы: добавить новый навык или новую ссылку. Если ссылку, то какого типа и к какому навыку привязать. Всё — один раз настроили и больше не трогаем.

Как это знание влияет на тестирование? Вот вы решили протестировать Testbase, а что нужно проверять в первую очередь? Так, вспоминаем, зачем сайт нужен.

И что же мы будем тестировать? Фактически... текст! Все ли ссылки на месте, нет ли там опечаток... Да, можно зайти в админку и начать пытаться: «А если я сделаю так, а сяк, а наперекосяк, а введу сюда слишком много и попробую вставить NULL»... Но зачем?

Админка — для внутреннего использования, то есть только для меня. Я не собираюсь ее ломать. Поэтому зачем мне такие тесты? Ну, сломается там что-то, если вставить куда-нибудь инъекцию или слишком длинный текст. И что? Мне нужно платить деньги за то, чтобы исправить «очень страшный баг, который никогда не случится»? Но зачем?

Не нужно проводить лишние тесты, лишние проверки. IT[15] — это не то место, где платят за просиживание штанов 8 часов на стуле, и пофиг, что ты делаешь бесполезную работу. Ну не везде, конечно, но я надеюсь, вы не попадете в такие компании.

Как накидать тестов на что-нибудь?

Допустим, вы пришли на собеседование, и я предложила вам протестировать... ключ. Требования? Нет требований, просто скоро вам дадут ключ, а пока нужно подготовить тесты. Что будете делать?

Еще даже не зная всех мудреных техник типа классов эквивалентности, граничных значений, таблиц решений и прочая, прочая, вы уже можете выполнить это задание. Потому что знаете, с чего начинается тестирование:

1. Выясняем суть — что за ключ, зачем он нужен.
2. Проводим тесты — проверяем в первую очередь, что объект тестирования делает то, что от него ждут. А всякие извращения и попытки сломать оставляем «на потом».

Запомнили? Правда-правда? Вы уверены? ;-)

В своей школе для начинающих тестировщиков я на первой лекции говорю всё то же самое:

— Выясняйте суть, иначе будете делать ненужную работу.

— Да-да, мамми, мы все поняли, так и будем поступать!

— Ну вот вам домашнее задание (ДЗ): тестируйте ключ.

Угадайте теперь, какие домашки я получаю? Правильно. Половина студентов присылает примерно такое ДЗ:

> Осмотрим ключ, есть ли на нем царапины? Какой внешний вид? Насколько походит к замку? Вставляем ключ в замок, откроется ли?

Некоторые добавляют:

> Возьмем простой ключ от квартиры. Что нам надо проверить? ...

Или так:

> А что за ключ? Для чего он будет использоваться?

И даже так:

1. Визуальный осмотр ключа.
2. Подходит ли к квартире?
3. Не отопрет ли и соседнюю дверь тоже?
4. ...

Хм, то есть вроде как в теории знаем, что задавать вопросы надо, но ответы нас на самом деле не особенно волнуют, проверки-то вот они! Уже готовы!

А что, если на самом деле это ключ-карта? Или ключ для шифрования

данных? Как вы будете осматривать пикселы в программе? А если это 3D-модель в метр высотой — какую квартиру отпирать будете?

Нет смысла заранее составлять набор проверок, не уточнив детали. Потому что иначе окажется, что тесты надо проводить в другом порядке, а то и вовсе выкинуть. Не додумывайте, уточняйте!

Какие вопросы задавать?

Самые главные вопросы:

1. Зачем эта программа нужна?
2. Какие проблемы она решает?
3. Что вы ожидаете от тестирования?

Если есть ТЗ — отлично, ответ на первые два вопроса можно узнать из него. Если ТЗ нет, что случается, увы, слишком часто, то уточняем детали.

Будьте готовы к тому, что на последний вопрос вы услышите раздражение: «Откуда я знаю? Чтобы багов не было». Это нормально. Заказчик может и не знать ответ. Ему сказали, что тесты нужны — ну, значит, тестируй. Но возможно, что у него уже есть какие-то ожидания. И лучше заранее их узнать.

Особенно полезно задавать такой вопрос не заказчику, а команде. Что они ожидают от вас и вашей работы? Возможно, разработчик хочет, чтобы вы «поскорее проверили вот эту и во-о-о-о-о-т ту задачу», а менеджер ожидает, что вы проведете полное тестирование системы. Всегда уточняйте, что от вас хотят и корректируйте эти ожидания.

Вопросы должны быть «открытыми». Это практически всегда лучше, чем «закрытый» вопрос.

Открытые и закрытые вопросы

Закрытые вопросы

Возможен только однозначный ответ:

- дата;
- время;
- название;
- количество;
- ДА или НЕТ?

Например:

- Сколько лет вы работаете в тестировании?
- Вы проводили нагрузочное тестирование?
- Система должна обрабатывать TXT-файлы?

Открытые вопросы

Нужен развернутый ответ, простым «да/нет» не отмажешься. Открытые вопросы обычно начинаются со слов:

- Что?
- Как?
- Почему?
- Каким образом?
- При каких условиях?

Например:

- Как именно вы проводили нагрузочное тестирование?
- Какие форматы файлов должна обрабатывать система?

Ответ на открытый вопрос всегда даст вам намного больше информации, чем ответ на закрытый. Возьмем пример с собеседованием и посмотрим, на каком вопросе спрашивающий понимает реальное положение дел:

— Вы проводили нагрузочное тестирование?

— Да!

— Как часто?

— Да года три уже. (*Ух ты, крутой какой!*)

— Расскажите, пожалуйста, как именно вы проводите эти тесты?

— Ну, на самом деле тесты все были написаны до меня. Я просто прихожу раз в неделю, нажимаю кнопочку, а потом смотрю на отчет, что ничего не просело. И все. (*Ну, не особо и крутой...*)

Фактически закрытый вопрос означает, что вы сами уже додумали себе требования и их же и будете тестировать. А тестировщик НИКОГДА не додумывает. Ведь иначе он будет тестировать вообще не то, что нужно.

Например, система принимает на входе файл с текстом и проверяет его на опечатки. Тестировщик додумал себе, что это простой блокнот, и уточняет у заказчика:

— Система должна обрабатывать TXT-файлы?

— Да, должна.

— Ok, спасибо.

Полностью уверенный в том, что узнал всё, что ему было нужно, тестировщик готовит свои тесты. А на самом деле заказчик ждет, что система будет уметь работать со всеми текстовыми форматами, включая Excel, а еще она сможет считывать текст с картинок! И именно в этом ее фишка. То есть главный функционал. А тестировщик картинки с текстом вообще не проверяет, так как считает это негативным тестированием.

Поэтому всегда старайтесь задавать максимально открытый вопрос. Если видите, что ваш вопрос «закрытый», подумайте, что именно вы хотели узнать, — об этом и спросите.

Примеры

Закрытый вопрос: Сколько лет вы работаете в тестировании?

Хотим узнать, какой у него опыт.

↓

Открытый вопрос: Сколько лет вы работаете? Чем именно занимались?

Может быть, опыта меньше полугода, но человек уже кучу всего умеет делать, потому что учится. Или, наоборот, стаж 10 лет, но просиживал штаны все это время.

* * *

Закрытый вопрос: Вы проводили нагрузочное тестирование?

Хотим узнать, умеет ли он это делать.

↓

Открытый вопрос: У вас написано про опыт нагрузочного тестирования. Расскажите, пожалуйста, что именно вы делали, как его проводили?

А на самом деле всё уже автоматизировали до него, и нагрузочное тестирование в его случае — нажать на кнопочку и проверить, что ничего не упало. Если упало, призвать разработчиков.

* * *

Закрытый вопрос: Система должна обрабатывать TXT-файлы?

Хотим узнать, какие форматы принимает система.

↓

Открытый вопрос: Какой формат должна обрабатывать система?

А на самом деле, кроме TXT, система еще должна уметь распознавать текст с картинок. Но вы же спрашивали только про TXT, про него ответ и получили.

* * *

Закрытый вопрос: Можно ли завести машину с брелка?

Хотим узнать, есть ли у нее автозапуск?

↓

Открытый вопрос: Есть ли у машины автозапуск? Какой?

На самом же деле можно запустить машину, позвонив ей по телефону или отправив СМС. Но вы додумали требование про брелок и только про него и спросили. А о возможности послать СМС узнали уже после окончания тестирования.

* * *

А закрытыми вопросами можно «дожать» информацию. Их используют, когда вам уже развернуто объяснили, для чего нужна система, а вы вдруг подумали о нетривиальном случае. Например:

* * *

— С какими форматами работает система?

— С любыми текстовыми: DOC, TXT, CSV.

— А таблицы понимает? Эксел, например? Там же тоже текст написать можно...

— Хм, пожалуй, да.

* * *

Уточнять информацию можно и закрытым вопросом. Но если вы начнете с него, то наверняка додумаете остальное. Помните, тестировщик не додумывает — он уточняет.

Какие вопросы НЕ задавать?

Выясняя у заказчика информацию о продукте, не надо спрашивать:

— Надо ли мне тестировать А, Б, В?

— А какие в России существуют города-миллионники? (*Или другой вопрос, который уместнее задать гуглу.*)

Это же вы тестировщик, вы и должны знать, что и как нужно тестировать. Если заказчик отвечает на эти вопросы, то зачем ему вы?

По своим студентам я вижу, что, задавая первый вопрос, они порой хотят узнать что-то о продукте. Просто кривовато формулируют мысль:

— Надо ли мне тестировать разные форматы файлов?

— Надо ли тестировать файлы эксел?

А на самом деле они тут хотят знать не «надо ли мне тестировать», а «как работает система с...». Ну так и спросите! Посмотрите на эти вопросы со стороны заказчика. Специалист, которого наняли тестировать, идет к бизнес-заказчику и спрашивает: «А что мне тестировать?». Заказчик, в свою очередь, пойдет к начальству тестировщика и спросит, на фига он такой нужен.

Про второй тип вопросов, надеюсь, не надо пояснять. Если вы что-то можете спросить у гугла, спросите у гугла. Но и тут есть вариант, когда вы просто криво сформулировали свою мысль:

— Какие в России существуют города-миллионники? (*Это вопрос гуглу.*)

— Система работает по всей России или только в конкретных городах? (*А это уже про систему.*)

Сколько вопросов задавать?

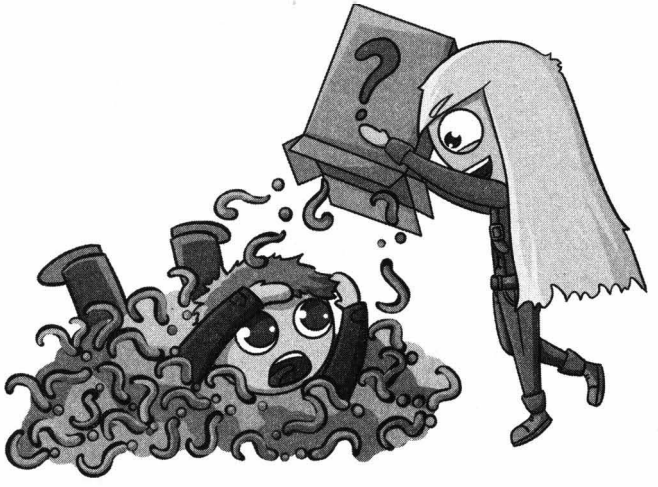
Как можно меньше, но чтобы все узнать. Важно не переборщить.

Например, решила я создать интернет-магазинчик кухонных мелочей. Пока есть только товар и общее представление, что «должно быть как Озон, а там придумаем, как сделать лучше». И вот я даю такую задачу разработчику, он молча кивает и уходит делать. Зато тестировщик заваливает вопросами:

— А сколько разделов будет на сайте?
— А что если их будет больше 100?
— А что если товаров в разделе нет?
— А что если я захочу купить один предмет?
— А если 10?
— А если 50?
— А если мне нужно два одинаковых?
— А что если я закажу доставку за МКАД?
— А внутри МКАД?
— А какого цвета будет кнопочка заказа?
— А в каком порядке пойдут разделы?
— А ...

Когда я вижу огромный пласт вопросов на «простенькую задачу», то прихожу в ужас: «И что, мне на все на это надо отвечать???». Потом начинаю вчитываться и раздражаюсь — ведь если это все вопросы ко мне, то зачем мне вообще работать с такой командой, которая шагу ступить без одобрения не может?

Не пытайтесь заменить аналитика. Это его задача — продумать требования вплоть до мелочей и уточнить все детали. И аналитиков отдельно учат



задавать вопросы. Так, чтобы не додумывали, но и чтобы не заваливали заказчика вопросами.

Поэтому все мелкие детали «а что если я проведу негативный тест?» спрашиваем у аналитика. При этом их допускается объединить. Сравните, при условии «заказать можно от 100 до 300 упаковок товара оптом» вы можете задать столько вопросов:

— А что будет, если заказать 5 товаров?

— А если 0?

— А если –1?

— А если 500?

— А если 100500?

— А если «сто штук» прописью?

А можете вместо них один такой:

— Что будет, если заказать не в диапазоне от 100 до 300?

Так мы узнаем, как себя система ведет в негативном сценарии — выдает пустой результат, ошибку или что-то иное. И узнаем мы это за ОДИН вопрос, а не 100500.

Или если заказчик говорит, что система должна обрабатывать только форматы эксела и CSV, сравниваем такой набор вопросов:

— А что если загрузить формат DOC?

— А что если загрузить формат DOCX?

— А что если загрузить формат TXT?

— А что если загрузить формат XLXSM?

— А что если загрузить формат JPG?

— А что если загрузить формат PNG?

— А что если загрузить формат RTF?

— А что если загрузить формат PDF?

— А что если загрузить файл без расширения?

И такой:

— Распознается любой формат CSV или только файл с расширением csv? (Если заказчик не понял вопроса, объяснить разницу между форматом и расширением.)

— Другой текст: Word, RTF, PDF — не работает?

— Что будет, если загрузить неподходящий формат?

Мысль всё та же — думаем о том, что именно мы хотим узнать с помощью этих вопросов. Об этом и спрашиваем. Обычно получается несколько вопросов совместить в один. Плюс выкидываем вопросы, которые не надо задавать:

- «Что мне надо протестировать?»;
- вопросы к гуглу.

Как ввести собеседника в контекст вопроса?

Всегда, когда задаете вопрос, вводите собеседника в контекст. Даже если вы общались 10 минут назад — он уже успел переключиться на другую задачу и забыл, о чем шла речь.

Просто, если вы общались недавно, введение будет кратким:

— Так, насчет логов по задаче TEST-8...

Всё, собеседник уже понял, что речь пойдет про задачу TEST-8 и логи в ней. А так как вы недавно ровно это и обсуждали, то он быстро понял, о чем речь, и смог поддержать разговор.

А вот если вы вбежали в комнату и продолжили свою мысль:

— Что мне делать с NPE?

то разработчик посмотрит на вас вот так: o_O

Это вы в контексте, вы сидели и последние полчаса копались в логах задачи TEST-8, а он уже переключился на глобальный рефакторинг системы и забыл всё, что обсуждалось ранее. Напомните ему, о чем идет речь.

Если вы общаетесь в каком-нибудь мессенджере типа скайпа, телеграма или вацапа, всё еще проще — киньте ссылку на задачу или статью, о которой хотите задать вопрос.

Незнакомы? Представьтесь!

Если вы обращаетесь к незнакомому человеку — не забудьте представиться. Допустим, вы работаете в компании-интеграторе и вам дали контакт человека, у которого можно спросить по поводу ТЗ. И вот вы добавляетесь к нему в скайп и сразу с места в карьер:

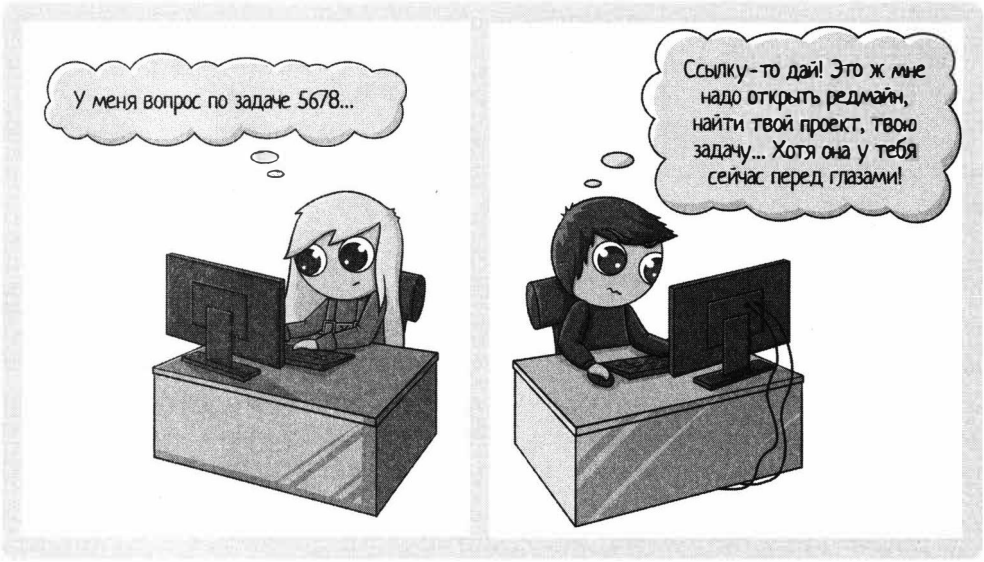
— А как бы мне закрыть отчетность? А то задача ошибку выдает.

Человек ещё даже не знает, кто вы такой, а тут такие вопросы... У него же NDA подписан — как он должен рассказывать незнакомому человеку о работе системы? Так что не забудьте представиться.

Дайте ссылку

- Обсуждаете задачу из джиры[16]? Кидайте ссылку!
- Вопрос по ТЗ из конфлюенса[17]? Дайте ссылку, у вас-то оно уже открыто!
- Не получается написать автотест в постмане[18]? Скиньте разработчику «проблемный» участок кода и скриншот всего теста + результат его прогона.

Давая ссылку, вы быстрее получаете ответ. Вот работает разработчик над своей задачей и получает вопрос. Что быстрее — тыкнуть на ссылку или открыть джиру и поискать, о чем речь... Первое можно сделать сразу, а второе «как-нибудь потом».



Это работает не только на работе, но и на тренингах. Вопрос по заданию? Дайте на него ссылку в систему дистанционного обучения (СДО). Или хотя бы название задания укажите, потому что по номерам тренер их может не помнить. Вопрос по лекции? Дайте примерный тайминг.

В тренингах, связанных с написанием кода, сразу вместе с вопросом показывайте свой код. Укладывайте код в репозиторий, давайте ссылку на него и говорите, какой тест нужно запустить, чтобы воспроизвести описанную проблему. Это ускорит ответ.

Зачем вам это?

Поясните, зачем вам нужна информация. Возможно, собеседник подскажет более оптимальный путь решения вашей проблемы.

Например, в одной из систем, с которыми я работала, есть несколько разных методов поиска: по всем полям, по конкретным, только по телефону и т. д. Один возвращает все поля, другой — только нужные, но работает быстрее. Каждый подходит для своей цели.

Эти методы могут вызывать другие системы. Так, на сайте оператора связи вы вводите номер телефона, а он отправляет к нам поисковый запрос. Но, чтобы это настроить, разработчик должен сделать вызов метода. Причем вызов правильного метода.

Разработчик нашел документацию, увидел ключевое слово «поиск» и пошел изучать поиск по всем полям. Он может прийти ко мне и задать вопросы по этой документации, и я даже отвечу ему, как оно работает, но в итоге он только расстроится:

— Уууууу, плохой метод, ищут по всем полям, а мне надо только по телефону!

— Погодите, такой тоже есть, вот он (ссылка на другой метод).

— А, вот это то, что нужно!

Вроде все хорошо, только этап вникания в другую документацию можно было бы опустить, если бы он сразу рассказал итоговую цель ;-)

А что ты сделал сам?

Если задаете вопрос разработчику, не забудьте упомянуть, что вы уже успели сделать сами. Это покажет коллеге, что вы цените его время и не отвлекаете на каждый чих. Иногда так и тянет задать вопрос товарищу, даже если то же самое можно найти в гугле или даже в документации. Но ведь так быстрее! Да, для вас. А коллега потеряет мысль и потом 10 минут будет вспоминать, о чем только что думал.

Например, вам сказали протестировать SOAP-метод. Можно тут же радостно прискакать к коллеге:

— Ой, а что это такое? А как послать запрос? А где скачать программу?

Но можно погуглить немного самому и прийти уже с конкретным вопросом:

— Мне по задаче TEST-11 надо SOAP-запрос отправить, я скачал SOAPUi, вставил туда пример из документации, а он падает с такой-то ошибкой. Что не так?

То есть базовую информацию (где скачать SOAPUi) человек нашел сам. Умение гуглить вообще полезно для тестирования — учитесь его применять хотя бы на мелочах.

Итого по контексту

- Если вы обращаетесь к незнакомому человеку — представьтесь.
- Если общаетесь с коллегами — вводите в контекст:
  - О чем речь?
  - Что вы от него хотите и зачем?
  - Что уже попробовали сами найти/сделать?

Описываем кратенько — не нужно делать поэму. Работает как для вопросов, так и для утверждений.

Неправильный вопрос	Что в нем плохо?	Как надо
Так что там с логгером?	С каким логгером?? Я уже давно другое делаю.	Мы с тобой логгер для МЕГА-БАНКА обсуждали полчаса назад. Я посмотрела — да, это то, что нужно. Когда сможешь сделать?
Когда сделаешь задачу TEST-3?	Это я, иди, открывай баг-трекер, находи задачу... Ты что, не мог ссылку дать?	Когда сделаешь задачу `TEST-3`? (дать ссылку на задачу)
Добрый день! Как мне посмотреть, сколько дублей объединилось за сегодня?	Ты кто такой вообще?? И что тебе надо?	Добрый день! Я Иванова Ольга из «Ольга и компания». Читаю документацию `<ссылка>`, но не совсем понимаю, как мне посмотреть, сколько дублей объединилось за сегодня или вчера?

Как часто задавать вопросы?

Пачкой, а не дергая человека каждые 5 минут. Помните — у него есть и свои задачи тоже.

Хотя, конечно, сама грешна. Вот сидишь ты себе, тестируешь, тестируешь. И вдруг БАГ! И ты весь такой на эмоциях бежишь к разработчику:

— Вася, Вася! У меня задача сломалась!!!

Вася отвлекается на вас, сочувствует, может быть, даже помогает. Всё хорошо, все довольны.

Но в итоге ты привыкаешь отвлекать Васю по поводу и без. Ведь быстрее спросить у коллеги, чем поискать информацию самому. Разработчик умный, он всё знает, а самому — это ж гуглить надо! И понеслось, каждые 5 минут:

— Ой, у меня отчет сломался! Посмотри?

— IDEA не хочет тесты гонять ((((Что не так??

— А как часто система пишет в лог?

— Сохраняем ли мы информацию об изменении данных?

— А что будет, если я пошлю пустой отчет?

— А что такое SOAPUi?

— А как мне выйти из vim?

— ...

Бедные коллеги!

И ведь так страдают не только разработчики, но и аналитики, и просто более опытные товарищи. Особенно легко поддаться искушению «как подумал, сразу спросил», если вы сидите рядом. Зачем гуглить, если можно просто спросить? Но помните про то, что коллеге нужно время, чтобы вернуться в состояние потока. Вы потратите минуту времени на самостоятельный поиск, а он даст ответ за 15 секунд и еще минут 10 будет вспоминать, на чем остановился.

Поэтому лучше задавать вопросы пачкой. Или договориться, что вы все их накидываете в скайп или телеграм, а коллега его читает два раза в день. Или, если он читает мессенджер постоянно, вы сначала фиксируете вопросы в блокноте и потом приходите сразу с пачкой.

Особенно актуально для:

- **начинающих** (джуниоров). Да, многое непонятно. И у вас будет много вопросов. Это нормально. Но не отвлекайте коллег слишком часто. Собрали вопросы, подошли, всё узнали, ушли. Цените время коллег. Это вам зачтется;
- **тестирования нового функционала**. Новые требования, куча замечаний. Не накидывайте их по одному, сначала дочитайте до конца, соберите в блокнотике список, а потом его целиком вложите в задачу. Это, кстати, спасет от случайной потери данных. Мне так JIRA несколько раз удалила длинные комментарии, когда я случайно закрывала окно или подыхал компьютер. Всё, теперь только Notepad++, а из него уже переносить, когда комментарий будет закончен!

И, конечно, не забывайте о правиле 20 минут!

Правило 20 минут

Сначала попробуй решить задачу сам, потом обращайся за помощью. Если за 10–20 минут не нашел решения, можно и коллегу спросить.

Правило 20 минут актуально для всего:

- забыл ссылку на документацию — хотя бы минутку поискал сам, а уже потом спросил в чатике;
- увидел незнакомое слово? Погуглил, потом пришел к разработчику с более осмысленными вопросами, показывая, что уже попробовал узнать что-то сам;
- сломался автотест? Сначала пытаешься понять сам, что не так, потом идешь к разработчику;
- делаешь домашнее задание на тренинге? Сначала попробовал сам, потом уже с конкретными вопросами приходишь к тренеру.

Фактически правило 20 минут помогает нам не отвлекать коллег тогда, когда мы в состоянии найти решение сами. Бывает и такое — спрашиваешь в общем чатике, а тебе никто не отвечает. И вот ты пять минут ждешь ответ, а потом плюешь и начинаешь пытаться решить проблему сам. И ведь решаешь! И когда, наконец, в чате кто-то откликается, ты уже давно всё сделал.

Так, может, перед тем как задать вопрос, сразу попробовать решить его самому? Это покажет коллегам, что вы цените их время, а еще вы получите хороший профессиональный рост. Потому что с каждой решенной задачей вы приобретаете опыт. А с каждой подсказкой остаетесь на том же месте, что и раньше.

Я попробовала отправить SOAP-запрос, не получается :(Говорит, авторизация провалена, но я вроде правильно логин-пароль указала...

Инструменты исследования

Вот мы прочитали ТЗ, задали вопросы... А как оформить результат, чтобы не забыть всё, что мы узнали? Я расскажу об инструментах, которые использую сама.

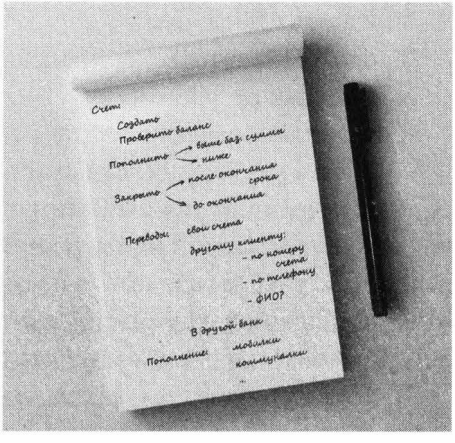
Блокнот

Rocket-science[19], не правда ли? ツ

Но на самом деле блокнот и ручка были и остаются лучшими инструментами тестировщика. Это даже лучше, чем открыть Word на компьютере и писать туда. Потому что задействуются разные участки мозга: стучать по клавишам или писать от руки (о пользе рисования от руки можно почитать в книгах по визуальному мышлению). Когда мы записываем свои идеи, это помогает сгенерировать новые. Что весьма полезно ;-)

Лично я, когда читаю требования, то обычно беру блокнот и записываю туда свои идеи:

- набор тестов, которые надо провести (это помогает найти проблемные зоны в ТЗ, но об этом мы подробнее поговорим в главе про тестирование документации);
- взаимосвязи — что от чего зависит. Могу нарисовать, могу написать. Потом уже перенесу свои рисунки в документацию, доведя до ума. Но вначале мне проще зарисовать тяп-ляп, ведь самое главное тут — работа мысли, а не красота.



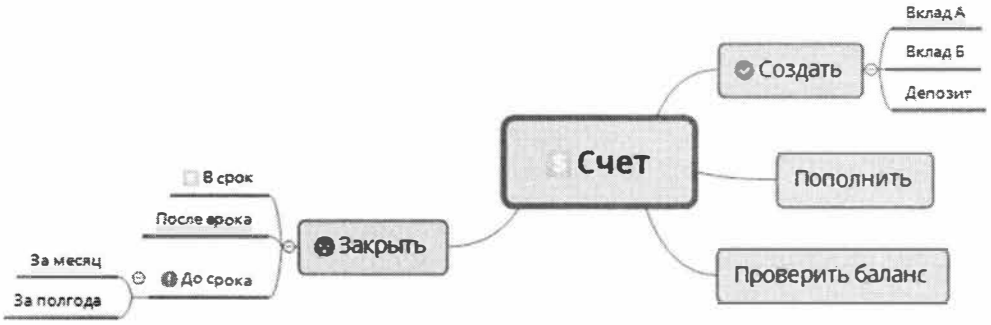
Интеллектуальная карта (mind map)

Это, пожалуй, лучший способ рисования. Вы выделяете тестируемый объект и рисуете вокруг стрелочки «а что он может делать». Получается очень наглядно:

Что можно сделать со счетом?

- создать его;
- пополнить;
- проверить баланс;
- закрыть.

Каждую веточку можно детализировать вглубь. Что значит «создать счет»? Они же бывают разных типов — так и пишем, вот такой есть вклад, сякой, а есть просто депозит. Каждую ветку можно углублять хоть до посинения.



А еще в каждой веточке можно поставить какую-нибудь картинку или значок. Так карта становится симпатичнее + по ней проще искать. Можно отмечать самые важные моменты или, наоборот, то, что используется реже всего. Тогда, когда мы откроем нашу карту и начнем составлять тесты, у нас уже будут готовые приоритеты — что тестировать в первую очередь, а что отложить на потом.

Инструменты для рисования mind map

- Для индивидуальной работы:
  - XMind — по моему мнению, самый удобный и красивый инструмент;
  - FreeMind — менее симпатичен, но потребляет меньше памяти и потому работает быстрее.
- Для работы в группе (можно параллельно редактировать одну карту):
  - MindMeister — пока самый популярный инструмент[20];
  - MindMup 2.0 — альтернатива.
- Ну и, конечно, всегда можно погуглить:
  - по ключевому слову `mindmap` — наиболее актуальный на сегодня инструмент будет выше в выборке;
  - по ключевым словам `mindmap online` — если хотите работать с группой, ищите сразу `online`.
- PowerPoint — а почему бы и нет? В нем можно нарисовать вполне себе симпатичную диаграмму. При желании, конечно. Но для карты я бы использовала все же блокнот или MindMap.
- Другое — разумеется, для описания требований существуют специализированные программы. Которые помогут нарисовать, скажем, UML-диаграмму. Но это все инструменты аналитика. А мы не аналитики — нам достаточно на базовом уровне понять, что именно делает система, и зарисовать это, если ничего лучше в требованиях до сих пор нет.

А вот если захотите развиваться и расти в сторону аналитика, то тогда можно будет взять уже другие книги и копнуть поглубже в сторону красивого описания требований.

Как нарисовать карту приложения?

Неважно, рисуете вы от руки или в XMind. Есть несколько основных правил, которые помогут сделать вашу карту простой и понятной.

- **Изучите ваше приложение**: прочитайте ТЗ, задайте вопросы аналитику. Да просто потыкайте систему и посмотрите, что она умеет.
- **Выделите основные функции**, задав себе вопросы:
  - ✧ зачем пользователю наш продукт?
  - ✧ что он там делает?
  - ✧ что мы хотим, чтобы он делал?

Рисовать мы начинаем с основных сценариев. А потом детализируем их и дополняем карту второстепенными сценариями.

Скажем, в любом интернет-магазине главная цель — найти товар и купить его. Сделать это можно разными способами: поиск через строку поиска, через фильтры в боковой панели, через баннеры с рекламой, через что-то еще... Это будет детализация основной ветки поиска.

А второстепенные сценарии — регистрация, публикация фоточек, обмен мнениями и всякое такое. Их оформляем после основных.

- **Запишите цели через глаголы**.

Мы рисуем карту так, чтобы на первом уровне были глаголы — цели пользователей, а мелкие веточки уточняли способ сделать крупные действия.

Стремитесь использовать именно глаголы, а не существительные — то есть названия функций, а не разделов. Пишите не «настройки» (описание кнопки), а «настраиваем приложение под себя» (цель пользователя).

Это важно, потому что вы получите сразу хорошие названия для чек-листов и тест-кейсов. Мы же их как раз из карты можем брать!

Еще раз делаю акцент на том, что *мы пишем цели через глаголы*. Есть очень важное правило:

Не надо рисовать интерфейс!

Как проще зарисовать сайт? Открыть его и аккуратненько запихать в карту все менюшки и кнопочки, которые мы нашли. Но что такая карта дает? Эти кнопочки я и так увижу, как открою сайт.

Вместо зарисовки интерфейса и кнопочек создайте *карту возможностей приложения*. Ищите цели пользователя и описывайте их через глаголы.

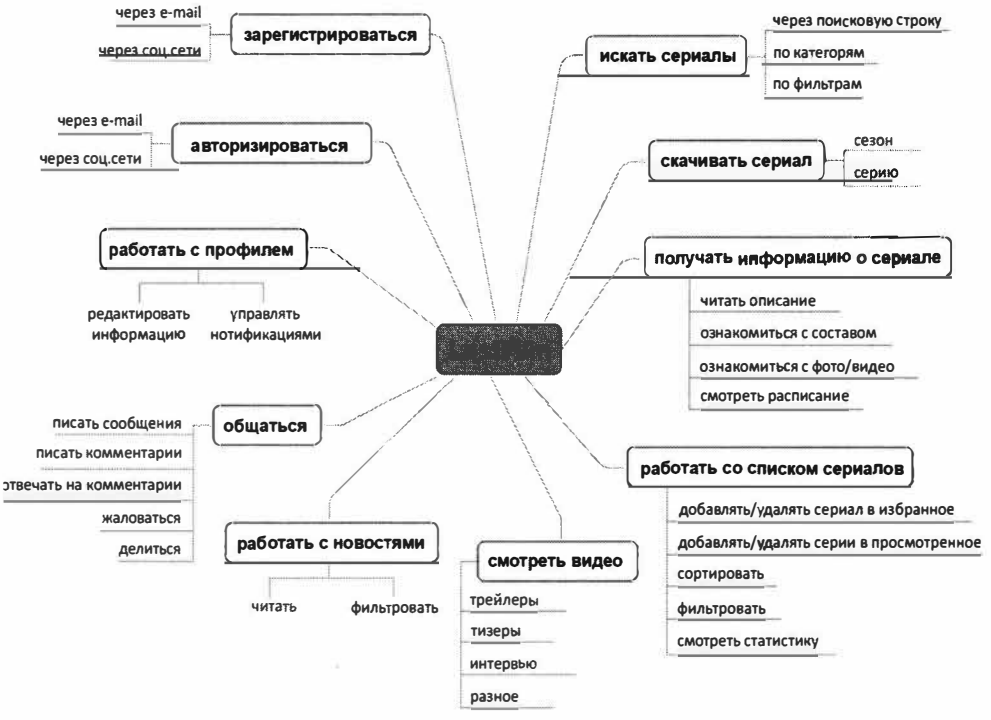
Это очень частая ошибка у студентов. Нужно нарисовать карту приложения? Просто перепишем все кнопочки! А нужно анализировать продукт, без этого никак.

○ **Расставьте приоритеты.** Зарисовав все функции, осмотрите свою карту. Проверьте приоритеты и при необходимости поменяйте ветки местами. Нужно, чтобы в глаза сразу бросалось самое важное. Карту читают по часовой стрелке. Открывая ее, я хочу сначала понять, что это вообще за сайт или приложение, для чего оно будет нужно, а потом уже всякие мелочи типа регистрации и т. п. Мы не используем приложения ради того, чтобы зарегистрироваться в них. С их помощью мы развлекаемся или решаем какие-то проблемы. Поэтому, если уж очень хочется вынести регистрацию/авторизацию на первый уровень карты — она должна быть где-то в конце.

Я специально делаю акцент на авторизации, потому что она есть почти в любом приложении. И некоторые студенты отстаивают право на приоритетность этой функции:

— Но ведь функция есть! И она очень важна! Я не могу сделать покупку без нее. Значит, она главная.

Нет. Никто душевно здоровый не приходит в продукт, чтобы зарегистрироваться или авторизоваться. «Я авторизовалась, ура! Все счастливые расходятся по домам!». Люди ленивы. Вы будете делать лишние телодвижения, если их можно избежать? Будете авторизовываться, если цель достигается без этого?



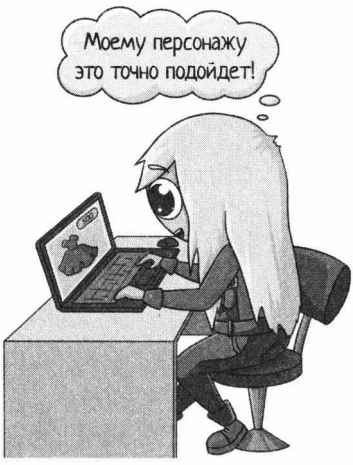
В ряде интернет-магазинов вообще отказались от авторизации — можно сделать заказ и без нее. Что правильно. А то ради одного заказа раз в десять лет нужно зарегистрироваться и спам еще потом от них терпеть... В общем, чем меньше действий — тем лучше!

Да, раньше «так было везде», поэтому до сих пор регистрация/авторизация во многих продуктах присутствует. Но это не повод называть такие функции основными. А еще раньше считалось, что ремень — лучшее средство объяснить ребенку, что он не прав. И что теперь, лупить детей, «потому что раньше все так делали», что ли?

Пример: онлайн-игра

Пример любезно предоставлен некогда тестировщиком игр Ольгой Алифановой.

Я прихожу в онлайн-игру прокачивать своего персонажа. Общаться. Вступать в противостояние с другими персонажами. Исследовать мир. Настраивать игру под себя. Это верно для 90% больших онлайн-игр, но могут быть еще и дополнительные стремления «Укрощать мир под себя» (этим занимается весьма юное поколение игроков — в Perfect World были любители домиков и платьишек, которым драить монстров было неинтересно, а вот платьишки собирать — таки да) — это от игры зависит.



Дальше идет специфика конкретной игры — как в этой игре я могу прокачать персонажа? Выполнять квесты. Убивать монстров (в просторечии «мободроч», извините, когда не по квестам бегаешь, а тупо на одном месте экстерминатус устраиваешь). Использовать предметы, может быть? В игровом внутреннем магазине иногда бывают всякие предметы, добавляющие персонажу очки опыта. Это уже второй уровень карты — раскрываем веточку «Прокачивать персонажа».

Обратите внимание, я нигде в этом примере интерфейс не отрисовываю. Я не пишу «Профиль. Главный экран. Экран параметров персонажа. Чат». Потому что я анализирую именно суть продукта, а не подробности его визуальной реализации.

Примеры хороших карт

Вы можете найти примеры хороших карт у меня в конфлюенсе в разделе «Работы студентов: Майнд карты»[21]. Посмотрите и сделайте по аналогии!

Подведем итоги

Не надо скрупулезно записывать интерфейс. Карта вида «Каталог: для женщин — джинсы, брюки, юбки, ..., для мужчин — ..., для детей — ...» — это зарисовка интерфейса. Карта вида «Просмотр товаров: по полу, по категории, по материалу» — это анализ продукта.

Для чего приходит пользователь? Для, скажем, просмотра ассортимента. Как он сможет его просмотреть? Например, вот так. Как это будет реализовано или уже реализовано — разделами каталога ли, фильтрами ли, еще каким-нибудь удивительным способом — нам неважно.

Таким образом, первый уровень карты — глаголы, отвечающие на вопросы: зачем наш продукт нужен и что пользователь будет там делать. Второй уровень карты рассказывает, как он сможет это сделать.

И не забывайте о приоритетах! Читая по часовой стрелке, я должна вначале видеть самое главное.

Такая карта позволит:

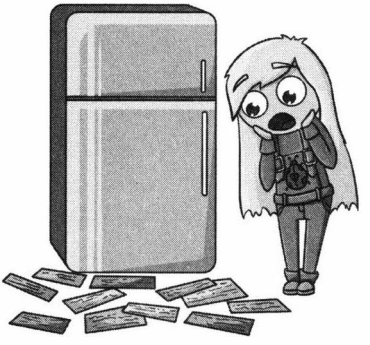
- понять, зачем именно нужно приложение, даже если я его никогда не видела;
- составлять тест-кейсы и чек-листы. Можно просто брать название ветки — и вот он, тест-кейс!

Домашнее задание

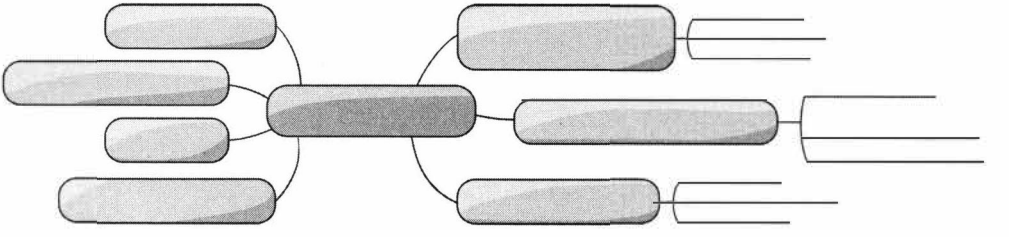
Вопросы для самопроверки

1. Приведите пример программы.
2. Протестируйте ручку. Какие ваши действия и проверки?
3. Чем открытый вопрос отличается от закрытого? Какой лучше использовать?
4. Много вопросов — это хорошо?
5. Обязательно ли наличие майнд карты на проекте?

Магнитики на холодильник



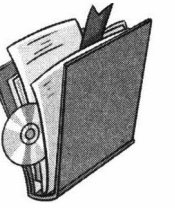
О, нет! Mind карта портала **http://software-testing.ru/** рассыпалась на пол. Сможете восстановить ее? Надо взять с пола только нужные магнитики и вернуть их на холодильник. На сопровождающей вопрос картинке нарисована лишь часть карты, но есть лишние кусочки! Что из этих магнитиков стоит взять, а что нет? В каком порядке расположить кусочки?



1. Software-testing
2. Портал
3. Работа
4. Ответить на чужой вопрос, в котором вы эксперт
5. Форум
6. Тренинги
7. Магазин
8. Найти работу
9. Обсудить интересные темы
10. Записаться не тренинг
11. Статьи на портале
12. Для начинающих
13. По проектированию тестов
14. По автоматизации
15. Купить обучающее видео
16. Разместить свою вакансию
17. Пообщаться с тестировщиками
18. Статьи в ленте блогов
19. Задать свой вопрос
20. Авторизоваться
21. Зарегистрироваться
22. Статьи в рассылке
23. Прочитать актуальные статьи по тестированию.

На всякий случай привожу скриншот портала (мало ли, когда и где вы будете читать эту книгу).

Портфолио



Начните составлять собственное портфолио!

1. Выберите программу или сайт, которые используете часто.
2. Нарисуйте карту возможностей приложения.
3. Покажите другу, который в глаза не видел это приложение.

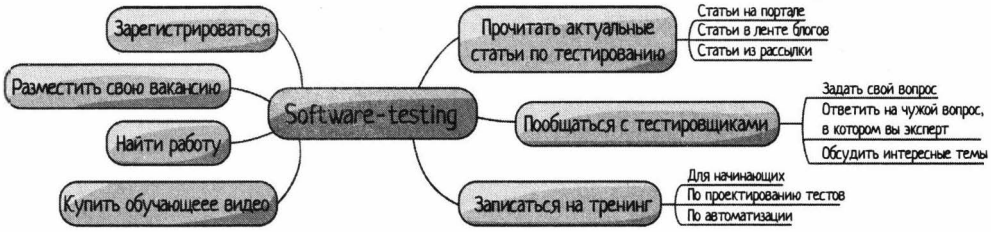
Если друг поймет по вашей карте, что за приложение вы ему показываете и для чего оно нужно, а также какие его особенности самые главные, — поздравляю, вы справились с ДЗ!

Ответы

Магнитики на холодильник

Помним, что, глядя на карту, я должна видеть не функционал, а возможности системы. Ради чего мне туда идти? Что я получу или смогу сделать? Поэтому перебор разделов меню (портал, работа, форум, тренинги, магазин) мы выкидываем и думаем, ради чего они там. Превращаем скупое название в глаголы, и вот у нас уже есть основа:

1. Прочитать актуальные статьи.
2. Пообщаться с тестировщиками.
3. Записаться на тренинг или купить обучающее видео.
4. Найти работу или тестировщика.



В дочерних пунктах раскрываем каждую тему.

Также помним о приоритетах. Регистрация — самый очевидный пункт, который есть почти в любом сайте, но я прихожу на сайт не ради регистрации. Поэтому вначале — все основное, а уже потом — побочные действия

Ответы на вопросы для самопроверки

1. Приведите пример программы.

Будильник на телефоне, встроенное в телефон приложение для SMS, OZON.ru, Wildberries и другие интернет-магазины. Word, Excel, калькулятор на винде, блокнот... Продолжите список ;-)

2. Протестируйте ручку. Какие ваши действия и проверки?

В первую очередь нужно узнать, что это за ручка и зачем она нужна. Иначе вы побежите исследовать, как хорошо она пишет, а вообще-то это была дверная ручка ;-)

3. Чем открытый вопрос отличается от закрытого? Какой лучше использовать?

На закрытый можно ответить только «да» или «нет». На открытый так ответить нельзя. И открытый вопрос лучше, так как дает больше информации. А закрытый может усугубить непонимание. Вот вы ручку из вопроса 2 взяли и спрашиваете «Вы часто ею пользуетесь?». Ответ «да» подтверждает вашу догадку, что речь о пишущей ручке, которую собеседник при этом держит в руке. А открытый помог бы узнать правду ;-)

4. Много вопросов — это хорошо?

Смотря каких. Ясное дело, что новичку ничего не понятно, и он будет задавать много вопросов. Но одно дело — задавать много «правильных» и вдумчивых вопросов и совсем другое — бегать к наставнику каждые 5 минут, не давая ему работать. И закидывая десятком закрытых вопросов вместо одного открытого (иногда думают, что это впечатляет, но нет). Задавайте вопросы пачкой и помните про правило 20 минут — сначала подумайте над ответом сами.

5. Обязательно ли наличие майнд карты на проекте?

Нет, конечно! Это просто один из инструментов, но вовсе не обязаловка. А вот вам как новичкам я его настоятельно рекомендую применять — поможет собрать мысли в кучу ;-)

Глава 2
ТЕСТ-КЕЙСЫ И ЧЕК-ЛИСТЫ

Карту нарисовали? Продукт изучили? Хватит бездельничать, пора тесты составлять! В этой главе:

- проектируем тесты;
- оформляем тест-кейсы;
- пишем чек-листы.

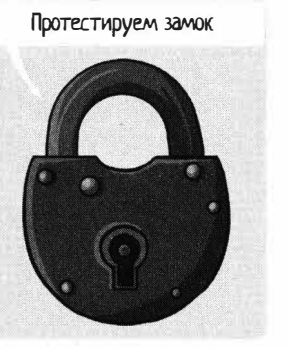
Проектирование тестов

В предыдущей главе мы обсудили, с чего начинается тестирование:

1. Выясняем суть — читаем ТЗ, задаем вопросы.
2. Проводим тесты — проверяем в первую очередь приоритетный функционал. А всякие извращения и попытки сломать оставляем «на потом».

Но вот мы выяснили, что именно тестируем. Мы даже примерно представляем себе, что именно надо проверить. Но примерное представление — это еще не тест. При продумывании тестов нужно помнить о том, каким должен быть тест и о приоритетах выполнения.

Каким должен быть тест?



Конкретным!

Прочитав тест, я должна понять:

- что мне сделать;
- как это сделать.

Допустим, мы тестируем навесной замок.

Как будем его тестировать? «Ну, я его открою и закрою» или «проверить, что он открывается и закрывается» — это не тест.

1. Я беру подходящий к замку ключ, вставляю в замок, поворачиваю дважды, дергаю за ручку — дверь должна открыться.
2. Дергаю за ручку очень сильно и резко — дверь не должна открыться.
3. Сильно бью, скажем, молотком в районе замка и после этого дергаю за ручку. Удар не влияет на запорный механизм — дверь не должна открыться.
4. ...

Вот это тесты! Из них понятно, что я делаю, как и какой результат конкретно я должна получить.

Пишите хорошие, конкретные проверки! Помните, ваши тесты могут читать другие люди. И если по тесту непонятно, что именно надо сделать, — это плохой тест. Потому что придется искать его автора и уточнять, что именно это значит. А автор заболел, уволился или просто забыл, что имел в виду.

У теста есть результат

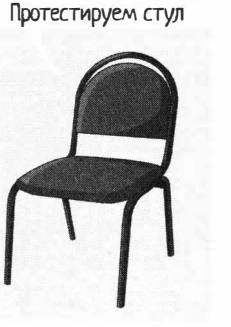
Это важно, поэтому вынесено в отдельный пункт со своими примерами.

Чем отличается тестировщик от обычного человека — он всегда фиксирует какой-то результат. У тестировщика может быть заранее некий ожидаемый результат, а может и не быть. Но что-то он всегда фиксирует, хотя бы для того, чтобы сравнить со старой версией приложения, — это и есть тест.

Протестируем стул. Самый простой, который на конференциях в залах стоит.

Обычный человек пишет: «сесть на стул», «уронить стул». Но тестировщик пишет: «Посадить на стул мужчину весом 150 кг и убедиться, что сидящий не упал, а стул сохранил форму».

Что вы хотите там проверить? Какой результат зафиксировать? Представьте, что через месяц вам дадут новую версию стула, и вам надо будет сравнить данные.



Приоритеты выполнения тестов

ВСЕГДА начинаем с позитивного тестирования и только потом проводим негативное. Потому что задача тестировщика — не найти как можно больше багов, а дать наиболее полную информацию по приложению[22].

Только в идеальном мире у тестировщика достаточно времени на выполнение всех тестов. А реальной жизни это выглядит так: «Я тут слегка изменил, проверь быстренько и через 15 минут релизимся[23]». Но начинаешь проверять, и там все сломалось... О чем важнее сообщить: о том, что один раз из ста при быстром быстром переключении вкладок некрасиво мелькает картинка, или о том, что кнопка «Купить» не работает?

Если пользователь зайдет к нам на сайт и не сможет положить товар в корзину, то ему будет глубоко наплевать, что при вводе всяких спецсимволов или SQL-инъекций мы пишем красивые сообщения об ошибке. Не работает основной сценарий — зачем дальше вообще что-то проверять?

Время есть! Проводим все тесты

У тебя 15 минут, проверь мне всё

Вот представьте себе — проверяем форму регистрации. Что только тестировщик не запихал при проверке в поле «Имя»: и цифры, и спецсимволы, и матные выражения, и целый том «Войны и мира»... Много проверил! Но тут приходит начальник и говорит:

— Хватит! Нам пора выпускаться. Что там, как? Работает?

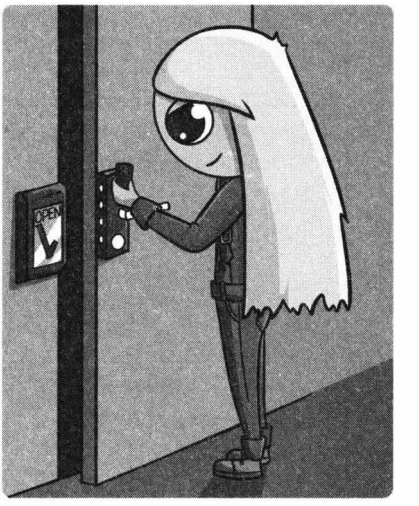
— Работает!

— О'кей.

Обновили приложение — не работает регистрация с именем «Иван». Слишком распространенное — система считает это попыткой обмана. Ну и кому теперь какая разница, что на «вредных» действиях все хорошо, если на нормальных плохо?

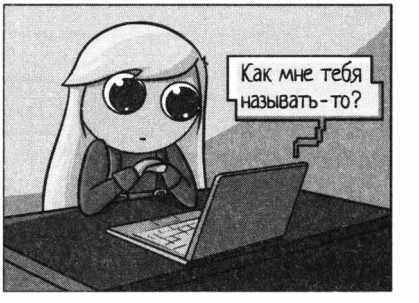
Расставляйте приоритеты, начинайте с позитива. Давайте разберем подробнее, что же такое позитивное и негативное тестирование.

Позитивное тестирование



Позитивное тестирование проверяет, что основной функционал работает. Все сценарии использования системы выполнимы и приводят к ожидаемому результату, а не к ошибкам.

Очень важно уметь генерировать самые разные проверки. Чуть позже мы еще поговорим о классах эквивалентности[24] и о том, как выкинуть лишние тесты. Но, чтобы уметь выкидывать, нужно, чтобы было, что выкидывать. Посмотрим на примерах.



Поле «имя» при регистрации

Казалось бы, что может быть банальнее? Просто строковое поле, обычно никаких ограничений нет. Но если у нас нет доступа к коду, мы не знаем наверняка. Поэтому генерим варианты. Какое бывает имя?

Вариации	Примеры
• Свое имя	• Ольга
• Мужское / женское	• Александр
• Род непонятный	• Саша
• Распространенное	• Иван
• Редкое	• Олеся
• Двойное	• Анна-Мария
• Короткое имя	• Ева
• Длинное имя	• Аполлинария
• Составное имя	• Розалинд Аруша Аркадина Луна
• Иностранное имя	• Айгуль
• Краткая форма имени	• Оля
• Уменьшительное	• Олечка
• Имя на английском языке	• Olga
• Может начинаться фамилия	• Александр(ов)
• Разные варианты написания	• Наталья/Наталия, Алена/Алёна
• Полное = краткое	• Лана, Ася
• С буквой Ё	• Пётр, Семён
• Разный регистр	• ОлЬгА

Окончание табл. 1.2

Вариации	Примеры
• Опечатки • Английская раскладка • ФИО • Фамилия • Пустое поле • Только пробелы • Апостроф • Символ «à» • Нерусский алфавит • Спецсимволы • XSS-атака	• Олга • Jkmuf • Киселева Ольга Евгеньевна • Киселева • • • д'Артаньян • Ivanà • 阪熊 • Ё!»№;%:?*()_+ö ß • Иван<script> alert(123)</script>

Функция вычисления корня в калькуляторе

$$\sqrt{4}=2 \quad \sqrt{3}=1{,}732\ldots$$

Основной тест — проверить, что корень из корректного числа действительно вычисляется.

И дальше мы уже думаем, а какие бывают варианты?

- После вычисления корня остается целое число (корень из 4 = 2).
- После вычисления корня остается дробное число (корень из 3).

Хм, а что если дробное число у нас будет не только после вычисления корня, но и до? Можем же мы взять корень из числа 2,2? Позитивный тест? Позитивный!

Также можно разделить числа на небольшие — до 100, например. Потом взять интервал от 100 до размера integer[25] и, наконец, сколько влезает в наш калькулятор.

Не забудем и про 0. Позитивный тест? А как же! Корень из 0 равен 0, а не ошибке!

$$\sqrt{0}=0$$

Из основного, пожалуй, всё. Но могу и продолжить...

Работа с корзиной в интернет-магазине

О, вот где простор для воображения!

Пользователь столько разных сценариев может выполнить! Но в первую очередь возьмем основные, самые короткие. Потому что, если уж они не работают, то длинные цепочки (добавил – отредактировал – удалил – снова добавил – и т. д.) проверять точно не стоит. Итак:

- Добавление товара в корзину.
- Добавление второго товара в корзину (того же самого, счетчик должен увеличиться).
- Добавление второго товара другого типа.
- Редактирование числа товаров, находящихся в корзине: увеличение на несколько, уменьшение.
- Удаление товара из корзины.
- Ваш вариант...

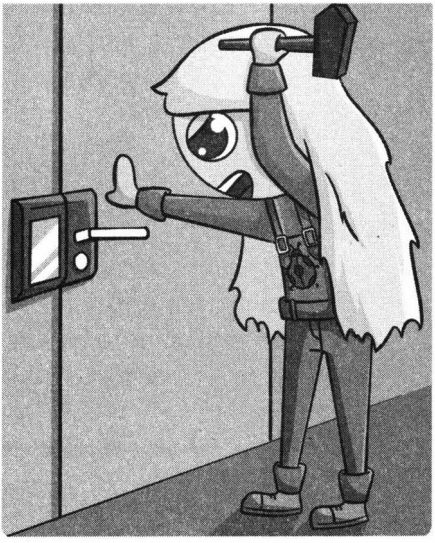
Придумать тестов здесь можно очень много. Особенно, если начать проверять цепочки: например, добавил товар в корзину — передумал, удалил — ой, нет, все-таки надо, снова добавил. Думаете, будет работать, если работает по отдельности? Не-е-е-ет, ребята, вы же тестировщики! Никогда не верьте программам «на слово»! Придумали сценарий? Проверьте!

Тем более что такой сценарий вполне может упасть — мы же уже удалили выбранный товар из корзины, так? Так вот, система вполне может не дать нам его снова добавить. Типа «ты уже отказался, алё, я все помню!». Корректно такое поведение? Нет!

А сам сценарий позитивный? Да! Хотя уже и с нотками извращения, надо признать.

Негативное тестирование

Негативное тестирование пытается сломать систему. Попросили ввести дату? Введу символы! Система падает при запросе без авторизации — отправлю такой запрос! Дали в руки карандаш? Обязательно сломаю! То есть моя задача — сделать что-то «не так, как надо» и посмотреть на поведение системы.



Надо помнить, что негативное тестирование не менее важно, чем позитивное. Потому что наши пользователи — люди, а не роботы. А людям свойственно ошибаться. И всегда надо об этом помнить.

Если я приду на сайт, сделаю заказ и все будет хорошо — я приду туда еще раз. Но если я приду, сделаю заказ и случайно где-то ошибусь — например, вставлю скопированное в аське сообщение вместо цифры, то я хочу увидеть тактичное замечание, а не краш[26] всей системы.

Вообще в наше время обычно для решения проблемы пользователя (например, что-то купить надо) существует большой выбор сайтов. Посмотрев на них и поняв, что функциональность, нужная ему, есть везде, пользователь выберет наиболее красивенький и удобный сайт.

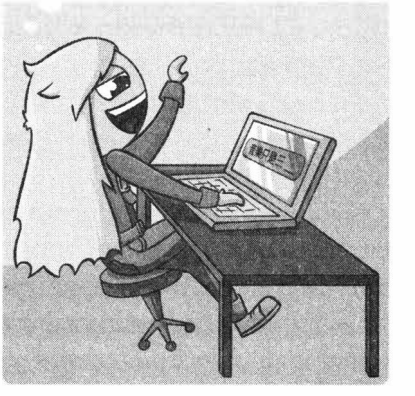
Но, как бы ни был такой сайт удобен, если он не в состоянии отработать при влиянии человеческого фактора, пользователь рано или поздно уйдет. «Шаг влево, шаг вправо — расстрел» — кому это понравится? Хочется иметь возможность ошибаться и исправлять ошибки, а не получать «по рукам» страшными сообщениями об ошибке на весь экран.

Поэтому мы и проводим негативное тестирование. Что такое негативное тестирование? Это ввод заведомо некорректных данных. Вводим и смотрим, как ведет себя программа, понятные ли сообщения об ошибке выдает...

Вспомним наши примеры, какие будут тут негативные тесты?

Поле «имя» при регистрации

Имя?! На тебе иероглифы!



Мы уже проверили «нормальные» варианты имени: женское, мужское, редкое, распространенное... Теперь попробуем задать себе вопрос «А что будет, если имя — не имя?»

Например, что если вместо имени написать фамилию? Или полное ФИО? А, может, просто ник, как в игрушке?

Что будет, если использовать апостроф или символ «à» — д'Артаньян, Ivanà?

Что если оставить поле пустым или заполнить его только пробелами? А если написать имя, перепутав раскладку клавиатуры? Или просто забить все известные спецсимволы? Или попробовать выполнить XSS-атаку?

То есть можно сначала пойти в сторону «не имя», помня, какое у нас поле. А потом перейти к «серебряной пуле» — проверкам на любую строку (спецсимволы, пустота, пробелы...).

Функция вычисления корня в калькуляторе

Первое, что приходит на ум — а что будет, если вычислить корень из отрицательного числа?

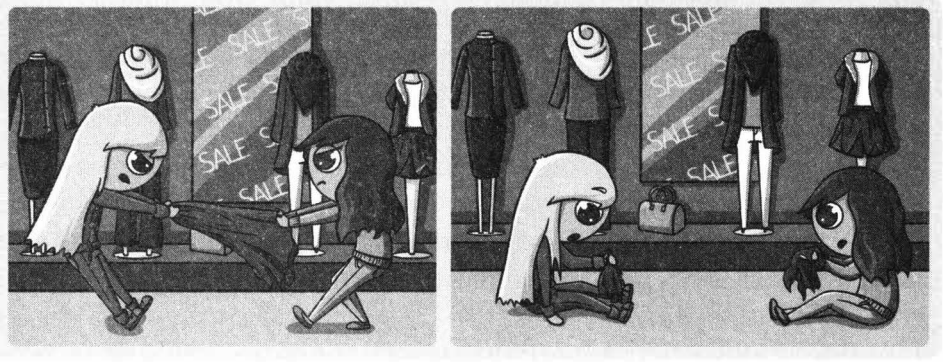
Но что еще тут можно придумать?

- Корень из пустоты — мы не можем ввести строку отрицательной длины, но вот граничное значение (строку нулевой длины) можем!
- Корень из символов — надо проверить, что скажет система, если ввести или вкопипастить туда что-то символьное. Причем символы мы делим на русские, английские и спецсимволы!
- Корень из значения «четыре». Строки «четыре», а не цифры. Вводимые символы можно поделить на абракадабру и «типа число».

Видите? На самом деле тестов не так уж и мало! Отдельно хочется высказать на тему «ввести очень большое число, максимально большое». Попробовать можно, почему нет? Но это более негативно скажется на сценарии возведения в квадрат, чем на вычислении корня.

При вычислении корня можно не вводить максимально большое число, а ввести такое число, чтобы корень из него (дробное значение) получился очень длинный и не уместился на экран, — вот что тогда будет, обрежется или сломается?

Работа с корзиной в интернет-магазине



Тут, опять же, можно найти числовое поле и поиграться с ним, как мы это только что проделали с калькулятором. Поле «количество товара» очень подойдет! Я повторяться уже не буду, делайте по аналогии.

А мне в этом примере хочется упомянуть о важной особенности всяких веб-приложений и главном негативном тесте, который обычно все и ломает.

Запомните всего два слова: разные вкладки!

Чувствуете? Давайте поясню. Негативные тест на удаление товара из корзины — попытаться удалить уже удаленный товар. И тут начинаются варианты того, как это может быть сделано:

- открыли корзину в двух вкладках браузера. Сначала нажали «удалить» в одной, потом — во второй. То есть произвести попытку удалить то, что ты сам уже удалил из своей же корзины;
- попытка удалить удаленный админом товар. В первой вкладке под админом удаляем товар вообще, в принципе, а в другой — пытаемся его под пользователем удалить из корзины.

И, кстати, также можно попробовать добавить удаленный админом товар или отредактировать его количество. А еще админ может не удалить товар, а перенести его в другую категорию. И вот тут сломаться ничего не должно! Если в случае удаления мы должны увидеть корректное сообщение об ошибке, то в случае переноса просто продолжить работу.

А что будет, если админ не передвинул товар в иерархии магазина (в другую категорию переместил, исходно неверно был размещен товар), а просто поправил, отредактировал описание? Тоже ничего сломаться не должно!

И даже если у нас не магазин, а что-то другое, всегда задумайтесь, как можно попытаться применить технику разных вкладок.

Например, мы можем создавать карточку — человека, здания, той же книги, чего-то еще... Попробуйте ее отредактировать в два окна. В одном изменить одно поле, сохранить, а потом во втором изменить другое поле и тоже сохранить. Или что-то удалить, а во втором окне добавить или изменить.

Всегда пробуйте играть такими вещами — это часто заканчивается плачевно для программы. И даже если команда примет решение пока не исправлять такие дефекты, как некритичные, неважно. Нам главное о них знать! Предоставьте информацию, а потом уже решайте, что с ней делать...

Хочется привести еще один пример из реальной практики. Тоже веб-интерфейс, в котором можно нажать кнопку «создать» и добавить новую карточку. Пользователь добавляет, а у него через раз формочка падает. Почему?

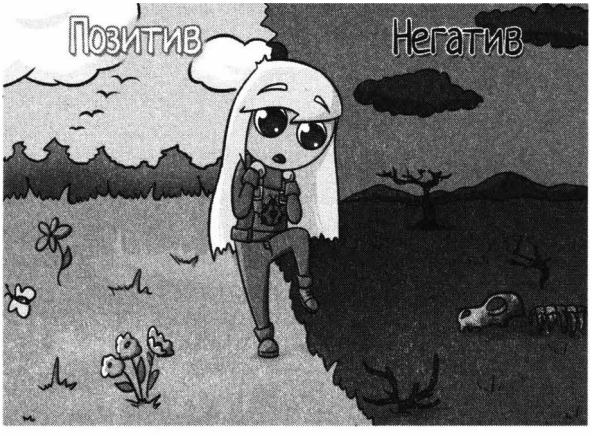
Стали выяснять. И поняли. Пользователю надо было завести сразу с десяток карточек, вот он и нажимал на «создать» несколько раз, зажав клавишу <Ctrl> (открыть в новой вкладке). И потом уже ходил по вкладкам и заполнял данные.

Вот, казалось бы, где тут негативное тестирование? Ведь пользователь не вносит противоречивых изменений, меняя одну и ту же карточку. Нет, он создает новые, то есть вносит информацию в разные карточки. Но вот

ведь как — система считала открытое окно «новая карточка» единым целым, громко возмущаясь наглыми попытками пользователя запихать туда то одну информацию, то другую.

Так что дерзайте! Открывайте разные вкладки и вперед, ищите информацию о том, как ведет себя именно ваша программа при противоречивых воздействиях на нее.

Где граница «позитив-негатив»?



Выучив новую тему про позитивное и негативное тестирование, мои студенты обычно радостно вводят эту границу в свои чек-листы. Вот туточки позитивные тесты, а туточки — негативные.

Но этой границы нет! Негативная проверка не означает, что система на нее должна выругаться непечатным образом. Позитив и негатив — вообще категории не абсолютные, а относительные.

Это как шкала — если речь об именах:

- Ольга позитивнее Оли;
- Оля позитивнее Олечки;
- Олечка позитивнее Цудлоыдвалоды.

Чем менее вероятен сценарий пользовательского поведения, тем негативнее проверка. Совершенно неважно, ругается там система или нет. Поэтому не надо жестко бить в чек-листах проверки на «блок позитива» и «блок негатива». Читающему тесты из порядка проверок должно быть ясно, что вы считаете наиболее позитивным, наиболее приоритетным значением, а что наименее. Если дать вам на форму пять минут, то вот что успели проверить, то успели, а все, что идет после — не успели. Ужасно будет, если Цукдоцудо успели, а Ольгу не успели, а потом окажется, что как «Ольга» я зарегистрироваться не могу.

Так что при группировке тестов используйте лучше знания о классах эквивалентности, о которых мы поговорим чуть позже. И уже внутри каждой группировки всегда «сначала позитив, потом негатив», не иначе.

Повторим...

Когда мы начинаем тестировать, то всегда начинаем с позитивных проверок —потому что времени часто в обрез. Что первое написано, то и проверим. Важно начинать с самого главного.

Но нельзя торопиться в ущерб качеству. Чтобы определить приоритеты, сначала выясняем, что мы вообще тестируем. Как эта форма/поле будет использоваться в дальнейшем? Потому что если мы просто сохраняем справочную информацию в профиле пользователя, которая видна только в профиле, — это одно. А если мы по имени определяем пол и делаем рассылки — совершенно другое. Не очень классные могут получиться рассылки:

- Уважаемый Наталья!
- Милая Иван...
- Дорогая Какашка!

Зная, как информация будет использована, мы уже может придумывать хитрые проверки «А что если?». Система считает женскими именами все, что оканчивается на гласную? А что если ввести Саша? Или Никита? Или Какашка? Так мы ставим под сомнение саму логику работы системы, выявляя в ней узкие места. В идеале мы говорим о проблемах на этапе чтения ТЗ и вместе придумываем другой вариант. Ну или хотя бы в момент тестирования...

Также не забываем, что нам важно понимать результат. Тестировщик не ограничивается описанием «что бы я проверил» — он заранее думает о результате. Вот мы взяли список из разных имен. А какой должен быть результат у каждого теста? Неизвестно! Все зависит от системы и ее функции. Если она выдает подсказки по именам — результат один, если просто сохраняет как строку — другой, если определяет по имени пол для рассылки — третий. Заранее подумайте про результат и найдите того, кого можно спросить о нем.

И если вы тестируете какое-то поле для ввода, всегда начинайте с тестов именно на это поле, а не на любое абстрактное (символы русские,

английские, перемешал). Я уже приводила пример тестов, которые можно придумать для имени (см. ранее *пример «Поле „имя“ при регистрации»*).

Да, часть из них мы выкинем на этапе тест-анализа. Но, чтобы уметь выкидывать тесты, сначала надо научиться их генерить! А для этого задаем себе вопрос: «Хм, имя? Какое бывает имя?» и записываем всё, что приходит на ум. Вот вам и тесты!

Так как оформлять-то?

Варианты оформления бывают разные. Да хоть майнд-карту нарисовать — вполне себе описание тестов. Но основные варианты оформления: тест-кейсы и чек-листы. О них мы и поговорим.

Тест-кейсы

Что такое тест-кейс?

Тест-кейс (ТК) — это подробное описание проверки. Такое, которое можно будет дать человеку с улицы, и он всё поймет.

Вы когда-нибудь покупали мебель в ИКЕА? Даже если взять там малюсенький комод, то получишь к нему увесистую инструкцию по сборке. Это такая мини-книжка с кучей картинкой. На каждый шаг, каждое действие — картинка. Каждый шурупчик показан отдельно и пронумерован: «Возьмите шуруп из пакетика 18, вставьте в этот паз». Очень сложно ошибиться в сборке, когда каждый чих так подробно расписан.

А вот подробное описание проверки для веб-приложения:

ТК1. Авторизация через email

Предварительные шаги

Зарегистрироваться с email **test@test.ru** и паролем **1** (см. тест-кейс «Регистрация по почте»).

Шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать на кнопку «Вход» в правом верхнем углу окна сайта.

3. В поле «Email» ввести значение: `test@test.ru`.
4. В поле «Пароль» ввести значение: `1`.
5. Нажать кнопку «Войти».

Ожидаемый результат

- Открылась главная страница сайта.
- В верхнем правом углу страницы отображается значок личного кабинета (недоступен без авторизации).

Вроде бы просто, да? Посмотрим на другие примеры.

ТК2. Загрузка отчета с учетом корректировок

Шаги

1. Зайти в раздел «Доходы и расходы»: **https://www.example.com/1**.
2. Заполнить данные за любой месяц 2018 года — например, январь: доходы — 100 000, расходы — 10 000.
3. Зайти в раздел «Ручные корректировки»: **https://www.example.com/2**.
4. Добавить корректировку на январь: + 700.
5. Зайти в раздел «Автоматические корректировки»: **https://www.example.com/3**.
6. Добавить корректировку на январь: – 150.
7. Загрузить отчет А за 2018 год: **https://www.example.com/4**.

Ожидаемый результат

Сумма за январь = сумме за год = доходы – расходы + ручные корректировки + автокорректировки = 100 000 – 10 000 + 700 – 150 = 90 550.

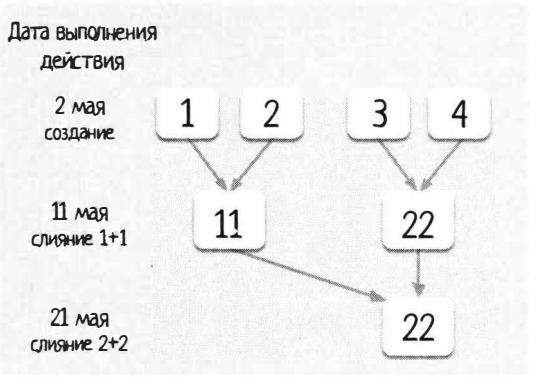
ТК3. Просмотр графа слияния 2 + 2

Шаги

1. Создать дубликаты для слияния 2 + 2: внутри группы объединятся автоматом, сами группы негарантированные. Пример запроса: `saveAndMerge` с ФИО «Двойной Михаил» см. в аттаче *create_duplicate_2_plus_2.txt*.
2. Открыть страницу просмотра негарантированных дубликатов: **https://www.example.com/**.
3. Найти там «Двойной Михаил».
4. Перейти в карточку дубликатов.
5. Подтвердить дубликат по кнопке «Дубликат».
6. Нажать на кнопку «Выполнить слияние».
7. Перейти по ссылке на карточку объединенного клиента.
8. Открыть вкладку «История».
9. Перейти в режим графа по кнопке слева.

Ожидаемый результат

В графе мы видим, что клиенты сначала объединились в группы по два, а потом уже одна группа вошла в другую, вот так:



Что из этих тест-кейсов вы поняли? ;-) При этом, если бы ссылки были рабочие и вас бы посадили за эти системы, вы смогли бы выполнить тест-кейсы и проверить результат? Да, потому что кейсы написаны максимально подробно.

Так что же отличает хороший тест-кейс от плохого? На что обращать внимание при создании кейса? Рассмотрим его по частям и обсудим каждую.

Название

Главное правило хорошего названия:

Причем, это касается любого названия, будь то тест-кейс или баг.

Из названия тест-кейса я должна понять, какую проверку мне надо провести, не заглядывая в описание шагов. Вот, например, я вижу тест-кейс «Корректное имя». Корректное — это какое? И какое действие я должна сделать? Куда это корректное имя вводить? При регистрации? Авторизации? В личном кабинете? При указании имени получателя? В загружаемом в систему файле? Где?

Допустим, при регистрации. Правильно ли будет переименовать тест-кейс в «Регистрация с корректным именем»? Или, скажем, сделать общую папку на тестирование регистрации и внутри уже писать «Корректное имя»?

На самом деле, это название все равно не отвечает на вопрос «А что мне делать то?». Поскольку, что такое «корректное имя», я могу и не знать. Для одной системы «Оленька» будет корректно, а для регистрации в госуслугах — нет. Да и даже если обратиться к проверкам, которые мы придумали

ранее[27], то сколько там корректных имен? Много! Тогда как будет выглядеть наш набор тест-кейсов? Примерно вот так:

Корректное имя
Корректное имя
Корректное имя
Корректное имя
Корректное имя
....
Некорректное имя
Некорректное имя
Некорректное имя

И если мне дадут задание выполнить тест-кейс с уменьшительно-ласкательной формой имени, то я буду долго его искать среди десятка похожих названий, заходя внутрь каждого, чтобы понять, о чем речь.

Намного лучше будет смотреться такой список:

Регистрация
Регистрация с длинным именем
Регистрация с составным именем
Регистрация с уменьшительно-ласкательной формой имени
...
Регистрация с пустым именем
Регистрация со спецсимволами вместо имени
...

Проницательный читатель заметит: «Эй! А что значит тест „Регистрация“? Что я пойму из этого названия?», — и будет прав. Вроде только что критиковала абстрактные названия и вот, ведь, сама даю как пример! Поясню.

Этот тест — основа тестирования функционала. Он показывает, как можно зарегистрироваться в системе. Ведь когда мы тестируем функционал, мы начинаем с самого-самого основного теста. Еще не пытаясь ничего сломать. Регистрация? Вводим везде корректные данные. А потом уже начинаем думать «Таааак, а какие бывают имена? А что, если...?». И вот этот тест — он как раз на то, что регистрация в целом работает.

На некоторые базовые тест-кейсы мы потом ссылаемся из других. Об этом мы поговорим подробнее в следующем разделе про предварительные шаги. Например, мы хотим что-то изменить в личном кабинете — для этого нам вначале надо в принципе зарегистрироваться в системе. И тогда мы

ссылаемся на другой тест: *см. тест-кейс «Регистрация»*. Такое короткое название помогает понять, о чем тот кейс, и легко его найти.

Возможно, вы все еще задаетесь вопросом: «Так, а чем плохо название „Регистрация с корректным именем“? Разве я не могу проверить в одном тест-кейсе все вариации корректных имен?». И это отличный, правильный вопрос!

Да, вы можете проверить в одном кейсе все корректные имена. Существует и такой способ оформления теста, почему бы и нет? И только тогда такое название будет оправдано. Проблема в том, что обычно начинающие любят давать «громкие» названия, которые реально проверяют пшик.

Например, на одном из моих курсов студенты тестировали Дадату[28]. Возьмем несколько примеров заголовков оттуда и разберемся, что в них плохо.

Исправление опечаток в ФИО

Дадата умеет в том числе исправлять опечатки в ФИО. И вот, значит, студент сдает домашнее задание с тест-кейсом. Читаешь название и впечатляешься: «Исправление опечаток в ФИО». Ух ты, думаешь, сколько же он проверок придумал?

Для понимания масштаба:

- справочник имен — примерно 50 тыс. строк;
- справочник фамилий — 500 тысяч.

Допустим, средняя длина слова будет 5 символов. Опечатка может быть в любом месте — уже для одной строки получаем 25 вариаций опечатки. А тут даже не по именам или фамилиям отдельно, а для ФИО в целом написан тест-кейс!

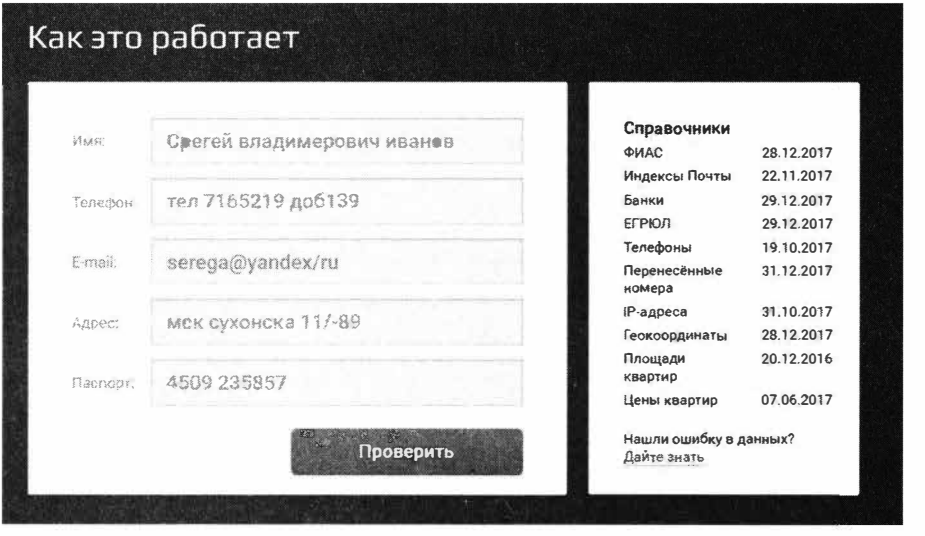
А потом открываешь сам тест, а там… Максимум 10 ФИО. И типа — функционал проверен. Вот уж нет, если даете настолько громкое название — соответствуйте ему. Или выбирайте тест «попроще».

Конечно, ребята защищают свои тест-кейсы: «А я просто хочу проверить саму возможность». Но этот вариант в Дадате не прокатит. Да, если бы опечатки исправлялись только через файл, проверить саму возможность на 5–10 данных вместо тысяч было бы вполне разумно. Например, для smoke-теста. Мы еще поговорим о том, что это такое (см. *главу 9. Классификация тестирования*).

Но в Дадате функционал очистки данных работает в двух режимах:

- по одному человеку;
- через файл.

Проверка «по одному человеку» доступна без авторизации: открыл сайт, ввел ФИО, нажал кнопку **Проверить** — профит!

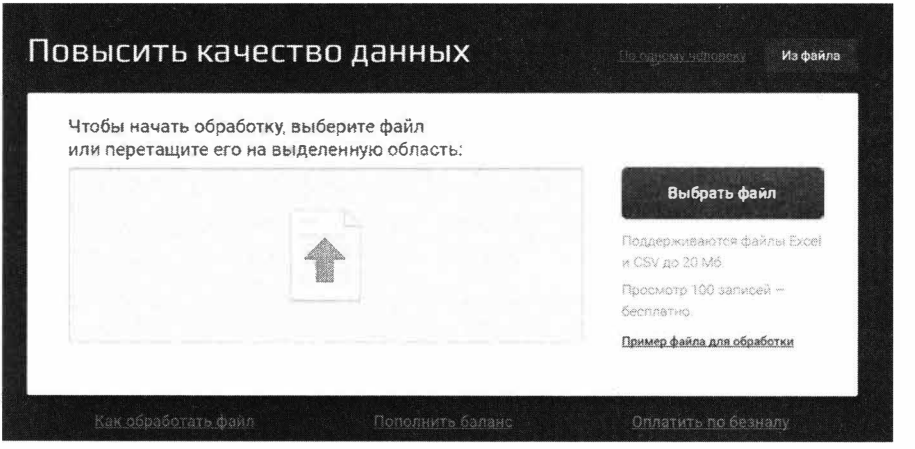


А вот чтобы загрузить файл, нужно:

1. Подготовить файл нужного формата.
2. Зарегистрироваться в системе.
3. Авторизоваться.
4. Загрузить файл — вместо нажатия одной кнопки пройти несколько шагов (все нужны и важны, но ведь путь усложняют).

Как проще проверить, что «опечатки в целом исправляются»? Правильно, через форму «по одному человеку». Но вот ведь засада — тренер-то просит тест-кейсы «на файл»: «Ну примите его так, мы же просто учимся оформлять». Не надо так думать.

Мы никогда не пишем бесполезные тесты, даже в рамках обучения. Это как когда я ходила на курсы по макияжу — там тренер говорила: «Как вы умудрились нанести тон за 5 минут? Профессионалы на это могут час тратить, чтобы все сделать идеально. У вас сейчас есть время, так учитесь. Это



только кажется, что „Я щас на курсах быстренько и так себе сделаю, а уж потоооооом, с реальными клиентами, буду все зоны прорабатывать". Если не научитесь делать хорошо сейчас, то и потом халтурить будете».

В тестировании, да и в любой другой области, все то же самое. Обучение для того и нужно, чтобы учиться делать сразу хорошо. Какой толк от того, что вы научились оформлять бесполезные тесты, которые никогда в реальной работе не будете применять? Все основные темы этой книги завязаны между собой. Нет смысла сразу учиться красиво описывать тест-кейс, если вы не можете придумать идею теста.

Поэтому сначала мы учимся генерировать идеи, потом их описывать, а дальше мы будем расширять количество наших проверок, после чего, наоборот, выкидывать лишнее. Так как в реальной жизни задач всегда полно, а времени мало.

Но вернемся к разбору неправильных названий.

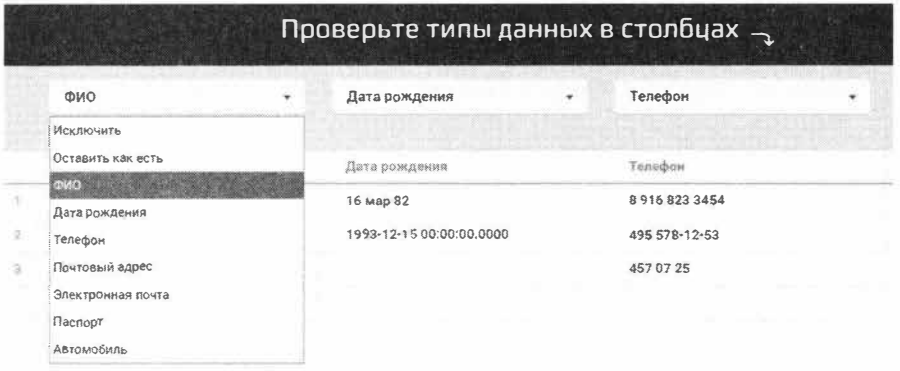
Проверка обработки данных на транслите

Дадата умеет понимать данные на транслите. Логично это проверить. Но как вы думаете, сколько данных должно быть внутри для проверки такого названия?

Дадата умеет обрабатывать:

- ФИО.
- Дату рождения.
- Телефон.
- Почтовый адрес.
- Электронную почту.
- Паспорт.
- Автомобиль.

Это можно увидеть на первом экране обработки файлов в структуре документа:



Сколько из этих данных можно записать как на русском, так и на английском или транслите? Да почти все, за исключением разве что даты рождения, если писать ее исключительно в формате ДД.ММ.ГГГГ. Хотя... Если попробовать написать: `19 мар 2010`, то вот уже и символы — можно попробовать транслит. Хотя его система разбирать и не должна. Ну и телефоны обычно пишут без текста, только цифры.

А самое интересное и полезное — это ФИО и адреса. Которые вполне могут заполнять на транслите. Например, в аэропорту. И такие данные было бы круто разбирать. Но как тестировать? Допустим, мы выделили пока только одну область — разбор транслита в ФИО. Заметьте, это уже сильно меньше, чем обещает название «Проверка обработки данных на транслите». При этом обычно студенты пишут такое громкое название, а внутри — только ФИО, да и тех штук пять.

Но это мы уже обсуждали чуть ранее. Саму возможность обработки транслита можно проверить в форме по одному человеку. Через файл же можно загрузить сразу много данных, проверив все одним махом.

Идем гуглить, исследуем. Нам надо проверить:

- каждую букву алфавита;
- хитрые комбинации (когда буква сама по себе читается так, а в паре с другой эдак — как О и ОО).

Уже получается неплохой набор различных данных! Но тут вспоминаем, что мы тестируем не какого-то абстрактного коня в вакууме, а конкретную систему. Это значит, что мы не можем просто додумать, по какому именно ГОСТу работает система, — нам надо пойти и уточнить у аналитиков.

И вот тут-то мы узнаем, что все еще сложнее, чем казалось! Потому что Дадата умеет распознавать 12 стандартов[29].

Июнь 2017: умная транслитерация ФИО

Раньше Дадата часто пасовала, если имя и фамилия набраны латиницей не по стандарту. Например, не догадывалась, что это всё Юлия Юрьева:

- Ulia Ur'eva
- Iuliia Iureva
- Yulia Yureva

Теперь алгоритм поумнел и понимает 12 стандартов транслитерации. И даже распознает привычные для людей способы записывать ФИО латиницей, которые не соответствуют никаким стандартам.

Тут на один-то стандарт можно 30+ тестов написать, а уж на двенадцать... А насколько больше становится «хитрых» кейсов, которые в разных стандартах работают по-разному. Ух! Сотни, тысячи!

Так что студентам я этот кейс брать не рекомендую. Проверить сразу всё они в любом случае не осилят, да это и не нужно. В конце концов, они должны научиться писать кейсы, а не протестировать нашу систему вдоль и поперек. Поэтому их основная задача — придумать тест-кейс на обработку, который будет полезным, но не на 100500 строк.

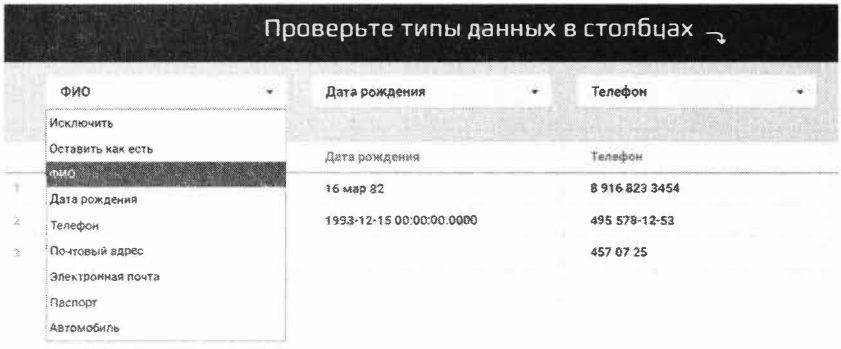
Домашнее задание

Зная информацию с картинок: какие файлы умеет обрабатывать система и какие типы данных она в состоянии обработать, — придумайте тест-кейсы на обработку данных.

Помните, что тесты должны быть полезными. Нет смысла пытаться проверить через файл саму возможность чего-то (исправление опечаток, определение пола, транслитерация). Но и не надо придумывать сложные тесты, которые вы будете составлять целую неделю.

Обработка файла

Вы уже догадываетесь, чем плохо это название? Оно ни о чем не говорит. Вот у меня разные кейсы есть: в файле только ФИО, ФИО + адрес, в файле 1 строка,



10 строк, миллион строк, это эксел, файл CSV с одним разделителем, файл CSV с другим разделителем и т. д. С таким названием кейсов у меня будет 100 штук тестов «Обработка файла» — как мне среди них ориентироваться?

Определитесь, что вы хотите проверить. Об этом и пишите. Кратко, но емко.

Если наполнение файла важно, так и пишите: «Обработка файла с тремя телефонами в одной строке». Или «Обработка опечаток в фамилии». Если это неважно, пишите то, что важно: «Обработка CSV-файла» или «Обработка файла весом более 20 Мбайт».

Тестирование функционала загрузки файла-образца

«Тестирование функционала загрузки» — лишние слова, которые можно выкинуть. Также можно выкинуть и слово «проверка» — это тест-кейс, мы в нем всегда что-нибудь проверяем!

Выкидывайте лишнее — текст ради текста не нужен: «Загрузка файла-образца», всё.

Всегда думайте, можно ли будет понять фразу/название/описание, если выкинуть какое-то слово. Если да — то зачем оно нужно?

Предварительные шаги

Предварительные шаги — это все то, что поможет нам пройти тест-кейс, но прямого отношения к текущему тесту не имеет. Например, регистрация.

Скажем, чтобы поставить лайк под фото, мне нужно войти в систему. Но, чтобы я смогла войти в систему, мне сначала надо зарегистрироваться,

если я не сделала этого раньше. Однако, если я подготовилась заранее, этот предварительный шаг можно выкинуть.

Это как когда готовишь. Скажем, шарлотку.

Шарлотка

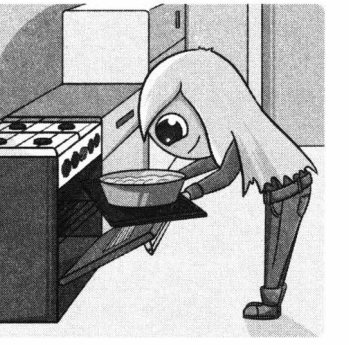
Предварительные шаги

Сходить в магазин и купить:

1. Яйца.
2. Яблоки.
3. Муку.
4. Молоко.
5. Сахар.

Шаги

1. Яйца взбить с сахаром (взбивать не менее 5–7 минут).
2. Добавить муку, хорошо перемешать.
3. Яблоки почистить, удалить сердцевину, нарезать небольшими дольками.
4. Форму для выпечки смазать маслом.
5. На тесто выложить половину яблок (яблоки можно посыпать корицей).
6. На яблоки вылить половину оставшегося теста.
7. На тесто выложить оставшиеся яблоки.
8. На яблоки вылить оставшееся тесто.
9. Поставить в разогретую до 180 градусов духовку.
10. Выпекать в течение 40–60 минут (в зависимости от размера формы).



Ожидаемый результат

Вкусная шарлотка! Которую родные уминают за 5 минут.

Фишка в чем? Если у меня уже есть яйца, я могу их не покупать. Но взбивать их мне все равно придется. Даже если я неделю назад взбивала яйца с сахаром, я не могу их взять сейчас (они уже протухли!). То есть основные шаги я выкинуть не могу, сделав их заранее. А вот предварительные — вполне.

Так же и в мире ИТ. Не надо радостно перетаскивать в предварительные шаги вообще всё. Например, так:

Предварительные шаги
Открыть сайт: **https://www.example.com/.**
Шаги
Щелкнуть на кнопке «Войти»...

Что? Какая кнопка? Где мне ее искать? На рабочем столе?

Шаги должны быть независимыми. Если говорить про веб-сайт, я должна открыть новую вкладку в режиме инкогнито и там пройтись по всем шагам, и у меня все получится. Поэтому выкидывать ссылку на сайт в предварительные шаги не надо — она важна для выполнения теста.

А вот если я уже заранее зарегистрировалась, то хоть в новой вкладке, хоть в новом окне открою всё и пройдусь по шагам. Авторизация-то будет работать, если вы укажете, под кем входить. А регистрация прямого отношения к тесту не имеет.

Какие еще могут быть предварительные шаги? Посмотрим на примере той же Дадаты[30]. Тестируем функционал обработки файла. Он доступен только авторизованному пользователю — надо зарегистрироваться. И он не бесплатный — нужно пополнить баланс. И, конечно, у нас должен быть на руках файл для загрузки.

Регистрация на сайте, пополнение баланса и подготовка файлов — предварительные шаги, они не имеют прямого отношения к тесту загрузки файла — это так, подготовка. Как они будут выглядеть? Допустим, мы хотим обработать файл-образец (есть такой в системе):

Предварительные шаги

1. Зарегистрироваться (см. тест-кейс «Регистрация»).
2. Пополнить баланс (см. тест-кейс «Пополнение баланса»).
3. Скачать файл-образец (см. тест-кейс «Скачивание файла-образца»).

На что обратить внимание?

Писать нужно обезличенно

Повелительное наклонение неприятно читать: пойди, открой, сделай, нажми. Фи...

Превращаем в нейтральные глаголы: пойти, открыть, сделать, нажать...

Пишите обезличенно, а не повелительно! Будь то баг или тест-кейс

Писать нужно в едином стиле

Все предложения должны быть написаны в едином стиле, а то читаешь потом такой текст и недоумеваешь:

1. Регистрация.
2. Пополни баланс.
3. Скачать файл-образец.

Странно же, правда?
Приведем в порядок:

1. Регистрация.
2. Пополнение баланса.
3. Скачивание файла-образца.

Или:

1. Зарегистрироваться.
2. Пополнить баланс.
3. Скачать файл-образец.

Оба варианта имеют право на существование — тут уж кому как больше нравится: существительное или глагол.

Можно ссылаться на другие тест-кейсы

Так как предварительные шаги прямого отношения к тесту не имеют, то мы и не расписываем их подробно. Если надо уточнить, как выполнить действие, дайте ссылку на другой кейс:

> Зарегистрироваться с именем «Д'Артаньян» (см. тест-кейс «Регистрация»).

Только помните, зачем делается отсылка на другой тест: чтобы, если у нас что-то поменяется в том действии (например, в регистрации), мы изменили бы это в ОДНОМ месте, в ОДНОМ тесте, а не в 100500.

Поэтому не надо писать:

> Зарегистрироваться в системе: зайти по ссылке А, нажать кнопку «Регистрация» в правом верхнем углу сайта, ввести в поле «имя» такое-то значение...

Завтра название кнопки изменится — вы во всех кейсах будете исправлять? А зачем?

Но не доходя до маразма ツ

Вот у нас в Дадате студенты пишут тест-кейсы на загрузку и обработку файлов. Чтобы им было проще, первый тест-кейс — на обработку файла-образца — тренер сделал сам. Того файла, который система предоставляет для демонстрации своих возможностей.

Предварительные шаги выглядят так:

Предварительные шаги

1. Зарегистрироваться (см. тест-кейс «Регистрация»).
2. Пополнить баланс (см. тест-кейс «Пополнение баланса»).
3. Скачать файл-образец (см. тест-кейс «Скачивание файла-образца»).

А потом студент тестирует, скажем, обработку файла в формате CSV. Угадайте с трех раз, как выглядят его предварительные шаги? Правильно!

Предварительные шаги

1. Зарегистрироваться (см. тест-кейс «Регистрация»).
2. Пополнить баланс (см. тест-кейс «Пополнение баланса»).
3. Скачать файл «Клиенты» (см. тест-кейс «Скачивание файла»).

Вот и как я тут должна понять, что за файл мне надо скачать? В формате CSV? С одной строкой и одной колонкой, с 10000 колонок? С разным форматом дат рождения? С весом в 5 Мбайт? Какой? ЧТО именно тестируется?

Некоторые студенты учитывают этот момент и пишут так:

Скачать файл формата CSV (см. тест-кейс «Скачивание файла»)

Но тут возникает новый вопрос — откуда скачать? Из тест-линка, внутри которого написан тест? Из какого-то общего хранилища? И что это за тест-кейс такой магический на скачивание файла, на который идет отсылка? Это ведь явная копипаста из примера. Там написано «тест-кейс на скачивание», значит, и я также напишу!

Почему в моем примере написано «скачать»? Потому что файл-образец в системе уже есть! И если мы хотим его протестировать, нам надо именно скачать то, что

находится по ссылке «образец», а не какой-то свой прошлогодний файл в систему запихивать. Иначе какой смысл в этом тесте?

Отдельный тест-кейс на скачивание образца тоже сделан не просто так. Ведь нам надо убедиться в том, что по ссылке «образец» скачивается ровно то, что нам нужно. Что написано в ТЗ. Ведь в образце не какие-то абстрактные данные — они подобраны специальным образом, чтобы что-то показать, какие-то возможности системы.

Отдельный тест-кейс на скачивание образца:

- проверяет, что файл реально скачивается (а то будет epic fail);
- проверяет, что внутри файла лежат правильные данные.

А еще на него можно сослаться в предварительных шагах других тестов. Там, где нам неважно, какой именно файл грузить, — когда мы тестируем работу системы с разным исходным балансом (денег на обработку хватает/не хватает), исключение столбцов (в Дадате есть такой функционал, чтобы не обрабатывать лишнего) или что-то другое.

В этом случае нам неважно наполнение файла. Мы просто хотим загрузить точно работающий файл. И образец в этом случае идеален! Ведь если система не в состоянии обработать собственный образец — какое к ней может быть доверие? Так что тест на обработку образца идет первым в приоритете тестировщика.

А дальше мы уже исследуем, как реагирует система на разные форматы, разный вес, разное количество столбцов и колонок... И для этих тестов файлы придется готовить самостоятельно. Скачать то неоткуда!

Поэтому в предварительных шагах мы пишем о том, какой именно файл надо *подготовить*. Так и пишем: «Подготовить такой-то файл, см пример в аттаче»:

> Подготовить файл формата DOC с данными из файла-примера (см аттач «Пример.doc»).
>
> Подготовить файл с разными форматами дат рождения (см аттач «Даты рождения.xls»).
>
> Подготовить файл с картинкой внутри вместо текста (см аттач «Картинка.xls»).

Еще раз: не скачать! Подготовить! И никаких отсылок на мифический тест-кейс «Скачивание файла». Что это за тест-кейс? Что он проверят в рамках нашей системы? И зачем нам на каждый тест-кейс писать отдельный тест-кейс на подготовку файла? Просто, чтобы сослаться ради ссылки? Не надо.

Заметьте, как описан подготовительный шаг — мы *готовим* файл. Не скачиваем аттач, а готовим файл. И написано, что это за файл, — вдруг аттач испарится завтра, случайно удалим? Все равно будет понятно, какой именно файл надо готовить.

А еще аттач может устареть: изменили функционал системы, файлы в старом формате уже не грузятся. Но если описано, ЧТО это за файл, тестировщик сможет его обновить!

Вопрос «на подумать»

Почему при тестировании формата DOC, который система не умеет обрабатывать, мы готовим файл с данными из образца?

Выкидывайте текст ради текста

«Кратко, но емко!» — главное правило оформления текстов. Будь то баг-репорт, тест-кейс или письмо заказчику.

Текст ради текста всегда выкидываем. Сравните:

Зарегистрироваться (см. тест-кейс «Регистрация»).

Зарегистрироваться на сайте **https://www.example.com/** (см. тест-кейс «Регистрация»).

Что лучше? Лучше первый вариант, так как там меньше текста. У нас ведь все тесты на сайте **https://www.example.com/** — зачем тогда лишний раз писать ссылку? Тем более, что потом придется продублировать ее в основных шагах.

А если разработчик решит поменять интернет-адрес (URL) ссылки? Зачем нам вносить лишние правки? Когда надо поменять в 10 местах, всегда есть шанс хоть одно продолбать — а в итоге у нас будет неактуальная тестовая документация.

Мы потому и выносим регистрацию в предварительные шаги — чтобы не исправлять сотни кейсов, если что-то изменится: поправить в одном месте, в одном кейсе.

Ок, а если выбирать из таких вариантов, что будет лучше? Подумайте сами, прежде чем прочитать ответ:

1. Зарегистрироваться (см. тест-кейс «Регистрация»).
2. Зарегистрироваться с именем Ольга и email **xxx@gmail.com** (см. тест-кейс «Регистрация»).

Правильный ответ — все зависит от контекста. Если нам важно зарегистрироваться именно с таким именем (проверяем женские имена или имена с апострофом, или что-то еще) — это нужно указать в предварительном шаге с регистрацией.

А если нам неважно, будет email **xxx@gmail.com** или **olala@gmail.com** — зачем об этом писать? Если я умею регистрироваться, я как-нибудь справлюсь с придумыванием email. Если не умею — пойду в тест-кейс регистрации и пройду по нему.

Поэтому, если нам важен сам факт регистрации, будет лучше вариант 1. Если важны данные — вариант 2.

Предварительных шагов может и не быть — это нормально

Не надо высасывать их из пальца там, где они не нужны. Именно так и получаются тесты, в которых просто отсекли первые 2–3 шага и запихали в раздел «предварительные шаги» непонятно зачем:

Предварительные шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать на кнопку «Войти».

Шаги

Ввести логин такой-то, пароль сякой-то.

А должно быть так:

Шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать на кнопку «Войти».
3. Ввести логин такой-то, пароль сякой-то.

Итого

Предварительные шаги — это все то, что поможет нам пройти тест-кейс, но прямого отношения к текущему тесту не имеет. Например, регистрация в системе. Или покупка ингредиентов для шарлотки ;-)

Правила описания предварительных шагов:

1. **Писать обезличенно** — так приятнее читать, чем в повелительном наклонении.
2. **Писать в одном стиле** — а не «то глагол, то существительное», «то регистрация, то зарегистрироваться».
3. **Ссылаться на другие тесты можно** — в шагах не стоит (чтобы они были независимые), а тут можно. Но без маразма типа «скачать файл, см. тест-кейс такой-то», и отдельный тест-кейс на подготовку файла...
4. **Выкидывать лишнее нужно** — кратко, но емко! Копипасту убираем, лишний текст тоже.
5. **Предварительных шагов может и не быть — это нормально**. Не стоит высасывать их из пальца просто потому, что «они должны быть!»

Шаги

Шаги тест-кейса — это те действия, которые мне надо выполнить, чтобы получить ожидаемый результат. В шарлотке это «взбить яйца с сахаром, добавить муку...». В ПО — «открыть приложение, ввести в строку поиска такой-то запрос...».

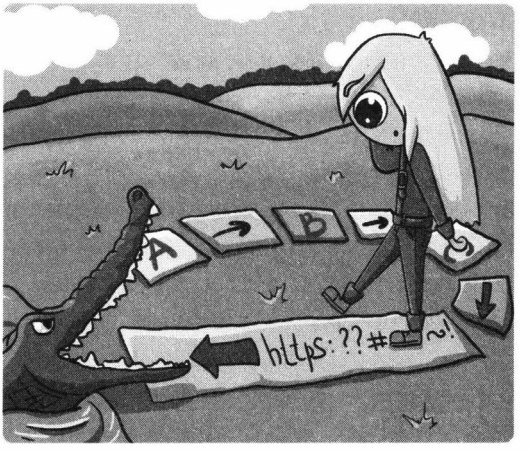
В остальном шаги похожи на предварительные — например, писать лучше обезличенно. Но есть одно важное различие.

Не надо ссылаться на другие тест-кейсы

Шаги должны быть независимыми. Вот я открываю тест-кейс и могу пройтись по шагам и получить результат. В таком подходе есть плюсы и минусы.

Плюсы

- **Нет дополнительных действий** — а то открываешь один тест, начинаешь выполнение, а на каждом шаге приходится открывать новую вкладку, проходить по другому тесту, потом возвращаться на прошлую страницу, вспоминать, где остановился, продолжать шаги, а тут снова отсылка к другому кейсу, и снова по кругу... Это еще хорошо, если отсылка на другой кейс будет гиперссылкой, а если нет, то еще и бегай, ищи по всему проекту. А уж если и название неуникальное!
- **Если другой кейс «сломался», наш останется жив.** Тест-кейс, на который идет отсылка, могли изменить или вообще удалить как неактуальный. А как вам тогда свой пройти? Если там первым шагом: «Сконструировать самолет, см. тест-кейс „Конструирование самолета“»? Если инструмент, в котором хранятся тест-кейсы, не показывает всех ссылок на текущий кейс, вы никогда не найдете все отсылки к нему. Даже если возьмете за правило скурпулезно записывать «на этот тест ссылаются раз, два, три», все равно кто-нибудь забудет проставить такую ссылку, и всё. При удалении теста какой-то не проактуализируют, и потом новичок его никогда не пройдет.



Независимые шаги позволяют избежать всех этих проблем, но привносят очевидные минусы.

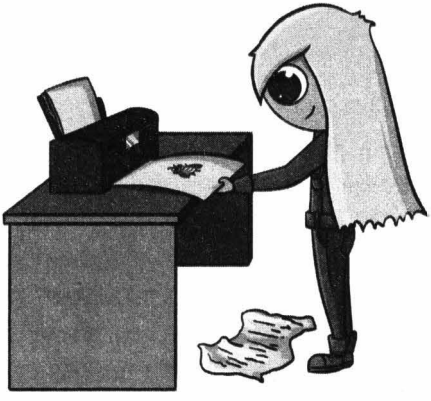
Минусы

- **Много копипасты** — так как мы не ссылаемся на другие тесты, то первые N шагов у нас из раза в раз повторяются. Попробуйте написать пять тест-кейсов на регистрацию с разными именами, которые обсуждались ранее (короткое, длинное, уменьшительно-ласкательное, унисекс...). Различие будет в одном шаге да иногда в результате. Всё.
- **Сложно поддерживать** — вытекает из первого минуса. Если что-то изменилось — скажем, URL главной страницы, надо пойти и обновить ВСЁ. Причем, чаще всего вручную, потому что #жизньболь. И ладно еще, если просто URL — делаешь поиск и автозамену, а если какой-то шаг изменился? В итоге тестировщик силился вспомнить, где шаг упоминается, вроде поправил все места, но что-то продолбал. А потом приходит джуниор, и ой — тест-кейс не выполняется.

На своих курсах я обычно даю задание студентам написать 3–5 тест-кейсов на один функционал. Чтобы они успели прочувствовать всю боль минусов, но не слишком в это закопались.

Конечно, копипасту можно сокращать, можно выносить общий множитель за скобку, делать страницы с общими инструкциями и т. д., и т. п. Но в таком случае проще сразу написать чек-лист, имхо. А вот где использовать тест-кейсы — поговорим далее. Но сначала я хочу сделать акцент еще на паре моментов, важных при описании шагов.

Текст ради текста удаляем



Применяйте правило минимальных чернил: если бы документ ушел на печать, то чем меньше чернил будет потрачено на печать, тем лучше. Чем меньше текста, точечек, буковок, знаков препинания — тем лучше.

И хотя мы пишем тест-кейсы так, чтобы их поняла даже обезьяна, совсем упарываться тоже не надо:

1. Запустить компьютер.
2. Найти на рабочем столе значок браузера Chrome (см. рис. Chrome).
3. Поставить курсор на верхнее поле, куда вводится URL (см. рис. URL).
4. Ввести текст: `https://www.example.com/`.
5. Нажать на кнопку «Войти».
6. Поставить курсор на поле «Имя».

7. Ввести: `Ольга`.
8. Поставить курсор на поле «Пароль».
9. Ввести: `1`.
10. Нажать «Войти».
11. ...

Гораздо лучше так:

1. Открыть страницу входа в систему: **https://www.example.com/#authorization_popup**.
2. Войти под тестовой учеткой (Ольга / 1).
3. ...

Да, первый тест-кейс выполнит даже ваша бабушка, но наша цель не в этом. Наша цель — чтобы кейс был максимально простым и понятным. Чтобы я, впервые увидев систему, могла выполнить кейс, не ища пять минут кнопку входа, потому что креативный разработчик поставил ее чуть левее середины страницы. Чтобы я, ничего не зная о бухгалтерии, могла загрузить сложный бухгалтерский отчет и проверить, что он выдает правильные данные,. Но учить меня включать компьютер будет излишне.

В тест-кейсах и так очень много текста. Поэтому там, где его можно сократить без потери смысла, — сокращайте. Причем, это правило действует всегда — мы будем упоминать его и при описании багов, и при тестировании документации.

Ссылок на PROD НЕ даем

PROD — PRODUCTION, это то, с чем работают реальные пользователи.

А внутри компании обычно есть свой тестовый стенд, а то и не один. Гиперссылки получаются разные, например:

- PROD — https://www.example.com/;
- TEST — http://www.test-example.com/.

Никогда нельзя проводить тестирование на PROD'е! Исключение составляет дымовой тест[31], проводящийся после обновления PROD-системы. Тестовый набор для этого создается отдельно и тщательно выверяется.

ВСЁ остальное тестирование проводится ТОЛЬКО на тестовом стенде. В описании тест-кейсов и багов должны быть ссылки только на тестовый сервер. Иначе попросим коллегу с другого проекта помочь нам с тестированием, а он пойдет на PROD и ... или сломает что-то, или испортит реальные данные.

Никогда не добавляйте человеческий фактор в свою работу: «Да, тут все ссылки на PROD, но в уме мы держим, что тестируем на TEST». Зачем

держать в уме? Просто давайте сразу правильные ссылки. Зуб даю, кто-нибудь да щелкнет случайно, а потом... Жди беды.

На PROD мы не тестируем,
чтобы не испортить реальные данные

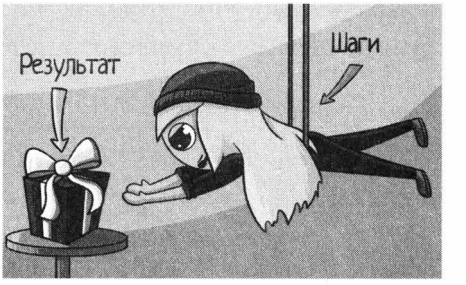
Конечно, если мы просто составляем тесты для своего портфолио, то у нас нет доступа к тестовым стендам. Мы в любом случае тестируем на PROD, в единственном доступном нам месте. И тут допустимо давать реальные ссылки, чтобы ваш тест-кейс можно было воспроизвести.

Но простому пользователю особых прав и не дают, разработчики-то не идиоты. Ну повводите вы разные слова в поиск, ну и что? Или создадите карточку клиента с именем «Тест», подумаешь. А вот зайти в админку, подхачить базу или сделать еще что-то опасное у вас не получится, поэтому для портфолио это разрешено.

Но и вы включайте мозг и не пытайте стереть чужие реальные данные. Если, допустим, мы проверяем систему типа википедии, не надо затирать нормальный полезный текст своими тестовыми данными.

Результат

Результат тест-кейса — это самое важное: то, что именно мы должны проверить. Любой тест направлен на проверку какого-либо функционала. Шаги, предварительные шаги — это все просто помощники в том, чтобы получить заветный результат.



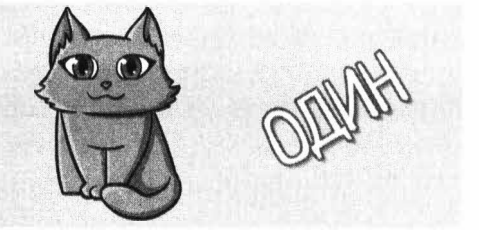
Что тут важно? Чтобы ваш результат понял любой человек. Вот я вижу систему впервые в жизни и должна пройтись по шагам, проверить результат и однозначно сказать: «работает» или «не работает». Четко, конкретно и по делу. Абстракции не допускаются.

Проблема в том, что обычно начинающие или студенты именно абстракцию и пишут. Ведь как новичок составляет тест? Запустил систему, выполнил функцию, посмотрел, как она работает, записал в ожидаемый результат. Профит! Но нет ☹

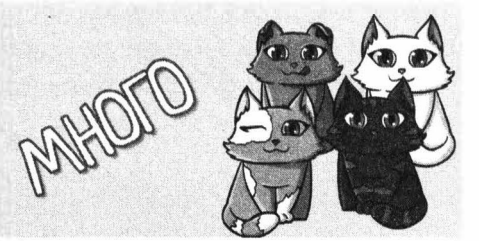
Такой результат быстро устаревает. К тому же сейчас система может работать неправильно (упс!), и если бездумно скопировать текущее поведение в ожидаемый результат, то баг вы не найдете. И любой, кто прогонит этот тест, его не найдет. А найдет пользователь в продакшене, что очень плохо.

Поэтому, когда пишем результат, вспоминаем о том, что мы вообще хотели проверить в кейсе. И описываем. Помня про главное правило: «кратко, но емко!».

Результат может быть один, или их будет много



Ожидаемый результат



Все варианты имеют право на существование. Важно лишь то, чтобы это читалось понятно. Выполнила шаг — сразу проверила. Или выполнила все шаги и получила результат. Только не пытайтесь запутать читателя, когда всё наоборот.

Один результат после всех шагов

Если результат один, то он идет после выполнения всех-всех-всех шагов. Вспомним рецепт приготовления шарлотки из *разд. «Предварительные шаги»*. Выполнили все 10 шагов? Получили шарлотку!

Один результат на тест: проверяем после выполнения всех шагов

Также и в ПО: выполняем все шаги, получаем результат.

Предварительные шаги

Зарегистрироваться на сайте с именем Ольга, логином ABC и паролем 1 (см. тест-кейс «Регистрация»).

Шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать на кнопку «Войти».
3. Ввести свои данные (логин: ABC, пароль: 1), нажать «Войти».

Ожидаемый результат

Открылась главная страница сайта.

В верхнем правом углу отображается приветствие: «Здравствуйте, Ольга»

Результат на каждый шаг

Можно писать ожидаемый результат на каждый шаг. Но тут важно подумать о тех, кто будет наш тест-кейс читать. Если вы пишете результат на каждый шаг — это должна быть табличка, а не так, что идут 1-2-3-4-5-6 шагов, а потом БАХ, в результате «открыта главная страница сайта» O_O

Если результат на каждый шаг — он пишется напротив него.

ПЛОХО

Шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать на кнопку «Зарегистрироваться».
3. Ввести имя `Ольга` и пароль `1`.
4. Нажать «Сохранить».

Ожидаемый результат

1. Открылась главная страница сайта.
2. Открылось окно регистрации.
3. Данные введены.
4. Редирект на главную страницу, после регистрации мы сразу входим в систему. В верхнем правом углу отображается приветствие: «Здравствуйте, Ольга»

Как люди обычно читают? Слева направо, сверху вниз. Так что я сначала выполнила 4 шага, а потом увидела ОР на первый... И как мне его теперь проверять? Все закрывать и повторять тест-кейс сначала? А если там не 4 простых шага будет, а 20 сложных? Это раздражает... Поэтому, если хотите, чтобы проверка была на каждый шаг, делайте сразу красиво — вот как-то так.

ХОРОШО

№	Шаги	Результат
1	Открыть сайт **https://www.example.com/**.	Открылась главная страница сайта

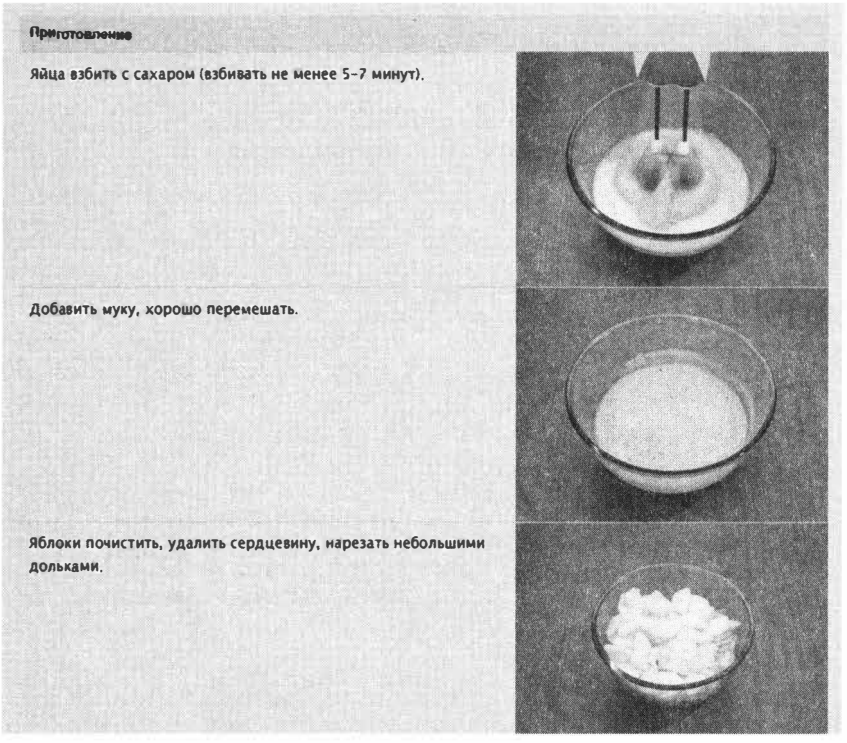
Окончание таблицы

№	Шаги	Результат
2	Нажать «Зарегистрироваться».	Открылось окно регистрации
3	Ввести имя Ольга и пароль 1.	Данные введены
4	Нажать «Сохранить»	Редирект на главную страницу сайта, сверху приветствие: «Здравствуйте, Ольга!»

Если вспомнить аналогию с рецептом, то в рецептах иногда так и делают — например, в рецепте шарлотки[32] на сайте Анастасии Скрипкиной (**https://www.say7.info/**). Открываешь любой рецепт и сразу видишь на каждый шаг ожидаемый результат в виде картинки.

А теперь представьте себе, если бы сначала шел рецепт, а после него — куча картинок. И вам бы пришлось тратить время на сопоставление, что вообще к чему относится. Так, шаг прочитал, ищешь картинку под него. Ну, вроде похоже. Возвращаемся к шагам. Стоп, а на каком шаге я остановился? Блин, надо вспоминать... О, вот на этом. Так, где под него картинка? Снова ищем...

При наличии альтернатив вернулись бы вы на сайт, который заставляет вас впустую тратить время? Вот то-то же. Думайте о людях, которые будут читать ваши кейсы. Они не телепаты, заранее не знают, что их поджидает после выполнения 10 шагов. И если результат не написан напротив каждого, то они будут ожидать один и общий.



Несколько результатов для разных проверок

Возможно, вы уже догадались о главном недостатке тест-кейсов — большом количестве копипасты. Если мы проводим несколько разных тестов на одно поле, то получаем десятки тестов с одними и теми же шагами, иногда даже с одним и тем же результатом. И если что-то изменится, надо исправлять их все...

В качестве одного из способов уменьшения копипасты используют объединение тест-кейсов и чек-листов. Это значит, что мы пишем вроде как тест-кейс, но начинаем засовывать в него несколько разных проверок. Так-то у нас «один тест-кейс = одна проверка», а здесь не совсем.

На самом деле это больше похоже на чек-лист с предварительными шагами: «как мне досюда дойти».

Регистрация с корректным именем

Шаги

1. Открыть сайт **https://www.example.com/**.
2. Нажать «Зарегистрироваться».
3. Ввести имя из таблицы, приведенной ниже, и пароль 1.
4. Нажать «Сохранить».

	Вводимое значение	Ожидаемый результат
1.	Свое имя (например, Ольга)	Успешная регистрация
2.	Короткое имя (Ева)	Успешная регистрация
3.	Длинное имя (Аполлинария)	Успешная регистрация
4.	Составное имя (...)	Успешная регистрация
5.	...	...

Ну и опять же, помним про «кратко, но емко» — если видим копипасту, избавляемся от нее. Пишем один результат на все пункты проверки.

Кратко, но емко: упрощаем информацию

Понимаете, стену текста никто читать не будет. Ну разве что джуниор, который только что пришел на работу и очень, ну просто очень боится ошибиться. Тогда он будет внимательно вчитываться в любую инструкцию, стараясь ничего не упустить. Однако тут вмешивается человеческий фактор — если пытаться проверить кучу текста, есть большой шанс все же упустить что-то. И по закону подлости куда закрадется баг? Верно, в непроверенный кусочек.

Поэтому стараемся визуально упростить информацию. Если проверок много — например, мы строим какой-то отчет, то можно:

1. Сделать эталонный файл, с которым можно сравнить таблицу. Обязательно разметить, на что обратить внимание и почему тут именно такая цифра.

2. Сделать табличку «Было-стало-комментарий», которая визуально воспринимается проще, чем стена текста.
3. Просто расписать результат, но разбив на абзацы и добавив заголовки.

Эталонный файл: даем сравнить

Если вы пишете результат, и пишете, и пишете... А он все никак не закончится — остановитесь! Никто не будет все это читать. Лучше приложите некий эталон, с которым можно сверить результат. Ведь когда у нас получается стена текста? При тестировании большого отчета, в котором много колоночек, например. Вложите «правильный отчет»! И кратенько укажите, на что обратить внимание.

Тестировщик — не тупая мартышка, чтобы побуквенно сверять с эталоном, объясните ему, что именно он проверяет

Это очень важно — не ограничиваться вложением эталона! Все равно надо подписать, на что конкретно мне обращать внимание. Это может быть текст отдельно, это может быть раскрашенный эталонный файл с комментариями внутри.

Главное — не делать из тестировщика тупую мартышку, которую заставляют бездумно сверять две таблички. «Не знаю, почему тут именно 2050, но в моем эталоне так же, значит, работает!». Это способен сделать и робот, да и сам тестировщик может в экселе просто сравнивать по ячейкам ожидание и реальность. Но знания системы ему не добавится.

Да и что будет, если эталонный файл устарел или просто продолбался? Можно сколь угодно долго бить себя пяткой в грудь и говорить, что «со мной такого точно не случится. Если я пишу „см. аттач", я его вложу», но... Все бывает!

Поэтому я не принимаю у студентов такие ожидаемые результаты и пишу здесь как раз о том, что файл может потеряться. Обычно это вызывает возмущение: «Но сейчас-то я его вложил! Так что нормально все!». Сейчас вложил, завтра забудешь. Или коллега его случайно удалит, вручную или скриптом каким. Что делать-то будете?

К тому же для меня «см. аттач» сразу показывает, что студент просто взял текущее поведение системы за эталон и вложил в результат, даже не пытаясь понять, почему система отработала именно так. То есть проверки как таковой и не было, и если система работает неверно, «да и хрен с ней, задание я сделал». Что меня совсем не устраивает. Хочу включения мозга!

И ладно бы вы просто забыли вложить файл. Допустим, вложили. Допустим, с ним даже можно сверяться, и все идет хорошо, тестировщики проверяют по нему работоспособность системы. Но потом система обновляется, и тест становится неактуальным. Документацию обновили, про тест-кейс забыли.

И вот новый тестировщик открывает тест «сравнить с приложенным файлом». Файлы различаются. Джуниор разведет панику и поставит баг, а потом выяснится, что надо просто поправить эталон. Опытный пойдет уточнять, где неправда. Хотя это все равно потраченное время, которого можно было бы избежать при конкретном описании проверки.

Помните — если вы вкладываете аттач, он идет как дополнение к проверке! Но не заменяет описания. Всегда старайтесь кратенько рассказать, что именно мне надо проверить. Представьте, что вы объясняете новичку в курилке, о чем именно тест. Вряд ли вы по памяти вспомните все 100500 циферок из отчета, скорее, опишете логику построения. Вот об этом и пишите!

Пишем конкретно, без абстрактных «бла-бла-бла»

И пишите конкретно. Что в шагах, что в результате абстракция не нужна. Вы выполняете конкретные действия, вводите конкретные данные, получаете конкретный результат. Избегайте абстракции, это главный бич всех тестов новичков.

Абстракция — это когда неважно, какие данные были на входе. Когда ваш результат можно смело копипастить еще в сотни кейсов. Скажем, «калькулятор работает корректно». Это — абстракция, мне она ни о чем не говорит. Подойдет и для теста на сложение, и на вычитание, и на умножение, и на деление... Пишите конкретно!

Давайте посмотрим на примерах, что такое хорошо и что такое плохо. А то как можно абстрактно рассуждать о пользе конкретики?

Логи

У моей коллеги Ольги Алифановой был такой случай на работе: «Сравнить лог с приложенным файлом, убедиться, что нет отличий», — весь ожидаемый результат. И приложен лог, который устарел на три версии.

- На что смотреть?
- Чего в логе не должно быть?
- Что должно?

В итоге убито полчаса времени. А написали бы русским языком, и нет проблемы. Да, приложенный лог устарел, но мне все равно ясно, что происходит и пройден ли кейс.

Это касается всех аттачей в ожидаемом результате — они нужны и важны, так как помогают провести быструю проверку: сравнил, и всё. Но учтите, что они быстро устаревают, а проапдейтить тест-кейс часто забывают. Поэтому мне должно быть понятно, на что именно в этом аттаче смотреть.

Дадата

Студенты тестируют Дадату[33]. Напоминаю, эта система умеет стандартизировать данные:

- приводить их к общему виду. Один и тот же адрес, например, можно написать сотней разных способов: от «Турчанинов, 6, 2» до «119034, г. Москва, пер Турчанинов, дом 6, корпус 2, бизнес-центр «Крымский Мост», звонить на проходной». Разные разделители, то «пер. Турчанинов», то «Турчанинов пер.», то, типа, с точкой, то без, ну и так далее. На выходе из Дадаты мы получаем единый вид записи, максимально гранулярный;
- исправлять опечатки;
- ставить код качества: насколько система уверена в «хорошести» информации. Можно ли делать рассылку по ней, или стоит отправить на ручной разбор человеку.

В системе есть файл-образец, в котором собраны наглядные примеры «что я умею». При этом собраны не просто так, от балды, они все разные. Показывают способности Дадаты. Но читать стены текста никто не любит, поэтому примеров мало. Да, можно сделать образец на тысячи строк, «и это умеем, и то, и вон то», но кому он нужен? Посмотрим на ФИО из образца:

1. Иванов Сергей Владимирович
2. Федотов Алексей
3. Ольга Павловна Ященко

Чем они различаются между собой? Почему именно такие?

Студенты обычно не заморачиваются этим и пишут вместо результата некую абстракцию: «Скачан файл, в нем обработанные данные». Хм, и что мне скажет такой результат? Иногда пишут подробнее: «Данные корректно обработались». Если я загрузила файл с ФИО «Киселева Ольга», а на выходе получила «Иванов Иван» — это нормально? Нет? А как тогда нормально? Что конкретно значит «корректно обработалось»?

Нужна конкретика. Что именно мне проверить? Вот тут придется включить мозг и подумать: а реально, ***что именно*** я проверяю? Я проверяю стандартизацию. Причем данных из образца. Образец нужен, чтобы показать, что именно система умеет, — так чем же различаются эти данные?

Вот, например: Иванов и Федотов. Да ничем не различаются, просто у Федотова отчества нет. Но что это значит? Если понимания работы системы нет, то начинается копипаста: «ФИО разделено на отдельные поля, просклоняли: родительный, дательный, творительный падеж». Авось, если текста много будет, то никто не заметит, что там одно и то же.

Если мы копипастим один и тот же результат для разных данных — это толстый намек, что написана абстракция. Раз падежи определяются для любых ФИО, то так ли это важно? Наверное, нет, можно вынести это как общий множитель «за скобку», а дальше пусть в эталоне смотрят.

Но что же различается у данных? Код качества тоже одинаковый, везде хороший, значит, и о нем не писать? А что тогда писать? А что вообще эта система делает и зачем? Выходим на главную страницу, внимательно ее изучаем и понимаем, что данные (ФИО, адрес и прочая) используются для рассылки. Если код качества хороший, то можно писать письмо. Если требует ручной проверки, то автоматическую рассылку делать нельзя, надо выверить вначале.

Ради этого я и обрабатываю данные. У меня есть файл, в нем вся моя база клиентов, скажем, 1000 клиентов. Я обработала файл, я могу все имена оттуда взять и сделать по ним рассылку?

> Здравствуйте, Жопа!
> Попробуйте наш новый крем от морщин...
> Привет, Пошли на хрен со своими именами!
> Попробуйте наш новый крем от морщин...

Ну и так далее — а зачем вообще было обрабатывать имена в Дадате? Если они корректно обработались, получается по статусам — корректный. Если код сомнительный, письмо лучше не слать, выверить данные вручную.

Соответственно, система берет 1000 ФИО и говорит, что из них 900 корректные, по ним можно рассылку делать. А 100 проверить вручную.

Фишка в уменьшении ручного труда — надо проверить 100 ФИО, не 1000.

Если система ставит плохой код качества, когда написано не ФИО, а ИОФ — в чем ее фишка? Она говорит «плохо» на хорошие данные, просто идущие в другом порядке. Если система ставит плохой код качества, когда не указано отчество — плохо.

Отчество мало где пишут, но рассылке это не мешает. Если мне без отчества поставят «на ручной разбор» всё, а у меня магазин, и 950 ФИО пишут только имя или ФИ, то смысла работать с Дадатой — 0. Она мне все мои данные оставит на ручной разбор. Ноль экономии.

И разные ФИО в образце, чтобы показать, как система работает с РАЗНЫМИ данными, а не просто «она все делает корректно», как бы пафосно и многословно вы ни написали эту абстрактную фразу.

Так что, прикладывая эталон, пишем кратенько различия ФИО:

- Иванов — полное ФИО разбирает, склоняет, рассылку делать можно;
- Федотов — аналогично, отсутствие отчества не влияет на хорошесть кода качества.

Видите? Мы вроде написали копипасту (работает-то аналогично), но указали на различие.

Вопрос на «подумать»

А теперь сами: чем отличается Ященко от остальных? У нее тоже определены падежи и хороший код качества. Напомню все три ФИО:

1. Иванов Сергей Владимирович
2. Федотов Алексей
3. Ольга Павловна Ященко

Хороший лайфхак для самопроверки, если вы еще не работаете. Представьте, что написали тест-кейс на обработку файла-образца в Дадате, вставили в портфолио и показываете работодателям:

Вот возможному работодателю оно надо — эталон изучать?

нет

А стену текста в результате?

нет

А что он поймет, НЕ открывая Дадату, из фразы «система отработала хорошо»?

ничего

А куда он выкинет резюме, если в тест-кейсе на обработку образца (который показывает, как работает система) будет просто отсыл к другому файлу в результате?

ну вы поняли ;-)

Проверяем, а не доверяем!

Какие приложения обычно выбирают для создания портфолио:

- форум;
- интернет-магазин;
- сайт авиакомпании;
- ...

На какой функционал можно написать тест-кейсы? Например, на поиск! И вот я получаю от студента примерно такой тест-кейс:

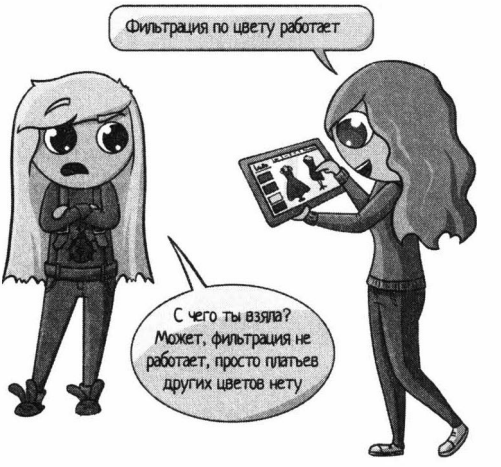
1. Поиск по названию темы

Шаги

1. Зайти на форум **https://www.example.com/**
2. Ввести в строку поиска слово `корова`.

Ожидаемый результат

Видим статью со словом «корова» в названии.



Вместо названия темы тут может быть цвет платьюшка или название города, куда мы хотим купить авиабилет.

Или вот такой тест:

2. Поиск статей за последние сутки

Шаги

1. Зайти в раздел статей **https://www.example.com/**.
2. Поставить галочку «отображать только статьи за последние сутки».

Ожидаемый результат

Остались только статьи, созданные за последние сутки.

Как вы думаете, можно ли доверять такому тест-кейсу? Проверяет ли он обещанный функционал?

Нет! Не проверяет. Мы просто выполняем какое-то действие и смотрим на результат. И если «ну, вроде, подошло», считаем, что функционал работает. Но ведь это неправильно! Это так пользователь может сайт проверить, а тестировщик продумывает проверку полностью.

Как понять, что после фильтра «за последние сутки» остались и правда статьи за последние сутки? И что именно значит «за последние сутки»? Откуда начинается отсчет? Минус 24 часа или от полуночи?

Чтобы проверить функционал, нам надо подготовить статьи за разные дату и время публикации, а потом поставить галочку и убедиться, что в фильтр попали только нужные. Выглядеть это будет примерно вот так:

3. Поиск статей за последние сутки

Предварительные шаги

Подготовить статьи за разное время публикации относительно текущей даты. Скажем, если сейчас 23.01.2018 10:00, делаем статьи:

1. 21.01.2018 10:00
2. 22.01.2018 09:59
3. 22.01.2018 10:00
4. 22.01.2018 18:00
5. 23.01.2018 08:00
6. 23.01.2018 10:00
7. 23.01.2018 10:05

Шаги

1. Зайти в раздел статей **https://www.example.com/**.
2. Поставить галочку «отображать только статьи за последние сутки»

Ожидаемый результат

Остались только статьи 3–6, созданные за последние сутки. 1 и 2 были созданы ранее, в фильтр не попали. Статья 7 создана позднее (нереальный кейс, но мало ли), также не попала в фильтр.

Студентам я привожу в пример историю про яблоко и унитаз, чтобы показать, как правильно. Ведь если я просто отредактирую их тест, то у них в головах ничего не останется. Поэтому я показываю на абстрактном примере, а они уже по аналогии правят свой.

Яблоко и унитаз

Предварительные шаги: создать товар «яблоко» с ярлыком «фрукты» и товар «унитаз» с ярлыком «ванная_комната».

Шаги: ... открыть список всех товаров с ярлыком «фрукты».

ОР: в список попало яблоко, но не попал унитаз.

Вот это — проверка того, что ярлыки действительно работают. А не просто «Открыть все товары с ярлыками А, Б, В». Это обычный пользователь просто открывает «что дают», а тестировщик готовит специальные данные, чтобы проверить, что функционал реально работает.

В тестах 1 и 2 проверки никакой нет. В них просто открывается страничка и предлагается поверить на слово, что всё работает как должно. Это так обычный пользователь смотрит, но наша задача — протестировать функционал, а не верить ему на слово.

Пользователь просто вводит данные и доверяет результату. Тестировщик же заранее готовит данные, чтобы убедиться в том, что функционал и правда работает

Что еще?

Нумерация (уникальный ID)

Нумерация — чтобы удобно было ссылаться на тест-кейс. Не «Регистрация через Твиттер», а «ТК01. Регистрация через Твиттер» (ТК — тест-кейс) или просто: «1. Регистрация через Твиттер». Как хотите, главное — чтобы можно было назвать коллеге номер, и он его легко нашел. Соответственно, номера не должны пересекаться.

Что делать, чтобы не переходить в большие цифры типа 2087 и так далее? Можно к названию добавить букву-другую по названию функционала:

- регистрация — Р1, Р2, Р3...;
- авторизация — А1, А2, А3...;
- загрузка файла — З1, З2, З3... Или Ф1, Ф2, Ф3...

Если сама спецификация нумеруется, можно писать идентификатор в формате * * * *:####, где:

- * * * * — номер спека;
- #### — номер тест-кейса.

Например: S1236:T4563, где S — значит, спек, 1236 — номер спека, T — значит, тест, 4563 — номер теста.

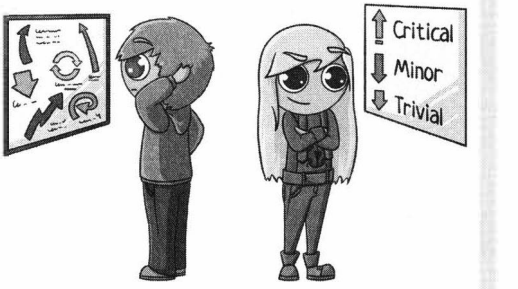
Удачная архитектура не только в коде хороша!

Приоритет

Приоритет помогает понять, в каком порядке выполнять тест-кейсы. Сначала — самые важные, потом все остальное, если время останется. Лучше не заморачиваться со сложной структурой приоритетов и оставить всего три:

- высокий;
- средний;
- низкий.

Приоритеты тест-кейсов должны быть понятными. Лучше меньше, но лучше!



История редактирования

Честно говоря, я не вижу особого смысла вести историю редактирования, по крайней мере, в том виде, в котором она у Савина, — когда вы в ворде сами пишете, кто когда файл редактировал. Выглядит это примерно вот так (табл. 2.1).

Таблица 2.1. *История изменений*

Дата	Автор	Содержание изменения
01.02.2021	Назина Ольга	Новый тест-кейс
02.02.2021	Иванов Иван	Добавил проверку пограничных значений
05.08.2021	Сидоров Василий	Выкинул лишнее из шагов

С одной стороны, очень удобно. Видим, кто, когда и зачем менял тест-кейс. Но... Давайте посмотрим правде в глаза — этот раздел устаревает быстрее всего. Его просто забывают пополнять. Так и будет висеть: «В 2021 году Назина создала, Иванов подправил», а кейс потом переписывали раз 10, но не подписались.

Я считаю, что это довольно тупая работа. Если хотите хранить историю — используйте инструменты, которые это умеют делать. Ну XXI век на дворе, вы чего? Любая вики-система, специнструменты под тест-кейсы... Да бог с ними, простые гуглодоки хранят всю историю! И даже если так сильно нравится Word — ну так храните его в Dropbox, на Яндекс-диске или Гугл-драйве. И всё! Вся история сохранена, можно откатиться на любую версию, посмотреть, кто и какие правки вносил.

Нет, конечно, можно сказать «Просто нужна дисциплина! Будем штрафовать, если не запишутся», но кому это надо? Когда работу может сделать компьютер, пусть её делает компьютер, освободите человека для более приятной работы!

Стандартные ошибки при оформлении тест-кейсов

Читать теорию — одно, делать на практике — другое. Обычно в теории все понятно, а на практике получаем примерно такой кейс (все совпадения случайны, тест-кейс написан как агрегация различных ошибок).

Легенда по ссылкам:

- PROD — https://www.example.com/;
- TEST — **http://www.test-example.com/**, на сайте висит двойная авторизация, чтобы лишние люди не прошли[34].

Тест-кейс № 01. Создание жильца

Шаги

1. Зайди на сайт **https://www.example.com/**.
2. Нажми на кнопку «Войти» в правом верхнем углу экрана.
3. Залогинься с правами администратора.
4. Перейди на вкладку «Жильцы».
5. Нажми на кнопку «Создать карточку жильца».
6. Введи корректные ФИО — например: `Иванов Иван Иванович` и сохрани карточку.

Ожидаемый результат: карточка создана.

Попробуйте, не заглядывая в ответ, самостоятельно найти проблемные зоны в этом тест-кейсе. А потом проверьте себя ;-)

Разберем ошибки кейса 01.

○ **Абстрактное название.**
На первый взгляд, название хорошее, короткое и понятное — мы ведь, правда, создаем жильца. Но! Если мы теперь создадим еще пяток тест-кейсов на ввод некорректных ФИО, то у них будет точно такое же название.
В итоге новый тестировщик, получив задание проверить кейс «Создание жильца», обнаружит в системе два десятка проверок с таким названием и впадет в ступор, какой выбирать?
Всегда помните про «кратко, но емко». По названию тест-кейса тестировщик, знающий проект, должен понять, что надо делать, не заглядывая в шаги. Так что дополняем название: «Создание жильца без отчества», «Создание жильца, цифры в поле „Имя“» и т. д.

○ **Повелительное наклонение.**
Чтобы коллегам было приятнее работать с тест-кейсами, лучше делать их описание обезличенным: «Выполнить», «Загрузить»... Неприятно же читать такое: «Пойди», Сделай»...

○ **PROD.**
В приведенном примере идет ссылка на PROD (**https://www.example.com/**).
А PROD мы не тестируем!

○ **Нет гиперссылки на сайт.**
Интернет-адрес (URL) написан, но он не «кликабельный». Его придется выделить, скопировать, открыть новую страницу, вставить... Гораздо лучше было бы просто нажать на него!

○ **Слишком детализировано.**
Шаг «Нажми на кнопку „Войти“ в правом верхнем углу экрана» содержит много подробностей про пользовательский интерфейс. Если кнопка в новой версии программы переедет в другое место, то придется вносить исправление и в тест-кейс. Чем меньше в документации зависимости от пользовательского интерфейса (UI, User Interface), тем лучше.
Перепишем этот шаг так: Войти под учетной записью администратора (admin/1).
Описание шага не стало менее понятным, но мы избавились от привязки к интерфейсу. Если вместо кнопки сделают ссылку, или человек просто <Enter> нажмет, то суть шага не изменится: мы же в этом кейсе не логин проверяем, а создание жильца.

○ **Нет нужной информации — непонятно, как авторизоваться.**

Есть шаг «Залогинься с правами администратора» — отлично, но как это сделать? Дойдя до этого шага, я пойду искать кого-нибудь, кто в курсе, есть ли тестовый пользователь с такими правами и какие у него логин и пароль.

Если такой пользователь существует (или создается каждый раз для тестирования) — его логин и пароль «статические» (не меняются), и их всегда надо прописывать, чтобы не возникало дополнительных вопросов.

Тест-кейсы составляются тогда, когда нужно, чтобы любой человек со стороны, не знающий проекта, мог присоединиться и помочь выполнить тест-кейсы. Не задавая коллегам при этом дополнительные вопросы.

Тест-кейс я должна уметь выполнять, впервые увидев проект, поэтому там должна быть ВСЯ нужная мне информация.

○ **Нет описания проверки.**

«Карточка создана» — кратко, но не емко. Не имея знаний о проекте, тестировщик может только предполагать, что включает в себя этот пункт.

Достаточно ли того, что карточка закрылась без ошибок? Или она должна теперь отображаться в списке карточек? А сколько в системе таких списков? Должна ли система отображать введенные данные, если открыть карточку на просмотр? Что конкретно нужно проверять?

Поправим этот тест-кейс с учетом всех замечаний. Вот что получилось:

Тест-кейс № 02. Создание жильца с корректными ФИО

Шаги

1. Зайти на сайт **http://www.test-example.com/**.
2. Войти под учетной записью администратора (логин: admin, пароль: 1).
3. Перейти на вкладку «Жильцы».
4. Нажать на кнопку «Создать карточку жильца».
5. Ввести корректные ФИО — например: `Иванов Иван Иванович`.
6. Нажать на кнопку «Сохранить».

Ожидаемый результат

1. Окно с информацией о жильце закрывается, и отображается общий список, в котором присутствует новая карточка.
2. Эту карточку можно открыть.
3. В открытой карточке отображаются введенные данные, то есть в поле ФИО указано «Иванов Иван Иванович».

Уже хорошо, но можно ли еще улучшить этот тест-кейс?

Сейчас снова попробуйте, не заглядывая в ответ, найти проблемные зоны и в этом тест-кейсе. А потом проверьте себя!

Итак, ошибки кейса 02:

○ **Абстрактное название.**

Слова «корректный», «правильный» и т. д. в названии тест-кейса такой же маркер, как «ошибка» в названии бага. Таких слов надо избегать.

Позитивных проверок можно придумать хоть сто. Но чем-то они будут различаться. «Создание жильца, у которого нет отчества», — это тоже кейс с корректным ФИО. Только из такого названия сразу ясно, про что кейс.

Поэтому забудьте про слова «корректный», «некорректный» и т. п., пытайтесь писать понятнее. И всегда помните принцип «кратко, но емко». А разделение кейсов на смысловые группы (позитивные тесты, негативные тесты, тесты на особые случаи) сделайте в системе управления тест-кейсами через флаги или отдельные наборы тестов.

○ **Нет нужной информации.**

Есть шаг «Зайти на сайт http://www.test-example.com/».

Ок, я открываю этот сайт, а там двойная авторизация, то есть мне выпадает окошко «введите логин/пароль». Как мне туда попасть?

Никак! Идти и узнавать логин/пароль. А зачем, если это легко было исправить указанием логина/пароля в скобках или ссылкой на страницу со всеми логинами и паролями (они все же могут меняться, и лучше менять в одном месте)?

Исправленная версия тест-кейса:

Тест-кейс № 03. Создание жильца с полным ФИО

Шаги

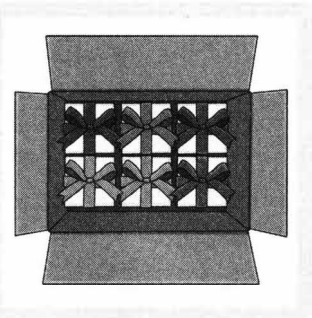
1. Зайти на сайт http://www.test- example.com/ (логин: test, пароль: test).
2. Войти под учетной записью администратора (логин: admin, пароль: 1).
3. Перейти на вкладку «Жильцы».
4. Нажать на кнопку «Создать карточку жильца».
5. Ввести полное ФИО — например: `Иванов Иван Иванович`.
6. Нажать на кнопку «Сохранить».

Ожидаемый результат

1. Окно с информацией о жильце закрывается, и отображается общий список, в котором присутствует новая карточка.
2. Эту карточку можно открыть.
3. В открытой карточке отображаются введенные данные, то есть в поле ФИО указано «Иванов Иван Иванович».

Ну вот, так уже получше будет!

Набор тест-кейсов



Test suite: набор тест-кейсов

Запомните: набор тест-кейсов — это НЕ тест-план!

Это именно набор тест-кейсов, по-умному он также называется TestSuite. А тест-план — это более сложная сущность, о которой мы поговорим в *главе 8*. В тест-план входит не только набор тест-кейсов, но и решение о том, какой функционал мы тестируем, а какой нет, какие виды тестирования проводим и т. д.

Просто не путайте эти понятия. И если вас, новичка-джуниора, просят составить тест-план, уточните, что имеется в виду. Возможно, как раз TestSuite, то есть просто набор тестов.

Особенности тест-кейсов

○ **В них максимум информации.**
Их легко можно отдать новичку, и он проведет проверки, не отвлекая вас вопросами «А это что? А это как сделать?» и так далее. Все просто и понятно — жми туда, жми сюда, получай результат.

○ **Они независимые.**
Хорошие тест-кейсы не зависят друг от друга, поэтому можно легко редактировать один, не боясь, что развалятся все остальные.

○ **Они небольшие.**
Один тест-кейс — одна проверка. Иначе это уже получается совмещение тест-кейса и чек-листа.

Когда применять тест-кейсы?

Тест-кейсы нужны, когда у нас:

- ○ жизненно важные системы, ошибка в которых может привести к гибели людей (авиастроение, медицина, ПО для атомных станций). Здесь надо тестировать очень аккуратно и тщательно;
- ○ сложная система или сложная часть системы. Чтобы каждый раз не вспоминать «а как мне это сделать?», лучше написать тест-кейс.
- ○ в команде постоянно появляются новые люди (все джуниоры проходят через этот проект).

Тест-кейсы не нужны, когда у нас:

- простые системы (веб-сайты одностраничники, мобильные приложения и т. п.);
- ситуации, когда в команде всего один или два тестировщика, знающие свой продукт. Время, потраченное на создание и поддержку тест-кейсов, никогда не окупится.

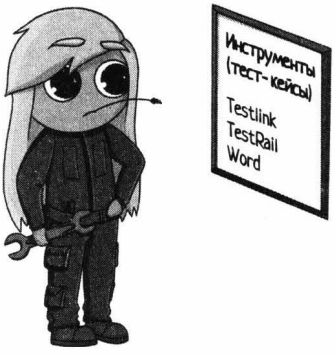
Познакомьтесь со своей системой и потом уже решайте, что подходит именно для нее: творческие чек-листы, формальные тест-кейсы или микс из этих подходов.

Так как тест-кейсы очень сложно поддерживать, то чаще используют чек-листы или комбинацию «чек-листы & тест-кейсы».

В последнем случае большинство проверок пишут в виде чек-листов, а особо сложные (пойди туда, не знаю куда, принеси то, не знаю что, кувыркнись три раза и громко крикни: «ДЕДЛАЙН!» — только тогда формочка и откроется) уже в виде тест-кейсов, чтобы каждый раз не вспоминать, как этот хитрый сценарий работает.

Инструменты для составления тест-кейсов

- Специализированные:
  - ✧ Testlink — бесплатный;
  - ✧ TestRail — платный, но есть trial-версия на месяц;
  - ✧ TestIT — платный, но в нем есть переиспользование шагов (меняешь в 1 месте, исправляются все!).



- Но часто хватает того, что всегда под рукой:
  - Word;
  - Notepad;
  - Гуглодоки;
  - Confluence;
  - ...

Лично я в свое время успела поработать с Testlink, но это было очень уныло. Там ведь надо заполнять все эти поля, расписывать все подробно: вот предусловия, вот шаги, вот результат, еще меток расставить, то, се... А на том моем проекте это не было нужно, я была там единственным тестировщиком от момента его зарождения и до закрытия. Но это не проблема инструмента, это проблема выбора «тест-кейсы VS чек-листы».

TestRail я вижу на курсах, он посимпатичнее смотрится, но готова ли компания за него платить? Я так скажу: если у вас кейсов пока мало, лучше брать бесплатные инструменты. Хотите специально под тест-кейсы? Берите Testlink. Еще не определились, чего хотите? Тогда забейте и используйте Word + Dropbox (или Яндекс-диск, или другое хранилище, куда все имеют доступ и где есть версионность), или гуглодоки. Дешево и сердито.

У меня на одном из проектов всё хранилось в вики-системе:

- документация;
- описание автотестов (фактически чек-листы);
- всякие HOWTO (фактически тест-кейсы, потому что как раз подробно поясняют, как выполнить то или иное действие).

Так что не торопитесь сразу влезть во все самое модное. Попробуйте на гуглодоках и подумайте, чего именно вам не хватает. Так и выберете свой инструмент!

А Testlink и Confluence вы можете «пощупать» на нашем портале[35]. Можно и самим попробовать, но Testlink нужно разворачивать, а Confluence платный.

Чек-листы

Что такое чек-лист?

Чек-лист — это список проверок. Человек с улицы их может и не понять, но это нормально. Главное, чтобы понимал тестировщик системы. Это просто напоминалка: «не забудь проверить то и то».

Наверняка вы уже знакомы с чек-листами и так или иначе применяете их в реальной жизни. Так, например, мы составляем чек-листы покупок: на

рынке купить арбуз, помидоры, огурцы, в магазине — молоко, яйца и творог.

Это не тест-кейс, так как нет подробного описания: как пройти в магазин от дома, как выглядит арбуз, какого цвета, размера и прочая. Согласитесь, такие подробности в бытовухе излишни ;-) Нам просто нужна напоминалка «не забудь это и то», все остальное мы и так знаем.

Другой пример чек-листа — когда мы едем в отпуск и составляем список вещей: паспорт, ноутбук, зарядка, купальник... Чек-лист? Чек-лист!

Чек-листы в обычной жизни

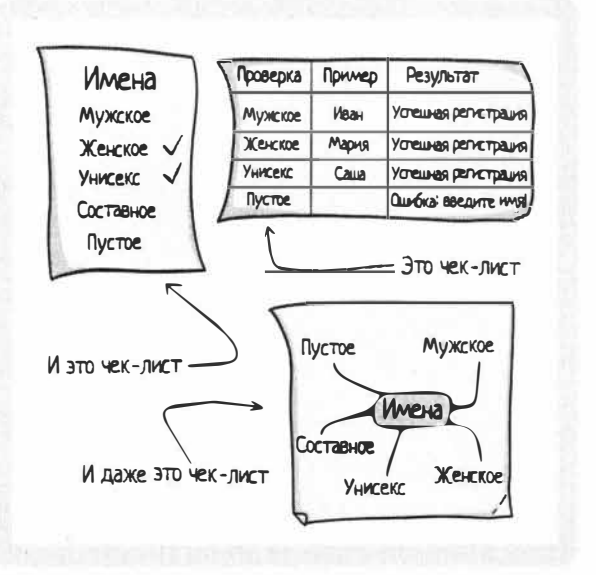
Задачка «на подумать»

А какие чек-листы составляете вы?

Аналогично в тестировании. Главная задача чек-листа — напомнить «не продолбать вот эту проверку». А вот как он будет выглядеть — дело десятое. Это может быть:

- простой список;
- две, три, пять, десять колонок;
- майнд-карта.

В общем, всё, что угодно, лишь бы вы и ваши коллеги понимали, что тут написано, и могли пройти по чек-листу.



Как оформлять чек-лист?

Лично я предпочитаю такую структуру:

- проверка;
- пример;
- результат.

Получается табличка на три колонки:

Проверка	Пример	Результат
Мужское	Иван	Успешная регистрация
Женское	Мария	Успешная регистрация
Унисекс	Саша	Успешная регистрация
...	...	...
Пустое		Ошибка: введите имя!

Давайте рассмотрим каждый пункт по отдельности — буду защищать свою структуру!

Как составлять описание проверки?

Думаю, вам уже знаком этот принцип: «кратко, но емко!» Описание проверки — это самая важная часть чек-листа, потому что присутствует всегда. Бывают чек-листы без результата, без примеров. Но «что мне надо протестировать» должно быть.

Описание может быть кратеньким — чисто напомнить, что не забыть. Например, какие форматы файлов проверить:

- текстовый;
- XLS;
- XLSX;
- DB;
- DLL;
- EXE;
- картинка;
- XML;
- HTML, CSS;
- архив;
- серверные файлы (PHP, JSP, ASP);
- скриптовые файлы (JS, ...)[36].

Может быть и посложнее — скажем, мы тестируем некий API-метод[37] «Закрыть объект»:

1. Закрыть контрагента.
2. Закрыть контрагента с двумя версиями.
3. Закрыть неактуальную версию контрагента.
4. Закрыть уже закрытого контрагента.
5. Закрыть объединенного контрагента.
6. Закрыть исходного контрагента, который уже был с кем-то объединен.
7. Закрыть контрагента, объединеного по схеме 2+2.
8. Закрыть атрибут.
9. Закрыть атрибут с двумя версиями.

Важно что? «Общий множитель» выносить за скобку. Скажем, если сходу непонятно, как вообще открыть форму, на которую написан чек-лист, выписываем это сверху. Не надо чек-лист превращать в тест-кейсы и внутрь каждой проверки засовывать предварительные шаги:

Неправильно	Правильно
1. Открыть страницу регистрации **https://www.example.com/**. Ввести в поле «имя» обычное имя, в поле «емейл» — корректный емейл, в поле «пароль» — корректный пароль. 2. Открыть страницу регистрации **https://www.example.com/**. Ввести в поле «имя» распространенное имя, в поле «емейл» — корректный емейл, в поле «пароль» — корректный пароль. 3. Открыть страницу регистрации **https://www.example.com/**. Ввести в поле «имя» редкое имя, в поле «емейл» — корректный емейл, в поле «пароль» — корректный пароль. 4. …	Открыть страницу регистрации **https://www.example.com/**. Проверки для поля «имя»: 1. Обычное. 2. Распространенное. 3. Редкое. 4. …

Чувствуете разницу? Чек-листы ведь потому и придумали, что гора текста утомила. Поэтому все лишнее — выкидываем. А дальше уже по принципу «кратко, но емко!» Поменьше текста, но чтобы смысл оставался понятным. Иногда для этого надо просто перечислить проверяемые данные (какие имена проверяем?), а иногда расписать, что будет в запросе, а что хранится в системе:

1	Запрос — Иванов В системе есть: 1. Иванов Иван 2. Петров Петр
2	Запрос — Иванов Иван В системе есть: 1. Иванов Иван 2. Иванов Петр 3. Петров Иван 4. Петров Петр
	...

Зачем в чек-листе нужны примеры?

Тут все очень просто. Примеры нужны для того, чтобы выполнить чек-лист, не включая мозг. Да, конечно, «работать надо не 12 часов, а головой», но, увы, иногда приходится пробежаться по чек-листу ASAP[38] после длинного рабочего дня, когда мозг уже не варит.

Примеры в чек-листе помогают выполнять его, даже если ты устал, мозг не варит и вообще тебя подняли среди ночи

И вот открываешь ты чек-лист, а там надо вводить реальные номера телефонов, чтобы проверить, что по ним город распознался. И вот такой клевый чек-лист:

- Москва.
- Питер.
- Омск.
- Новосибирск.
- Нижний Новгород.

Слабо навскидку реальный номер телефона в каждом городе вспомнить?

Кстати, вполне реальная задача. Дадата[39] умеет стандартизировать данные. В том числе — определять кучу всего по номеру телефона: город, таймзону (часовой пояс), оператора, регион. При смене часовых поясов тестировщику надо их проверить. Условие задачи при этом примерно такое:

27 апреля 2016 изменяются таймзоны:

- Ульяновская область со 2-й (МСК, московское время, UTC+3) на 3-ю (МСК+1, московское время плюс 1 час, UTC+4).
- Республика Алтай с 5-й (МСК+3, московское время плюс 3 часа, UTC+6) на 6-ю часовую зону (МСК+4, московское время плюс 4 часа, UTC+7).
- Алтайский край с 5-й (МСК+3, московское время плюс 3 часа, UTC+6) на 6-ю часовую зону (МСК+4, московское время плюс 4 часа, UTC+7).
- Сахалинская область (вся территория) будет находиться в 10-й часовой зоне (МСК+8, московское время плюс 8 часов, UTC+11).
- Астраханская область из 2-й часовой зоны (МСК, московское время, UTC+3) на 3-ю часовую зону (МСК+1, московское время плюс 1 час, UTC+4).
- Забайкальский край переходит из зоны МСК+5 в зону МСК+6.

Ну вот вам чек-лист, нормальное описание даже! Какая область, из какой часовой зоны в какую переехала. Сможете прямо сейчас взять и выполнить? Вот то-то и оно... Это надо гуглить города, а потом телефоны в них. В свое время я гуглила из серии `Сбербанк Омск` или `Кухни Омск`, так как Сбербанк есть везде, ну и мебель обычно тоже везде продают. Заходила на сайт любого магазинчика и забирала оттуда телефон. Вот и пример.

Одна только проблема — это ведь все время отнимает. Я потратила два часа на поиск примеров, проверила. Если их никуда не записать, зуб даю — задача снова всплывет. Скажем, разработчик что-то докрутит и попросит перепроверить. И что? Все телефоны уже забылись! И снова тратим два часа на поиск городов и телефонов в них.

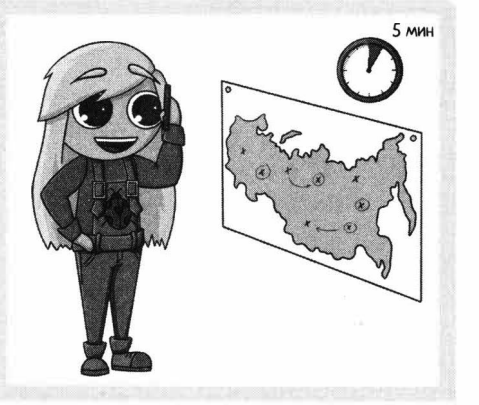
А если записать примеры, то перепрогон чек-листа займет минут 5. Удобно же!

Причем иногда кажется, что примеры будут из серии «капитан-очевидность». Скажем, тестируем мы поле «имя»:

- Простое.
- Мужское.
- Женское.
- Распространенное.
- Редкое.
- ...

Составление чек-листа

Здесь вроде бы все просто, и зачем приводить примеры, «неужели так сложно придумать мужское имя?». Мужское — нет, а вот с другими вариантами сложнее! Что, например, значит «распространенное имя»? А после 10-часового рабочего дня может и на «редком» заклинить. И это мы еще не дошли до пункта «Кызы Оглы», где реальное имя придется гуглить...



Прогон чек-листа, в котором есть примеры

По своему опыту могу сказать: когда ты думаешь, что чек-лист очевиден и примеры в нем не нужны, то тебе обязательно придется к нему вернуться. Через полгодика или хотя бы пару недель. И о своей лени ты пожалеешь ;-)

Какой результат писать в чек-листе?

Ровно тот, который вы ожидаете! Но помня о принципе «кратко, но емко!» — что именно мне надо проверить? На что обратить внимание? Об этом и пишем.

Когда начинающие тестировщики впервые сталкиваются с оформлением чек-листа, они впадают в ступор — какой должен быть результат?

Верный гугл говорит: «О, зацени, как все просто!»[40]:

○ Значение поля принимается.

○ Сообщение о некорректных данных.

Давайте попробуем применить на практике! Напишем чек-лист для регистрации на сайте Дадаты[41]. Результат может получиться абстрактным, конкретным или пустым.

Абстрактный результат — плохо!

Попробуем сделать все по ~~феншуй~~ найденному в гугле примеру. Обратите внимание на третий столбик. Хорошо ли, что там повторяется текст?

Чек-лист для формы регистрации — 1

Описание	Пример	Результат
Все поля заполнены правильно	Имя: Ольга Email: **ok.molechka@gmail.com** Пароль: 1	Регистрация прошла успешно, на почту отправлено письмо-приветствие
Проверка поля «Имя»		
Свое имя	Ольга	Регистрация прошла успешно, на почту отправлено письмо-приветствие.
Короткое имя	Ян	Регистрация прошла успешно, на почту отправлено письмо-приветствие.
Длинное имя (составное)	Розалинд Аруша Аркадина Алталун Флоренс Луна	Регистрация прошла успешно, на почту отправлено письмо-приветствие.
...	...	...

Окончание таблицы

Описание	Пример	Результат
Проверка поля «Email»		
Корректный email (популярный домен)	**olga@mail.ru**	Регистрация прошла успешно, на почту отправлено письмо-приветствие.
Точка внутри email	**ok.molechka@gmail.com**	Регистрация прошла успешно, на почту отправлено письмо-приветствие
Кириллический email	**олечка@мусики.рф**	Регистрация прошла успешно, на почту отправлено письмо-приветствие
Пустая почта		Ошибка
Одно слово вместо домена	**olga@fdgfdg**	Ошибка
...		

Повторение — мать учения. А копипаста — зло!

Это знает каждый тестировщик. Будем хитрее — вынесем одинаковый текст «за скобку».

Чек-лист для формы регистрации — 2

Если «Регистрация прошла успешно, на почту отправлено письмо-приветствие» — результат «OK».

Описание	Пример	Результат
Все поля заполнены правильно	Имя: Ольга Email: **ok.molechka@gmail.com** Пароль: 1	OK
Проверка поля «Имя»		
Свое имя	Ольга	OK
Короткое имя	Ева	OK
Длинное имя (составное)	Розалинд Аруша Аркадина Алталун Флоренс Луна	OK
...	...	...
Проверка поля «Email»		
Корректный email (популярный домен)	**olga@mail.ru**	OK

Окончание таблицы

Описание	Пример	Результат
Точка внутри email	**ok.molechka@gmail.com**	OK
Кириллический email	**олечка@мусики.рф**	OK
Пустая почта		Ошибка
Одно слово вместо домена	**olga@fdgfdg**	Ошибка
...		

Какие мы молодцы! И чек-лист написали и «текст ради текста» выкинули, круто же! Или нет?

Можно ли понять из этого чек-листа, зачем нужны были все эти проверки? Давайте посмотрим, что можно писать вместо абстрактного «OK».

Конкретный результат — хорошо!

Берем абстрактный результат. Думаем, что именно мы хотим узнать в этой проверке, — об этом и пишем:

Логин с длинным именем — проверяем, что имя влезает туда, где оно должно отображаться после регистрации.

Логин с русской почтой — проверяем, что на кириллическую почту нормально уходит письмо.

Буква ё в имени — проверяем, что она нормально отображается в письме-приветствии и в личном кабинете: в виде буквы, а не иероглифа.

и т. д.

Мы же не просто так 20 кейсов проверяем, чем-то же они отличаются. Описывайте эту специфику вместо банального «OK» — это и будет конкретика.

А результата «не OK» вообще не бывает! Разные ошибки обрабатываются системой по-разному. И это тоже надо отобразить в чек-листе. Иначе как проверяющий поймет, нормальный текст ошибки или не очень? Не всегда очевидно, что там должно быть...

Вот ты видишь в чек-листе результат «Ошибка» на ввод неправильного пароля. Система честно выдает ошибку: «Вы ошиблись, у пользователя admin пароль не marusia, а pupsik». Нормальный текст ошибки? А что, формально подходит! И уж как будут рады нечестные на руку пользователи такой системе. Ты смотри, не надо хакером быть — лишь логин подобрать!

Или, допустим, система пишет «Неправильный пароль». Правильное сообщение об ошибке или нет? Тоже плохое, потому что подтверждает существование логина в базе — взломщику проще перебирать. Сначала подобрал логин, который точно есть, а потом берет его брутфорсом[42].

Или система пишет «ПОШЕЛ НАФИГ, ИДИОТ!» — это ОК, тест пройден? Сообщение об ошибке ведь есть! Конечно, пример немного утрирован, хотя кто знает... Бывает такое, когда разработчики ставят шуточные сообщения об ошибках, чтобы перед релизом убрать. И, разумеется, о них забывают. Так что все бывает...

Ну или если разработчик не смог обработать сообщение об ошибке, и пользователю вывелось гениальное окошко «Null» — это нормально? И ладно бы окошко, а то ведь может и «синий экран смерти» показать... Тоже сообщение об ошибке, но вот ни разу не нормальное.

Давайте попробуем конкретизировать проверки для чек-листа регистрации в Дадате.

Чек-лист для формы регистрации — 3

Описание	Пример	Результат
Все поля заполнены правильно	Имя: Ольга Email: **ok.molechka@gmail.com** Пароль: 1	Регистрация прошла успешно, на почту отправлено письмо-приветствие

Окончание таблицы

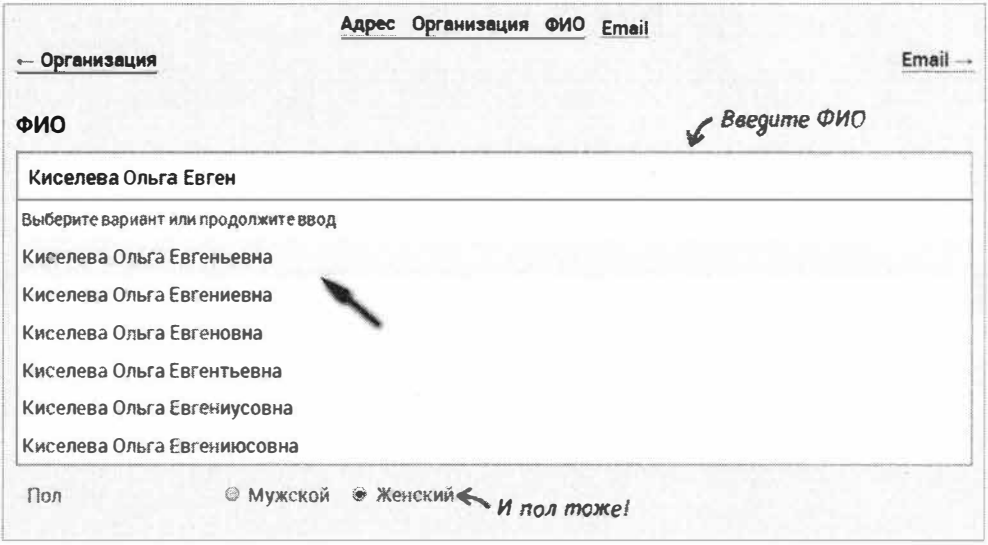
Описание	Пример	Результат
Проверка поля «Имя»		
Свое имя	Ольга	Успешная регистрация
Короткое имя	Ян	Ограничений на минимальное имя нет
Длинное имя (составное)	Розалинд Аруша Аркадина Алталун Флоренс Луна	В личном кабинете имя отображается нормально, влезает в отведенное под него место
...	...	...
Проверка поля «Email»		
Корректный email (популярный домен)	**olga@mail.ru**	Успешная регистрация
Точка внутри email	**ok.molechka@gmail.com**	Успешная регистрация
Кириллический email	**олечка@мусики.рф**	На кириллический email ушло письмо
Пустая почта		Ошибка. «Введите email»
Одно слово вместо домена	**olga@fdgfdg**	Ошибка. «Введите адрес в формате **mail@site.com**»
...		

Если ожидаемый результат дублируется — объединяем ячейки в таблице. Так мы избавляемся от лишнего текста, а чек-лист становится легче читать. Посмотрим это на примере поля «Email».

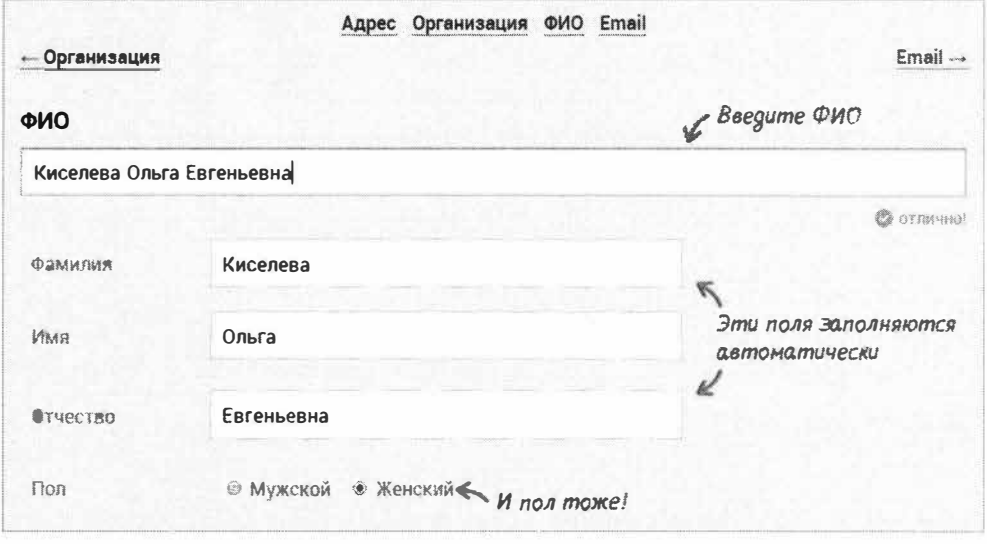
Чек-лист для формы регистрации — 4

Описание	Пример	Результат
Корректный email (популярный домен)	**olga@mail.ru**	Регистрация прошла успешно, на почту отправлено письмо-приветствие
Корректный email (корпоративная сеть)	**olga@company.ru**	
Точка внутри email	**ok.molechka@gmail.com**	
Кириллический email	**олечка@мусики.рф**	На кириллический email ушло письмо
Пустая почта		Ошибка. «Введите email»
Одно слово вместо домена	**olgak@fdgfdg**	Ошибка. «Введите адрес в формате **mail@site.com**»

Давайте взглянем на еще один пример — подсказки по ФИО. Вводим данные, система подсказывает.



Как будет выглядеть чек-лист с абстрактным результатом?



Чек-лист для подсказок — 1

Описание	Пример	Результат
С какой буквы начинаются подсказки	О	Выпадает список подсказок

Окончание таблицы

Описание	Пример	Результат
Женское имя, распространенное	Ольга	Имя распознано корректно
...		
ФИО	Киселева Ольга Евгеньевна	Данные разнесены по правильным полям: • Фамилия в поле «Фамилия». • Имя в поле «Имя». • Отчество в поле «Отчество». • Пол определен верно.

Смотрится хорошо и правильно... Но что значит «выпадает список подсказок»? Я ввожу букву «О», а мне выдается список подсказок с первой буквы алфавита — это ОК? Нет, не ОК, хотя результату соответствует.

Что такое «данные разнесены по правильным полям»? Если я ввела «Киселева Ольга» и Дадата запихала эту строку в отчество, выставив мужской пол — это ОК? Нет, не ОК. А что тогда ОК?

Ответ кажется очевидным, можно отмахаться — тренер/начальник придирается. Но очевиден результат далеко не всегда. Представьте себе, что вы — алгоритм, который определяет, где фамилия, где имя, где отчество. Распределите эти данные:

Константин Семенович Маркович
Саша Савченко
Кенгатаран Урутирарубини
Курувитаге Манори Исанка
Бжадуг Маруан Талал
Гохил Харпалсинх Полубха
Улига Тумэнжаргал
Абдул Вахаб (это все фамилия!)
Аслан оглы Назила
Казы Кызы Гулянда
Пан Полина Сон-Деровна
Тен Вова Алексейвич
Кан Рая Николаевна
Нгуен Ту Ань
Нгуен Нгок Лонг
Кожокару Ион Георге
Фан Се Рюн

Ну как, все еще очевидно?

Давайте добавим в чек-лист конкретики:

Чек-лист для подсказок — 2

Описание	Пример	Результат
С какой буквы начинаются подсказки	О	Выпадает список имен и фамилий, начинающихся на букву «О»
Женское имя, распространенное	Ольга	В списке подсказок «Ольга» есть. При выборе имени заполняются поля: • Имя — Ольга. • Пол — женский.
...		
ФИО	Киселева Ольга Евгеньевна	В списке подсказок все компоненты есть. При выборе заполняются поля: • Фамилия — Киселева. • Имя — Ольга. • Отчество — Евгеньевна. • Пол — женский.

Такой чек-лист сможет выполнить любой тестировщик — от начинающего до опытного, от знающего каждый кусочек продукта до того, кто впервые Дадату видит. В этом и заключается прелесть конкретного результата.

Пустой результат — прокатит

Если результат писать лень и очень хочется скатиться в абстрактное «ОК» — вообще его не пишите. Конкретика нужна не всегда.

Есть отличный пример[43] чек-листа без результата — для загрузки файлов в Дадату. Почитайте его, нужны ли тут пояснения, примеры, однотипный результат? Нет, не нужны!

Но, повторюсь, тут стоит быть очень осторожным. Когда вы набираетесь опыта, вам многое кажется очевидным, включая набор телефонов разных городов России. Ведь вы тестируете это каждый день, ну или просто «прямо сейчас», — вы в контексте, результат кажется само собой разумеющимся. И примеры не нужны, все в голове держится. И кажется, так будет всегда.

Однако поверьте, буквально через пару недель вы успеете забыть и примеры, и ожидаемый результат. Или дадите чек-лист коллеге и придется прояснять каждую деталь. Так что, с одной стороны, не надо писать «капитан-очевидные» вещи, но стоит проверить «на кошках», правда ли они так очевидны.

Написали чек-лист? Покажите коллеге, да хоть разработчику или аналитику. Ваша задача: проверить, понятен ли он. Пусть опытным коллегам, а не любой обезьянке с улицы, ну и что? Это ж не тест-кейс! Главное, чтобы команда все понимала. И вы в том числе, даже спустя полгода.

Итого: когда и какие результаты использовать?

- **Абстрактный** — забудьте как страшный сон, особенно «не ОК».
- **Конкретный** — когда непонятно, что именно означает «ОК» (для каждого пункта результат разный).
- **Пустой** — когда результат превращается в унылую копипасту.

Особенности чек-листов

- **В них минимум информации.**

Ровно столько, сколько надо для понимания проверки опытным тестировщиком. Никаких тебе «найди эту кнопочку вверху экрана», на любого человека с улицы не ориентируемся. Чисто список-напоминалка, «кратко, но емко!»

- **Они независимые.**

Этим чек-листы похожи на тест-кейсы, но здесь независимость достигается проще. Один чек-лист — это *N* проверок на один и тот же функционал. Поэтому и не надо никуда ссылаться!

Плюсы и минусы

Плюсы

- **Нет копипасты** — это главный бич тест-кейсов, от которого мы избавились в чек-листах.
- **Меньше текста** — проще и быстрее читать.
- **Проще поддерживать** — за счет предыдущих плюсов в основном. Если меньше текста, он медленнее устаревает.
- **Быстрее писать** — аналогично: меньше текста, больше дела!

Минусы

- **Не все поймут**. Вот отдашь джуниору чек-лист для SQL-инъекций, а он даже не поймет, что это такое и куда это вставлять.

Однако мне кажется, плюсов все-таки больше ;-)

Когда применять чек-листы?

Чек-листы применимы, когда:

- вы — единственный тестировщик на проекте. Тогда нет смысла тратить время на подробное описание шагов, которые вы и так уже все наизусть знаете;
- система не суперсложная, и без указания конкретных кнопочек понятно, что и как делать.

В крайнем случае, всегда можно совмещать: описали подробно, как дойти до отчета, а потом уже в виде чек-листа накидали варианты тестирования.

Инструменты для оформления чек-листов

- Специализированные:
  - «Ситечко»[44];
  - Test IT[45];
- Остальные:
  - Word;
  - Excel;
  - MindMap;
  - Гуглодоки;
  - Confluence;
  - ...

В случае чек-листов я рекомендую использовать специальный инструмент — «Ситечко»! Особенно, если пока вы их нигде не вели. Почему? Потому что его делают тестировщики! Автор системы — Наталья Руколь, а это уже о чем-то говорит. ;-)

С другой стороны, не надо усложнять без необходимости. «Ситечко» — крутой инструмент для ручных тестов: вы можете проходить по чек-листам и ставить галочки «тест пройдет/тест провален». И получается красивая отчетность — всегда видно, когда баг в системе появился, а когда был исправлен.

У меня на одном из проектов было так: ручных тестов почти нет, всё автоматизировано. Поэтому там мы хранили свои чек-листы (описания автотестов) в конфлюенсе и не парились. Результаты каждого прогона? Пожалуйста, для этого есть TeamCity[46]. В этом случае «Ситечко» не нужно.

Думаю, в начале карьеры у вас все же будут ручные тесты, а начальство захочет видеть графики. Так что рекомендую «Ситечко» хотя бы попробовать. А там уже смотрите сами: где удобнее, там и работаем. Это самое важное правило!

Сравним тест-кейсы и чек-листы

Мы много говорили отдельно о тест-кейсах и отдельно о чек-листах. Хочется свести воедино всю полученную информацию и наглядно ее показать.

Если мы пишем про авторизацию по email, то выглядеть один и тот же тест может по-разному:

Выводы:

- тест-кейс — подробно все расписываем;
- чек-лист — кратенько.

Но это в теории понятно, пока читаешь книжку или слушаешь лекцию. Как только доходит до практики, сразу: «Ой!». Моим студентам тяжело дается разница. Зачем в тест-кейсе писать, что именно за файл создается, как его загружать в систему (на какие кнопки нажимать, какие действия выполнять)?

Я предлагаю им пару вариантов сравнения, которые вы можете найти на сайте Testbase[47].

Тест-кейс

Шаги

1. Открываем сайт https://www.example.com/
2. Нажимаем кнопку «Вход» в правом верхнем углу.
3. Устанавливаем курсор на поле «email».
4. Вводим значение «olga@mail.com».
5. Устанавливаем курсор на поле «Пароль».
6. Вводим значение «12345».
7. Нажимаем «Сохранить».

Ожидаемый результат

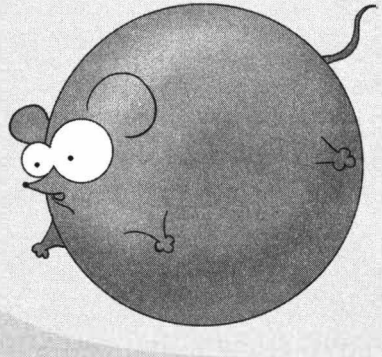
Открылась главная страница сайта.
В верхнем правом углу отображается приветствие — «Здравствуйте, Ольга!»

Чек-лист

- Авторизоваться через email

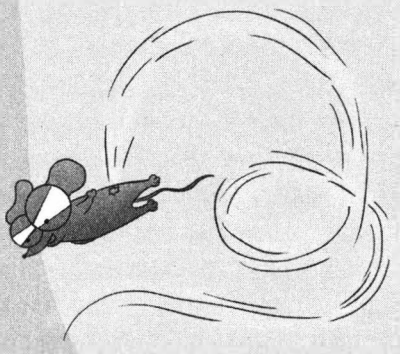
Тест-кейс

Понятен любому → много текста



Чек-лист

Понятен, но не всем → мало текста



Идея 1: дите или подросток

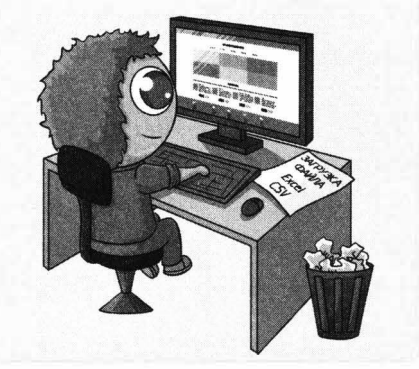
Я даю им такое пояснение:

Тест-кейсы — тупые до невозможности, словно ребенка на работу привели и показываем: «Вот мамочка сейчас файл обработает. Нажимаем кнопочку А, потом кнопочку Б, потом…», а не просто: «Ну вот загрузили, и все получилось».

Ну а чек-листы — это когда не нужны все эти подробности, как именно мы загружаем файлы, на какие кнопочки нажимаем. Нужна просто напоминалка: «Проверить загрузку Excel, CSV, JPG…»

Тест-кейс максимально подробный, не только ЧТО тестируем, но и КАК

Чек-лист говорит, ЧТО тестируем, но не КАК, без подробностей



Идея 2: IKEA

Вы делали ремонт? Покупали шкафы, собирали их? А я делала, и отсюда у меня вторая ассоциация.

Мы купили комод в ИКЕА. Он небольшой и простой в сборке. Но инструкция выглядит как талмуд — все настолько подробно. Каждое действие, каждый шаг. Каждый винтик — все в новом пункте на пол-листа А4, максильно доступно.

Такой комод соберет даже полный профан. Потому что ребята не считают нужным пропускать этапы из-за «Ну это же очевидно, куда ввинчивать

Тест-кейс. Всё подробненько и чётко. Вплоть до названия нажимаемой кнопки

этот шуруп». Очень напоминает «Ну это же очевидно, на какую кнопочку нажимать, чтобы загрузить файл», что в тест-кейсах недопустимо.

А вот диванчик в коридор мы купили в другом месте. Он тоже небольшой и не сильно сложный в сборке — кубик шкафчика собрать и прицепить к самому диванчику. Но инструкция — полный швах. На ней ровно одна картинка — уже собранный диванчик, все детали чуть на расстоянии друг от друга. Ну это же очевидно, как его собрать!

Мы, кстати, не осилили, оставили мастеру

Разница: «Обычная инструкция — инструкция из ИКЕА» очень похожа на «Чек-листы — Тест-кейсы». Когда будете писать тест-кейс, помните об этом. И о том, что очевидное вам — темный лес для кого-то другого...

Чит-листы

Серебряной пули не бывает, но на типовые формочки уже давно написаны чек-листы. Их еще называют «чит-листами».

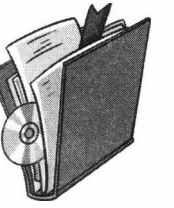
Поэтому, перед тем как проводить тестирование поля для email, попробуйте погуглить: `Чит-лист заполнения email`. А то, может, вы упустите какой-то тест, не догадавшись про класс эквивалентности, а в чужом чек-листе тест будет. Потому что кто-то уже огреб проблем и знает, что такую проверку лучше выносить отдельно.

В общем, типовые поля можно погуглить и пополнить свои списки проверок! Есть даже программы с чит-листами внутри, но об этом мы поговорим в следующей главе.

Вопросы для самопроверки

1. Какое самое важное качество теста?
2. Какий тест мне выполнить вначале для количества книг в заказе: пустое поле или «3»? Почему?
3. Какие поля в тест-кейсе обязательные, а какие нет?
4. Чем отличаются тест-кейсы от чек-листов?

Портфолио



Продолжаем делать портфолио:

1. Напишите два тест-кейса на свое приложение: один с результатом на каждый шаг, второй с одним результатом.
2. Напишите чек-лист на любой функционал.

Потом покажите товарищу, если он сможет выполнить тест-кейсы — значит, вы справились! Если задает уточняющие вопросы — потренируйтесь еще.

Ответы на вопросы «на подумать»

1. Почему при тестировании формата DOC, который система не умеет обрабатывать, мы готовим файл с данными из образца?

Предварительный шаг в этом случае:

> Подготовить файл формата DOC с данными из файла-примера (см аттач «Пример.doc»).

Если мы тестируем формат файла, то нам неважно, какие именно данные будут внутри, но они должны быть позитивными (иначе мы будем тестировать не формат, а наполнение).

Нужно сначала провести тест на данные, а потом взять тот же файл и поменять расширение. А где взять позитивные данные? Проще всего — в образце. Ведь его мы тестируем в любом случае (см ранее, почему).

2. А теперь сами: чем отличается Ященко от остальных? У нее тоже определены падежи и хороший код качества. Напомню все три ФИО:

1. Иванов Сергей Владимирович

2. Федотов Алексей

3. Ольга Павловна Ященко

Ответ:

1. Несклоняемая фамилия, пол определяли по ИО.
2. Формат ИОФ вместо ФИО не влияет на код качества.

Ответы на вопросы для самопроверки

1. Какое самое важное качество теста?

Он должен быть понятным, что я делаю и как.

2. Какий тест мне выполнить вначале для количества книг в заказе: пустое поле или «3»? Почему?

Начинаем всегда с позитива, то есть с «3».

3. Какие поля в тест-кейсе обязательные, а какие нет?

Обязательные: название, шаги, результат. Еще может быть номер и предварительные шаги, остальное так — доп. инфо.

4. Чем отличаются тест-кейсы от чек-листов?

Тест-кейс — максимально подробное описание проверки. Что и как подготовить, какой сайт открыть, какие кнопочки потыкать. По тест-кейсу может пройти человек, абсолютно незнакомый с системой, даже если она сложная.

Чек-лист — краткое описание проверки. Оно для тех, кто с системой знаком, тут нет подробностей. Это просто список «что не забыть проверить». Его проще поддерживать в актуальном состоянии и быстрее писать, да он и используется намного чаще.

Глава 3
КЛАССЫ ЭКВИВАЛЕНТНОСТИ И ГРАНИЧНЫЕ ЗНАЧЕНИЯ

Мы разобрались, что нужно тестировать в первую очередь, а что потом. И даже написали первую версию чек-листа. Пришла пора ее усовершенствовать. В этой главе:

- выделяем классы эквивалентности;
- ищем граничные значения;
- расширяем свои чек-листы.

Несколько вступительных слов

Вот мы и подобрались к самой важной теме. Без нее — никуда. И если оформлению тест-кейсов и чек-листов учатся начинающие, то на тест-дизайн приходят люди с пятилетним опытом и все равно узнают что-то новое для себя.

Поэтому я сделала эту главу небольшой. Я расскажу основы и дам ссылки на дополнительные материалы. А дальше — практика, практика и еще раз практика! Когда надоест, почитайте мою вторую книгу «Тест-дизайн для новичков»[48] и найдите в ней новые источники вдохновения ;-)

Также я надеюсь, вы заметили немного необычную структуру книги. Я не начинаю с темы про классификацию тестирования или с чего-то такого. Сначала — все самое важное и нужное. Чтобы книга приносила пользу даже в том случае, если вы прочитали только ее часть.

Что нужно от тестировщика? Тестировать да баги оформлять. Чтобы тестировать, надо понимать, ЧТО именно вы будете тестировать. Об этом мы и говорим в первых четырех главах: что и как тестировать. Чтобы начать тестирование, надо изучить продукт и задать правильные вопросы. А потом уже придумывать тесты, помня про их приоритеты.

Это потом, в *пятой главе*, мы обсудим не менее важную тему: постановки багов. Но, чтобы найти баг, нужно сначала протестировать программу! Поэтому — сначала все, что вам стоит знать о составлении тестов, чему и посвящена эта глава.

Вы уже набросали чек-лист для своего будущего портфолио? Если нет, вернитесь к предыдущей главе и обязательно набросайте. Чтобы по мере чтения вы смогли его расширить и заметить разницу «до-после».

Тест-дизайн

Тест-дизайн — набор техник, которые позволяют нам приложить мало усилий и получить много результата.

Ведь если в арсенале есть только молоток, вам придется им и гвозди забивать, и винтики закручивать. Или наоборот, отверткой гвозди забивать, что можно, но очень неудобно.

Поэтому приятно иметь доступ к самым разным техникам. Когда есть из чего выбирать, задачи решаются быстрее и приятнее.

Итак, основные техники тест-дизайна (табл. 3.1).

Таблица 3.1. *Техники тест-дизайна*

Английское название	Русский эквивалент
Equivalence Class Testing	Классы эквивалентности
Boundary Value Testing	Граничные значения
State-Transition Testing	Диаграмма состояний и переходов
Decision Table Testing	Таблица решений
User Case Testing	Варианты использования
Allpairs Algorithm testing	Попарное тестирование
Orthogonal Arrays Testing	Ортогональная матрица
...	...

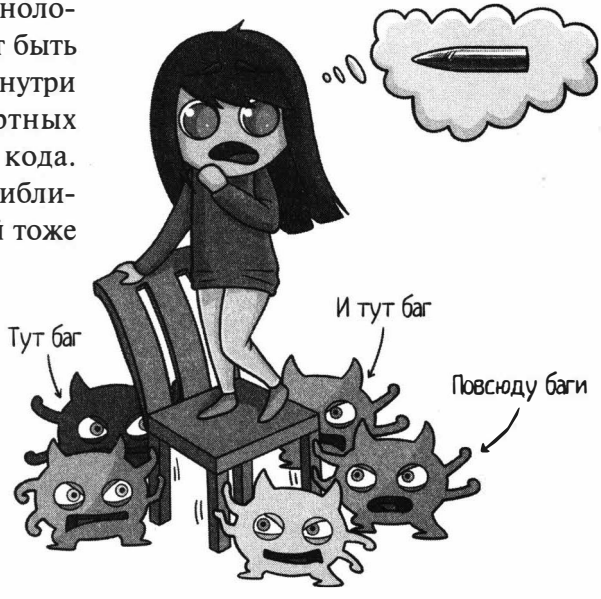
Но в этой главе мы не будем рассматривать их все. Помните, о чем я писала чуть ранее? Сначала я хочу дать вам базу. Ту базу, без которой никуда. Вот вообще никуда. И только пото-о-о-о-ом... Всякие отдельные плюшки. Но нет смысла пытаться применять навороченные техники, пока не отточили самые базовые.

Поэтому здесь и сейчас будем говорить только про классы эквивалентности и граничные значения. Они — МЕГА важны. Даже новичкам. Но и опытным тоже. Тот же allpairs вы можете вообще не применять, но про границы знать обязаны.

Тест-дизайн — это не наука

Помните об этом. Закон Ньютона — он и в Африке закон Ньютона, не меняется и всегда работает одинаково. Но в тестировании «серебряной пули» нет. Нельзя один раз написать мега-клевый чек-лист и сказать, что он подойдет вообще подо все.

В каждой программе свои баги, свои проблемы и особенности. Есть некоторые фишки, которые стоит проверять всегда[49], но все равно даже универсальный чек-лист вам придется подкручивать под особенности своей системы. Каждый разработчик совершает какие-то свои ошибки, каждый функционал хоть немного, но отличается от того, что вы видели раньше.



К тому же меняются технологии, а внутри них тоже могут быть свои ошибки. Что уж там, внутри проекта есть набор стандартных библиотек — уже готового кода. Разработчики подключают библиотечку в свой проект, а в ней тоже могут быть баги, зависимые от версии библиотеки. Да и свой код разработчики пишут по-разному. Как же тут единой «серебряной пулей» обойтись?

Поэтому нельзя точно сказать, что проверки для поля с именем — они только такие, и точка. Нет. Для одного поля они будут одни, для другого — другие. В русской и китайской системах проверки будут разные. Да даже в двух русских системах они могут быть совершенно разные, потому что в одной разработчик добавит подсказки или ограничение на ввод символов, а в другой просто скажет, что это string, вводи туда все что угодно. И если мы будем проводить одни и те же тесты, то никогда не найдем багов, которые есть именно в нашей системе. Так что увы, «серебряной пули» не будет.

Техники тест-дизайна — это набор эвристик, с помощью которых мы можем улучшить наши результаты[50]. Если вы неправильно выбрали эвристику, то пропустили баг.

Это очень сложный материал, который можно изучать долго, совершенствоваться в нем годами, даже если у вас уже год опыта, два, три, четыре. На

курс по тест-дизайну к нам приходили порой люди с пятилетним опытом и при этом узнавали для себя что-то новое, и это помогало им взглянуть на систему по-новому. Поэтому развивайтесь и прокачивайтесь.

Главное, что стоит помнить — нельзя стоять на месте. Идите вперед. Техники тест-дизайна, они, как ориентир, помогают нам понять, как проводить дальнейшее тестирование. Когда-то я была у Алексея Баранцева на тренинге, и он рассказал такую историю (я не помню ее точно, поэтому, возможно, немного перевру. Главное, что смысл останется):

> Группа военных заблудилась где-то в Альпах. Генерал достает из-за пазухи карту и говорит: «Слушайте, вот у меня карта, я точно знаю, куда нам надо идти. Нам туда!».
>
> Группа пошла за генералом. Они шли-шли-шли, и, наконец вышли к людям. Те уже вызвали спасательную бригаду, она приехала, нашу группу забрали, привезли обратно в штаб, и все было хорошо.
>
> Но полковник-спасатель подошел к генералу и попросил показать ему ту карту. Он взял ее, посмотрел и говорит:
>
> — Слушай, это же не Альпы, это же Гималаи.
>
> — Я знаю.
>
> — Это не та карта. Как ты смог их вывести??
>
> Генерал пожал плечами:
>
> — Нам нужен был ориентир. Мы им воспользовались и смогли выйти. Без паники, никто же не знал, что реальной карты нету.
>
>
>
>
> А если бы они остались на месте или стали плутать туда-сюда, то поднялась бы паника, в итоге ситуация бы усугубилась, и они бы просто замерзли.

Вот видите? Карта же не та была! Но они все равно выжили. И даже не волновались, так как были уверены в том, что идут в правильном направлении. Точно так же в техниках тест-дизайна. Если вообще ничего не делать, то вы никуда не придете и багов не найдете. А если вы будете хоть куда-то идти, пусть даже вы определили неправильные эвристики, неправильно выделили классы эквивалентности, все равно вы куда-нибудь да придете. Со временем вы научитесь корректировать свои ориентиры и находить более быструю дорогу к тому же самому результату.

Ну а я предлагаю начать с классов эквивалентности. Это самая сложная для восприятия тема, потому что ее нелегко применить «в жизни». Но давайте попробуем!

Классы эквивалентности

Классы эквивалентности — что это?

Два значения называются *эквивалентными*, когда приводят к одному результату. В таком случае нам неважно, взять значение А или Б для теста — мы знаем, что они идентичны.

Эквивалентны — результат один

Во всех базовых книгах по тестированию — например, в книге LeeCopeland[51] (очень люблю эту книгу, сама с нее начинала чтение о тестировании) — дают пример с калькулятором, чтобы наглядно пояснить, зачем классы нужны. Не буду отходить от классики — давайте посмотрим на примере с калькулятором.

Классы эквивалентности в калькуляторе

Допустим, мы разрабатываем новый модный калькулятор. На текущий момент готова только функция сложения, и нам надо ее проверить. Как будем проверять? Если у нас нет доступа к коду, мы не можем сказать уверенно, как разработчик эту функцию реализовал. И, чтобы точно сказать, что «все работает, багов нет», нам придется проверить сумму каждого с каждым:

- 1 + 0
- 1 + 1
- 1 + 2
- ...
- 1 + 999999999999999
- 2 + 0
- 2 + 1
- ....
- 2 + 999999999999999
- ...
- 9999999999999999 + 1
- ...

Уже получится очень много тестов, а это мы еще до дробных значений не дошли. Что, если один знак после запятой? А если два? А если три-четыре-десять? Тестов получаются триллионы. На один простой функционал...

Мы просто не можем позволить себе полный перебор всех значений, иначе на проверку одного только сложения уйдет несколько лет работы полной командой. А еще вычитание, умножение, деление, комбинации, у-у-у-у-у..

Ну и потом, если нам надо делать полный перебор — это значит, что и разработчик у нас слегка туповатый и в коде тоже делает перебор вместо функции суммирования. Но в таком случае он бы тоже несколько лет писал простой функционал, а, раз он его сделал за пару дней — значит, там все как-то пооптимальнее работает.

Вот и нам надо пооптимальнее работать. Какие месяцы и года, нам и двух дней, да что уж там, порой двух часов не дадут на проверку функции. Что логично — функционал простой, зачем его долго тестировать?

Поэтому мы начинаем делать предположения. Наверное, если «2+2» работает, то и «2+3» будет работать, так как сама функция сложения работает. В этом случае тесты эквивалентны. Если мы проверили «2+2», то «2+3» проверять уже не надо, мы заранее предполагаем, что он тоже будет работать.

Теперь осталось придумать название для этого класса эквивалентности. Зачем? Затем, что название подтолкнет нас к другим классам. Можно сказать, что мы проверили «сложение чисел», но это слишком общая фраза. Так можно сказать, что «мы проверили функционал калькулятора», но будет ли это правдой? Функционала-то там много. Если мы проверили сложение, это не значит, что вычитание и умножение тоже будут работать.

Мы предполагаем, что, если проверено «2+2», то и «2+3» работать будет, эти значения эквивалентны

Поэтому начинаем думать, а какие вообще бывают числа?

- Простые — которые делятся только на 1 и сами на себя.
- Обычные — 1, 2, 3... Те, что чаще всего используются.
- Большие значения — мы чуть позже поговорим о граничных значениях и обсудим, почему большие значения надо выносить в отдельный класс. Но фишка в том, что, если «1+1» работает, то «максимум + максимум» может и упасть. Ну и просто, мало ли, число не влезет на экран, что делать будем? Глазки потупив, говорить, что тестировали?

- Дробные — стоп, а какие возможны варианты записи? Дробная часть через точку или запятую — уже два разных класса!

Таким образом, мы выделили разные классы эквивалентности, сгруппировав проверки по какому-то принципу. И вся соль тестирования — выделить классы правильно. Не просто «сложение чисел», а подумать, какие классы могут скрываться внутри.

Главное — правильно понять принцип разбиения. Не слишком детализированный, но и не слишком общий

Если класс будет слишком общий (сложение чисел) — мы пропустим баги, так как проверим только «2+2», а на больших или дробных значениях все может упасть.

Если класс будет слишком узкий (числа до 10, до 20...) — получится слишком много тестов. Поэтому вся наша задача сводится к наиболее правильному предположению о том, какие проверки дадут одинаковый результат, то есть какие из них можно выкинуть.

Классы эквивалентности через Золушку

Я в 2015 году думала, как объяснить тему классов эквивалентности студентам. И придумала аналогию с... Золушкой[52]! Помните, там мачеха рассыпала на полу гречку и овес, перемешала и сказала: «Разбирай!».

Аналогично с тестированием — когда мы тестируем какой-то функционал, перед нами куча идей для тестов. Но вот что из этого будет гречкой, а что — овсом? Можем ли мы с уверенностью сказать, что имена «Павел» и «Мария» будут одним классом эквивалентности?

Все зависит от контекста. Если это просто строка, которая никак не валидируется и (что важно!) никак дальше не используется, просто хранится — тогда это один класс (гречка — и то, и другое). А вот если по имени мы шлем в письме «уважаем*ый*» или «уважаем*ая*» — это уже будут разные классы.

Сядьте и подумайте — для чего именно нужно тестируемое поле. Что будет происходить с этим значением дальше? Отталкивайтесь от этой информации при выделении классов. Вот напоминалка:

1. Подумайте, какими бывают «тестируемые поля».
2. Представьте, какими не бывают «тестируемые поля».
3. Поищите границы.

Идеи для тестов

Я понимаю, что в начале пути очень тяжело придумывать тесты «с нуля». Идей просто нет, потому что пока нет понимания, чем одно значение отличается от другого и отличается ли вообще.

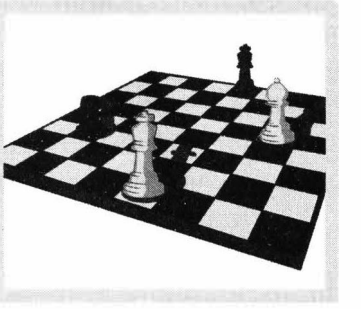
Поэтому я собрала для вас подборку идей для тестирования разных полей. Найти её можно на Хабре в статье «Где брать идеи для тестов (подборка полезных ссылок)»[53].

Что там есть?

- Текст:
  - тестирование текстового поля;
  - тестируем поля логин/пароль;
  - тестирование полей ввода.
- Число:
  - чек-лист для тестирования числового поля;
  - классы эквивалентности для строки, которая обозначает число.
- Дата:
  - классы эквивалентности для строки, которая обозначает дату.
- Мобильные приложения:
  - чек-лист для тестирования мобильных приложений.
- Остальное:
  - классы эквивалентности для имен;
  - классы эквивалентности для населенных пунктов в адресах;
  - классы эквивалентности для стандартного грида — то есть для шапки отчета, по которой можно сортировать;

- ✧ «Это еще не конец!» — в этой статье Michael Hunter рассказывает про разные методы ввода, файлы, сетевое соединение, сообщения об ошибках, доступность, меню...
- ✧ Юлия → Iuliia. Схемы транслитерации — если ваша система что-то транслитерирует, это будет полезно.

Когда остановиться?



Когда остановиться?

Это самый важный вопрос при выделении классов эквивалентности — когда остановиться-то? Когда перестать генерировать все новые и новые идеи?

Об этом мы поговорим в следующей главе, когда будем проводить тест-анализ. Но, в любом случае, чтобы выкинуть проверки, нужно, чтобы было что выкидывать. И первоочередная задача — уметь сходу нагенерировать кучу идей для проверок.

Не просто «буковки русские, буковки английские, спецсимволы, перемешал», а конкретно под ваше поле. Если их в голову приходит ОЧЕНЬ много (Маша, Даша, Глаша, Оля, Света...) — значит, вы не выделили какой-то класс эквивалентности, в который они все входят.

Помните — классы эквивалентности тоже нужны для того, чтобы уменьшать количество тестов. И вместо перебора всех женских имен проверить лишь одно. Предположив, что система поведет себя на остальных аналогично.

Остановились, перевевели дух, а потом осмотрели полученные классы и проверили их границы.

Эффект пестицида

Если повторять одни и те же тесты снова и снова, в какой-то момент они перестанут выявлять новые ошибки.

Это положение обосновал Борис Бейзер в 1983 году в своей книге «Software Testing Techniques». Он привел пример обработки полей пестицидами.

> Поле обрабатывается неким пестицидом в первый раз, и значительная часть вредителей погибает. Но некоторые всё же выживают, потому что их организмы оказались устойчивы к действию яда. Если повторно обработать поле тем же пестицидом, то выжившие после первой обработки с большой вероятностью выживут и после второй.

Аналогично в тестировании. Повторное применение тех же тестов и тех же методик приводит к тому, что в продукте остаются именно те дефекты, против которых эти тесты и эти методики неэффективны.

Чтобы избежать эффекта пестицида, нужно каждый раз брать разные значения для тестов. Например, у нас есть такой чек-лист для тестирования имени при регистрации (табл. 3.2).

Таблица 3.2. *Чек-лист для тестирования имени при регистрации*

Проверка	Пример	Результат
Мужское	Иван	Успешная регистрация
Женское	Мария	Успешная регистрация
Унисекс	Саша	Успешная регистрация
...	...	...
Пустое		Ошибка: введите имя!

В первый раз берем пример, а потом используем что-то своё. Мужское имя? Иван, Александр, Ян, Анатолий...

Это помогает найти ошибки, о которых мы даже не думали! Например, система ругается на вполне нормальное имя «Иван», пропуская «Саша», «Антон» или даже «Фигня» (вполне реальная история с Иваном, см. подробнее тут: **https://okiseleva.blogspot.com/2019/10/blog-post_15.html**).

Или мы исходно предполагаем, что имя — это простая строка, поэтому ее не надо делить на «мужское/женское/...». А потом выясняется, что система по имени ставит пол и всех считает мужчинами. Или ругается на женские имена, мол, «таких не бывает». Или что-то ещё.

То есть смена тестовых данных может помочь нам найти класс эквивалентности, о котором мы не подумали. Но на который натолкнулись случайно.

Граничные значения

Помните историю про Золушку? Мы выделяем разные мешки (гречка, рис, ячмень, пшено), то есть разные классы эквивалентности. И у каждого из мешков есть свои границы, иначе крупа бы просто рассыпалась!

Эти границы очень важно проверять... Когда мы можем это сделать? В случае с мешком гречки граница вроде бы есть, но как ее проверить? Это же не конкретное число или значение. Другое дело, если речь идет о цифрах. Там с границами всё проще!

Границы на числовой оси

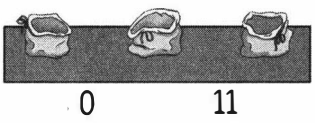
Допустим, мы тестируем интернет-магазин, где для бесплатной примерки можно заказать не более 11 вещей за раз.

У нас получается две границы:

- 0 — так как отрицательного количества вещей не бывает;
- 11 — ограничение сверху.

Между ними — корректный интервал количества вещей (один мешочек с крупой).

Но если посмотреть на числовую ось, то мы увидим, что есть еще два интервала, еще два мешка. С точки зрения системы это «плохие» интервалы, которые она не должна обратывать. И нам надо проверить: а что если указать такое значение? Можно ли это сделать в принципе? Понятным ли будет сообщение об ошибке?



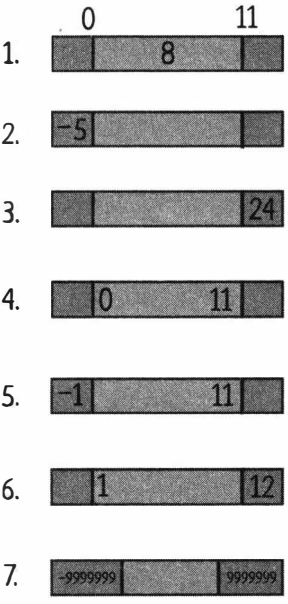
Очень важно также проверить пограничные значения. Чтобы убедиться, что граница именно в этом числе. Потому что, возможно, разработчик ошибся и поставил «от 0 до 12 вещей», а не до 11. Если проверить только границу, этот баг мы никогда не найдем.

А еще стоит заглянуть далеко-далеко вперед и поискать границу там. Иначе может быть, что разработчик обработал условие по ТЗ, и все работает. Но если какой-то вредитель случайно или специально вставит в поле большое значение — система сломается, так как она не готова к большим данным. Разработчик поставил тип данных поля — integer[54], подумав, что никто большее значение не введет. Ну и всё!

Таким образом, зная ТЗ: «максимальное количество вещей — 11 штук», мы проверяем:

- ○ внутри диапазона;
- ○ слева;
- ○ справа;
- ○ границы;
- ○ граница −1;
- ○ граница +1;
- ○ далеко-далеко...

Проверяем:
1. Внутри диапазона
2. Слева
3. Справа
4. Границы
5. Граница −1
6. Граница +1
7. Далеко-далеко



Границы в нецелых числах

Отдельно хочу рассмотреть нецелые числа. Особый интерес представляют значения, близкие к граничным. При приближении к границе на расстояние меньше точности вычислений мы можем попасть в ситуацию, когда проверки значения на допустимость успешно проходят, но вычисления завершаются неуспешно.

Проще говоря, система может округлить ваше значение, что приведет к беде. Например, мы исследуем уравнение $Y / (X - 1)$. Как система себя ведет при разных значения X:

- 0 — нормально поделилось;
- 2 и более — тоже все ОК;
- 1 — ошибка (деление на ноль). Разработчик ошибку предусмотрел и обработал: пользователю выдается нормальное сообщение об ошибке;
- 0,99 — все норм.;
- 0,99999999999999999999999999 — УПС, ВСЕ ПРОПАЛО! Система округлила это значение до 1 и получила деление на ноль. Но такую ситуацию разработчик не предусмотрел, и пользователь вместо внятного сообщения об ошибке видит «синий экран смерти»;
- 1,01 — все норм.;
- 1,00000000000000000000000000001 — КОШМАР, ОПЯТЬ НА НОЛЬ ДЕЛИМ!

Контрольные вопросы: что будет, если ввести значение X,99999999999 (после запятой 11 девяток)? А если X,00000000000000000000000001? Что будет, если разделитель «запятая»? А если «точка»?

Задавайте себе эти конрольные вопросы при тестировании значений, где можно вводить дробь.

Границы там, где «нет числа»

С числовым полем (количество платьев в заказе, возраст участника и т. п.) проблем обычно не возникает. Вот у нас в ТЗ границы уже указаны, так что проверяем их и пограничные значения — всё норм.

А вот если мы тестируем что-то другое, где, на первый взгляд, нет числа, сразу ступор. Как искать границы?? Мой вам совет — ищите число. И тогда все сразу станет просто:

- строковое поле — у него есть длина! Ее и проверяем на границы;
- авторизация — у нас есть состояния «пользователь авторизован/нет». Фактически, это «1/0», что тоже число;
- загрузка отчета — а что если в нем очень мало или много строк? Опять нашли число;

- строим в игрушке бассейн — его размеры тоже числа!
- и так далее...

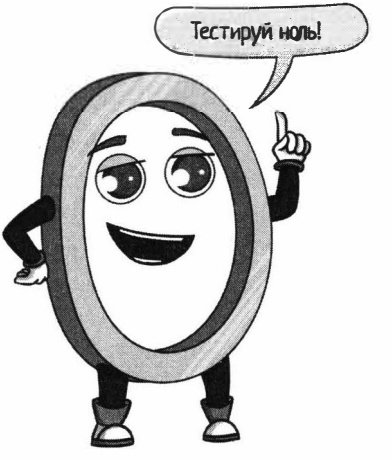
Впрочем, не везде вы сможете найти границу. И это нормально. Не надо высасывать ее из пальца и искать там, где точечной границы нет (как мешок с гречкой — это не одна точка и не одно значение).

Ноль

Особое внимание я хочу уделить проверке нуля[55]. Это как раз та волшебная «серебряная пуля», которую можно (и нужно!) применять всегда и везде. Тестируете что-то? Найдите там ноль!

И это не только число «0» в числовой строке. Это может быть:

- длина строки — обязательно пробуем оставить строку пустой;
- количество строк/колонок в файле;
- размер файла (максимально близко к нулю);
- авторизованность в системе — да / нет (1 / 0)
- ноль на выходе — система как-то обрабатывает данные? Что если обработать она их не сможет?



Здесь очень часто встречают баги. Особенно если это не ноль в прямом понимании слова (число в числовом поле), то разработчик просто забывает обработать такую ситуацию. А вы не забудьте проверить!

Типы границ

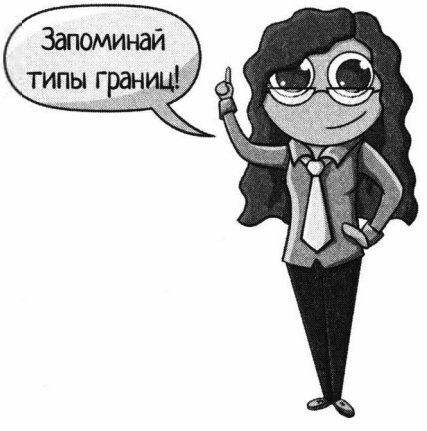
Про типы границ я впервые услышала на тренинге Алексея Баранцева (уже не помню точно, каком, я их много прошла). Алексей вывел собственную типизацию границ:

- **физическая** — которую физически нельзя преодолеть;
- **логическая** — ограничение, накладываемое логикой, не программой;
- **технологическая** — ограничение, накладываемое используемой технологией;
- **произвольная** — ограничение, наложенное аналитиком или заказчиком.

Лично я предпочитаю совмещать физическую и логическую. Потому что физическая — это то, что мы вообще преодолеть не можем. Например, при всем желании мы не введем строку отрицательной длины, ну никак не сможем. Но то что физически сделать нельзя, часто в программе сделать можно. Например, ввести в количество участников митапа «1,5 человека» — физически невозможно, но программа-то позволяет. Значит, для программы это уже логическая граница, мы же понимаем, что это невозможно.

Так что в моей классификации есть всего три типа границ (мнемоника[56] ЛТП):

- **логическая** — ограничение, накладываемое логикой, а не программой. Те аксиомы, что мы знаем с детства (в минуте 60 секунд, возраст меньше 0 быть не может, строка с отрицательной длиной тоже);
- **технологическая** — ограничение, накладываемое используемой технологией. Например, при попытке запихать в поле типа integer больше чем 2 147 483 647. Для поиска этой границы пытаемся вставить БОЛЬШОЕ число (или строку БОЛЬШОЙ длины). Помните, что на дворе XXI век. 1000 символов — это слишком мало для поиска технологической границы;
- **произвольная** — ограничение, накладываемое аналитиком или разработчиком. Например, «нельзя заказать больше 5 пар обуви за 1 раз». Это значение придумал заказчик, его легко изменить.



Я вам очень рекомендую ознакомиться с моей наработкой для проверки границ — мнемоникой БМВ[57]. Запоминается очень легко, позволяет проверить самое главное:

- **Б**ольшой — ищем технологическую границу, впихиваем невпихиваемое;
- **М**аленький — ищем около нуля;
- **В** самый раз — позитивный тест (он, конечно, должен идти в самую первую очередь, еще до того как мы исследуем границы или изучаем мнемоники).

Помните — баги чаще всего встречаются именно на границах. Разработчик может включить границу в два разных интервала или вообще никуда не включить. Чтобы найти границу, ищите число. А потом варьируйте его.

И всегда ищите ноль. Там нередко кроются баги!

Инструменты

Есть много инструментов, которые скрасят ваше тестирование. Они могут подкинуть идеи классов эквивалентности или помогут найти граничное значение.

Поиск технологической границы

Для поиска этой границы нужно вставлять в поле БОЛЬШУЮ строку. И есть куча инструментов, которые вам в этом помогут:

- Perlclip (Windows)[58] — тестовая утилита от Джеймса Баха, известного тестировщика. Работает на винде;
- Random string generator[59] (Mac, Linux, Windows) — генерирует строку онлайн, подходит для любой платформы!

Наверняка, есть и другие инструменты. Да и кто знает, что будет популярным через 5–10–15 лет? Гуглите и выбирайте инструмент под себя! Но генерите, пожалуйста, 100 млн вместо 1000 символов, иначе это будет уже не технологическая граница, а так, простая ;-)

Снятие ограничений на клиенте веб-приложения

Если вы тестируете веб-приложения, вам нужно освоить:

- консоль разработчика — на уровне «посмотреть консоль, нет ли там ошибок; посмотреть название элемента и снять ограничение длины через `maxlength`». Вот статья в помощь[60];

- плагин Web developer toolbar — я его использую в Firefox, он классный. Делает кучу всего, но для тестирования изучите вкладку **Forms**, чтобы снять границы на клиенте: **Forms | Remove Maximum Lenghts**.

Автоматическое заполнение полей

Заполнять поля вручную весело только в первый раз. А потом можно использовать плагинчики, которые сделают это за вас. А еще такие плагины подсказывают классы эквивалентности, что для новичка вдвойне полезно!

Мой топ-3 плагинов[61]:

1. Fake Filler — по одной кнопке заполнит поля сам рандомными значениями;
2. Web developer toolbar — тоже по одной кнопке, но значения всегда одинаковые, для обхода уникальности поля надо менять значение ручками;
3. Bug magnet — плагин для исследовательского тестирования, заполняет по одному полю, но дает много идей для тестов!

Типичные ошибки

Какие основные ошибки допускают студенты, когда пытаются применять классы эквивалентности?

Классы: символы, циферки, перемешал

Типичная попытка выделить классы эквивалентности для строкового поля:

- русские символы — проверки по полю (длинное/короткое имя, простое, распространенное...);
- английские символы — транслит, на английском, перепутали раскладку...
- символы — циферки, спецсимволы;
- перемешал — всё сказанное;

- длина поля — нижняя, верхняя границы, поиск середины...

Это плохо тем, что получается «серебряная пуля». Непонятно, ЧТО мы тестируем — такие проверки мы можем применить всегда.

Но нам надо тестировать, в первую очередь, конкретное поле и его бизнес-смысл. Поэтому для имени проверки будут одни, для адреса другие, а для паспорта — третьи.

Тестируйте конкретное поле, а не абстрактную строку!

Ищем границу сверху — аж 1000 символов

Проверили длину поля по ТЗ, а потом идет пункт «поиск технологической границы — ввести 1000 символов». Это несерьезно, XXI век на дворе. Вводите реально МНОГО.

Это хотя бы миллион. А лучше их 10 или 100 ;-) Используйте инструменты для генерации больших строк текста и ищите границу далеко, а не близко. Иначе вы можете пропустить баг: 1000 символов обработает система, а на 10 млн зависнет.

Если границы нет, в чек-лист не пишем

Попробовали вставить 1 млн символов в поле — система не зависла. Тогда перешли дальше, а этот тест даже не внесли в чек-лист. Но если сегодня бага нет — это не значит, что он не появится завтра. Вот только завтра его уже не найдут, пройдя по вашему чек-листу, — проверки-то нету.

Начинаем тестировать с длины строки

Это самые простые и понятные тесты: проверить поле на длину. Тут и классы эквивалентности, и границы, всё есть! Но это — не самые важные и приоритетные тесты.

В итоге система дает красивые ошибки на попытки «сломать» длину, но падает на корректном значении... Длину строки проверять можно и нужно, но только после того, как вы проверили бизнес-смысл поля.

Не тестируем пограничные значения

Границы всегда проверяем с двух сторон:

- тут еще можно;
- а тут уже нельзя.

Вместо «можно/нельзя» может быть «поведение 1/поведение 2». Но важно проверить, что граница не попала в оба интервала или что разработчик не поставил «больше» вместо «больше или равно».

Границы применяем лишь в длине строки

Если поле числовое, границы найти легко. Если выбираем числовой параметр строки (длина поля) — тоже всё понятно. А дальше ступор, кажется, что границ больше нигде нет.

Но число может быть и неявным:

- размер вольера для животных в игре;
- состояние авторизованности (1/0);
- переключение состояния (звук есть/звука нету, те же 1/0);
- размер окна браузера;
- ...

Для вдохновения можно почитать мои уже упомянутые ранее статьи «Класс эквивалентности „Ноль–не ноль“»[62] и «Мнемоника БМВ»[63].

Не проверяем нечисловые значения

Если вы тестируете числовое поле, то, помимо обычных чисел, обязательно попробуйте «не совсем числовые значения»: 1.0e2, 0xba, Infinity...

Идеи для тестов ищите в статье «Классы эквивалентности для строки, которая обозначает число»[64].

Вопросы для самопроверки

1. Что такое классы эквивалентности?
2. Для всех ли классов можно определить границу?
3. Какие типы границ бывают?
4. Можно ли найти границы НЕ в поле ввода? Как?

Домашнее задание

Банковская карта

Проверка номера банковской карты — как будете тестировать?
Наводящие вопросы:
1. Каким чаще всего бывает номер?
2. Каким он точно не бывает?
3. Можно ли его писать и слитно, и через несколько пробелов?
4. Что если пробелы будут в «неправильных» местах?
5. Что если в номере будут буквы?
6. Что если номер будет пустой?

7. Что если номер будет длиннее положенного?
8. Что если номер будет из сплошных нулей?
9. ...

Сортировка по строке

Есть система, в которой хранятся данные. По всем полям, выведенным в общий список, можно сортировать:

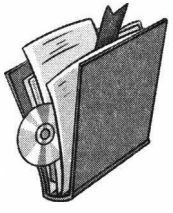
1) тыкаешь на стрелочки — сортируется по алфавиту в порядке возрастания;
2) тыкаешь еще раз — сортируется в порядке убывания;
3) тыкаешь третий — сортировка сбрасывается.

Вполне стандартный функционал.

Ваша задача — протестировать сортировку по строке. Скажем, по фамилии. Никаких ограничений на строке нет, это просто string.

Приведенный скриншот взят из приложения Folks[65], задачу сортировки решают мои студенты, так что поверьте, это очень полезно. На практике вижу ;-)

Ответ в конце главы не ищите, спойлеров для ДЗ с курса не даю, только возможность полезной практики.



Портфолио

Продолжаем делать портфолио. Расширьте свой чек-лист, созданный ранее, с помощью новых знаний о классах эквивалентности и граничных значениях. Потом перечитайте типичные ошибки моих студентов и исправьте их у себя ;-)

Ответы на вопросы для самопроверки

1. Что такое классы эквивалентности?

Значения, эквивалентные друг другу: какое ни выберешь, результат не изменится.

2. Для всех ли классов можно определить границу?

Нет, границы четкие есть только там, где есть число.

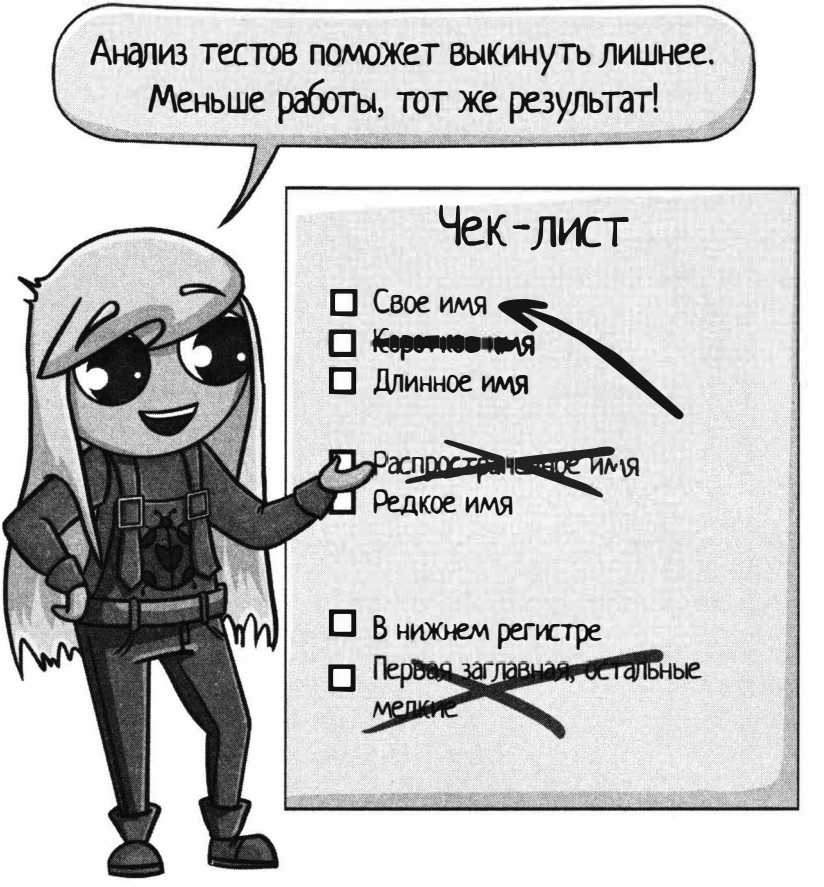
3. Какие типы границ бывают?

ЛТП: логическая, технологическая, произвольная.

4. Можно ли найти границы НЕ в поле ввода? Как?

Можно и нужно! Это может быть состояние (авторизован/нет, была миграция/нет), размеры окна браузера, количество одновременно работающих вкладок...

Глава 4
АНАЛИЗ ТЕСТОВ



В предыдущей главе мы учились расширять чек-лист. А теперь мы начнем... Его сокращать! Проведем анализ написанных тестов и выкинем все лишнее.

В этой главе:

- анализируем чек-листы;
- удаляем ненужное.

Разберемся с определениями

Наверняка во время гугления тестирования вы встречались со словами «тест-анализ». О нем ли здесь пойдет здесь речь? Чем различаются «тест-анализ» и «анализ тестов»?

Тест-анализ — это тот же тест-дизайн, только немного с другим уклоном. На самом деле их часто вообще не различают. Но тест-анализ ближе к работе аналитика и написанию требований:

- State-Transition Testing (Диаграмма состояний и переходов);
- Decision Table Testing (Таблица решений);
- User Case Testing (Варианты использования).

А то что ближе к автоматизации и комбинационному анализу, то есть уже собственно построение тестов, — это тест-дизайн.

Сведем это всё в таблицу (табл. 4.1).

Таблица 4.1. *Техники тест-дизайна и техники тест-анализа*

Техники тест-дизайна. Или все же тест-анализа?	
Boundary Value Testing Equivalence Class Testing Allpairs Algorithm testing Orthogonal Arrays Testing …	Классы эквивалентности Граничные значения Попарное тестирование Ортогональная матрица …
Техники тест-анализа	
State- Transition Testing Decision Table Testing User Case Testing …	Диаграмма состояний и переходов Таблица решений Варианты использования …

Анализ тестов

Анализ тестов — это не конкретная техника, а просто выкидывание лишнего из вашего чек-листа. Работа из серии «сесть и подумать»:

- какие проверки можно объединить?
- какие и вовсе выкинуть?

Что мы изучаем в книге?

Мы будем говорить сейчас именно про анализ тестов. В общем-то, его вполне можно назвать и «тест-анализом» — это просто игра слов.

Лично мне не нравится ситуация, когда два определения обозначают одно и то же. Не зря же их часто путают, правда? Поэтому я предпочитаю разделять понятия.

Путь эволюции чек-листа начинающего:

1. Первая попытка — когда узнал, что такое чек-лист, что такое позитивные и негативные тесты, попробовал накидать разных вариантов.
2. Расширяем чек-лист — группируем проверки, добавляем новые с учетом знаний про классы эквивалентности и особенно про граничные значения.
3. Анализируем результат и выкидываем лишнее.

А как иначе-то? Нельзя выкинуть лишнее, если выкидывать не из чего. Поэтому сначала делаем «много», а потом уже убираем ненужное. И хотя по идее это тот же тест-дизайн — ведь обычно мы убираем одинаковые проверки, сворачивая их в один класс эквивалентности... Но как тогда разделять «тест-дизайн на расширение» и «тест-дизайн на убирание лишнего»?

Я считаю, что фазу «делаем МНОГО» пропускать никак нельзя. Особенно для начинающих. Это потом

вы научитесь сразу делать условно-оптимальное количество тестов, отсеивая все лишнее в голове. Но в начале пути вам надо учиться видеть в поле конкретное поле, а не простую строку. Видеть, какие тесты можно применить именно под это поле. А потом уже подключать знание «да это просто строка» и выкашивать ненужное.

Поэтому в этой главе у нас именно анализ тестов. Хотя в самой книге мы рассматриваем техники тест-анализа, как его видят американские авторы или ISTQB[66]. Просто знайте — в терминах часто бывает путаница. Если готовитесь сдавать сертификат, обязательно посмотрите, что подразумевается под конкретным термином там.

Когда проходите собеседование и слышите незнакомые слова, не смущайтесь, попросите пояснить, что имеет в виду говорящий. Вполне возможно, вы знаете, о чем речь, даже проводили такое тестирование, просто называли его по-другому. И это нормально, я считаю. Мы еще вернемся к этому вопросу в *главе 13* про собеседования, а пока я просто обозначила свою позицию про определение тест-анализа.

Итак, в этой главе мы будем включать мозг и избавляться от лишних тестов. Не более того. Тем не менее тема очень важна, ведь у вас обычно мало времени на тестирование. Так что совмещаем то, что можно совместить, и выкидываем то, что можно выкинуть.

Как выкидывать лишние тесты?

Было бы здорово дать некий алгоритм, который поможет всегда и везде, но нет, увы. Универсальная фраза здесь только «сесть и ПОДУМАТЬ». А самое главное — вместе с водой не выплеснуть ребенка. Убирайте тесты аккуратно, особенно в первые годы работы. Возможно, выкинутое было отнюдь не лишним...

Но общий принцип анализа примерно такой:

1. Объединить позитивные тесты.
2. Выкинуть одинаковые классы эквивалентности.
3. Не тестировать один функционал 10 раз: проверять его в одном месте, а в остальных лишь то, что он в принципе работает.

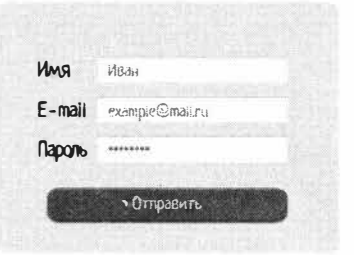
Рассмотрим каждый пункт по отдельности.

Объединить позитивные тесты

За одну проверку можно многое успеть. И если мы хотим ужать количество тестов, то можно провести несколько позитивных за один раз. Проще всего это сделать, когда надо проверить сразу несколько полей. Но не только!

Разные поля

Возьмем для примера форму регистрации. Предположим, на ней три поля поля: имя, email, пароль.



Для каждого поля мы расписываем проверки, которые надо провести. Например (табл. 4.2).

Таблица 4.2. *Пример проверки полей формы регистрации*

Имя	Email	Пароль
1. Мужское 2. Женское 3. Унисекс 4. Редкое 5. Распространенное 6. Длинное 7. Короткое 8. ...	1. Свой email 2. Email с точкой 3. Email с цифрами в имени аккаунта 4. Цифры в домене 5. Тире в имени аккаунта 6. Тире в домене 7. Email 10-минутный 8. ...	1. Простой 2. Сложный (защищенный) 3. Среднесложный 4. С пробелами 5. Русские и английские символы 6. Спецсимволы 7. Разный регистр 8. ...

Смотрите, за 7 тестов мы проводим фактически 21 проверку: 7 — на имя, 7 — на email, 7 — на пароль. Нет смысла проводить 21 тест, когда можно сделать 7. Но учтите, что объединяем мы только позитивные тесты.

Негатив комбинировать не надо. Оставили все поля пустыми — что будет? Ну ругнется система «заполните обязательные поля», а откуда вы знаете, какие именно она считает обязательными? Может, только «имя», хотя должна еще «email» и «пароль» считать.

Одно поле

В рамках тестирования одного поля тоже можно совместить разные проверки. Например, длину поля + регистр + алфавит:

1. Длинное имя + CamelCase («Ольга»: начинается с заглавной, потом нижний регистр) + русский алфавит.
2. Короткое на английском в CAPSLOCK.
3. ...

Разные параметры

Мне очень не хочется, чтобы вы думали, что объединять тесты можно только в больших формах ввода на *N* полей и больше это нигде работать не будет. Будет!

Мы можем объединять проверки на разные состояния объектов или разные параметры:

- файл: размер, количество строк, столбцов, формат;
- картинка (аватарка или фоточка в инстаграм): расширение, размер, высота, ширина;
- покупка вещей в интернет-магазине: категория, цвет, размер, фасон... (зеленое платье в пол 44-го размера, желтое короткое 42-го, голубое миди 50-го...);
- SOAP-запрос: разные параметры идут как разные поля на форме ввода.
- ...

Выкинуть дубли

Главное правило мозгового штурма: накидать идей БЕЗ редактуры, а потом уже редактировать, убирая лишнее/неосуществимое. Когда мы накидываем идеи для тестов, они могут повторяться, попадая в разные варианты проверок. Например, имя «Ольга» может быть в блоках:

- мужские/женские имена;
- разный регистр (CAPSLOCK, нижний регистр, ВерблюжийРегистр, Змеиный_регистр);
- русский/английский алфавит;
- длина поля.

Когда накидываем блоки для проверок, не останавливаемся для поиска дублей. Но потом можно пройтись по своему чек-листу и удалить лишнее.

И это на самом деле самое простое, но самое приятное, что мы можем сделать при анализе тестов. На это способен любой новичок. Причем начинающие некоторые дубли видят заранее. Вот, например, мои студенты обычно

еще до лекции с анализом тестов задаются вопросом: «Зачем в разделе „длина строки“ проверять нормальную длину? Мы же ее уже тестировали ранее, не получается ли дубль?».

Получается! И здорово, когда ребята это сразу подмечают. Но в школе я их прошу пока оставить дубли, потому что следующая лекция про анализ и домашнее задание как раз «возьмите свой чек-лист и выкиньте лишнее». Мы работаем в Ситечке[67] и в рамках анализа просим не удалять проверки физически, а поставить ненужным самый низкий приоритет. Сразу видно, сколько тестов выкинуто — это правда круто!

И я рекомендую делать именно в такой последовательности — нагенерили идей без фильтрации, потом уже фильтруем. Это очень важно особенно на начальных стадиях, пока вы в тестировании недавно, нет накопленного опыта, где вообще бывают баги, мало идей... Если еще и их контролировать, совсем ничего не придумаете.

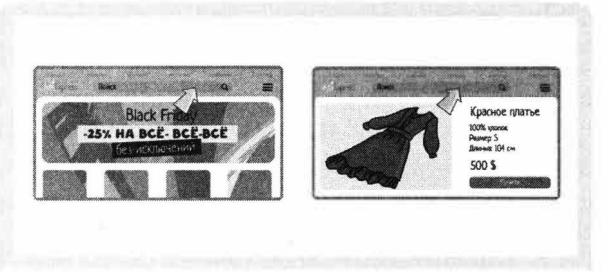
И опять же по опыту студентов: дублирование замечают заранее только в длине строки. Хотя в их чек-листах есть и другие его варианты. Поэтому сначала записываем, потом вычеркиваем. Сделаете так 5 раз, 10 — уже поймете, где тесты обычно дублируются, и исходно НЕ будете писать заведомо лишние проверки.

Не тестировать один функционал 10 раз

Когда вы тыкаете на кнопочки в интерфейсе, внутри программы задействуются те или иные участки кода. Хороший код — с грамотной структурой и переиспользованием вместо копипасты. Это значит, что разработчик пишет функционал в одном месте, а потом просто дергает вызов этого функционала из разных мест интерфейса.

Вот, допустим, поиск на каком-либо сайте. Вы открываете главную его страницу — справа вверху есть строка с поиском. Вы переходите на карточку красного платьишка — вверху осталась та же строка. Вы переходите на карточку синих брючек — строка поиска все еще есть.

Внимание, вопрос: нужно ли тестировать эту строку на каждой странице отдельно, снова и снова?



Строка поиска есть на каждой странице сайта.
Нужно ли тестировать её каждый раз с нуля?

Разумеется, нет. Мы понимаем, что это один и тот же функционал, который просто привязали к кнопке с лупой. Тыкаешь на лупу — запускается код поиска. И неважно, где эта лупа присутствует: на главной странице, в результатах прошлого поиска или на карточке конкретного товара.

Мы помним про тест-дизайн, про классы эквивалентности. Про пример с калькулятором, где можно писать сотни миллионов тестов только на одно лишь сложение, если не использовать классы эквивалентности (см. *главу 3*). Так и тут: если на сайте 10 000 товаров, то тестировать раздел навигации на каждой странице будет стоить дорого, а толку не принесет.

Типичные ошибки

Объединили негатив

Важное правило — негативные тесты объединять нельзя! Иначе как вы поймете, какая именно проверка сработала? Допустим, у нас на форме регистрации есть поля:

- Имя.
- Эл. почта.
- Пароль.

Все три — обязательные для заполнения. И вот я решаю провести один тест: оставляю все поля пустыми. Варианты развития могут быть разными:

1. Система делает проверки на обязательность и, сломавшись на первом же поле, останавливается. В итоге мы видим сообщение «Заполните поле Имя», заполняем его, снова нажимаем «Сохранить» и получаем следующую ошибку «Заполните поле Email», а потом то же для пароля. В итоге все равно провели три теста, сократить не удалось.
2. Система пишет около каждого поля: «Обязательно для заполнения», подсвечивая его красным. Но верно ли, что она сработала для каждого поля? Вдруг там в принципе сбой: если не заполнить имя,

подсвечиваются все поля. Или вдруг подсветка работает только тогда, когда незаполненных полей больше одного? А если только имя пустым оставить, она не загорится? Непонятненько!

3. Система пишет «Заполните обязательные поля», никак эти поля не подсвечивая. И звездочек около них нет. И ты сидишь весь такой в сомнениях, что от тебя вообще хотят. Это еще и баг документации — непонятное сообщение об ошибке. Хорошо, что мы его нашли, но проводить отдельные тесты все равно придется.

Итого: если объединить негативные проверки, нельзя четко ответить на вопрос, правда ли система умеет реагировать на каждую ошибку в отдельности. Может, она обрабатывает лишь одну из двух, и вам просто повезло ее увидеть. Объединяйте позитив, а негатив проверяйте отдельно, оставляя другие параметры «точно хорошими».

Понапихали в один тест всего и сразу

Применение техник тест-дизайна и тест-анализа — это здорово и круто! Но, пожалуйста, не надо усложнять.

Когда люди узнают про анализ тестов, про то, что можно сокращать количество тестов, объединяя несколько в один и выкидывая малоприоритетные, в головах устанавливается какой-то флажок «Меньше тестов = круче!». Иначе не знаю, как это объяснить.

К каким проблемам приводит максимальное уменьшение количества тестов?

1. **Писать тесты тяжело** — нужно держать в уме, что уже проверено, а что нет. По названию не отследишь.
2. **Проводить ревью тестов тоже сложно** — по той же причине, читая чек-лист джуниора, нужно держать в уме, что он уже проверил, а что еще нет.

3. **Описание непонятное** — а уж название тем более ;-) В него же не запихаешь 10 проводимых проверок.
4. **Пропускаются баги** — как следствие всего перечисленного. Рано или поздно сработает человеческий фактор. И именно та проверка, которую вы забыли провести, могла бы найти баг.
5. **Если нашли баг, его сложно локализовать** — а где именно не работает? Какой из 10 проверяемых параметров вызвал сбой? Иногда это очевидно из текста ошибки, но не всегда...

А как же анализ? Анализ тестов отлично применяется для комплексных тестов. Вот вы проверили каждый параметр и убедились, что сам по себе он работает. Теперь надо как-то комбинировать.

Но мы уже не проверяем каждый с каждым, а проверяем саму связку «И». Работает ли, когда у нас не один параметр, а два? А если максимум? Если знаем, что параметры влияют друг на друга, их тоже проверяем отдельно. Но все равно получается сильно меньше перебора «все со всем».

Сейчас я расскажу про pairwise. Если очень хочется сократить количество тестов, все равно распишите подробно по каждому параметру, а потом засуньте результат в pairwise. И вот вам и тестов мало, и все учтено.

А вообще стоит смотреть по ситуации, нужно ли сокращение базовых проверок. Одно дело — если их получилось 100, а времени есть только на 10. И совсем другое — если вы составляете автотесты. Один раз написал, и готово, дальше всю работу делает робот. И вот чтобы падения автотестов разработчик мог легко локализовать и исправить, они должны быть небольшими и простыми. Один тест — одна проверка, все по феншуй.

Техника pairwise

Техника pairwise применяется для тестирования попарных значений. Эффективна она тогда, когда у нас много параметров, но мало значений в каждом:

- Параметр 1: да / нет.
- Параметр 2: да / нет.
- Параметр 3: да / нет.
- Параметр 4: раз / два / три.
- Параметр 5: вкл / выкл.
- ...

При этом параметры взаимосвязаны, их результаты складываются или как-то влияют друг на друга. Например, фильтры в интернет-магазине или приложение, в котором можно установить целую кучу галочек. Вручную перебирать все варианты — скукота.

Pairwise позволяет сократить количество проводимых тестов, причем иногда в разы. Проводится автоматически: вы загружаете свои параметры в программу, на выходе получаете набор тест-кейсов. Удобно!

Инструменты

Самые популярные инструменты на момент подготовки книги:

- Allpairs — самый простенький, но есть под все платформы (включая macOS), очень удобно.
- PICT — имхо, он «красивее» показывает входные файлы (строки, а не столбцы) + позволяет задавать условия IF ;-)

Есть еще куча разных, ищите в гугле[68].

Для тех, у кого Мак — пара идей:

- Allpairs можно скачать отсюда: https://sourceforge.net/projects/allpairs/. Там python-скрипт, который запускается через терминал;
- сайт http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp.

Так же есть онлайн-тулза: **https://inductive.no/pairwiser-tool/**. Позволяет отметить отдельные пары как невозможные.

Плюсы и минусы метода

Плюсы

1. Сильно снижает количество проводимых тестов, особенно если есть много параметров и мало значений в каждом. Там, где сотня чек-боксов, полный перебор займет сотни тысяч вариантов, а pairwise выдаст 20 кейсов с большим покрытием.
2. Хорошее покрытие — найдет столько же ошибок, сколько и полный перебор. Но за меньшее время!

Минусы

Если тест нашел баг, его сложно локализовать. Непонятно, какая именно пара значений повлияла на ситуацию. С другой стороны, это часто оправданно...

Когда применять?

Когда мало времени, а много всего надо успеть ☺

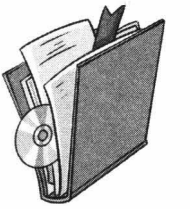
На самом деле pairwise часто применяется в автоматизации тестировании, когда за тебя работает робот. В ручном тестировании применяйте его там, где у вас много параметров, но мало значений в каждом.

Еще как вариант — тестирование совместимости. Проверить все браузеры на всех платформах — будет очень долго. Покрытие через pairwise в таком случае будет оптимальным решением.

Вопросы для самопроверки

1. Что такое тест-анализ по ISTQB и чем он отличается от тест-дизайна?
2. Как можно уменьшить количество тестов?
3. Когда НЕ надо схлопывать тесты?

Портфолио



Продолжаем делать портфолио. В предыдущей главе мы расшили свой чек-лист, а теперь будем его ужимать! Пересмотрите свои проверки и:

1. Расставьте приоритеты (что в первую очередь проверять, что в последнюю).
2. Выкиньте лишнее.

Для наглядности можно сделать два файла:

- в одном вы просто подкрашиваете красным все выкинутое и подписываете, почему выкинули проверку;
- во втором вы удаляете все эти строки, и у вас остается готовый чек-лист для портфолио! Где все разбито по блокам, есть тесты на ваш функционал, а не просто серебряная пуля «символы-циферки-перемешал», где нет ничего лишнего... Одним словом, красота!

Ответы на вопросы для самопроверки

1. Что такое тест-анализ по ISTQB и чем он отличается от тест-дизайна?
То, что ближе к Коберну[69] (State-Transition Testing, Decision Table Testing, User Case Testing), — это тест-анализ. А то, что ближе к комбинационному, то есть уже собственно построение тестов, — это тест-дизайн.

- ✧ Тест-анализ граничит с аналитикой.
- ✧ Тест-дизайн — с автоматизацией.

2. Как можно уменьшить количество тестов?
Объединить позитивные тесты, выкинуть дубли, проверять функционал один раз, а не 10.

3. Когда НЕ надо схлопывать тесты?
Когда итоговый тест получается слишком сложным для восприятия. Или когда мы смешиваем разный функционал (тесты на данные и тесты на функционал).

Глава 5
БАГ-ТРЕКИНГ

Писали-писали мы тест-кейсы и чек-листы. И ведь наверняка уже находили баги, пока применяли их на реальном проекте! Пора научиться оформлять ошибки.

В этой главе:

- разбираемся с определением бага;
- знакомимся с инструментами баг-трекинга;
- учимся локализации ошибок;
- оформляем четкие и красивые баги.

Что такое баг?

Баг

Разумеется, у вас так и чешутся ручки поскорее побежать оформлять найденные в процессе составления портфолио ошибки. Но, перед тем как мы будем говорить об оформлении багов, сначала нужно разобраться с его определением.

Если вы читали Романа Савина, то можете сходу сказать: «Баг — это отличие фактического результата от ожидаемого!

Но минуточку... Подумайте, чем такое определение плохо? Правильно — тем, что ожидать вы можете все что угодно, хоть космолет:

- баг на главной странице гугла: я ожидаю там котика видеть каждый день, обязательно няшного!
- баг в интернет-магазине: после регистрации мне не положили бонусные рубли на счет, а я-то ожидала;
- баг в Скайп: я ожидаю при каждом сообщении в любой чат перевод 1000 долларов на мой счет;
- ...

Получается, нужно уточнять, кем результат ожидается. Например, заказчиком или иным лицом из целевой аудитории (ЦА). И почему он этого ожидает. Ведь мы либо ожидаем какой-то функционал заранее, и он есть в ТЗ, либо текущее поведение доставляет ***проблемы***. Нет проблемы? Ну так и пусть дальше работает как работает, делов-то...

> Баг — это проблема для лиц, чье мнение имеет для нас значение.

А сказал это Джеймс Бах (он, в свою очередь, ссылается на Джерри Вайнберга)[70].

Группа людей, чье мнение имеет значение, очень сильно разнится от проекта к проекту. Поэтому важно вывести именно критерий, на основе которого мы учитываем или не учитываем чьи-то проблемы. И чтобы знать, как выявлять эту группу людей.

«Отличие реального результата от ожидаемого» — да, имеет право на существование. Когда у вас есть четкое ТЗ, вы проверяете по нему, и вдруг реальность отличается. Баг? Баг!

Но только ТЗ бывает не всегда. И в нем могут быть ошибки. И в нем может быть что-то не описано. Поэтому такой вариант ответа «Что такое баг?» является лишь частью всего подмножества реальных багов.

Баг — это проблема для лиц, чьё мнение имеет для нас значение (Джеймс Бах)

«Отличие реального результата от ожидаемого» — да, когда у вас есть чёткое ТЗ, вы проверяете по нему, и вдруг реальность отличается. Баг? Баг!

Но ТЗ бывает не всегда. И в нём могут быть ошибки. Или что-то не описано. Сравнение с ТЗ — только часть всего подмножества реальных багов

Иногда баг может быть признан багом, но ему ставится статус «не будет исправляться» — такое бывает чаще всего, если уж слишком велики затраты, а проблема не столь серьезна. И даже если куча пользователей недовольна, это не всегда повод вносить исправления.Хотите пример?

Моя коллега Ольга Алифанова много лет проработала в игровой индустрии. Она и гайды игрокам писала, и админом была, и ГМ, и тестировщиком. В том числе и в компании, где делают многомиллионные по количеству пользователей игры. Оля привела такой пример:

> Компания гордится историчностью своих кораблей и танков. Все флаги в игре реальные, на японских кораблях даже есть хризантемы на носу, как и у реальных их прототипов.
>
> Однако в китайской версии игры хризантем нет, а все флаги фальшивые. И игроки могут до посинения и хрипоты ныть, что это неисторично и баг.
>
>
>
> Но это не баг, и исправлению не подлежит. Китайское законодательство запрещает использование нацсимволики в играх, и уж тем более японской.
>
> Однако даже одна жалоба на отсутствие хризантемы в русской версии игры, и это будет оформлено как баг и исправлено — в России такого требования нет.

Количество нытиков может, конечно, влиять на вопрос, надо ли исправлять эту проблему, но это совершенно не обязательно так. И если ПО,

предназначенное для профессионального использования, неясно тому, кто с мороза ворвался, — это их личная драма.

Нашел баг, что дальше?

Процесс баг-трекинга

Баг-трекинг — это заимствованное из английского языка слово. Идет от bug tracking — процесс слежения за багом. Вы ведь хотите, чтобы ваши баги исправлялись? Тогда за ними надо следить!

Если в компании не используется система отслеживания ошибок, то процесс выглядит примерно так: тестировщик нашел ошибку, сообщил разработчику. Тот быстренько исправил и отдал новую версию на тестирование. Такое вполне возможно в небольших компаниях, почему нет?

Подход хорош... Но есть и минусы:

1. На каждый вопрос ты отвлекаешь коллегу от работы. И это колоссальная потеря времени. Почитайте книгу Фредерика Брукса «Мифический человеко-месяц» — это ведь только кажется, что вопрос на две минуты, а человек потеряет нить размышления и потом может полчаса вспоминать, на чем остановился и что делать дальше. Прерывания — зло.
2. Разработчик сказал «Да, это баг, сейчас исправлю», но... Забыл. А если еще и вы забыли, то ой ;-)
3. Или наоборот, разработчик вышел покурить, когда вы баг нашли. Пока вернулся, вы уже про баг забыли, увлеклись новой задачей. А баг-то остался, и он не исправлен!

Все-таки «самый тупой карандаш острее самого острого ума» © Записывайте! А куда записывать? В инструменты баг-трекинга (отслеживания задач).

Инструменты

На самом деле в качестве инструментов баг-трекинга можно использовать все что угодно:

- гуглодоку;
- ворд;
- скайп;
- почту;
- ...

И я абсолютно серьезно это говорю. Про гуглодоку слышала, а ворд и почту сама использовала. Но всё же есть специализированные инструменты баг-трекинга, в которых работать намного удобнее:

- вы видите, в каком статусе проект сейчас: сколько задач всего, сколько из них в разработке, сколько в тестировании;
- вы видите всю историю бага: с чего всё начиналось, чем закончилось, что нашли в процессе;
- вы видите вообще все найденные ранее баги — а история тоже бывает полезна;
- можно настроить инструмент под себя.

Итак, сначала список удобных и «красивых» инструментов[71]:

- JIRA — платная, но крутая;
- Redmine — основной конкурент, бесплатный;
- Youtrack — чем-то похож на JIRA, но менее распространен.

Менее удобные, но относительно популярные (по состоянию на 2021-й год):

- Mantis — в целом не так уж плох, но не сравнится с JIRA или Redmine;
- Bugzilla — она вообще не конкурент, но если не верите... Тут даже отредактировать задачу нельзя, только новый комментарий написать;
- Trac — как по мне, так вообще убожество, там надо писать в HTML-разметке, а зачем заморачиваться, если даже в гуглодоке и то про это думать не надо?

Если у вас еще нет никакого инструмента на работе, предложите установить какой-нибудь. Лично я фанатка JIRA, она мне кажется простой и удобной. Тем более, что в облачной версии все можно создать/настроить в пару щелчков и работать «из коробки». А вот чтобы редмайн стал удобным, с ним надо повозиться администратору...

В любом случае вы можете попробовать парочку разных инструментов, чтобы сделать выбор. Какой понравится, тот и ставьте. Обычно в платных инструментах есть trial-версия (попробуй бесплатно 1 месяц), используйте ее.

Ну или тестовые площадки, находящиеся в открытом доступе. Например, на Testbase есть такие: **http://testbase.ru/test-it**. Все равно мне нужны были инструменты для моего курса — дать студентам «пощупать». Почему бы не выложить их в общий доступ? Конечно, мои площадки действуют на момент подготовки книги, и кто знает, что будет лет через 5–10... Инструменты могут устареть или закрыться, открытый доступ может закончиться...

В общем, если есть где «пощупать» — используйте варианты! Они обычно легко гуглятся. Да и trial-версии никому не мешают. Пробуйте, выбирайте — всё в ваших руках.

Workflow: жизненный цикл задач

Жизненный цикл задачи — это то, по каким статусам она проходит от момента заведения до полного исправления, проверки и закрытия. В каждом баг-трекинге есть стандартный Workflow, но его всегда можно поменять под свои нужды.

Учить его не нужно! И это в целом бесполезно — ведь на каждой работе свои привычки. Где-то разработчик сразу закрывает задачу, а где-то переводит на тестировщика. Но давайте я вам на пальцах объясню, как это работает.

Допустим, тестировщик нашел баг в системе. Он его грамотно оформил (как это делать, мы поговорим чуть позже) в системе баг-трекинга. Это как если бы он написал его на стикере, просто этот стикер — в компьютере.

Дальше тестировщик «вешает» задачу на разработчика, то есть указывает его исполнителем. И когда разработчик открывает систему баг-трекинга, он видит все стикеры, что висят на нем. Это работает как напоминание «тебе нужно сделать задачи 1, 2, 3...».

Разработчик, выполнив задачу, перенаправляет стикер тестировщику:

— Я сделал, проверяй!

И теперь уже стикер служит напоминалкой тестировщику: «Не забудь меня проверить!». Вот и всё ;-)

Ну а если вы хотите узнать подробнее о Workflow задач, то найдите статью «Жизненный цикл (Workflow) задач»[72].

Как заводить задачи в баг-трекер?

Если вы заведете задачу плохо, на нее просто «забьют». ОЧЕНЬ важные качества тестировщика — знание русского языка и умение формулировать свои мысли. Учитесь ставить такие задачи, которые будут исправлять. Или хотя бы обсуждать JO Баг — ошибка в программе.

O Улучшение — все OK, но хотим с перламутровыми пуговицами.

O Новая разработка — такой возможности нет, а очень хочется.

Допустим, заказчик захотел новую возможность, а вы завели ее не как новую возможность, а как баг. Разработчики весь месяц делали другие новые функции и до вашей не добрались. Заказчик в ярости: вы же обещали... А виноват постановщик задачи — умей выбирать тип (табл. 5.1)!

Таблица 5.1. *Типы задач*

Баг	Улучшение	Новая разработка
Ошибка 500 при загрузке XLXS-файла	Загружать файлы драг-н-дропом	Возможность грузить сразу несколько файлов
Город «Москва» исправляется на «Масква»	Выводить код ФИАС в результатах разбора адреса	Обработка иностранных адресов
Нет подсказок по букве «Б» на английской раскладке	Распознавать непереключенную раскладку в подсказках для email	Транслитерация в подсказках
При загрузке файла большого размера сообщение об ошибке «некорректный тип»	Увеличить ограничение на размер загружаемого файла до 30 Мбайт	Загрузка файлов формата DJVU

Осторожнее с ошибками. Разработчик не любит слышать, что в его детище что-то не так. Он будет топать ногами и кричать: «Это не баг, и исправлять его не надо». Делайте «финт ушами» — ставьте улучшения вместо некритичных ошибок.

Да, это немного утрировано. Но помните: первая защитная реакция — это нападение. Это лимбическая реакция организма, ничего не поделаешь.

Представьте, что вы сделали домашнее задание из этой книги. Например, написали тест-кейс. Красивый такой! Большой! Целый час старались, сначала

написали, потом прошлись по всем типичным ошибкам, нашли у себя, исправили... Гордость за свое детище неимоверная!

И вот вы показываете ее другу или коллеге, а тот кривит нос и говорит: «Это что за хрень?!». Обидно? Да! Первая реакция — начать защищаться. Вы возмущаетесь:

— Да почему?!

— Тут же стена текста, ничего не понятно!

— Да все понятно, это просто ты дурак!

Системный архитектор реагирует на появление ошибки

Первая защитная реакция — это нападение

Критику сложно воспринимать с благодарностью — это отдельное умение, которому нужно учиться. Я бы посоветовала начать со статьи Максима Ильяхова «Критики и ненавистники»[73].

Но ведь критиковать можно по-разному. Недаром конструктивной критикой называют: «Сначала похвали, а потом предложи, что улучшить». Сравните предыдущий диалог и следующий:

— Посмотри какой я тест-кейс написал!

— Ух ты, как круто! Молодец!!! А как подробно всё, никакие уточнения не понадобятся для прохождения! Я бы еще разбил текст на абзацы, а то получается wall of text, читать сложно...

— Хм, да, пожалуй, ты прав... Сейчас поправлю!

Так же и с разработчиками. Некоторые воспринимают постановку бага как плевок в душу: «Ты написал ПЛОХОЙ код!». Не могу сказать, что это очень адекватное поведение, но иногда и правда улучшение мелкое сделают охотно, а оформленное как баг закинут в дальний угол: «С глаз долой, из сердца вон!».

Конечно, не надо доходить до крайностей и заводить явную ошибку (падение системы, орфографический баг в тексте) как улучшение. Но про финт ушами помните!

Локализуйте проблему

Локализация — это поиск первопричины. Не торопитесь заводить конкретную проблему — это ниточка. Потяните за нее и распутайте весь клубок. Если этого не сделать, разработчики исправят последствия, а исходная проблема останется.

Когда мы тестировали определение пола по ФИО, то обнаружили, что «Гунько Александр» — женского рода. Первая реакция — бегом ставить баг! Хотя нет, постойте... В чем тут проблема?

- Система всех «Гунько» считает женщинами?
- Или «Гунько Марина» тоже сменит пол?
- А «Иванов Петр» — мужчина?

Это и есть локализация — докопаться до причины проблемы.

Придумайте короткий и емкий заголовок

Упоротый менеджер в 12 ночи просматривает баг-трекер, и он должен по заголовку понять, насколько критично исправить проблему.

Если он пропустит серьезную ошибку из-за непонятного названия, утром полетят головы. Если он поднимет панику из-за несерьезной задачи, он выставит себя дураком. И сильно обидится на того, кто ставил задачу. Если он не поймет из названия, в чем проблема, задачу закроют, даже если там реальный косяк. Соизмеряйте важность проблемы и название задачи (табл. 5.2).

Упоротый менеджер в 12 ночи просматривает баг-трекер, и он должен по заголовку понять, насколько критично исправить проблему

Таблица 5.2. *Соизмерение названия задачи и важности проблемы*

Нет	Да
АААААААА, КОШМАР, ВСЕ ПРОПАЛО, НИЧЕГО НЕ РАБОТАЕТ!	Авторизация через твиттер падает с HTTP 500
В исследуемой системе при вводе в поле «Имя» символов русского алфавита, английского алфавита, спецсимволов и символов в неправильной кодировке поведение программы неверное	Падает отправка письма в кодировке UTF-08
Двойные имена	Нет подсказок по двойным именам в ФИО

Если в заголовке появились слова «Ошибка», «Неправильный», «Некорректный», «Неверно» — перепишите заголовок. Вы заводите задачу с типом «Ошибка» — уже понятно, что что-то работает не так. Объясните проблему: «Можно зарегистрироваться с именем Ктулху». Но если вы заводите улучшение, заголовок должен предлагать, а не ставить перед фактом: «Запретить регистрацию с именем Ктулху» (табл. 5.3).

Таблица 5.3. *Ошибка vs улучшение*

Баг	Улучшение
Можно зарегистрироваться с именем Ктулху	Запретить регистрацию с именем Ктулху
Сообщение об ошибке указывает на неверный пароль при вводе недопустимого имени	Выводить в сообщение об ошибке детальную информацию по причине
Нет подсказок по двойным именам в ФИО	Выводить подсказки по двойным именам в ФИО

Окончание таблицы

Баг	Улучшение
Можно зарегистрироваться с паролем 123	При регистрации добавить проверку небезопасных распространенных паролей
Нельзя загружать несколько файлов сразу	Обрабатывать загрузку нескольких файлов

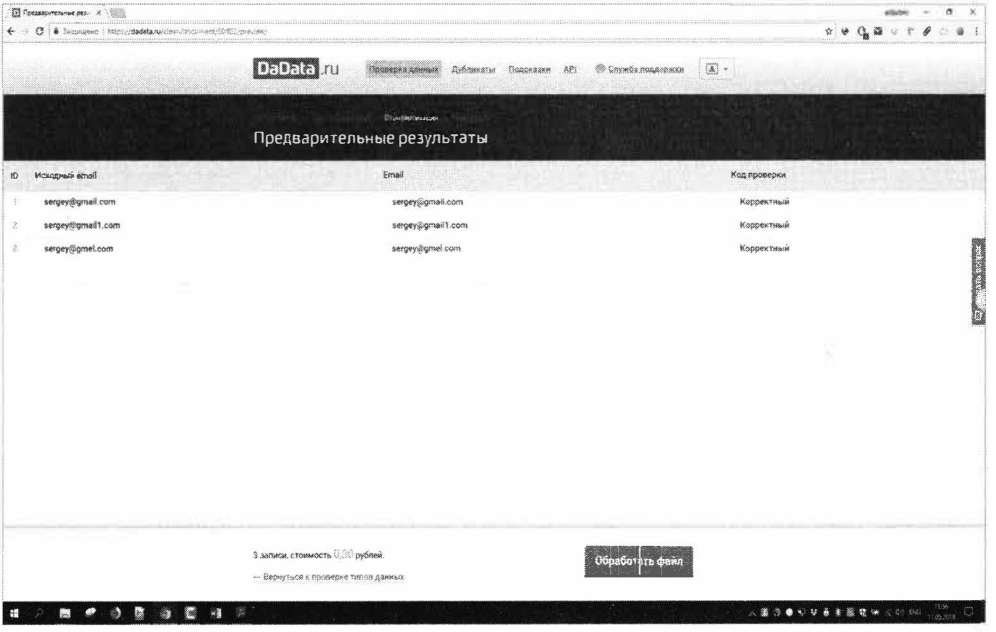
Приложите скриншот

Первое, что цепляет взгляд — это картинка. Иногда разработчику достаточно названия и картинки, чтобы понять, где проблема.

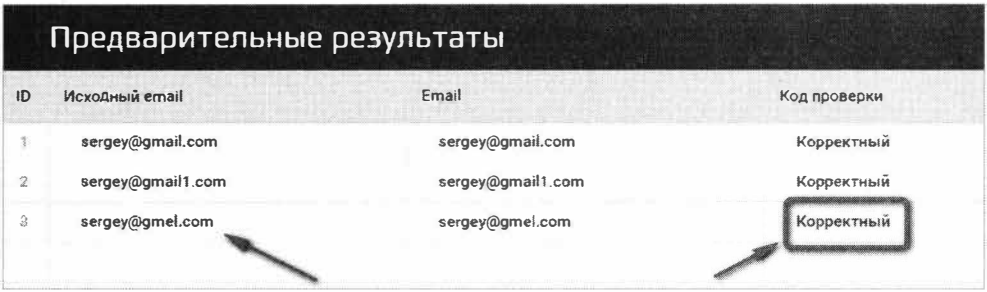
Когда ставишь задачу, скриншот делать лень. Кажется, что и без него все понятно. Но баги не всегда исправляются сразу. Спустя месяц изменится интерфейс, и без скриншота будет непонятно, о чем речь в задаче. Картинка поможет тестировщику увидеть, «как было до того», чтобы сопоставить с настоящим.

На картинке не должно быть ничего лишнего. Если нужна только маленькая часть экрана — прикладывайте ее, а не весь рабочий стол. Если скриншот получается большим, выделите в нем область, на которую надо смотреть, — например, нарисуйте стрелочку.

Лучше не так:



А так:



Опишите шаги воспроизведения и результат

Если названия и скриншота недостаточно, разработчик читает шаги воспроизведения. Что ему нужно сделать, чтобы увидеть проблему?

Шаги — короткие, емкие и последовательные. Разработчик должен выполнить их, а не задумываться над тем, что и в каком порядке ему делать.

Если шаги непонятные, разработчик не станет в них разбираться. Ему это не нужно. он закроет задачу: «не воспроизводится». Увы, на баг «забили»!

Все знают, что шаги должны быть короткими и емкими. На практике мои студенты ударяются в две крайности: слишком кратко или слишком длинно (табл. 5.4).

- Слишком кратко — когда нужно тратить время, чтобы понять, о чем речь. Студенты думают, что у разработчика уже открыт сайт на нужной странице, поэтому пишут просто: «Сделай так!». Но разработчик может вести сразу несколько проектов, и такие шаги ставят в ступор. Где сделай? Что за проект? В каком месте системы этот раздел находится? Где ссылки в конце-то концов??
- Слишком длинно — когда поленились локализовать. Бродишь по сайту, отчаявшись найти проблему, и вдруг... ОШИБКА! Первое стремление — скорее, скорее ее завести. Кажется, что *все* действия крайне важны. Даже если они не имеют реального отношения к проблеме. Ведь так воспроизводится? Заводим!

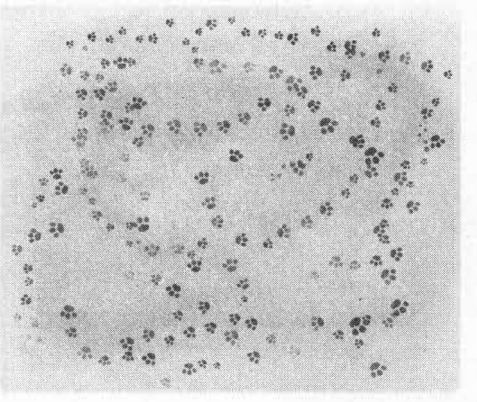
Таблица 5.4. *Слишком кратко vs слишком длинно*

Нет	Да
Загрузить файл с опечатками в email-ах	1. Авторизоваться в системе: **https://dadata.ru/#authorization_popup** (логин: abc, пароль: 1). 2. Загрузить файл с опечатками в email-ах (см. пример в аттаче: emails.xlxs) 3. Нажать «Просмотреть результат»

Окончание таблицы

Нет	Да
1. Открыть сайт Дадата. 2. Авторизоваться в системе. 3. Открыть раздел «Подсказки». 4. Открыть раздел «ФИО». 5. Ввести: `Киселева Ольга Евгеньевна`. 6. Присесть 20 раз. 7. Поплясать с бубном. 8. Переключиться на вкладку «Организации». 9. Ввести `ОАО Ромашка`	1. Открыть подсказки по организациям: **https://dadata.ru/suggestions/#party**. 2. Ввести: `ОАО Ромашка`

Как для разработчика выглядят шаги воспроизведения бага



Обоснуйте ожидаемый результат

Раз вы ставите баг, значит, вы считаете, что в системе есть проблема. Но почему это проблема? И для кого? Насколько она реальна?

Недостаточно сказать: «Поле email должно быть рассчитано на 23 символа» — а с чего вы это взяли? Вспомните, что такое баг, и обоснуйте свои ожидания.

У разработчика и тестировщика разные взгляды на проблемы. То, что мешает тестировщику, разработчик считает нормальным поведением. И если его просто ставить перед фактом: «Это баг, исправляй!», пощады не ждите. Разработчик не станет бегать за автором с вопросом: «А почему это проблема? Объясни, пожалуйста!» Он просто закроет баг.

Опишите сценарий использования, в котором возникает ошибка. Покажите, что в другом похожем месте система ведет себя по-другому (неединообразно — плохо!). Дайте пруфлинки (табл. 5.5).

Таблица 5.5. *Описание сценариев использования*

Нет	Да
Увеличить поле «Имя» до 30 символов	Увеличить поле «Имя» до 50 символов, так как туда часто вводят ФИО целиком, и оно обрезается
Исправлять непереключенную раскладку в подсказках по ФИО	Исправлять непереключенную раскладку в подсказках по ФИО, как это уже сделано в подсказках по адресам и организациям. Сейчас система работает неединообразно
Адрес «Россия, Москва, Большая Пироговская улица, 37-43кб» разбирается корректно	Адрес «Россия, Москва, Большая Пироговская улица, 37-43кб» разбирается корректно, потому что такой вариант записи диапазонных домов нормален. И этот адрес существует, см: **http://maps.yandex.ru/-/CVGlMR3g**

Что еще?

Достаточно ли приведенных шести пунктов? На самом деле нет. Они охватывают минимальный набор полей:

- тип;
- название;
- описание.

У вас же на работе может быть еще целая куча дополнительных! Про основные из них я расскажу.

Приоритет

Это может быть одно поле, а может быть два разных: Priority и Severity. Иногда их совмещают, иногда нет. Что эти поля значат?

- Priority — насколько критичен баг для бизнеса.
- Severity — насколько критичен баг сам по себе, с технической точки зрения.

Шкала может быть разной — от простой:

- Minor;
- Major;
- Critical.

До более сложной и насыщенной. Если два поля разнесены, то тестировщик обычно заполняет только Severity, а Priority определяет менеджер.

Кажется, что смысла в двух шкалах нет и они всегда будут совпадать. Это неверно. Вот примеры совершенно разных приоритетов:

1. Опечатка в имени спонсора или вашего сайта на главной странице:
   - ✧ Severity — самый низкий, текстовый баг.
   - ✧ Priority — самый высокий, блокер, страдает репутация.
2. Краш приложения[74] при выполнении комбинации очень сложных условий (подпрыгнуть три раза, притопнуть, прихлопнуть, пройтись на голове...):
   - ✧ Severity — crash, приложение все же падает.
   - ✧ Priority — самый низкий, потому что таких случаев 1 на 1 000 000.

Если вы сами выставляете приоритет задачи (иногда это делает менеджер) — не увлекайтесь. Понятное дело, что для своей задачи хочется всегда поставить высокий приоритет. Ведь это же такой страшный баг, надо срочно исправить! Особенно когда речь идет о новичках ;-)

> Представьте, что вы пришли в магазин за продуктами. И оказалось, что банковские карты временно не принимаются, банкомата рядом нет, а налички очень мало.
>
> И вот тут начинается приоритезация. Что действительно важно купить? Кошке корм хотели, но он еще есть, зато молока совсем нету, лучше взять его. Высокий приоритет получают те продукты, которые важно купить именно сейчас. А всё, что можно отложить на потом (завтра/в среду/через неделю), — это уже невысокий приоритет.

Аналогично в багах. Высокий приоритет — это когда задачу нужно исправлять срочно! Вот у нас на курсе для новичков ребята тестируют реальные приложения, то есть PROD (система, с которой работают реальные пользователи).

И если они находят баг с действительно высоким приоритетом, то ко времени зачета (пока нормально оформят, опишут результат, обоснуют... Занимает около недели обычно) она уже исправлена и не воспроизводится. Такие задачи мы тоже принимаем, если в них есть подтверждения (скриншоты).

Но смысл в том, что если ваша задача так и висит пару недель или месяц — то для разработчиков это явно невысокий приоритет ;-) Ну или в компании нет тестировщиков, и о баге они просто не знают...

Версия

Версия приложения:

- в которой баг нашли — для истории;
- в которой его нужно фиксить[75] — для планирования ресурсов на релиз и отслеживания задачи на дашборде релиза (в баг-трекере).

Если речь о кроссбраузерном баге, который воспроизводится четко в одном браузере (обычно этот несчастный — IE), то тогда еще версия браузера.

Компоненты

Где именно произошла ошибка? В модуле регистрации? Работы с корзиной? Отчетности?

Добавление компонентов помогает искать дубликаты (просматриваешь все открытые задачи по компоненту), а также анализировать: где больше всего багов копится? Это помогает понять, что надо сделать, чтобы багов было меньше. А тестировщику подсказывает, что надо тестировать в первую очередь.

Все остальное

Варианты полей, которые могут быть в вашем баг-трекере:

- время на исполнение;
- время на тестирование;
- закрывающий/тестировщик;
- ...

На каком-то докладе в SQADays[76] я слышала про 50 полей в JIRA на каждую задачу! И автор даже пытался всех убедить, как это круто и удобно. Ага. Менеджеру для отчетности. А вот исполнителям по 50 полей обязательных ради каждой мелочи заполнять совсем неудобно! Так что того докладчика на вопросах из зала просто заклевали ;-)

В любом случае, какие поля будут у вас — зависит от вашей компании. Но самые важные из них это:

- тип;
- название;
- описание.

Без них — никуда. Также часто есть приоритет и компоненты. Остальное уже мелочи жизни, про которые вам расскажут на работе.

Тотальная паранойя — друг тестировщика!

Конечно, даже идеально поставленную задачу не всегда исправят. Но лучше поставить задачу, чем не ставить. Пусть хранится для истории. Это может вам помочь.

На прошлом месте работы я успешно справлялась с тестированием своего проекта и подключилась на самый большой проект в компании. Проект я еще не знала, поэтому с вопросами сначала шла к аналитикам: баг это или не баг? Или к разработчикам, если вопрос касался технической стороны.

Как-то раз нашла я проблему и пришла к главному разработчику — баг? Он отмахнулся от меня: «Нет нет, все OK, это не баг. Так и должно работать». А через две недели прибежал к тестировщикам с выпученными глазами и начал кричать: «Тестировщики лентяи, такую багу пропустили!!!». Пропущенным багом оказалась... Та самая проблема.

— Погоди, я же тебе ее уже показывала, ты сам сказал, что это не баг.

— Не было такого! Вы лентяи, такую бажину пропустили!

И не докажешь уже, что не индюк, не записано!

Так что иногда лучше поставить «не баг», чем не ставить вовсе. Другое дело, что на это может быть завязан ваш KPI — сколько задач вы поставили, сколько из них оказалось не багами и т. д. Тут уже смотрите сами — заводить или не заводить.

Сколько задач заводить?

Вот вы обнаружили две орфографические ошибки: это два бага или один? А если кнопка «назад» как-то странно работает, но только в двух местах? А если вы грузили файл и схлопотали несколько ошибок?

Сколько ставить задач? Есть несколько принципов:

- **1 проблема = 1 баг** — это удобно, потому что задачи проще искать (более точный заголовок) и нет проблемы «разработчик исправил 4 пункта из 5».
- **Схожие проблемы = 1 задача** — разработчику часто проще исправить все похожие задачи за один присест. А вот двигать 10 статусов потом скучновато... А когда задача одна, удобно! Но тестировщику важно следить, что ни один пункт не продолбался при этом!
- **Золотая середина** — это то, что удобно именно вашей команде. Уточните у разработчиков, какие задачи вам группировать, а какие не стоит. Этим и руководствуйтесь!

Локализация ошибок

Что такое локализация?

Локализация — это поиск первопричины. Из-за одной причины может быть десяток следствий. Например, не работает загрузка фотографий в альбом. А кнопка загрузки есть в нескольких местах:

○ на главной странице сайта;

○ в личном кабинете в разделе фотоальбомов.

И там и там, если нажать «Загрузить», вылетит ошибка. Но не надо ставить два разных бага — ведь по сути он один, не работает функционал.

Если заводить разные задачи, вы усложняете жизнь разработчику. Ему надо при исправлении одного бага резолвить 2–4–6 задач. При этом надо открыть список из 20–50 задач, которые висят на нем, и самому искать дубликаты.

Конечно, он этого делать не станет. Закроет только ту задачу, которую увидел первой и по которой нашел проблему в коде. В итоге менеджер будет смотреть на статистику задач и думать, что разработчику еще 5 багов чинить. А они уже все давно исправлены. Нехорошо!

Поэтому перед постановкой задачи ее стоит локализовать. Это первый вариант локализации — когда мы нашли баг и хотим убедиться, что наша теория о том, что именно сломалось, верна.

Другой вариант локализации — для воспроизведения бага. Например, к нам пришел пользователь и стал жаловаться, что у него система падает. Или файл не грузится. Или что-то еще. И снова наша задача — локализовать баг. Найти причину некорректного поведения.

Как локализовывать ошибки?

Для локализации проблемы:

○ поисследуйте область рядом (а если не только «А», но и «Б» ввести?);

○ стройте догадки (вводить можно все, что захочешь)...

○ и опровергайте их!

Последний пункт может показаться немного странным. Но только на первый взгляд. Видите ли, есть такая ментальная ловушка: если мы уже что-то придумали, нам сложно отказаться от этой идеи.

Вот только так можно распаниковаться: «Ничего не работает», хотя если копнуть чуть глубже, окажется, что проблема небольшая. Бывает такое, что ты провел тест — ого, не работает! Провел второй подтвердить локализацию — ну точно, не работает!! И кажется, что всё не работает. А на самом деле мы просто подбираем примеры, подходящие под выдуманный нами образ. Сейчас разберемся подробнее.

Копаем рядом (принцип лопаты)

Когда вы находите баг, нужно покопаться рядышком. Вдруг вы нашли всего одно проявление, а их сильно больше?

Например, я загружаю аватарку, и система выдает ошибку. Можно поставить баг: «Вот на моей картинке упало», а можно покопаться рядышком. И выяснить, что другая картинка формата JPEG тоже не загружается, и вообще этот формат не грузится. Об этом уже в баге и пишем: «Не грузятся картинки формата JPEG».

Я называю это *принципом лопаты*: нашел баг? Покопай рядышком!

Можно ввести букву в числовое поле? Исследуем:

- любые символы можно ввести?
- или только е?
- или только русский алфавит?
- или только английский?

Система падает на загрузке DOC-файла:

- на любом файле?
- или только DOC?
- с любыми данными?
- с любым названием?

Учитесь задавать себе дополнительные вопросы, которые помогут точнее локализовать проблему. Ведь именно это отличает начинающего

тестировщика от более опытного. Не сможешь локализовать — не сможешь воспроизвести, снесут базу тестовую, а оно воспроизводиться перестанет, так как корень зла ты не нашел. И всё, се ля ви ¯_(ツ)_/¯.

Строим догадки и... опровергаем их!

Здорово, когда вы не просто сообщаете о проблеме, а сначала копаете рядом. Это помогает вывести догадку — не просто «А» вводится в поле, а, вообще, любой символ.

Но второй этап локализации намного сложнее. Когда мы уже построили какую-то теорию, нужно попробовать... опровергнуть ее! Слишком поспешный вывод не всегда будет правильным.

Например, мы ввели в числовое поле символ «а», символ «е» — и делаем вывод, что вводить можно всё что угодно! Это наша теория: «Можно ввести любые символы». Но иногда стоит копнуть чуть глубже, попытаться её опровергнуть. И вот уже выясняется, что разработчик позволил вводить числа в формате «1.2e+2». А это уже не баг ;-)

Сначала покопали, чтобы обобщить найденное последствие. Потом проверяем свою теорию. Будьте придирчивы, пытайтесь не только доказать ее, но и попробовать опровергнуть.

Сразу скажу — это сложно! Просто на уровне подсознания сложно. Мы уже вывели теорию, как можно ее опровергнуть? Это же будет значить, что она плохая! Так и попадаем в ментальную ловушку: «Свою теорию я только доказать попробую».

Учитесь тестировать свои же теории, смотреть на них непредвзято.

Не увлекаемся!

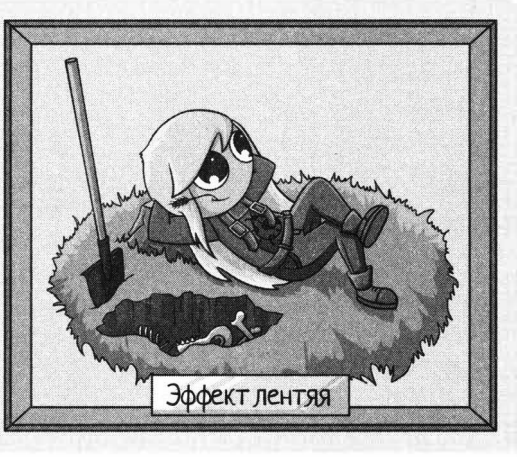
Копать — это, конечно, хорошо. Но не менее важно уметь вовремя остановиться.

Помните про правило 20 минут — сначала поисследовали баг сами. Если не получается, попробуйте спросить разработчика. Вполне возможно, что вы будете копать весь день, пытаясь уловить логику воспроизведения,

а разработчик посмотрит в код и поймет все за 5 минут. Не усложняйте себе жизнь. Сопоставляйте усилия и полученный результат.

Но и не усложняйте жизнь разработчику. Не надо бегать к нему каждые 5 минут. Иначе зачем на проекте нужны вы, если все за вас делает разработчик? Сначала сами, потом просим помощи.

Этот принцип я называю *эффект лентяя*. Копать — это очень хорошо. Но не надо закапываться, потому что у вас наверняка есть много разных задач. И потеря времени на одну из них может быть неоправданной. Покопали 20 минут — оформляем как есть, вдруг разработчик быстрее поймет.



Используйте правило 20 минут: нашли баг? Покопали рядышком. Если ничего больше не нашли, оформляем как есть

Принцип помогает в разных ситуациях. Вот мы нашли проявление бага, теперь нам надо, как мы помним:

1. Построить теорию.
2. Подтвердить ее.
3. Попробовать опровергнуть.

Но если тратить на каждый пункт кучу времени, то в итоге вы быстро бросите такое унылое дело, как локализация ;-)

Я рекомендую в начале пути пробовать, пока есть время. Если надо подтвердить теорию (это только «А» можно ввести или любую букву алфавита?), пробуем продумать несколько разных вариантов. Потом обязательно хотя бы минутку тратим на то, чтобы проверить теорию и опровергнуть ее.

Со временем у вас это станет отнимать все меньше времени, зато вы научитесь локализовывать баги. Конечно, всегда попадутся баги, которые сложно локализовать, но таких будет меньшинство.

Минимальные данные для воспроизведения бага

Другой вариант локализации — это поиск того, на чем именно упала система.

Иногда баги сами нас находят. Вот мы впихали большую строку данных — и система подвисла. Это она из-за миллиона символов упала? Или ей какой-то конкретный не понравился?

Или файл загрузили в систему, и он упал. Отчего? Из-за названия, расширения, данных внутри или размеров? Можно спихнуть локализацию на разработчика —пусть сам думает, что плохого в файле. Но часто можно найти причину и самому, а потом более точно описать проблему.

Если найти минимальные данные для воспроизведения бага, то:

- вы сэкономите время разработчику — ему не придется подключаться к тестовому стенду, самому грузить файл и дебажить;
- менеджер сможет легко оценить приоритет задачи — это нужно срочно исправлять или баг может подождать? Пока название: «Некоторые файлы падают, хз почему» — это сделать сложно...
- описание бага от понимания причины падения тоже только выиграет.

Но как же найти минимальные данные для воспроизведения бага? Если есть какие-то подсказки в логах, применяем их. Если подсказок нет, то самый оптимальный метод — метод бисекционного деления (также известный как метод «деления пополам» или «дихотомия»).

Метод применяется для поиска точного места падения:

1. Взять падающую пачку данных.
2. Разбить пополам.
3. Проверить половину 1:
   - ✧ если упало — значит, проблема там. Работаем дальше с ней;
   - ✧ если не упало — проверяем половину 2.
4. Повторяем шаги 1–3 до тех пор, пока не останется одно падающее значение.

Дихотомия позволяет весьма быстро локализовать проблему, особенно если это делается программно. Разработчики встраивают такие механизмы в обработку данных. А если не встраивают, то сами и страдают потом, когда к ним приходит тестировщик и говорит: «Вот на этом файле падает, а точную причину я не смог найти».

Подробнее о методе дихотомии можно почитать на Хабре в статье «Метод бисекционного деления в тестировании»[77].

Итого про локализацию

Для локализации нужно:

1. Найти четкую последовательность шагов.
2. Подумать:
   - ✧ а в каких условиях еще может такое произойти? Проверил ли я их?
   - ✧ где у меня подобная функциональность? Как она работает в этой ситуации?
   - ✧ а точно ли нужны именно такие условия, или часть условий можно изменить, а баг все равно будет воспроизводиться?
   - ✧ а есть ли условия, при которых вот при этой последовательности шагов баг не воспроизведется?

Оформление задач

Оформление названия

Название бага должно четко отвечать на вопросы:

- ○ Где проблема?
- ○ В чем она состоит?
- ○ Насколько это срочно?

Если название сложное и надо долго вникать — мозг отказывается это делать. Приходится заходить внутрь и читать описание. Если название «ни о чем», ситуация аналогичная. Вот что мне должно сказать название «Все сломано» (табл. 5.6)?

Таблица 5.6. *Название (заголовок) бага*

Как плохо	Как хорошо
Заголовок «Что-то пошло не так» ни о чем не говорит	Название конкретное — оно сразу показывает, в чем именно проблема
Он слишком общий (как мы разбирались в тест-кейсах)	Отвечает на 3 главные вопроса: Что случилось? Где, в каком модуле программы? Когда?
Некорректно работает поиск	Поиск не ищет по красному цвету

Если вы ставите баг, то помните: баг — это всегда проблема. И наша задача — сразу же ее обозначить. При этом избегайте слов-паразитов (табл. 5.7). Это те слова, которые можно трактовать совершенно по разному:

- некорректно — это как именно?
- неправильно — что это значит?
- ошибка — какая именно?

Таблица 5.7. *Использование слов-паразитов*

Нет	Да	Оказывается, что...
Некорректное поведение системы	При авторизации через Фейсбук не приходит приветствие на почту	Проблема только на этапе авторизации и только через одну соцсеть. При первом названии это было неясно
Неправильно подсказываются адреса	Крымские адреса не подсказываются	Проблема не во всех адресах, а только в крымских
Ошибка в системе	При загрузке пустого файла выдается пустое сообщение об ошибке	Проблема только при загрузке пустого файла, значит, некритичная, нефиг грузить что ни попадя. Да и система не зависает, даже сообщение выдает, просто пустое

Я рекомендую вам использовать сервис Багред[78] при постановке задач. Я сделала его специально для своих студентов и других страждующих. Логика совершенно простая — он ловит по регэкспам типичные слова-паразиты.

Если вы ставите улучшение, то опишите, что вы предлагаете сделать в будущем (табл. 5.8):

- запретить регистрацию с именем Ктулху;
- выводить в сообщение об ошибке детальную информацию о ее причине.

Помните, что:

- баг — настоящее время (проблема есть);
- улучшение — будущее время (что я хочу сделать).

***Таблица 5.8.** Баг vs улучшение*

Баг	Улучшение
Можно зарегистрироваться с именем Ктулху	Запретить регистрацию с именем Ктулху
Сообщение об ошибке указывает на неверный пароль при вводе недопустимого имени	Выводить в сообщении об ошибке детальную информацию по причине

Оформление описания бага

Шаблон бага

Шаблон помогает не ломать голову, как бы оформить «красиво, понятно, при этом быстро». Я сама использую такой шаблон последние лет пять или больше. Так что он не только для новичков ;-)

* * * \

Принципы оформления

- выделяем жирным заголовки (шаги, результат, ожидаемый результат);
- нумеруем шаги. Единственный шаг нумеровать не надо;
- отделяем шаги от результата и ожидаемого пустой строкой, дабы не получилась нечитабельная простыня текста;
- сначала фактический результат (ФР), а потом — ожидаемый (ОР).

В баге всегда пишется сначала ФР, а потом ОР. Потому что менеджер или разработчик читают задачу, чтобы понять, в чем проблема. Читают, читают шаги, и тут бац — нормальный результат. Так в чем проблема? А в том, что потом фактический идет, и это сбивает с толку;

- ОР есть всегда.

В баге всегда пишется ОР. Даже если он кажется вам очевидным. При этом:

- ✧ он может быть очевидным только для вас. Другие члены команды не понимают, что нужно сделать;
- ✧ он может быть очевидным для всех, но только сейчас. Если баг отложить на полгода, можно долго вспоминать: «Что же тут ожидалось?».

Шаги

На что обращать внимание, когда вы пишете шаги бага?

1. Давайте ссылку, если она есть!

Если проблема в веб-приложении, дайте на него ссылку. Это ускорит выполнение шагов. Вместо: «Таааак, открыть отчет по прошлому месяцу, где он там находится, говорите? А на каком из трех тестовых серверов мне это вообще проверять?» — мы просто переходим по ссылке, и всё!

Вместо:

> Открыть отчет по прошлому месяцу.

напишите:

> Открыть отчет по прошлому месяцу: **http://example.com/**.

Важный момент: если есть прямая ссылка на форму, дайте ее:

вместо:

> Открыть главную страницу: **http://example.com/**
> Пролистать до середины экрана — раздела **how_to**.

напишите:

> Открыть раздел «Как это работает» на главной странице: **http://example.com/#how_to.**

Зачем писать два шага там, где можно написать один? И зачем мне для воспроизведения что-то куда-то листать, переходить по 10 пунктам в меню, если можно сразу дать ссылку туда, куда мне нужно попасть? Экономьте время воспроизводящего ваш баг человека!

А вот вам полезная статья в помощь: «Как получить прямую ссылку на всплывающее окно»[79].

2. Ссылку надо пояснить.

Из крайности в крайность: ссылку дали, засим и успокоились: «Открыть **http://example.com/»**. Зачем писать, что это за ссылка? Щелкни и проверь!

Я открою вам страшную тайну: ссылки устаревают. Может быть, баг отложили на месяц как некритичный. За это время базу тестовую несколько раз снесли, и ссылка в личный кабинет конкретного пользователя теперь ведет на 404.

Или мы решили изменить URL ссылки. Старая теперь недоступна. И чтобы понять, о чем был баг, надо вспоминать, что значила эта ссылка раньше.

Так что ссылку дали, дайте и ее описание:

вместо:

> Открыть **http://example.com/.**

напишите:

> Открыть личный кабинет: **http://example.com/**.

3. Дайте данные для авторизации.

Ссылка есть, описание есть, но... Разработчик переходит по ней и видит окно авторизации. Что ему делать? Какие данные вводить? Это важно вообще, что будет за пользователь? И где взять тестовые данные?

Иногда тестировщики не пишут данные, потому что «их и так все знают». Потому что есть пачка тестовых пользователей, на которых они гоняют все тесты. И они даже где-то описаны.

Это все здорово и хорошо. Но если ваш баг читает разработчик или менеджер проекта, он может про тестовых пользователей не знать. И придется ему тратить время на регистрацию. Или бегать к тестировщику узнавать, под кем можно войти. Сохраните команде время, укажите данные для входа.

Если пользователь должен быть каким-то особым, укажите это:

Войти под свежезарегистрированным пользователем, который еще не обрабатывал файлы (например: test / 123).

или:

Войти под пользователем, который уже участвовал в опросе XXX (например: test / 123).

Если неважно, что за пользователь, так и пишите: «пользователь любой»:

Войти под любым пользователем (например: test / 123).

или:

Открыть отчет XXX (данные для входа: test / 123, могут быть любыми).

4. Указывайте локализацию в шагах.

Если вы выяснили, в чем именно проблема, пишите именно о ней.

Вместо:

Загрузить на аватарку фото из аттача.

укажите:

Загрузить на аватарку любое фото формата JPEG корректного размера.

Подробнее об этом можно прочитать в статье «Не пишите в баге „Ввести 6,9“!»[80].

А еще локализация — это поиск минимальных шагов.

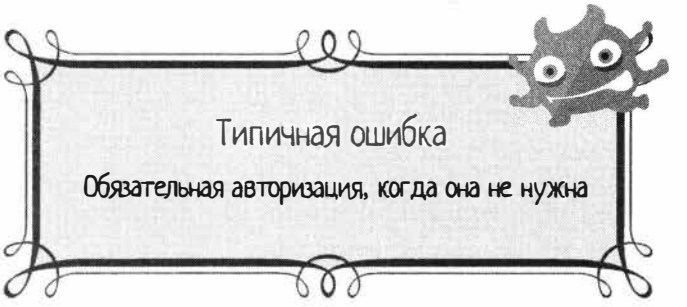
Вместо:

1. Открыть главную страницу: **http://example.com/**.
2. Войти в систему.
3. Нажать кнопку «Выход».
4. Попить кофе.

5. Снова войти.
6. Пойти покурить.
7. Снова выйти.
8. Снова войти.
9. Нажать кнопку «Личный кабинет».

напишите:

Открыть личный кабинет: **http://example.com/lk**.



Когда мы рассказываем студентам, что надо искать минимальные шаги для воспроизведения бага, они обычно критически осматривают свои и многие убирают. Но не авторизацию. Это воспринимается как истина в последней инстанции: раз я сейчас авторизован и у меня воспроизводится — авторизация нужна!

Правда, нужна? Да, понимаю, вы с этим приложением работаете постоянно, вот и сидите в нем вечно авторизованным пользователем. Но попробуйте разлогиниться и воспроизвести ошибку. Если получится — то это явно более короткие шаги для воспроизведения, чем «зарегистрироваться и войти»! Вот их и пишите.

5. Всё лишнее уносим в доп. инфо.

Если очень хочется указать, что баг воспроизводится в разных местах, сделайте это. Если, например, воспроизводится в любой форме ввода данных, можно так и написать:

В любую форму ввода денежных данных ввести символ — например: «А». Пример формы — форма доходов **http://example.com/income**.

Если хотите привести ссылки на все доступные формы — сделайте это в доп. инфо. Вполне допустимо даже в шагах написать про одно место, а остальные упомянуть потом. Это в любом случае будет лучше, чем делать разные ветки шагов.

Вместо:

Шаги

1. В форму ввода доходов ввести «А».
2. В форму ввода расходов ввести «А».
3. В форму ввода процентной ставки ввести «А».

...

напишите:

Шаги

В форму ввода доходов **http://example.com/income** ввести любой символ — например: «А».

Доп. инфо

Воспроизводится также для полей:

1. Расходы: **http://example.com/outcome**.
2. Процентная ставка: **http://example.com/persent.**
3. ...

Конечно, если воспроизводится на любой форме, об этом лучше сообщить в шагах. Но не всегда так делают по разным причинам:

- воспроизводится не на всех формах, а на 3 из 5. Перечислять их в шагах будет лишним нагромождением;
- воспроизводится везде, но решили сократить шаг для лучшей читаемости. Этот вариант менее хорош, но, будем честны, его часто используют. И иногда разработчики сами так просят.

Ведь разработчику важно что? Он прошелся по шагам и воспроизвел. И в идеале он воспроизвел для доходов, исправил в коде, а в остальных местах оно «само поправилось», если разработчики грамотно переиспользуют код.

В любом случае сразу все ссылки вываливать не надо, если вас отдельно об этом не просили. Сохраните их для себя, в дополнительной информации. Это ускорит проверку бага, но не помешать его читабельности.

6. Не забудьте про скриншоты и примеры!

Почему в этой книге столько картинок? Потому что они привлекают внимание! Это первое, за что цепляется взгляд.

Хорошо оформленный баг — это когда разработчик прочитал название, посмотрел на скриншот, и всё. Ему больше ничего не нужно, он понял, где ошибка. А шаги... Шаги остаются тестировщику для перепроверки задачи.

Пример — это не только какое-то значение, которые мы вводим в строку:

«Например, 6,9».

Это может быть файл, на котором воспроизодится баг. Или SOAP-запрос, с помощью которого вы нашли проблему. Вложите это все в баг-трекер, пригодится.

Вместо:

Загрузить файл.

напишите:

Загрузить файл весом более 20 Мбайт.
(Например, см. аттач more_then_20mb.csv).

Читая такую задачу, я сразу вижу локализацию. И, что важнее, легко могу воспроизвести ошибку. Не бегая при этом среди тестировщиков с вопросом: «У кого есть корректный файл на 20+ Мбайт весом?».

Баг — не тест-кейс, разработчику предварительные шаги не нужны. Он не будет готовить файл, это ваша работа. А он лучше возьмет готовый. Поэтому убираем подготовительные шаги и описываем локализацию (что за файл) в тот момент, когда на него ссылаемся.

Вместо:

1. Готовим файл A.jpg.
2. Грузим его в систему.

напишите:

Загрузить файл весом более 20 Мбайт (Например, см. аттач more_then_20mb.csv).

Результат

При оформлении результата тоже есть свои правила, но их немного.

1. В задаче есть фактический результат (ФР) и ожидаемый (ОР).

Иногда кажется, что ОР не нужен, потому что он очевиден. Но поверьте, это далеко не всегда так. «Загрузили отчет, и он упал, а падать не должен» — а что он должен выводить в такой ситуации (где-то в расчетах деление на ноль)?

Лучше прописать ОР, чем смотреть спустя три месяца на свою же задачу и размышлять, «а что тут ожидалось-то?».

2. Сначала идет фактический.

В баге обычно пишут сначала ФР, а потом ОР. Почему? Потому что вы описываете какую-то проблему. А читают люди сверху вниз и слева направо.

Логично читать: «Я загружаю фоточку, и система падает». Потому что если я читаю: «Я загружаю фоточку, и всё работает», то сразу вопрос — а в чем проблема-то? Сначала опишите проблему текущую, а потом ее решение.

3. Ожидаемый результат надо обосновать.

Почему вы ожидаете именно такой ОР? Объясните это один раз письменно, тогда не придется отвлекаться от работы и отвечать всем желающим устно.

Вместо:

Результат

Генеральный директор — Вася Пупкин.

Ожидаемый результат

Генеральный директор — Иван Иванов.

Это точная инфа, ***мамой клянусь***!

напишите:

Результат

Генеральный директор — Вася Пупкин.

Ожидаемый результат

Генеральный директор — Иван Иванов.

Он был изменен 03 августа 2017 года, согласно записи в ЕГРЮЛ[81] (ссылка на ФНС + скриншот этой записи)

Когда я говорю про обоснование задач моим студентам и смотрю на результат, то выделяю три типичных антипаттерна обоснования — три ступени гнева, отрицания и обоснования багов.

1. Очевидно же!

Нам очевидно, вот и не обосновываем. А потом получаем: «Ой, я забыл, почему так хотел»...

Это очевидно только вам, только здесь и только сейчас. Через полгода сами забудете, почему именно так. Объясните, как для тупых, что там за очевидность такая.

2. Мамой клянусь, так правильно!

Зачем клясться? Почему-то же вы считаете, что так правильно, вот и расскажите, почему. Дайте ссылку на требования, например.

3. Зайчики обиделись...

«Ах, ты не добавил котика на главную? Ну все! Я обиделся... И УШЕЛ! А вы! Потеряли клиента!!». Но то, что неудобно вам, может быть удобно другим. Так что уберите эмоции и приведите факты.

Вместо них стоит использовать правильное обоснование.

4. Пруфлинк.

ТЗ, Интернет, письмо клиента, в котором он просил этот функционал...

5. Единообразие.

Если система всегда работает так, а в одном месте по-другому, стоит исправить!

6. Проблема.

Опишите проблему, которая возникает у вас или у пользователя (если вы общаетесь с клиентами). Реальная проблема всегда лучше, чем просто негативный тест.

Обязательно пишем, почему мы решили, что это действительно баг, и почему мы хотим, чтобы его исправили именно так, как мы тут написали. Задаем себе вопросы из серии «три почему»:

— почему я считаю, что это очевидно?

— почему я считаю, что так правильно?

— почему я считаю, что пользователи расстроятся?

Даем ссылки на требования, в Интернет, указываем на единообразие или рассказываем о реальной проблеме пользователя.

И, когда будете спустя год перечитывать грамотно оформленные задачи, вы еще скажете мне: «Спасибо!».

А подробнее о паттернах/антипаттернах можно почитать в моей статье на Хабре «Паттерны и антипаттерны обоснования задач»[82].

Оформление описания улучшения

Шаблон улучшения

Шаблон для бага[83]— простой и удобный. Вставляешь в баг-трекер и заполняешь. Всё. В улучшении нет такого ярко выраженного шаблона. Оно формулируется своими словами. Но по следующей схеме:

> **Что предлагаем и зачем.**
>
> Покупка товара в один клик без регистрации — чтобы пользователи не уходили к конкурентам, плюнув на тонну «обязательных» полей. Вон у конкурентов из «XXX» так и сделано, их пользователи в отзывах хвалят, а нас ругают за эти поля.

Можно сначала кратко описать проблему, а потом «что предлагаем». Можно наоборот: что предлагаем и почему. Но структурированно, кратко и наглядно.

Вот и всё! А теперь нюансы.

Нюансы

1. Как везде — не повод повторять.

На многих сайтах вводят ограничение на пароль: «Минимум одна заглавная буква, две буквы разного алфавита, три цифры, идущие не подряд,

один спецсимвол, три приседа с бубном и одной свистопляской». Но это не повод ставить «улучшение» к нашему сайту, чтобы сделать так же. Боже упаси так улучшать — пользователи не любят запретов.

Если все прыгают с крыши — это не значит, что нам тоже надо ;-)

2. Сразу продумайте тесты.

Не оставайтесь на этапе идеи. Найти «классную возможность» и радостно поскакать заводить улучшение — удел джуниоров, которые не хотят учиться. Опытные тестировщики додумывают до конца, что они ожидают увидеть. Возможно, «овчинка выделки не стоит», а вы и не в курсе. Если же начнете продумывать улучшение, то сами увидите, что его лучше вообще не заводить.

3. Избегайте поп-апов.

Встречали сайты, на которые не успеешь зайти, а тебе всплывающее окно на половину экрана? Бесит, правда?

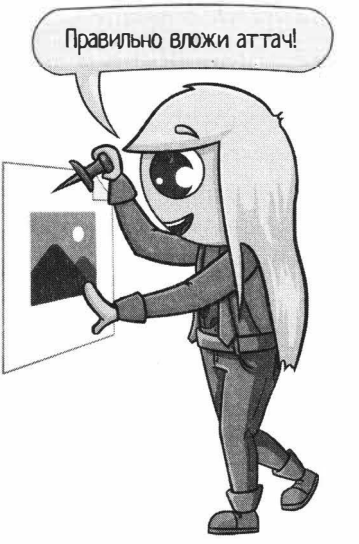
А ведь сообщения об ошибках часто делают как раз в таких всплывающих окнах! Но зачем? Если можно написать текст ошибки около «проблемного» поля, так и сделайте, такое и предлагайте. Выводить поп-ап в этом случае — не комильфо.

Как правильно вложить аттач?

Редкий баг обходится без дополнительных вложений. Обычно у нас есть название, описание + некий аттач. Аттач — от английского слова attachment, вложенный файл. Это может быть:

- скриншот (картинка экрана);
- видео;

- файл для воспроизведения;
- файл с доп. информацией;
- ...



Чтобы сделать задачу лучше, используйте следующие правила:

1. Говорящее название аттача.

Не 555.jpg, а «Ошибка при регистрации» или «Борода у женского персонажа».

2. Ссылайтесь на аттач по тексту задачи.

Иначе будет непонятно, в какой момент надо смотреть на рисунок или скачивать файл. Поэтому в шагах/результате пишем: «См. рис. такой-то» или «Загрузить такой-то файл для примера в аттаче таком-то».

3. Удаляйте лишнее.

Если файл, то минимально необходимый. Если скриншот, то только проблемной зоны, а не всего экрана.

4. Используйте стрелки на скриншотах.

Покажите ими, куда надо смотреть.

Дополнительные поля

Вот вроде все важное обсудили, но это же еще не все!

Если открыть баг-трекер, то обычно мы видим там гораздо больше полей, нежели просто название и описание.

Я не могу описать все-все-все возможные варианты, потому что это нереально. Баг-трекинговые системы разрешают добавлять не только стандартные поля, но и какие-то свои. Так что какой комплект будет у вас на работе — узнаете уже на работе. Но тут не страшно — один раз узнали, зачем каждое поле, и всё!

Я упомяну здесь про некоторые типовые поля:

- **Приоритет** — насколько важно исправить задачу (это обычная задача или СУПЕРСРОЧНАЯ?). Иногда разделяют Priority и Severity, но вы вряд ли будете заполнять оба. О разнице между ними мы говорили в этой главе ранее (см. *разд. «Приоритет»*).
- **Версия** — в какой версии баг появился? А в какой его нужно исправить? Каждое внесенное изменение меняет версию программы. Важно понимать, когда надо исправить и с какого момента ошибка есть.
- **Исполнитель**— кто в настоящий момент работает над задачей.

Пример оформления

Если вы хотите посмотреть на примеры оформления или то, какие баги вообще встречаются в жизни, почитайте мои посты в блоге с меткой «шаблон бага» или «повсюду баги»[84].

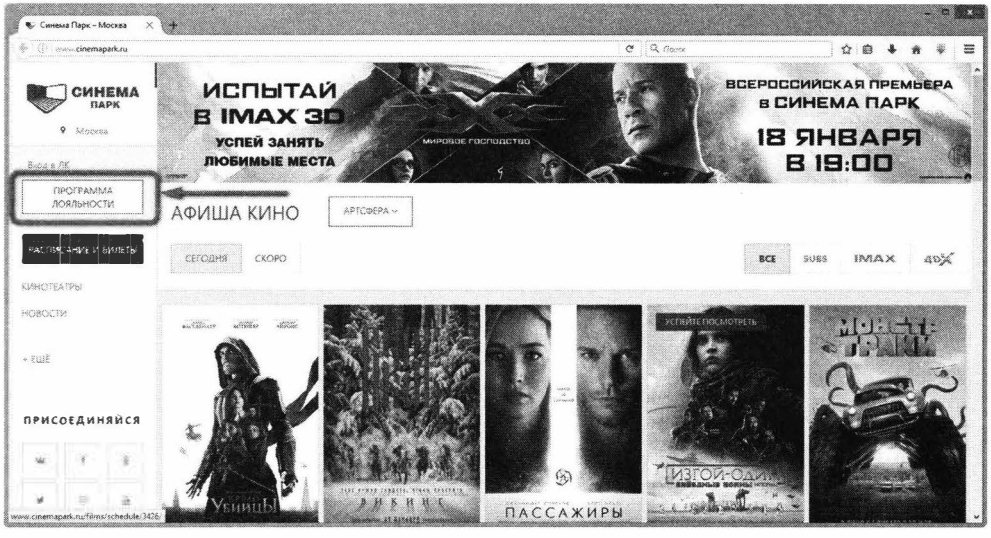
Я их собираю также в посте «Баги повсюду. Сборная солянка»[85]. В нем вы можете увидеть, какие баги я встречала в интернет-магазинах и банках, на сайтах и в мобильных приложениях. Там же можно и идеи для тестов поискать ;-)

Но здесь в книге я все же приведу один баг в качестве примера оформления (от названия до результата).

Nginx error при просмотре программы лояльности

Шаги для воспроизведения

1. Открыть главную страницу: **http://www.cinemapark.ru/**.
2. Нажать на кнопку «Программа лояльности».



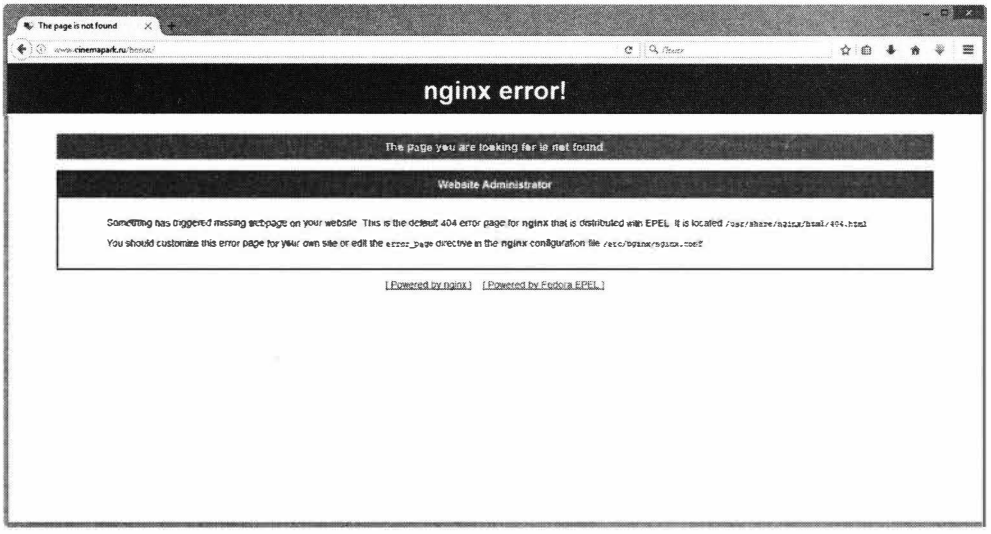
Результат

Открывается страница **http://www.cinemapark.ru/bonus/**, но Nginx ее режет. В итоге видим ошибку **Nginx error**:

Nginx error!

Something has triggered missing webpage on your website. This is the default 404 error page for nginx that is distributed with EPEL. It is located /usr/share/nginx/html/404.html

You should customize this error page for your own site or edit the error_page directive in the nginx configuration file /etc/nginx/nginx.conf.



Ожидаемый результат

Открывается программа лояльности. Ссылка должна открываться не только внутри сети, ведь на нее идет отсылка с главной страницы сайта. То есть это не внутренняя документация.

* *

На что следует здесь обратить внимание:

1. Шаги отображают реальный сценарий.

Можно, конечно, сразу в шагах дать прямую ссылку:

открываем **http://www.cinemapark.ru/bonus/**, и все падает!

Но такой баг могут закрыть, потому что не видно проблемы. Откуда у пользователя прямая ссылка? Может, ему кто-то скинул информацию, которая и должна быть доступна только внутри корпоративной сети. И все ОК — неча лезть куда не просят! А так мы показываем, что вот у тебя на главной странице большая призывная кнопка — и когда она не работает, это фейл[86] ;-)))

2. Названия скриншотов говорящие, а не просто «рис. 1, рис. 2»...

Специально обращаю на это ваше внимание, потому что отсутствие говорящих названий скриншотов — типичная ошибка у моих студентов.

3. Вложены логи (текстом, не скрином).

Недостаточно приложить скриншот ошибки, лучше выписать ее текст. Потому что по тексту можно искать, а по картинке — нет. А еще бывает, что ты забыл вложить аттач, и сиди потом, гадай при проверке, что за ошибка такая выскакивала.

Типовые ошибки

В этом разделе я хочу рассказать вам о некоторых типовых ошибках, которые часто встречаются в системах. И которые лучше проверить!

Кроссбраузерность

Некоторые баги встречаются только в конкретном браузере. Это когда ошибка не в функционале, а в отображении: текст расколбашивает, картинка не влезает на экран...

Чтобы понять, кроссбраузерный баг или нет, просто проверьте его в другом браузере. Если и там все плохо, значит, дело не в браузере.

Основные браузеры:

- Internet Explorer (IE) — правда, Майкрософт его больше не поддерживает;
- Edge;
- Firefox;
- Chrome;
- Opera.

Разумеется, в вашем приложении набор основных браузеров может быть другой. Вообще, если тестируете веб, обязательно уточните, в каких браузерах должна работать система. Потому что если это где-то обещано, то работать должно, и это стоит проверить.

Если система должна работать в IE, то вам есть где разгуляться: именно в этом браузере много проблем, которых нет в хроме или фаерфоксе. Проверяйте все в IE, найдете кучу багов.

Чтобы тестировать старые версии IE, есть разные пути:

- виртуальные машины с установленным браузером — есть официальные виртуалки от Майкрософта, берите их;
- инструменты типа «IETester». Удобно, но будьте аккуратны! Ведь тестируя через инструмент, всегда есть шанс, что вы тестируете именно инструмент, а не саму систему. И то, что воспроизводится в инструменте, не воспроизведется на реальном браузере.

Concurrency (параллельная работа)

Concurrency (англ.) — параллелизм. В случае тестирования — параллельная работа с одним и тем же приложением:

- веб — открыть в разных вкладках и попробовать выполнить одно и то же действие или противоречащие друг другу действия (в одной вкладке редактируем, во второй удаляем);
- десктоп, мобильные — запустить одно приложение несколько раз.

Эти простые действия постоянно приносят баги. Например, я встречала такие проблемы:

- редактируешь в разных вкладках, и на одной из них получаешь страшный эксепшен[87]. А ведь это вполне возможный вариант — когда разные пользователи решили отредактировать одну карточку;
- открываешь много-много вкладок на создание карточки — в итоге система падает на попытке сохранить данные. Очень мучились с локализацией, тестировали с разработчиком создание новой карточки и так, и сяк... Но пока за спиной пользователя не постояли и не увидели много окошек, не смогли воспроизвести.

Так что особенно хороша параллельная работа для тестов CRUD:

- C — Create;
- R — Read;
- U — Update;
- D — Delete.

Валидация клиент-сервер

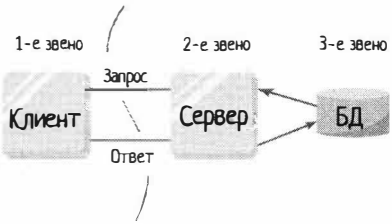
Клиент-серверная архитектура — наиболее типичная в последнее время. Это раньше разработчик писал код и запускал на своем же компьютере. А теперь код программы поднимается на одном компьютере (сервере), а пользователи обращаются к нему со своих машин (клиентов).

Получается, что у нас один сервер и сколько угодно клиентов. Например, система Users[88]. Откройте ее в браузере — у вас нет доступа к серверу, на котором лежит исходный код, но вы видите систему и можете с ней работать.

В качестве клиента выступает браузер. Он может быть запущен на компьютере, ноутбуке, планшете или телефоне. Не суть.

Сервер — это отдельная машина, на которой развернуто приложение. Клиент отправляет ему запросы вида «покажи мне всех пользователей» или «только пользователей с именем Маша», а сервере возвращает ответ.

Сами данные чаще всего хранятся в базе данных (БД). Получается трехзвенная архитектура: клиент — сервер — БД.



Так вот, если на поле ставится ограничение, его могут установить на стороне:

- клиента;
- сервера;
- клиента и сервера.

Самый правильный вариант, разумеется, последний. Если ограничение установлено на клиенте, его обязательно надо продублировать на сервере. Потому что ограничение на клиенте элементарно снимается.

Типичное ограничение на клиенте — поставить на поле атрибут `maxlenght`. Скажем, есть поле «Имя». Разработчик вешает ограничение `maxlenght = 10`. Теперь, если я буду в браузере вводить символы в это поле, после 10-го символа система перестанет печатать. Все, поле ограничено.

Это все хорошо, но такое ограничение легко обходится через панель разработчика[89] или плагин Web developer toolbar. Сняли ограничение, попробовали послать 200 символов — и что? Будет плохо, если система развалится. Или зависнет.

Особенно плохо, если один человек попробует сломать, а сломается на сервере — то есть у других пользователей система тоже перестанет отвечать. Это уже критично. Вот если просто у него страшная ошибка вылезла — переживет, неча ломать было. А если проблема теперь у всех — ее надо решать.

Чтобы ошибка была «красивой», ее надо обработать в коде на сервере. Поэтому обычно разработчик страхуется и, кроме ограничения на клиенте,

ставит ограничение на сервере. Другое дело, что он может перепутать что-то, и границы будут разные:

- клиент — 100 символов;
- сервер — 90 символов.

Тогда это уже баг — надо сделать, чтобы границы были одинаковые.

Если видите ограничение на клиенте, обязательно попробуйте его обойти и проверить — а есть ли такое же на сервере?

Буква «ё»

Символы «ё» и «Ё» — не входят в интервал «А–Я» в регулярных выражениях, не входят в последовательный диапазон ASCII[90] и часто заменяются на «е».

Поэтому букву «ё» надо проверять в текстовых полях. Заменится она на «е» или нет? Как работает поиск по «е» и «ё»?

Если на поле стоит ограничение «можно вводить только русские символы», тем более проверяем, ведь такое ограничение легко реализовать как раз через регулярное выражение `[А-Я]`, куда буква «ё» не попадает, — вы просто не сможете ее ввести, или при сохранении данных выскочит ошибка «введите только русские символы». Косяк!

Как закрывать задачи?

Статью на эту тему я написала в рабочем конфлюенсе в 2013 году. И на момент подготовки этой главы (2021 год) она все еще актуальна.

Исходно чек-лист я записала как напоминание, в том числе и себе. Потому что к задачам приходится обращаться, в том числе людям, которые их НЕ проверяли. Например, во время регрессии надо хотя бы базово проверить функционал.

И вот ты открываешь задачу, листаешь до последнего комментария — посмотреть, какая документация, что как работает, а там... Пусто. Или скромное «Все проверено, все ОК». А документация где? Я же не в теме задачи, я хочу прочитать побольше!

Или если заказчик пишет, что у него что-то не работает, и ты хочешь проверить, покрыта ли ситуация автотестами. Идешь в задачу, а там нет ссылки на автотесты. Их вообще не писали? Или просто ссылку не дали? Приходится выяснять...

Так и появился чек-лист закрытия задачи:

1. Проверить задачу (здравствуй, кэп). Используйте готовые чек-листы для типовых задач + избегайте типовых ошибок в автотестах.
2. Написать по ней документацию (min — пользовательскую, max — пользовательскую и «для коллег»).
3. В баг-трекере оставить комментарий «проверил на сборке 123, посмотрел то, то и то, написал такие-то автотесты (ссылка на них), дока — вот она».
4. Приложить к задаче тестовые данные, если что-то проверялось вручную.

Чек-лист вроде как «про нас», но на самом деле универсальный. Ну разве что в вашей компании тестировщик не пишет документацию — тогда второй пункт меняем на «проверить, что вся нужная документация написана или актуализирована».

Рассмотрим каждый пункт отдельно.

Документация

- Если документации никакой по задаче нет (неважно, баг это или improvement), она переоткрывается.
- Если документация есть, но в JIRA нет на нее ссылки в последнем комментарии — задача переоткрывается *(суровые времена, суровые меры)*.

- Минимально написать/исправить пользовательские требования.
- Если же были изменения в конфигах проекта, надо проверить, что есть техническая документация по такому изменению. Если ее нет, пишем.
- Если добавлялись задачи миграции — сразу написать общую инструкцию по обновлению релиза.

Комментарий

- Если придется возвращаться к задаче позднее, иногда бывает полезно узнать, на какой версии тестировал тестировщик и что он проверял (вкратце).
- Записываем версии из системы контроля версий, даем ссылку на документацию и краткое описание: «Проверил вручную через интерфейс / SOAP / REST».

Тестовые данные

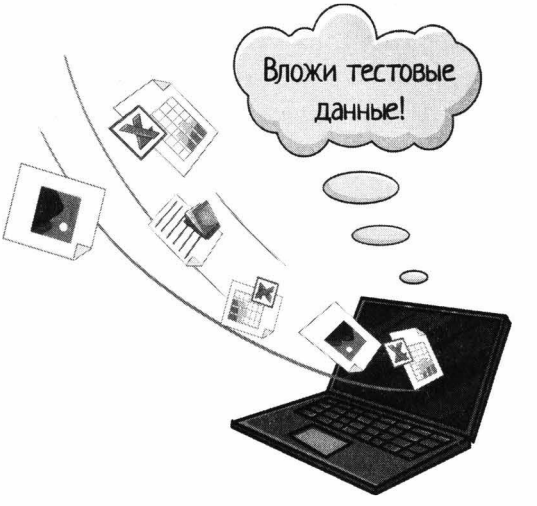
Если задача проверялась вручную, обязательно прикладываем к ней тестовые данные (если они не весят 3 Тбайт).

Если через полгода у заказчика всплывет проблема и будут подниматься старые задачи для воспроизведения, эти файлики очень помогут — проверено и не раз.

Помните — взять готовый SQL-запрос и проверить по базе займет одну минуту. А снова сидеть и составлять этот запрос — это может и час занять, если не больше. Экономьте свое время!

Поэтому:

1. Создалиданные по большому SOAP-запросу — сохранили его в текстовый файл и вложили в задачу.
2. Выполняли сложный SQL-запрос — сохранили в файл и вложили в задачу. Если короткий, можно прямо в комментарии весь записать (поиск зато будет работать по нему).
3. Загружали тестовые данные из хранилища — аттачим загруженный файл.



Коммент с тестами, докой и данными должен быть заключительным. А не так, что «я написал такие-то тесты, нашел такие-то проблемы» и потом переписка с разработчиком на 20 комментов, а твой «заключительный» затерялся где-то в середине. Если затерялся — продублируй (старый можно удалить).

Ретроспективный анализ ошибки, или Как анализировать пропущенные баги

Нельзя найти все баги. Вы все равно что-то упустите. И в продакшене всплывет ошибка, которую вы не заметили. Найдет ее уже пользователь или клиент. Что делать в такой ситуации?

Искать виноватых бесполезно. О'кей, виноваты не вы, проверку не выполнил Вася. Дали ему втык, отобрали зарплату. Полегчало? Через месяц Маша допустит ту же ошибку. Повторяем наказание. Ничего не меняется. Ребята ходят по граблям, на которые уже наступали.

Поэтому вместо наказания оглядываемся назад, на причины ошибки, и размышляем, откуда она взялась. Анализируйте баг методом «5 почему», ищите корень зла. И исправляйте именно его.

А после исправления подумайте, как не пропускать такие баги:

- если налажал новичок — возможно, ему где-то что-то не объяснили. Надо провести обучение или отправить его на курсы;
- если налажал старичок — тем более важно подумать, почему. Может, проверять надо столько всего, что без чек-листа какие-то пункты продалбываются. В целом он знает, что это надо проверять, но тут вылетело из головы. И тогда надо записать чек-лист проверок.

«Важно не то, сколько раз ты упал, а то, сколько раз поднялся» ©

Если вы будете учиться на своих ошибках, то это круто! Ошибаются все, а вот выводы делают немногие.

Если на вашей работе нельзя открыто рассказывать о таком анализе, чтобы не получить штраф, — все равно проводите анализ. Но уже для себя, а не в общем доступе.

Анализируйте баг методом «5 почему».

Найдите причину, а не последствия.

И не забудьте подумать, как
не пропустить такие баги в будущем!

Шпаргалки

От Павла

Эту шпаргалку написал мой коллега Павел Абдюшев[91] в помощь моим студентам еще года 3–4 назад.

С тех пор она так и используется активно ребятами! Ведь по оформлению бага столько разных статей, ну разве удержишь это все в голове? А тут единый списочек. Так сказать, чек-лист заведения бага!

- ○ Пойми суть бага.
- ○ Напиши заголовок.
- ○ Проверь его в Багреде.
- ○ Определи серьезность.
- ○ Напиши шаги воспроизведения.
- ○ Напиши текущий результат.
- ○ Напиши ожидаемый результат.
- ○ Добавь версию системы и других связанных компонентов.
- ○ Добавь скриншот и необходимые файлы.
- ○ Добавь логи, если нужны.
- ○ Закрой глаза и посиди полминуты.
- ○ Прочти заголовок и сформулируй его понятно.
- ○ Строго по своим шагам воспроизведи баг. Добавь пропущенную информацию, убери ненужную.
- ○ Сделай ссылки ссылками, добавь все необходимые данные.
- ○ Сократи шаги воспроизведения, насколько это возможно.
- ○ Убери лишнее из заголовков, шагов, результатов.
- ○ Убери всё лишнее.
- ○ И еще раз убери лишнее.

- Проверь ожидаемый результат: пользователь ожидает именно этого? Мы будем лучше решать проблему пользователя, если сделать именно так, как написано?.
- Подумай, точно баг, а не фича?.
- Еще раз проверь критичность бага.
- Проверка орфографии и пунктуации.
- Финальная вычитка и воспроизведение.
- Заводи!

Плакат НЛО: Найти, Локализовать и Оформить ошибку

А это вам уже от меня чек-лист перед заведением бага. В картиночках!

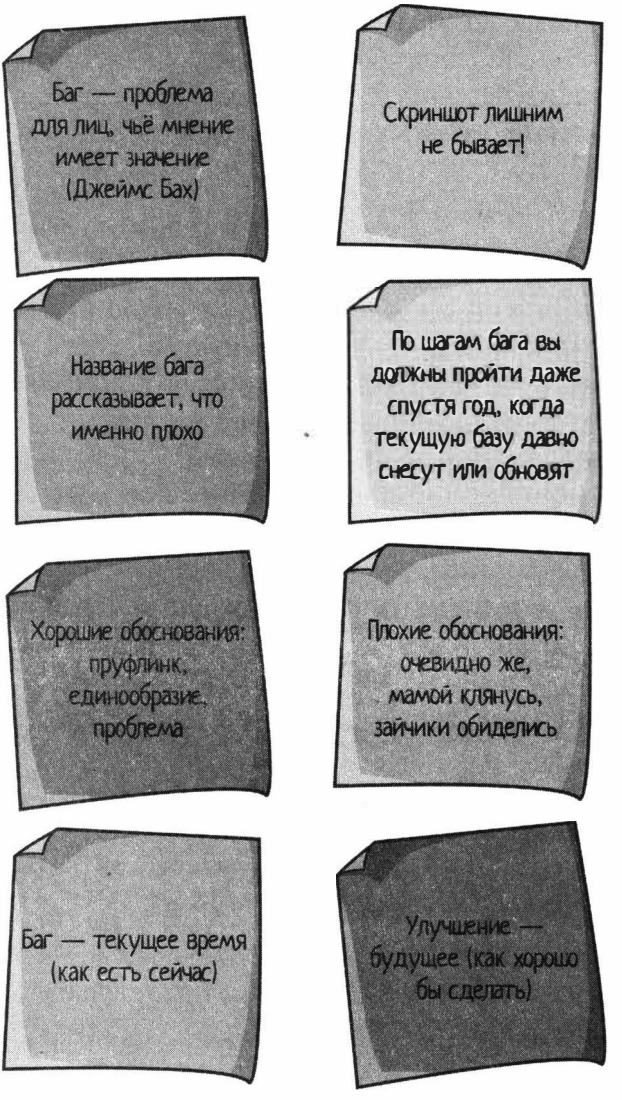
Полную версию плаката можно найти у меня в блоге, статья так и называется: «Плакат НЛО (найти, локализовать и оформить ошибку)»[92].

План действий:

1. Скачать плакат.
2. Отдать в типографию, напечатать на цветном принтере формата А1.
3. Повесить на стену и сверяться с ним при заведении бага.

Он яркий, прикольный, да еще и полезный. Три в одном! 😊

Ключевые моменты

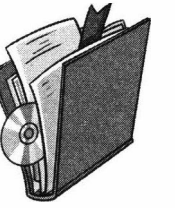


Вопросы для самопроверки

1. Чем плохо определение бага от Савина: «Отличие фактического результата от ожидаемого»?
2. Что такое баг?
3. Чем баг отличается от ошибки? А от улучшения? Или просто задачи?
4. Чем баг-трекинговая система лучше простого ворда?
5. Какая баг-трекинговая система самая лучшая?

6. Какой workflow задач наиболее правильный?
7. Что такое локализация бага?
8. Какие основные правила вложения аттачей? А скринов?
9. Зачем нужен метод бисекционного деления в тестировании?
10. Чем отличается название бага от названия улучшения?
11. Как правильно обосновывать баги? А какое обоснование будет плохим?

Портфолио



Продолжаем делать портфолио.

Во время написания чек-листов вы наверняка натыкались на баги. Или хотели что-то улучшить. Попробуйте оформить эти задачи — так, чтобы любому было понятно:

- баг — в чем проблема и почему его надо исправить;
- улучшение — зачем тратить на это время.

Идеи багов можно поискать у меня в блоге с метками «повсюду баги»[93] или «панбагон»[94] (вторая метка появилась позже). Все они вынесены в общий блог-пост «Баги повсюду. Сборная солянка»[95].

Для портфолио вполне хватит пары задач, но для тренировки чем больше, тем лучше!

Ответы на вопросы для самопроверки

1. Чем плохо определение бага от Савина: «Отличие фактического результата от ожидаемого»?

Оно работает только тогда, когда есть четкое ТЗ. Тогда да, отклонение от ТЗ является багом. В системе или самом ТЗ ;-)

2. Что такое баг?

Баг — это проблема для лиц, чьё мнение имеет для нас значение.

3. Чем баг отличается от ошибки? А от улучшения? Или просто задачи?

Баг — это и есть ошибка. В английской версии JIRA обычно тип задачи — Bug, в русифицированной — «Ошибка». В других системах может быть как-то иначе.

Баг — это проблема, которая уже есть в программе. Улучшение — какой мы хотим видеть программу в будущем. Задача — ни то, ни другое (например, «написать автотесты или документацию»).

4. Чем баг-трекинговая система лучше простого ворда?

Простой и удобный поиск, просмотр несколькими пользователями одновременно, сохранение истории изменений.

5. Какая баг-трекинговая система самая лучшая?

Вопрос с подвохом ;-)) Нет такой! У каждого свой фаворит: для вас он будет лучшим инструментом, а кто-то плеваться будет от него. Попробуйте разные и сделайте выводы.

6. Какой workflow задач наиболее правильный?

Аналогично, для каждой организации, команды, даже проекта внутри одной команды воркфлоу может быть разный. И это — нормально! Не надо навязывать то, что вы проходили на курсах как истину в последней инстанции.

7. Что такое локализация бага?

Поиск первопричины возникновения бага, а не одного из последствий.

8. Какие основные правила вложения аттачей? А скринов?

Название аттача — говорящее! Вложив аттач, сошлитесь на него в теле задачи (не просто «см. аттач»). На скрине отметьте стрелкой «куда смотреть» и уберите все лишнее, обрежьте его в Paint или другой программе.

9. Зачем нужен метод бисекционного деления в тестировании?

Для локализации багов, если нельзя понять причину, исходя из логов или анализа данных

10. Чем отличается название бага от названия улучшения?

- ✧ Баг: в названии — проблема! (оно УЖЕ есть, в настоящем).
- ✧ Улучшение: в названии — предложение «как должно быть» (в будущем).

11. Как правильно обосновывать баги? А какое обоснование будет плохим?

Правильное обоснование:

- ✧ Пруфлинк.
- ✧ Единообразие.
- ✧ Проблема реального пользователя, а не просто негативный тест-кейс.

Плохое обоснование:

- ✧ Очевидно же!
- ✧ Мамой клянусь!
- ✧ Зайчики обиделись.

Часть II

О том, что еще полезно знать и уметь тестировщику после усвоения знаний из части I

Глава 6
ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ

Ну, всё! Основные навыки тестировщика мы уже получили. А теперь поговорим о дополнительных плюшках.

И начнем мы с исследовательского тестирования, которое очень помогает нам в поисках багов!

В этой главе:

- проводим тест-туры;
- фиксируем результат.

Что такое исследовательское тестирование?

Исследовательское тестирование — тестирование, не требующее написания тест-кейсов. Но не стоит путать его с хаотическим тыканием всевозможных кнопочек, нет! Это более осознанный и вдумчивый процесс!

А вот начиналось все как раз с небольшого хаоса! Ну, мне так кажется ;-)

Сначала ведь и тестирования как такового не было. Разработчики сами проверяли свои творения. Но чем сложнее бизнес-логика, тем сложнее все удержать в голове и проверить. Тем более сложно усомниться в качестве своего детища. Всегда кажется, что «в моем коде багов нет!». Так что самостоятельное тестирование — не лучшая идея.

Потом стали тестировщики тестировать. А ведь вся теория про тест-дизайн, классы эквивалентности — это же не сразу появилось, а методом проб и ошибок. Поняли, что проверять все нереально, стали выделять классы. Поняли, что удержать все в голове нельзя — стали записывать тест-кейсы и чек-листы. А до этого ведь сплошной рандомайз был!

Потом поняли, что записывать тест-кейсы — унылое занятие, стали возвращаться к методам «без написания тестовой документации». Кто-то тестировал полным рандомом, кто-то более продвинутым. Одно дело — совсем хаотично что-то тыкать. Совершенно другое — иметь в голове план!

Так вот исследовательское тестирование — это когда мы ставим какую-то цель, прикидываем план и тестируем. Не хаотично, а преследуя конкретный результат. Например, найти все сообщения об ошибках в модуле.

А какие варианты есть, кроме него? Давайте упорядочим теорию про виды тестирования.

Виды тестирования: краткие определения

○ **Ad-hoc тестирование.**

Под ad-hoc тестированием будем понимать тестирование без использования спецификаций, планов и разработанных тест-кейсов: чистая импровизация[96].

У adhoc есть своя классификация[97]:

- ✧ **Buddy testing** — процесс, когда два человека, как правило, разработчик и тестировщик, работают параллельно и находят дефекты в одном и том же модуле тестируемого продукта. Такой вид тестирования помогает тестировщику выполнять необходимые проверки, а разработчику исправлять множество дефектов на ранних этапах.
- ✧ **Pair testing** — процесс, когда два тестировщика проверяют один модуль и помогают друг другу. К примеру, один может искать дефекты, а второй — их документировать. Таким образом, у одного тестера будет функция, скажем так, обнаружителя, у другого — описателя.
- ✧ **Monkey testing** — произвольное тестирование продукта с целью как можно быстрее, используя различные вариации входных данных, нарушить работу программы или вызвать ее остановку (простыми словами — сломать).

○ **Исследовательское тестирование.**

Более формальная версия ad-hoc — тестирование, не требующее написания тест-кейсов, но подразумевающее, что каждый последующий тест выбирается на основании результата предыдущего теста. А по Сэму Канеру[98] «исследовательское тестирование» — вдумчивый подход к ad-hoc тестированию.

○ **Сценарное тестирование.**

Классическое тестирование по предварительно написанным и задокументированным сценариям. Тест-кейсы, чек-листы — это всё сценарное тестирование.

Эвристики и мнемоники

Эвристики

Эвристика — это ваш прошлый опыт или опыт ваших коллег. Тестирование при помощи эвристики — тестирование, основанное на предыдущем опыте или информации о вероятности различных событий.

Свой опыт

— Ага, раньше мой разработчик постоянно косячил, когда расставлял границы. Вечно у него значение в оба интервала попадает. Значит, мне надо обязательно проверить граничные значения!

Информация о вероятности

— Ага, я тут читала статью «Ноль-не ноль». Автор уверяет, что в нуле постоянно есть баги. Хм, пожалуй, стоит проверить!

Чем больше мы работаем, тем больше эвристик собираем. Это общие эвристики — «серебряные пули», которые стоит проверять на любых ПО. И уникальные эвристики — под ваш проект или разработчика.

Поэтому любят набирать тестировщиков с ненулевым опытом. Вы уже сталкивались с ошибками в реальных проектах, пропускали баги в ПРОД. Это неприятно, но это ваш опыт, ваша копилка знаний. Которая поможет избежать ошибок в будущем.

Мнемоники

Мнемоника — слово или фраза, которая помогает нам что-то запомнить. Самая известная мнемоника: «Каждый охотник желает знать, где сидит фазан», помогающая запомнить цвета радуги.

В тестировании они тоже применяются. За рубежом — очень активно, там понимают, как это здорово и полезно!

Американцы вообще любят придумывать мнемоники. Примеры:

- SFPDO (San Francisco Depot);
- RCRCRC;
- FCC CUTS VIDS;
- FAILURE;
- CCD IS EARI;
- ...

И они молодцы! Потому что мнемоники помогают нам не продолбать какие-то важные проверки. Здорово, когда люди не стесняются использовать любые подручные средства, которые помогают им в работе. Не только официальные документы и громоздкие тест-кейсы, но еще и зарисовки, майнд-карты, придуманные на ходу мнемоники...

Каждый охотник желает знать, где сидит фазан

Да-да, придуманные на ходу! Почему бы и нет? Даже те мнемоники, что сейчас получили известность (типа San Francisco Depot), исходно были придуманы кем-то для личных нужд. Для своего проекта, своей команды. А потом автор понимал, что мнемоника пригодится не только ему, вот и делился с общественностью!

Лично вам чужая мнемоника может подойди, а может и нет. Она может не подойти под вашу систему, под ваши процессы или что-то еще. И на самом деле нет ничего такого плохого и зазорного в том, чтобы взять и написать свою мнемонику. Которая будет под вас, под вашу команду, под ваши особенности, под ваши процессы.

На своих курсах я предлагаю студентам самим придумать мнемонику. Конечно, это задание необязательное для получения сертификата. Потому что творчество — птичка своевольная, в клетке петь не будет.

Я тоже в свое время придумала мнемонику — БМВ[99]. Стараюсь ее активно пиарить. Ну, как минимум, студентам всем рассказываю, в книге вот о ней написала. Ведь она, правда, полезная, кучу багов находит! Однако других русскоязычных мнемоник, увы, не знаю...

Используйте чужие мнемоники, составляйте свои — если это помогает вам тестировать, то почему бы и нет?

Исследовательские туры Уиттакера

Джеймс Уиттакер (James A. Whittaker) в своей книге «Exploratory Software testing»[100] предложил методику проведения тестовых туров. Это один из вариантов исследовательского тестирования, который настолько прост в применении, что доступен даже начинающим!

Джеймс любезно разрешил мне перевести свои туры и выложить в открытый доступ. Вы можете найти полный список его туров с подробным описанием на портале Testbase в разделе «Как искать баги»[101].

А здесь я расскажу о том, как использовать туры, и мы пробежимся кратенько по всем имеющимся. Итак, как использовать этот раздел:

1. Выбрать тур.
2. Изучить его цели.
3. Поставить таймер на 2 часа (час, полчаса).
4. Провести исследование системы строго по целям тура. Ни на что не отвлекаясь — только «миссия» тура.
5. При необходимости повторить.

И пока вы новички, активно используйте эти туры. Они помогут вам увидеть идеи для тестов и найти баги!

○ **Туры поделовому центру (Tours of the Business District).**

Деловой центр — это функции, ради которых пользователи покупают и используют приложение. Это те киллер фичи[102], которые описывают маркетологи и которые упомянет любой из ваших пользователей при опросе «зачем им ваше приложение».

Тур по деловому центру фокусирует внимание на главных частях вашего приложения и показывает сценарии их использования вашими клиентами.

○ **Тур по путеводителю (The Guidebook Tour).**

Представьте, что вы турист. Вы не знаете, куда идти и что делать? Используйте путеводитель и четко следуйте ему. В приложении роль путеводителя представлена руководством пользователя или разделом Help.

Важно помнить, на чем делать акцент и где не отвлекаться. В путеводителе написано только самое главное, без дополнительных опций. Поэтому при тестировании важно уделить время основным сценариям, не сильно углубляясь. То есть сосоредоточьтесь на позитивных сценариях, не увлекаясь негативом.

○ **Денежный тур (The Money Tour).**

У туристов должна быть веская причина поехать в конкретное место. В Лас-Вегасе — это казино и стрип-клубы, в Амстердаме — кофейные магазины и улица красных фонарей, в Египте — пирамиды. Уберите эти достопримечательности, и туристы уйдут тратить деньги в другое место.

В ПО аналогично — у пользователя должна быть веская причина для покупки конкретного продукта. Найдите «продающие» функции приложения. Спросите маркетологов, которые проводят демонстрации. Узнайте у них, что нужно пользователям! Это и тестируйте.

○ **Тур по по ориентирам (The Landmark Tour).**

Ориентиры помогают не сбиться с пути. Джеймс Уиттакер вырос среди лесов и полей. Брат научил его использовать компас: ищешь Север и идешь туда. Можно не Север — просто выбираешь одно направление и придерживаешься его. Если долго идти по прямой, выйдешь к дороге — правило XXI века!

Тур по ориентирам для тестировщика похож — мы выбираем ориентир и идем к нему «сквозь» приложение. Уиттакер, например, выбирал ориентирами функции, полученные в турах по путеводителю.

○ **Интеллектуальный тур (The Intellectual Tour).**

В рамках тестирования по туру мы задаем приложению сложные вопросы. Как мы можем заставить ПО работать так тяжело, как только возможно? Какие функции нагружают приложение до предела? Какие входные значения и данные вызовут максимальную нагрузку? Какие внешние и внутренние данные приводят к специфичным результатам?

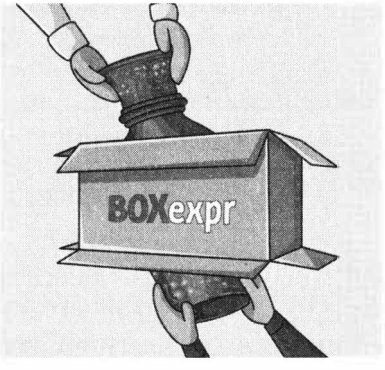
Вариацией этого тура является **Тур высокомерного американца (Arrogant American Tour)**. Вместо того чтобы задавать сложные вопросы, мы задаем раздражающие глупые вопросы, не иначе как для того чтобы привлечь к себе внимание.

Мы преднамеренно ставим препятствия, чтобы посмотреть на реакцию приложения. Вместо наиболее сложных документов мы рассматриваем наиболее красочные, инвертируем каждую страницу, печатаем только лучшие страницы. Это не поддается здравому смыслу... мы делаем это, потому что можем.

○ **Тур доставки FedEx (The FedEx Tour).**

FedEx — символ доставок во всем мире. Находясь в любой точке планеты, можно отправить посылку родным в Подмосковье. И нам неважно, через сколько временных складов она пройдет. Мы уверены, что родные получат груз в целости и сохранности. Так и у нас: данные проходят сквозь ПО, как посылки FedEx из точки А в точку Б.

В FedEx посылки путешествуют вокруг планеты. В тестировании данные проходят сквозь приложение, путешествуя по его закоулкам. Попробуйте идентифицировать входные данные, которые сохраняются в системе, и проследуйте за ними по приложению.



Попробуйте найти все функции, которые имеют отношение к входным данным. Как FedEx управляет своими посылками, так и вы включаетесь в каждый этап жизни данных.

○ **Внеурочный тур (The After-Hour Tour).**

В шесть вечера деловая жизнь останавливается, и работники разъезжаются по домам. Это время давки в транспорте и на улицах города. Туристы предпочитают держаться подальше от бизнес-районов в это время.

Но не тестировщики! Когда основные функции уже давно не работают, многие приложения продолжают пыхтеть. Они выполняют задачи поддержки и мониторинга: архивируют информацию, создают бекапы. Что будет, если грохнуть фоновый процесс? Или само приложение, пока оно работает в фоне? Если оно пересылает данные на сервер, можно рубануть не само приложение, а только Интернет. Нанесут ли подобные действия непоправимый вред?

Вариацией этого тура является **Тур утренней дороги на работу (Morning-Commute Tour),** цель которого — протестировать скрипты запуска приложения.

○ **Тур сборщика мусора (The Garbage Collector Tour).**

Сборщик мусора (дворник по-нашему) идет улица за улицей, дом за домой. Он пересекает окрестности в методической манере, останавливаясь около каждого дома на несколько мгновений, перед тем как пойти дальше. Однако так как он торопится, то не остается на одном месте слишком долго.

В тестировании тур — методическая проверка. Мы можем решить покопаться в интерфейсе, где мы идем экран за экраном, диалоговое окно за диалоговым окном (предпочитая, как дворник, кратчайшую дорогу) и не останавливаемся для тестирования в деталях, но «чекая» (от англ. check, проверка) очевидные вещи (как в Туре супермодели, описанном далее).

○ **Туры по историческим районам (Tours Through the Historical District).**

Исторические районы — части города, где есть старые здания и достопримечательности. В Бостоне они разбросаны по всему городу и соединены только пешеходными тропами. В Кёльне есть «старый город» — часть города, не тронутая современной экспансией.

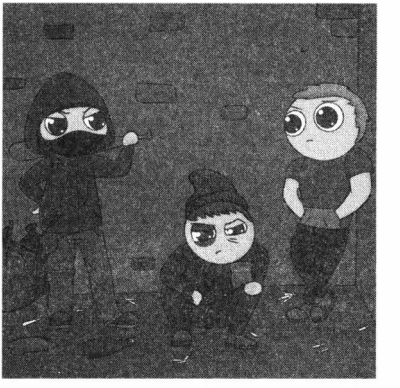
В ПО исторические районы могут быть так же слабо соединены, как в Бостоне, или сосредоточены в одном месте, как в Кёльне. Исторические районы в ПО представляют собой:

- ✧ унаследованный код (legacy code);
- ✧ функции, созданные в предыдущих версиях;
- ✧ исправления багов.

Туры по историческим районам проверяют старую функциональность и исправление ошибок.

○ Тур по плохому району (The Bad-Neighborhood Tour).

В каждом городе есть «плохие» районы — преступные. В ПО тоже есть плохие раойны — разделы кода, населенные багами. Разница между обычным человеком и тестировщиком заключается в том, что первые стараются избегать плохих районов, тогда как вторые уделяют им настолько много времени, насколько это возможно.



Чем сложнее функциональность, тем вероятнее ошибки. Чем слабее разработчик, тем вероятнее баги в его коде. Используйте баг-трекер для анализа. Посмотрите — в каком компоненте больше всего ошибок? Там и ищите!

○ Музейный тур (The Museum Tour).

Музеи с антиквариатом — любимое место туристов. Они собирают несколько тысяч посетителей в день. Антиквариат в коде залуживает такого же количества внимания от тестировщика. В нашем случае под антиквариатом мы понимаем legacy code («устаревший код» — распространенное название, поэтому оставила далее в тексте без перевода).

Нетронутый legacy code легко найти быстрым просмотром даты и времени создания в репозитории (месте хранения кода). Многие репозитории также поддерживают изменения. Таким образом, тестировщик может сделать небольшое исследование, чтобы увидеть, какая часть старого кода недавно изменялась.

Старый код, измененный в новой сборке, — плодотворная почва для поиска багов! Так как исходный разработчик мог уйти, и документация обычно бедная, легаси код сложно изменять, сложно ревьюить, и разработчики уклоняются от покрытия его юнит-тестами, которые они обычно пишут для нового кода.

○ **Тур предыдущей версии (The Prior Version Tour).**

Релиз продукта — накатывание обновления поверх предыдущей версии. Было бы неплохо при релизе выполнять все сценарии и тест-кейсы, которые были доступны в прошлой версии. Мы должны убедиться, что функциональность, которую пользователи уже используют, продолжит работать и в новой версии.

Если имеющая функциональность изменилась — проверяем все сценарии использования функционала. Не только основные сценарии, но и альтернативные ветки. Всё должно работать!

Если функционал был удален — проверяем, что всё, связанное с ним, продолжает работать. Что удаление никак не повлияло на работу системы. По сути, это регрессионное тестирование.

○ **Туры по развлекательным районам (Tours Through the Entertainment District).**

В каждом отпуске туристам необходим перерыв в их плотном графике. Посещение развлекательного района, шоу или длинный тихий обед вне основного пути создают такие перерывы. Туристы приходят в развлекательный район ради отдыха, а не достопримечательностей.

В большинстве приложений есть сходные функции. Например, деловой район в текстовом редакторе — набор функций для создания документа, подготовки текста, вставки графики, таблиц и рисунков. Развлекательный район — функции для разметки страницы, форматирования, изменения фона. Другими словами, работа заключается в создании документа, а развлечение — в наведении красоты.

Туры по развлекательным районам исследуют, скорее, второстепенные, нежели основные функции, и позволяют убедиться, что они дополняют друг друга без противоречий.

○ **Тур актера второго плана (The Supporting Actor Tour).**

Приложения заворачивают свои предложения в красивую обертку и старательно «ведут» пользователя нужным путем: «Нажми сюда и увидишь классные шортики! А теперь щелкни здесь и оформи заказ. Заказ оформи, говорю!».

Но это же неинтересно! ;-) Тестировщики знают все о непредсказуемости пользователя. Именно поэтому они задают приложению коварные вопросы: «А если так? А вот так?», отмахиваясь от типового

шаблона, как от назойливой мухи. Наша задача в туре — проверить функции, которые не главные, но находятся рядом с ними.

○ **Тур глухого переулка (The Back Alley Tour).**

Куда поехать в отпуск? Что такое «хороший тур»? Если думать лень, берите тур по знаменитым местам, не прогадаете. Раз туда ломятся тысячи — значит, не просто так. Значит, хороший тур, хорошие места. Альтернативой таким турам является посещение мест, в которые больше никто бы не захотел пойти. Ну, например, тур по публичным туалетам.

В терминах исследовательского тестирования Тур глухого переулка — наименее привлекательные для пользователей функции. И наименее часто используемые (когда есть статистика)

Интересная вариация этого тура — **Тур смешанного места назначения (Mixed-DestinationTour)**. Посетите комбинацию самых знаменитых и наименее популярных мест. Вы можете думать об этом как о Туре по ориентирам (Landmark Tour) с большими и маленькими метками.

○ **Тур полуночника (The All-Nighter Tour).**

Тур полуночника также известен как **Тур завсегдатая клуба (Clubbing tour).** Завсегдатай не идет домой вечером, он направляется в ночные клубы и тусит там всю ночь. Тур не должен останавливаться — всегда будет еще один клуб или еще один последний стакан выпивки. Некоторые верят, что такие туры проверяют характер. Сможешь продержаться? Выдержишь всю ночь?

В исследовательном тестировании вопросы те же: может ли ваше приложение продержаться? Как долго оно может работать и обрабатывать данные, перед тем как развалится? Не закрывайте приложение как можно дольше, включая всё новый и новый функционал.

Альтернативой Туру полуночника является тур «наоборот» — **Тур чашки кофе**. Это когда вы выключаете приложение и долго с ним не работаете. А потом пытаетесь снова запустить и продолжить. как ни в чем не бывало. Допустим, мы играли в игру, а потом забили на нее. Прошло полгода, вышло 10 обновлений, рассчитанных на то, что их ставят постепенно. А тут вы, такой красивый, вернулись и пытаетесь накатить сразу все!

○ **Тур чашки кофе**

> Этот тур я решила вынести отдельно, потому что вижу его немного по иному, чем Уиттакер ;-)

Тур чашки кофе — это когда ты работал-работал с приложением, а потом решил немного отдохнуть. Пошел и выпил кофе, например. А потом вернулся и решил продолжить работу.

Обычно попить чай/кофе — это недолго, минут на пять. Именно это мы и делаем — уходим на 5 минут, оставив приложение на каком-то этапе. Желательно, чтобы это был один из проходных этапов, то есть не заключительный — например, начали оформлять заказ в интернет-магазине и не закончили, отвлеклись. А потом вернулись и пытаетесь продолжить.

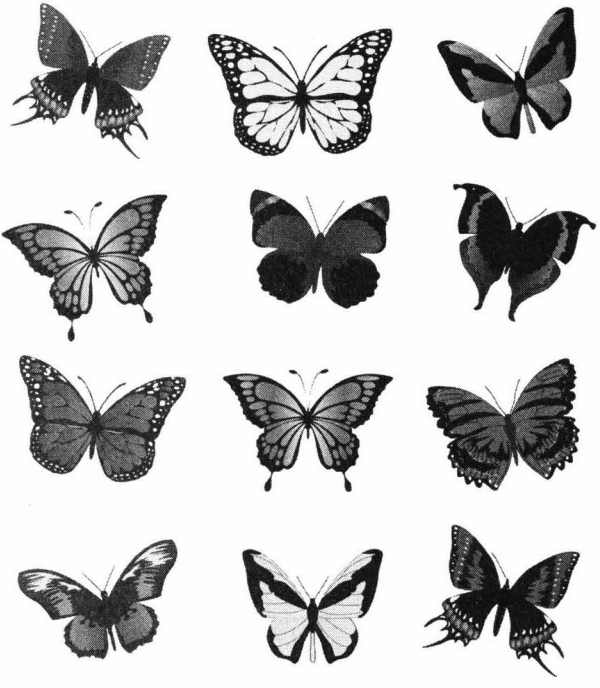
○ **Туры по туристическим районам (Tours Through the Tourist District).**

В каждом городе есть районы притяжения туристов. Там много сувенирных лавок, ресторанов и других мест для максимизации времяпрепровождения туристов и увеличения прибыли местных продавцов.

Здесь можно найти магнитики на холодильник и предметы коллекционирования. В тестировании вместо набегов на сувениры мы создаем небольшой чек-лист для набегов на некие функции приложения.

○ **Тур коллекционера (The Collector's Tour).**

Кто-то собирает бабочек, кто-то гербарий, а кто-то — баги. Цель коллекционера — собрать полную коллекцию бабочек каждого вида, каждого возраста. Аналогично и в тестировании — выбираем, что будем коллекционировать, и собираем полный комплект.



Идея тура — пройтись в приложении везде, где только можно, и задокументировать все выходные данные.

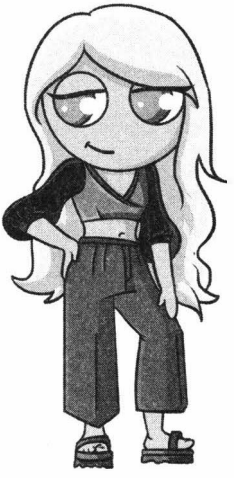
○ **Тур одинокого бизнесмена (The Lonely Businessman Tour).**

У Джеймса Уиттакера есть друг, который часто путешествует по бизнес-делам. Он посетил множество величайших стран мира, но большинство из них видел только из аэропорта, отеля и офиса. Чтобы исправить ситуацию, он стал заказывать отель как можно дальше от места встречи. И потом гулял, катался на велосипеде или брал такси до офиса, что помогало ему хоть немного увидеть город.

В тестировании подобный подход очень эффективен. Идея в том, чтобы посетить (и, конечно, протестировать) функции, которые расположены дальше всего от точки входа в приложение. Какие функции требуют больше всего щелчков, чтобы до них добраться? Выберите одну из таких и протестируйте путь к ней.

В каких функциях нужно пройти максимальное число экранов, чтобы получить результат? Выберите их и протестируйте. Идея в том, чтобы путешествовать сквозь приложение как можно дольше, прежде чем достигнете места назначения. Предпочитайте длинные пути коротким. Выберите страницу, похороненную в приложении глубже всего.

○ **Тур супермодели (The Supermodel Tour).**



В этом туре мы обращаем внимание только на первое впечатление. Снаружи супермодель — ВАУ!, а вот какая внутри: добрая и милая или ужасная стерва, 98% подписчиков не касается, ведь они никогда не встретятся за ланчем.

В этом туре вы должны скользить по поверхности. Чтобы вы ни делали, не погружайтесь глубоко. Этот тур не проверяет функции или процедуры, он проверяет, как приложение выглядит и какое первое впечатление производит. Не фокусируйтесь на функциональности или реальном взаимодействии. Только на интерфейсе.

○ **Тур «Второй бесплатно» (The TOGOF Tour).**

Этот тур — игра в акронимы с *Buy One Get One Free (BOGOF)*, что очень популярно в интернет-магазинах. В тестировании мы не будем ничего покупать, а поменяем акроним на *Test One Get One Free (TOGOF)*.

TOGOF — простой тур, созданный только для тестирования многократно повторяющихся копий одного приложения, запущенных одновременно. Начните тур с запуска своего приложения, потом запустите вторую копию, потом третью. Теперь запустите на них функции, которые отжирают память или пишут на диск.

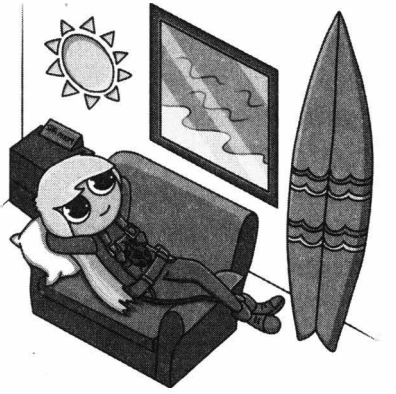
○ **Тур шотландского паба (The Scottish Pub Tour).**

Пабы расположены в маленьких, захудалых домах на отшибе. Мимо такого пробежишь мимо, опасливо озираясь по сторонам. И никогда не угадаешь, что именно там можно выпить отменного пива.

В тестировании — как в жизни. Если приложение большое, в нем обязательно будут клевые функции, о которых мало кто знает. Примеры из жизни: Microsoft Office, eBay, Amazon, MSDN.

Нам нужно найти и протестировать функции, которые сложно найти, если заранее о них не знаешь. Эти функции передают «из уст в уста» наученные опытом коллеги, и их можно нагуглить. Но с наскока простой пользователь их не найдет. Так и житель мегаполиса пройдет мимо крутого паба, не зная, что внутри классное пиво и вежливые официанты.

○ **Туры по отельным районам (Tours Through the Hotel District).**



Отель — убежище для туриста. Это место, куда можно сбежать из давки и суеты популярных мест для небольшого отдыха и расслабления.

Сюда приходит тестировщик, уйдя от главной функциональности, чтобы потестировать второстепенные или сопутствующие основным фичам функции, которые часто игнорируются в тест-плане.

○ **Тур, отмененный из-за дождя (The Rained-Out Tour).** ☆

В мае в Таиланде начинается сезон дождей — время внезапной отмены экскурсий. Причем отменяют их за день до назначенной даты. Заранее непонятно, будет дождь или нет. И вот мы из-за дождя остались дома.

Для туриста отмена — плохо. Для тестировщика — хорошо, даже идеально! Тестировщик просто обязан использовать отмену всегда и везде, где только возможно. Идея тура в том, чтобы начать операцию и остановить ее.

И я говорю вам по своему опыту — этот тур чаще остальных ловит ошибки. Поэтому я его и отметила звездочкой. Мой фаворит ;-) Используйте его всегда и везде!

○ **Тур домоседа (The Couch Potato Tour).**

В группе всегда найдется один человек, не слушающий экскурсию. Он стоит позади толпы. Ему скучно, у него нет энергии и сил. Он думает: «Зачем я вообще платил за поездку?! Посидел бы дома, попил пивка...». Гиду приходится выпрыгивать из штанов, чтобы заинтересовать скучающего домоседа.

И наша задача такая же — лениться! Делать так мало работы, как только возможно: соглашаться со всеми дефолтными значениями, оставлять поля пустыми, заполнять в форме минимум значений...

○ **Туры по захудалым районам (Tours Through the Seedy District).**

Это непривлекательные места, о которых расскажет редкий путеводитель. Они полны мошенников и сомнительных личностей, и лучше обходить их стороной. Тем не менее они привлекают определенный класс туристов.

Для тестировщика обязателен тур по этим районам для выявления тех опасностей, которые могут подстерегать пользователей продукта. Для такого тура отлично подойдут входные данные, ломающие приложение или способные каким-либо образом ему навредить.

○ **Тур саботажника (The Saboteur Tour).**

В туре саботажника мы будем подрывать приложение (нарушать его работу) всеми возможными способами. Мы попросим приложение прочитать информацию с файла на диске, а потом подстроим все так, чтобы операция провалилась: испортим файл, отключим диск, оборвем соединение...

Наши действия:

- ✧ заставить приложение выполнять какое-то действие;
- ✧ понять, какие ресурсы необходимы для успешного выполнения этого действия;
- ✧ удалить или спрятать эти ресурсы.

○ **Тур антисоциального типа (The Antisocial Tour).**

Антисоциальный человек делает все «назло», особенно, если его насильно помещают в противный социум.

Антисоциальный тур требует вводить наименее хорошие входные данные и/или заведомо плохие. Если реальный пользователь должен делать X, тестировщик в антисоциальном туре никогда не должен делать X, а вместо этого найти наименее значимые входные данные.

Есть три специфичных пути достичь антисоциального поведения, которые автор организовал в три подтура:

- ✧ **Opposite tour** — вводить наименее хорошие входные данных везде, где только возможно: данные, которые выходят из контекста, явно глупые или полностью бессмысленные.
- ✧ **Illegal inputs** — вводить значения, которые не должны быть введены: входные данные неверного типа, неверного формата, слишком длинные, слишком короткие и т. п. Аналогичен туру по плохим районам.
- ✧ **Wrong turn tour** — выполнять действия в неправильном порядке. Взять группу правильных действий и перемешать их так, чтобы последовательность оказалась неверной.

○ **Обсессивно-компульсивный тур, или Тур невротика (The Obsessive-Compulsive Tour).**

Смотрели «Теорию большого взрыва»? Вот у Шелдона как раз было OCD — Obsessive-compulsive disorder, обсессивно-компульсивное расстройство. Оно заставляло Шелдона стучать в дверь и говорить «Пенни» обязательно три раза, даже если дверь открывали после первого стука.

В реальной жизни сложно представить себе ситуацию, в которой OCD помогает жить. Но быть обсессивным в тестировании — выгодно!

OCD-тестировщики вводят одинаковые значения снова и снова. Они выполняют одно действие снова и снова. Они повторяют, отменяют, копируют и вставляют. Снова и снова, снова и снова. Игра такая: повторение — мать учения.

Мои любимые туры

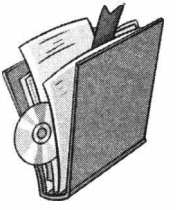
А здесь я уже говорю от себя — это моя подборка, а не от James A. Whittaker, автора туров. Итак, самые клевые:

1. **Тур, отмененный из-за дождя** — ВСЕГДА находит баги, иногда очень крутые! Главное, помните, что отменить действие — это не только «закрыть браузер» ;-)
2. **Тур «второй бесплатно»** — при concurrency тоже баги очень часто вылезают, рекомендую.
3. **Интеллектуальный тур** — ну, а куда без него? Сложные вопросы разработчик мог просто не продумать.
4. **Тур чашки кофе** — включил приложение и пошел себе...
5. **Тур полуночника** — что если оно будет работать очень долго?
6. **Тур по путеводителю** — если у нас есть документация, она должна работать. И примеры из нее должны работать, обязательно это все проверяйте!

Вопросы для самопроверки

1. Чем отличается Ad hoc от Monkey testing?
2. Когда проводить сценарное тестирование, а когда — исследовательское?
3. Можно ли самому придумать мнемонику или тур для исследований?
4. Зачем нужны мнемоники?
5. Чьи исследовательские туры переведены в этой главе?

Портфолио



Продолжаем делать портфолио.

1. Выбрать для своего приложения три лучших тура, которые стоит пройти в первую очередь.
2. Засечь 20 минут.
3. Исследовать по любому из туров.
4. Записать результаты.

Как должен выглядеть отчет о результатах?

1. Какие три тура выбрали, почему?
2. Ссылка на тур.
3. Что успели протестировать в ходе путешествия, на что обратили внимание?
4. Какие баги нашли?

Ответы на вопросы для самопроверки

1. Чем отличается Ad hoc от Monkey testing?

Ничем. Monkey testing — подвид Ad hoc (свободного тестирования). Самый простой и самый глупый подвид ;-)) Тем не менее он тоже работает!

2. Когда проводить сценарное тестирование, а когда — исследовательское?

Нет правильного ответа, оба подхода хороши. Главное, чтобы ответ был не «исследовательское» — когда лень документацию писать. Отчеты и план тестирования всегда стоит составить. Просто в сценарном подходе — более подробно. Но исследовательское тестирование новички провести не смогут, разве что по турам Уиттакера.

3. Можно ли самому придумать мнемонику или тур для исследований?

Конечно, можно! Даже нужно, так вы сделаете тесты под свой проект, а не общие.

4. Зачем нужны мнемоники?

Чтобы не забыть проверить что-то. Это как чек-лист, только в одно слово или фразу. Запоминается проще!

5. Чьи исследовательские туры переведены в этой главе?

James A. Whittaker, из его книги «Exploratory Software testing».

Глава 7
ТЕСТИРОВАНИЕ ДОКУМЕНТАЦИИ

Когда у нас есть четкое ТЗ, мы обязательно проверяем, что всё работает, как там написано. Но и в документации бывают баги, их мы и будем сегодня искать.

В этой главе:

- изучаем типы документации;
- проверяем ее.

Какая бывает документация?

Когда мы говорим о тестировании документации, то обычно подразумеваем тестирование требований, ТЗ. И это тестирование на полноту, однозначность и прочая. Смотрим, как новый функционал будет коррелировать со старым, не будет ли проблем. Заранее продумываем свои тесты, обсуждаем реализацию...

Однако, помимо ТЗ, есть еще куча другой документации, которую тоже стоит проверить. Как минимум, вычитать, нет ли ошибок. Давайте посмотрим, какая бывает документация.

ТЗ — требования

ТЗ — техническое задание. Это основной тип документации для тестировщика. Если ТЗ есть, его всегда надо тестировать. Как само ТЗ, так и систему на соответствие этому ТЗ. Помните самые простейшие баги? Когда есть четкие требования, но система работает по-другому.

Виды требований

В каком виде могут быть требования? Есть разные варианты:

- **техническое задание** — сухое описание того, как система должна работать. Что уметь делать и как реагировать на ошибки;

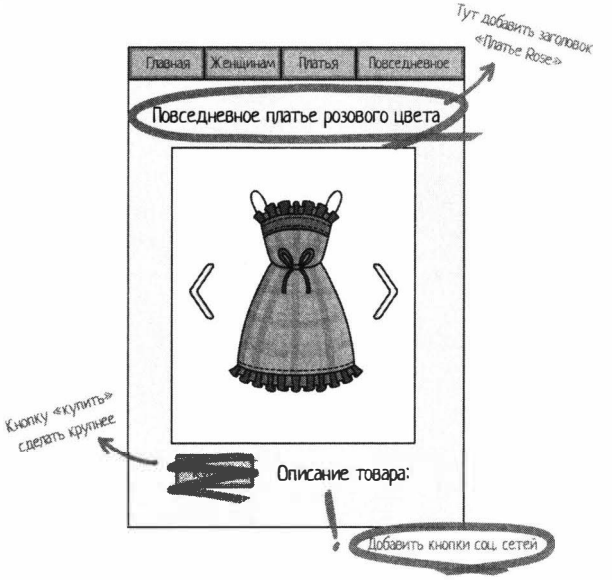
- **вариант использования** — описание через взаимодействие пользователя с системой:
  - ✧ пользователь делает то...
  - ✧ система в ответ делает се.

Мы подробнее поговорим о такой документации в следующей главе. Основное ее отличие от технического задания — более «читаемый» текст, так как вместо «Доступна авторизация через соцсети» написано: «Пользователь входит в систему с помощью соцсетей и делает то-то». Уже понятно, для чего эта функция — для выполнения какого-то действия;

- **пользовательский сценарий** — еще более «человеческий» и читаемый вид требований. Теперь уже не просто описано взаимодействие пользователя с системой, а рассказана целая история. Что за пользователь, зачем ему эта система, почему он сюда пришел, что хочет сделать, какие эмоции ощущает...

В общем, градация от «сухое ТЗ» до «интересная история» широкая. Как ваш аналитик решил записать, так и будет.

- **картинка** — да, да, именно картинка! Вид ее может быть любой:
  - ✧ блок-схема;
  - ✧ диаграмма состояний и переходов;
  - ✧ скриншот, раскрашенный красным: «тут должно быть вот так, а тут эдак»;



✧ схема движения транзакций;

✧ ...

То есть снова есть простор для творчества. Хотя обычно, конечно, используют от силы блок-схемы. Но и на том спасибо ;-)

○ **что-то другое**. Я уверена, что могут быть и другие варианты. Но мы же не на аналитиков учимся, поэтому ограничимся базовыми.

У меня на одном из проектов, например, ТЗ было в виде презентации PowerPoint. Вот как генеральный «продавал» проект спонсорам, так мы и должны были сделать. Картинки + немного текста — вот и всё ТЗ. Что непонятно, уточняйте устно.

Размер требований

Размер требований может быть любой. Все зависит от проекта, на который вы попадете:

○ **подробное описание** — обычно встречается на гос. проектах. Ооооо, сколько там ТЗ!

Я участвовала в одном таком проекте, где описание функционала, который описывается одной строчкой (есть авторизация через email) занимало 10 листов А4 — пошаговое описание интерфейса с картинкой через строку. Почти как в этой книге, даже чаще ;-)

Проблема, думаю, понятна: когда у нас на мелкую функцию столько документации, на всю систему ее будет еще больше. Поддерживать ее в актуальном состоянии просто невозможно — это надо нанять человека, который весь день будет только этим и заниматься.

Но зато начинающему тестировщику очень легко работать с такой документацией. Там должно быть прописано ВСЁ. И позитивные сценарии, и негативные. Читай да выполняй, сверяясь с требованиями.

- Плюсы:
  - ✧ требования может проверить даже новичок (даже если там бухгалтерия или еще что-то сложное, в чем он ни в зуб ногой);
  - ✧ если это ТЗ для интеграции — у заказчика будет возникать меньше вопросов «А что будет, если...?». Ведь всё уже прописано в ТЗ.
- Минусы:
  - ✧ неактуальность документации (я бы написала «сложно поддерживать актуальность», но она все равно в итоге будет неактуальна — слишком ее много);
  - ✧ заказчик может пропустить требования: когда их слишком много, он вроде всё прочитал, но половину забыл. Сделал интеграцию неправильно, а потом начинаются выяснения: «Кто виноват и за чей счет исправлять будем».

- **краткое описание** — буквально одна страничка на довольно большой объем работы. Там не расписываются подробно все ошибки или «что может пойти не так» — это, скорее, напоминалка, какие функции должны быть.

Однако тут описано всё необходимое для разработчика. Просто без кучи скриншотов, воды и прочего лишнего.

- Плюсы:
  - ✧ легко поддерживать актуальность;
  - ✧ при грамотном описании все понятно, без «воды».

- Минусы:
  - ✧ начинающий тестировщик может пропустить тесты — он не догадается, что тут можно было бы проверить (иногда ведь в ТЗ расписываются и все сообщения об ошибках, даже думать не надо, читаешь и делаешь);
  - ✧ если отдать такое ТЗ третьим лицам (например, заказчику или его тестировщикам), получите много вопросов: «А что будет, если...?», и придется дополнять требования.

Однако минусы не так уж страшны. Так что если есть возможность — пишите кратко!

- **нечто среднее** — это как раз то, что вы получите на своем проекте. Если нет особых требований на «много текста», то лучше стремиться к краткому варианту. Поддерживать проще, воспринимать тоже. Всегда можно немного разбавить дополнительной информацией — например, сообщениями об ошибках.
- **требований нет!** — бывает и такое. Приходите на проект: «На, держи, тестируй». А где документация? А нету ее, так проверяй. Что делать?

На самом деле не бывает такого, что требований нет. Они всегда есть. Просто иногда — в голове. Заказчика, аналитика, архитектора...

Ваша задача в этом случае — эти требования узнать. И сохранить:

- ✧ Найти носителя информации.
- ✧ Выяснить все подробности.
- ✧ Кратенько записать.
- ✧ Попросить его проверить.

Тут вы можете сказать:

— Эй, але! Я тестировщик! Поставлю задачу аналитику и пусть он делает.

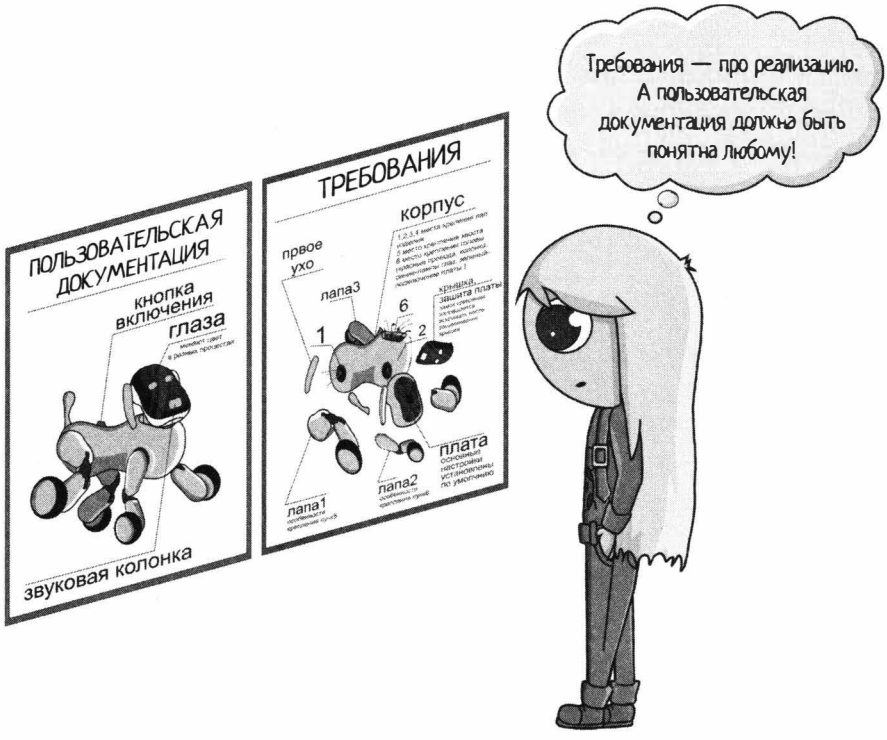
Ну-ну. Вы думаете, если до вас процветало разгильдяйство, то сейчас все резко изменится? Что нужно было просто задачку на аналитика поставить? Если документации до сих пор нет — значит, всех всё устраивает.

Хотите изменить мир? Начните с себя. Заставить других — так не работает. Если вам плохо без отсутствия документации, сделайте ее сами. В следующей главе я расскажу, как это можно сделать. Если же вас не парит отсутствие документов — живите как живёте.

Пользовательская документация

Это вся документация, которую вы передаете пользователю. Описание системы и ее функционала, скриншоты интерфейса, даже требования — всё, что находится в открытом для пользователя доступе.

Чем она отличается от ТЗ? Тем, что в ТЗ вы можете использовать сленг, сокращения, не писать какую-то очевидную для коллег информацию. А вот пользователь ваших сокращений не знает, ему пишите понятнее.



У меня на одном из проектов в вики-системе были разные разделы:

- **внутренний** — пользователи его не видят. Там все потроха. Описание реализации, инструкции по исправлению кода, описание автотестов...
- **внешний общий** — доступен всем пользователям. Тут лежат инструкции по разворачиванию системы (для администраторов) и по работе в ней (для пользователей), описание API-методов, примеры их вызовов... Это всё — пользовательская документация;
- **внешний заказчика** — доступен только конкретному заказчику, там описана специфика его сборки (коробочный продукт, который модифицируется под нужды пользователя). Плюс требования к новым доработкам.

Таким образом, в пользовательскую документацию могут входить и примеры, и требования. Но... Зачем смешивать все в единую кучу? А как вы потом будете на своем проекте искать документацию? Я предлагаю все же разделить:

- требования — требования для реализации;
- пользовательская документация — информация о работе системы.

Ограничивается ли документация этими пунктами? Разумеется, нет!

Примеры

Всегда проверяйте примеры. Просто ВСЕГДА!

Потому что пользователи — люди ленивые. Если они хотят познакомиться с системой, то не будут придумывать свои данные, а возьмут ваш пример. И если он не работает, зачем продолжать работу с программой?

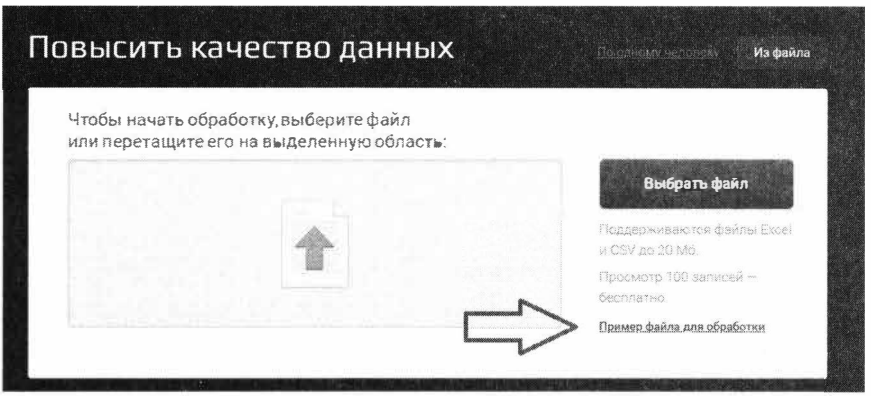
Другой вопрос — а что относится к примерам? Что проверять-то нужно? Давайте разбираться!

Файл-образец

Возьмем для примера Дадату. Туда можно грузить файлики, и система «причешет» данные внутри. Как вы будете проверять, подойдет ли вам Дадата?

1. Возьмете свой файлик и попробуете сразу на нем.
2. Возьмете пример из системы.

Первый вариант возможен, да. Но часто ли у вас под рукой нужный файл? Это надо еще пойти в систему, сделать выгрузку... А зачем напрягаться, если вас Дадата не устраивает исходно? Конечно, сначала проще взять образец и посмотреть на нем, что умеет система.

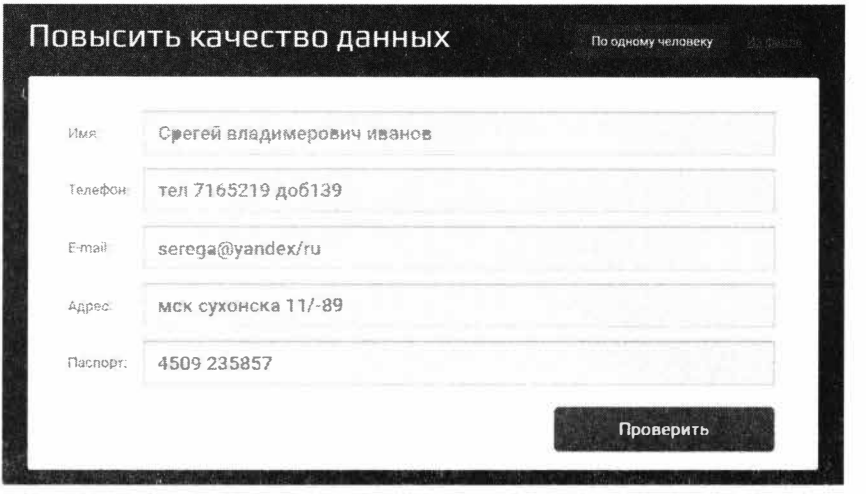


Если ОК, всё устраивает, то пробуем на своих данных. Но это обычно второй шаг, а начинают с готового примера. Поэтому очень важно проверять в регрессе, что файл-образец:

- скачивается (вдруг ссылка побилась??);
- обрабатывается без ошибок, выдает правильный результат.

Предзаполненные поля в форме ввода

Что если у нас просто форма ввода? И туда можно добавить пример! Вот как в Дадате сделано.



Все поля заполнены по умолчанию. Можно просто ткнуть на кнопку **Проверить** и посмотреть результат. А потом уже обрабатывать свои данные. И поверьте, большинство просто ткнет на кнопку для начала просто потому, что так проще.

Примеры вызова API-методов

Что может быть приятнее, чем скопировать пример и отлаживаться на нем? То есть сначала послали «запрос, который 100% работает», а потом уже варьируем под свои данные.

Это очень важно для интеграции. Вы ведь пытаетесь соединить свою систему с чужой. Чужую — как она работает — вы почти не знаете. Спасает только наличие ТЗ.

Но что если требований тааааак много и они довольно сложные, а интегрировать уже пора начинать? Сначала стоит выполнить простейший сценарий. А где его взять? Конечно, из примера! Выполняем пример

вручную. Работает? Тогда пишем вызов уже из кода. Написали? Теперь уже сидим и вчитываемся подробнее, разбирая всякие альтернативные сценарии.

Именно поэтому примеры очень важны. С них начинают работу с системой.

Другие примеры

Примеры могут быть в любой документации: лендинг-страница, пригласительное письмо, документация на работу системы... Посмотрите, сколько пунктов в этой статье, и в любом из них могут скрываться примеры!

Ну а я только что рассказала об основных. Не забудьте проверить, есть ли у вас в системе предзаполненная строка или файл-образец. И тестируйте их каждый релиз (желательно автоматически, но тут уж как пойдет).

Итого по примерам

Примеры очень важны. С них начинают работу с системой.

- Если у вас их нет — просите добавить.
- Если они есть — тестируйте их!

Если вы делаете внешнее API для заказчика или вообще публичное, опишите его и добавьте примеров. Именно с них пользователь начнет знакомство с методом.

Но и в «простом» интерфейсе есть место для примеров. Можно облегчить пользователям жизнь, подготовив файл-образец или предзаполнив форму ввода. И они обязательно этими примерами воспользуются. Поэтому примеры мы ОБЯЗАТЕЛЬНО тестируем. Всегда. Это — лицо нашего приложения, оно должно работать.

Письма от системы

О, это тоже очень важный пункт! Сейчас многие системы присылают приветствия, напоминания, уведомления... Так вот, эти письма — это тоже документация, причем очень важная ее часть!

Типичные проблемы писем — плейсхолдеры, которые:

- не разименовались;
- разименовались, но неправильно.

Что такое плейсхолдер? Это заглушка, которая меняется на нужное значение при отправке письма. Разработчик же не знает заранее, как вас зовут. Но допустим, что он хочет сделать именное письмо:

> Привет, Ольга! Спасибо за регистрацию на нашем сайте...
> Привет, Андрей! Спасибо за регистрацию на нашем сайте...
> Привет, Сладкий пупсик! Спасибо за регистрацию на нашем сайте...

Разработчик пишет письмо, оставляя плейсхолдер вместо имени:

> Привет, `$username`! Спасибо за регистрацию на нашем сайте...

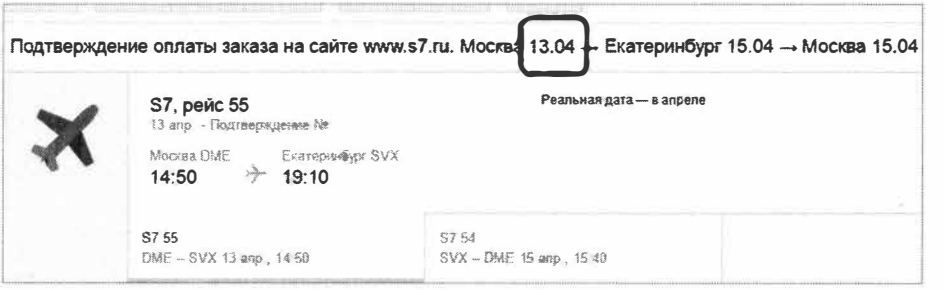
И говорит в коде: «Вместо `$username` подставь имя, которое пользователь ввел при регистрации. Возьми значение из такого-то поля».Теперь, когда Маша-Ваня-Петя регистрируются на сайте, они получают личное письмо со своим именем!

Обычно такие подмены делают на имена и даты. Например, дата, на которую куплен билет. Главное — подставить нужное значение! А с этим бывает беда...

> В апреле 2017 года я летела в Екатеринбург, на конференцию DUMP[103]. Купила билеты в S7 Airlines заранее — в феврале. Перепроверила даты: вроде все правильно, вылетаю 13.04.

Упорный бот пытался впарить мне отель через их сайт и отказ не принял. А 02.02.2017 он прислал мне «напоминалку», что у меня скоро полет... И поставил дату... «Сегодня» o_O

Эй, але, я покупала билеты на апрель! Холодный мороз по коже — а вдруг я ошиблась? Нашла письмо с билетами, нет, все хорошо:



Вот же, 13.04!

Значит, просто накосячили в письме. Благо, понятно как: раз дата сегодняшняя — то вместо моего реального времени полета вставили `sysdate`. Или там и полагалось быть `sysdate`, но письмо-то должно было быть отправлено в апреле...

Проблемы бывают самые разные:

- иногда письмо приходит с опозданием и получается «оплатите счет до вчера»;
- иногда тебе пишут «Привет, username!». Забыли подставить значение ;-)
- иногда неправильно склоняют имя;
- иногда неправильно определяют пол;
- ...

То есть в первую очередь мы проверяем в письмах все переменные и плейсхолдеры — правильно ли они подставились? Если это строка с именем, проверяем такие моменты:

- а если я пустым оставлю поле?
- а если на другом языке напишу?
- а если спецсимволы введу?
- а если очень длинное имя?

Аналогично с датой:

- когда придет письмо — сразу после регистрации, за день до назначенного времени?
- правильная ли дата будет в письме?

Ну и, конечно, надо хотя бы разок вычитать все письма на орфографические ошибки!

Сообщения об ошибках

Да, да, это тоже документация! Поэтому их надо все найти и проверить. Зачем? Давайте я расскажу вам про Pretty roses[104] (пример взят из Интернета, исходного автора давно утеряли):

PASSWORD EXPIRED

Sorry, your password has been in use for 30 days and has expired – You must register a new one.

roses

Sorry, too few characters.

pretty roses

Sorry, you must use at least one numerical character.

1 pretty rose

Sorry, you cannot use blank spaces.

1prettyrose

Sorry, you must use at least 10 different characters.

1fuckingprettyrose

Sorry, you must use at least one upper case character.

1FUCKINGprettyrose

Sorry, you cannot use more than one upper case character consecutively.

1FuckingPrettyRose

Sorry, you must use no fewer than 20 total characters.

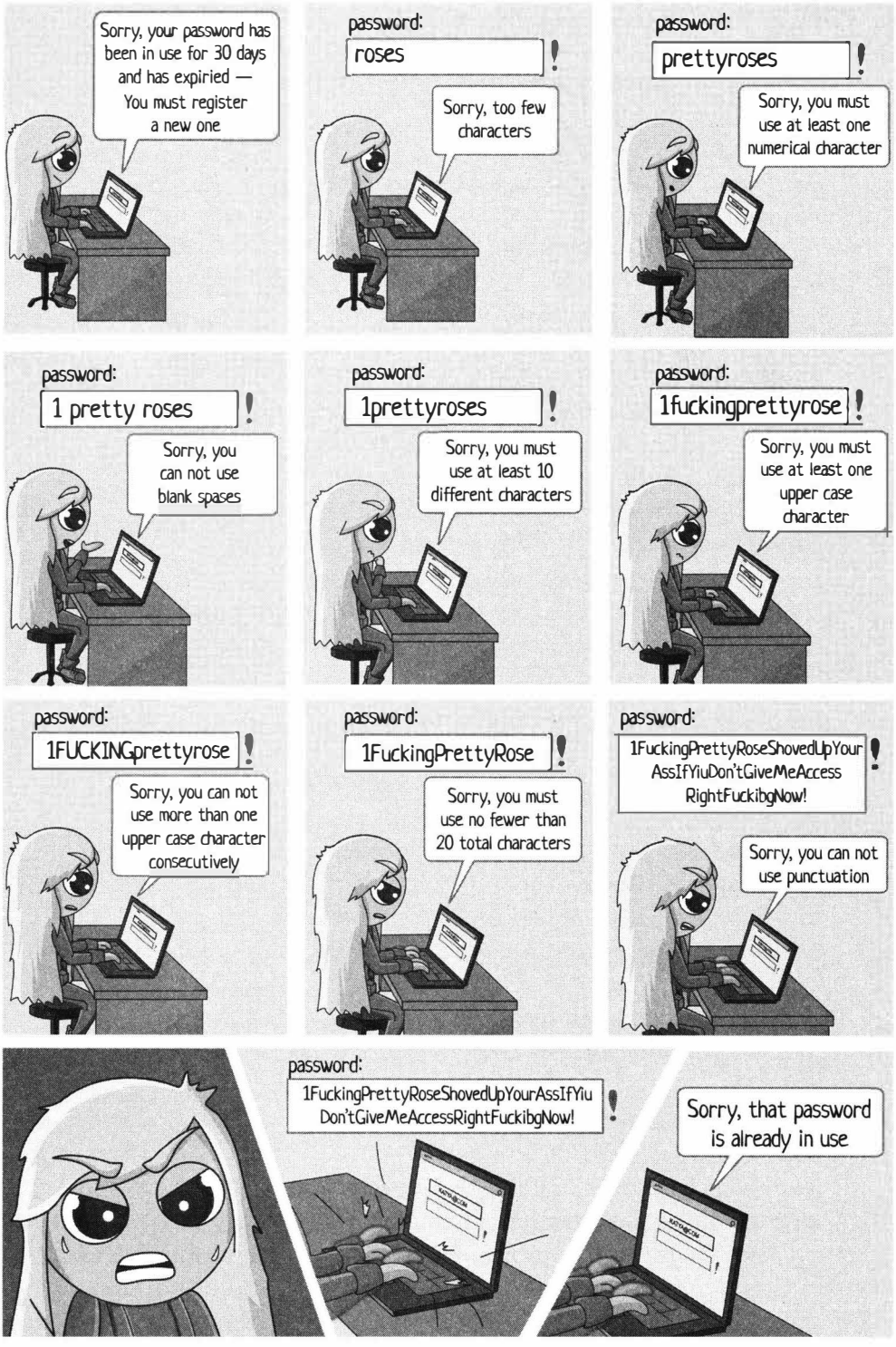
1FuckingPrettyRoseShovedUpYourAssIfYouDon'tGiveMeAccessRightFuckingNow!

Sorry, you cannot use punctuation.

1FuckingPrettyRoseShovedUpYourAssIfYouDontGiveMeAccessRightFuckingNow

Sorry, that password is already in use.

Пользователи всегда ошибаются. Неумышленно делают что-то не так. Они могут вставить вместо имени копипасту огромного текста из буфера обмена, вместо картинки загрузить файл TXT, промахнувшись при выборе файла... Они могут сломать что-то просто потому, что не знают, как система работает...


Sorry, your password has been in use for 30 days and has expiried — You must register a new one
password:
roses
Sorry, too few characters
password:
prettyroses
Sorry, you must use at least one numerical character
password:
1 pretty roses
Sorry, you can not use blank spases
password:
1prettyroses
Sorry, you must use at least 10 different characters
password:
1fuckingprettyrose
Sorry, you must use at least one upper case character
password:
1FUCKINGprettyrose
Sorry, you can not use more than one upper case character consecutively
password:
1FuckingPrettyRose
Sorry, you must use no fewer than 20 total characters
password:
1FuckingPrettyRoseShovedUpYourAssIfYiuDon'tGiveMeAccessRightFuckibgNow!
Sorry, you can not use punctuation
password:
1FuckingPrettyRoseShovedUpYourAssIfYiuDon'tGiveMeAccessRightFuckibgNow!
Sorry, that password is already in use

Именно поэтому так важно обрабатывать негативные сценарии. Вообще, в идеале нужно делать так, чтобы пользователь просто не мог ошибиться! Например, если в поле можно вводить только целые цифры (количество товара в корзине), что будет лучше?

1. При вводе символа ругаться: «Ты что, дурак? Тут только цифры!».
2. При вводе символа ругаться: «В поле можно вводить только цифры».
3. Не давать возможность ввести нецифровые символы — обрезать их при копипасте и игнорировать при вводе с клавиатуры.

Разумеется, идеальным будет третий вариант. А второй мы оставим на случай, если пользователь обойдет наше ограничение.

Ведь «не дать ввести символы» — это ограничение, установленное на клиенте. При знании нужных инструментов его легко обойти (начните с **Web developer toolbar**, вкладка **Forms**). Поэтому нужно оставить защиту на сервере — если пользователь символы-таки ввел, выведем ошибку.

Общая мысль:

- если негативного сценария можно избежать, сделайте это;
- если избежать его нельзя, тогда уже делайте сообщение об ошибке.

Ну а подробнее про то, как будет лучше пользователю, смотрите в литературе по usability[105].

Как искать сообщения?

- В систему можно загружать файл? Тогда пробуем грузить пустой файл, неправильного формата, расширения, разрешения...
- В форме редактирования есть обязательные поля? Пробуем их не заполнять или заполнять неправильно...
- Система передает ответы через SOAP/JSON? А если отправить неправильный запрос, пустой, с неполными или некорректными данными?
- И так далее, и тому подобное.

Если есть доступ к коду — отлично! Смотрим, в какие сообщения разработчики оборачивают эксепшены, находим их в приложении и проверяем отображение.

Каким должно быть сообщение об ошибке?

- ○ коротким, иначе пользователь заснет, читая его;
- ○ понятным, иначе как понять, что я сделала не так?

Из сообщения должно быть понятно:

- ✧ в чем моя ошибка?
- ✧ что мне сделать, чтобы исправить ее?

При этом сообщение должно быть в мире пользователя. Мне будут одинаково непонятны тексты:

- ○ Извините, что-то сломалось.
- ○ Error 38759245, see line 333 in code.
- ○ Неправильно введены данные.

Особенно на форме из 100 полей. Какие именно данные я ввела неверно? И даже если их подсветили красным, а что неверно-то? Как верно будет? Вы мне скажите, что делать-то надо.

А еще можно сделать сообщение интересным и смешным! Как? Вот, посмотрите, какая прелесть:

Примеры сообщений об ошибках

Dadata

В Дадату можно загружать файлы формата Excel или CSV. Система заранее предупреждает о том, какой должен быть файл.

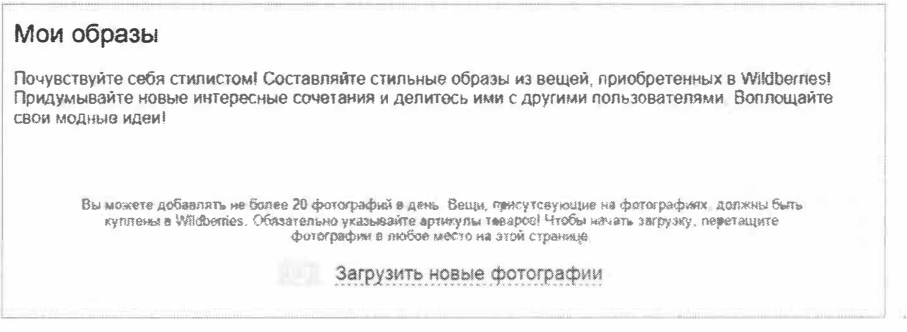
Если попробовать грузануть туда JPEG, система понятно объясняет, что надо сделать.

Это пример «как хорошо», а дальше разберемся, как плохо ;-)

Wildberries: галерея стиля, загрузка фото

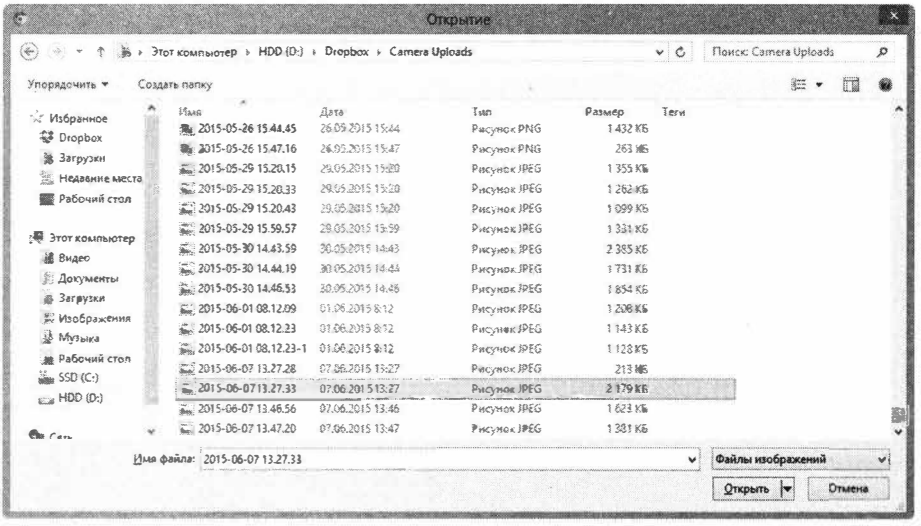
Однажды я попробовала загрузить фото в галерею стиля (туда пользователи выкладывают фоточки вещей из магазина) и невольно потестировала документацию сайта.

Итак, заходим в личный кабинет, оттуда в «Мои образы».Читаем информацию, но увы, ни слова о том, какие файлы допустимы.

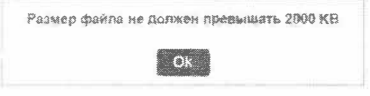


Вроде никаких ограничений нет...

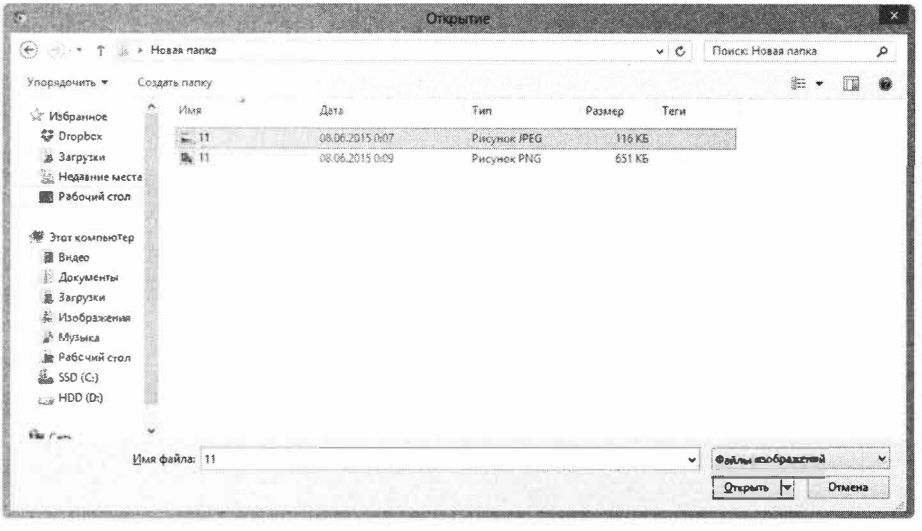
Итак, возрадовались возможности создать образ: выбираем фото для образа, пытаемся загрузить фоточку...



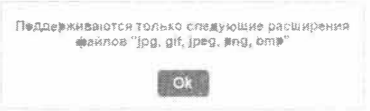
Ага, вот и первое ограничение! Файлы должны быть размером поменьше...



Но ведь так айпад снимает... Ладно, берем стандартные виндовые ножницы, обрезаем фото и сохраняем как JPEG, чтобы уж наверняка мало весить стал, грузим снова: не хочешь 2 Мбайт, на тебе 100 Кбайт...



И что бы вы думали, мне говорят?



Что значит «поддерживается только jpg»? А я что гружу? Я гружу JPEG, а мне говорят, что формат неверный, «грузи JPEG» o_O

Ладно-ладно! Снова открываю фоточку, снова вырезаю ножницами и сохраняю как PNG — на скриншоте мы его видим, он в 6 раз побольше весит, качество получше, все дела... Но ситуация повторяется...

Ох ох ох, у меня так было на первых попытках загрузки фоток... Что ж, пошла «старым дедовским методом»: вставить картинку в Paint, уменьшить масштаб и сохранить как jpg. И вуаля, блин! Загрузилось!

Я, как пользователь, посылаю разработчикам лучи ~~ненав~~ добра и нежности в такие минуты.

Знаете, в чем оказалась проблема? В регистрозависимости расширения файла:

- jpg — грузится;
- JPG — нет.

Виндовые ножницы сохраняли расширение заглавными буквами, и система не могла его обработать.

Как это исправлять? Можно, конечно, разжевать пользователю, что «jpg — могу, JPG — не могу», но... Зачем? Лучше сделать функцию проверки формата регистронезависимой. Помните о том, что лучше предотвратить ошибку, чем красиво ее расписать?

Далее, если загрузить слишком маленькую картинку, то получим ошибку:

Файл должен быть не менее 500 пикселей в ширину и 700 в высоту.

Откуда пользователю узнать про это ограничение? Нужно предупредить! Можно завести такое улучшение:

* *

Подсказывать правильный формат и размер файлов для загрузки в ГС

Указать ограничения еще до того, как пользователь начнет грузить файлы. В раздел «Мои образы» **https://lk.wildberries.ru/mylooks** добавить надпись:

Принимаются фото:

- jpg, jpeg, gif, png, bmp до 2 Мбайт
- не менее 500 пикселов в ширину и 700 в высоту

* *

Но будь у меня «доступ к телу», то есть к разработчикам, я бы попыталась их убедить, что проверка пикселов — это просто кошмар. Это вообще не в мире пользователей. Обычные пользователи, которые грузят фоточки в галерею, — это женщины. Они делают фото на свой айфон и пытаются их загрузить в галерею. Какие пикселы? Что надо делать, чтобы сделать их больше или меньше?

Имхо, лучше разрешить загрузку фото покрупнее, до 3–4 Мбайт, и ужимать их на программном уровне, чтобы не хранить «тяжесть» в базе. В галерее все равно небольшая фоточка отображается, поэтому вряд ли будет обидно, если сделать функционал, который сам уменьшает фото до 800 px.

Если такую пикселяцию поставили для ограничения снизу, чтобы не прошли нечеткие фото, то это тоже стоит обсудить. Если она предотвращает что-то, лучше ее вообще снять и посмотреть, сильно ли ухудшится качество загружаемых фотографий. Потому что 100 Кбайт JPEG вполне себе нормально отображается. Если же нагрузка на модераторов возрастет, подумать, что можно сделать и здесь. Но пока этот отсев создает проблемы пользователю, а не модератору.

И если уж его оставлять, то хотя бы внятно описать это ограничение, чтобы пользователи не мучились и не страдали при загрузке фоток.

И мы, как тестировщики, в процессе тестирования документации должны такие баги находить и, как минимум, извещать команду разработки.

Wildberries: форум, загрузка фото

Попробовала добавить аватарку на форуме.

Иду на форуме в «Мой кабинет», далее «Изменить фотографию», выбираю фото и...

БЛИН, ДА ВЫ ЧТО, ИЗДЕВАЕТЕСЬ, ЧТО ЛИ?!!!

И что я должна понять из этого сообщения? Почему неудачно? Я что-то сделала не так? Система сломалась? Совсем? Или мне попробовать через 5 минут?

Помните, что сообщение должно говорить, что сейчас плохо и как я могу это исправить (если могу).

Wildberries, галерея стиля, комментарии

Снова про галерею стиля. Все комментарии к образам отслеживаются модераторами, но это второй рубеж. Первый — автоматический отсев ссылок и черт знает чего еще.

Почему черт знает? Потому что текст сообщения непонятный: **Поле содержит ссылку или некорректное выражение**.

Вроде все ясно, да? Ну нефиг ссылки лепить, всего и делов... Но под огонь попадают нормальные слова или фразы. Например, слово «ребенок»...

А теперь представьте, что вы написали большой комментарий в духе: «Ой, какой красивый ребенок! А как хорошо на нем курточка сидит, прямо вау, самой такую захотелось взять...».

Пытаетесь отправить и получаете комментарий: «Поле содержит некорректное выражение». А как его найти с такими вводными? Только разве что бисекционным делением[106].

В общем, когда пишете сообщение об ошибке, думайте о пользователе. Что я из него пойму? Смогу ли исправить ошибку? Пойму вообще, что я сделала не так? Если да — то всё круто! Если нет — сделайте так, чтобы поняла.

Поп-ап сообщения

Pop-up — это всплывающее окошко, как навязчивое: «А вы уже лайкнули нас в Фейсбуке?».

- В веб-приложениях это обычно:
  - ✧ уведомление о регистрации;
  - ✧ акционное предложение;

- ✧ мини-версия корзины с покупками;
- ✧ ...

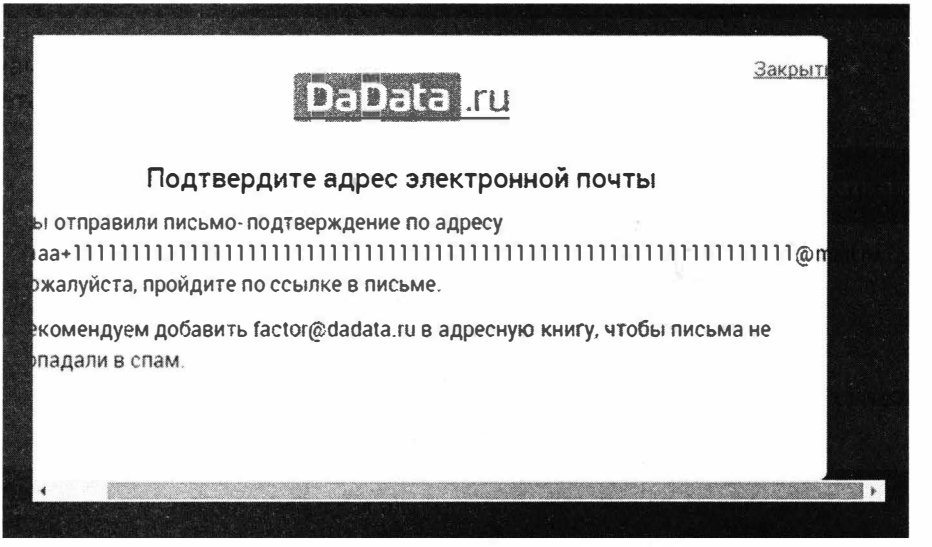
○ В десктоп-приложениях поп-ап обычно не используются, разве что в старых системах, где всё вылезает в мелких окошках: и ошибки, и позитивный текст.

○ В мобильных игрушках:

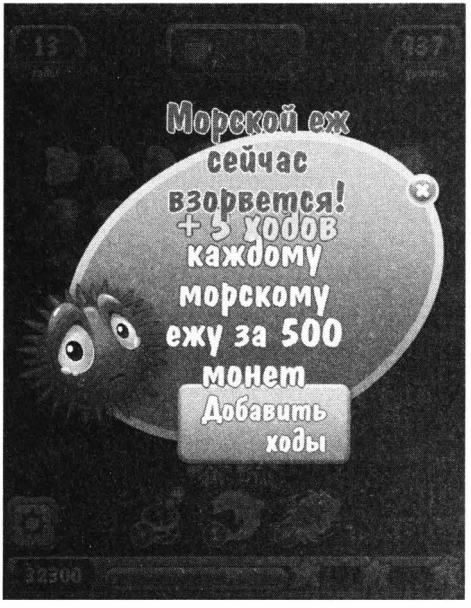
- ✧ реклама новой игрушки;
- ✧ временная акция;
- ✧ поздравляшка с успешным окончанием уровня;
- ✧ непоздравляшка «ты провалил уровень»;
- ✧ ...

Что тут стоит проверить? Влезает ли текст в отведенное ему место ;-)

Пример бага в веб-приложении мы находили в рамках The FedEx Tour[107]: если ввести слишком длинный емейл при регистрации, получим такую картину:

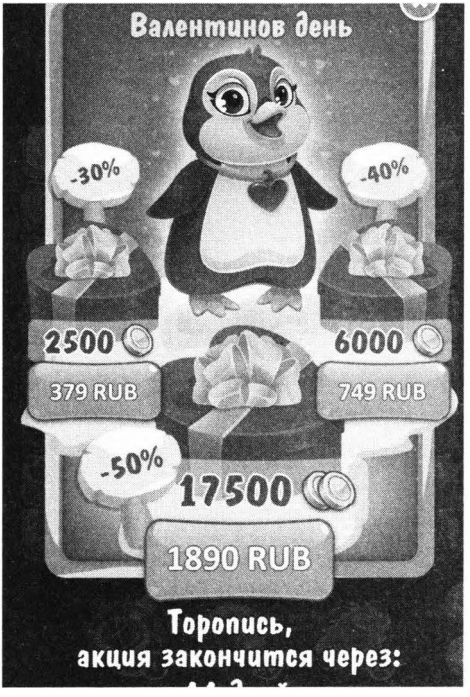


А вот и пример поп-апа в игрушке из серии «три в ряд». Если не успел освободить ежа за *N* ходов, выпадает такое сообщение:



Это на iPad mini было дело[108] — пример сообщения об ошибке во всплывающем окне.

Или вот еще пример[109] из той же игрушки, но уже не сообщения об ошибке, а рекламный поп-ап. Прочтете текст снизу: когда закончится акция? ;-)



Так что учтите, что тестировать надо на самых мелких экранах, а потом и на средних. А то, может, для смартфонов изображение уменьшили, а для айпадов оставили одно — что для макси, что для мини: «Большой же экран, влезет!».

Предупреждения «что вводить»

Это когда около поля заранее пишут требования к нему:

- Логин должен содержать латинские символы и цифры.
- Пароль должен быть длиннее 8 символов и содержать как символы, так и цифры.
- ...

Чтобы не допустить сообщение об ошибке, предупреждаем о правилах заранее! А требования те же самые, что и у сообщений об ошибках, — предупреждения должны быть:

- понятные;
- в мире пользователя;
- выполнимые.

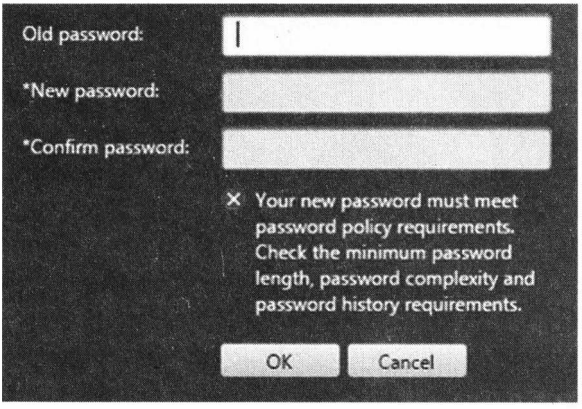
Не надо писать их на языке разработки. Пишите для людей, они оценят это больше, нежели формальный стиль общения.

Про выполнимость расскажу такую историю.

Банк: изменение пароля

На одном проекте я работала с банком. Чтобы посмотреть логи сервера, нужен удаленный доступ к виртуальной машине.

И вот меня зарегистрировали в закрытом портале банка их админы, прислали мне ссылку и первичный пароль. После первой авторизации пароль нужно изменить. Страница изменения выглядит вот так:



Вроде все верно: тут говорят сразу, что пароль должен соответствовать политике безопасности. Я не против, но это вся страница. Никакой ссылки на политику безопасности... И где же мне ее взять?

Тут нужно учесть, что на страницу заходят люди, которые НЕ работают в банке и НЕ знают, где взять внутреннюю документацию (да ее никто и не даст). Поэтому лучше просто заранее вынести информацию в это сообщение.

Я же подбирала пароль наугад, а что делать...

Думайте о своих пользователях — кто они? Если вы делаете софт только для сотрудников компании, можно ссылаться на некую внутреннюю документацию. И то всегда лучше дать ссылку, чем заставлять пользователя искать ту же политику безопасности.

А если к вашему приложению будет доступ у третьих лиц, то ссылки на внутреннюю документацию стоит убрать. Напишите лучше около поля, каким должен быть пароль. Это не займет места больше, чем эта унылая цитата ;-)

Инструкция по установке

Пользовательское приложение

Если это простое или общедоступное ПО — его будут устанавливать сами пользователи.

В идеале, конечно, инструкции нет, вместо нее просто EXE-файл (если речь про Windows). Помните, как последний раз что-то устанавливали? Скачал, запустил, 10 раз щелкнул «далее-далее-далее», профит!

Но что если вы решили установить какое-либо приложение на Linux? Ну ладно, ладно, там тоже бывает просто — запустил команду скачивания, и вот у тебя уже всё есть.

Но что если недостаточно просто запустить установочный файлик? Что если нужно прописать какую-то переменную в PATH, например? Тогда комбинируем: файл быстрой установки + краткая инструкция.

Например, установка maven на Windows. Скачиваем, читаем инструкцию по установке. Она мегапростая, но она есть[110]:

✧ Check environment variable value e.g.

```
echo %JAVA_HOME%
C:\Program Files\Java\jdk1.7.0_51
```

✧ Adding to PATH: Add the unpacked distribution's bin directory to your user PATH environment variable by opening up the system properties (WinKey + Pause), selecting the „Advanced" tab, and the „Environment Variables" button, then adding or selecting the PATH variable in the user variables with the value C:\Program Files\apache-maven-3.6.0\bin. The same dialog can be used to set JAVA_HOME to the location of your JDK, e.g. C:\Program Files\Java\jdk1.7.0_51

✧ Open a new command prompt (Winkey + R then type cmd) and run mvn -v to verify the installation.

Переведем:

✧ Проверьте текущую переменную окружения: есть ли у вас Java на компьютере?

```
echo %JAVA_HOME%                    → команда проверки;
C:\ProgramFiles\Java\jdk1.7.0_51    → результат команды, он говорит,
```

что Java установлена, так как переменная %JAVA_HOME% существует;

- ✧ Добавьте maven в PATH: добавьте ссылку на каталог bin в пользовательскую переменную PATH. Для этого откройте системные настройки (WinKey + Pause), выберите вкладку «Advanced», нажмите кнопку «Environment Variables» и добавьте переменную PATH или выберите ее, если она уже существует. Добавить туда надо значение пути до каталога bin, например: C:\Program Files\apache-maven-3.6.0\bin. Аналогично можно и JAVA_HOME добавить, прописав путь до JDK — например: C:\Program Files\Java\jdk1.7.0_51.
- ✧ Откройте НОВУЮ командную строку (Winkey + R, а потом написать cmd) и выполните команду mvn -v, чтобы проверить установку.

Фактически для установки нужно выполнить ОДНО действие — добавить ссылку на каталог bin в PATH. И еще два пункта для проверки того, что всё установилось. Maven обращается к JAVA_HOME, поэтому такая переменная должна существовать, что мы и проверяем. И саму установку maven тоже проверяем командой `mvn -v`.

Это оптимальная инструкция, ничего лишнего. Или приложение устанавливается само, или к нему есть короткая инструкция на действия, которые надо сделать именно вам.

Серверное приложение

Это полноценное приложение, которое заказчик сам устанавливает на свои серверы. Например, систему работы с клиентами в банке. Она требует конкретных ресурсов и настроек системы, просто «далее-далее-далее» тут не обойтись.

В этом случае нужно обеспечить полноценную инструкцию по установке, считая, что:

1. Сервер создан с нуля, там НИЧЕГО нет.
2. А вот Касперский наверняка есть, и он может обрубать вашу работу. Да, и на линукс некоторые умудряются его вкорячить.
3. Разворачивать серверы будут админы, которые с компьютером на «вы».

Учитывая третий пункт, пишем максимально подробно, максимально понятно. Если просим выполнить команду — даем саму команду, чтобы вставил в командную строку, и всё. Если сервер можно поднимать как на винде, так и на линуксе, дублируем инструкции: одна для Windows, другая для Linux. И везде даем примеры команд.

Разумеется, если вам попадется прошаренный админ — это будет счастье великое. Но крутой по простой инструкции пройдет, а джуниор по инструкции для крутых — нет. Выводы?

В идеале, конечно, ставим на голом сервере докер и не паримся с операционными системами и настройками. Выдали готовый докер-файл, готово!

Но иногда докер ставить нельзя. Служба безопасности не одобряет. Мало ли, что вы в этом докере на сервер установите? Нет-нет, давайте инструкцию по развертыванию с нуля, мы сами всё настроим.

Тогда инструкция будет сильно больше! Что может в нее входить?

- Подготовка к развертыванию:
  - ✧ методика расчета требований к аппаратному обеспечению — будьте уверены, вас обязательно спросят, откуда такие требования к «железу». Проще один раз записать, чем каждый раз снова объяснять;
  - ✧ требования к аппаратной платформе: сколько места свободно, какая ширина канала и другие детали. Если вам нужна будет рабочая станция для настройки или другой доступ, включаем сюда, чтобы заранее запросить все, что нужно;
  - ✧ требования к настройке программно-аппаратной платформы: какие программы установить, какие права доступа выдать. Соберите всю информацию в одном месте:
    - настройка рабочей станция для ваших сотрудников;
    - настройка сервера приложений для ОС *NIX;
    - настройка сервера приложений для ОС Windows;
    - настройка сервера СУБД Oracle;
    - настройка сервера СУБД MariaDB;
    - настройка Active Directory;
    - ...

Видите? Windows — отдельно, Linux — отдельно. Если умеете работать с разными базами данных — напишите под каждую отдельную статью.

- Инсталляционный пакет — описание файлов, которые подрядчик передает заказчику, с их описанием.
- Установка системного и специального ПО.

Например, продукт написан на Java, а сервер приложения — WildFly. Тогда здесь будут примерно такие разделы:

- ✧ установка параметров ОС Windows;
- ✧ создание пользователей ОС Linux;
- ✧ установка параметров ОС Linux;
- ✧ установка Java;
- ✧ установка WildFly.

Каждая статья подробно расписывает, как выполнить установку. И не забывайте о примерах!

- Установка системы — инструкция по установке системы.
- Запуск системы — короткая инструкция, в которой прописаны команды для запуска и остановки системы.
- Что-то ещё.

Тут может быть ещё целая куча документации:

- ✧ о том, как настраивать систему ПОСЛЕ установки;
- ✧ о дополнительных настройках типа шифрования паролей, горячего резерва и прочего.

Главное, о чем следует помнить: если по вашей инструкции будет проходить другой человек, она должна быть максимально простой и понятной. Желательно с примерами — прям копируй и вставляй.

Описание полей

Правило хорошего тона — когда ты просишь пользователя заполнить поле, объясни, зачем ему нужно это делать. Это будет лучше, чем просто ставить звездочки обязательности.

Вот, например, регистрация в Дадате:

- ○ Зачем указывать имя? Чтобы к вам обращались в письмах по нему.
- ○ Зачем указывать почту? Чтобы мы могли прислать состояние обработки и баланс.

Все эти подписи — это тоже документация! Не надо думать, что документация — это только документы про работу системы, вынесенные отдельно. Нет! Это и сообщения об ошибках, и текст на главной странице, на кнопках, *в подписях к полям... Это всё — тоже документация.*

Маркетинговые материалы

- Рассылки, которые вы шлете пользователям.
- Статьи на Хабре или другом ресурсе.

Это — тоже документация, которая имеет прямое отношение к вашей системе. И будет неплохо ее хотя бы изучить. А в идеале — участвовать в процессе создания. Потому что лучший материал пишут специалисты по системе, а не по маркетингу или копирайтингу. Работайте с вашими авторами, тестируйте их статьи!

Еще один вариант рекламы — какой-нибудь купон, который вылезает на сайте или в мобильном приложении:

- вычитываем текст, чтобы не было ошибок;
- если это поп-ап окно, проверяем, что его можно закрыть;
- если это всплывашка в мобилке, проверяем, не вылезает ли она за экран на небольшом смартфоне.

Обучение: FAQ, презентации

Обучающие материалы — это тоже документация. И снова — они бывают самых разных видов, что зависит от типа продукта и бюджета.

- **Tutorial.**

В играх или сложных программах при первом запуске появляются подсказки: куда нажимать, что вообще происходит. Особенно актуально для игр, разумеется.

Это обучение — тоже документация, его важно проверить. Не только на орфографию и то, что текст не вылезает за пределы экрана, но и на сами подсказки. Помогут ли они тому, кто видит программу впервые? Помогут ли человеку, у которого это первая игрушка на планшете, все ли он поймет?

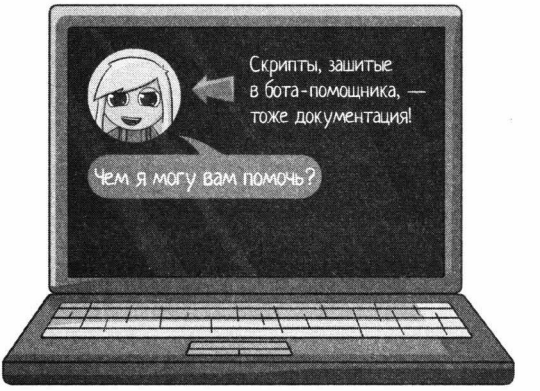
Важно проверить этот туториал при первом запуске, когда вы еще и сами не знакомы с игрой. Потом уже привыкнете и будет казаться, что «тут все очевидно».

- **FAQ** — одна или несколько статей, где собраны ответы на типовые вопросы.
- **Клавиша <F1>** — документация из серии «Помощь».
- **Online help** — если у вас есть робот-помощник с зашитыми в него скриптами — это тоже документация!

- **Обучающие статьи.**

Они могут быть в закрытом или открытом доступе. Это то, где вы рассказываете, как сделать ту или иную вещь с помощью вашей программы:

- ✧ если статья открытая, то туда может вести ссылка прямо с сайта;
- ✧ если статья закрытая, вы даете к ней доступ только своим клиентам. Она может быть в вашей вики-системе, ворд-файле или гуглодоке.



- **Видео.**

Если у вас есть видео в помощь — их тоже надо тестировать на устаревание!

Это может быть полноценное видео или «кликабельные слайды» — их можно сделать в Wink[111].

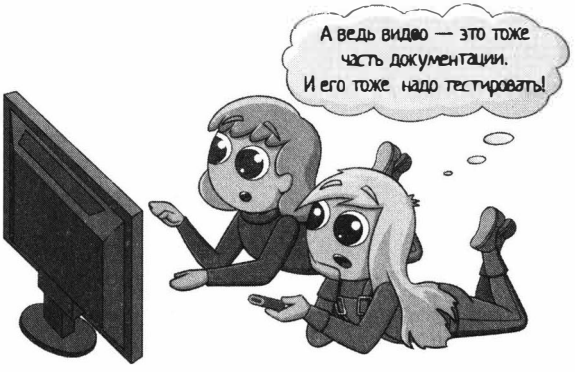
- **Презентация.**

Это может быть:

- ✧ продающая презентация «что умеет наш продукт»;
- ✧ обучающая презентация «как им пользоваться».

И это тоже документация, которую надо тестировать:

- ✧ выдержан ли стиль компании в презентации?
- ✧ актуальны ли картинки? Не менялся ли интерфейс?
- ✧ насколько полно раскрыта тема? Может, стоит что-то добавить?



Дорабатывайте свои презентации!

У меня на одном из проектов было серверное приложение, коробочный продукт. И предусматривалось обучение администраторов и простых пользователей — наш сотрудник приезжает к заказчику и проводит обучение очно. Показывает презентацию и демонстрирует товар лицом.

Я и сама проводила обучение сотрудников заказчика. Тестировщику вообще очень полезны такие поездки: поговорить с теми людьми, которые будут использовать ваш продукт, послушать их вопросы... И понять, что для них важно!

Так вот, после каждой поездки я переделывала свою презентацию. Где-то вопросы задавали, где-то сама во время обучения подумала: «О, стоит еще рассказать вот о чем». Это процесс постоянных улучшений — не стоит думать, что презентацию сделаете один раз, и всё готово. Тестируйте ее, дорабатывайте и улучшайте!

Поздравляшки

И снова это касается в первую очередь игрушек. Такая ностальгия по тем временам, когда я именно игры для мобильников и тестировала...

Так вот, когда вы заканчиваете уровень или всю игру, вылезает «поздравляшка». Обычно этот всплывающее окно с текстом «Поздравляем!». Еще чаще текст будет английским: «Congratulations!»

Английская версия длиннее, отсюда могут возникнуть проблемы разряда «сообщение вылезло за экран смартфона». Конечно, сейчас уже нет тех экранчиков, на которых тестировала я (старый Siemens, например, с диагональю 3 дюйма). Но и среди смартфонов встречаются малыши, которые стоит проверить!

И даже если разработчик дает вам чит-коды, чтобы пропускать уровень, обязательно проверяйте, как такие поздравляшки отображаются. Особенно здорово, если можно попасть сразу на конец игры и посмотреть, что там будет.

Кнопки

Текст на кнопках — тоже документация! Вроде очевидно, но все же... Я, например, сталкивалась и с таким:

Как можно пропустить такой баг? Ведь регистрация в интернет-магазине — основной кейс, потому что без нее пользователь не может продолжить покупки. Значит, ее тестируют. Видимо, у всех так замылился глаз уже, что подписи на кнопках просто не читают... А вы — читайте! ;-)

Раз уж столкнулась с такой проблемой, то вот, решила вынести ее в отдельный пункт. Не забывайте всё читать — не по диагонали, а целиком.

Остальное

Документации очень много. Это все, что видит пользователь и что может показаться неочевидным:

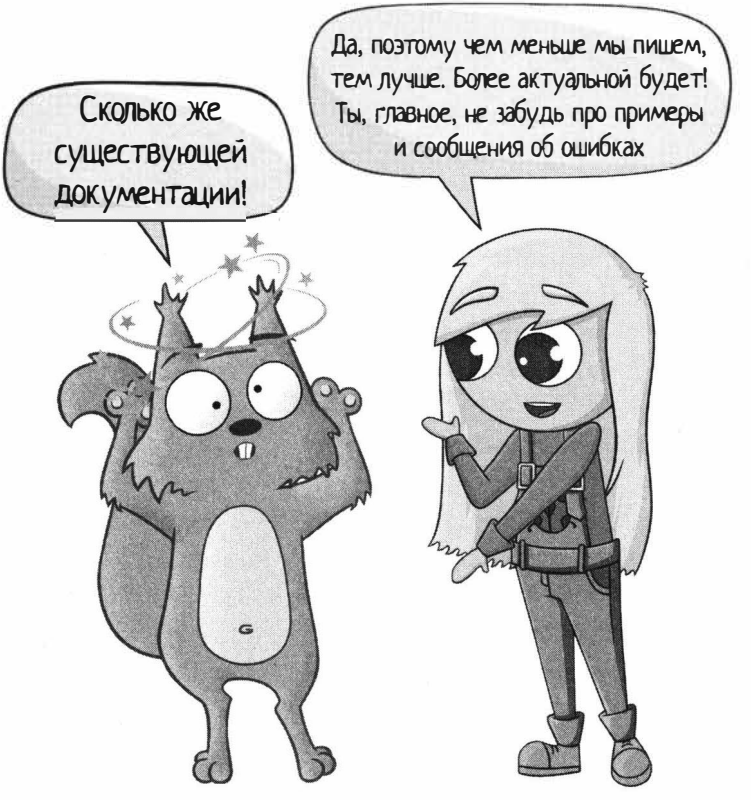
- **гарантийный талон** — это тоже документация (которую редко читают, но всё же). Он обычно встречается у бытовой техники или компьютера;
- **пользовательское соглашение/оферта** — а это то, что заменяет нам договор или гарантийный талон в современном ИТ-мире. Пользуясь онлайн-сервисом, вы автоматически соглашаетесь с офертой, которая есть у него на сайте, даже если вы ее не читали ;-)
- **упаковочный текст и графика** — помните, раньше игры покупали на DVD в специальных магазинах? Так вот обертка диска, текст и графика на нем — это тоже документация, которую надо проверять!

А то вдруг у вас слишком завышенные требования к «железу», и никто не купит диск? Или вдруг требования занижены, и игра просто не запустится на слабом компьютере? Тогда рассерженный пользователь вернется в магазин и будет требовать назад свои деньги.

Подведем итоги

Видов документации очень много! Она не ограничивается только лишь требованиями. Не забывайте про остальную документацию на проекте. И обращайтесь к этому чек-листу за поиском вдохновения: «Что я ещё забыл проверить» ;-)

- Самые важные виды документации:
  - ТЗ — требования;
  - пользовательская документация;
  - примеры;
  - шаблоны;
  - сообщения об ошибках.
- Чуть менее, но все еще важно:
  - предупреждения о том, как заполнять поле;
  - описание поля («Зачем мне его заполнять?»);
  - предзаполненный текст в поле ввода;
  - инструкция по установке;
  - маркетинговые материалы;
  - статьи на Хабре или где-то еще;
  - обучение:
    - материалы;
    - FAQ;
    - Online help;
  - презентации.



- А еще есть:
  - ✧ пользовательское соглашение/оферта;
  - ✧ гарантия;
  - ✧ упаковочный текст и графика.

Это тоже нужно проверять, но уже не в приоритете!

Как тестировать документацию?

Чек-лист проверки

Тестируем документацию на следующие характеристики:

- полнота;
- однозначность;
- непротиворечивость;
- необходимость;
- осуществимость;
- тестируемость.

Пунктов может быть больше — вот, например статья «Тестирование документации к программным продуктам»[112]. Но эти — основные ;-) А теперь подробнее о каждом.

Полнота

Все ли описано? Ничего не забыли? Вдруг у нас остался неописанный функционал или параметр API-метода?

Чтобы проверить этот пункт, просто напишите чек-лист проверок функционала. Вот как начали читать ТЗ, сразу записывайте тесты. Важно именно ***писать***, а не просто прикидывать в уме. Иначе что-то обязательно забудете.

Из собственного опыта...

Как-то раз мне прислали на проверку ТЗ на новый функционал. Да, по-хорошему следовало бы сразу прикинуть тесты, которые буду писать. А это надо взять ручку, блокнот и начать тест-дизайнить... Но я занималась другой задачей, а ТЗ требовалось проверить «сегодня», чтобы отправить оценку заказчику.

Поэтому решила проверить «мысленно». Читаю пункт ТЗ, прикидываю, какие могут быть тесты. В итоге задала парочку уточняющих вопросов:

— А что если... ?

У заказчика уточнили, документацию пополнили! В остальном меня всё устроило: вроде нормально описано, всё учтено.

Заказчик согласовал ТЗ, разработчик сделал. Пришло время тестировать. Начинаю писать автотесты и... Да, разумеется, сразу получаю еще 5–10 дополнительных вопросов. В ТЗ не были предусмотрены все сценарии, и я о них тоже не подумала, пока прикидывала тесты мысленно.

А как только стала расписывать план автотестов, сразу увидела «белые пятна». И это у меня 10 лет опыта в тестировании.

Так что не полагайтесь на память и «быстренько мысленно прикину» — выделите полчаса и распишите чек-лист проверок подробно!

Чем плохо отложить вопросы на потом? Не слишком профессионально согласовать требования, которые написала твоя команда, а через пару недель задавать уточняющие вопросы. Заказчик подумает: а чего сразу не спросили?

Плюс иногда можно пропустить очень важный вопрос, который трудозатратно делать. А ТЗ, напомню, уже согласовано. Так что всё, что продолбали на этапе согласования, делаем за свой счет.

Да, написать тесты — это дольше, чем просто прочитать документацию. Но зато вы экономите время потом, ведь чек-лист уже готов, бери да проверяй!

Однозначность

Требования должны трактоваться всеми одинаково.

«Отчет должен загружаться быстро»... Что значит «быстро»?

- пользователь будет уверен, что страница загрузится за доли секунды, даже если это сложный отчет на многомиллионных данных;
- разработчик прикинет, что в таких объемах 5 секунд — нормальное время отклика, даже быстрое.

Налицо конфликт интересов. И ведь каждый станет тыкать в ТЗ для отстаивания своей позиции. Лучше конкретизировать:

- отчет за год должен загружаться не более секунды;
- отчет за весь период времени должен загружаться не более 5 секунд.

Если в требованиях не указано, что у нового поля с суммой дохода должно быть значение по умолчанию:

- аналитик решит, что там будет значение 0. Деньги же, цифра!
- разработчик сделает пустое поле — не указано ведь ничего! А это что-то сломает...

Если в требованиях не указано, как обработать ошибочный сценарий, разработчик может:

- не подумать о нем и никак не обработать, — и пользователь огребет страшную ошибку;
- подумать о нем и обработать так, как считает правильным, — а тестировщик потом будет доказывать, что надо было делать по-другому. Пойдут к аналитику, потратят и его время тоже, а потом еще код придется переделывать...

Требования должны быть однозначны, иначе каждый увидит там что-то своё

Всё, что можно прочитать двояко, лучше исправить. Это не значит, что нужно описывать каждую мелочь, но всё зависит от читателей документа.

Если это внутренний документ, а у вас сильная команда, — можно не расписывать подробно.

Если этот документ отправят заказчику, надо расписать вообще всё — потому что у заказчика свои тестировщики, и они обязательно зададут кучу «а что если...?» Если они хорошие тестировщики, разумеется. Это вы знаете свою программу и представляете, как она реагирует на ошибки или что-то такое. Тестировщик заказчика этого не знает, он будет уточнять.

Непротиворечивость

Требования не должны противоречить сами себе. Такое обычно бывает, когда требований много. Аналитик просто забывает, что уже писал про параметр и снова придумывает его поведение. Иногда придумывает немного по-другому.

Например, есть страница нефункциональных требований, где написано, что *любая* страница должна грузиться не более трех секунд. Аналитик делает ТЗ на новый модуль отчетности, который использует много данных и сложные формулы. И он пишет, что отчет может грузиться вплоть до минуты. Явное противоречие!

Если заказчик найдет первый раздел документации, он будет требовать именно такую скорость. И будет прав, кто же хочет ждать? ;-)

Необходимость

Помните главный принцип: «Кратко, но емко!»? Он действует и в документации тоже.

Круто, когда документация полная. Но это не значит, что простой функционал надо растянуть на 10 листов А4. Когда документации много, сложно проверить полноту, сложно удержать в голове, о чем уже говорил, не повторяться и не противоречить самому себе.

Подумайте, так ли уж нужно описывать каждую кнопочку интерфейса? Это, правда, актуально? Пользователи, правда, не догадаются, что фильтры по строковым колонкам работают одинаково?

Пишите только то, что необходимо:

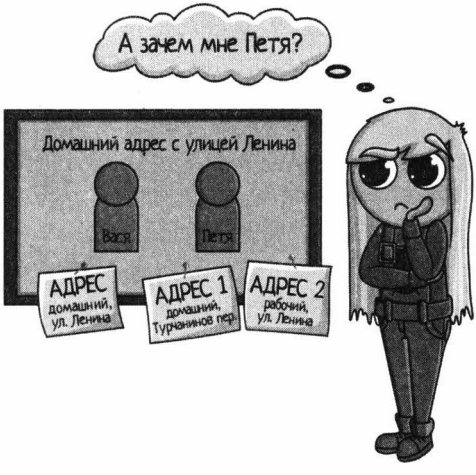
- в ТЗ — функционал, основной сценарий и альтернативы, типы ошибок;
- в пользовательской документации — то, как пользоваться системой. Но не доходя до крайностей и обучения включению компьютера.

Осуществимость

А можно ли реализовать то, что вы написали? Насколько это будет сложно и дорого?

Этот пункт обычно проверяют разработчики. Они остужают буйные фантазии из серии «загружать миллионы данных за 0,1 секунды» или что-то архитектурно сложное. Бывает такое, что на бумаге всё звучит просто, а вот сделать это займет человеко-месяц в лучшем случае.

Из собственного опыта



В одной из систем, с которыми я работала, был точечный поиск. Не просто «найди мне все данные, где встречается „Ленина“», а именно «найди мне адреса, у которых улица Ленина». Это отсеет фамилию «Ленина», комментарий к телефону и другие нерелевантные данные.

Но вот беда — нельзя поискать по двум параметрам ОДНОГО атрибута. Если сказать: «Найди мне домашний адрес с улицей Ленина», система вернет:

- Васю: домашний адрес с улицей Ленина;
- Петю: домашний адрес с Турчаниновым переулком и рабочий с улицей Ленина.

Почему так?

Это баг в системе? Или просто нельзя сделать такой поиск, это неосуществимо? Нужно смотреть по коду.

В той системе для поиска использовали Lucene[113]. Ее можно настроить по-разному:

- поиск по любому полю;
- поиск по конкретному полю;
- множественный поиск по конкретному атрибуту (в одном адресе проверить и тип, и улицу);
- ...

Но! Чем сложнее сценарий поиска, тем медленнее он работает. Дольше сохраняет данные (потому что структура внутри усложняется), дольше ищет — или потребляет больше ресурсов для той же скорости.

Когда данных с гулькин нос, это неважно. А когда их миллионы, нужно искать баланс. И такой выбор возможностей поиска — это именно компромисс для скорости и потребляемых ресурсов под сценарии использования.

Осуществимо желание отсеять Петю? Да. Но это просто не нужно. Потому что вреда принесет больше, чем пользы. Ради того, чтобы в выборку из 1000 человек не попали 10 лишних, платить несколько миллионов за дополнительные мощности серверов смысла просто нет.

Тестируемость

Можно ли протестировать этот функционал?

Подумайте об этом заранее. А то бывает так, что разработчик уже всё сделал, и тут только тестировщик понимает, что задачу никак нельзя проверить.

Или можно проверить вручную, но нельзя написать автотесты — фреймворк под новый функционал не заточен.

Если в компании принято все покрывать автотестами, то это станет проблемой. Может, разработчик прочитает ТЗ и сам поймет, что ещё фреймворк тестов дорабатывать надо. А может, он не вспомнит об этом. И тогда ваша задача — вспомнить. Чтобы сразу заложить время на доработки.

Из собственного опыта...

У меня бывали ситуации, когда мы делали задачу в текущем релизе, а потом ставили новую: «Доработать фреймворк автоматизации, чтобы поддержать изменения» — в следующий. Иногда забывали про фреймворк, а потом времени в релизе уже не оставалось. А иногда сразу понимали, что всё сразу сделать не успеем.

Иногда про тесты не думаешь, так как уже есть похожие. Например, у нас давно были автотесты на обратный поток в JMS-очередь. А потом для одного из заказчиков мы сделали обратный поток в две JMS-очереди.

Сначала я не подумала о доработке автотестов — ведь возможность проверить «что ушло» есть! А когда добралась до тестирования, пошла к разработчику:

— А как мне указать вторую очередь? Или папка jms— это то, что в обе уйдет?

— Хм, нет, погоди, это только в старую. Надо доделывать фреймворк, поддержать разные очереди.

Ну и ничего, доделали, написала тестов!

Хотя лучше об этом помнить сразу, иначе велик шанс, что тестировать за вас придется разработчику. Или половину проверок переносить на следующую итерацию, что тоже не очень хорошо.

При этом бывает и так, что тестировать все равно придется разработчику. Скажем, когда делают рефакторинг, что может проверить тестировщик? Только регресс провести, посмотреть, что ничего не отломалось. А если есть автотесты, то это проверят они ;-)

Однако если тесты (авто или ручные) прошли успешно, это еще не значит, что рефакторинг прошел хорошо. Сама суть рефакторинга — переписать код, чтобы он был оптимален и более читабелен. Чтобы его было легче поддерживать в дальнейшем и интегрировать с другими частями системы.

И именно это и надо проверить! А проверить это может только разработчик. Он и выполняет тестирование в таком случае.

Мнемоника CIRCUSMATTA

В *главе 6* мы обсуждали разные мнемоники. Есть мнемоника и для ревью пользовательских историй. Это как раз про тестирование требований!

- **C**ompleteness — полнота.
- **I**ndependent — независимость.
- **R**ealisable — реализуемость.
- **C**onsistency — консистентность.
- **U**nambiguity — однозначность;
- **S**pecific — специфика заказчика.
- **M**easurable — измеримая.
- **A**cceptable — приемлемая.
- **T**estable — тестируемая.
- **T**raceable — трассируемая (можно проставить взаимосвязи).
- **A**chievable — достижимая.

Примеры багов из жизни

Это будет раздел со ссылками. Зачем тратить бумагу на то, что проще прочитать в блоге? ;-)

1. `Sysdate` вместо `Fly_date` в письме-напоминалке: **http://okiseleva.blogspot.com/2017/02/sysdate-flydate.html.**
2. Снова письмо — «Оплатите наш счет до вчера»: **http://okiseleva.blogspot.com/2017/02/blog-post_10.html**.
3. Опять письмо — «Для гонки гладиаторов ты всегда мальчик»: **http://okiseleva.blogspot.com/2017/06/blog-post_19.html**.
4. Gmail, сообщение об ошибке «Неправильный ярлык», угадай почему: **http://okiseleva.blogspot.com/2017/07/blog-post.html**.
5. Опечатка на кнопке «Продолжить покупки»: **https://okiseleva.blogspot.com/2019/01/blog-post_29.html**.
6. Примеры плохих сообщений об ошибке (PayPal): **https://okiseleva.blogspot.com/2018/10/paypal.html**.

7. Картинка морского ежа не влезает в отведенное ей место: **http://okiseleva.blogspot.com/2016/03/blog-post_30.html**.
8. Картинка акции Валентинова дня не влезает в экран iPad mini: **http://okiseleva.blogspot.com/2017/02/mini-ipad.html**.

Другие примеры смотрите в моем блог-посте с копилкой всех найденных багов[114]: раздел «Тестирование документации — шедевры плохих текстов об ошибках».

Где искать документацию?

Везде ;-)

Я просто хочу еще раз обратить ваше внимание на то, что документация — это не только техническое ТЗ и описание для пользователя.

Ищите типы документации:

- сообщения об ошибках — вызывайте недопустимые сценарии, проверяйте их обработку;
- примеры — это не только примеры в ТЗ, это предзаполеннные поля, это файл-образец для обработки!
- письма, которые шлет система! Особенно, если в них есть плейсхолдеры;
- текст на главной странице;
- текст под полем для ввода;
- надписи на кнопках;
- pop-up сообщения;
- FAQ;
- статьи по теме от компании;
- Tutorial в игре;
- ...

А еще небольшой лайфхак: если есть доступ к коду, можно поискать сообщения там! Особенно если у вас многоязычная игра. Тогда при правильном проектировании у разработчика будут файлы «русские тексты», «английские», «немецкие»... То есть код и текст разделены. Когда вы что-то делаете в игре и надо показать текст, система смотрит, какой язык выбран, и лезет в нужный файл.

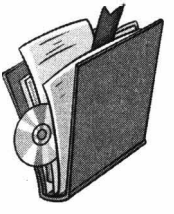
В таком случае можно открыть сами файлики и вычитать их. Поискать орфографические ошибки, оценить перевод на адекватность. А заодно посмотреть там, где самый длинный текст, его и найти в игрушке, проверить на «влезает ли в экран»!

Вопросы для самопроверки

1. Какая бывает документация?
2. Сообщение об ошибке — документация или нет?
3. Как тестировать документацию?

В этой главе ответов не будет, так как их можно найти по заголовкам разделов 😉

Портфолио



Продолжаем делать портфолио.

1. Найдите всю документацию на своем сайте/в приложении.
2. Выпишите ее по разделам: тут основное ТЗ, а вот примеры, вот сообщения об ошибках, вот что-то еще...

Глава 8
СОЗДАНИЕ ДОКУМЕНТАЦИИ: ТЕСТОВОЙ И НЕ ТОЛЬКО

Как тестировать документацию, вроде понятно. А если ее нет? Можно ли создать самим? И что же такое «тестовая документация»?

В этой главе:

- вспоминаем о тестовой документации;
- пишем вариант использования;
- составляем Decision Table;
- рисуем State-Transition.

Тестовая документация

Сначала обсудим, что такое тестовая документация — ведь именно ее в первую очередь запрашивают у тестировщика. Подкину вам хорошую ссылочку на статью «Зачем и почему нужна тестовая документация?»[115].

Тестовая документация — это все документы, которые готовят тестировщики.

Если говорить про описание проверок, это может быть:

- тест-план;
- тест-кейсы;
- чек-листы;
- что-то среднее;
- майнд-карта проверок;
- что-то еще.

Если про все остальное, то это:

- отчет о тестировании;
- баг-репорты;
- улучшения;
- другие задачи в баг-трекере.

Тестовую документацию также называют *артефактами тестирования*. Чтобы вы не пугались, если услышите новые слова на собеседовании!

А пока обсудим, что есть что. Большую часть этих документов мы уже рассматривали:

- майнд-карты — в *главе 1*;
- тест-кейсы и чек-листы — в *главе 2*;
- баг-репорты — в *глава 5*.

А что означают остальные?

Тест-план

Тест-план — это именно план:

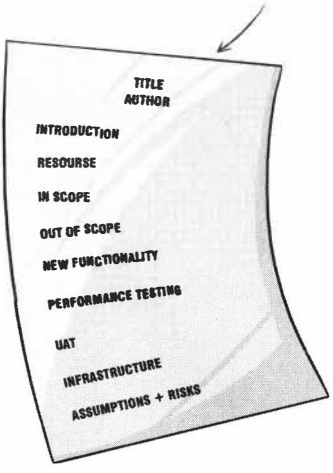
- когда;
- что;
- зачем;
- какими ресурсами.

Сюда вы пишете, что будете тестировать и что НЕ будете, потому что: нет времени, не готов функционал, не стоит такая задача, заказчик не хочет за это платить, нет железа (для нагрузочного тестирования, например).

В нашу задачу не входит учиться составлять тест-план просто потому, что это задача не для новичка. План составляет опытный тестировщик. Однако, когда придет время его писать, помните, что принцип «Кратко, но емко!» действует везде. Включая тест-план.

Если погуглить шаблоны тест-планов, вы найдете кучу многостраничных документов. Да, так по ГОСТу: описать то, сё, пятое, десятое. Но такой план никто читать не станет. Просто поверьте мне. Или попробуйте сами почитать любой пример, многое запомните?

Это канонический шаблон для тест-плана. Осторожно! Если написать много текста, никто его не прочитает!



Чем меньше информации и текста — тем лучше. Делайте тест-план на одну страничку, это вполне возможно! Есть даже такая статья «Тест-план на одну страницу», вот будет вам в помощь[116].

Test-plan или test-suite?

Выпускники моих курсов иногда приходят в чат выпускников с вопросом «Что такое тест-план?», периодически также сокрушаясь, почему я не учила его составлять, злодейка, ведь на собеседовании просят!

Так вот, если на собеседовании для джуниора у вас просят тест-план, то тут два варианта:

- на самом деле они ждут тест-сьют;
- они, правда, ждут от джуна тест-план, но это очень странные требования. Примерно, как «опыт работы 10 лет, возраст не более 18 лет».

Обычно под планом тестирования они понимают именно набор тестов. Учтите, что собеседование проводят люди, далекие от тестирования. Особенно если речь о стартапе, куда ищут первого тестировщика.

Они не обязаны знать, что такое тест-план. Для них это «как вы будете тестировать функционал». Ну а что, план ведь? План! Так и называют.

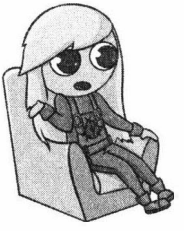
Поясню, что test-suite (тест-сьют) — набор тест-кейсов. А тестовый набор — набор тестов, то есть это может быть как пачка тест-кейсов, так и чек-лист. И, скорее всего, чек-лист их устроит.

Если вы прямо на собеседовании сидите, уточните у спрашивающего, чего именно он ждет. Поясните свой вопрос, расскажите о разнице между

набором тестов и тест-планом, который отвечает еще на кучу дополнительных вопросов.

Если вам прислали тестовое задание по почте — все равно уточните. Или напишите чек-лист и пришлите в ответ, уточнив, почему именно так сделали.

Беседа у камина



Test plan

Test suite

Слушай, а зачем ты нужен? Ведь можно написать тест-план и решить сразу несколько задач, а не одну!

Я везде, а тебя что-то давненько не видел...

Но ведь ты — часть меня!

Возможно, но заметь, вполне самостоятельная часть

И все же...

Меня можно использовать при составлении плана, да. Ну и что? Если соединить листья разных деревьев в гербарий, разве это будет значить, что клен и осина — просто часть гербария пятиклассника Васи?

Ну хорошо, ты отдельная личность. Зато я отвечаю на кучу вопросов: когда, что, зачем, какими ресурсами. А ты — только на «что», да и то не полностью. Ведь в «что» могут быть вообще разные виды тестирования, а не тест-сьют. Тут у нас системное, тут интеграционное, тут бету начинаем, тут нагрузку тестим...

Все сказал?

Нет, подожди, я еще не закончил перечисление. А еще в «что» может быть...

(зевает)

Тебе неинтересно? Во мне столько возможностей!

Да, поэтому меня используют постоянно, а тебя нет. Не стоит усложнять. Простая программа, выполняющая конкретную задачу, всегда лучше громоздкого интерфейса. Если писать полноценный план, тестировщик потратит кучу времени! А если никто не будет читать результат, то это просто деньги на ветер...

Руководители будут! Я очень важен! *(раздувается от гордости)*.

Руководители? Смешно! Им вообще не до тебя, своих забот хватает.

Вообще-то план можно сделать кратким, вон статья про тест-план на одну страничку!

Ну, может и можно. И все равно все эти нюансы никого не интересуют. «Когда, зачем...». Набор тестов — вот всё, что нужно для счастья!

Да? А когда заказчик будет спрашивать с нас, почему не проверили функционал, который не входил в исходный план работ, ты ему что ответишь?

Что мы так не договаривались!

Не смеши меня, без бумажки ты букашка. Останешься виноват.

Пожалуй, ты прав, ты тоже бываешь нужен...

Ну, спасибо!

На том и порешим!

Отчет о тестировании

Это документ, который вы показываете начальнику после проделанной работы. Нужен не всегда: я такое делала только на фриланс-подработке — там надо было как-то рассказать, чем я занималась.

Что важно? Если начальник просит вас сделать отчет, уточните, зачем это и как будет использоваться. И присылайте максимально краткий отчет. Его быстрее читать и писать. Не стоит тратить кучу времени на то, что, возможно, читать-то не станут.

Вот, например, в работах студентов есть описание исследовательских туров[117] — как они по ним проходили. Фактически это — отчет о проделанной работе, о проведенном тестировании. И он может быть кратким!

Тур супермодели. The Supermodel Tour

Цель тура — проверить, как приложение выглядит и какое первое впечатление производит. Этот тур выбран, так как основная цель приложения — предоставить пользователям удобный, максимально понятный интерфейс для осуществления совместных поездок. Приложение BlaBlaCar «Поиск попутчиков»: **https://play.google.com/store/apps/details?id=com.comuto&hl=ru** проходит тур супермодели.

Что успели протестировать в ходе путешествия, на что обратили внимание?

- Для поиска и просмотра поездок не обязательна авторизация.
- Для публикации, бронирования поездки авторизация обязательна.

- Найти поездку можно, заполнив в поле поиска два обязательных поля (откуда и куда осуществляется поездка).
- При отсутствии поездок по запросу можно создать уведомление о том, что поездка появилась.
- Привязка отправления и прибытия к конкретному месту в городе. Улучшение: сделать возможным выбор любого адреса, сейчас доступен выбор только определенных мест в городе (ТЦ, АЗС), что не всегда удобно.
- Приложение работает только в вертикальном режиме смарфона. Улучшение: сделать возможность работы приложения в горизонтальном режиме.

Всё! А можно написать целую историю — тогда получится длиннее, но интереснее. Посмотрите работу другого студента: «Тур супермодели для Азбуки Вкуса»[118].

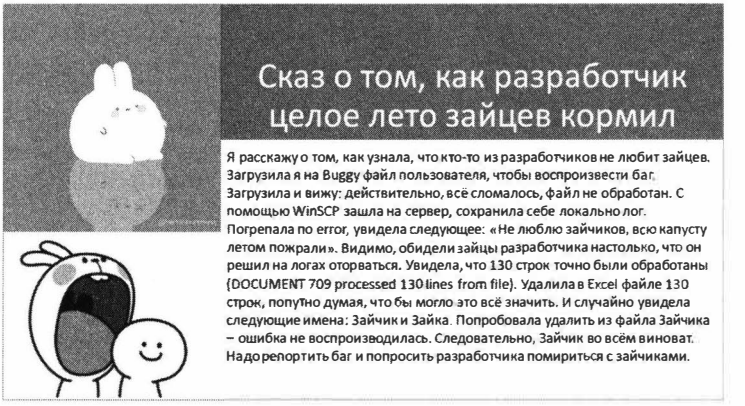
Вообще, если есть картинки или написана интересная история — читать веселее. Значит, больше шансов, что отчет хоть кто-то осилит. Думаете, это несерьезно и начальник читать веселые картиночки не будет?

Неправда, вот вам история о том, как мы улучшили Release Notes[119]— информацию об изменениях в продукте. Мы рассылаем эту информацию бизнесу в банках и страховых. И поверьте, когда вместо сухой информации там появились истории, документ стали читать.

У меня на курсах для тестировщиков тоже часто бывает нужен отчет о тестировании. Так тренер понимает, правильно студент думает или нет.

И вот есть задачка: «Загрузить файл, проверить в логах, из-за чего система развалилась, локализовать». Можно прислать буквально пару строк: «Ошибка такая-то, понял так-то». Одна из студенток, Александра, пошла

дальше и смастерила презентацию на три слайда о том, как она расследовала этот баг. Даже картиночки подобрала!



Я долго смеялась, пока рассматривала презентацию, поставила «Отлично!». И именно так и работают картинки, шутки-прибаутки, истории «Как я пытался локализовать баг» — ваш отчет ЧИТАЮТ.

Да, не всем понравится такой стиль изложения. Да, можно обойтись без картинок. Но ведь можно сделать график, диаграмму или что-то такое! Это все равно освежит ваш отчет.

Можно сделать рассказ чуть интереснее, описав в виде истории. Попробуйте! Вернуться к простой отчетности можно всегда. А даже если у вас сухие факты, помните про «кратко, но емко!».

Я, например, когда работала на фрилансе, написала чек-лист проверок функционала в экселе. И когда вечером тестировала, просто раскрашивала названия проверок:

- зеленый — прошла успешно;
- красный — ошибка!
- белый — не тестировала (времени не было).

Каждый день можно было посмотреть, что именно я проверяла. Но начальство посмотрело такой «отчет» раз, два, а потом забило: «Работает — не трогай». Откуда я это знаю?

В один прекрасный момент баг нашла не я, а заказчик. Разумеется, ко мне пришли с вопросом:

— Почему ты это не нашла?

Я в растерянности смотрю свой чек-лист: неужели не догадалась о такой проверке? А там проверки функционала белые все — я до них еще не дошла!

Стали договариваться о том, сколько времени я уделяю тестированию, нужен ли мне помощник и т. д. Но факт остается фактом — мои отчеты перестали смотреть. Так стоит ли тратить много времени на то, что никому не нужно?

Обсудите с начальством, в каком виде им нужен отчет и сколько времени вам стоит на него выделять. Если начальник просит большой отчет, просто покажите свои трудозатраты за неделю — хочет ли он оплачивать один или несколько человеко-дней на отчет, который читает через раз и по диагонали?

Проектная документация (ТЗ)

Прочитали Савина? Помните, что «баги — это несоответствие ТЗ»? А теперь окунитесь в реальность — полноценного технического задания не существует. Это миф. Не получится прийти на работу и сказать: «Тестировать буду только по ТЗ, а пока отдыхаю».

Если компания:

- маленькая — ТЗ нет вообще, все в головах разработчиков;
- большая — ТЗ есть и много, но актуально оно процентов на 15.

Хорошо, когда есть небольшая инфостильная[120] документация, которую заботливо написал внятный трезвый аналитик. Но мы рассматриваем ситуацию, когда ее нет. Что делать? Писать!

Писать придется вам. Если ходить и пинать других людей: «Напишите доку, это ваша обязанность» — никто ничего не напишет. Но и тестировать без документации грустно. Вы забудете что-то проверить. Или разработчик реализует иначе, чем вы думали, или мир сойдет с ума. Но будет грустно.

Что делать? Общаться с носителями знаний и записывать результат. Да, самому. Нет, не надорветесь. Хочешь изменить мир — начни с себя.©

Как это сделать? Я не буду учить вас подробному анализу и прочему — мы все же не аналитики. Тут можно обратиться к соответствующей литературе. Я предлагаю простой вариант:

1. Нашли носителя информации.
2. Расспросили.
3. Поправили документацию или написали новую.
4. Показали носителю, попросили проверить, правильно ли поняли.

Общаясь с носителем знаний, записывайте результат!

Тут вот какая фишка:

- писать с нуля — долго;
- редактировать существующее — быстро.

Писать с нуля — долго.
Редактировать существующее — быстро

Так что даже занятой аналитик может посмотреть, что вы наваяли. И все в плюсе: документация появилась, вы можете спокойно тестировать, не боясь что-то забыть, а аналитик не потратил много времени.

В каком виде записывать ТЗ? Да в любом! Ваша задача — сохранить информацию. Всё. Хоть какая-то заметка, написанная корявым языком, будет лучше, чем ничего.

Я предлагаю несколько простых вариантов, которые еще и в тестировании помогут:

- шаблон компании;
- вариант использования;
- state & transition diagramm.

Шаблон компании

Если в компании уже есть свой стиль описания ТЗ и он простой и понятный — используйте его.

Это могут быть краткие заметки, графические рисунки, блоки кода — что угодно, что принято у вас. Если вы нашли «белое пятно» в документации — просто заполните его по шаблону. И всё!

Вариант использования

Это описание взаимодействия пользователя и системы. Пользователь что-то делает, а система ему отвечает. Расписали вариант использования системы? Подумали об альтернативах. Смотрим на каждый шаг — может ли он пойти иначе? Если да, то как? Ну а под конец расписали параметры — переменные, которые не влияют на описание шага, но нужны и важны. Вот пример варианта (сценария) использования.

СИ01. Найти товар и сделать покупку

Легенда

П — пользователь;

С — система.

Сценарий использования

1. П открывает список товаров и фильтрует по категории.
2. С отображает товары выбранной категории.
3. П видит интересный товар и переходит на его карточку.
4. С отображает карточку товара, оценку покупателей и отзывы.
5. П изучает товар и кладет его в корзину.
6. С добавляет товар в корзину.

7. П переходит в корзину и оформляет заказ.

8. С сохраняет заказ, отправляет уведомление по email.

Альтернативные варианты

1а. П фильтрует список по несуществующей категории. Система выдает ошибку. Завершение сценария.

2а. Товаров не найдено. Вывод сообщения об ошибке. Завершение сценария.

2б. Товаров слишком много. Система выводит первые 100 и предлагает сузить поиск.

5а. П возвращается к покупкам. Переход к шагу 1.

Параметры

Категории товаров: платья, джинсы, свитеры.

Время хранения товара в резерве: 2 часа с момента добавления в корзину.

Подробнее об этом можно почитать в книге Алистера Коберна «Современные методы описания функциональных требований к системам». Достаточно прочесть первые пару глав. Но и от его варианта можно отсечь лишнее.

А ещё больше подробностей можно найти в моем блог-посте «Как составлять вариант использования»[121]. Это как раз для тестировщика инструкция!

Написание варианта — это здорово. Ведь это сразу и документация, и практически готовые тест-кейсы:

- вариант — основной позитивный тест;
- каждая альтернатива — негативный тест;
- каждый параметр — дополнительный позитивный тест.

Decision Table (таблица решений)

Таблица решений — техника, помогающая наглядно изобразить комбинацию условий из ТЗ. Каждая строка или столбец таблицы — готовый тест-кейс.

Как составлять таблицу?

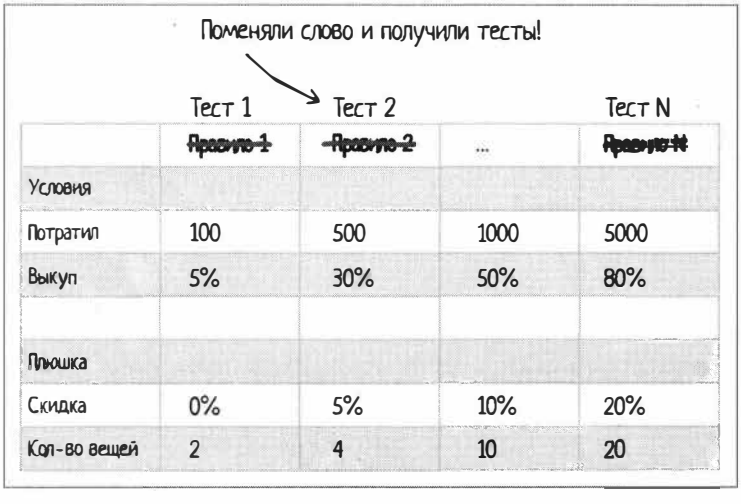
- по горизонтали — выписываем условия, которые влияют на результат. А чуть ниже — сам результат, в оригинале Action: действие, которое нужно выполнить;
- по вертикали — правила: конкретная комбинация входных условий.

Условий может быть сколько угодно. И действий тоже может быть больше одного. Получается больше правил и больше возможности их скомбинировать. Выглядит это примерно так, как показано в табл. 8.1.

***Таблица 8.1.** Примерная схема таблицы решений*

	Правило 1	**Правило 2**	**...**	**Правило *N***
Условия				
Условие 1	Нет	Нет	Да	Да
Условие 2	Да	Нет	Нет	Да
Условие 3	Нет	Нет	Да	Да
Действия				
Действие 1	Do X	Do Y	Do X	Do Z
Действие 2	Do A	Do B	Do B	Do A

Такая табличка отлично помогает воспринимать текст. Намного нагляднее же. Но ещё это готовые тест-кейсы. Поменял в заголовках слово «правило» на «тест-кейс», и вот они, готовые тесты! И это будут основные позитивные тесты, которые мы проводим в первую очередь.



Если неудобно, что тест приходится читать сверху вниз, а не слева направо, таблицу можно инвертировать — перевернуть (табл. 8.2).

Таблица 8.2. *Инвертированная примерная схема таблицы решений*

Тест-кейсы	Условие 1: Потратил	Условие 2: Выкуп	Результат
Кейс 1	100	5%	Do X / Do A
Кейс 2	500	30%	Do X / Do Y
Кейс 3	1000	50%	Do B / Do C
Кейс 4	5000	80%	Do B / Do Z

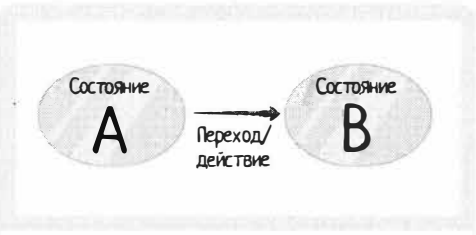
Я настоятельно рекомендую применять эту технику хотя бы однократно — к тем требованиям, которые вы уже знаете наизусть. Думаете, проверили все возможные комбинации? Нарисуйте таблицу решений, и результат вас удивит!

Таблица решений отлично подходит для размыливания взгляда и действительно сложных ТЗ, когда она содержит 100 колонок. Поддерживать ее достаточно накладно, и вряд ли кто-то станет это делать, а вот одноразовая акция найдет баги!

Подробнее о применении этой техники см. в статье на Хабре: «Decision Table — что это и как применять»[122].

State & Transition Diagramm (схема состояний и переходов)

State & Transition Diagramm (сокращенно S&T) — схема состояний и переходов. Она наглядно показывает, как некий объект тестирования переходит из одного состояния в другое. Вот он находился в состоянии A, потом произошло какое-то действие, и объект попал в состояние B.



Состояние и переход

Потом он попадет в состояние C и другие... Принцип не меняется: было одно состояние, стало другое.

Мы рисуем:

- кружочки — состояния объекта;
- стрелочки — то, благодаря чему перешли из состояния A в состояние B.

Это действие, но его может совершить не только пользователь, но и система. Например, задача запустилась автоматически в 10 часов вечера

Такая схема позволяет нам сразу визуально оценить, какие переходы вообще возможны и что надо протестировать. Ведь нам надо протестировать и эту стрелку, и эту... Так что стрелочки — это готовые тест-кейсы!

Очень важно: S&T рисуется на объект! Один объект! В идеале — на объект, имеющий аналог в базе данных продукта.

Как рисовать схему:

Шаг 1. Вы выбираете объект в своём проекте (рабочем или учебном, не суть).

Шаг 2. Думаете, какие у него состояния. Они не должны пересекаться, то есть: объект не может быть разом в двух состояниях, и при этом он всегда хоть в каком-то одном есть.

Шаг 3. Рисуете эти состояния кружочками.

Шаг 4. Соединяете их стрелочками. Стрелочки — это действия, их надо подписать.

Шаг 5. Смотрите, что получилось, и анализируете, есть ли у объекта другие состояния? А другие действия между текущими состояниями?

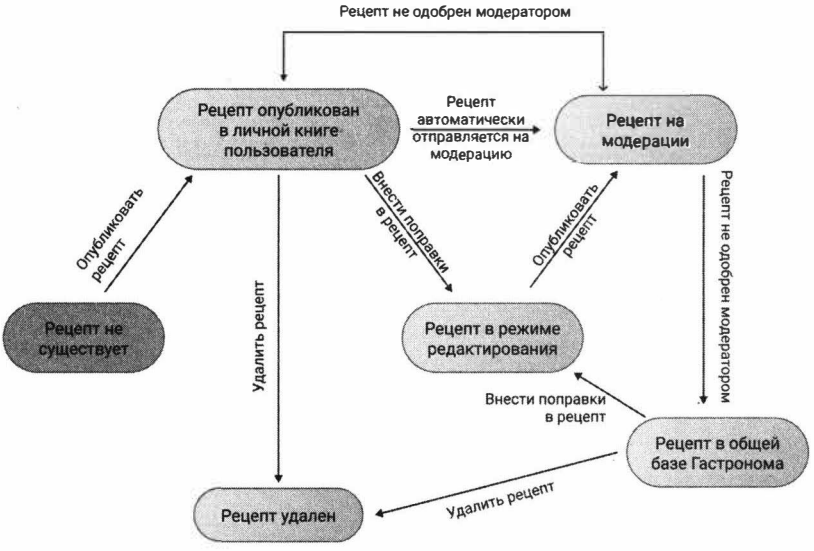
Переход на Шаг 2.

Кто не будет выполнять эту последовательность шагов, очень рискует вместо S&T нарисовать схему вышивки крестиком ;-)

Чтобы начать, задайте себе вопросы:

1. Какой конкретно объект вы выбрали? Как он называется (только один!)?
2. Какие у этого объекта есть состояния?

Основное определение состояния: «набор доступных и недоступных действий с объектом». Продукт всегда должен знать, в каком состоянии каждый его объект. Вот пример хорошей диаграммы:



Ещё больше примеров вы можете найти в выложенных в общий доступ работах моих студентов[123].

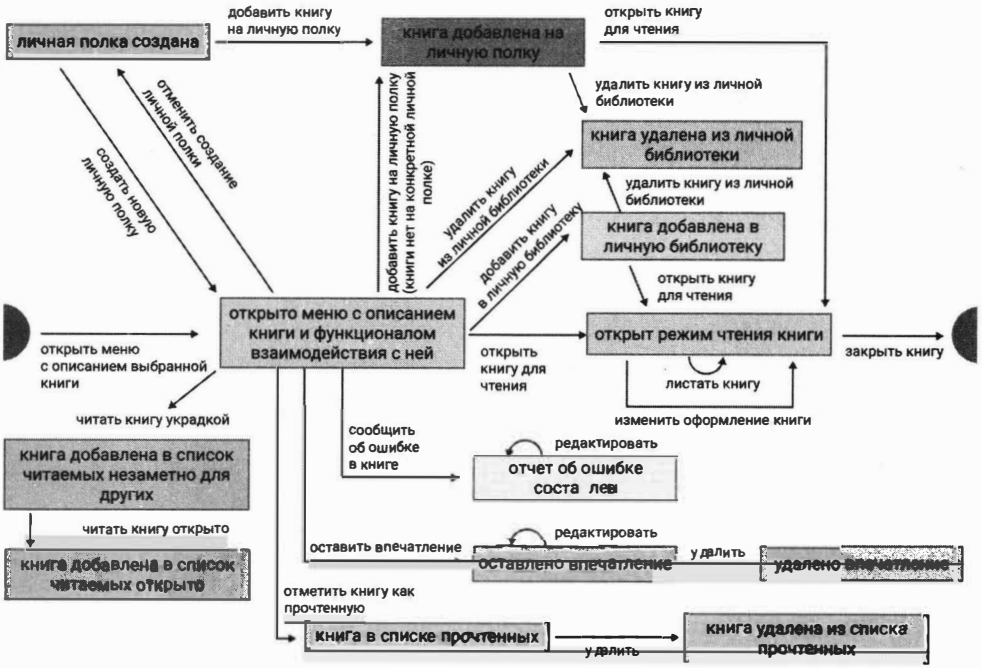
Ну а подробнее о технике S&T читайте в статье на Хабре: «State & Transition Diagram — что это и как применять»[124]. Я вам настоятельно рекомендую эту технику. Ведь рисунок — мощнейший инструмент визуализации.

Вот вы читаете эту книгу, и в ней везде картинки. На что вы смотрите в первую очередь: на картинку или на текст? Правильно, на картинку. Поэтому я за то, чтобы рисовать! Нарисовали? Добавили в ТЗ! Всем удобнее, даже заказчику. Ведь с картинками текст становится понятнее.

Другие диаграммы, схемы, картинки

Не S&T единым, как говорится... Нам не обязательно рисовать именно состояния объекта. Можно нарисовать вообще всё, что угодно. Берем требование и представляем в графическом виде. Всё!

Вот, например, как может выглядеть функционал взаимодействия с конкретной книгой[125]:



Это карта сценариев, а не S&T, но от этого она не менее полезная! Ещё больше примеров вы можете найти в выложенных в общий доступ работах моих студентов[126].

Двойной профит визуализации: пока вы рисуете, то видите все слабые зоны ТЗ. А еще делаете его более наглядным. Теперь каждый, кто будет читать ТЗ после вас, сразу всё поймет!

Рисунок — мощнейший инструмент визуализации. Если вам сложно что-то понять, зарисуйте это! Это может быть сложный участок ТЗ или чек-лист тестов. Пробуйте, экспериментируйте — это время обязательно окупится. Причем сразу же — ведь, пока рисуешь, приходит вдохновение: «проверить еще вот тут».

Ну а подробнее о технике визуализации читайте в статье на Хабре: «Визуализация ТЗ — диаграммы, схемы, картинки»[127].

Подведем итоги

Да, тестировщик — не аналитик. Но писать требования самому безумно интересно. Вы узнаёте детали реализации, прикидываете будущий комплекст тест-кейсов и просто приносите много добра!

Пока вы методично выписываете все условия в таблицу решений либо зарисовываете все условия и думаете, какие еще стрелочки на диаграмме могут быть, — это уже позволяет заметить какие-то тесты, о которых вы не подумали раньше.

Таблицы решений и графические решения вполне можно использовать одноразово. Пришли, протестировали с помощью класов эквивалентности, граничных значений и прочая, и прочая... А потом сидим, смотрим на нашу систему и думаем «Хм, да я, вроде, уже столько раз ее тестировал и даже не представляю, где еще могут быть баги». И вот здесь как раз можно применить что-то новое. Один раз нарисовали табличку на 100 колонок, получили новые тесты, нашли новые баги, а табличку потом перенесли в архив или выкинули. Свою работу она уже сделала!

Это тоже — нормально! Главное, что на текущем этапе вы свой результат получили! Вы нарисовали, вы визуализировали, возможно, вы эту схему показали коллегам, чтобы вместе обсудить, как это стоит реализовывать.

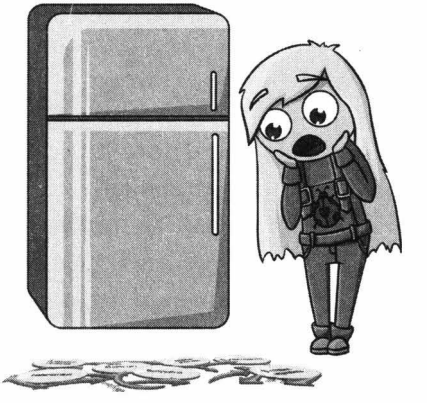
Домашнее задание

Вопросы для самопроверки

1. Кто пишет тест-план?
2. В чем отличие альтернатив от параметров в варианте использования?
3. Чем плох вариант: «Пользователь выбрал файл. Система начала загрузку»?
4. Когда не нужно использовать Decision table?
5. Что мы рисуем в кружочках в S&T?

Магнитики на холодильнике

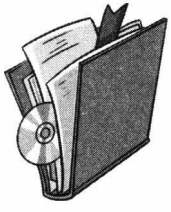
Схема S&T для этой книги рассыпалась на кусочки! Попробуйте собрать ее. Учтите, что некоторые магнитики лишние.



Кружочки:
1. Книги нет.
2. В процессе создания.
3. В стадии черновика.
4. На редактировании.
5. Отдана в печать.
6. Напечатана.
7. Выпущена.

Стрелочки:
1. Написать главу 1.
2. Исправить опечатки.
3. Придумать картинки.
4. Придумать идею книги.
5. Начать делать.
6. Вставить картинки в книгу.
7. Отдать редактору.
8. Правки по комментариям.
9. Обсудить.
10. Отдать в печать.
11. Напечатать.
12. Разослать по почте.
13. Добавить в магазин.

Портфолио



Продолжаем делать портфолио.
1. Напишите вариант использования на основной функционал вашей программы.
2. Нарисуйте S&T для вашей программы — тут очень важно правильно выбрать объект!
3. Попробуйте еще как-то визуализировать тесты или ТЗ.
4. Составьте Decision table там, где это оправданно.

Магнитики на холодильнике (ответ)

Создание книги

Ответы на вопросы для самопроверки

1. Кто пишет тест-план?

Опытный тестировщик, не джуниор! Джуниор может составить test-suite (набор тестов).

2. В чем отличие альтернатив от параметров в варианте использования?

- ✧ Альтернатива — это когда ВМЕСТО исходного события происходит другое.
- ✧ А параметры — это когда В ОДНОМ И ТОМ ЖЕ событии есть несколько вариаций, как его совершить.

3. Чем плох вариант: «Пользователь выбрал файл. Система начала загрузку»?

Потому что система — ванга-терминатор. Пользователь мысленно выбрал, а она что-то фигачит. Так не бывает, нужно действие со стороны человека.

4. Когда не нужно использовать Decision table?

- ✧ Слишком простое условие — она не нужна.
- ✧ Слишком много входных данных — тут можно, просто трудозатратно. Но рассмотрите одноразовый вариант, если функционал критичный

5. Что мы рисуем в кружочках в S&T?

Состояния объекта.

Глава 9
КЛАССИФИКАЦИЯ ТЕСТИРОВАНИЯ

Пожалуй, пора научиться классифицировать проводимые тесты. Что такое черный и белый ящики? Как отличить нагрузку от производительности? Что такое НФТ?

В этой главе мы обсудим классификацию:

- по знанию системы;
- по позитивности;
- по целям (по объекту);
- по исполнителям (по субъекту);
- по затраченному времени;
- по степени автоматизации;
- по состоянию системы;
- по формальности.

Классификация: что это и зачем?

Что такое классификация?

Классификация — это упорядочивание данных. Допустим, у нас есть коллекция бабочек. Если не упорядочить ее, будет хаос.

Пока бабочек в коллекции мало, можно ее не упорядочивать. Вам хватит одного взгляда на коллекцию, чтобы найти нужный экземпляр.

А вот если коллекция растет, находить там что-то становится все сложнее. Без упорядочивания это будет долгий процесс. Если друг спросит:

— А чего тебе не хватает для коллекции? — вы даже не сможете сразу ответить на этот вопрос. Ведь чтобы понять, какого вида у вас нет, надо выписать все существующие. А если один раз отсортировать всех бабочек, то всегда будет видно:

— Так, из белянок мне только альпийской не хватает, и конопряда сибирского нет.

Более приземленный пример — бумаги на рабочем столе. Мало? Легко найти нужную даже в творческом беспорядке. Много? Фиг найдешь. Другое дело — если всё разложено по папочкам: тут у меня счета, на это надо ответить, а это подписать у директора.

Так же и с тестированием — классификация помогает нам упорядочить наши знания и ответить на вопросы:

Где есть порядок, там легко искать.
Классификация = порядок

Классификация = упорядоченность

- что мы уже знаем?
- что умеем?
- где у нас есть пробелы (пустые места, которые стоит прогуглить и найти книгу или курс) и куда развиваться дальше?

Минусы классификации

У самой классификации особых минусов нет. Но есть большой минус в культе определений. Классификацию считают настолько важной, что:

- она идет в первых главах книг по тестированию (но ЗАЧЕМ??);
- ее часто спрашивают на собеседованиях.

И вот ради собеседований новички гуглят, читают, учат... А какой смысл в зазубривании определений? Ну сможешь ты рассказать, чем тестирование производительности отличается от нагрузочного, чем smoke отличается от sanity, и что? А если при этом ни одного теста придумать не в состоянии, кроме «пройтись по ТЗ по главному сценарию», то в чем смысл проверки теории?

И уж тем более мне непонятно, зачем давать классификацию одной из первых тем в книгах. Чтобы что? Человек еще вообще ничего про тестирование не знает, что он поймет? Имхо, классификация должна идти где-то под конец. Когда уже что-то умеешь, чтобы наложить ее на свои знания. А теория ради теории не нужна.

Плюсы классификации

1. Вы видите свои слабые и сильные места.

Прочитали классификацию:

— Так, это я знаю, это знаю, а вот это не знаю и знать не хочу. А вот эту область хочется копнуть — надо погуглить статейки.

2. Менеджер знает, кому отдать задачу.

Допустим, сверху пришла задача проверить высоконагруженный сервис типа фейсбука. Менеджер знает свою команду, он прекрасно понимает, кто

знаком с нагрузочным тестированием, а кто нет. Он может отправить на тестирование новичка, чтобы тот узнал что-то новое, но только вместе со специалистом — нельзя же завалить проект!

Поэтому у менеджеров обычно есть своя классификация: перечень знаний и умений по каждому тестировщику. Какие сильные и слабые стороны у него есть. Зная эти особенности, легко набрать команду — где один слаб, другой силен, и наоборот.

3. Классификация помогает заметить риски.

Если мы составляем план тестирования, нам нужно решить:

- ✧ что мы тестировать будем;
- ✧ что НЕ будем.

Да, новички тест-план не составляют, но знайте на будущее. Когда вы решаете, какое тестирование НЕ будете проводить, можно заглянуть в классификацию — а все ли вы учли вообще? Есть ли непокрытые области?

Как принимать решение? На основе опыта, поэтому тест-план пишут не новички. Или на основе выделенных ресурсов. Тогда все решают приоритеты.

Если сайт делается для 10 человек, мы принимаем риск, что система может тормозить при нагрузке, — ведь мы четко знаем, что нагрузки не будет, зачем тратить силы на тестирование? А вот если мы делаем сайт, который станет вторым фейсбуком, стоит подумать о нагрузочном тестировании.

Поэтому мы просматриваем классификацию и говорим:

— Черный ящик будем проводить, белый не будем. Да и тестирование безопасности пока отложим.

Классификация помогает принять верхнеуровневое решение — посмотреть с высоты птичьего полета на то, что вы напланировали.

4. **Ее часто спрашивают на собеседовании новичков!**

И именно поэтому эта глава есть в этой книге. А иначе не было бы — научился основам, иди погугли сам. Но для собеседования пригодится. Только помните — лучше не заучивать определения, а понять, что чем от чего отличается.

А раз спрашивают, давайте мы классификацию и рассмотрим!

По знанию системы

Черный ящик (black box testing)

Тестирование черного ящика — это когда мы не знаем, как система устроена внутри. У нас нет доступа к коду или мы не умеем его читать. Поэтому ориентируемся только на внешнее поведение или ТЗ.

Это как начинающий автомобилист. Заглянуть под капот? Ой, там слишком страшно, будем верить приборам, и точка.

Тестирование черного ящика — это основной вид тестирования, с которого все начинают. Перед тем как залезть в код, стоит научиться писать тесты на основе сценариев использования и знаний про тест-дизайн и классы эквивалентности.

Если я сейчас открою любой сайт:

- http://software-testing.ru/;
- http://www.ozon.ru;
- http://users.bugred.ru/

и начну его тестировать: «А что если платьишко именно красное поискать? А красное с белым? А книгу по автору? А если спецсимволы вбить?» и так далее — это как раз тестирование черного ящика. Доступа к коду у меня нет, я просто проверяю, как программа работает.

У меня либо есть ТЗ, либо его нету. Составляю тесты, выполняю через интерфейс — типичный black box.

Плюсы подхода

Так как вы не знаете код, то не верите ничему на слово. Все проверяете по ТЗ, каждое слово. А бывает такое, что разработчик упустил в требованиях какое-то условие и просто его не сделал, или сделал, но не так, как надо. Если читать сам код, то все работает правильно. Но это не совсем то, что было нужно.

Белый ящик (white box testing)

Тестирование белого ящика — это когда у нас есть доступ к коду, и мы его тестируем. Читаем сам код (статическое тестирование), запускаем в дебаге, пишем автотесты...

Если сравнивать с автомобилистом, то тут уже не новичок, а прошаренный.

Сломалось что-то? Нафиг приборы — сразу лезем под капот, где все устройство чуть ли не наизусть уже знаем. Роемся там, ищем причину...

Самостоятельно освоить белый ящик трудновато, обычно с ним сталкиваются уже на реальной работе. Но можете поискать проекты с открытым *исходным кодом и тестировать их.*

В качестве примера открытого проекта вы можете взять *Folks*[128] — его можно скачать, запустить и даже написать свои автотесты, прогон которых

как раз и будет тестированием самого кода, то есть белого ящика. Пользовательского интерфейса (user interface) у Folks нет и не будет, так что тестирование черного ящика там недоступно.

Плюсы подхода

Можно писать меньше тестов. Мы ведь знаем, что тут и тут будет работать одинаково, видим по коду. Но иногда настолько углубляемся в код, что читаем именно его, а не ТЗ, и можем пропустить какие-то баги (см. плюсы черного ящика).

Серый ящик (gray box testing)

Тестирование серого ящика — это совмещение. Мы смотрим в код и понимаем, как он устроен. А потом открываем само приложение и проверяем, как этот код отображается уже в нем, но ориентируемся при этом больше на ТЗ.

Вернемся к примеру автомобилиста (это самый распространенный сценарий): чтобы получить права, мы должны пройти специальный курс, на котором рассказывается, как устроена машина внутри, — это знание кода, white box. Но потом мы начинаем кататься на своей машине и в основном ориентируемся на приборы — user interface, black box. При этом мы трактуем показания приборов на основе знаний кода.

И на приборы посмотрел, и под капотом порылся!

Если говорить про тестирование, то это любое тестирование системы через интерфейс, когда у вас есть доступ к коду, но нет знаний разработчика. Да, вы можете подсмотреть в коде какие-то простейшие вещи: какой тип данных у этого значения, какие настройки у полей и т. д. Но вот «потроха» сложных методов остаются за гранью ;-)

В итоге мы вроде знаем, как это все устроено, но частично. Поэтому перепроверяем по черному ящику, где-то напишем чуть больше тестов для перестраховки... В целом, серый ящик — самый оптимальный метод тестирования: вы и в коде баги можете найти, и по ТЗ проверить!

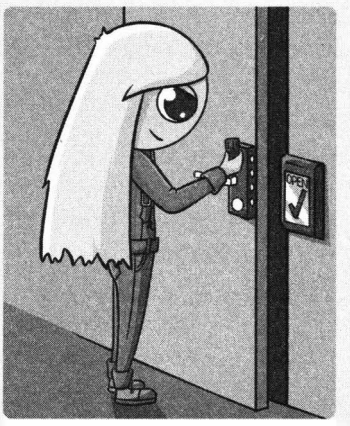
Главное, чтобы доступ к коду дали, хотя бы частичный ;-)

Хотя есть и другая версия — знание высокоуровневой структуры приложения: модулей, баз, например, но без доступа к коду. Или знание алгоритма, но опять же без доступа к коду. Например, по псевдокоду.

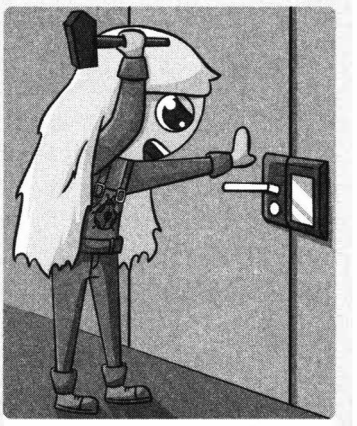
Смысл один: мы что-то из кода знаем, а что-то нет.

По позитивности

Это разделение мы обсуждали в самых первых главах, потому что оно очень важно. Помните главное правило? Сначала позитивное тестирование, а потом негативное! Хотите подробнее? — вернитесь к *главе 2*, а здесь — только напомню.



Позитивное тестирование проверяет работу



Негативное тестирование пытается сломать

Позитивное тестирование

Проверка того, что приложение работает так, как задумано:

- если есть ТЗ — проверяем по нему;
- если ТЗ нету — проверяем, как работает система, не пытаясь ничего сломать.

Это как послушный ребенок. Знает, что хочет от него мама, именно это и делает!

Позитивное тестирование (positive testing) — следуем ТЗ

Негативное тестирование

Круши, ломай, ПО побеждай!

- Форма ввода имени? Оставили ее пустой или вбили абракадабру.
- Можно грузить JPG-файл весом 100 Кбайт? Загрузили 1 Гбайт или PNG-картинку.

Всячески пытаемся пойти против правил и посмотреть, что нам на это скажет система.

Здесь аналогия с обиженным ребенком, который все делает маме назло!

Негативное тестирование (negative testing) — идём против правил, как отреагирует ~~мама~~ система

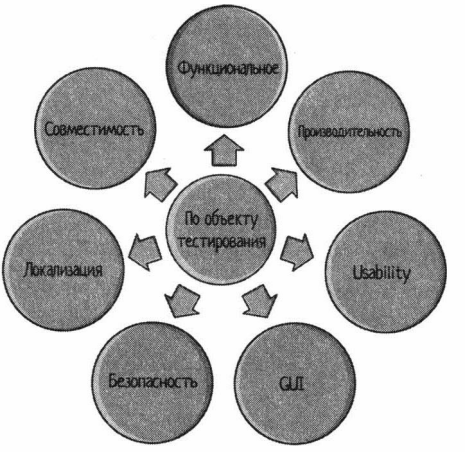
По целям (по объекту)

По объекту тестирования можно выделить два варианта:

- функциональное тестирование;
- нефункциональное тестирование (НФТ).

Или чаще делают разбивку более детализированную:

- функциональное тестирование;
- тестирование производительности;
- тестирование usability;
- тестирование GUI;
- тестирование безопасности;
- тестирование локализации;
- тестирование совместимости.



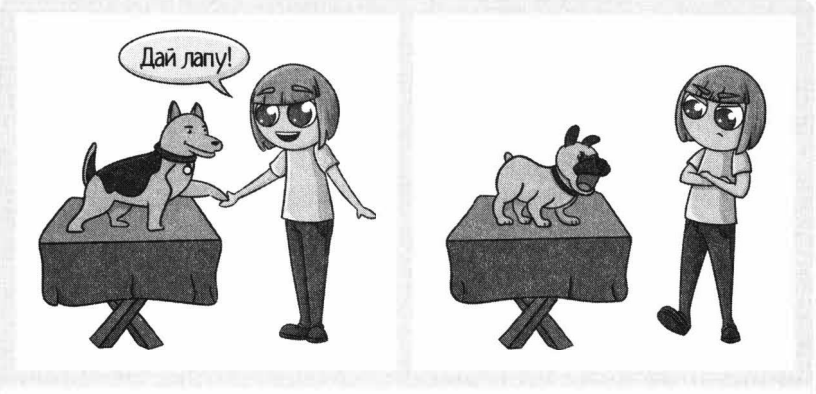
Пройдемся на каждому пункту!

Функциональное тестирование

Тестирование функционала:

- Могу ли я создать пост на форуме?
- Приложить скриншот к багу?
- Работает ли сложение в калькуляторе?
- Строится ли отчет?
- ...

Это основной вид тестирования, который применяет каждый тестировщик. Вы можете вообще не столкнуться с НФТ (что, впрочем, сомнительно — как минимум, на usability каждый замахивается в меру своего понимания), но функционал вы будете тестировать всегда.



Функциональное тестирование — смотрим на функционал

Нет, конечно, можно сразу заниматься тестированием производительности или безопасности, копая только эти темы. Но это редкость. Все-таки если уж вы носите звание тестировщика, значит, тестировать умеете. А тестирование — это про функционал в первую очередь.

Нефункциональное тестирование (НФТ)

Здесь мы смотрим не на то, что система умеет делать, а на то, как она это делает: быстро или медленно, понятно или неочевидно и т. д:

- Очевидно ли, как изменить фото в профиле?
- Удобно ли использовать наше приложение?
- Нет ли уязвимостей?
- Быстро ли загружается отчет?
- ...

НФТ — смотрим на красоту и другие характеристики

Раньше это было не столь важно. Если функционал есть — ура, используем! Но теперь вокруг слишком много конкурентов. Если мне не нравится один интернет-магазин, я пойду в другой и наверняка найду там похожее платье.

Если мне нужна читалка на смартфон, я делаю поиск и вижу сразу десяток аналогов. Функционал у всех этих аналогов примерно одинаковый. Поэтому я начинаю выбирать по другим критериям: приятный дизайн, скорость работы...

И именно поэтому НФТ сейчас набирает обороты — даже если вы НЕ умеете тестировать производительность или usability на профессиональном уровне, вы все равно обращаете внимание на скорость приложения и удобство его использования. И если что-то не нравится, то заводите баг или хотя бы обсуждаете возможность улучшения в курилке.

Мы в книге поговорим о НФТ как раз на таком, дилетантском уровне — чтобы понимать, что это в принципе такое. Но если область интересна, дальше изучайте сами. Я не эксперт + про каждую область можно отдельную книгу написать, зачем всё в одну-то запихивать ;-)

Итак, НФТ делится на разные области. В каждой мы смотрим на что-то одно: быстрота, красота и т. п. Давайте разберемся с каждой хотя бы на уровне базового понимания.

Тестирование производительности

Производительность — это скорость работы приложения:

- Как быстро открывается главная страница?
- Сколько времени будет работать REST-запрос на создание карточки?
- Как долго будет загружаться отчет с максимальным количеством настроек?

Нужно помнить, что один и тот же отчет может загружаться разное время. В одном случае грузим данные за последнюю неделю, а в другом — за последние 10 лет. Время загрузки получится разное.

Один и тот же поисковый запрос `Иванов Иван` на разных базах отработает по-разному. Одно дело — поискать по тестовой базе в 100 записей, совершенно другое — на реальной нагруженной системе на 100 млн данных.

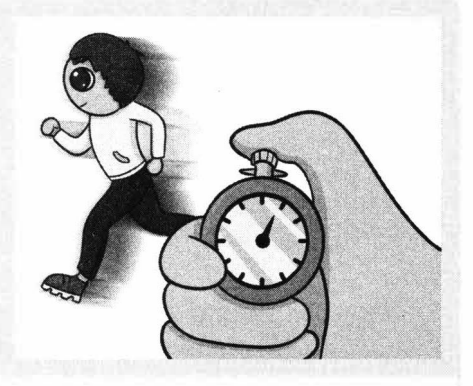
Поэтому проводим тестирование производительности:

- с разными параметрами;
- с разными настройками;
- в разных условиях.

Важно понимать, что разные параметры — это:

- минимум;
- максимум.

А то бывает как — тестировщик думает: «Раз производительность, значит, надо тестировать сложные случаи, большие объемы и т. д. Ведь именно это может тормозить». Но не менее важно проверить, как ведет



Производительность = скорость работы

себя система в ненагруженном состоянии. Как быстро отработает поиск на небольшой базе? Ведь если система в этом случае тормозит, то тестировать на большой базе просто бессмысленно.

В XXI веке от производительности сильно зависит настроение пользователя. Точнее, от плохой производительности. Это «гигиенический фактор»[129].

Если страницы грузятся быстро — я этого даже не замечу, ведь так и надо! А вот если мне приходится ждать по две минуты, чтобы увидеть расписание сеансов в кино, я лучше поищу другой сайт.

Поэтому важный вопрос — а как у конкурентов? Если у вас есть новостной сайт, который грузится две минуты, тогда как полно сайтов, отображающихся моментально, что выберет пользователь?

Используйте отсылку на конкурентов как обоснование бага. Это поможет избежать ситуации «быстрее нельзя, кучу новостей потеряем».

Если взять в пример интернет-магазины, там стараются на главную страницу запихать «всё самое лучшее сразу». Получается много информации и много картинок. Если разработчик поленился их ужать, страница будет грузиться долго.

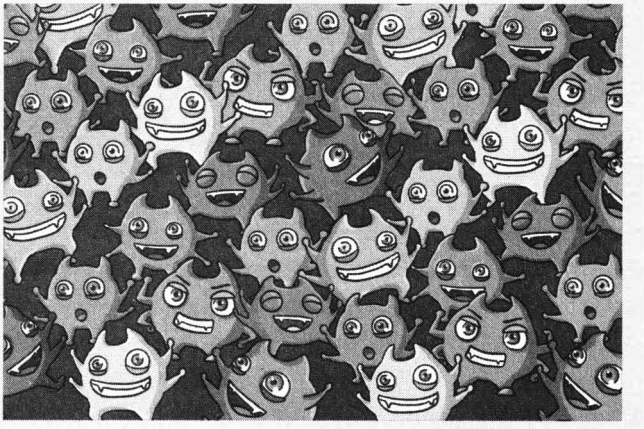
Сайт с рецептами бета-тест
на производительность не прошёл

И одно дело — подождать любимый сайт, сидя дома в тепле. А если человек смотрит товары со смартфона на улице с плохим Интернетом? А если при этом на улице холодно, и руки из варежек он достал только ради выбора новой шапки? Плюнет ведь и уйдет без покупки...

В наши дни мобильный Интернет очень распространен. Поэтому все сайты стараются делать как веб-версию, так и мобильную. Иначе они просто потеряют аудиторию. Мобильная версия оптимизирована по скорости (работать будет по плохому Интернету наверняка) + адаптирована по верстке. И производительность в этом случае очень важна!

Нагрузочное тестирование

Нагрузка — возможность одновременной работы с приложением большого числа пользователей.



Нагрузка — возможность одновременной работы

Сколько человек каждый день заходят в Инстаграм? А в Фейсбук? Сбербанк-онлайн? Если у вас высоконагруженное приложение, то тестировать нагрузку просто необходимо.

Во время нагрузочного тестирования важно проверить:

- производительность — насколько шустро приложение работает под нагрузкой?
- в какой момент наступит отказ?

Поэтому по результатам проведения теста строят такого вида табличку (табл. 9.1).

Таблица 9.1. *Пример результатов проведения теста*

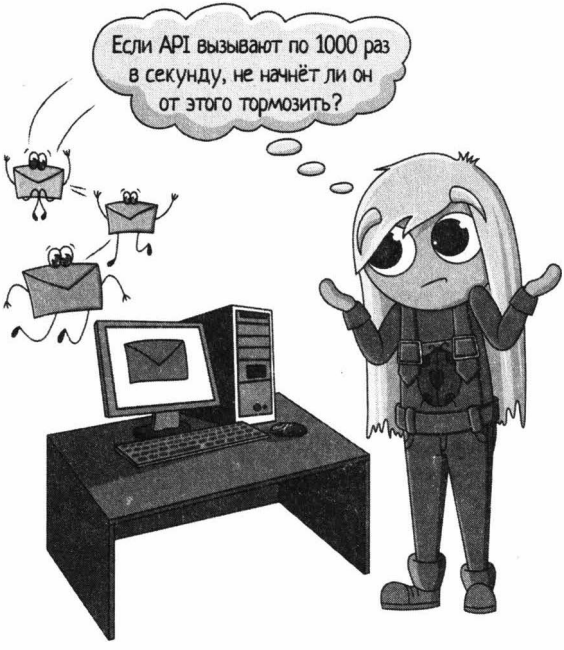
Число пользователей	Время ответа, с	Ошибок, %
1	0,01	0
100	0,1	2
1000	3	10
2000	n\a	100

Благодаря ей мы знаем, когда наступает отказ. Знаем, сколько ошибок будет для 100 или 1000 пользователей. И дальше уже думаем: устраивают нас эти цифры или нет?

Аналогично с производительностью. Если сайт ненагруженный, то и пусть идет деградация по времени ответа, не страшно. А вот если мы ожидаем 1000 пользователей, то придется оптимизировать код, чтобы он работал шустрее.

Нагрузочное тестирование нужно проводить не только для веб-сайтов с графическим интерфейсом. Это лишь самый простой для понимания пример. Но, разумеется, применимо это везде:

- Веб-сайт — сколько пользователей одновременно зайдет?
- Обработка данных — сколько потоков одновременно работает?
- API-интерфейс — сколько запросов идут в параллель?
- ...



Если приложение нагруженное, то оптимизировать надо всё. Не только циклы внутри кода, но и базу данных, и сервер приложения... Здесь пригодится помощь админов БД, которые смогут оптимизировать базу. А разработчики должны использовать хинты в коде и строить более оптимальные запросы.

В зависимости от характеристик «железа» нужно настраивать параметры операционной системы и менять количество потоков на чтение и запись из файлов... Проведите нагрузочный тест, чтобы понять, что именно нужно улучшить.

Стресс-тестирование

Стресс-тестирование (stress testing) — одна из форм тестирования, которая используется для определения устойчивости системы или модуля в условиях превышения пределов нормального функционирования. © Википедия.

Другими словами, стресс-тестирование — это проверка того, как система себя ведет в стрессовой ситуации. Почти как стресс-собеседование, только для системы ;-)

Тестирование производительности, нагрузочное и стресс

Студенты при изучении классификации часто спрашивают, чем отличаются между собой:

- тестирование производительности;
- нагрузочное тестирование;
- стресс-тестирование.

Моя коллега Ольга Алифанова привела прекрасный пример! Хотя среди опытных тестировщиков он вызывает споры: мол, это неправильные примеры, и всё такое. А по-моему, под определения эти примеры подходят и позволяют понять разницу на простом уровне. Как примеры с огурцами у Романа Савина (положил огурец в карман — вот и тест-кейс!)

1. Производительность: как быстро машина разгонится до сотни?

2. Нагрузка: как быстро она разгонится до сотни с 4 пассажирами и багажом?

То есть мы обращаем внимание на скорость и на то, что нет отказа, но наше приложение работает под нагрузкой.

3. Стресс: при каком весе на осях у нее подломятся балки?

Тестирование надежности (стабильности)

Тестирование стабильности или надежности (Stability/ReliabilityTesting) — проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

Если не перезагружать компьютер, рано или поздно даже ворд начнет тупить. Потому что «ну хватит уже, месяц ап-тайма[130], дай мне почистить внутренние кеши!». Или браузер — открыли вы кучу вкладочек, работает нормально. А через день-два-три-неделю начинает тормозить, пока не перезапустите. Это и есть надежность приложения — сколько он проработает в нормальном режиме?

Особенно это важно для мобильных смартфонов — вы вообще часто закрываете приложение? Я обычно просто жму на домашнюю кнопку, сворачивая его. А потом снова открываю. Приложения, не проверенные на надежность, постоянно зависают, вылетают и/или теряют соединение с сетью.

Цели тестирования надежности:

1. Выяснить, сколько времени сервис может проработать безотказно в конкретном окружении под конкретной нагрузкой. Не экстремальной нагрузкой, а простой.
2. Выявить утечки ресурсов. То есть в процессе теста тщательно мониторятся ресурсы системы (память, процессор, загрузка диска, файловые дескрипторы, сокеты).

И как раз из-за второго пункта приложения могут зависать после долгого ап-тайма (времени с момента запуска). Утечки ресурсов часто бывают на мобильниках — по крайней мере, в 2019 году я с этим постоянно сталкиваюсь, еще нет стандартных фреймворков, позволяющих этого избежать, видимо[131].

Также нужно внимательно следить за утечками на сервере приложений. Вот вы открываете интернет-магазин или тот же Testbase[132]. Для вас это просто страничка в браузере. А на самом деле на сервере запущено приложение. И оно ОЧЕНЬ редко гасится. Поэтому, если приложение выжирает ресурсы, за этим нужно следить.

Согласно Википедии, тестирование надежности — один из видов автоматизированного тестирования ПО. Да, в идеале это автоматизируется. Но если вы не умеете автоматизировать и автоматизации на проекте нет — это не значит, что надежность проверить нельзя. Можно, используя тур «чашки кофе»[133]. Включили приложение и оставили его куковать.

Для этого нужны специальные стенды — тестовые-то обновляются каждый день, как разработчик сборку новую выкатит. Поэтому, если нет отдельного стенда, который обновляется РЕДКО, вы можете упустить проблемы с утечками памяти. Просто не успеете упереться в лимит. Учтите это!

Тестирование usability

Usability-тестирование — тестирование удобства использования. Насколько наше приложение нравится пользователю? Хочется ли его использовать?

Так как функционал у конкурентов часто похож, то юзабилити выходит на первый план. Если я хочу купить красное платье и нашла одно «прям вау», я его закажу, даже если сайт будет вставлять мне палки в колеса: для заказа обязательно зарегистрируйся, заполни 100500 полей и так далее.

Да и то бывают такие сайты, которые снижают уровень «хочу-хочу-хочу!» до нуля — плюешь и бросаешь. А что если я хочу неуникальную вещь? Ну, скажем, неваляшку сыну купить. Я найду ее на десятке разных сайтов: Детский мир, Кораблик, Toy.ru и т. д. Вот теперь я могу выбирать!

Как я выбираю интернет-магазин? По цене товара. Если она одинаковая, то выберу тот, где дадут какие-то бонусы. Может, я уже зарегистрирована в каком-то и буду получать 5% баллами на карточку. А если еще нигде не зарегистрирована, то выберу магазинчик, который наиболее симпатично смотрится. Если же он окажется не user-friendly при оформлении заказа, поищу другой.

Вот, например, купила я в «Кораблике» сыну пару кофточек. Бренд одежды понравился — он на высоких и худых. То, что нам надо, а то обычно купишь кофточку, а она широкая. Решила взять еще парочку на вырост. А в ближайшем ко мне магазине их нет. Захожу в интернет-магазин, пытаюсь заказать. И что же? В корзину добавляются, а заказать нельзя: как только указываешь «доставка курьером», вещи из корзины ВЖУХ и испаряются.

Магазин, удаляющий товары из корзины, не user-friendly!

Можно только забрать из реального магазина. Но не из любого! А только того, где эти кофточки есть, — на другом конце Москвы. Опции «прислать в магазин в Бутово» у них тоже нет. Ну что за бред! Это же одна сеть магазинов, ну как так, почему товар есть, а я не могу его заказать? Пару раз пробовала что-то там заказать, обплевалась, теперь обхожу магазин стороной.

И это как раз проблема usability, потому что мне неудобно использовать сайт. Заметьте, что usability-проблемы бывают разного уровня:

- интерфейсные — когда вместо кнопки «заказать в один клик» нужна регистрация, при которой обязательно указать клички всех домашних животных, одноклассников, всех учителей... Подключили разработчика, исправили формочку — всё!
- человеческие — это как раз тот случай, когда нельзя заказать через интернет-магазин то, что есть в одном из магазинов сети. Для разработчика добавить кнопку — тьфу, раз плюнуть. Но проблема находится выше, тут надо принять решение о том, что мы ХОТИМ дать пользователю такую возможность. Это вопрос к руководству.

Самое главное правило юзабилити:

Если ты — не целевая аудитория продукта,
то ты понятия не имеешь, что такое
«удобно», а что такое «неудобно»!

Поэтому, если лично вам неудобно — это не значит, что надо что-то менять. Может быть, пользователям именно так и нравится! Для начала определитесь с целевой аудиторией (ЦА), а потом стройте предположения и обсуждайте их с командой.

На что обращать внимание при тестировании usability?

1. На скорость обучения.
2. На скорость использования.
3. На количество пользовательских ошибок.
4. На субъективную удовлетворённость...

Важно: на удовлетворенность не вашу, а целевой аудитории!

Если я скачала новую игрушку на смартфон, но не понимаю, как в нее играть, — я долго пытаться не буду, плюну и закрою. Других полно. Поэтому в игры встраивают обучающие режимы: «Тапни туда, нажми сюда», чтобы показать все возможности программы.

Такие режимы не только в игрушках есть! Программа для рисования, аудиоблокнот... Да любое приложение можно снабдить обучением — лишним не будет, а кучу вопросов снимет!

Увеличивайте скорость обучения
за счёт подсказок

Скорость использования — вспомните, как заходили в последний раз в банк за чем-то. Вам нужна простая операция — скажем, подписать досрочное погашение ипотеки или заплатить через кассу за ЖКХ.

Да-да, я знаю, что сейчас такие операции делаются онлайн. Во время написания книги тоже делались! И все же те, у кого нет Интернета или кто не умеет им пользоваться (бабушки), идут ножками в банк платить.

И вот сижу я в банке перед операционистом. Казалось бы, делов на две минуты. Но он что-то ищет, вбивает, ждет... В общем, минут 10 я там сижу. Медлительный операционист? Нет! Виновата программа. Потому что в ней надо сделать кучу телодвижений для простой операции, да еще и подвисает она на каждом этапе, вот все и ждут...

Ну и, конечно, если можно избежать пользовательских ошибок — сделайте это! В том же банке самый распространенный кейс — заполнить ФИО, паспорт, адрес клиента. Как часто ошибаются в написании ФИО или адреса! А ведь можно подключить подсказки и избежать 90% ошибок. Конечно, останутся редкие ФИО, но не будет имен типа «Блия» (Юлия).

А для тестирования субъективной удовлетворенности просто сообщайте команде обо всем, что вам не нравится. Не факт, что это исправят, ведь вы можете не знать потребности целевой аудитории. И все-таки у тестировщика обычно есть опыт, который поможет сделать программу лучше.

В любом случае обязательно запишите все свои комментарии по использованию продукта, если вы в команде новичок. Остальные уже привыкли к интерфейсу и функционалу, для них все удобно. А вот вы видите

это впервые и можете реально оценить скорость обучения и использования. Грамотный менеджер всегда расспрашивает новичка обо всех первых впечатлениях, ведь это кладезь информации!

Коридорный тест

Дешевый способ провести юзабилити-тестирование вашего продукта — коридорный тест.

Если хотим знать первое впечатление незнакомого с продуктом человека:

1. Ловим в коридоре коллегу, который не знает ваш продукт.
2. Просим его оценить интерфейс — вот что первое в голову придет, то пусть и расскажет.

Если интересует скорость обучения:

3. Снова ловим в коридоре коллегу.
4. Просим выполнить какое-то действие/задачу.
5. Внимательно следим, куда он при этом будет тыкать

Литература для чтения

Для того чтобы ставить баги по юзабилити, надо сначала ознакомиться с литературой. Потому что иначе это будет 100500 «багов», поставленных с обоснованием: «Это лучше для пользователя, юзабилити баг же!».

Типичная проекция: «Мне неудобно = всем неудобно». Вот чтобы лучше понимать про удобство, прочтите:

- Алан Купер, «Психбольница в руках пациентов» — лучшая книга про usability. Стоит читать даже тем, кто этим видом тестирования пока не занимался.
- Дэвид Платт, «Софт — отстой! И что с этим делать?» — офигительная книжка по usability. Ничуть не хуже «Психбольницы», при этом в два раза веселее ;-)
- Стив Круг, «Не заставляйте меня думать» — шикарная книга, читается легко и быстро!

Ссылки на эти и другие книги см в моем блог-посте «Список книг (по тестированию и не только) с отзывами»[134].

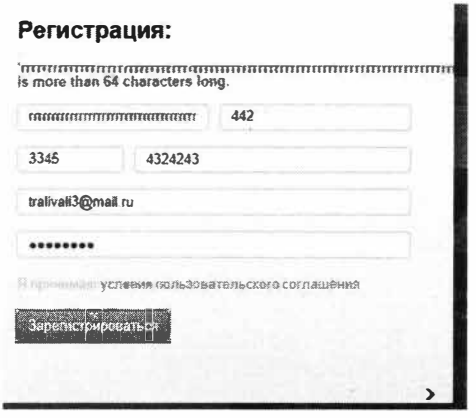
Тестирование GUI

GUI — Graphical User Interface, графический пользовательский интерфейс. Это то, с чем работает обычный пользователь: открыл сайт и тык-тык по кнопочкам.

Тестирование GUI — это проверка того, что интерфейс выглядит как задумано. Иногда это означает выверку по макетам из ТЗ. И даже если видишь сдвиг на один пиксел — заводишь баг. Но чаще всего это означает просто проверить, что все кнопочки нажимаются, текст за границы нигде не вылезает и других косяков нет. Баги верстки в вебе, баги наложения текста в мобилках.

Если у вас не стоит задачи выверять расположение каждого пиксела, то отдельно тестирование GUI не проводится. Вы проверяете функционал, обращая при этом внимание на интерфейс. И всё. Баги верстки вы и так заметите, по крайней мере, должны.

Этим человек выгодно отличается от робота. Робот проверит ровно то, что ему сказали. Сказали «тыкнуть „войти“ и ввести такие-то данные», он и тыкнет. И пофигу будет роботу, что формочку всю перекосило, свою задачу-то он выполнил! А человек заметит косяк и поставит баг.

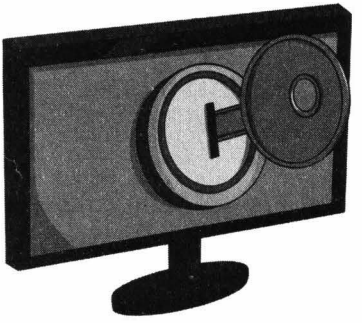


Вот, например, проверяем: «А что будет, если ввести длиииииииинное ФИО при регистрации?». Нам выдается абсолютно корректное сообщение об ошибке, вот только оно слегка выехало за пределы поля[135]...

Человек такую проблему сразу заметит. Робот — нет. Он проверит текст ошибки. Текст корректный? Тест пройден!

Тестирование безопасности/защищенности

Тестирование безопасности — это отдельная область тестирования. О которой я почти ничего не знаю ;-(Потому что область сложная. И если юзабилити, в принципе, может проверить даже джуниор, то в тестирование безопасности ему лучше не лезть. Потому что когда безопасность важна — пропущенный баг стоит миллионы.



Насколько безопасно ваше ПО? Легко ли его взломать? Это очень важный вопрос, если приложение работает с персональными данными или деньгами.

Периодически всплывают сайты из серии «Введи свой емейл, и мы скажем твой пароль, ведь мы взломали большую базу, аха-ха». Если ваш пароль и правда взломали — значит, злоумышленник обнаружил дыру в безопасности.

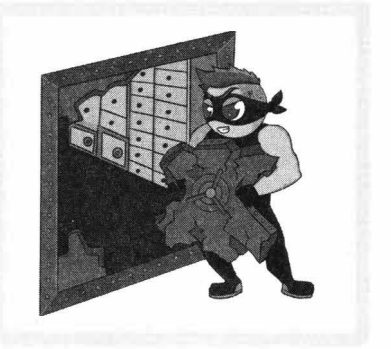
Если он найдет дыру в работе банкомата, то сможет снять оттуда все деньги при нулевом или минимальном балансе на карточке.

Если он найдет дыру в веб-приложении, то сможет войти под вашим логином/паролем. И если вы сохранили данные карточки, злоумышленник может их считать, купить что-то или просто вывести деньги. Поэтому банки сейчас ограждаются от покупок добавлением двухфакторной авторизации: вы делаете покупку на сайте и подтверждаете ее кодом из СМС.

Рассмотрим, какие бывают способы взлома.

Брутфорсинг

Brute force — метод «грубой силы»



Полный перебор (или метод «грубой силы», от англ. brute force) — это когда мы пытаемся подобрать данные, перебрав все возможные варианты.

Не секрет, что админы в линукс-системе часто заходят под логином root. Это по умолчанию, а изменять так лень... Так что логин злоумышленнику известен. Остается подобрать пароль.

Метод брутфорсинга — это тупо перепробовать все возможные комбинации. И чем умнее и быстрее становятся компьютеры, тем быстрее они перебирают варианты.

Думаете, что это все фигня и от этого давно есть защита? Пожалуйста, недавняя статья[136]: **https://www.banki.ru/news/lenta/?id=10894280**.

> В результате выявления уязвимости веб-ресурса ликвидированного Бинбанка персональные данные граждан, направляемые кредитной организации при запросе на выдачу кредита или карты, оказались в открытом доступе, сообщает «Коммерсант».
>
> В понедельник компания DeviceLock сообщила о выявленной утечке данных клиентов-физлиц, подававших заявки на получение кредитной карты Бинбанка «Элixir», которые не находятся в открытом доступе, однако извлечь их можно методом подбора буквенно-цифровых сочетаний, поясняется в сообщении компании.
>
> По словам основателя DeviceLock Ашота Оганесяна, в настоящее время известно, что уже найдено более 1 тыс. анкет, однако при помощи автоматизированного перебора возможно получить все заявки, а их могут быть десятки или даже сотни тысяч. Также велика вероятность подобных проблем и в других банках, использовавших то же решение для системы безопасности.
>
> Как напомнил вице-президент группы компаний InfoWatch Рустем Хайретдинов, схожая публичная история была в 2015 году с банком «Санкт-Петербург», когда злоумышленники также получили доступ к данным клиентов, после чего банк пересмотрел свой взгляд на безопасность. Руководитель группы исследований безопасности банковских систем Positive Technologies Ярослав Бабин сообщил, что очевидно также отсутствие защиты от атаки перебором.

Если говорить про линукс-сервер, то их пытаются взломать перебором IP-адресов. Ведь для IP-адресов есть какая-то определенная маска[137], и почти

наверняка там будет логин root. А уж если IP известен, рут обязательно попробуют. Хотя бы «ламерским брутфорсом» — перебором типичных комбинаций типа:

- root / root;
- root / admin;
- ...

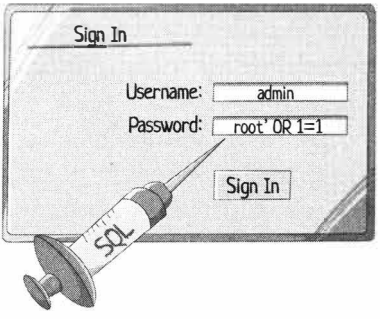
Ну а что, вполне может сработать, если админ не удосужился поставить для рута нормальные логин/пароль.

Перехват трафика

Вы отправляете какую-то информацию. Например, вводите в формочку входа свой логин и пароль. Эти данные клиент (браузер) передает серверу. Злоумышленник влезает на пути передачи данных и перехватывает сообщение. Так он получает данные для входа.

И всё, если у вас в приложении привязана карточка, злоумышленник получает к ней доступ. А если приложение открыто отправляет «денежные» данные, то это совсем фейл. Ведь сейчас есть куча платежных систем, которые обеспечивают безопасность. Если вы делаете свой сайт, подключите готовую систему, не делайте самописную на коленке — ее будет слишком легко взломать.

SQL-инъекции



Это когда вы передаете вместо нормального параметра SQL-запрос. Если приложение уязвимо, можно сделать много плохого.

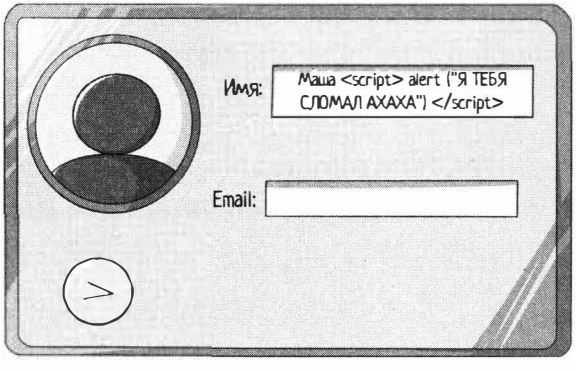
Самая типичная инъекция — ввести в поле одинарную кавычку и за ней какой-то запрос. Вот как на рисунке. Логин админа-то обычно стандартный: admin или administrator, а в пароле пишем любой пароль, а потом «ИЛИ 1=1». Пароль не подошел, но 1 = 1, возвращается true — и вот мы уже вошли в систему!

XSS-атаки

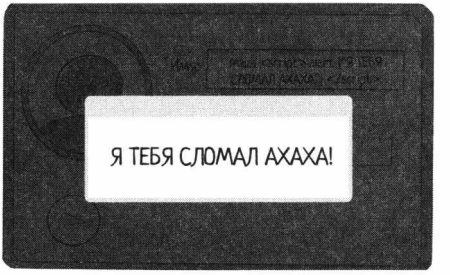
Межсайтовый скриптинг — это когда вы внедряете в код страницы вредоносный код. Например, у вас есть поле ввода: имя в профиле. Разработчик не защитил это поле и позволил вводить все что угодно.

И вот я могу туда ввести:

```
Маша<script>alert("Я ТЕБЯ СЛОМАЛ АХАХА")</script>
```



Что в итоге? Как только я сохраню изменения, то увижу поп-ап с текстом «Я ТЕБЯ СЛОМАЛ АХАХА».



Если мы увидели поп-ап — поле не защищено. А вместо безвредного поп-апа можно внедрить правда плохой скрипт. Который утащит информацию, перенаправит реального пользователя на другой опасный сайт или сделает что-то еще.

Поэтому можете использовать лайт-проверку из приведенного примера с алертом. Если алерт вызвался — бейте тревогу. Ну а как можно навредить такими атаками — изучите в гугле ;-)

Что-то более умное

Так как я не умею взламывать, то и научить этому не могу. Про простые атаки знаю, но наверняка есть что-то более сложное и изощренное. Если интересует эта область — идите в тестирование безопасности, копайте и развивайтесь.

Хороший тестировщик безопасности получает большую зарплату. Рынок, правда, невелик. По крайней мере, требуются такие люди реже, чем простые тестировщики или автоматизаторы. Но хорошего с руками оторвут.

Тестирование локализации

В нашем случае под локализацией понимается не локализация бага, а локализация ПО — перевод на разные языки.

Это когда вы можете переключать язык в интерфейсе — обычно «Рус/Eng», но бывают и многоязычные приложения! Что тестировать в таком случае?

1. **Адекватность перевода** — при наличии знания языков. Скорее всего, тексты лежат где-то в одном месте в коде, файлики с русскими фразами, английскими и на других языках. Можно вычитать сами файлики.
2. **Пункты меню** — все ли переведены?
3. **Кнопки** — про них часто забывают. В итоге сайт весь на английском и большая кнопка «Купить».
4. **Рисунки** — про них обычно даже не думают. Переключаешь язык, а там картинка на главной с другим языком осталась...

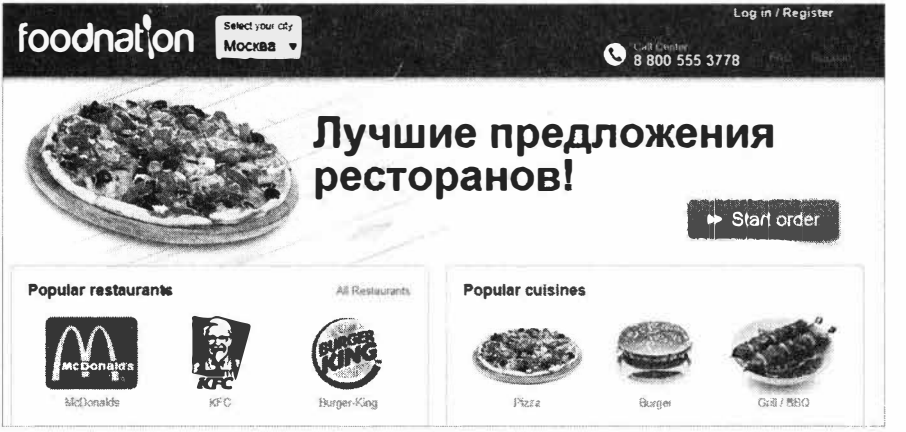
5. **Длинные фразы** — часто любят вылезать за пределы области. Что в одном языке нормально влезает, в другом займет много места. Например, «Поздравляем!» и «Congratulations!». Найти такие фразы помогут файлики с переводом в коде.

Из собственного опыта...

Вот яркий пример того, как на сайте foodnation в 2013 году забыли про картинку на главной странице сайта [138].

Переключаешься на английский:

- кнопочка «Login / Register» обновилась;
- названия блюд и ресторанов тоже;
- а вот огромная картинка «Лучшие предложения ресторанов!» гордо висит на русском. С английском кнопкой «начать покупки» ;-)



Иногда бывает такое, что приложение вообще «крашится» при выборе другого языка. У меня такое было в игрушке Broken Sword 1[139].Суть в чем — нужно расшифровать записку. Если открыть ее на русском языке, приложение вылетит. Если на английском — все хорошо.

Возможно, при открытии записки программа идет в определенную папку в своей памяти и ищет там файлик «записка.язык». На всех языках она есть, а русский файлик забыли сделать. Но программа-то его ожидает! Поэтому происходит сбой при попытке обращения.

Так что учтите, ошибки локализации — это не всегда просто картинка с русским текстом среди английского языка!

Тестирование локализации: собранные грабли

Это не мои примеры, я взяла текст у Светланы Скуратовской[140]. Историю свою Света написала, посетив митап по тестированию в своем городе.

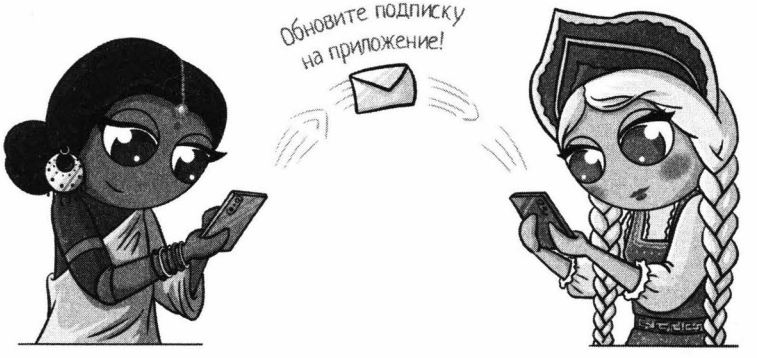
И это дополнение к тому, что рассказывала на митапе Саша Ковалева. Дальше со слов автора:

Саша рассказывала про локализацию, и я вспомнила свои проекты...

Первый раз я столкнулась с переводом сайта в 2008 году. Небольшой стартап каким-то чудом получил заказ на локализацию сайта одного автомобильного гиганта. «Фиксед прайс», что подразумевало сжатые сроки. Мы наступили на все возможные грабли. Я узнала очень много нового про кодировки, сортировку, файлы ресурсов и так далее... Хорошо что там были только европейские языки. Плохо что был немецкий ;-)

После этого проекта я с легкостью справилась с адаптацией одного банковского продукта под наш рынок: перевод «в одну сторону» и тестирование. Казалось, я знаю про переводы всё.

Но тут у меня появился проект с мобильным приложением. Кроме ограниченного размера экрана и различных вариаций календарей и денежных разделителей, я столкнулась с тем, что письма и СМС пользователь тоже хочет получать на «своём» языке. А про такую возможность никто заранее не подумал.



А не очень давно мне нужно было тестировать сайт с локализацией на пяти языках. И я узнала как много зависит от разработчиков :-) Оказывается, при криво написанной смене языка часть адресной строки может задваиваться, и тогда пользователь попадает на страницу 404. И при переходах по сайту тоже. Но не всегда :-). Оказывается, возврат к предыдущей странице — абсолютно не банальная штука :-). Оказывается, один лишний символ в файле ресурсов может сделать нечитаемым для программы весь файл. Слишком много откровений для одного проекта.

Но поскольку тестировщик не всегда знает, кто же разрабатывает то, что приходит в тестирование, я хочу дополнить список того, что важно протестировать при локализации:

- все взаимодействия с пользователем (письма, нотификации, СМС);
- взаимодействия с оборудованием (принтер, сканер, экспорт куда-то);
- переходы по страницам и обратно (!) со сменой языка;
- правильно ли выбирается дефолтный язык.

Тестирование совместимости

Что может повлиять на работу приложения?

- разные ОС (Windows, Linux, Mac);
- разное железо (видеокарта, процессор и т. д.);
- разные браузеры (Chrome, Firefox, Opera Mobile, Safari, IE);
- разный сторонний софт (в браузере могут мешать плагины, на самом компе — Касперский или другое ПО, которое, например, выжирает память).

Совместимо ли ваше приложение с разными браузерами? А разными операционными системами? Именно в этом заключается тестирование совместимости — проверить и предоставить информацию.

Если веб-приложение работает только в конкретных браузерах, надо писать об этом в ТЗ и сделать страничку old-browser-error для пользователя. Иначе он подумает, что ваш сайт в принципе нерабочий, а не то, что у него браузер устарел.

Разумеется, если ваше приложение ориентировано на большой рынок (интернет-магазин), чем больше браузеров вы поддерживаете, тем больше покупателей у вас будет. С другой стороны, если собрать статистику, можно увидеть, что «из IE приходит 10 пользователей, из них что-то купит один. Выхлоп: 10 рублей. А чинить баги занимает 50 рублей». Отсюда вывод — овчинка выделки не стоит.

Если у вас мобильное приложение — опять же, чем больше комбинаций операционных систем, браузеров, размеров экрана вы поддерживаете, тем больше будет пользователей. А дальше уже считаете, насколько это выгодно.

Если десктопное приложение, оно все равно можем зависеть от операционной системы или установленных программ. Тогда сразу в инструкции по установке прописываете все, что важно:

- если приложение работает только на винде — пишем об этом;
- если поддерживается Linux 7.0, но не поддерживается Linux 6.0 — пишем об этом;
- если нужна именно Java 7, а не 6 или 8 — обязательно пишем об этом!
- Если работаем только на Oracle 11, а Oracle 12 не поддерживается— пишем об этом;
- ...

Особое внимание советую уделить антивирусу. Потому что именно он чаще всего является помехой для вашего ПО. Пропишите заранее в инструкции, что антивирус надо или отключить, или добавить ваш сервер в исключения.

По исполнителям (по субъекту)

По исполнителям тестирование делится на:

- альфа-тестирование — тестирует команда разработки;
- бета-тестирование — тестируют реальные пользователи.

Альфа/бета-тестирование

Альфа-тестирование

Тестирование альфа-версии программы. Альфа-версия — это первая версия. Или не первая, но еще «сырая», на то она и альфа. Может быть, программа вообще не готова целиком, только отдельные кусочки.

Тестирует команда. То есть тестировщики, разработчики, аналитики. Обычный цикл разработки: разработчик сделал, проверил, вроде норм. Тестировщик проверяет внимательно. Аналитик и другие заинтересованные лица смотрят тот функционал, который для них важен.

- Плюсы подхода:
  - ✧ отчеты профессиональные;
  - ✧ баги локализованы;
  - ✧ тест-кейсы понятные.
- Минусы подхода — работая с продуктом каждый день, можно получить замыленный взгляд. Команда может не заметить проблему, которую найдет человек, использующий ПО, а не просто тестирующий.

Альфа/бета-тестирование

Бета-тестирование

Тестирование бета-версии продукта — это почти готовый продукт. Команда разработки уже вроде как все баги нашла и готова выпускать релиз. Но если продукт большой, свежий взгляд со стороны не помешает!

Бета-тестирование часто применяется в играх: «Потрай[141] нового босса первым!».

Игроки не просто бесплатно тестируют, но и проходят отбор в группу бета-тестировщиков. Многие хотят увидеть новый контент раньше остальных.

Плюсы подхода:

- быстрый фидбек[142] от реальных пользователей;
- охват конфигураций.

Фидбек

Если говорить про игры: вот сделали нового босса, потестировали. Вроде нормально. Но насколько сложно его будет победить реальным игрокам? Или, наоборот, насколько легко? А то выпустишь легкого босса с крутым лутом[143], и игровая экономика рушится, если дорогие вещи легко получать.

А со сложным можно и переборщить. Делаешь реально тяжелого босса. Но сколько команд смогут его убить? И сколько времени потратят? А какой процент нужен нам для баланса? Запускаем бета-пользователей и смотрим.

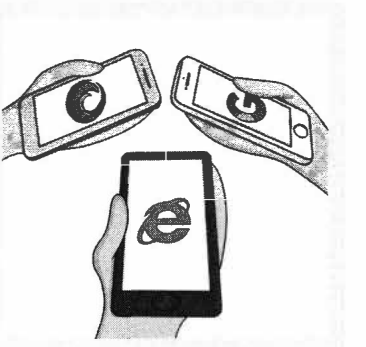
Только учитываем тот факт, что бета-тестировщики — ОЧЕНЬ заинтересованные пользователи, настоящие задроты. И если им босс дался легко — остальным будет сложно. Если же и они его не смогли победить, то 99% не смогут.

Конфигурации

Тестировщики при всем желании не могут проверить вообще все конфигурации. Столько различий!

- Железо компьютера (видеокарта, процессор...).
- Скорость Интернета.
- Разные операционные системы.
- Разные версии браузеров.

- Разные плагины в браузерах.
- Разные мобильные телефоны — снова разные операционные системы.
- Разные размеры экранов.
- Разные приложения, которые фоном выжирают вам всю память.
- ...



Тестирование конфигураций

Поэтому мы делим конфигурации на классы эквивалентности:

- самые крутые — на них хорошо тестировать функциональность, чтобы не отвлекаться на баги;
- самые слабые — потянет ли наше ПО минимальная конфигурация?
- самые распространенные — тут гугл и статистика в помощь.

Всё мы протестировать не можем, да и профита не будет. Много времени займет, багов минимум, а денег потрачено ого-го. Вот тут-то и помогает бета-тестирование! Отдаем пользователям — у каждого своя конфигурация, за час получаем огромный охват.

Если где-то проблема, мы узнаем об этом очень быстро, поэтому чем больше у нас людей принимает участия в тестировании, тем больше охват. Но и тем сложнее анализировать результаты из-за минусов подхода — простые пользователи не умеют оформлять тест-кейсы и баг-репорты.

Тест-кейсы

Если вы хотите посмотреть, насколько легко новичок ориентируется в приложении, — инструкции не нужны. Отдали приложение и ждем фидбек.

Если вы тестируете сложность нового босса в игрушке, то тут тоже все просто: запустили игроков и следим за временем прохождения, количеством попыток и т. д.

Но если вы тестируете какую-то программу и хотите за счет бета-тестирования убедиться, что на разных конфигурациях все хорошо, — нужно, чтобы пользователи выполнили конкретные сценарии. Но пользователи — не тестировщики. Нельзя просто отдать им приложение и надеяться, что они его хорошо протестируют.

Поэтому если вы используете пользователей как тестировщиков, то:

1. Продумайте сценарий тестирования.
2. Напишите подробные тест-кейсы.
3. Отдайте их пользователям.
4. Если возникают вопросы, улучшайте тест-кейсы, чтобы вопросов не возникало.

Ну а дальше уже смотрите по ситуации. Варианты разные:

1. Все проходят одни и те же тесты (проверить на разных конфигурациях).
2. Все проходят разные тесты (тестов очень много, вот и параллелим проведение).
3. Комбинация вариантов 1 и 2 — раздаем всем небольшой набор основных тестов, которые важно проверить + задания на свободное плавание.

Баг-репорты

Как воспроизвести баг? Что именно сломалось? Пользователи этого не пишут! Они не умеют оформлять баг-репорты, поэтому просто сделают скрин и скинут нам, ожидая, что мы из него все сами поймем и тут же исправим. Или пишут «ничего не работает», если не смогли открыть инвентарь. И сиди разбирайся потом, что именно «ничего» у них там не работает.

А если у вас открытый бета-тест и пользователей очень много? Как разгрести поток плохо оформленных репортов? Как понять, где баг, а где не баг? Как собрать дополнительную информацию?

В итоге иногда просто отказываются от бета-тестирования. Или ограничивают количество пользователей. А зачем? Да, пользователи не умеют описывать баги. Но их можно научить:

- вводные лекции;
- инструкции;
- шаблоны оформления;
- собиратели информации (это когда в смартфоне тыкаешь «отправить отчет», и он сам собирает информацию про твой смартфон, операционную систему и т. д. Не надо ничего уточнять потом).

Потратьте немного времени на обучение и написание инструкции — зато потом сэкономите кучу времени при чтении отчетов и попытках воспроизвести чужие баги! И получите больше пользы от проведения бета-тестирования!

По затраченному времени (дымовое тестирование)

Дымовое тестирование (Smoke/Sanity/Acceptance-тесты) — это когда мы проверяем, что в целом всё работает. Быстро-быстро пробегаем основной функционал, не углубляясь в каждую функцию.

Представьте, что вы пожарный. Ваша задача — войти в горящее здание и вынести всех людей. Ну, может, животных еще, хотя приоритетность тут явно ниже. У вас нет задачи спасти телевизор, ноутбук или другие милые сердцу вещи. Только самое главное.

Допустим, у нас можно загружать файлики, строить по ним отчеты. Все это доступно для зарегистрированного пользователя. Смоук-тестом пробегаемся по функционалу:

1. Приложение стартовало.
2. Все ссылки рабочие (не битые).
3. Регистрация работает.
4. Отчет загружается.
5. Файлики грузятся.

Smoke / Sanity
Дымовое тестирование

Без копаний, по первому заходу. Мы проверяем, что отчет в принципе загружается, а не тестируем десятки возможных комбинаций загрузки. Мы проверяем, что файлики в принципе грузятся, а не все классы эквивалентности перебираем.

Различия санитарного тестирования (Sanity), дымового (Smoke) и приемочного (Acceptance)

Санитарное и дымовое тестирование в одних источниках считают синонимами. В других — разделяют по «векторам движения»:

- Дымовое тестирование (Smoke testing) направлено вширь для покрытия тестами как можно большего функционала в кратчайшие сроки.
- Санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, какого-то одного модуля или сценария.

То есть если в смоуке мы просто проверили, что регистрация работает (через мейл, через соц. сети), то в санити проверяем несколько разных мейлов по классам эквивалентности, разные учетки соц. сетей, делаем негативные проверки.

На практике обычно нужно именно «смоук» — дымовое тестирование. Пробежались по всему функционалу. В целом работает? Ну и окей.

- Приёмочное (acceptance) тестирование обычно проводится тестировщиками при передаче новой фичи[144] на тестирование или заказчиком на показе продукта.

В обоих случаях сценарий такой:

- вам приносят совершенно новый для вас набор функций. У вас есть БАААААЛЬШОЙ чек-лист, но сначала вы делаете пару проверок, что переданное вам вообще-то работает. Например, фича «Печать отчета». Вы проверяете, что с какими-то рандомными значениями кнопка «Печать» действительно нажимается и что-то происходит;
- с заказчиком аналогично, для него ВСЁ — новые функциональности, и ему дают потыкать продукт, чтобы он убедился, что хотя бы на первый взгляд всё работает и за это нужно/можно платить.

Sanity, smoke & acceptance

Этот пример рассказала моя коллега Анна Хворостьянова.

Часто возникает путаница, потому что в большинстве небюрократизированных компаний для приемочного и смоук-тестирования используется один и тот же набор проверок — кейсов самого высокого приоритета.

В моей практике мы проводили смоук, когда выкатывали релиз на прод. То есть провели регресс, решили, что версию выпускаем, выкатили на прод последнюю сборку, ночью открыли сайт, провели смоук, и если все ок — оставили сайт до утра Если не ок — откатили сборку назад.

Примечание автора: у меня тоже была такая практика! Когда делали гос. проект, релизись ночью, и я прогоняла смоук. Ок? Оставляем. Не ок — откатываем. Все ночью, пока пользователей нет.

Если подытожить всё сказанное, то:

- дымовое тестирование (Smoke testing) проводится при релизе продукта на прод, после регресса и перед тем как новый релиз увидят пользователи;
- санитарное тестирование (Sanity testing) проводится, если в смоуке/регрессе обнаружены какие-то баги (то есть это может быть в любой момент проведено). Сразу скажу: ни разу не слышала, чтобы в русскоязычной среде использовали этот термин;
- приемочное тестирование (Acceptance testing) проводится:
  - ✧ тестировщиком при получении новой фичи от разрабов;
  - ✧ заказчиком на показе.

Тестирование нового функционала

Проверяем лишь новый функционал. В старый НЕ лезем. Точнее, лезем, но *только* если уверены, что он связан с новым.

Типичная рабочая задача: разработчик сделал новый модуль или улучшение к старому. Мы проверяем ровно одну конкретную задачу + связанный с ней функционал. И всё.

У нас просто нет времени после каждого чиха тестировать вообще всё: «а вдруг там что-то сломалось» (а ведь могло, но об этом в следующем разделе — про регресс). Я знаю людей, у которых проверка всего функционала

занимает неделю! Представляете, что было бы, если бы они пытались проверить все после каждого изменения системы? Разработка велась бы годами!

Так что если есть новый функционал — мы тестируем именно его. Не углубляемся во все возможности своего приложения.

Регрессионное тестирование

То, что работало раньше, продолжает работает и сейчас. Именно это и проверяет регрессия, она же регрессионное тестирование, regression testing.

- В релизе — смотрим кусочками отдельный функционал.
- В регрессе — смотрим на всё в целом.

Зачем это нужно? В сложных системах функционал связан между собой. Внося изменения в один модуль, мы можем сломать что-то в другом. В ряде случаев это становится полной неожиданностью как для разработчика, так и для тестировщика. Иногда про связь знают, но забывают проверить в тестировании отдельного кусочка.

В любом случае, если вы готовитесь отдать конечный продукт пользователю, нужно убедиться в том, что он работает. Работает так же, как раньше + добавлены новые функции. Если вдруг в регрессе вы нашли баг в коде, который «вроде не трогали», — значит, вы нашли взаимосвязь модулей. Обязательно узнайте у разработчика причину падения и добавьте проверку в свои чек-листы конечных функций:

- изменяя ручную корректировку, проверить отчеты А, Б, В;
- изменив имя в профиле, поучавствовать в акции и проверить, какое имя придет в письме!
- исправив адрес ИП в I квартале, проверить, как распределится годовой доход между налоговыми!
- ...

Очень полезно проводить регрессию свежим взглядом. У нас, например, если делалась большая фича, то:

- тестирует фичу один тестировщик;
- регрессию проводит другой.

Второй тестировщик не проводит полное тестирование фичи, он лишь знакомится с ней, базово проверяя работу. Он может задать пару вопросов «а что если...?», исходя из своих знаний и опыта. И вполне может найти баг, который пропустил первый тестировщик. Такая вот двойная проверка и дополнительная польза.

Но проблема регресса в том, что это каждый раз одно и то же. Скукота! Именно поэтому, когда начинают изучать автоматизацию, начинают с автоматизации регресса. Первый кандидат — стабильный функционал, который скучно тестировать вручную.

По степени автоматизации

По степени автоматизации различают тестирование:

- ручное;
- автоматизированное;
- полуавтоматизированное.

Ручное тестирование

Все делаем руками:

1. Открываем браузер.
2. Открываем сайт.

3. Нажимаем кнопку А.
4. Нажимаем кнопку Б.
5. ...

Такое тестирование еще называют *мануальным*, от слова manual, ручной. И именно с него начинают все тестировщики. Мануальщики:

- пишут тест-кейсы;
- составляют чек-листы;
- исследуют приложение;
- оформляют баги;
- ...

И хотя сейчас даже от новичков иногда требуют автоматизацию (что оооочень странно, но логично — за минимум денег всё и сразу), тестировщик всегда начинает с ручного тестирования.

Можно, конечно, просто автоматизировать чужие тесты. Тогда тестирование знать не надо, выучил программирование и вперед! В некоторых компаниях есть такое разделение:

- мануальщики, которые придумывают тесты;
- автоматизаторы, которые пишут код под эти тесты.

Но обычно ты автоматизируешь те тесты, которые сам же и написал. Значит, перед тем как начать автоматизировать, стоит научиться основам тестирования. По крайней мере, если хочешь быть тестировщиком, а не просто кодером. И в таком случае сначала — ручное тестирование! А потом уже автоматизация.

Не стоит думать, что ручники — это только те, кто бездумно тыкает на кнопочки по чужим тест-кейсам, а хочешь расти — выбирай любое НФТ, например, автоматизацию. Нет, в ручном тестировании тоже можно расти! Исследовательское тестирование, тест-дизайн — этому можно учиться годами.

Не стоит думать, что если вы прошли по туру Уиттакера — то вы царь и бог исследовательского тестирования. Такие туры — это верхушка айсберга, доступная новичкам. А вот полноценное исследовательское тестирование новичок уже не проведет.

Аналогично с тест-дизайном. Применили классы эквивалентности на форме ввода имени и рады? Думаете, что все знаете про тест-дизайн? Это не так. Тест-дизайну учатся годами. Здорово, что вы уже знаете больше стандартного новичка или даже человека «с опытом», который ничему не учился, а просто проходил по чужим тест-кейсам пять лет. Но тест-дизайн — слишком сложная тема, чтобы сказать: «Я книжку прочитал, пару чек-листов написал и теперь все знаю».

Если хотите быть ручным тестировщиком — будьте им! Не обязательно автоматизировать, чтобы быть востребованным на рынке труда. Грамотный тест-дизайнер и получает хорошо, и ценится сильно.

Но в любом случае начинают все тестировщики именно с ручного тестирования. Потому что без него нет и автоматизации.

Автоматизированное тестирование

Всё делает робот:

1. Открывает браузер.
2. Открывает сайт.
3. Нажимает кнопку А.
4. Нажимает кнопку Б.
5. ...

Тестировщик отдыхает ;-)

Специально обученный человек (автоматизатор) один раз написал код, а потом он выполняется самостоятельно. Ну разве что что-то изменится в приложении, и придется код обновить, а так — робот работает сам.

Очень удобно! И в автоматизацию тестировщик вполне может развиваться, когда освоит ручное тестирование хотя бы на базовом уровне.

Автоматизированное тестирование

ПолуАвтоматизированное тестирование

Часть делает робот:

1. Открывает браузер.
2. Заполняет 100500 полей регистрации.

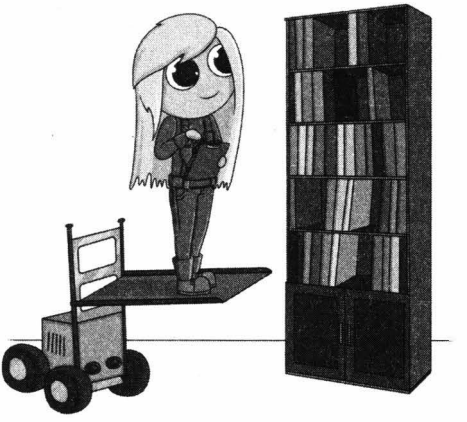
Часть — тестировщик:

3. Открывает отчет.
4. Нажимает кнопку А.
5. Нажимает кнопку Б.

Робот избавил от рутины, а дальше — вручную.

Так как это не полностью автоматизированные тесты, то вам не надо обладать сильными познаниями в программировании, чтобы создать «помощника». Можно использовать функционал «record and play», который есть почти в любом инструменте. Устарело? Выкинули и записали по новой. Никаких проблем.

Я в свое время написала робота для избавления от рутины. В моей системе, для того чтобы начать тестирование, нужно было создать объект, заполнив 20 полей. Один раз заполнил, два, три, четыре... А потом это ужасная рутина!



Я в то время изучала программирование и робота написала на Watin[145]. Да-да, про плагины, помогающие автоматически заполнять поля[146], я в то время не знала, а автоматизацию все равно изучала.

Так как в автоматизации я была новичком, времени убила много. Даже думала, что зря. Но это окупилось! Потом я запускала робота, он быстренько заполнял все поля, и я могла приступать непосредственно к тестированию нужной мне функции. Раз за разом глядя, как «оно работает само», я очень радовалась тому, что все-таки это сделала!

Автоматизация рутины — очень важная штука. И она может быть простой! Так что обсудим ее на уровне новичка в следующей главе. А пока вернемся к классификации тестирования.

По состоянию системы

Статическое тестирование (static testing)

Система НЕ запущена. Что же тогда сюда входит?

- ○ Тестирование документации.
- ○ Ревью кода.

Статически можно тестировать как черный, так и белый ящики:

- ○ Black box — тестирование документации;
- ○ White box — ревью кода.

Только учтите, что в статическом тестировании мы просто читаем сам код! А не подсматриваем в него, работая с действующим приложением, — это уже будет динамическое тестирование.

Динамическое тестирование (dynamic testing)

Система запущена. Так что сюда входят все действия внутри системы:

- регистрация / авторизация;
- загрузка файла;
- создание карточки объекта;
- любое другое воздействие на систему;
- работа с любой функциональностью.

Динамическое тестирование — основной вид взаимодействия с системой. Большая часть времени уходит именно на него, ведь практически любой тест требует запущенной системы. А тестирование ТЗ занимает небольшую часть нашего времени!

По формальности

По формальности тестирование делится на:
- тестирование по готовым тестам;
- исследовательское тестирование.

Тестирование по готовым тестам

Подготовили заранее тесты:
- тест-кейсы;
- чек-листы;
- что-то среднее.

Вот четко по ним и идем.

Плюсы подхода — даже новичку легко выполнить тесты. Он четко знает:
- что проверять;
- какой должен быть ожидаемый результат.

Минусы подхода:
- эффект пестицида — может, в функции есть баг, но вы его не найдете, если каждый раз будете вводить одни и те же данные;
- тесты устаревают — особенно тест-кейсы! Пытаешься выполнить, а такой кнопки или даже функции уже нет, и что делать? Приходится искать тех, кто знает систему лучше тебя.

Исследовательское тестирование

Идем и изучаем систему. Без тест-кейсов и чек-листов.

Исследования проводятся в виде туров. Ставишь цель на тур, засекаешь время (он обязательно ограничен по времени!), и вперед! Максимум документации — небольшой чек-лист «на что обратить внимание в туре». Сюда же относится описание туров Уиттакера — это памятка «что тестировать», но не конкретный чек-лист.

Чтобы грамотно провести исследовательское тестирование, нужен опыт. Опытный тестировщик знает, на что обращать внимание. Он умеет НЕ обращать внимание на все подряд — это тоже важно в исследовательском туре. Нашли параллельно мелкий баг? Записали мысль и пошли дальше. Потом к ней вернетесь. Тут важно записать так, чтобы потом легко вспомнить, а не оформлять по всем правилам — на это времени просто нет.

Если начинающему тестировщику предложить поисследовать продукт, он будет сидеть перед ним с широко раскрытыми глазами: «А что делать-то надо??».

Но для начинающих есть лайт-версия. Это исследовательские туры Уиттакера. Мы о них говорили в *главе 6*. Хотите стать еще круче? Тогда вам гуглить. Учтите, что про исследовательское тестирование пишут обычно англоязычные гуру. От себя могу порекомендовать книгу: James A. Whittaker, «Exploratory Software testing».

- **Плюсы подхода:**
  - можно найти очень много багов;
  - нет эффекта пестицида, потому что нет четко прописанных проверок, а каждый выполняет тур по-своему. Да даже один человек два раза проверит чуть иначе;
  - нет устаревания тестовой документации, потому что этой документации нет.
- **Минусы подхода** — новичку сложно:
  - отвлекается от тура;
  - не знает, какой ожидаемый результат должен быть;
  - не знает, что вообще проверять стоит — в итоге просто потратит время зря.

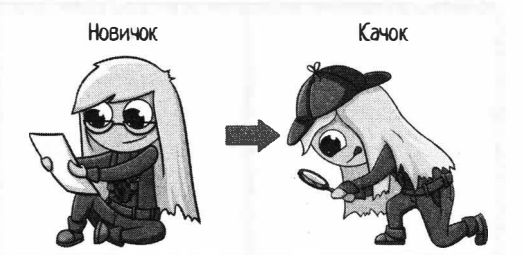
Исследовательское тестирование не используется для:

- проверки нового функционала (там надо как раз составить тесты и идти по ним);

- проверки самого главного во время smoke — тут мы тоже идем по чек-листам.

Исследования очень хороши, но это не единственный вид тестирования!

Сравним подходы



По тестам / Исследовательское

Провести тестирование по готовым кейсам может любой. Хоть новичок, хоть «качок». Это — самый простой вид тестирования.

Пройтись по туру Уиттакера тоже может даже новичок, но провести исследование не по туру он полноценно не сможет.

Поэтому можно представить переход от «пройтись по готовому» до «продуктивно работать без чек-листа» как этап развития тестировщика.

Когда какое тестирование выбрать?

По тестам — когда есть очень важный или сложный модуль

Например, вы тестируете элемент из системы автопилота самолета — пропустите малейшую недоработку, и могут пострадать люди. Или

если вы тестируете кардиостимулятор сердца. Ошибок быть не может. Даже если вы систему знаете как свои пять пальцев — все равно пропишите тесты и идите по ним. Потому что забытая проверка может обернуться чьей-то жизнью.

Конечно, вы вряд ли будете тестировать такое ПО. Скорее всего, это будет простая программа. Про развлечения, помощь в делах, но не про жизнь. Там тесты не нужны? Нужны!

У вас все равно может быть очень важный функционал. На котором завязаны интеграции, например. Если что-то изменится, сломаются десятки систем. Или работа с денежным модулем — деньги тоже не хочется терять, согласитесь! Или просто заказчик всегда проверяет именно этот кусок как самый важный. И он должен работать. Всегда.

Также функционал может быть сложным. Чтобы в отчете были все данные, нужно зайти в 10 разных мест и там всё заполнить. Чтобы не пропустить какой-то шаг и не вспоминать мучительно «как это сделать» — лучше записать.

Исследовательское — когда замылился глаз

Смотрите на систему в 100-й раз и уже не знаете, что проверить? Пройдитесь по исследовательскому туру, освежите свой взгляд!

Подведем итоги

Существуют различные:

- виды тестирования;
- типы тестирования;
- форматы тестирования.

Чтобы вы не путались, распишем все это еще раз.

- Различные виды тестирования:
  - ✧ функциональное тестирование (Functional testing);
  - ✧ нефункциональное тестирование (НФТ), которое делится на:
- тестирование производительности (Performance testing);
- нагрузочное тестирование (Load testing);
- стресс-тестирование (Stress testing);
- тестирование надёжности (Reliability testing) = тестирование стабильности (Stability testing);
- тестирование удобства использования (Usability testing);
- тестирование графического интерфейса (GUI testing);
- тестирование безопасности (Security testing);
- тестирование локализации (Localization testing);
- тестирование совместимости (Compatibility testing).
- Различные типы тестов:
  - ✧ позитивные тесты;
  - ✧ негативные тесты.

Могут быть как функциональные, так и нефункциональные.

- Различные форматы тестирования:
  - ✧ скриптовое тестирование (по тестам);
  - ✧ исследовательское тестирование (свободным поиском);
  - ✧ автоматизированное тестирование.

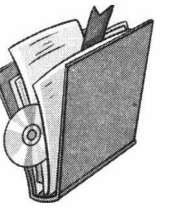
Если вдруг на собеседовании вас спросили о какой-то другой классификации — не бойтесь. Попросите уточнить, о чем речь. Ведь один и тот же тип тестирования могут называть по-разному. И делить на разные типы и виды — тоже. Ведь не это же главное, правда? Так что без паники! ;-)

Домашнее задание

Вопросы для самопроверки

1. Зачем нужна классификация?
2. Какие виды классификации вы можете назвать?
3. Чем белый ящик отличается от серого?
4. Можно ли проводить тестирование белого ящика и статическое тестирование одновременно? А динамическое? Приведите примеры тестов.
5. Чем тестирование производительности отличается от нагрузочного тестирования?
6. Чем sanity testing отличается от smoke testing?
7. Какие минусы у бета-тестирования?

Портфолио



Продолжаем делать портфолио. Подумайте, какие функциональные и нефункциональные тесты вы можете провести для своего приложения. Сделайте табличку и выпишите туда по паре примеров тестов для каждого вида тестирования. Не забывайте делать как позитивные, так и негативные тесты!

Также укажите в табличке различные форматы тестирования:

- скриптовое тестирование (по тестам): для каких (конкретно) частей продукта и в каких случаях будет полезнее всего?
- исследовательское тестирование (свободным поиском): для каких частей продукта и в каких случаях будет полезнее всего?
- автоматизированное тестирование: для проверки какого функционала наиболее полезно? Почему именно для него?

Ответы на вопросы для самопроверки

1. Зачем нужна классификация?

Чтобы упорядочить свои знания. Чтобы определить «белые пятна», куда стоит копнуть.

2. Какие виды классификации вы можете назвать?

По знанию системы, по позитивности тестов, по объекту, по субъекту тестирования, по затраченному времени, по степени автоматизации, по состоянию системы, по формальности.

3. Чем белый ящик отличается от серого?

Серый — смесь черного и белого. Вы смотрите в код, но также знаете и применяете знания о том, как этим всем будут реально пользоваться.

4. Можно ли проводить тестирование белого ящика и статическое тестирование одновременно? А динамическое? Приведите примеры тестов.

Можно, конечно!

- ✧ Статическое — ревью кода.
- ✧ Динамическое — отладка автотестов или кода. Проверка интерфейса, но глядя в код.

5. Чем тестирование производительности отличается от нагрузочного тестирования?

- ✧ Производительность — как быстро приложение откликается.
- ✧ Нагрузка — столько посетителей выдержит.

6. Чем sanity testing отличаетсяот smoke testing?

- ✧ Санитарное тестирование (sanity testing) направлено вглубь проверяемой функции.
- ✧ Дымовое тестирование (smoke testing) направлено вширь для покрытия тестами как можно большего функционала в кратчайшие сроки.

7. Какие минусы у бета-тестирования?

Бета-тестировщики не учились тестировать — они не смогут проверить приложение на важные функции без ваших чек-листов. Они не умеют грамотно оформлять баги и будут норовить прислать вам «Все плохо. Все сломалось. Ничего не работает». Но это все поправимо!

Глава 10
АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

Робот работает, человек отдыхает — ну какая красота! Давайте узнаем подробнее об этом интригующем слове: «автоматизация»!

В этой главе:

- обсудим пирамиду автоматизации;
- исследуем области ее применения;
- познакомимся с Pairwise;
- посмотрим на инструменты полуавтоматизации.

Чего мы в этой главе делать НЕ будем

О, какой соблазн! Купил книжку по тестированию, в которой одна глава посвящена автоматизации. Прочитал ее, и такой сразу красавчик, умеешь автоматизировать! И не надо читать толстенные книги по программированию, всего одна глава, красота.

Увы, так не бывает. И нет, я не научу вас автоматизировать в этой главе. Мы только разберемся с терминами и поймем, что это вообще такое. А еще я расскажу про инструменты полуавтоматизации. Но это не то, что в вакансиях имеют в виду под «автоматизацией», увы и ах!

Автоматизация — необязательный навык для получения работы тестировщиком. Да, сейчас ее требуют чуть ли не в вакансиях на junior'а, но на самом деле она идет позже. Вы можете успешно работать, вообще ничего не автоматизируя. Тест-дизайн — тоже важная и нужная область.

Так что давайте разберемся с определениями, а вы потом сами решайте, хотите развиваться в этой области или нет. Если хотите — ищите соответствующую литературу, видео на YouTube, курсы и т. п.

Что такое автоматизация?

Автоматизация — это написание автоматических тестов. Допустим, мы тестируем регистрацию на каком-либо сайте (подойдет практически любой). Как мы это делаем?

1. Открыли браузер.
2. Открыли нужный сайт.
3. Нажали на кнопку «Регистрация».
4. Ввели тестовые данные.
5. Нажали «Зарегистрироваться».
6. Убедились, что регистрация успешна — например, что появилась кнопка с личным кабинетом, а внутри ваше имя и емейл.

Эту процедуру надо повторить *N* раз. Разные имена, пароли, емейлы... Если вы все это делаете сами — это ручное тестирование. Автоматизация — когда это делается автоматически роботом. Один раз написали скрипт, а дальше он сам все проверяет.

Конечно, не все так просто — иначе ручное тестирование было бы не нужно. Робот — тупое существо. Ему что скажешь, то он и будет делать. А значит, для того чтобы попивать кофеек, нужно:

1. Продумать тесты для автоматизации.
2. Расписать их по шагам. ОЧЕНЬ подробно. Вот многие не любят тест-кейсы за их очевидность: «какую кнопочку нажать», а тут именно так и надо.
3. Написать скрипт, который будет этот тест выполнять.
4. Поддерживать автотесты.

Разберемся с каждым пунктом.

Продумать тесты для автоматизации

Не все можно автоматизировать. Не все нужно автоматизировать. Нельзя просто взять чек-лист ручного тестирования и переложить на код.

Бывает, что автоматизировать тот или иной функционал ОЧЕНЬ сложно. Разработка автотеста займет много времени, причем разработчика, а не тестировщика, — это не окупится, быстрее проверить вручную.

Бывает, что чек-листов у вас очень много. Тогда надо выбирать те, которые нужно автоматизировать в первую очередь. По каким принципам выбирать? Сначала самое:

- **важное** — у человека замылен взгляд, а робот каждый раз проверяет все досконально, ему же не скучно;
- **нудное** — у тестировщика прям настроение падает, когда нужно делать такую работу. Например, чтобы проверить, как отображается связь на двух карточках, надо зарегистрироваться, создать одну карточку, другую, а для этого нужно пройти три экрана по 50 полей... Пусть это делает робот!
- **частое** — если постоянно проверять одно и то же, рано или поздно вы пропустите баг прямо у себя под носом. Потому что «ну всегда же работало, чего сейчас ломаться». В итоге проверяешь вполглаза. По закону подлости именно в это время что-то ломается. Робот же всегда честно проверяет все, что ему сказали.

Учтите, что автоматизация помогает там, где есть стабильность. Нет смысла писать автотесты, да еще работающие через пользовательский интерфейс, если интерфейс находится в разработке и постоянно меняется.

В таком случае вы будете приходить на работу утром, а там все тесты красные[147]. Вроде «ахтунг, все плохо», а на самом деле просто кнопку переименовали, и робот залогиниться в систему не может... Приходится актуализировать тесты!

А вот нудный регресс, который не меняется, можно и роботу отдать.

Расписать тесты по шагам

В принципе, если вы сами будете перекладывать описание на код, можно не расписывать на русском подробно. Но если готовите описание для другого человека, указывайте все. Не заставляйте автоматизатора бегать за вами и уточнять: «А как выполнить это действие, куда нажать?».

Разработчик скрипта должен взять ваше описание и сразу понять, что от него нужно. В маленьких командах, где все знают продукт, достаточно чек-листа и краткого описания. Но бывает такое, что есть ручные тестировщики, а есть автоматизаторы. Автоматизаторов подключают на разные проекты, и конкретно ваш они не знают — какой функционал там есть? На какие кнопочки нажимать? Поэтому вы пишете подробные тест-кейсы.

А иногда, прописывая каждый шаг, вы сразу пишете автотесты! Например, в Gherkin[148] тесты пишутся на простом и понятном языке: русском или английском. Составить их может ручной тестировщик или даже аналитик, а потом уже автоматизатор немного подпилит код.

Вот как выглядит тест в Gherkin[149]:

```
# language: ru
@all
Функция: Аутентификация банковской карты
    Банкомат должен спросить у пользователя PIN-код банковской карты
    Банкомат должен выдать предупреждение, если пользователь ввел неправильный PIN-код
    Аутентификация успешна, если пользователь ввел правильный PIN-код
```

```
Предыстория:
    Допустим пользователь вставляет в банкомат банковскую карту
    И банкомат выдает сообщение о необходимости ввода PIN-кода

@correct
Сценарий: Успешная аутентификация
    Если пользователь вводит корректный PIN-код
    То банкомат отображает меню и количество доступных денег на счету

@fail
Сценарий: Некорректная аутентификация
    Если пользователь вводит некорректный PIN-код
    То банкомат выдает сообщение, что введённый PIN-код неверный
```

Все понятно же, да? А робот берет это описание и выполняет автотест!

Еще статья в тему: «Gherkin: говорим с автоматизаторами на одном языке», https://www.a1qa.ru/blog/gherkin-govorim-s-avtomatizatorami-na-odnom-yazyke/.

Написать скрипт

Чтобы это сделать, нужно знать язык программирования. Вы берете описание теста и перекладываете его на код — создаете робота, который выполняет все нужные проверки.

Иногда скрипт написать легко, во многих фреймворках автоматизации есть функция «Record And Play». Нажимаешь на кнопку, выполняешь действия — всё, робот готов. Он сам записал скрипт и будет повторять его при каждом вызове.

Минус в том, что робот не умеет делать проверки, так что он упадет только в том случае, когда совсем все плохо, — например, страница исчезла или кнопка испарилась, на которую ему надо нажать.

И, конечно, такой робот неоптимально пишет код. Он не переиспользует функции, каждая запись пишется

с нуля. В итоге получается много копипасты. А когда у нас много копипасты, это сложно исправлять. Но об этом уже в следующем разделе ;-)

Другой вариант — сразу писать код, без функции «Record And Play». В таком случае нужно продумать архитектуру автотестов. А еще знать язык программирования и потратить больше времени. Зато поддерживать такие тесты будет проще!

Некоторые тесты пишут разработчики, некоторые тестировщики. Это мы обсудим в разделе «Пирамида автоматизации» этой главы.

Поддержка автотестов

Почему прошаренные автоматизаторы не любят скрипты «Record And Play»? Потому что они понимают, как сложны такие тесты в поддержке.

Одно дело, если вы записали пару скриптов, просто чтобы жизнь себе облегчить. Если что-то изменится в системе, перезапишите робота, да и всё.

Но ведь когда тесты так легко писать, возникает соблазн наколбасить кучу тестов! А что, это потом и начальству можно показать как результат работы: «Я за неделю написал 1000 автотестов, покрыл такие-то модули!». Свеженаписанные тесты работают хорошо, но ведь мы их пишем, чтобы они работали долго...

Проходит время, и в системе что-то меняется. Особенно плохо, если на этапе регистрации или авторизации. Кнопку переименовали или передвинули, например. И БАХ, рассыпались вообще все тесты! И надо или все их перезаписать, или пройтись по каждому и поменять упавшую строчку кода. Уныло! Не ради этого мы тесты писали, да? ;-)

Поэтому в коде важно переиспользование. Особенно в коде автотестов! Создаем отдельно функцию регистрации. И если изменилось название кнопки, нам надо исправить это ровно в одном месте — это хорошая структура кода.

Разумеется, идеальной структуры не бывает. Вопрос в том, сколько очевидных граблей вы соберете, а сколько обойдете. Но учтите: чем больше автотестов — тем больше кода. А код нужно поддерживать.

Вышла новая версия инструмента? Что-то в тестах разломалось, потому что «это теперь записывается по-другому». Нужно актуализировать тесты. Чем их больше, тем более трудоемкая задача.

Или бывает, вы ничего не меняли — бах, тесты красные. Сломались. Почему? Потому что во время их написания вы что-то не учли, и вот оно выстрелило. Ведь код автотестов — это тоже код, там тоже могут быть баги. Ложноположительные срабатывания теста — не редкость. И с каждым нужно разбираться.

В итоге бывают компании, где есть отдельно выделенные люди, занимающиеся только поддержкой автотестов. Когда автотестов много и они идут через пользовательский интерфейс, их поддержка сжирает все время.

Так что учтите, автотест — это не просто «написал и забыл», пока он баги не поймал. Иногда баги находятся в самом тесте. И чем больше у вас тестов, тем больше времени будет уходить на их поддержку. А чем хуже они написаны, тем сильнее растет кривая потраченного впустую времени.

Когда автотест начнет ловить баги?

Вы не поверите! Сразу же ;-)

На самом деле максимум багов за один раз можно найти... Пока пишешь автотесты! Вы смотрите на собственную систему абсолютно новыми глазами — это я вам по своему опыту говорю. И находите баги в тех тестах, которые выполняли 100500 раз.

Из собственного опыта...

Когда я начала изучать автоматизацию, то начала с простейших тестов. Тех, которые выполняла уже много раз. Создание, редактирование базовых сущностей...

Но теперь я медленно выполняла каждый шаг — ведь его надо было выполнить вручную и записать в виде скрипта:

- так, заполнили все поля;
- нажали «сохранить»;
- открыли карточку;
- начинаем выверять каждое поле (что оно реально сохранилось).

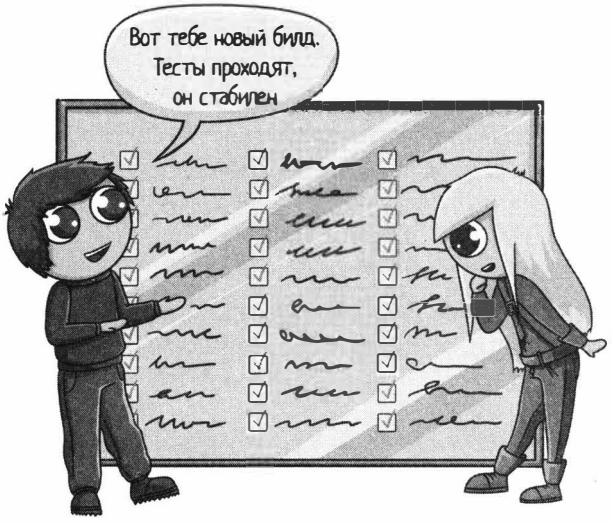
И пока я писала автотесты, нашла кучу ошибок! Потому что обращала внимание на все подряд, размышляя, должен ли это проверять робот. Там сообщение об ошибке вылезло, хотя поле заполнено верно. Тут вылезло сообщение с неправильным текстом, там что-то еще случилось...

И ведь, главное, я совсем недавно это все тестировала! Ну откуда тут взяться багам, казалось бы?? Так вот, написание робота — тоже способ снять эффект «замыленного глаза» и заметить то, что вы в упор не видите!

Главная ошибка начинающего автоматизатора — он думает, что автотесты должны постоянно падать, ведь это отлавливает изменения и баги! И поэтому можно и нужно писать их на изменяющийся функционал. Но это не так.

Автотесты помогают увидеть баги во время их написания, потому что вы внимательно выполняете те шаги, которые привыкли пробегать полуглядя. Так что автотесты должны ломаться редко. Ведь мы что ожидаем? Что разработчик, делая новую фичу, учтет все зависимости и не сломает то, что работало ранее. Иногда все же ломает — вот тут автотесты и падают, показывая на проблему.

В идеале автотесты работают быстро (и поэтому это НЕ тесты через графический интерфейс), и разработчик может прогонять их локально, на своей машине. До того как отдать версию в тестирование. Тогда он может сделать модуль, прогнать все тесты и поправить то, что упало. А потом отдать «хороший» билд на тестирование.



Таким образом, тесты редко должны «краснеть». Если они падают часто, нужно разбираться, почему:

- разработчик не знает продукт — может, он джуниор? Пусть учится у старших товарищей;
- требования постоянно меняются, автотест устаревает — возможно, стоит временно удалить этот тест или «заморозить», пока требования не устаканятся;
- плохая архитектура кода: меняешь одно, ломается все что угодно — пора выделить время на рефакторинг (когда мы переписываем код, не меняя логику, просто делая его проще, читабельнее, красивее...);
- ...

Узнаём, что не так — устраняем причину. Иначе поддержка автотестов окажется слишком дорогой, и это станет невыгодно.

А мы поговорим о том, кто какие тесты поддерживает. Чтобы это понимать, нужно знать про пирамиду автоматизации!

Пирамида автоматизации

Почему именно пирамида? Потому что как пирамида Маслоу[150] ;-) Только вместо потребностей человека — потребности программы в том или ином количестве тестов.

Давайте разберемся с каждой частью этой пирамиды отдельно.

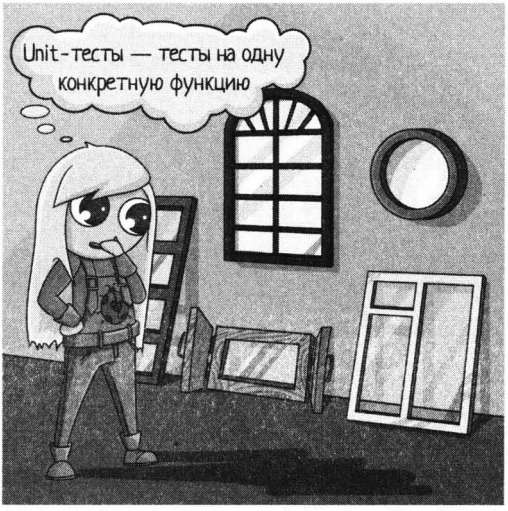
Unit-тесты

Unit-тесты — это самые мелкие тесты, которые пишут на *одну* конкретную функцию. Если для загрузки отчета надо:

1) посчитать ячейку А;
2) посчитать ячейку Б;
3) ... (посчитать еще 50 ячеек);
4) отрисовать таблицу,

то юнит-тест будет на что-то одно — проверять подсчет либо ячейки А, либо ячейки Б. Мы не проверяем тут правильность всего отчета, а исследуем его небольшую часть. Один кирпичик, из которого потом будет построен дом.

Если мы хотим установить окна в доме, на этом этапе мы тестируем одно конкретное окно. Целое ли стекло, правильно ли открывается?



○ **Скорость.**

Самые быстрые! Просто потому, что тут нет лишней обвязки, mock-сервисов[151] и прочего. Вызывается ровно одна функция, а это делается очень быстро.

○ **Как тест выглядит?**

Чистый код на языке программирования. Если само приложение на java, то и код юнит-теста будет на java.

○ **Нужен ли доступ к исходному коду?**

Нужен! Вы не можете написать юнит-тесты на любое приложение из сети — например, на Google, OZON или Users[152]. Только на свой проект, если разработчики пустят в код. Или на open-source приложение с открытым исходным кодом.

○ **Локализация багов**.

Очень простая. Так как мы задействуем ровно одну функцию, то упавший тест сразу указывает на место возникновения ошибки.

○ **Автор.**

Разработчик. Иногда автоматизаторов пускают писать юнит-тесты, но обычно это невыгодно по трудозатратам. Тестировщику надо вникать в код и писать код юнит-теста. А разработчик код уже написал, он и тестовый класс может сделать.

Хотя я рекомендую проводить ревью юнит-тестов, написанных разработчиками. И призывать их писать тесты понятнее ;-)

API-тесты

API-тесты — это тестирование функционала без необходимости поднимать сервер приложения, открывать браузер и т. д. Если для загрузки отчета надо:

1) посчитать ячейку А;
2) посчитать ячейку Б;

3) ... (посчитать еще 50 ячеек);

4) отрисовать таблицу,

то апи-тест может проверить весь отчет целиком! Кроме отрисовки таблицы, так как это уже часть интерфейса. И все равно этот тест окажется быстрее, чем «честный» тест черного ящика. Потому что мы выкидываем ожидание загрузки веба и его отрисовку, сосредоточив внимание на главном — на данных отчета.

Убедившись в unit-тестах, что функции работают по отдельности и каждая ячейка считается, мы проводим более интересный с точки зрения пользователя тест: как функции работают вместе, какие данные попадают в отчет? Проверяем интеграцию разных участков кода. А то, может, по отдельности работают хорошо, а вместе конфликтуют[153].

Если вернуться к примеру с окном, то мы проверили окна по отдельности. А теперь смотрим, как они работают вместе. Не мешают ли друг другу?

2 unit tests. 0 integration tests

○ **Скорость.**

Быстрые! Но не такие быстрые, как unit-тесты, потому что для них некоторая обвязка все же нужна. Хоть мы и не запускаем реальный браузер и не поднимаем сервер приложения, мы должны подменить все входы/выходы на заглушки, а это уже подготовка тестового окружения.

Для сравнения — если у нас есть 20 тестов, сколько они примерно идут:

- ✧ Unit — 2 секунды;
- ✧ API — 1 минута.

С другой стороны, может быть и такое, что половина этой минуты тратится на подготовку окружения, а потом — что 20 тестов 30 секунд займет, что 50, что 100.

○ **Как тест выглядит?**

А вот тут уже пространство для фантазии! Как разработчик сделает, так и будет. Это может быть как «чистый» код, так и более человекочитаемый код, или вообще не код!

Например, у меня на одном из проектов для проверки данных в БД использовались эксельки. Что на входе, что на выходе. Остается лишь заполнить эти Excel-таблицы, а это может сделать человек, который код вообще не понимает. Ведь кода нет!

Конечно, «под капотом» у этих тестов, где не надо знать языка программирования, лежит целый фреймворк, написанный разработчиком. Зато один раз написал — и даже джуниор может использовать!

Вот пара вариантов, как можно сделать автотесты, которые сможет пополнять даже нетехнический человек:

- ✧ разработчик написал фреймворк, а тестировщику остается заполнять эксельки входными данными. Фреймворк — это заготовка, тот код, который скрыт «под капотом».

Похожий принцип используется в folks[154]. Чтобы писать эти тесты, нужен тест-дизайн. Про java вы можете вообще ничего не знать;

- ✧ разработчик написал фреймворк и примеры тестов. На каждое возможное действие он создал человекочитаемую функцию. И тесты выглядят как-то так:

```
register("Ольга", "olga@mail.ru", "1");
login("olga@mail.ru", "1");
```

Какие параметры можно в функцию передать — разработчик где-то описал или просто рассказал словами. А тестировщик теперь сам пишет автотесты, не зная код.

В итоге от тестировщика нужно знание английского да тест-дизайн.

○ **Нужен ли доступ к исходному коду?**

Нужен! Вы не можете написать API-тесты на любое приложение из сети — например, на Google, OZON или Users[155]. Только на свой проект, если разработчики пустят в код. Или на open-source приложение с открытым исходным кодом. Но тут хочу отметить, что мы сейчас ведем речь о внутреннем API. То есть том API, которое вызывается внутри программы. Именно о таком API идет речь в пирамиде автоматизации.

Но при этом учтите, что бывают еще тесты на внешнее API, для которых доступ к коду *не нужен*. Внешнее API — то, что выставлено в общий или не очень общий доступ для интеграции. Обычно это SOAP или REST API.

Например, в Users есть такое API[156]. У JIRA есть API, у подсказок с Дадаты есть публичное API. Что значит «не очень общий доступ»? Это если доступ

есть только с логином/паролем или в определенной подсети. Например, когда разрабатывается API для банка, оно будет — ради безопасности — работать только в его подсети. Или платные услуги — та же JIRA платная, поэтому просто так подергать ее API нельзя. Вот в Users авторизация не нужна, это апи сразу в общем доступе!

Так или иначе, у нас есть некое API, которое можно подергать. В идеале даже документация к нему есть! Вот по ней и дергаем. Или без документации... В любом случае нам не нужен доступ к исходному коду самого приложения. Мы работаем только с этим API.

На него можно даже автотесты написать. В том же Postman — создать коллекции запросов, сделать проверки на то, что в ответе правильный статус-код, нужный набор полей и так далее. Вы можете написать такие тесты на Users, хотя исходного кода программы у вас нет.

○ **Локализация багов**.

Тут уже посложнее. В API-части может быть задействовано сразу несколько разных функций. И, чтобы понять, в какой именно нашлась ошибка, разработчики должны грамотно работать с внештатными ситуациями, выдавая понятные сообщения об ошибке.

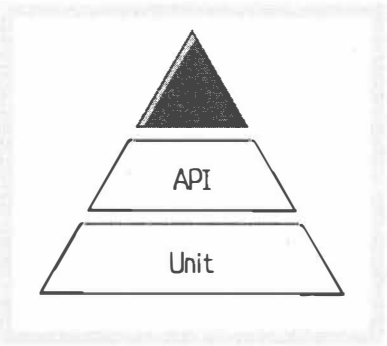
Если сообщения написаны хорошо — понять их сможет даже тестировщик. Но, даже если из сообщения не понятно ничего, локализовать баг не проблема. Ведь он запускается из исходного кода приложения! Разработчик просто дебажит тест и смотрит, где ошибка. И всё!

○ **Автор.**

Разработчик + тестировщик. У меня на одном проекте разработчики писали фреймворк, а тестировщики — сами тесты.

UI-тесты

Пирамида автоматизации



UI-тесты — тесты через графический пользовательский интерфейс. Это верхушка пирамиды, самые честные тесты. Робот все делает так, как делал бы реальный пользователь.

Помните пример с отчетом? Для загрузки отчета системе надо:

1. Посчитать ячейку А.
2. Посчитать ячейку Б.
3. *... (посчитать еще 50 ячеек).*
4. Отрисовать таблицу.

Но это внутренняя логика приложения! Пользователь о ней не знает и знать не должен. С его стороны это выглядит так:

1. Открыть браузер.
2. Открыть систему.
3. Нажать «Войти».
4. Авторизоваться.
5. Перейти на вкладку «Отчеты».
6. Выбрать период.
7. Нажать «Загрузить».
8. Дождаться окончания загрузки.
9. Профит!

Вот именно это и делает UI-тест! Тестирование черным ящиком. Вы можете вообще ничего не знать про потроха системы, как она устроена и т. д.

Вам не надо иметь доступ к коду (в случае Unit- и API-тестов — надо). То есть вы можете автоматизировать вообще любое приложение: хоть Google, хоть Facebook, хоть тот же Users.

А что там система делает внутри, сколько функций вызывает для расчета каждой ячейки — нас не волнует. Нас волнует конечный результат.

Иногда кажется — зачем эти тесты, ведь я все проверил на уровне API? Но «честные» тесты тоже нужны. Бывает так, что в автотестах все хорошо, а реальный отчет не грузится. На то могут быть разные причины:

- ошибка в фреймворке автоматизации — он не учитывает деталь, которая выполняется в реальности и которая все ломает;
- ломается что-то именно в интерфейсной части: поехала верстка, не найден файл CSS или что-то такое.

Я лично сталкивалась с тем, что в интерфейсе ломалось какое-то базовое действие. На него был автотест, он НЕ упал. А «в реале» сломалось... Разработчик посмотрел, почему. Оказалось, там внутри системы вызывалось еще одно действие, которое в автотесте не использовалось, оно шло до заглушки.

Именно поэтому уровень GUI-тестов находится на верхушке пирамиды. Они могут быть, но их должно быть мало. Большинство тестов должны быть на уровне кода: Unit или API. Иначе вы найдете баг, но:

- его будет сложнее локализовать (что именно сломалось?);
- исправлять сложнее, чем на этапе начала разработки.

Вот небольшой чек-лист вопросов, которые стоит задать себе, перед тем как начинать писать GUI-тесты:

- все готово?
- меняться интерфейс не будет?
- кнопочки не переделаем?
- ID не изменятся?
- функции останутся?

Если ответ «нет» — не лезьте. UI-тесты пишутся на стабильный функционал, иначе вы просто задолбаетесь их актуализировать. Во время разработки

интерфейс очень быстро меняется, а это значит, что написанные вчера тесты сегодня можно выкинуть и переписать по-новой.

— А если у меня нет UI?

Хороший вопрос! На самом деле даже без UI пирамида не меняется, просто вместо UI будет написано «честные тесты» или «интеграционные тесты».

Вам все равно нужны честные тесты. Потому что все равно могут возникать ошибки, которые не поймали тесты на уровне API.

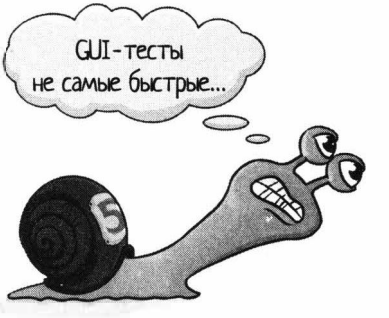
Что делают такие тесты? Поднимают честный сервер приложения и выполняют операции через него. Без заглушек. У нас, например, два сервера приложений общаются между собой. Это разные приложения, у каждого своя команда тестирования. Каждое приложение покрыто Unit- и API-тестами, но обязательно есть хотя бы несколько интеграционных тестов, которые разворачивают оба сервера и проверяют их общение «по-настоящему».

Так что пирамида не меняется: максимум легколокализуемых и быстрых тестов, минимум честных интеграционных, проверяющих «сразу и всё».

○ **Скорость.**

Самые медленные. В легких случаях мы просто тестируем Сеть через черный ящик, и то они идут долго, ведь это надо:

1. Открыть браузер.
2. Загрузить сайт, дождаться загрузки.
3. Авторизоваться.
4. Перейти в нужный модуль.
5. Дождаться загрузки.
6. ....



А если мы еще и сами сервер поднимаем, это еще дольше:

1. Собрать билд.
2. Задеплоить его — запустить сервер приложения, а это 1–5 минут.
3. Открыть браузер.
4. ... (см. шаги ранее).

○ **Как тест выглядит?**

Это снова код на языке программирования. Но! Для записи и запуска теста вы можете этот язык и не знать, если будете использовать функцию «Record And Play» — она есть во многих инструментах и идеальна для легкого старта.

«Record And Play» — обычно большая красная кнопка. Нажали ее — дальше все ваши действия будут записываться. Дотронулись до мышки, нажали на пробел на клавиатуре — запишется все. Так что открываем браузер и выполняем все нужные нам действия. Система это все запишет и сохранит. Тест потом можно будет запускать повторно.

Конечно, нельзя строить автоматизацию на таких тестах. У них есть куча минусов:

- ✧ легко ломаются — потому что робот не может записать оптимальные локаторы (те штуки, которые ищут элемент на странице);
- ✧ сложно исправляются — так как там получается много копипасты. На каждый тест нужно авторизоваться? А если кнопку входа переименовали? Нужно менять все тесты! А вот если писать код самому, то это можно оптимизировать (написать в одном месте и ссылаться на него).

Но зато такие тесты хороши как помощники. Они не заменят грамотную автоматизацию, но помогут ручному тестировщику. Накидали пяток тестов, которые важно проверить? Ну и хай себе выполняются! А если развалятся, перезаписал — и всё!

○ **Нужен ли доступ к исходному коду?**

Не нужен! Вы МОЖЕТЕ написать UI-тесты на любое приложение из Сети: хоть на Google, хоть на Users.

И это — главное преимущество таких тестов. Если Unit- и API-тесты может написать только команда разработки продукта, UI может протестировать кто угодно. Но зачем тестировать UI не своего продукта?

- ✧ Если вы являетесь заказчиком продукта — вам надо как-то его проверять, но кода то нету, только само приложение.
- ✧ Новичок для практики может писать автотесты ;-)

○ **Локализация багов.**

Самая сложная локализация. Потому что в одном тесте задействована куча всего. И разные функции, и функционал клиентской стороны (отрисовка данных) — где именно сломалось? Тут нужно разбираться.

Фактически это как при тестировании вручную методом черного ящика. Сначала что-то упало, а потом мы разбираемся, что и почему. Но в автотестах есть плюс! Разработчик все еще может подключиться к приложению и подебажить его. Так он быстрее найдет ошибку.

○ **Автор.**

Тестировщик. Может и разработчик такие тесты написать, если очень захочет. Но вообще-то это задача именно тестирования.

Подведем итоги

В программе:

○ UI-тесты — честный тест, «как это делал бы пользователь». (они же GUI, Graphical User Interface);

○ API-тесты — опускаемся на уровень ниже, выкидывая лишнее;

○ Unit-тесты — тесты на отдельную функцию.

Начинаем писать тесты снизу, потому что сначала логичнее проверить небольшой участок кода, а потом усложнять:

○ Unit — тесты на отдельную мелкую функцию (посчитать одну ячейку отчета);

○ API — тесты на конкретный функционал, который состоит из отдельных функций (загрузить весь отчет);

○ GUI, интеграционники — честный тест через графический интерфейс: «Как это делал бы пользователь» (открыть браузер, войти в систему, перейти в отчеты и, наконец, вызвать отчет).

На примере танца:

○ Unit — одно движение показали, отработали.

○ API — связка из 3–5 движений, небольшой отрывок.

API

○ GUI, интеграционники — весь танец целиком.

UI

На примере платья:

- Unit — раскроили и с выкройкой сверили каждую деталь.

- API — обметали и проверили.

- GUI, интеграционники — когда уже платье полностью готово, проверяем еще раз.

При правильном подходе у вас количество тестов должно быть максимальным внизу пирамиды и минимальным — вверху.

Да, если вы новичок в автоматизации, так и тянет начать с UI-тестов, ведь про них есть куча курсов, статей и прочего. И даже в вакансиях постоянно мелькают модные словечки типа Selenium[157].

Но! Если вы хотите начать применять автоматизацию на своем проекте, а не учите ее для себя или красивого резюме — договаривайтесь с разработчиками, чтобы писать Unit- и API-тесты. Если им некогда, пусть делают понятные функции с одним примером, которые уже вы будете пополнять.

- GUI-тесты — плохой выбор, потому что:
  - ✧ они медленные;
  - ✧ они хрупкие (легко ломаются);
  - ✧ если они найдут баг, его сложно локализовать.

Автоматизация рутины

Если вы хотите начать изучать автоматизацию, у вас есть два пути:

1. Изучать GUI-тесты, тот же Selenium.
2. Автоматизировать рутинные задачи.

Угадайте, от чего вы получите больше профита, что положительно скажется на вашей работе? Правильно, путь 2!

Подумайте о том, что занимает у вас много времени. Или немного, но что вы ОЧЕНЬ не любите делать?

Из собственного опыта...

На одной работе для входа в систему нужно было заполнять анкету. Большую такую анкету! Полей на 20. Я делала эту вручную и очень скоро стала страдать даже от заполнения бездумными данными типа «лтмаолм»

На другой работе в задачу тестировщика входил деплой сервисов. Нужно было собрать проект и задеплоить. Или на своей машине, или на DEV-стенде, доступном всей команде. А это каждый раз по 5–10 минут тратить, чтобы два билда собрать и поднять.

А еще на одной работе у нас нужно было заполнять ворклоги в JIRA. Не всем это нравится ;-) Есть люди, которые не хотят открывать баг-трекер и вносить туда данные, поэтому они пишут все в блокнот. Но ведь потом надо из блокнота переносить в инструмент!

А когда у заказчиков что-то ломалось, мы просили их собрать логи с двух серверов приложения за нужный период времени. Каждый сервер пишет свой лог. Логи разбиты на разные файлы: ***server.log***, ***console.log***, ***stats.log***, ***errors.log***... Каждый день создается новый лог — получается ***server.log*** (актуальный), ***server-2019-10-09.log***, ***server-2019-10-08.log***... Так что собрать логи за неделю — это штук 40 разных файликов вручную натаскать. Тоска! И обязательно нужный файлик продолбается при сборке...

Унылые задачи

Именно такую работу и нужно автоматизировать!

Это может быть как полуавтоматизация, так и автоматизация процесса. Помните, из раздела классификации тестирования *главы 9*? Автоматизация — когда робот делает всю работу за тебя. Полуавтоматизация — когда он помогает с какой-то рутиной, а остальное делаешь сам.

Вот, например, мой коллега написал утилиту, которая берет его записки из блокнота и по ним заполняет ворклоги в джире. Но ему все равно надо вести записи! Это не полностью автоматизированный процесс.

Или заполнение анкеты — это один из шагов тестирования. Я написала робота на C#, который заполнял анкету за меня. Я потратила на него время и даже думала «зачем я это все делаю?? Быстрее было бы заполнить ручками». Но потом много раз, запуская робота и попивая чаек, я наслаждалась процессом. Поля заполняются сами, а ты сиди да отдыхай. Красота! Пусть этот отдых меньше минуты, все равно классно.

А вот сбор логов мы просто заавтоматизировали. Теперь достаточно нажать одну кнопочку — и готово!

И для деплоя сервисов написали скрипты. Сначала для линукс-сервера, просто в помощь тестировщику. Выложил билд в конкретную папочку и запустил скрипт. Работает!

Потом для наших DEV-платформ решили вообще весь процесс заавтоматизировать. Теперь любой человек, хоть начинающий тестировщик, хоть менеджер проекта, в глаза не видевший код, может зайти в TeamCity[158], нажать на кнопочку — и вуаля! Работает!

Очень важно автоматизировать такие вещи. Да, вручную это тоже занимает мало времени. Иногда буквально 1-2-3 минуты. Но если это делается каждый день, по несколько раз в день... То лучше облегчить себе жизнь! Поверьте, это окупится.

Где-то вы можете помочь себе сами, где-то стоит попросить помощи у разработчиков. Зависит от сложности задачи. Но если у вас есть рутинная работа — избавьтесь от нее. Это, в том числе, поможет справиться с профессиональным выгоранием.

Работа может очень быстро надоесть, если из раза в раз делать одно и то же. А если вы новичек, то наверняка вам достанется как раз «черная» работа типа прогона одного и того же чек-листа критически важных проверок. Помогите себе, начните автоматизировать.

Если автоматизировать интерфейс программы нельзя или очень сложно (desktop-приложение сложнее автоматизировать), автоматизируйте рутину. Или если у вас есть автотесты, но все равно осталась какая-то унылая работа типа разворачивания серверов.

Инструменты полуавтоматизации

Напомню еще раз, что такое полуавтоматизация. Мы обсуждали это в *главе 9* (см. *разд. «ПолуАвтоматизированное тестирование»*). Автоматизация — когда робот делает всю работу за тебя. Полуавтоматизация — когда он помогает с какой-то рутиной, а остальное делаешь сам.

Так вот, если для автоматизации надо знать язык программирования, то для полуавтоматизации это не всегда бывает нужно! Иногда достаточно освоить регулярные выражения и применять их в блокнотике!

Хотя, безусловно, знание языка программирования расширит ваши возможности. Давайте пройдемся по инструментам, которые могут вам пригодиться.

○ **Pairwise.**

Техника pairwise применяется для тестирования попарных значений. Эффективна тогда, когда у нас много параметров, но мало значений в каждом.

Мы изучали ее в *главе 4* (см. *разд. «Техника pairwise»*). Там же описаны и инструменты для ее применения. Тут я хочу лишь напомнить, что

pairwise — это автоматизация рутины! Причем доступная каждому, ведь для нее вам не нужно знать код вообще.

○ **Валидатор битых ссылок.**

Вставляем URL: **http://validator.w3.org/checklink** — робот сам проверяет все ссылки на странице! И расскажет вам, есть ли среди них битые.

○ **Регулярные выражения.**

В программе Notepad++ встроен функционал регулярных выражений. Очень помогает при обработке текста (логи, ТЗ). Или можно использовать любой сайт для регулярок.

Унылые задачи

В предыдущем разделе мы обсудили примеры задач, наводящих тоску и уныние. И я рассказала, как мы их решали:

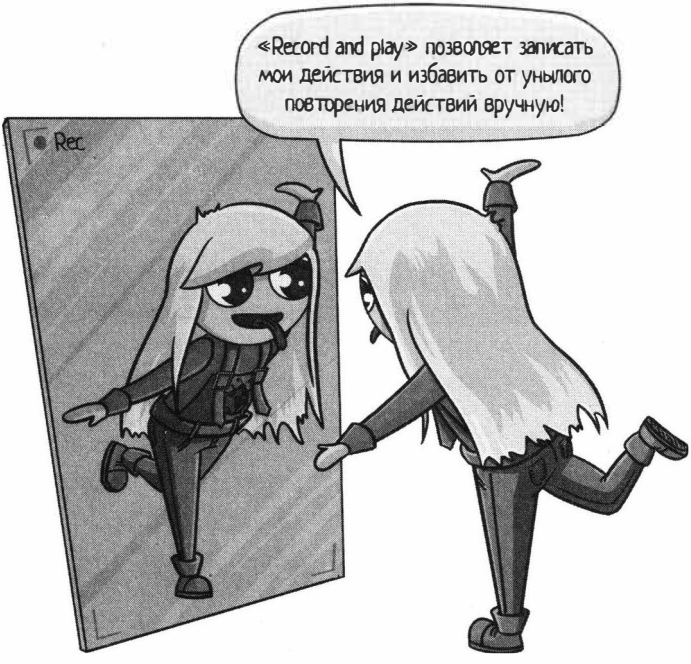
○ заполнение анкеты — робот на C#;

○ деплой сервисов — скрипт, позже встроенный в CI;

○ ворклоги — самописная утилита;

○ логи — самописная утилита.

Заполнение анкеты я сделала, когда училась автоматизации. Это несложная работа даже для новичка. Можно сделать запись даже через «Record And Play».



Утилиту для ворклогов коллега тоже написал сам на питоне. У нас на работе это не первый случай автоматизации на коленке с помощью процедурных языков программирования.

Другой коллега написал утилитку, которая проходит по файлику с данными и для каждой строки дергает REST-сервис — так стало возможным обрабатывать файл, хотя исходно в сервисе такой возможности нет. И что вы думаете? Все коллеги тут же стали использовать эту утилиту!

Более того, наш внедренец смастрячил подобную штуку на коленке во время внедрения продукта. Так ее вывели в PROD! Хотя он ее оставил там чисто для тестирования системы...

Или те же логи. Собрать собрали, а дальше что? Когда-то ответственный за поддержку вручную открывал логи и делал поиск по ключевым словам — искал ошибки и варнинги (уровни логирования `ERROR` и `WARN`). Но чем больше логов, тем скучнее процедура... Поэтому снова написали питонячий скриптик, который сам сортировал логи и делал нужные выдержки в отдельные итоговые файлики.

Он еще и статистику по задачам умел считать! Идут в логах записи вида:

- отработала задача загрузки данных. Новых клиентов создано 100 000, обновлено 50 000, ошибок 3;
- отработала задача поиска дубликатов. Найдено дубликатов — 600 000, из них новых 100 000, объединено 20 000 карточек.

Раньше человек находил эти записи и вручную переписывал данные в табличку. Когда задача началась, когда закончилась, сколько времени шла, какой результат — ведение таблицы помогает отслеживать деградацию

производительности и вовремя с ней разобраться. Но вести эту табличку очень уныло! Самому... А когда ее заполняет робот — жизнь прекрасна 😊

Помимо питона может помочь знание регулярных выражений! Это полезно, когда ищешь что-то в логах на сервере и у тебя есть только командная строка, никакого графического интерфейса. Или если логи такие огромные, что никакой блокнот их не откроет. Так что тестировщику очень полезно знать на базовом уровне:

- основные команды bash;
- регулярные выражения.

А если вы знаете регулярные выражения, то их можно применять даже в... простом Блокноте! Да-да ;-) Правда, для меня «простой Блокнот» — это Notepad++, не понимаю, как после него можно использовать стандартный виндовый... Вот в Notepad++ регулярки есть!

Его можно применять, когда есть несложная, но нудная работа. Я так и делала — и тем спасалась от тоски. А то ведь это все копится, уже и работа не в радость, когда каждый день одно и то же, или проблемы в процессе есть и пока не решаются...

Хорошо помню, как исправляла тесты — ну нудная задачка же! А потом придумала сделать автоматизацию в Блокноте. То есть решить ту же задачку... через Блокнот! Надо признать, это заняло больше времени, чем «сделать вручную». Зато проснулся интерес и азарт! Скуку как рукой сняло ;-)

Так что такая автоматизация идет и как вектор развития, и как способ борьбы со скукой и профессиональным выгоранием!

Инструменты автоматизации тестов, а не рутинных задач, я оставляю вам на самостоятельное изучение. Ну просто потому, что одной главы для охвата автоматизации просто не хватит! Я даже не возьмусь. Упомяну только в общих чертах:

- если мы говорим про API-тесты внутри приложения — это, скорее всего, будет самописный фреймворк на том же языке программирования, что и основной код;
- если это внешние API-тесты — инструменты типа Postman, SoapUI, но их рассмотрение явно выходит за рамки нашей книги;
- если это GUI-тесты — инструмент, который у всех на слуху. Сейчас это Selenium для веба, TestComplete для десктопа.

Если вам интересна именно автоматизация — начинайте копать в ее сторону. Но учтите, что для начала надо понять основы тестирования. Иначе вы будете не автоматизатором дорогостоящим, а простым кодером, который может лишь перекладывать на код составленные другими людьми тесты. Надеюсь, это не ваш выбор ;-)

Лично я раньше никогда не любила автоматизацию и заниматься ею не планировала. Хотя, когда у нас стало мало задач (фирма находилась на стадии слияния), от скуки стала изучать автоматизацию. Робота вон написала[159]. И все же не планировала связывать с этим карьеру.

А потом на другой работе стала писать тесты на уровне API. Это прикольно, даже код знать не надо! Потом стала проводить ревью Unit-тестов, просто читать код — у нас на работе это нужно уметь. Система написана на java, изучала ее по книгам, выполняя все домашки, чтобы закрепить в голове материал.

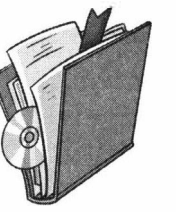
Конкретно сейчас меня очень увлекла автоматизация на уровне Postman. Я пишу автотесты на JS, обучаю этому студентов. Даже книгу в стиле HeadFirst O`Really хочу написать! Что будет дальше? Не знаю, посмотрим...

Вопросы для самопроверки

1. Зачем нужна автоматизация?
2. Сколько GUI-автотестов надо написать на постоянно меняющуюся систему?
3. Когда нужно применять автоматизацию?
4. Чем отличаются Unit-тесты от тестов API?

5. А API-тесты от тестов GUI?
6. Назовите инструменты, которые можно применять для автоматизации рутины без знания языка программирования.

Портфолио



Если вас заинтересовала тема автоматизации, вы можете изучить ее самостоятельно и добавить написанный код в своё портфолио. Просто залейте его в git или bitbucket и приложите ссылку на свой репозиторий в резюме.

На чем тренироваться, если вы еще нигде не работаете? Конечно, на Users! Эта система бесплатна, лежит в общем доступе... Мучайте ее, только без нагрузочных тестов!

Здесь вы найдете:

- кучу функционала, который можно проверить через GUI-тесты;
- SOAP и REST API, их тоже можно пощупать!

Ответы на вопросы для самопроверки

1. Зачем нужна автоматизация?

Чтобы робот работал, а человек отдыхал! На самом деле автотесты:

- ✧ избавляют от рутины;
- ✧ ускоряют процесс тестирования: робот может прогнать сотни тестов за 5 минут, человек тот же функционал будет проверять сутки;
- ✧ тщательно все проверяют, ничего не забыв. У человека может замылиться глаз, робот же всегда четко выполнит все инструкции. Он не устает и ему не надоедает.

2. Сколько GUI автотестов надо написать на постоянно меняющуюся систему?

Ноль! Не надо писать тесты, которые завтра развалятся. Вы потратите много времени на то, что вскоре выбросите в помойку. Автоматизация нужна не для этого.

3. Когда нужно применять автоматизацию?

Когда функционал и/или графический интерфейс стабилен. Вы один раз пишете тесты, проверяя, что все работает. А потом робот это будет проверять за вас. Регрессия — отличная тема для автоматизации!

4. Чем отличаются Unit-тесты от тестов **API?**

Unit-тест проверяет одну конкретную маленькую функцию, а тест API — целый набор. API-тест проверяет функционал, при вызове которого может вызывать хоть 100 небольших функций.

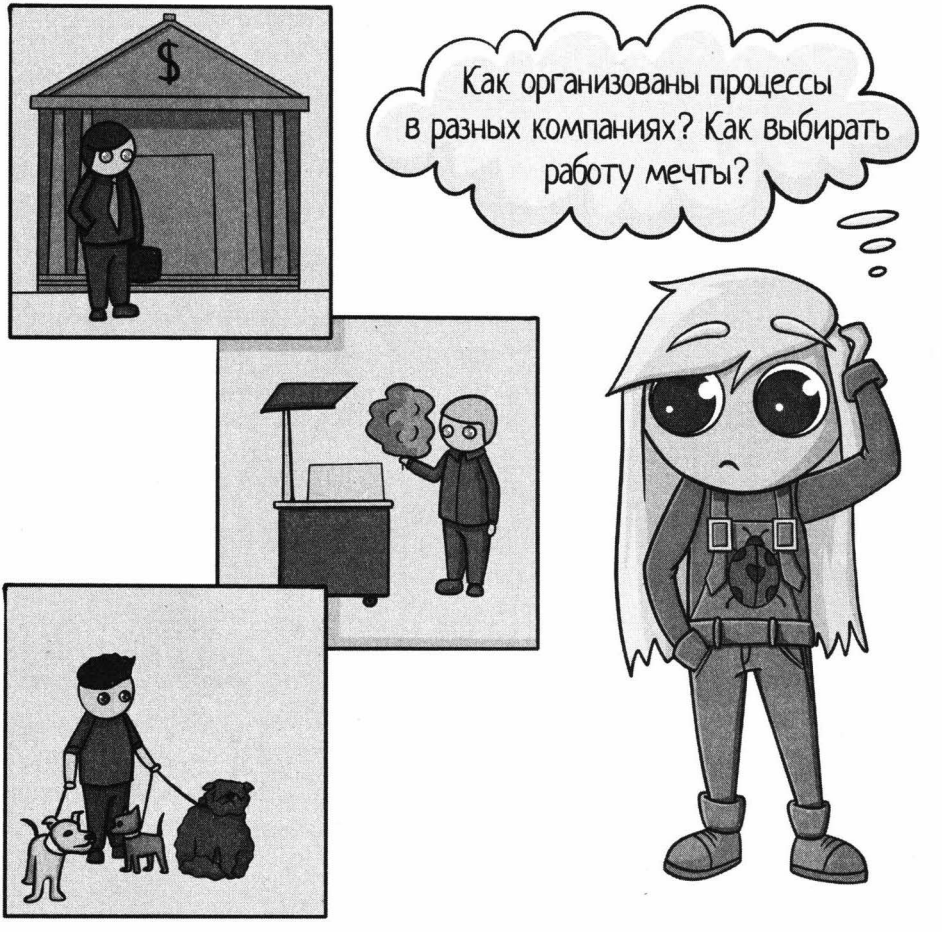
5. А API-тесты от тестов GUI?

GUI использует графический интерфейс, API — нет. GUI-тесты медленнее и легче ломаются, зато они честные, делают все «как сделал бы человек».

6. Назовите инструменты, которые можно применять для автоматизации рутины без знания языка программирования.

Allpairs — для попарного тестирования. Блокнот или командная строка — если нужны регулярные выражения.

Глава 11
ОРГАНИЗАЦИЯ ПРОЦЕССА



Скоро вы уже будете искать работу своей мечты. Давайте посмотрим, как работают разные компании, чтобы понимать «А куда я хочу пойти?»

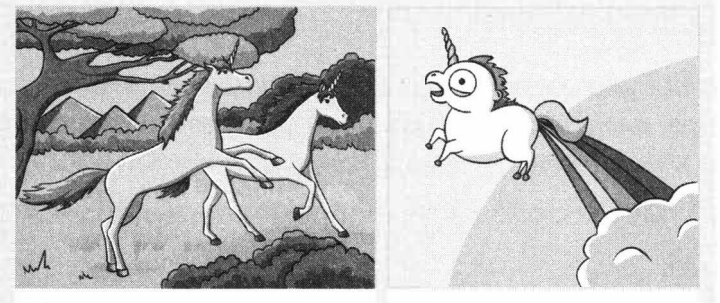
В этой главе мы рассмотрим процессы в разных компаниях:

- гиганты;
- стартапы;
- аутсорс.

Стоит ли вмешиваться в процесс?

Мы сейчас обсудим, какие процессы бывают в разных компаниях, плюсы и минусы. Вы придумаете свой идеал, придете на работу... А там все не так! Но как же так? Я же в книжках читал / меня на курсах учили, что нужны ежедневные митинги[160] и всё такое. Ну уж сейчас-то я тут порядки наведу! Да? Нет! В чужой монастырь со своим уставом, знаете ли...

Подумайте сами — текущий процесс в компании давно. К нему уже все привыкли. Может, что-то где-то не устраивает, но не настолько, чтобы это менять. И тут приходит новичок и начинает учить всех уму-разуму. Как будто он знает, как лучше. Особенно забавно это слышать из уст человека, который еще нигде не работал, только теорию знает.



С другой стороны, нельзя не признать, что иногда новые люди реально могут улучшить процесс. Потому что пришли со свежим взглядом и сразу видят проблемы, с которыми старички давно смирились.

У нас так новый PM когда на продукт пришел, активно использовал свой свежий взгляд. И до сих пор[161] благодарит за любую критику, обдумывая, как бы улучшить процесс. При этом он не вносит кардинальных изменений, которые могут вызвать протест команды. Он делает по чуть-чуть и смотрит на результат. Внедрять улучшения надо уметь! ;-)

Так что, если вас просят поставить процесс с нуля — ставьте, экспериментируйте! Хотя лично я против того, чтобы новичок с нуля строил процесс тестирования. Ведь он по дороге соберет все грабли! Будет лучше нанять опытного специалиста, который все настроит, а потом уйдет или останется тест-менеджером.

Но, увы, компании любят экономить. У меня во флудилке выпускников ребята постоянно пишут, что устроились первым и единственным тестировщиком. И их это вполне устраивает, ведь атмосфера дружеская, и вроде все получается. Наверное, спустя лет пять они поймут, что могли все сделать намного лучше, но... Если всех всё устраивает, то почему бы и нет.

В любом случае решать всегда вам. Хотите пробовать ставить тестирование с нуля? Не рекомендую, но кто ж вам запретит? Я бы на вашем месте сначала поучилась у опытных.

А пока запомните главное правило программиста: «Работает? Не трогай!». Не спешите менять устоявшийся процесс. Если неудобно вам — это не значит, что все вокруг страдают. Тут вспоминается «бородатая» притча:

Сидит программист глубоко в отладке. Подходит сынишка:

— Папа, почему солнышко каждый день встает на востоке, а садится на западе?

— Ты это проверял?

— Проверял.— Хорошо проверял?

— Хорошо.— Работает?

— Работает.— Каждый день работает?

— Да, каждый день.

— Тогда ради бога, сынок, ничего не трогай, ничего не меняй.

Поживите в процессе, осмотритесь. Что неудобно вам? А коллегам? Опросите их и запишите проблемы. Выберите самую «бесящую». Предложите решение. Небольшое, а не «давайте полностью все поменяем!». И попробуйте с ним жить, а потом беритесь за следующую проблему.

И помните:

— люди не любят изменений;

— хочешь изменить мир? Начни с себя!

Не стоит ждать, что меняться будут другие. Что вы скажете: «Давайте напишем ТЗ понятнее через сценарии использования», и аналитики все бросят и побегут переписывать. Нет, хотите улучшить? Начните это делать сами. Команда увидит результат, и если он им понравится, сами пойдут по вашему пути.

Процессы в компании-гиганте

Кратко

1. Много людей.
2. Много бюрократии.
3. Четко отлаженные процессы.
4. Тест-кейсы.
5. Много подразделений.
6. Большая зарплата, «белая».

Подробнее

Бюрократия

Большая компания = много людей. А чем больше в компании людей, тем больше бюрократии. Разумеется, есть исключения.

В ИТ-конторах, если процесс хорошо построен и отлажен, вы будете в шоколаде. Что-то надо? Попросил HR или другого специально обученного человека — и получил. Если бюрократия есть, то она скрыта от вас под капотом.

А если говорить про банки, то все наоборот. Сломалась компьютерная мышка? Делай запрос и жди. Нужно поехать на конференцию? Согласуй у пяти разных начальников. И так далее.

Процесс тестирования

Компания живет давно, уже наверняка перепробовала все, что только можно. И выработала свой собственный подход к процессу разработки ПО. Вам просто остается влиться в этот процесс. Не надо изобретать собственные велосипеды.

Налаженный процесс — огромный плюс. Он, как хороший сисадмин — его вообще не замечаешь, словно так всегда и было. Хотя, чтобы понять преимущества, нужно сначала побывать на проекте с процессом «полный хаос» ;-)

Для тестирования используются тест-кейсы. По разным причинам:

- так как людей в компании много — могут быть ротации между командами. Значит, нужно, чтобы незнакомый с проектом человек мог пройти по тестам;
- джуниоров тоже много, и их надо обучать. А обучать лучше всего по тест-кейсам.

Разумеется, где-то есть и чек-листы, и небольшие команды со своими процессами. Не бывает такого, что «большая компания — только так, маленькая — иначе». Я говорю только про общую тенденцию, а дальше уже уточняйте на собеседовании, как работают в конкретной компании.

Команда

В идеале, конечно, даже внутри большой компании у вас будет небольшая команда на 7–10 человек. Внутри которой и аналитик, и разработчик, и тестировщик — все работают вместе. Но такое встречается редко.

В большой компании много народа. Большой проект, его делают не 1–2 разработчика, а 5–10. И столько же тестировщиков. Значит, в каждой команде есть главный. Главный разработчик, главный тестировщик (тест-менеджер).

Часто команды разделены: отдельно команда разработки, отдельно отдел тестирования. Тут есть плюсы — всегда можно обернуться к соседу

и обсудить тестирование. Есть и минусы — рядом нет разработчика, чтобы быстро обговорить какую-то деталь.

Зато у вас наверняка будет ментор — человек, который поможет прокачиваться в новой профессии. Подскажет, как лучше тестировать, какие книги и доп. материалы прочитать, на что сделать упор.

Обучение

Есть внутреннее обучение — оно дешевле и направлено на те навыки, которые нужны именно для этой компании. То есть не просто тест-дизайн, а, например, классы эквивалентности для банковского ПО. Вам расскажут, на что обращать внимание, куда в вашем приложении копать.

Есть внешнее обучение: или вы выбираете курс, на который хотите пойти, или ваш руководитель решает, что будет полезнее. Далее варианты:

- компания полностью оплачивает обучение;
- компания полностью оплачивает обучение при наличии у вас сертификата — то есть когда вы реально учились, а не пинали балду вместо занятий;
- компания компенсирует часть стоимости обучения.

Конференции — да, а как же! Туда отправляют сразу несколько человек. Новички слушают и вникают, более опытные выступают. Книги нужны? Компания закажет, может быть даже корпоративная библиотека есть.

А еще есть индивидуальное обучение — новичку дают ментора, который будет следить за его ростом. Это может быть тест-менеджер или специально обученный человек. Он все покажет, расскажет, научит. И именно из-за ментора и грамотных процессов большая компания — отличный старт для новичка!

Зарплата

Официальная, белая.

А в банке — еще и большая. Особенно если много бюрократии или плохие процессы, вас держат деньгами.

Плюсы

1. Четко отлаженные процессы.
2. Есть ментор — с ним вы быстрее прокачаетесь, нежели наступая на собственные грабли.
3. Обучение внутри компании или покупка курсов.
4. Белая зарплата.

Минусы

1. Бюрократия.
2. Рутинная работа (особенно у новичка — просто проходить тест-кейсы, например).
3. Может быть дресс-код, если работа в банке.

Процессы в стартапе

Кратко

1. Мало людей.
2. Нет бюрократии.
3. Процессы? Не, не слышал.
4. Нет времени на чек-листы — пыщ-пыщ и в продакшен!
5. Подразделений нет.
6. Зарплата небольшая, «черная» или «серая».

Подробнее

Бюрократия

Бюрократии нет! Хочешь что-то? Попроси. Более того, в небольшой компании иногда есть возможность напрямую поговорить с директором. Почему бы и нет?

Это если у него 1000 человек в подчинении, нет времени с каждым обедать, обсуждая KPI, узнавать новости и всё такое. А в стартапе основатель вообще может сидеть вместе с вами в одном большом кабинете!

Это очень мотивирует. Я сама помогала стартапу, где «бигбосс» сидел вместе со всеми, бегал с горящими глазами, рассказывая, что удалось сделать сегодня. Искренне восхищался нашим продуктом, своим детищем. Это огромный заряд мотивации!

Процесс тестирования

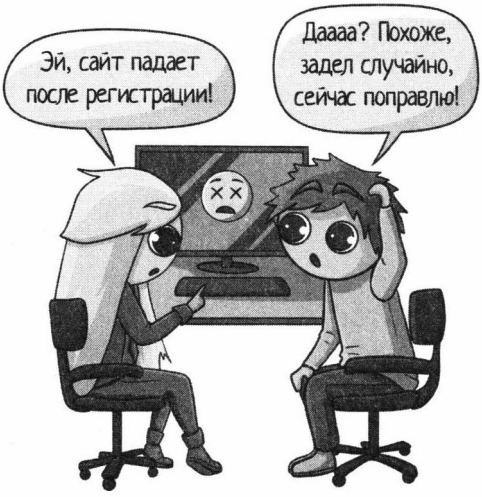
Никакого процесса нет ;-)

Очень может быть, что и тестировщиков там тоже нет. Тестировали своими силами: или разработчики, или аналитики. А потом осознали, что надо что-то менять, вот и ищут человека. Который придет и поставит процесс с нуля.

Конечно, в идеале это должен сделать опытный тестировщик. Но так как денег нет, ставить процесс ищут новичка. По крайней мере, мои ребята во флудилке рассказывают о таких случаях. Некоторые так устроились, и им нравится. Ведь атмосфера живая, коллектив классный, никто не гнобит, что ты делаешь что-то неправильно. Никто ведь и не знает, как правильно! Так и растете вместе.

Команда

Команда небольшая, может быть все даже сидят в одном кабинете. Вы сидите рядом со своим разработчиком, так что некоторые баги можно даже в баг-трекер не заносить. Повернулся, рассказал, через пять минут проверил — профит!



Ребята все классные, потому что подбирает генеральный «под себя» и чтобы глаза горели. Атмосфера в коллективе дружеская, нет подпольных игр «виноват вон тот отдел, и вообще все вокруг враги, а я хороший».

Обучение

Денег в стартапе тоже обычно нет, поэтому конференции и курсы могут не оплатить. А если деньги есть, то вообще без проблем! Ведь тут бюрократии мало ;-)

Как вариант — частичная компенсация. Вы едете на конференцию, вам возвращают 30 или 50% стоимости. Аналогично с внешними тренингами.

Внутреннего обучения нет, просто потому что процессов нет. И народу мало. Но вы сами можете его организовать, если хотите вытянуть из разработчиков знания ;-)

Библиотеки нет, книги покупаете сами.

Зарплата

Зарплата небольшая, денег то нет ;-)

Она может быть серой или полностью черной, если стартап начинающий. Ну или белая в 2 раза меньше...

Плюсы

1. Отличная команда, все чуть ли не друзья.
2. Позитивная атмосфера на работе.
3. Интересный опыт первопроходца.
4. Нет дресс-кода!

Минусы

1. Нет ментора или тест-менеджера — обучаешься сам на своих граблях, изобретаешь велосипеды.
2. Серая или черная зарплата.
3. Небольшая зарплата.
4. Хаос вместо процессов, а это не всем подходит.

Процессы на аутсорсе

Аутсорс[162] — это когда твоя компания сдает тебя в аренду другой компании. Когда проект заканчивается, забирает назад и сдает новой компании.

Тестировщик на аутсорсе

Кратко

1. На проекте мало людей, в компании много.
2. Бюрократия зависит от проекта.
3. Процессы заказчика, меняются.
4. Оформление заказчика.
5. Команда часто меняется.
6. Зарплата ой, зато «белая».

Подробнее

Бюрократия

Все сильно зависит от проекта, на который тебя поставят. Если это банк, то дресс-код, бюрократия и вот это вот всё. А на другом проекте можно приходить в чем хочешь, и бюрократии нет вообще. Рулетка, как повезет!

Процесс тестирования

И снова все зависит от проекта. Где-то процесс построен, где-то царит бардак. Где-то процесс гибкий, и вы можете попробовать что-то улучшить. Где-то будет жесткий: «прими и терпи».

Хорошая новость в том, что проекты обычно не длятся годами. Месяц поработали там, месяц тут. Так что если текущий процесс не устраивает, нужно лишь чуть-чуть потерпеть.

Команда

Ну вы поняли ;-) Одна команда будет большая, другая маленькая.

Иногда на аутсорс отдают сразу отдел тестирования. И тогда вы будете перемещаться между заказчиками со своими коллегами. Это морально проще. То есть будет меняться только заказчик, компания. Разные процессы, разный уровень бюрократии, но вокруг знакомые лица.

Обучение

Вот тут уже интереснее! Так как компании нужно вас продавать, она оплачивает обучение. Иногда это внешние курсы, иногда внутренние. Если компания специализируется на «поставке тестировщиков», то там, скорее всего, будут внутренние тренеры и тренинги.

Если компания небольшая: пара разработчиков, пара тестировщиков — скорее, оплатит внешние курсы. Полностью или частично, тут вопрос. Аналогично с конференциями. Компании выгодно.

Если компания распределенная, библиотека вряд ли будет. Да и книги за свой счет.

Зарплата

Крутой специалист всегда сможет продать себя подороже, но вообще на аутсорсе денег не слишком много. Почему? Потому что конкуренция и демпинг.

Есть ведь еще индусы, которые готовы тестировать за гроши. Да, они это делают плохо, зато дешево! А некоторые компании выбирают только по цене.

Если команда распределенная, то руководителю выгодно искать тестировщиков в небольших городах. Допустим, в Москве зарплата середнячка 60 тыс. рублей. А в Урюпинске вообще ИТ-контор нет, а средняя зарплата по рынку 15 тыс.

Тогда мы нанимаем человечка на зарплату 30 тыс. В своем городе он царь и бог, может многое себе позволить, поэтому доволен. Для нас он обошелся дешевле рынка в нашем городе, и мы тоже довольны. При этом заказчику мы его продаем по 60, получая неплохую прибыль.

Плюсы

1. Обучение — чтобы тебя продать, нужны корочки! Так что с обучением проблем нет.
2. Ментор — он бдит, что ты прокачиваешься.
3. Постоянная команда — если вас продают командой.
4. Много разного опыта: за короткий срок можно поработать и в банке, и в страховой, и т. д. Вы успеете увидеть разные процессы, пощупать разные инструменты. А это большой плюс в резюме!

Минусы

1. Низкая зарплата.
2. Постоянная смена проекта: это не всем нравится, что только привык, и тут другое место работы, новые люди...

Подведем итоги

То что я описала — не объемлет все возможные варианты. Может быть маленькая компания с дикой бюрократией. Может быть компания-гигант, где бюрократии нет вообще. Аналогично с зарплатами, обучением, наличием ментора и прочее.

Я показала вам стереотипы, чтобы вы примерно представляли себе, что вас ждет. Чтобы задумались, какие плюшки вам важны, а какие не очень. Это поможет сориентироваться при поиске вакансий, а также подскажет вопросы для собеседования. Вам же надо узнать, подойдет ли компания именно вам!

Лично я считаю, что для новичка лучшее место — большая контора или аутсорс. Там, где уже поставлены процессы, где тебя всему научат. Это будет гораздо быстрее, чем самому собирать все грабли.

В большой конторе вы сможете расти внутри одной

компании. И постоянно работать над одним проектом. Нет стресса от перемещения в новую команду каждые сколько-то недель или месяцев. И есть время действительно хорошо разобраться в проекте.

В аутсорсе вы сможете за короткий период времени перепробовать множество разных техник, методологий, инструментов. Увидите самые разные процессы, пообщаетесь с кучей разработчиков... Это круто! За пару лет узнаете столько, сколько другие не узнают никогда ;-)

Так что думайте, где бы хотели работать. И обязательно уточняйте на собеседовании все важные для вас пункты!

Домашнее задание: работа мечты

Подумайте, чего вы хотите. Какая она, компания вашей мечты? Как построен процесс? Будете вы единственным тестировщиком или одним из целой команды? Будет над вами тест-менеджер или нет?

Нужен ли вам ментор? Важно ли вам оплачиваемое обучение и посещение конференций? Критичен ли дресс-код?

Выпишите все, что считаете важным. И уточняйте на собеседованиях: «А как у вас?» Это поможет сделать выбор!

Компания мечты. Какой он, твой идеал?

Глава 12
КАК СОСТАВИТЬ РЕЗЮМЕ?

Ну всё, работу мечты придумали, теперь нужно ее найти! А чтобы искать работу, нужно составить резюме.

Что мы и сделаем в этой главе.

Составить самому или заполнить форму на сайте?

Самый простой вариант — пойти на сайт для поиска работы и заполнить там форму создания резюме. Именно это резюме будут просматривать работодатели чаще всего. Потому что они ищут по параметрам и смотрят резюме на сайте + вы ищете компанию по параметрам и отликаетесь на вакансии. И снова работодатель видит ваше резюме на сайте.

Сейчас самый популярный сайт — HeadHunter[163], он же HH. Есть еще LinkedIn, но он запрещен в России, так что им пользуются только в обход правил (через прокси, например).

Я учу своих студентов так:

— Резюме, выгруженное с HH, — отстой, сделайте отдельное. Да, понятное дело, что вы будете откликаться на HH на вакансии и будут видеть именно его. НО! Если работодатель просит вам выслать ему резюме, пришлите своё красивое, а не выгрузку с хедхантера.

И действительно, на HH ваше резюме смотрится нормально, да и эйчары[164] уже привыкли к его структуре и знают, куда смотреть. Но если сделать выгрузку, то у новичка с нулем опыта будет резюме на 3–5 листов. И куда это годится?

Открою вам страшную тайну: никому не интересно читать много буковок. Да и просто нет времени на это. У эйчара на одну вакансию прилетают десятки резюме. А если вакансия для новичка — еще больше. Он читает все резюме по диагонали.

Чтобы выделиться из толпы однотипных резюме, нужно:

- написать хорошее сопроводительное письмо (но об этом мы поговорим в следующей главе);
- сделать красивое короткое резюме, а не просто выгрузку с HH.

Поэтому я за то, чтобы делать резюме отдельно. Для новичка желательно на одну страничку. Но если прям много чего есть рассказать — на две. И тогда, если откликаетесь на сайте, работодатель видит резюме с сайта, но если вы пишете напрямую в компанию, то аттачите короткое и красивое резюме!

Однако мы недавно проводили опрос в чате выпускников — кто как нашел работу? Половина ребят — просто заполнив профиль на HH. Своё резюме им просто не пригодилось. Поэтому в первую очередь стоит прокачивать навык правильного заполнения резюме на HH или аналогичном сайте. Не стоит это недооценивать!

Но я рекомендую иметь про запас свой вариант. Говорю как человек, который проводил собеседования. Когда присылают короткое и хорошо оформленное резюме, это сразу бросается в глаза. И производит приятное первое впечатление.

Структура резюме

1. Навыки + технические скиллы.
2. Опыт.
3. Образование.
4. Ключевые слова.
5. Портфолио.

Причем именно в таком порядке. Да, если вы новичок, то у вас вполне может быть более-менее релевантный опыт, куча документов об образовании, но... В первую очередь вы должны рассказать о том, что вы умеете. То есть показать навыки.

Давайте рассмотрим каждый пункт отдельно.

Навыки

Пишите о том, что реально умеете делать. Если вы прочитали одну статью о том, как в JIRA можно создать проект и настроить его под себя, — это не значит, что у вас есть навык администрирования джиры!

Посмотрел бесплатное видео про создание первого автотеста и просто повторил за тренером? Это не значит, что вы умеете писать автотесты. Подключился к линукс-машине через WinSCP? Это не значит, что есть базовые знания Linux.

Конечно, написать в резюме можно все что угодно. И вы даже можете пройти отсев по резюме: работодатель увидит список того, что вы умеете, впечатлится и позовет на собеседование! Но достаточно будет пяти минут, чтобы понять реальный уровень ваших знаний.

Из собственного опыта...

Собеседую синьор-тестировщика. У него в резюме 3+ года нагрузочного тестирования. Ух ты! А мы на тот момент нагрузкой еще не занимались. Круто же, интересно. Начинаю уточнять:

— А тут написано, что 3 года нагрузочное проводите...

— Ну да.

— Расскажите, пожалуйста, как это происходит?

— Ну... (замялся). Там уже все написано было до меня. Так что я просто нажимал «Запустить» и смотрел на графики.

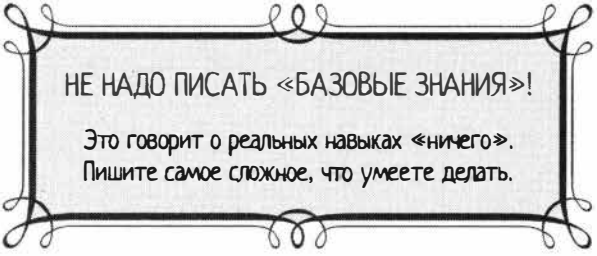
Вот сразу и виден реальный уровень. А как гордо звучит резюме! Несколько лет нагрузочного тестирования, это ж мегаспец! А на деле такое тестирование может и джун проводить...

При этом, если бы был реальный интерес у человека, он мог бы хотя бы разобраться, как работает тот скрипт, который он запускает. Как его можно написать, что именно он делает. Ведь если скрипт написан давно, его наверняка можно улучшить! Было бы желание...

Теперь главный вопрос — а как написать-то? Вот если книгу по SQL прочитал, это уже «базовые знания SQL»? Или еще нет? А если три упражнения на sql-ex[165] выполнил? А если первый join смастрячил?

Что не надо писать?

Правило простое:



Читал книгу или смотрел видео без практики? Так и пишем: «SQL — посмотрел три видео на YouTube». А потом задумываемся... Звучит не очень, правда? Всё верно, как только убрали абстракцию, поняли, что гордиться тут особо нечем. Убираем из резюме вообще. Это — не навык. Навык нарабатывается *только* на практике.

Что надо писать?

Думаем — что самое сложное по навыку умеем делать? Об этом и пишем:

- SQL — join трех таблиц;
- PHP — создал свой блог: **http://example.com/**.
- Java — прошел 15 уровней на Javarush.
- ...

Да, прохождение онлайн-тренажеров тоже считается. Это sql-ex по SQL, Javarush по java, и наверняка будет куча других, когда вы будете читать эту книгу. Если вы не можете сами оценить свой уровень —пройдите такие тренажеры и пишите про уровень в них.

Это будет вполне показательно. Тот же sql-ex весьма популярен, и работодатель может о нем знать. И есть разница между кандидатом, который прошел два задания, и тем, кто прошел 98.

Если можете доказать свои навыки — супер! Нет, я говорю не о сертификатах с курсов, о них мы поговорим в *разд. «Образование»*. Я говорю о портфолио, которое вы составляли на протяжении всей книги! Что лучше докажет навык, как не пример работы?

- Умеете писать тест-кейсы? Докажите! Приложите любой свой тест-кейс.
- Классно пишете чек-листы? Покажите свой лучший чек-лист!
- Изучали язык программирования? Создайте блог или простенькую программу, залейте на хостинг, выложите на github.

Если читая книгу по программированию, вы сразу будете думать о будущем резюме — это поможет лучше понять материал. Да, хорошие книги сами по себе являются тренажерами, в них есть ДЗ. Но самая лучшая практика — сделать что-то своё, а не по книге.

Может, на работе можно что-то заавтоматизировать? Продумайте задачу и решайте ее по мере чтения. Еще нет работы? Придумайте себе задание — например, сделать простенькое ПО типа задачи про треугольник[166]. Или создать свой сайт. Блог — самое простое решение.

А потом добавьте ссылку на сайт в портфолио. Или на исходный код, если приложение надо собирать локально. В любом случае это покажет навык, а запись «базовые знания языка программирования» не говорит ни о чем.

Опыт

Главным плюсом тут, разумеется, будет опыт в тестировании. Может быть, вы устроились на работу за копейки, зато теперь у вас полгода или год опыта набралось. Или вы работаете пока на прежней должности, не в ИТ, но тестируете на фриланс-бирже. Все это можно и нужно вносить в резюме!

Где начинающим тестировщикам получать опыт?

Все начинающие тестировщики задаются вопросом: где набраться опыта? Где применять полученные знания? А ведь способов-то много!

Работаем за еду, то есть за опыт...

Самый лучший способ получения опыта — устроиться на работу. Но поскольку такой совет из разряда «привет, капитан-очевидность», то этот пункт явно не из начала списка. На работу ведь не берут как раз из-за отсутствия опыта. Что же делать?

Можно работать бесплатно, получая опыт и проходя обучение на реальном проекте. Такие проекты, как правило, open source и времени просят всего по 6+ часов в неделю. Можно совмещать с основной работой, получая драгоценный опыт и не теряя свой заработок.

Погуглите следующие варианты, актуальные на сентябрь 2021 года:

- Open source проект, которому нужны тестировщики;
- Хомячки — проект, направленный специально на получение опыта начинающими.

Тестируем любимые сайты

На самом деле тестировать можно все что угодно. Но удобнее всего тестировать сайты, которыми вы сами пользуетесь. Так, например, я протестировала сайт кинотеатров Люксор, потому что пользуюсь им практически каждую неделю.

Важно: проводя подобное тестирование, помните правило регистрации по email, иначе никто вам «спасибо» не скажет!

Никогда не используйте «левые» email'ы для тестирования.

Чем это плохо?

1. Система на каждый такой «левый» адрес высылает почту, то есть фактически спамит mail.ru по несуществующим адресам. Mail.ru банит такого отправителя как злостного спамера. А менеджер проекта уже отрывает руки тому вредителю, благодаря которому их занесли в черные списки. Вы же не хотите быть таким вредителем, правда?
2. Тексты писем, которые отправляются пользователям, тоже надо тестировать. Если все время вбивать в поле всякую фигню, почту мы в итоге не получим и не сможем ее проверить.

На выбранном сайте вы можете провести два вида тестирования:

- функциональное;
- usability.

Остальные виды не рекомендую, особенно нагрузочное и тестирование безопасности — его нельзя проводить без разрешения владельца сайта, так что для этих целей ищите другие возможности. Да и потом от начинающего тестировщика не будут требовать умения тестировать нагрузку или безопасность (а если и будут, то бегите от этой конторы, потому что они не понимают всей сложности таких тестов!)

А вот функциональное тестирование надо уметь проводить. Это тестирование по документации, разбиение на классы эквивалентности, граничные значения... В теории все эти термины звучат очень легко, но вот с применением на практике возникают проблемы. Поэтому берем любой сайт и рассматриваем его с точки зрения тестировщика.

Многие начинают спрашивать: как же так, у нас же нет документации! Ребята, это же публичные сайты, они должны быть простые и понятные. И документацию мы знаем и сами. Если

это магазин, там должна быть функция добавления в корзину и заказа товара, возможно, сразу же и оплаты. Надо объяснять, как это работает? Нет, все и так понятно! Поэтому тестируем по тому, что уже есть.

А так как мы сами пользуемся этими сайтами, то это дает возможность проведения тестирования удобства использования (usability testing) — насколько удобно работать с сайтом. Все ли очевидно и понятно в его работе? Или приходится долго искать кнопку добавления товара в корзину, потому что у других она на виду, а тут практически не видна?

Таким образом, мы можем зайти на любимый сайт и:

- попрактиковаться в составлении тест-кейсов или чек-листов;
- попробовать применить знания о классах эквивалентности и граничных значениях на любой форме;
- выделить позитивное и негативное тестирование;
- проверить удобство использования и качество локализации (если сайт доступен на разных языках).

А потом у нас есть возможность попрактиковаться в составлении баг-репортов! Наверняка ведь какие-нибудь проблемы найдутся. Можно удовлетвориться тем, что «я молодец, нашел баги!». Но лучше всего (и для вас, и для разработчиков сайта) — дать обратную связь.

Для этого находим на сайте какой-нибудь email создателей сайта и пишем им письмо примерно такого содержания:

> Добрый день!
>
> Я начинающий тестировщик, учусь делать мир лучше! Вашим сайтом пользуюсь давно и часто и могу сказать, что он очень удобен с точки зрения простого пользователя! Однако я нашла в нем несколько ошибок.
>
> Так как я только учусь, ответьте мне, пожалуйста, — понятны ли вам мои сообщения об ошибках? Или приходится долго разбираться, что я хотела сказать? Были ли полезны эти сообщения? Эта информация очень важна для меня, поэтому буду весьма признательна за отзыв.
>
> Баг № 1. «Название»
> «Описание»
> Баг № 2. «Название»
> «Описание»

Если сайт не очень удобен для использования, то в письме также расписываем, что именно неудобно и как было бы удобно. Очень важно указывать ожидаемый вами результат!

Чтобы разработчики захотели вам ответить, письмо должно быть уважительным и понятным. То есть без наездов и огромной простыни сумбурного

текста без абзацев. И помните: при оформлении бага критикуйте проблему, а не людей!

И еще важное замечание: отсортируйте баги в письме по убыванию критичности. Допустим, вы нашли падение системы при попытке ввести корректный номер телефона во время бронирования билета и небольшую орфографическую ошибку где-то в лицензионном соглашении, которое никто никогда не читает. Если в письме будет отмечена сначала минорная ошибка (низкоприоритетная), не факт, что будут читать дальше: «Ха, ну есть там орфографическая ошибка, и что, все „баги“ такие?». Начинайте с важного!

Если разработчики ничего вам не ответили, не расстраивайтесь, такое бывает. Пробуйте дальше, стучитесь в новые двери. Кто-то даст вам обратную связь, еще и благодарен будет! А где-то, возможно, открыта вакансия тестировщика, и вас туда даже пригласят, если найденные ошибки будут хорошими и понятно описанными!

Если же баги не находятся и руки опускаются («Я никогда ничего не найду!»), то рекомендую обратить свой взор на баги локализации. Это когда приложение переведено на разные языки. Наверняка там что-то забыли! Название кнопки, текст на картинке... Ищите!

Тестируем мобильные приложения

Этот раздел очень похож на предыдущий. Только там предлагалось тестировать именно сайты, а здесь я хочу напомнить, что помимо компьютера настольного у многих есть еще и компьютеры карманные! Не стоит о них забывать. Наверняка есть приложения, которыми вы сами пользуетесь. Это может быть игра, календарь, таймер, органайзер — да что угодно!

И снова есть простор для тестирования: попрактиковаться в «как бы тебя сломать?» и посмотреть на приложение с точки зрения пользователя, но умеющего обосновывать свои предложения по улучшению. В таких приложениях часто есть функция «напишите нам» — туда можно отправлять feedback.

Вот, например, я на iPad часто пользовалась приложением myEnglish, туда и писала. Здесь действуют все те же правила уважительности и понятности. Когда я написала гневное письмо: «Да вы офигели, почему теперь все такое убогое?!», то оно было проигнорировано.

А когда я смирилась с кардинальным изменением дизайна и написала не содержащее наездов письмо с предложениями по улучшению и описанием багов, в котором было понятно все расписано: «Вот тут не очень хорошо, потому что я, как пользователь, ожидаю то-то. Исправить можно так-то или так-то», то мне ответили и поблагодарили за баги, а также внесли предложенные мной улучшения в игру! Вот вам и фидбек!

Так что тестируйте и пишите, пишите разработчикам. Вы прокачаете свои навыки нахождения и описания багов, а они улучшат свои приложения!

Тестируем «тестовые» сайты

Есть сайты, специально созданные для тестирования. Ну или вынесенные в общий доступ демостраницы.

Ищите тестовые сайты в разделе «Test IT»[167] портала Testbase!

Работаем на фрилансовых биржах

Плюсы: можно получить деньги за найденные баги. Ну и опыт на реальных проектах!

Минусы: нужно более-менее владеть английским и на нем оформлять баги (и оформлять нормально и читабельно!). Чтобы зарабатывать хоть сколько-то, нужно сначала наработать репутацию. Для новичков открываются проекты, которые тестируются сразу сотней людей, — какой шанс, что вы первый найдете багу? А вот наработаете репутацию, и вас станут приглашать на тестирование в ограниченном кругу.

Вот список таких бирж:

- www.upwork.com;
- www.free-lance.ru;
- www.utest.com;
- www.fixber.com.

Получаем опыт на тренингах

Хороший тренинг — это много практики. Такой, что будет ждать вас потом на реальной работе. Но под присмотром куратора, который подскажет вам, как и что лучше исправить.

Устраиваемся на позицию junior

Вот круг советов и замкнулся... Самый «капитанский» совет: лучше всего получать опыт уже на работе, там же расти и развиваться. Проблема в том, что туда еще надо устроиться, и желательно к этому моменту уже хоть что-то да уметь... Но для этого у вас есть целых пять предшествующих подразделов! Так что удачи! ;-)

Что не надо писать?

Опыт с курсов: «Как работал в такой-то компании». Например, в свое время мы давали студентам тестировать Дадату. Да, это приложение разрабатывает моя компания. Но для студентов у нас было 10 задач, которые затрагивали функционал лишь поверхностно.

Подобный опыт можно записать так: «Тестировал такой-то сайт во время такого-то курса», но не как работу в компании. Вы общались с разработчиками, аналитиками? Вы придерживались графика компании? Вы вообще хоть что-то знаете о компании, кроме того, что они сделали такой сайт?

Обычно ответ на все эти вопросы «нет», но студентам хочется указать курс как опыт. А для указания опыта нужно написать компанию, в которой работал...

Не надо так! Это легко проверяется, да и на собеседовании быстро узнают, что в самой компании вы не работали, только «щупали» ее проект.

Если тестил сайтик «для себя» — это тоже не работа в компании. Пишем: «Тестировал такой-то сайт для саморазвития, результаты вот (ссылочка)», но тут даже срок особо не укажешь. Конечно, хочется гордо написать: полгода или год. Но... Что ты делал целый год?

Другое дело — работа в компании. Там да, каждый день по 8 часов тест-кейсы, летучки, всё такое... А когда тестируешь для себя, для опыта это слишком долго: возникают вопросы, а что он делал-то? Или вы один раз 10 тест-кейсов написали и раз в месяц их прогоняли (типа регресс)? В общем, убираем время из этого блока. В НН, если время работы указать надо, переносим работу для себя в другой блок.

Что надо писать?

Релевантный опыт

Тут все просто: если вы работали в тестировании, пишите об этом! Это самый большой плюс!

Если вы работали не в тестировании, но тоже в ИТ — пишите об этом:

- навыки сисадмина помогут быстрее решать задачи с автоматизацией рутины;

- навыки программиста (да, да, бывает, что и оттуда в тестирование уходят!) помогут лучше понимать, где могут быть баги, + проводить ревью юнит-тестов + писать автотесты;
- навыки аналитика помогут выявлять слабые места в ТЗ задолго до разработки.
- ...

Работа на линии техподдержки — вообще отличный опыт для тестировщика! Тут вы можете пощупать свой продукт и поговорить с реальными пользователями. Услышать их боль и взглянуть на своё ПО другими глазами.

У меня на одном из проектов запросы от пользователей принимал тестировщик. Он являлся первой линией техподдержки.

Такая практика — это очень круто! Ты уверен, что тут все понятно, а тебе задают кучу вопросов по функционалу: ага, ясно понятно, ставим запросы на улучшения. Видишь, какие баги всплывают во время прохождения инструкций? Упрощаешь инструкции, автоматизируешь работу админов... Это познавательно и интересно!

Нерелевантный опыт

А как быть, если опыт вообще нерелевантный? Бармен, юрист, бухгалтер, официантка... Писать или не писать?

Я за то, чтобы писать, но кратко. Не надо на 10 страниц расписывать все свои обязанности на всех подработках. И если их правда много, можно вообще объединить в один пункт.

Вместо:

Январь 2020: Официантка в кафе «Мороженка»
Февраль 2020: Официантка в кафе «Карамель»
Март 2020: Курьер службы «Бегунки»
...
Июль 2021: Менеджер на ресепшене клуба «Ноготки»

написать:

Январь 2020 — Июль 2021
Мелкие подработки (официант, курьер, менеджер)

Но ваша работа может быть плюсом. Например, бухгалтер — отличный опыт! Полно программ, которые разрабатываются именно под бухгалтерию. И вас там с руками оторвут, потому что вы будете не просто проверять, что «все работает, как сказано в ТЗ», вы будете тестировать само ТЗ и проводить проверки осмысленно!

Я в свое время тестировала похожую программу. Вот только для всех тестировщиков слова «ручные и автоматические корректировки» были пустым звуком. Просто некой сущностью, которая описана в ТЗ. Это сейчас я понимаю, как было бы круто посмотреть на ПО со стороны пользователя. Но тогда знаниями по бухгалтерии никто не заморачивался.

У меня на курсе для новичков через курс есть хоть один бухгалтер. Надоедает профессия или конкуренция демпингует... В любом случае, учтите, что такой опыт — отличный опыт!

Особенно если стучаться в интересные вам фирмы, где ваши знания пригодятся. Рассылайте резюме в компании, которые делают ПО для бухгалтерии, и этот опыт выделит ваше резюме из остальных.

Так что пишите про свой опыт! И ищите в первую очередь те конторы, в которых нерелевантный для ИТ опыт может пригодиться.

А как же достижения?

Есть разные мнения:

- нужно обязательно писать о своих достижениях;
- это вообще никому не нужно, потому что: «А что туда писать? Усердно работал?».

Конечно, если у вас есть достижения, которыми вы гордитесь, — напишите о них. Особенно это важно для руководящей должности: что вы уже успели внедрить? Чего достигли на прошлом месте работы?

Если вы не руководитель — вы все равно могли чего-то достичь. Например, неначинающий тестировщик мог:

- построить отдел тестирования с нуля;
- настроить автоматизацию с нуля;
- ввести ревью кода и покрытие юнит-тестами (решается переговорами с разработчиками, но если до вас этого не было — достижение!)
- сдать проект за неделю;
- ...

Я встречала статьи, высмеивающие этот подход. Что-то из серии «5 тупых вопросов на собеседовании и остроумные ответы на них»:

— Почему именно наша компания?

— Да мне начхать, я рассылал резюме во все места подряд. Если выберете меня, тогда и подумаю, окей вы или нет.

— Кем вы себя видите через пять лет в нашей компании?

— Никем, только дураки на одном месте столько работают.
— Чего вы достигли на прошлой работе?
— Работу работал! Некогда мне вашими достижениями заниматься.

Мне не очень нравится такой подход. Я работаю в одной компании 6 лет[168], рассылала резюме не во все места подряд, у меня есть некие достижения. Я не считаю эти вопросы тупыми, а приведенные ответы шибко умными.

Но я осознаю, что у простого тестировщика достижений может и не быть. «Работу работал» — это тоже полезно. Более того, иногда нужны именно люди без амбиций. Кому-то ведь нужно выполнять рутинные задачи, верно?

Так вот, люди все разные. Кто-то может пять лет заниматься ручным тестированием по готовым тест-кейсам, и ему будет норм. Потому что работа для него — средство заработка, и пофиг, что там делать. Или потому, что он ловит кайф в монотонной работе. Или... Ну я не знаю причин, потому что я сама другая! Но наличие разных типов людей признаю ;-)

Отсюда вывод: писать или не писать достижения на всех работах, каждый решает для себя. Если вы их приводите, это тоже дает работодателю некую информацию о вас как о человеке. Ставите ли вы перед собой «большие» цели или любите «работу работать».

А я в любом случае рекомендую задуматься: вот если надо было бы заполнить этот раздел, что бы вы туда вписали? А что хотели бы вписать? Осуществимо ли это? А то, может, на текущей работе есть куда развиваться, но вы не видите леса за деревьями. А наметите себе цель, и работать станет интереснее!

Образование

Тут все просто:

1. Указываем институт.
2. Указываем курсы, которые проходили.

Если указываете курсы, то название курса делайте ссылкой на свой сертификат (при наличии электронного сертификата).

Как насчет чтения книг? Стоит ли об этом вообще упоминать? Зависит от вас и работодателя. Я как работодатель считаю плюсом, когда человек прочитал хотя бы пару книг по профессии. Об этом можно написать в разделе «Остальное» или «Образование»:

Прочитал книги:

- «Тестирование Дот Ком» Романа Савина;
- « Что такое тестирование. Курс молодого бойца» Ольги Назиной;
- ...
- + читаю Хабр и блоги, смотрю видео на YouTube.

Не надо упоминать все-все-все прочитанные книги, но о тестировании — можно и нужно. Почему? Потому что многие ничего не читают! Говорю как человек, проводивший собеседования не только джуниоров, но и людей с 5-летним опытом:

— Книги какие-нибудь читали?
— Нет.

— Блоги, статьи?
— Нет, не особо.

Или другой диалог:

— Книги какие-нибудь читали?
— Да!
— Какие?
— Ну эту... Синенькую такую... Не помню автора.
— Хорошо, тогда сделайте задачку на треугольник
— Ээээ, бе ме...

Недостаточно просто читать книги. Нужно ведь из них хоть что-то извлекать! Именно поэтому я даю в конце каждой главы упражнения. Это ваш выбор: делать их или листать дальше. Но если не делать, то потом будут «бе» и «ме» на собеседовании, потому что просто прочитанное усваивается плохо. Нужно пробовать.

И, тем не менее, прочитать книгу лучше, чем не прочитать. И для меня, как для книголюба, ссылка на прочитанные книги в резюме — это плюс кандидату. Особенно джуниор-кандидату!

Не обязательно проходить какие-то курсы, можете заниматься самообразованием. И обязательно пишите в резюме, каким.

Ключевые слова

Посмотрите вакансии для тестировщиков в вашем городе. Найдите там ключевые слова и используйте у себя, если есть соответствующий навык.

Учтите, что в некоторых компаниях ваше резюме вначале просматривает HR. Он не разбирается в ИТ, не знает, что именно значат слова SQL, Java, тест-кейсы... Ему сказали искать такие-то навыки, вот он и ищет. По чек-листу ключевых слов. И если в вашем резюме их не будет — резюме не пройдет отбор.

Портфолио

Если вы работали с этой книгой, а не просто читали ее — у вас уже должно быть готово портфолио ваших работ. Добавьте его в резюме! Для этого залейте куда-нибудь в гугл, яндекс-диск, дропбокс и дайте ссылку.

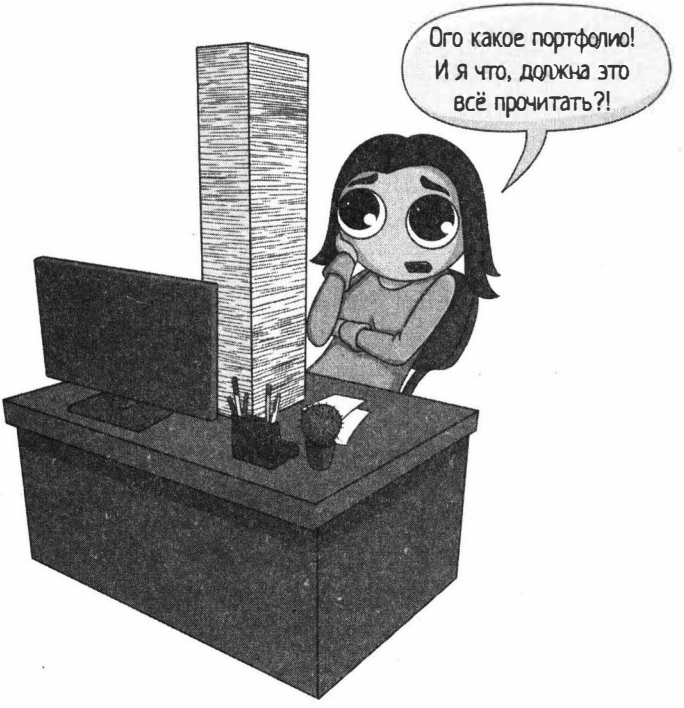
Что добавлять в порфтолио? В первую очередь:

1. Чек-лист функционала.
2. 1–2 тест-кейса, можно разных по стилю оформления.
3. Парочку багов, самых заковыристых, что вы находили!

Потом уже все остальное: можно сделать диаграмму State Transition, нарисовать таблицу решений... Но лучше это сложить отдельно, чтобы не мешало.

Цените время людей. Я понимаю, что вы старательный ученик и сделали уже 20 тест-кейсов, несколько чек-листов, нарисовали кучу табличек и дико горды своими усилиями! Но. Люди просматривают десятки резюме

в день, у них иногда само резюме нет времени прочитать, а уж 100 листов дополнительной литературы они точно изучать не станут.



Поэтому ваше портфолио должно занимать 2–3 листа. На одном чек-лист, на другом лучшие тест-кейсы, на третьем баги. Если у вас ссылка на гугл-диск, можно добавить еще папку «остальное», куда запихать все, что захочется. Если работодателю будет любопытно, он посмотрит разные работы. Если нет — только основное.

Папка «остальное» не должна быть свалкой. Структурируйте ее, чтобы сразу было видно, где чек-листы, где тест-кейсы, где баги, где таблицы решений... Не вперемешку! Иначе это будет говорить о вашей аккуратности не в лучшую сторону...

Остальное

Так, о чем еще не поговорили? О софт скиллах. Не надо писать в резюме:

- Аккуратен.
- Трудолюбив.
- Усидчив.
- ...

Это само собой разумеется. Никто не пишет в резюме «Ленив, делать ничего не хочу и не буду, неаккуратен, неопрятен...». Хотя иногда шуткуют на эту тему, но в резюме реальном так все же не пишут.

Особенно забавно читать «грамотный», когда в резюме есть пара опечаток. Или «пунктуален», когда кандидат опаздывает на собеседование...

Получается текст ради текста, который не несет никакой смысловой нагрузки. Зачем он нам? Выкидываем!

Что еще НЕ писать:

- Знание Word.
- Знание Office.

И тому подобное. Это явно слизано с каких-нибудь резюме не ИТ-людей. Когда от бухгалтеров, секретарей стало требоваться уметь работать с MS Office, эти пункты давали преимущество. Но пардон, вы устраиваетесь в мир ИТ! Тут НЕ знать про офисные программы — моветон.

При этом опытный человек может вас легко затроллить на собеседовании, задав пару наводящих вопросов по Word. Про нестандартные фишечки — и вот будет облом удивленно хлопать на собеседующего глазками!

Умение открыть ворд и что-то в нем написать, чуток поменяв шрифты и цвет текста — это не навык. Если вы действительно продвинутый пользователь ворда и знаете кучу тонкостей — можно написать об этом. Только — см. *разд. «Навыки»* — напишите прямо в резюме то, что умеете делать:

- MSWord: расстановка буллитов по тексту.

Вот, пожалуй, и все!

Подведем промежуточные итоги

В резюме указываем:

1. Навыки + технические скиллы.
2. Опыт.
3. Образование.
4. Ключевые слова.
5. Ссылку на портфолио.

Но учтите, поток резюме на одну вакансию в хорошей компании большой. А уж поток резюме для начинающих — очень большой! Ну нет ни у кого желания и возможности читать опус на пять листов А4, честное слово. Поэтому помните: краткость — сестра таланта!

Хотя в чатах тестировщиков я иногда читаю возмущения на тему «А у меня 10 листов, и я считаю что это мегакруто». Ну, считай. Но, во-первых, это уже не джуниор, раз ему есть что написать. А во-вторых, меня терзают сильные сомнения на тему крутости такого резюме.

Я сама проводила собеседования. И среди новичков, и среди неновичков. Так вот, когда у меня было резюме на 5+ листов, я читала его бегло по диагонали, а потом на самом собеседовании просила кандидата:

— Расскажите, пожалуйста, подробнее про свой опыт. А тут что делали? А там?

Разумеется, даже если я успела прочитать резюме про опыт работы в какой-то компании, я все равно задам уточняющий вопрос. Потому что такие вопросы помогают увидеть, как человек думает, чем гордится и что ему важно на самом деле.

Пару раз кандидат возмущенно отвечал, что все это описано в резюме. Но я хочу узнать то, что в резюме не описано.

Подстройка резюме под вакансию

Как это работает в идеале?

1. Вы нашли компанию, в которую хотите устроиться.
2. Изучаете ее требования к кандидату.
3. Подстраиваете своё резюме под эти требования.
4. Отправляете его в компанию.

Как подстроить резюме? Просто перепишите его в словах вакансии! Например, у нас есть резюме:

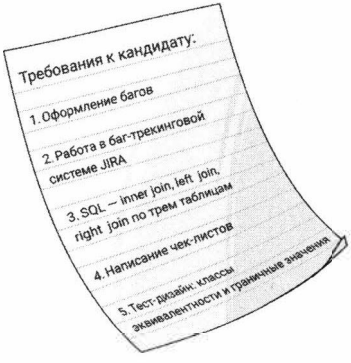
Навыки:

1. Создание тест-кейсов.
2. Создание чек-листов.
3. Составление баг-репортов.
4. Применение классов эквивалентности.
5. JIRA — создание workflow.
6. SQL — простые join.

Видим вакансию:

Требования к кандидату:

1. Оформление багов.
2. Работа в баг-трекинговой системе JIRA.
3. SQL — inner join, left join, right join по трем таблицам.
4. Написание чек-листов.
5. Тест-дизайн: классы эквивалентности и граничные значения.



Меняем свое резюме:

Навыки:

1. Оформление багов.
2. Работа в баг-трекинговой системе JIRA.
3. SQL — inner join, left join, right join по четырем таблицам.
4. Написание чек-листов и тест-кейсов.
5. Тест-дизайн: классы эквивалентности и граничные значения.
6. JIRA — создание workflow.

Что мы, по сути, сделали?

1. Поменяли порядок пунктов — подстроив под вакансию.
2. Изменили формулировку. Не «Составление баг-репортов», а «Оформление багов». Мы понимаем, что это одно и то же, а HR может смотреть на полное соответствие.
3. Раскрыли свое знание SQL. Оценили вакансию, поняли, что мы знаем все, что там нужно, переписали пункт под вакансию. Теперь в резюме есть ключевые слова, по которым могут делать отбор.

Подредактировав резюме, отправляем его работодателю. И ждем фидбек. Нууууу, это в идеальном мире. А на самом деле в ИТ не так!

Из собственного опыта...

Я начала работать в тестировании с 17 лет, это моя первая работа, поэтому не знаю, как в других сферах. Но обсудила с коллегой Юлей:

— (Юля) Нужно учить ребят, как адаптировать своё резюме в HH под конкретную вакансию.

— (Я) Да я сама этого не знаю ;-) Это как-то странно, человек обычно откликается на десяток вакансий. Вот он адаптировал под одну, и чего — ждать, пока компания просмотрит профиль, и потом уже адаптировать под другую?

— (Юля) Ну, вообще, да... Просто мы в IT балованные ;-) В среднем, когда тестировщик с опытом открывает свое резюме, он получает сразу штук пять предложений и в течение недели еще штук пять. В других работах не так ;-)

Реалии таковы, что вы не будете ждать фидбека от одной компании, а станете стучаться в разные двери. И это правильный подход. Некоторые

отвечают спустя день-три-неделю, потом собеседование, возможно, тестовое задание... А если отказ, то так по кругу — это можно очень долго работу искать.

Конечно, логичнее разослать резюме в разные компании. Я рекомендую сначала повыбирать и написать в те компании, куда вы очень хотите попасть. Это можно сделать по всем правилам: подстроить резюме под вакансию и скинуть отдельным файлом, написать красивое сопроводительное письмо... А потом уже откликаться на все вакансии подряд, мало ли какая выстрелит. Работа-то обычно нужна вот прям щас ;-)

И да, есть отличный лайфхак подстройки резюме под вакансию: сделайте отдельное от HH резюме! Его можно раскопипастить, чуток подредактировать каждую вакансию и отправить. Как это сделать?

1. Найти сайт компании, отправить резюме по указанным там контактам (и это уже плюс вам — значит, вы не поленились найти сайт компании, а не просто ткнули «откликнуться» на HH!).
2. Отправить ссылку на гуглодок или гуглодиск в HH. Только будет большой фейл, если ссылка окажется недоступна (а в резюме наверняка при этом написано «внимателен и аккуратен»).

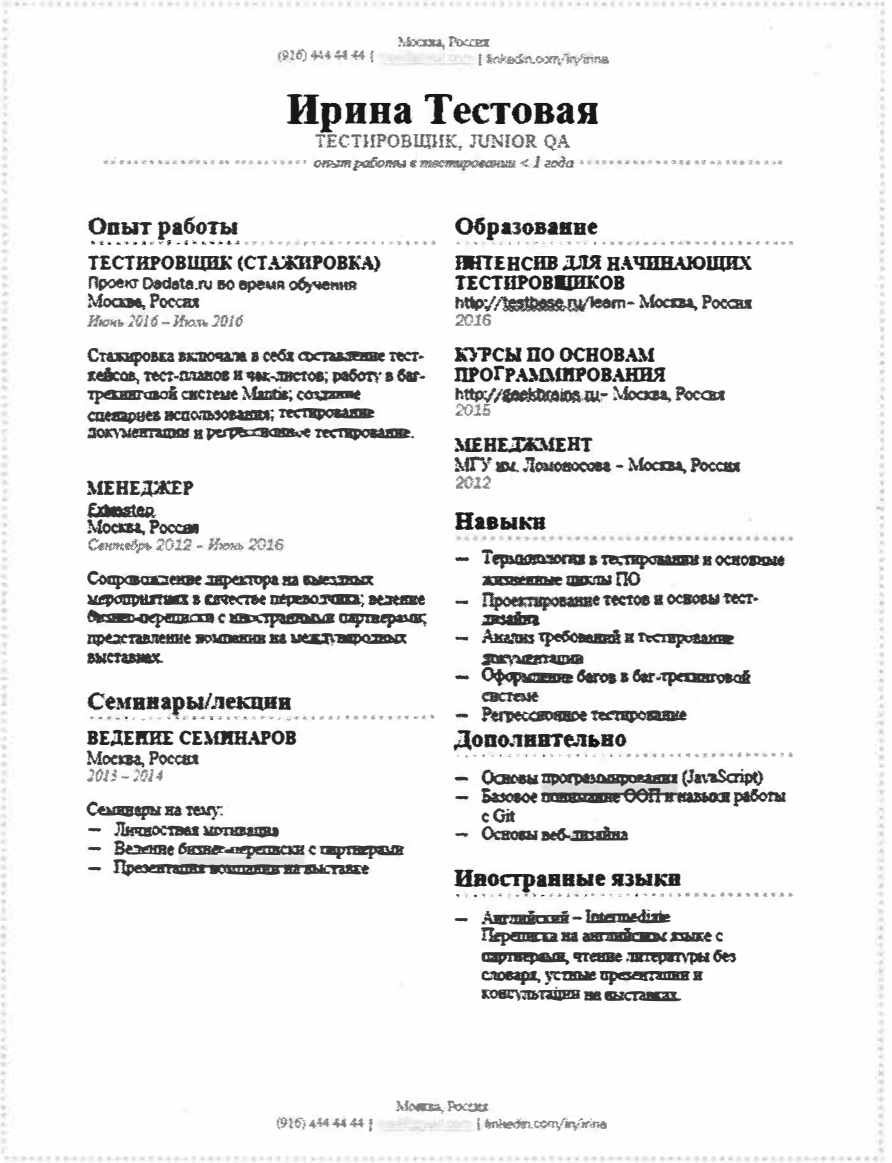
Да, это сложнее, чем просто 20 раз нажать «откликнуться». Но вам шашечки или ехать? Да, у меня в чате выпускников есть случаи, когда студент устраивался сразу после курсов. Иногда даже с первого собеседования!

Однако это умение подать себя + доля везения. Есть люди, которые находили работу через полгода или даже год поисков! Ну так что, это по-прежнему так сложно — потратить дополнительные пять минут на один отклик? Возможно, именно эта доработка резюме выделит ваше из толпы, и вы сможете получить работу!

Шаблоны для резюме

Ссылку на шаблоны вы всегда можете найти на моем портале Testbase.ru в разделе про навыки составления резюме[169]. Там они могут быть даже актуальнее, чем в книге. Вдруг после выхода книги мне интересные примеры скинут!

Вот, например, как может выглядеть резюме новичка — все влезло на одну страничку!



Москва, Россия
(916) 444 44 44 | [redacted] | linkedin.com/in/irina

Ирина Тестовая

ТЕСТИРОВЩИК, JUNIOR QA

опыт работы в тестировании < 1 года

Опыт работы

ТЕСТИРОВЩИК (СТАЖИРОВКА)
Проект Dadata.ru во время обучения
Москва, Россия
Июнь 2016 – Июль 2016

Стажировка включала в себя составление тест-кейсов, тест-планов и чек-листов; работу в баг-трекинговой системе Mantis; создание сценариев использования; тестирование документации и регрессионное тестирование.

МЕНЕДЖЕР
Extrastep
Москва, Россия
Сентябрь 2012 – Июнь 2016

Сопровождение директора на выездных мероприятиях в качестве переводчика; ведение бизнес-переписки с иностранными партнерами; представление компании на международных выставках.

Семинары/лекции

ВЕДЕНИЕ СЕМИНАРОВ
Москва, Россия
2013 – 2014

Семинары на тему:
- Личностная мотивация
- Ведение бизнес-переписки с партнерами
- Презентация компании на выставке

Образование

ИНТЕНСИВ ДЛЯ НАЧИНАЮЩИХ ТЕСТИРОВЩИКОВ
http://testbase.ru/learn - Москва, Россия
2016

КУРСЫ ПО ОСНОВАМ ПРОГРАММИРОВАНИЯ
http://[redacted].ru - Москва, Россия
2015

МЕНЕДЖМЕНТ
МГУ им. Ломоносова - Москва, Россия
2012

Навыки

- Терминология в тестировании и основные жизненные циклы ПО
- Проектирование тестов и основы тест-дизайна
- Анализ требований и тестирование документации
- Оформление багов в баг-трекинговой системе
- Регрессионное тестирование

Дополнительно

- Основы программирования (JavaScript)
- Базовое понимание ООП и навыки работы с Git
- Основы веб-дизайна

Иностранные языки

- Английский – Intermediate
  Переписка на английском языке с партнерами, чтение литературы без словаря, устные презентации и консультации на выставках.

Москва, Россия
(916) 444 44 44 | [redacted] | linkedin.com/in/irina

Сопроводительное письмо

Вы можете просто откликаться на все вакансии подряд со стандартной копипастой: «Я считаю, что мои навыки вам очень подойдут, резюме в аттаче». А можете написать сопроводительное письмо, которое увеличит ваши шансы на собеседование.

Хочу еще раз напомнить: кандидатов на позицию джуниор-тестировщика ОЧЕНЬ много. Кому-то опостылела текущая работа, у кого-то друзья в ИТ, кто-то пошел за большой зарплатой... В любом случае, это крутых тестировщиков с опытом разбирают за пару часов. А джуниоров выбирают.

Давайте разбираться, как увеличить наши шансы на успех. Каким должно быть сопроводительное письмо, а каким — не должно.

Главное правило сопроводительного письма

Я серьезно.

Пару лет назад я читала статьи про HR, которые просто копипастят один и тот же текст всем кандидатам: «Здравствуйте! Мы такие классные. Приходите к нам!».

Помню обсуждения в кулуарах на таких конференциях, как SQA Days. Все сходятся во мнении, что такие рассылки можно и нужно игнорировать. Я помню высмеивание этих сообщений: «Ну видно же, что текст не лично для меня, а стандартный. Обратитесь ко мне, и тогда больше шансов, что меня заинтересует ваше предложение».

Согласитесь, приятнее получать от HR сообщение, по которому видна заинтересованность именно в вас. Тогда почему вы пишете сопроводительные письма тем же методом, который недавно высмеивали? Как реагируют на такие сопроводительные письма? Правильно, никак. Словно и нет этого письма.

Стандартные ошибки сопроводительного письма

Рассылка на всех

Лучше вообще не писать сопроводительное письмо, чем писать стандартное:

> Добрый день!
>
> Меня заинтересовала Ваша вакансия, размещенная по адресу: (приводится адрес).
>
> Предлагаю Вам ознакомиться с моим резюме по ссылке: (приводится ссылка).

Откройте сайт компании, в которую хотите попасть. Изучите его, изучите навыки в их вакансии. И напишите в письме, почему вы хотите именно к ним и почему именно вы подходите на эту должность. Даже если все, что вас привлекло, — это зарплата, пишите о том, почему вы ее достойны.

Напишите 3–4 нормальных сопроводительных письма в те компании, куда реально хотите попасть. А сотню рассылочных оставьте для «количества», чтобы потренироваться ходить на собеседования. Но если вы, правда, заинтересованы в вакансии, потратьте пять минут, напишите личное обращение, покажите свой интерес.

Повтор резюме

Не имеет смысла просто копировать часть резюме в сопроводительное письмо. «Спасибо, читать мы и так умеем». Если хочется выделить конкретные навыки, то обоснуйте, зачем вы об этом пишете именно нам (табл. 12.1) или это очередная рассылка?

Таблица 12.1. *Примеры хороших и плохих формулировок для сопроводительного письма*

Плохо	Хорошо
Я умею писать тест-кейсы, составлять чек-листы, формировать баг-репорты...	У вас на первом месте стоит грамотное оформление баг-репортов. Я отправляла разработчикам игр описание багов, и они благодарили меня за них, хваля оформление. Аттачу последнюю переписку с большим спасибо. И хотя я только новичок без «реального опыта», зато у меня уже есть несколько исправленных багов! Очень надеюсь, что и вам смогу пригодится ;-)
Знаю Java, Python, JavaScript, SQL, MySQL...	С Java знакома только по книжкам, а вот по SQL прошла 70 упражнений на sql-ex.ru! Прошла за 3 дня, я быстро учусь, так что смогу быстро начать приносить пользу вашему проекту!

Орфографические ошибки

Без комментариев.

Принижение себя

Когда люди откликаются на вакансии, для которых не подходят, они так и пишут в сопроводительном: я то не могу, это не умею. Возникает вопрос: а зачем вообще откликался тогда?

Может, не стоит откликаться на вакансии, куда не подходишь «целиком»? Нет, стоит. Не бывает кандидата, который подходит на 100%.

Возможно, вы идеально подходите по всем пунктам, но вот SQL, например, никогда не щупали. На работе не нужно было, а самостоятельно не добрались. Откликаться? Да! И писать позитивно:

> Я быстро учусь — на прошлой работе закончила испытательный за неделю вместо 3 месяцев! Ставила баги лучше опытных сотрудников, и меня сразу взяли в штат. Так что смогу быстро влиться в вашу команду и приносить пользу.
>
> Я отлично подхожу вам по всем пунктам, кроме SQL. С ним никогда раньше не сталкивалась, на текущей работе в базу тестировщиков не пускают. Начала разбираться: читаю книгу «Изучаем SQL» Линн Бейли, погуглила доп. материалы, смогла написать простой select! Уверена, что быстро смогу достигнуть необходимого для работы минимума.

Видите? Вроде указали, что SQL не знаем, но:

- обещаем быстро разобраться (и тут недостаточно обещать, придется реально разбираться, даже если учить его ночами. Обещал? Делай!);
- показываем, что уже начали что-то делать: нагуглили статьи и написали первый селект. А то так можно книгу читать полгода и к базе не прикасаться, пока не дочитаешь. А профит-то где?
- сначала пишем в позитивном ключе: если в вакансии писали, что нужен быстро обучающийся кандидат, хвалимся своими успехами, если они есть. Даже если вы не работали в тестировании, у вас могут быть подходящие результаты: взяли в штат раньше конца испытательного срока, доверили ведение клиента через неделю вместо двух и так далее.

Но и про минусы тоже пишем, обязательно. Ведь, может быть, компании этот пункт настолько важен, что они не готовы вас взять. Но ваше сопроводительное не должно напоминать письмо нытика «Ничего не могу и не умею». Пишите о том, чем вы подойдете. А потом обещайте нагнать то, чего пока нет.

Как написать письмо?

1. Погуглите компанию.
2. Изучите ее сайт.
3. Почитайте статьи на Хабре.
4. Напишите для них, а не абстрактно. Без избитых фраз.

Пример хорошего письма

Вот пример того, как хорошо написанное сопроводительное письмо может привести к приглашению на собеседование даже при не очень хорошем резюме. Письмо публикуется с согласия автора, который уже давно и успешно работает в компании :-)

> Прочитав тексты ваших вакансий здесь и на **http://hflabs.ru/career/**, очень обрадовалась: вот, вот оно! именно то, что мне нужно! И те, кому пригожусь именно я!
>
> Дальнейшее изучение ЖЖ несколько отрезвило. [Что тут за ЖЖ? Пож., уточните.] Да, мне за 30, и я до сих пор стремлюсь к обучению, и я просматриваю

вакансии раздела «без опыта», потому что совесть не позволяет назвать себя квалифицированным специалистом... Исходя из всего этого, я вряд ли дождусь приглашения на собеседование :-)

Но на вакансию все же откликнусь. Во-первых, потому что верю в свои способности, пусть и нереализованные по разным причинам своевременно :-) А во-вторых, верю в неслучайность некоторых совпадений. Сегодня утром, до того как увидеть эту вакансию, кидала ссылку на Фактор своему папе, который долго бился в своей гос. организации с аналогичными проблемами средствами VBA ;-)

Это письмо короткое, что тоже важно — не надо сильно растекаться мыслью по древу. И оно эмоциональное, за счет чего и цепляет. Конечно, если убрать ссылки на места с вакансиями, упоминание ЖЖ и Фактора (продукт компании), то вроде как можно рассылку таким письмом делать.

Но оно было именно для нас. Потому что у нас в вакансии было про стремление к обучению. Да, многие это любят, но далеко не все указывают в вакансии. Так что это письмо может быть как массовой рассылкой, пусть и чуть лучше, чем стандартное «я вам подойду», а может быть про текст вакансии.

Я очень рекомендую вам делать личные сопроводительные письма. Пусть лучше их будет немного, зато качественные!

Подведем итоги

- Заполните резюме на HeadHunter — его увидят работодатели + вы сможете откликаться на вакансии.
- Сделайте отдельную «красивую» версию в PDF и именно ее высылайте по запросу.

- Выберите компанию, в которой хотели бы работать.
- Изучите сайт этой компании — чем они занимаются?
- Напишите сопроводительное письмо «про них»: про их сайт, про их вакансию. Не дублируйте свое резюме, но напишите, чем можете быть им полезны.
- Повторите все это несколько раз — для всех компаний, куда очень хотели бы попасть.
- Откликнитесь на HH на все остальные вакансии — ковровая бомбардировка, мало ли где выстрелит? Откликайтесь даже на вакансии, где просят полгода-год опыта, иногда и туда берут хороших новичков (и хорошее сопроводительное письмо вам в этом очень поможет!)

А вот три совета от моего коллеги Павла Абдюшева (тестировщик, аналитик, тренер и просто умный человек):

1. Формат HH по-прежнему один из самых отстойных. А посему всегда старайтесь отправить нормальное резюме, даже если откликаетесь в HH. Просто приложите резюме в читаемом виде.
2. Пишите нормальное сопроводительное письмо, краткое и по делу — чем именно вы будете полезны для этой компании, проекта, позиции. (ДА, надо уделить хотя бы полчаса ознакомлению с тем, куда вы откликаетесь.) 90% писем: «Я тут нашел вашу вакансию, мне интересно, думаю, что полностью подхожу для нее, возьмите меня, спасибо за уделенное время» — вот хочется сразу в топку отправить такие отклики на аналитика.

Еще все любят в сопроводиловке писать, какие они неумехи, но могут научиться быстро и круто. Вот это мне вообще непонятно: ни слова о плюсах, зато каждый второй обязательно напишет, чем именно он не подходит. Не надо так делать.

3. Даже HH-шное резюме можно сделать хорошим, грамотно описав, чем вы занимались и чего достигли на каждом месте работы + какими навыками и в каком объеме вы владеете. Не «базовые знания SQL», а какой самый сложный запрос умеете писать.

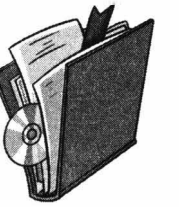
Павел написал эти советы для моих выпускников после массового просмотра откликов на нашу вакансию тестировщика. Прислушайтесь к советам человека, который проводит отсев по резюме!

Домашнее задание

Вопросы для самопроверки

1. В каком порядке писать резюме?
2. Как описывать навыки?
3. Хорошее ли описание «базовые знания SQL»? Если нет, то как это можно исправить?
4. Как подстроить резюме под вакансию, если откликаться на 10 вакансий в день?
5. Зачем нужно сопроводительное письмо и что туда писать?
6. Как вставить портфолио в резюме?

Резюме



А теперь попробуйте составить своё резюме! Портфолио у вас уже есть, это будет дополнительной плюшкой.

Заполните резюме на HeadHunter, правильно расставив акценты на том, «что я умею». А потом поищите симпатичный шаблон и сделайте одностраничную версию для рассылки.

А потом — в бой! Выбирайте компании, которые вам понравились, пишите им сопроводительные письма, а на остальные вакансии просто откликайтесь. Мало ли, где стрельнет! И ходите, ходите по собеседованиям, это, как минимум, уберет страх перед ними.

Ответы на вопросы для самопроверки

1. В каком порядке писать резюме?

Навыки, опыт, образование — самое интересное для работодателя вверху.

2. Как описывать навыки?

Что самое сложное умеешь делать? Это и пиши вместо «базовые знания», потому что база для каждого своя.

3. Хорошее ли описание «базовые знания SQL»? Если нет, то как это можно исправить?

Плохо, потому что для кандидата базовые знания «один раз в жизни скопипастил селект», а для работодателя это простые джойны по 3 таблицам. На собеседовании будет неприятно узнать о таком противоречии. Умеете только простой селект дергать? Так и пишите «только select», если еще order, то будет «select с сортировкой по одной таблице». Если еще и join, то пишите о них. И так далее.

4. Как подстроить резюме под вакансию, если откликаться на 10 вакансий в день?

Сделать отдельную PDF-версию, ее и подстраивать! И рассылать в каждую компанию немного измененную версию.

5. Зачем нужно сопроводительное письмо и что туда писать?

Оно увеличивает шансы на собеседование, если написано грамотно — без воды и копипасты, а именно про конкетную вакансию этой компании. Зацепите коротким сопроводительным — и вас позовут, даже если опыта мало.

6. Как вставить портфолио в резюме?

Выложите на дропбокс или гуглодиск и дайте ссылку. Не надо впихивать все свое портфолио в резюме и раздувать его до невероятных размеров. Выкладывайте по паре примеров тест-кейсов и багов, короткий чек-лист, этого будет достаточно.

Глава 13
СОБЕСЕДОВАНИЕ

Резюме разослали, отклики получили, тестовые задания выполнили, получили приглашение на собеседование! Что дальше? Как себя вести, о чем говорить, о чем спрашивать?

В этой главе обсудим:

- стоит ли делать тестовое задание;
- как проходит собеседование;
- как подготовиться к нему;
- что спрашивают на собеседовании;
- что стоит спросить вам.

Стоит ли делать тестовое задание?

Когда компания ищет человека, она отсеивает кандидатов в несколько этапов:

- по резюме;
- по тестовому заданию;
- по результатам интервью (их может быть несколько).

Тестовое задание вам все равно дадут, просто некоторые компании дают его прямо на собеседовании, а некоторые — как домашнее задание на дом. Это помогает увидеть, как думает кандидат и правда ли у него есть гордо описанный навык выделения классов эквивалентности.

Делать тестовое или нет — решать вам. Некоторые гордо задирают нос: «Да я, да ни за что, вакансий и так полно». Да, возможно, вы устроитесь в другом месте. Но также возможно, что вы упускаете компанию мечты просто из-за снобизма.

Хотя я сама такая была — вообще не понимаю, как мне удалось попасть в мою компанию. Она ведь всем дает тестовое, а мне не дала. А я в те времена тоже считала, что тестовое ниже моего достоинства, что ли... В общем, не стоит тратить время! *(при этом на Word of Warcraft его тратить можно, Л — Логика!)*

Тут, правда, стоит заметить, что у меня на момент поиска работы было четыре года опыта. И я себя довольно высоко ценила, потому что на двух предыдущих работах всегда была среди лучших тестировщиков. И очень этим гордилась. Если бы я была новичком и искала первую работу, может, думала бы иначе.

Чего боятся новички, получая тестовое задание?

1. Компания эксплуатирует кандидатов, используя бесплатный труд.
2. Я потрачу время впустую.

Давайте рассмотрим оба варианта.

Компания эксплуатирует кандидатов

Компания выдает тестовое задание из серии: «Потестируйте наш сайт неделю, пришлите найденные баг-репорты». И так всем кандидатам. Потом присланные чек-листы отдаются своим тестировщикам, баги исправляются, а кандидатам отказывают. В итоге кандидаты побатрачили бесплатно.

Да, это нехороший сценарий. Не надо делать такие задания и ходить в такие компании. НО! Не надо путать бесплатный труд и тестовое, «основанное на реальных событиях».

Некоторые кандидаты настолько панически боятся того, что «компания просто бесплатно нас использует», что любое задание воспринимают как попытку эксплуатации.

Из собственного опыта...

На одном месте работы мы занимались обработкой данных: ФИО, адреса, телефоны. Тестовое давали на «своей» задаче, просили протестировать телефон. Буквально 10 проверок накидать.

Задача на 15–30 минут для опытного тестировщика, для новичка может занять час-два-до бесконечности, зависит от его уровня обучаемости.

Если кандидат плохо делает тестовое и получает отказ, то может сказать что-то в стиле:

— Ну, конечно, вы с этим каждый день работаете, а я откуда знаю вашу специфику!!!

Но, пардон, проблема не в специфике. Мы не просили проверить адрес, в котором 30+ гранулярных частей (страна, город, тип населенного пункта, населенный пункт...). Мы не просили написать чек-лист на 200 проверок, всего на 10.

Мы просили проверить телефон, а телефон в наше время у всех есть. И все знают, что это такое и зачем он нужен. Но да, для нас телефон — это именно телефон, а не просто строковое поле на страничке регистрации.

Поэтому там не прокатят проверки «Ввел число, теперь не число, проверено!». А многие только на такое разделение классов эквивалентности и способны. Увы, но это так.

Если вам предлагают несколько дней тестировать их «рабочий» софт — это выглядит подозрительно. Если предлагают потратить на это полчаса-час-два, то это нормально, и это не попытка использовать вас как бесплатную рабочую силу. Уверяю вас, у HR есть типовой ответ, с которым он будет сверять ваше решение.

Я потрачу время впустую

— А вдруг я буду делать несколько вечеров тестовое, а меня даже на собеседование не позовут???

Обидно, согласна. Собеседование-то обычно длится час, ну два. Там если откажут, вроде и немного потерял. Хотя добавьте еще дорогу. Час туда, два там (приехать пораньше + закладываем время на неспешное собеседование), час обратно... Уже четыре часа! Это если утром приезжать, на текущую работу все равно после обеда только попадете.

Так не лучше ли будет сделать тестовое задание дома? Потратить часик, сэкономить три. А то вдруг вы не подходите компании, зачем тогда тратить время и ехать на собеседование? Ну разве что скилл[170] прохождения собеседований покачать...

Весь вопрос в том, сколько времени займет тестовое и сколько вы готовы ему уделить. Мое ИМХО — делайте тестовое, которое займет не больше 2 часов.

Если вам дают задание на несколько дней — предложите работодателю его оплатить и посмотрите на реакцию :-) А если будете делать, делайте его «для себя». Ради практики. Проверить, осилите ли такую задачу.

Из собственного опыта...

Большие задания дают не только джуниорам. Я до сих пор помню, как на SQADays[171] слушала доклад «про то, как мы собеседуем». Спикер рассказывал, что они дают кандидатам задание на неделю (!) времени. И отсеивают всех, кто не тратил хотя бы по 2 часа каждый день.

Сколько человек потратил на задание, можно примерно оценить, глядя на результат. И вот спикер рассказывает о том, как они отсеивают тех, у кого не слишком красиво оформлен тест-план или чек-листы: «Значит, он не сидел над этим несколько часов. Значит, не слишком заинтересован в нашей вакансии. Нафиг!».

И так они ищут не миддла и даже не джуна, а сеньора. Человека с большим опытом и большой зарплатой. Надо ли говорить, что ищут долго? ;-) Вот где самомнение компании огромное! Название компании я не запомнила, ладно бы это был всем известный гигант или просто мегакорпорация, нет. Просто компания, которая считает, что человек должен батрачить неделю, чтобы попасть к ним на собеседование. МОЖЕТ БЫТЬ, попасть...

Лично я считаю, что идеальное тестовое задание должно быть рассчитано на 10–30 минут работы опытного тестовщика. Писать тест-планы подробные? А зачем? Тестовое нужно, чтобы увидеть, как человек думает, разбирается ли в классах эквивалентности. Это можно проверить на мелком задании.

В конце концов, если вы провалите небольшое тестовое, будет не так жалко потраченного времени! А если неделю сидеть, корпеть, а потом получить отказ, да еще без фидбека.... Нуууууу, обидненько!

Подведем итог

- Хорошее тестовое — на 10 минут опытного тестировщика.
- За бесплатный труд не беритесь, только ради опыта. Если задание не нравится или смущает — забейте.
- Но само по себе тестовое задание может быть, это распространенная практика и это — нормально. Вы можете воротить нос: «вакансий и так хватит», но джуниоров сильно больше, чем компаний, которые готовы их взять. Воротите нос и дальше — конкурентов меньше ;-)

Как делать тестовое задание?

1. ***Внимательно*** (!) прочитать условие.
2. Обдумать его.
3. Выполнять.
4. Не стесняться гуглить — если бы это было запрещено, вам бы не дали задание на дом.

Казалось бы, капитан-очевидность, зачем об этом говорить? Чтобы сделать акцент на пунктах 1 и 4.

Некоторые люди читают задание по диагонали, по крайней мере, складывается такое ощущение. В задании сказано: «Напишите конкретные тесты, чтобы взять и пройти», а тебе строчат абстракцию: «Возьмем адрес школы, стоящей на углу двух улиц....». Чтобы пройти такой тест, придется гуглить его исполняющему, какая тут конкретика?

Другой вариант: человек прочитал условие, но забил на него. Подумал, что это «текст ради текста» или что это неважно. Или что он будет выглядеть круче, если пропустит условие.

Например, в задании сказано написать 10 тестов. Человек пишет 30–50–100. И думает, что мегакрут, потому что у него идей сильно больше!! А на самом деле нет. Может, собеседующий знает, что на этот функционал тесты можно пачками писать. И специально просит именно 10, чтобы:

Невнимательно прочёл условие ДЗ?
Будешь делать лишнюю работу, за которую никто не похвалит!

- проверить, как кандидат расставляет приоритеты;
- сократить время на тестовое, чтобы оно укладывалось в разумные сроки (сделал больше — сам себе буратино).

Поэтому обратите внимание на пункт 2: обдумайте, почему именно такое условие. Если вас просят 10 проверок, ошибочно присылать больше или меньше 10 проверок:

- если меньше, это значит, что вы не умеете придумывать кейсы;
- если больше, значит, вы не умеете делать, что вас просят, и приоритизировать важные проверки от остальных.

Как попросить фидбек на тестовое?

Если вы выполнили тестовое хорошо, вас пригласят на собеседование. Там, скорее всего, вашу работу еще раз обсудят. А если нет, сами начните разговор на эту тему, услышите фидбек по работе «из первых уст».

А если вы выполнили тестовое, после чего вам пришел вежливый отказ? Попросите фидбек! За спрос не бьют ;-)

Это кажется очевидным, но не всем. Некоторые ребята в чате выпускников жалуются:

— Вот, сделал тестовое, мне прислали отказ... Даже не сказали, что не так!

— Попроси фидбек у них.

— А что, так можно?

Кто-то стесняется, кто-то просто не знает, что это нормально. Попросить один раз — это нормально. Если вам прислали стандартный отлуп «спасибо, но вы нам не подходите», попросите фидбек:

Спасибо за ответ! Я пока только учусь, мне будет полезна любая обратная связь. Подскажите, пожалуйста, какие у меня были ошибки? Куда стоит копать, что почитать, чтобы их не повторять? Буду благодарен за любую критику ;-)

Возможно, вам смогут подсказать хорошую книгу, статью, курс или что-то иное. Может быть, подскажут, чего именно не хватило в выполненном вами тестовом задании.

И тут важный момент: если вам дали фидбек, вы МОЖЕТЕ переделать тестовое и отправить его еще раз. С сопроводительным письмом о том, как вы хотите работать в этой компании. Да, где-то вас проигнорируют (ведь попытка уже была), где-то вежливо или не очень скажут о том, что шанс упущен... А где-то оценят настойчивость! И если переделанное ДЗ стало сильно лучше, могут и на собеседование позвать!

Конечно, везде нужно знать меру. Второй шанс вам дадут, а вот третий-пятый-десятый — нет. И даже фидбек второй раз могут не дать, потому что уже все сказали, что хотели. Указывать на конкретные ошибки желания нет, а что изучать — «см. предыдущее письмо».

Однако тут есть важный момент: фидбек вам никто не обещает! И если вам не дают ***подробный*** фидбек, это не значит, что компания отстой и все в ней сволочи. Это просто слишком дорого стоит, да и компания не нанималась к вам в бесплатные репетиторы, как бы ни хотелось считать по-другому.

В одной компании у нас был стандартный отлуп:

> Добрый день.
> Благодарим за уделенное время.
> В настоящий момент компания не готова сделать вам предложение.
> Удачи в поиске интересной работы!

Иногда мы сразу добавляли советы с предложением прочитать некоторые статьи Ольги Назиной, то есть меня ;-)

А вот в чатике выпускников ребята иногда жалуются:

— Воооот, я сделал тестовое, а мне фидбек не дали! Просто сказали, что не все тесты написал. Или что классов эквивалентности не учел. А каких именно? Вот редиски, сказали бы подробнее...

И у меня, как у работодателя, сразу возникает вопрос: а почему работодатель должен давать тебе подробный фидбек? Нет, это понятно с точки зрения кандидата. Классно было бы задание сделать, подробный фидбек получить. А еще возможность доработать и снова получить фидбек... И так до победного конца!

Но полезный фидбек дает не отдельно выделенный HR, который не разбирается в тестировании. Его дает специалист, который проверил тестовое. И этому специалисту надо каждому джуниору подробно расписать, что тот сделал не так.

Написать фидбек личный — это время. А на вакансию джуниоров откликаются многие. Да, не все выполняют тестовое задание, и это хороший отсев для работодателя — показывает заинтересованность именно в этой вакансии. При наличии разумного тестового (на полчаса максимум) можно сразу увидеть, заинтересован собеседник или сопроводительное письмо тоже накопипастил.

Так вот, новичков много, фидбек занимает время. Если давать его каждому — то работать когда? Далеко не всегда фирма снимает с тебя прямые обязанности, приглашая провести собеседование.

Вы это не поймете, пока сами не попадете в такую ситуацию. В чате выпускников одна выпускница так и написала:

— Я раньше не понимала. Пока мне не дали проверять тестовые и фидбек писать. Столько времени сжирается! Да я ничего не успеваю! Некогда каждому отвечать лично.

К тому же личный фидбек может оказаться неблагодарной работой...

Из собственного опыта...

HR переслала мне тестовое от кандидата. Она должна была скинуть мне само решение, но в этот раз сделала форвард. А я была в отгуле, и сработала отбивка на рабочей почте. В ней указан мой номер телефона — на случай если у заказчика будет вопрос.

И вот выхожу я на работу, проверяю это тестовое. Говорю HR: отказать. А потом мне звонят с незнакомого номера:

— Здравствуйте! Я хочу обсудить свое тестовое задание!

— Эээээ....

— (что-то продолжает радостно обсуждать, словно я в контексте).

— Какое тестовое, погодите?

— Ну, я вам присылал тестовое! Мне ответили отказом!! Я хочу поговорить с тем, кто его проверял, с тестировщиком. А то, может, это HR у вас глупый, не умеет тесты писать, вот и отказала. А я все классно сделал.

Я пока еще не понимаю, откуда у него мой номер, но отвечаю:

— Я тестировщик, и я смотрела ваше тестовое. Нет, это не ошибка, вы нам не подходите, к сожалению.

— А что не так в моем решении???

Тут я подумала, ну а почему бы и не дать фидбек:

— Ну смотрите, в условии было написано прислать список КОНКРЕТНЫХ проверок, а вы написали абстракцию.

— *(презрительный тон)* Ой, ПОДУМАЕШЬ! Ну, ЕСЛИ ВАМ ПРЯМ ТАК НАДО *(и такой язвительный тон при этом. Именно мне это и надо, да!)*, я могу переделать. *(Тонкий намек: ТАК второй шанс просить не надо).*

— Нет, спасибо. Тем более вы начали с негативных тестов, а не с позитивных. И по классам эквивалентности нет проработки. Рекомендую прочитать статьи, которые прислала вам Маша (HR).

— *(проснулась злость в человеке)* Да вы вообще даете неправильные задания!! Конечно, я не нашел нужных тестов, ведь это ВАША СПЕЦИФИКА. Бу-бу-бу...

Так что развернутый фидбек отнимает время, а иногда еще и вызывает волну негатива. Ну как так-то, не оценили твою работу. Лучшая защита — нападение. Поэтому если вам действительно дадут ***честный*** фидбек, то первая реакция будет оборонительная:

— Вы начинаете с негатива.

— Ой, ну и что, позитив же тоже есть!

— Имя тестируете как простую строку.

— Ну так это наверняка просто строка, зачем я буду проверки придумывать??? Это бесполезно, а вы тупые и классов эквивалентности не знаете, если ждете кучи тестов на разные имена!

— ...

И так далее. В итоге вы останетесь при мнении, что проверяющий был дурак, а вы «весь такой в белом», ну чуть-чуть ошиблись или недоделали. А проверяющий будет размышлять, нафига вообще решил дать вам фидбек поразвернутее.

Разумеется, когда я рассказываю эту историю о фидбеке (когда дал фидбек, еще и виноват остался), то сразу слышу:

— Нет, ну я-то хорошо воспринимаю фидбек! По нему же научиться можно! И вообще, если один раз «послали», чего теперь всем страдать?

Я уверена, что тот парень, который наехал на меня по телефону, в беседе с друзьями тоже скажет, что он умеет слушать фидбек и ему надо все говорить честно. А «та девка просто дура была, и задание у них дурацкое».

Тут можно возразить, что «это смотря как предподнести». Но, подождите, мы сейчас не говорим об откровенном хамстве и фидбеке «ты дурак». Но если просто сразу честно все сказать (раз попросили), то получится примерно так:

— Ну вот вы начали с негатива, а надо с позитива. А еще классов эквивалентности почти нет, все тесты однотипные. И тестов сказано 10 написать, а у вас 20, по 2 теста в ячейке...

Оправдать реакцию защиты можно тем, что «хоть мне и не хамили, но можно повежливее говорить». Ну, уж простите, вам реверансы отвешивать или фидбек? Шашечки или ехать? Если вы правда умеете быть благодарным за фидбек, то в любом случае скажите за него «спасибо», запишите себе, а чуть позже проанализируйте, когда первая реакция «да у меня все норм там было!» пройдет.

Подведем итоги:

1. Если вас позвали на собеседование — просите фидбек на нем, «глаза в глаза», это полезно. Проверяющий может указать вам на ваши слабые места.
2. Если вам отказали после тестового — попросите фидбек. Его могут не дать, никто не обязан работать личным репетитором всем джунов. Но вдруг?

3. Если вам дали фидбек, проанализируйте его и переделайте работу. Это мало кто делает, так что ваше повторное письмо может приятно удивить работодателя, что будет мощным плюсом. Но, опять же, второй шанс никто не обещает. Просто, мало ли, вдруг дадут?
4. Если вы провалили второй шанс, не надо с пеной у рта требовать пояснений, а также заваливать работодателя новыми вариантами: «Потыкал пальцем в небо, переписал, так верно? А так? А так? А так?».

Как подготовиться к собеседованию?

Посмотрите вакансию — какие навыки нужны работодателю? Освежите знания по ним. Если что-то не знаете вообще, погуглите, чтобы понимать, о чем речь.

Сделайте себе шпаргалку, если надо. И читайте ее в метро по пути на собеседование.

Из собственного опыта...

Когда я искала третью работу, у меня не было опыта работы с SQL. На тренажерах я не тренировалась, хотя собственную базу с помощью разработчика создала. На ней же осваивала всякие джойны. А перед собеседованием читала в метро такую шпаргалку — чтобы от волнения не забыть, какой оператор отвечает за сортировку, например:

`select` — выбрать.

`from` — таблицы, из которых идет выборка.

`orderby` — отсортировать по возрастанию.

`orderby desc` — отсортировать по убыванию.

`where` — ограничение на выборку, можно использовать >, <, «равно».

`where in (перечисление)`, где выбранное значение — одно из значений из перечисленного множества.

`where like 'часть строки%'` — где выбранное значение содержит строку.

«`%`» — любые символы, «`_`» — один любой символ.

`isnull` — проверка на пустую строку (именно `is`, «равно» работать не будет).

`from a inner join b on` — пересечение областей.

`from a left join b on` — левая область.

`from a right join b on` — правая область.

`from a cross join b on` — обе области.

```
insert into имяТаблицы(колонка1, колонка2) values
('значение1','значение2')
update имяТаблицы set параметр1='Значение1', пара-
метр2 = 'Значение2' where условие
delete from имяТаблицы where условие
delete сумма from счет1  where тыХозяинСчета
insert into счет2 values('сумма')
select p.name, g.name,sum(b.price) summ
from dbo.Genre g
inner join dbo.Book b on g.id = b.genreId
inner join dbo.Publisher p on p.id = b.publisherId
group by rollup (p.name, g.name)
group by rollup(измерение1,измерение2)
group by cube(измерение1,измерение2)
```

`rollup` — по перечисленным измерениям.

`cube` — по всем комбинациям.

`having` — после группировки (в отличие от `where`, который идет до), *исп сгруппир знач.*

`COUNT(ba.bookId)` — ненулевые записи по bookId.

`COUNT(*)` — все записи.

```
select
from
where
group by
having
order
```

Когда вам надо поменять что-то в базе данных, вы должны приконнектиться к ней и совершить некую работу. Которая называется транзакцией — упорядоченным множеством операций, переводящих базу данных из одного согласованного состояния в другое.

Одной операции всегда соответствует одна транзакция, но в одной транзакции можно совершить несколько операций.

Зачем они нужны? Транзакцию надо открыть, совершить ее операции и закрыть — или вкоммитить, или сделать откат. Допустим, вы переводите весь свой капитал с одной карточки на другую. Выглядит это «внутри» как несколько операций:

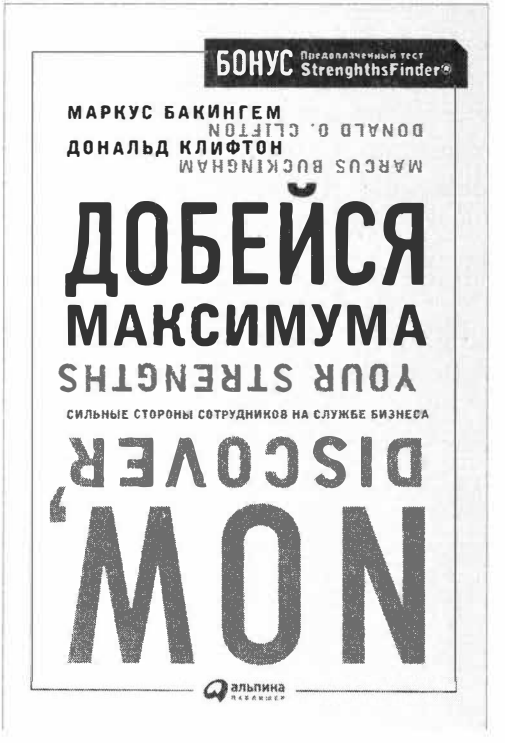
```
delete сумма from счет1  where тыХозяинСчета
insert into счет2  values ('сумма')
```

Так вот. После удаления денег с одного счета и до того как они поступят на второй, транзакция закрываться НЕ должна. Иначе, если что-то пойдет не так, она откатится не полностью, и деньги будут утеряны.

Аналогично можно записать себе что-то по языку программирования или теории тестирования. Хотя я противник зазубривания теории, но мало ли...

В общем, смотрите, что нужно работодателю, и вспоминайте нужные навыки!

А еще полезно подумать о своих сильных сторонах. Это придаст вам уверенности в своих силах. Для этого рекомендую книгу «Добейся максимума» Маркуса Бакингема и Дональда Клифтона.



Авторы составили некий тест на определение талантов. Он платный, но можно просто прочитать книгу и самому прикинуть, что вы умеете лучше всего.

Я вот выделила свои сильные стороны:

- писательство — иначе бы я не написала эту книгу и не вела столько лет блог;
- рассказывание сложного через простое — иначе бы не смогла делать курсы;
- организация людей — если хочешь в настолки поиграть, на спортивный квест сходить или на гонку героев, надо организовать себе команду! А кто, если не ты?
- ответственность — иначе бы я не дописала эту книгу ;-) ну и вообще, люблю доводить начатое до конца.

HR вполне может расспрашивать вас о себе, просить описать сильные и слабые стороны. Чтобы не тупить на собеседовании, отчаянно вспоминая «в чем же я силен ?!», подумайте об этом заранее.

И список слабых сторон можно обдумать. О чем можно рассказать HR`у? Чтобы от вас не было завышенных ожиданий, но вы и не казались при этом последним неудачником? Я, кстати, к такому не готовилась ;-) Сейчас написала этот абзац и стала как раз панически думать: «Аааааа, а что же ответила бы я???». Чтобы избежать паники на собеседовании, подумайте заранее.

Делать ли шпаргалки по теории тестирования? Я рекомендую не заучивать теорию, а практиковаться. Это можно сделать на курсах[172], можно по этой книге честно выполнять все ДЗ. Желательно еще найти друга-тестировщика, который вам ваше портфолио прокомментирует при этом. Ведь именно исправления по фидбеку принесут максимум пользы и понимания!

Если будете волноваться, заученное забудется. А если щупали руками, особенно много раз (несколько попыток сделать хорошо), это запомнится и о нем легко будет рассказать. Ну а уж если компании нужны только четкие определения, подумайте, а нужна ли такая компания вам?

Из собственного опыта...

Больше всего меня дрючили по теории тестирования на собеседовании в EPAM. Там 1,5 часа мне молодой человек задавал всякие вопросы. Если я не понимала термин, просто просила пояснить. После пояснения уже могла сказать:

— О да, я это знаю! Мы делали так и так, просто называлось это по-другому.

Собеседование я, правда, не прошла... И хорошо, я нашла себе отличное место. Где ценят не заученное, а опыт и светлую голову ;-)

* * *

В другой компании мне дали листы А4 с вопросами из ISTQB. А я тест не сдавала, так что просто по своему опыту опять же отмечала ответы. Но местами прям тяжело было, с учетом опыта в 4 года!! И думалось: «Ну вот нафига это все?».

То была аутсорс-компания, она продавала своих тестировщиков на проекты в банки. Им были важны сертификаты и теория. Типа: знаешь теорию (получил сертификат) — ты клевый, тебя можно продать подороже.

Увы, в аутсорсе теория тестирования важна. Придется учить!

Я рекомендую делать шпаргалки по SQL. Или регэкспам, или языку программирования. И смело в бой!

Как проходит собеседование?

Вас встречают, предлагают чай, кофе, плюшки. Потом проводят в переговорную / пыточную, где и будет основное действие. Стандартная практика такая.

Собеседующий:

1. Рассказывает о своей компании и своем проекте.
2. Просит рассказать немного о себе.
3. Задает уточняющие вопросы по вашему рассказу / резюме.
4. Обсуждает сделанное дома тестовое задание.
5. Дает тестовое задание и либо сидит рядом, либо выходит минут на 10–15 (чтобы вы не психовали лишний раз от того, что он висит над душой).
6. Обсуждает ваше решение тестового.
7. Спрашивает, остались ли у вас еще вопросы, отвечает на них.

Конечно, это не прям панацея, пункты могут меняться местами, чего-то может вовсе не быть, что-то может добавиться. Но примерно план такой.

Рассказ о проекте и компании

Это очень важная часть собеседования. Если вам не рассказывают о проекте, куда хотят поставить, спрашивайте сами! Возможно, вам не подходит стиль работы в компании. Или не нравятся задачи, которые предстоит выполнять. Лучше узнать это в начале собеседования и сразу разбежаться, не тратя время.

Для начала уточните, на каком проекте вы будете работать. Вдруг с вами разговаривает человек не из будущей команды? А вам надо понять, как выполняется работа на «вашем» проекте. Попросите рассказать, как проходит релиз:

1. Сколько митингов?
2. На сколько они человек?
3. Сколько времени длятся?
4. Как часто проходят?
5. Как проходит планирование?
6. Есть ли ретроспектива?
7. Как она проходит?
8. Кто общается с пользователями?
9. Кто пишет требования?
10. Есть ли автоматизация? Как она выглядит?
11. Кто занимается автоматизацией?
12. Будете ли вы заниматься ручным тестированием? Автоматизацией? Нагрузкой?
13. Есть ли переработки и насколько часто возникают?
14. Какая иерархия в компании, какой вас ждет рост: горизонтальный или вертикальный?

Проводя собеседования, я сразу рисовала перед собеседником цикл продукта: от внедрения до поддержки (коробочный продукт). Потом рассказывала, что делает тестировщик на каждом этапе. В одной компании

тестировщик был и швец, и жнец, и на дуде игрец — он и анализирует данные на внедрении, и требования сам пишет простые, и тестирует вручную, и автотесты фигачит, и с заказчиками общается.

Не всем это подходит. Кто-то не любит писать документацию, и дальше мы думаем, насколько важен этот навык. Или человек не хочет заниматься поддержкой вообще, а для нас это важно. В таком случае можно сразу и распрощаться, не тратить время друг друга зря.

Очень полезно сразу проговорить все детали и ответить на уточняющие вопросы. Может быть, человек пришел, думая, что будет фигачить код на java целыми днями, а у вас тестировщики в код почти не лезут... Или он хочет расти в менеджеры, а у вас просто нет такой позиции.

Когда меня расспрашивают про автоматизацию и нагрузку на проекте, я сразу рассказываю, как это есть у нас. Потому что автоматизация у нас — это не просто «наколбасил тесты на селениуме», такого почти нет. А есть уже готовый написанный фреймворк. В код тестировщик лезет редко, нужен именно тест-дизайн.

Были кандидаты, которых явно расстраивал мой рассказ — они-то думали, что придут и будут именно автоматизаторами. Такими, которые пишут код и потом добавляют гордую фразу в резюме. Если бы я не рассказала «как реально есть», было бы большое разачарование после прихода на работу.

Если кандидату нужен хардкор, нагрузка, автоматизация — то пусть лучше сразу знает, что у нас по-другому. Если его устраивает мой ответ, мы продолжаем диалог.

Из собственного опыта...

Другой пример из жизни. Рассказываю я о том, как у нас все устроено. Смотрю, лицо у кандидата вытянулось:

— Подождите, подождите... Так у вас нет начальников?

— Нет, каждый сам себе босс и руководитель проекта.

— А кем тогда я буду руководить??

Тут уже лицо вытянулось у меня, мы не искали тест-менеджера:

— Эээээ... Никем...

Кандидат сразу потупил глаза, и мы вскоре свернули собеседование, потому что к нам он явно передумал идти ;-)

Так что если что-то важно для вас — обязательно спрашивайте! Конечно, скорее всего, для первой работы почти ничего не важно: «Готов работать за еду». Но на будущее знайте, что рассказ о компании — важная часть собеседования. И если вас что-то напрягает в текущей работе, обязательно уточняйте, как этот процесс происходит «у них».

Рассказ о себе

Потом собеседующий просит вас рассказать о себе. Или начинает с этого. И он не будет рад ответу «всё есть в резюме», потому что хочет послушать именно ваш рассказ.

Я обычно читаю резюме, но по диагонали. Фотографической памятью не обладаю, а заучивать все места работы смысла не вижу. Поэтому прошу рассказать о себе и в процессе рассказа читаю резюме кандидадата. Если меня что-то заинтересует, уточню:

— Я вот работал в компании А, потом в компании Б...

— О, у вас тут написано, что автоматизировали процесс! Расскажите подробнее, пожалуйста.

Или:

— А почему ушли?

Почему ушли с предыдущего места?

Вопрос про «почему ушли» — стандартный. Отвечать на него лучше честно, не надо пытаться придумать социально положительный ответ. По крайней мере я спокойно отношусь к комментариям вроде «там мало платили, тут предложили больше», особенно когда речь о первых местах работы.

Да, всегда есть шанс, что человек и от вас сбежит за большей зарплатой. Но можно предложить хорошую зарплату хорошему специалисту. Ведь есть какой-то порог, после которого не только сумма влияет на «хочу тут работать»... Хотя бывает и такое, что это единственный фактор, ну что ж. Бывает, и не за зарплатой убегают, от этого никто не застрахован.

Лично меня смущает, когда человек каждые 2–3 месяца меняет работу. Ну или каждые полгода. Вот тут уже задумываешься: это правда на его работе

все плохо, или просто ему не угодишь? А некоторые считают, что «торчать в одной компании больше двух лет — это значит перестать развиваться!». Ха-ха-ха!

Я до декрета отработала 6 лет, а развиваться мне еще ого-го куда было! Дело же не в компании, а в вас. Хотите развиваться — будете это делать. Да, бывают компании, где действительно нет роста и вообще скукота. Но если вы уже опытный тестировщик, то легко найдете себе более интересное место.

А вот мне как работодателю не хочется брать человека, который через полгода ускачет «пробовать новое». Некоторые именно так и отвечают на вопрос:

— Почему так часто меняете место работы?

— А что делать после целого года? Я уже всему научился! *(ну да, ну да...)*

Первые пару месяцев вы даже особой пользы приносить не будете. Ну разве что вас берут на прохождение по чужим тест-кейсам или «легкую» систему. В целом, простых систем сейчас достаточно, и есть большой шанс трудоустроится в такую контору.

Протестировать интернет-магазин вы будете в состоянии сразу же. Ведь у вас есть опыт пользователя, вы понимаете, что-как-почему. А вот если приложение узкоспециализированное, сначала вы будете долго вникать...

Расскажите подробнее, чем вы занимались

В резюме все не напишешь. Когда вы рассказываете о своей работе, собеседущий может сделать выводы о том, что:

- вам важно;
- вас бесит;
- вас вдохновляет;
- ...

Ведь об одной и той же работе разные люди будут говорить по-разному. Кто-то возведет глазки к небу:

— А еще тестировщики документацию писали! *(ну вы прикиньте, а?!!)*

А другой ту же фразу расскажет с контекстом «вау, как классно, не соскучишься!». И если в нашей команде написание документации — часть задачи тестировщика, то я сразу скажу об этом обоим кандидатам. И с первым мы будем заканчивать, а с другим продолжим диалог.

Также рассказ дает представление о том, какой у вас реальной опыт. До сих пор помню человека, у которого в резюме было 5+ лет нагрузочного тестирования. Уточняю:

— Расскажите подробнее, как проводили нагрузку?

— Ну... *(замялся)* Вообще разработчики сделали все скрипты, а я просто на кнопку нажимал и потом на графики смотрел.

Такой кандидат скрипты нам не напишет. Потому что если бы было желание, то можно было бы хотя бы разобраться в том, что именно сделали программисты. Впрочем, мы на тот момент и не искали человека, умеющего проводить нагрузку. Просто впечатлилась опытом, переспросила, а оказалось пшик...

Поэтому не нужно на просьбу рассказать о себе или об опыте работы надменно отвечать:

— Это есть в резюме.

Я в курсе, спасибо. Но я хочу услышать вашу версию. Плюс не учу резюме наизусть, поэтому буду туда подсматривать, слушая ваш ответ. И если увижу там что-то интересное, обязательно уточню.

Расскажите про самый интересный баг

Или про самый главный фейл в работе. Или про самый интересный день. Или что-то еще в таком роде.

У новичка без опыта это спросить нельзя, разумеется. А если есть хотя бы полгода опыта — можно! Такие вопросы помогают понять, насколько вам интересна ваша работа. Потому что если ничего не припоминается, то ничего интересного и не было. Или вы внимания не обращали.

Есть люди, которые могут только «работать от рассвета и до заката». Они ничего не запомнят из своего дня. «Работу работал». Иногда нужен именно такой человек — на рутинную работу типа прогона тест-кейсов. Один на ней взвоет, а другой будет выполнять каждый день, и ему хоть бы хны.

Лично мне нравится моя работа. И меня удивляет, когда люди с опытом в 5+ лет не могут ничего запоминающегося рассказать. За пять лет вот вообще ничего не запомнилось? Конечно, может сказываться волнение от собеседования. Но иногда люди даже не пытаются вспомнить или подумать, лишь меланхолично пожимают плечами.

Зато бывает очень интересно поболтать с кандидатами, которым есть что рассказать! Иногда что-то новое узнаешь, иногда просто разговор завязывается, ты можешь человеку что-то из своей практики рассказать. Выгода всем!

Тестовое «на дому»

Вы сделали тестовое хорошо, ведь вас уже пригласили на собеседование! Но «хорошо» — не значит «идеально». Поэтому стоит обсудить ваш результат.

Скорее всего, собеседующий сам поднимет эту тему. Я иду на собеседование, держа в руках резюме кандидата и его решение тестового задания. Обсуждение полезно:

- это отличная тема для разговора;
- вам легко говорить на эту тему, если вы делали задание сами (снимаем стресс у кандидата);
- собеседующий вживую видит, как человек мыслит (и если тестовое вам сделать помогли, будет большой провал);
- можно обсудить плюсы и минусы сделанной работы, узнать куда двигаться дальше.

Если собеседующий не завел разговор на эту тему, сделайте это сами. Получить фидбек — бесценно. Даже если провалите собеседование, будет что дорабатывать дома.

Совет от капитана-очевидности — не пытайтесь отправить чужое решение тестового. Попросили коллегу, мужа или друга помочь? Попросите объяснить на пальцах, почему именно так. Иначе за одну минуту рассказа «как я это делал» собеседующий вас расколет.

Также не рекомендую делать тестовое за кого-то. Вам прикольно ради опыта? Сделайте для себя. Отдавать решение не надо. А то на форуме

Обсуждение тестового поможет кандидату расслабиться

тестировщиков раз в неделю появляются темы «Вот мне дали задание протестировать то и се. Подскажите, как это делать!». И ведь некоторые подсказывают!

Ну камон, человек поленился сделать ХОТЬ ЧТО-НИБУДЬ. Ваш готовый ответ — медвежья услуга. Да, возможно, такого кандидата позовут на собеседование. И это будет потерянное время технического специалиста компании. Он ведь легко поймет, что кандидат делал задание не сам. Отказ и время на дорогу, кому от этого стало лучше?

Делайте задание сами. И если просите помощи, просите именно помочь, а не сделать за вас. Иначе это все впустую.

А на собеседовании просите прокомментировать. Расскажите, как делали задание. Вам могут подсказать пропущенные кейсы, а это уже опыт. Будете знать на будущее!

Тестовое «в офисе»

Было тестовое «на дому» или нет, в офисе обычно дают задачку. Она короткая, ведь надо решить «здесь и сейчас», а уровень сложности может разниться.

Например, если я сомневаюсь в тест-дизайнерских качествах собеседника, то прошу решить простенькую задачку типа треугольника[173]. Если мы обсудили его решение задачи на дому и оно мне понравилось, я даю более сложную задачу.

Дав задачу, собеседующий или сидит рядом, открытый для вопросов, или куда-то уходит. Уходит обычно для того, чтобы не нервировать человека. Когда кто-то стоит над душой, думается хуже...

Задачка дается минут на 10–15–20, но это в среднем. Зависит от того, какие навыки нужны от кандидата. Иногда дают целые тесты на час!

Я так пару раз собеседовалась. Мне давали несколько листов A4 с кучей вопросов: и по теории тестирования, и по программированию, и по SQL, и по английскому... Потом собеседующий уходил на час. И ведь этого времени едва-едва хватало!

В любом случае, в итоге собеседующий вернется, и вы обсудите задание. Что сделали хорошо, чего не хватило? Цените этот фидбек! Это ведь в любом случае опыт.

В чем прийти?

В ИТ нет дресс-кода, и это большой плюс! Поэтому, если вы идете не в банк, то оденьтесь опрятно, но обычно. Не вижу никакой проблемы в джинсах и кофте или толстовке. Наоборот, если для вас отсутствие дресс-кода важно, то надо прийти именно так, как планируете ходить на работу в дальнейшем.

А то представьте себе. Пришли в костюме, белая рубашечка, черный пиджачок, брючки... Произвели впечатление, вас приняли, а потом... На работу является чудо в перьях ;-)

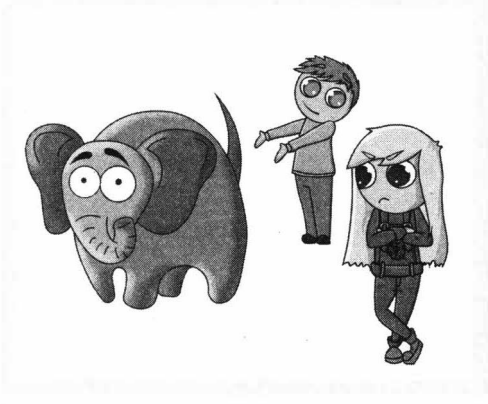
Для меня важно не ходить каждый день в костюме, поэтому я обязательно сама спрашивала на собеседовании, есть ли в компании дресс-код. И вы спросите.

Если вас собеседует человек в костюме — это еще ни о чем не говорит. Может, надо иногда в костюме приходить, а может, у него сегодня переговоры с клиентом были, а так-то он в джинсах ходит. В общем, не парьтесь! И уточните этот момент.

Как себя вести?

1. Расскажите о себе.
2. Узнайте о компании.
3. Делайте выбор!

И, главное, ничего не бойтесь. Помните — не только компания выбирает вас, но и вы выбираете компанию!



Мы выбираем, нас выбирают...

Что спросят вас?

Вопросы на собеседовании всегда разнятся. Что компании важно, о том и спросят. Смотрите на навыки, которые запрашивают у кандидатов. Спросят:

- теорию тестирования;
- вопросы по этим навыкам (SQL — задача по SQL, java — на знание языка и т. д.).

Но из раза в раз в чатике новичков возникает вопрос: «А кто ходил уже? Что там спрашивают?». Как будто можно подготовиться ко всем вопросам ;-) Мое ИМХО — лучше практику нарабатывайте. Тогда, даже если будете плавать по теории, сможете объяснить своими словами.

И все же для успокоения нервов приведу вопросы, которые задавала сама.

Общие вопросы

- Какие проекты у вас оставили самые интересные воспоминания? Почему?
- В каких случаях применяется автоматизированное тестирование?
- Какие книги по тестированию читали?
- Чем оправдано выделение роли QA в компании?
- Как попали в тестирование?
- Почему ищете новую работу?
- Какой продукт сейчас тестируете?
- Про один из скиллов в резюме.

Технические вопросы

Windows

- Где посмотреть в Windows (2007/10) переменные среды? Чем отличаются переменные системы от переменных среды пользователя?
- Как посмотреть IP-адрес компьютера?
- Чем отличается имя компьютера от IP-адреса?

Linux

- Как создать каталог, файл?
- Как понять, где находишься?
- Как посмотреть, какие процессы запущены на компьютере?
- Как понять, где установлено приложение?

SQL

- Написать SQL-запрос из двух таблиц
- Что выдает запрос: `select * from A,B`?
- Что такое первичный ключ? Внешний ключ?
- Что такое транзакция?

На «поболтать»

Приведенные далее вопросы — это примерные топики для общения (а именно во время общения вы понимаете, подходит ли вам собеседуемый).

- Что вы сделали (оценивают опыт)?
  - Самый запоминающийся проект.
  - Три самых любимых проектных активности.
  - Самая большая неудача.
- Что вы хотите делать (оценивают требования к работе)?

Я часто рисую фазы проекта:

- Анализ.
- Настройка.
- Тестирование.
- Обучение.
- Приемочное тестирование.
- Поддержка.

И прошу отрейтить с 1 по 6.

- Что вы точно не хотите?
- Если бы вы могли выбирать абсолютно любую работу, что бы это было?
- Что вам нравится (оценивают личность)?
- Какие у вас сильные и слабые стороны?
- Что вы любите читать?

Что спрашивать вам

1. Расскажите о ваших процессах.
2. Как проходит релиз?
3. Как выглядит ваш рабочий день?
4. Зарплата?
5. Парковка?
6. График гибкий?
7. Больничные оплачиваете?
8. За сколько надо предупреждать об отпуске?
9. Есть ли дресс-код?

Спрашивайте все, что вам интересно и важно. Ваша задача — понять, будет ли вам комфортно тут работать.

Уточните:

- какие задачи вам придется выполнять — вдруг вы бежите от написания документации, а тут вам тоже ее придется писать?
- как построены процессы в компании?
- какие есть плюшки, льготы, прочее?

Домашнее задание

Ходите по собеседованиям! Приходите туда в одежде, в которой вам будет комфортно. Уточняйте условия труда. И не волнуйтесь — все когда-то были новичками.

Перед собеседованием сделайте себе шпаргалку для повторения в метро. Не зубрите теорию — это вылетит из головы! Но напомнить себе, какие бывают join-ы, вполне можно.

Если прям очень переживаете — то сначала идите в компании, в которые НЕ хотите попасть. Или не особо хотите. Так не страшно будет провалиться, но уже появится опыт собеседования. Чем больше собеседований пройдете, тем меньше будете переживать. Меньше стресса — больше шансов вспомнить теорию или просто произвести хорошее впечатление. Так что, вперед!

Глава 14
КУДА РАЗВИВАТЬСЯ?

Книга подходит к концу. Вы уже знаете основы, которые нужны начинающему тестировщику. А теперь поговорим о том, куда будем развиваться дальше.

В этой главе мы рассмотрим направления развития:

- тест-дизайнер;
- автоматизатор;
- тест-менеджер;
- аналитик;
- швец, жнец, на дуде игрец.

Несколько вступительных слов

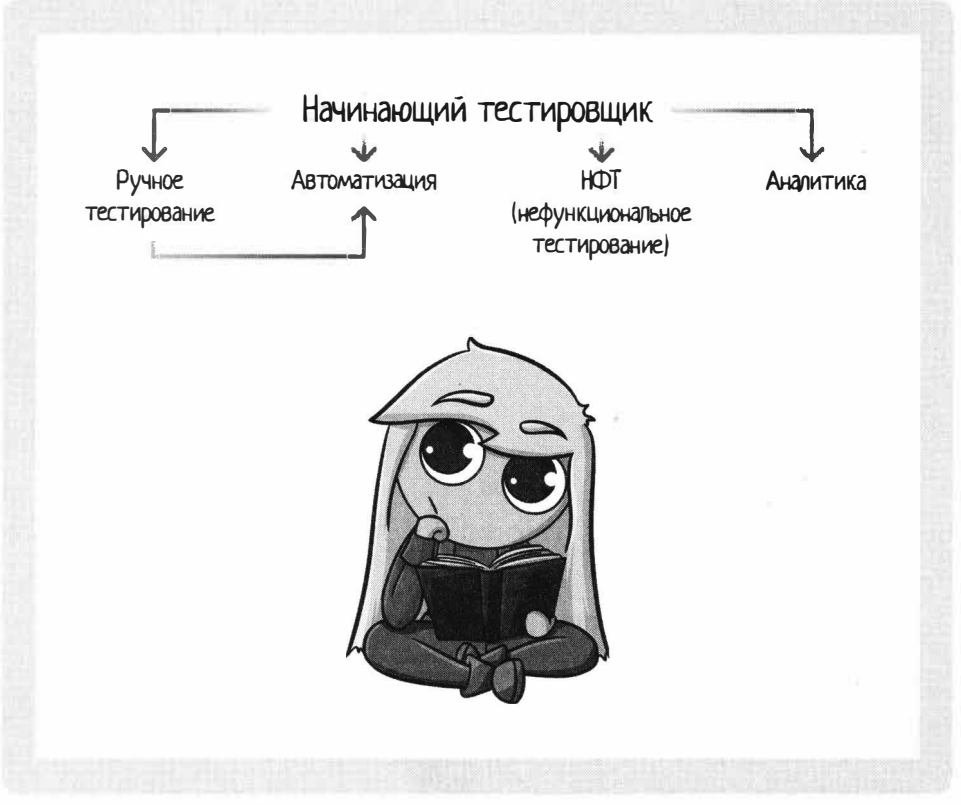
В этой главе я расскажу про основные направления развития в тестировании. По каждому я порекомендую какие-либо книги. Ссылки на них вы можете найти в моем блоге в общем списке книг[174]!

Направления развития

Для начала обсудим, какие в принципе есть направления для развития у тестировщика, а потом пройдемся по каждому.

Вот четыре основные области развития тестировщика:

- ручное тестирование;
- автоматизация;
- НФТ (нефункциональное тестирование);
- аналитика.



Плюс вам в любом случае будет нужна какая-то общая грамотность. Что в нее входит?

Понимание цикла разработки

Чтобы получить готовое приложение, нужно:

1. Написать код на языке программирования — код при этом хранится в системе контроля версий (Version Control System, VCS).
2. Собрать код в приложение — этим занимается сборщик продукта.
3. Запустить это приложение — этим занимается сервер.
4. Автоматизировать пункты 2 и 3 — этим занимается CI (Continuous Integration), иногда используя Docker.

Что там внутри?

Приложение, скорее всего, будет клиент-серверное. Поэтому надо понимать, что это такое.

И архитектура наверняка будет трехзвенная, с учетом базы данных. Поэтому нужно:

- понимать, что вообще такое БД;
- знать простейшие SQL-запросы — это сейчас требуют даже от джуниоров.

А еще нужно знать, что такое API, и уметь его тестировать. Потому что, даже если вы тыкаете на кнопочки в графическом интерфейсе, вы всё равно тестируете API, просто вызывает его система.

А еще тестировщик может тестировать «чистое» API — обычно это запросы SOAP или REST. Запросы в API обычно составляются в форматах XML и JSON. Так что с форматами тоже познакомьтесь.

Общая компьютерная грамотность

По факту это умение работать с командной строкой. Что вы должны уметь тут делать? План-минимум:

- перемещаться по папкам;
- копировать файлики;
- запускать приложение;
- архивировать файлы.

Так что гуглим, как это делается. Плюс почитайте, что вообще такое bash / shell.

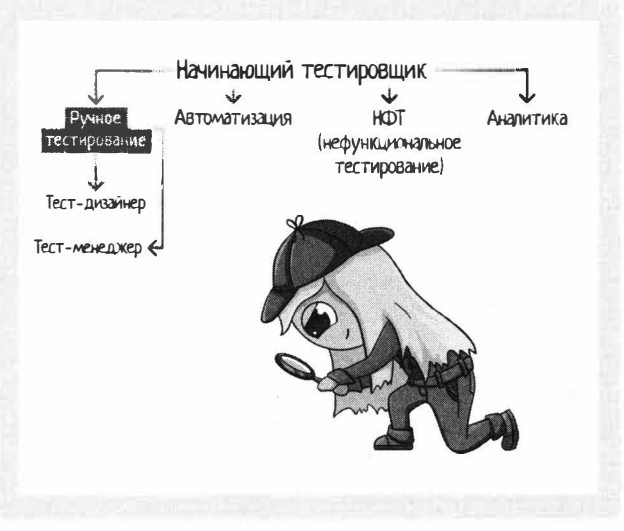
Это то что стоит знать в любом случае, какое бы дальнейшее направление вы ни выбрали. А мы переходим к областям развития.

Ручное тестирование

Как ни крути, а без ручного тестирования никуда. По крайней мере, если вы планируете работать тестировщиком. Даже если вы хотите автоматизировать или заниматься usability-тестированием.

Если не знать основ тестирования, то что вы сможете делать? Разве что перекладывать на код чужие сценарии. К тестированию такая работа имеет очень отдаленное отношение. А если вы хотите писать обдуманно, то нужно сначала научиться тестировать.

Некоторые считают, что ручное тестирование — путь в никуда. Так и останешься мартышкой, которая только ручками что-то делает и при этом получает на порядок ниже тех, кто освоил автоматизацию.



Нет, это не так. Знаю людей, которые не занимаются автоматизацией в принципе и при этом неплохо получают. Если ты специалист в своей области, ты будешь цениться. Не стоит циклиться только на автоматизации, даже если она вам поперек горла стоит.

Куда расти из ручного тестирования:

- тест-дизайнер;
- тест-менеджер;

Классный тест-дизайнер будет цениться всегда. Потому что именно он придумывает тесты, в том числе для автоматизации.

А если вы будете работать в большой компании, то в отделе тестирования должен быть главный. Тот, кто станет координировать работу, обучать новичков и прочее. Тест-менеджер — это плюс одна ступенька в типичной иерархии компаний.

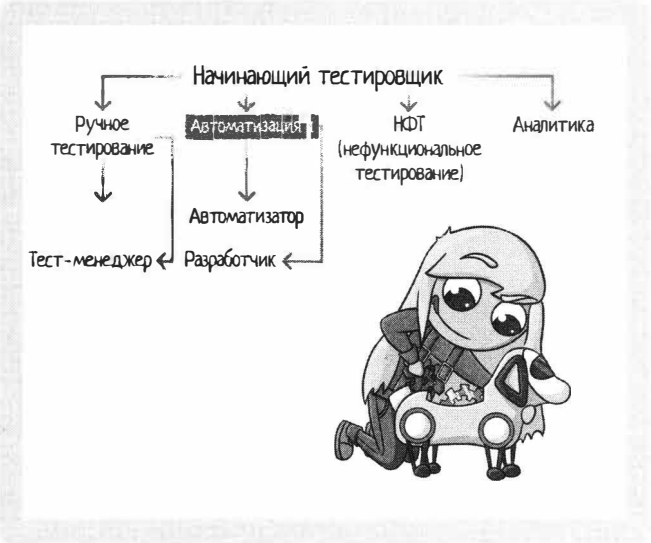
Автоматизация

Самое очевидное направление развития — автоматизация. Если посмотреть вакансии, то в половине надо знать язык программирования, а то и иметь опыт автоматизации.

Типичный рост люди видят так:

- автоматизатор;
- разработчик.

Но это неверно! Если вы хотите быть разработчиком — идите сразу в разработку. Тестирование будет лишь потерей времени.



Я читала на форумах что-то типа: «Хочу быть разработчиком, но там столько всего надо знать. А в тестировании ничего знать не надо, поэтому пойду пока туда». Ну-ну... И что вам это даст? Тестировщик ≠ разработчик. С точки зрения разработки время в тестировании будет проведено даром. Потому что навыков разработки вы получите 0.

Бывает, конечно, когда компания дает вырасти тестировщику в автоматизатора, а потом и в разработку. Но это ооооочень долгий путь. Зачем это надо, если намного быстрее вы станете джуном в разработке?

Подкину лайфхак. Научились что-то программировать — идите на фриланс-биржу. Тестировщики там не сильно в почете, просто потому что многие не понимают, зачем они нужны.

А вот разработчики нужны! Ведь заказчик — это человек, далекий от мира ИТ, он просто хочет свой интернет-магазинчик или что-то такое. Готовое решение. И если вы сможете его обеспечить, то честь вам и хвала, а также плюсики в карму.

Так начинающий разработчик может сразу начать получать деньги. Буквально за пару недель. А тестировщик на бирже может ооооочень долго искать заказ.

Поэтому если хочется в разработку, идите сразу в разработку. Да, в тестирование проще впрыгнуть, но толку для будущей карьеры будет ноль.

А вот в автоматизацию развиваться можно и нужно (если желание есть). Даже если вы не будете сами писать автотесты, то нужно знать хоть один язык программирования хотя бы на уровне «понял, что прочитал». Понял простейший класс, а не замудреную бизнес-логику.

У меня есть бесплатное приложение с открытым исходным кодом — Folks. Оно основано на реальном проекте, поэтому вы можете потренироваться «читать код» на нем. Вот кусок модели человека:

```
// ФИО
@Searchable(search = true, filter = true, sort = true)
privateStringsurname;
@Searchable(search = true, filter = true, sort = true)
privateStringname;
@Searchable(search = true, filter = true, sort = true)
privateStringpatronymic;

// Докладчик
@Searchable(filter = true)
@Type(type = "org.hibernate.type.NumericBooleanType")
privatebooleanspeaker;
```

Зная английский и минимально понимая java, можно легко прочитать, что тут написано. И иметь представление о том, какие у нас есть поля в модели и каких они типов.

В реальном приложении есть специальные правила на обновление карточки. Они записываются в XML примерно в таком виде:

```
<updateName>
<ifEqualThenOldWin/>
<MoreActualityWin/>
</updateName>
```

Тоже несложно прочитать, если XML-разметка не пугает. Конечно, тут и разработчики должны стараться делать код понятным и читабельным. Но уж если они постарались и у вас есть доступ к коду, грех не воспользоваться!

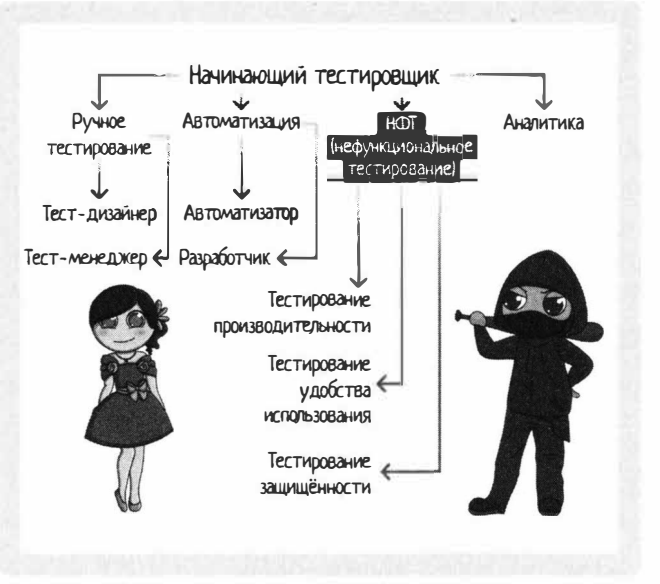
Автоматизаторы очень неплохо получают, что привлекает будущих тестировщиков. И если есть желание расти в эту область — растите! Остальным достаточно базовых навыков, чтобы не писать код, но хотя бы понимать красиво написанный.

НФТ

НФТ — нефункциональное тестирование. Подробно его мы обсуждали в *главе 9*. Куда расти в этом плане?

- Тестирование производительности.
- Тестирование удобства использования.
- Тестирование защищенности.

Выбираете направление и изучаете его.



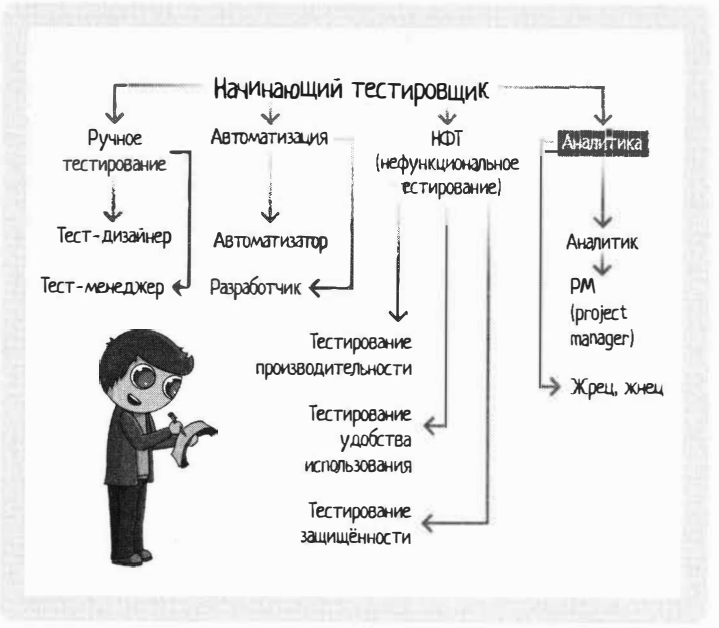
Наименее перспективное — это тестирование удобства использования. Просто потому, что его редко заказывают. Все считают, что его отлично проведет обычная команда и отдельный специалист тут вообще не нужен. Поэтому прокачаться здесь в крутого спеца, у которого отбоя от заказов не будет, проблематично.

А вот нагрузочник или безопасник получают неплохо. Просто потому, что тестировать нагрузку и защищенность нужно, а делать это мало кто умеет. И тут готовы платить приглашенному специалисту.

Аналитика

Можно перейти в соседнюю область аналитики. Что тогда ждет по карьере?

- Аналитик.
- PM (project manager).
- Швец, жнец, на дуде игрец — когда и тестируешь, и аналитику проводишь. В некоторых компаниях именно такие и нужны.



В ряде компаний аналитиков просто нет. Поэтому задача «описать ТЗ» ложится на разработчиков или тестировщиков.

Поэтому изучить эту область хотя бы на верхнем уровне я рекомендую. Пригодится в тестировании.

Тест-дизайнер

О профессии

Основные задачи тест-дизайнера:

1. Придумывать тесты.
2. Выкидывать лишнее.
3. Анализировать документацию.

Только кажется, что это очень легко. Ведь вы новичок, а уже можете придумывать тесты! Все верно. Только опытный человек напишет тестов меньше, а багов найдет больше. Потому что он знает, где баги водятся, и бьет точечно, а не ковровой бомбардировкой «проверю-ка я всё».

Какие техники надо уметь применять?

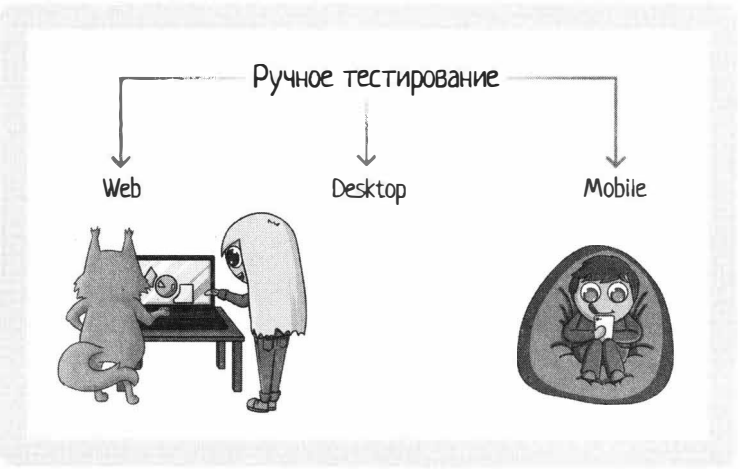
1. Классы эквивалентности.
2. Decision Table (таблица решений).
3. State-Transition (схема состояний).
4. Исследовательское тестирование.

А еще может пригодиться умение:

1. Работать с базой данных — знать SQL.
2. Отправлять API-запросы.
3. Работать в Linux-системе.
4. Читать код.

И это всё для ручного тестирования! По крайней мере, сейчас эти навыки активно требуют. Не все и не везде, но бывает. И в базу пускают, и API дают пощупать. Причем вне зависимости от направления:

- Web;
- Desktop;
- Mobile.



Так что даже если вы не занимаетесь автоматизацией, все равно нужно кучу всего знать. А ведь SQL и API — это «побочные» навыки, без непосредственно тест-дизайна они бесполезны!

SQL нужен, чтобы достать значение из базы данных. Но ведь нам надо понимать, ЧТО мы хотим оттуда достать и ЗАЧЕМ. Иначе какой смысл в том, что у вас есть доступ к БД? Сначала продумайте свой тест, а потом уже пишите SQL-запросы.

Или API. Мы всегда тестируем API, даже работая с графическим интерфейсом. Просто в этом случае оно скрыто под капотом. Но иногда API выставляют наружу, для интеграций. Методы, которые доступны по этому интерфейсу, нельзя найти в графическом интерфейсе, чтобы проверить там. Надо смотреть «напрямую».

При этом, когда люди приходят ко мне на курс по тестированию REST API, они удивляются, что «тут ведь тоже можно и нужно применять тест-дизайн!». Да, есть проверки, специфичные именно для API. Но в остальном

это такое же ТЗ, которое надо проверить. Параметры на входе, данные на выходе и разные их комбинации. Без тест-дизайна никуда!

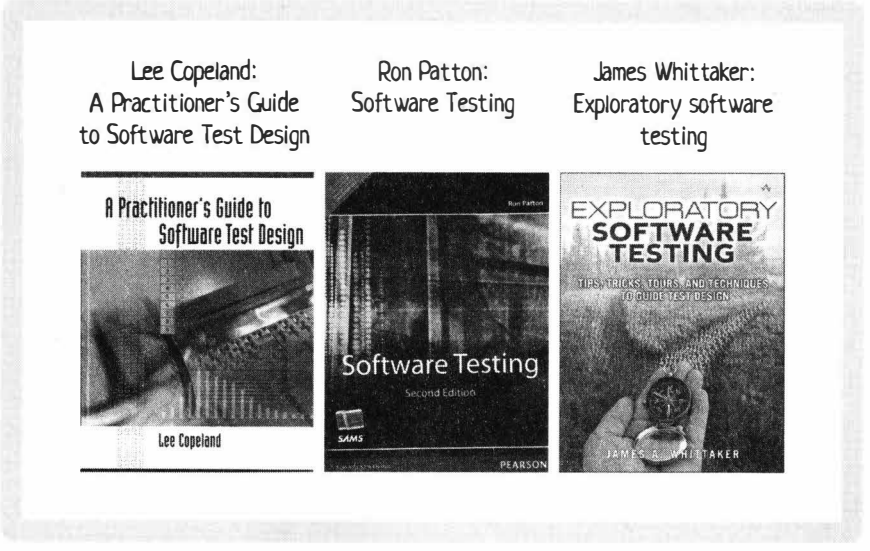
Поэтому старайтесь прокачиваться в тест-дизайне, когда есть возможность. Он пригодится вам всегда и везде. Даже если вы уйдете в автоматизаторы! Но об этом чуть позже.

Книги и ресурсы

Тест-дизайн

Топ-3 получается англоязычным:

- Lee Copeland: «A Practitioner's Guide to Software Test Design».
- Ron Patton: «Software Testing».
- James Whittaker: «Exploratory software testing».



Рекомендую также книгу «Мастерство. Путешествие длиною в жизнь» Джорджа Леонарда — поможет понять, что вы только в начале своего пути.

SQL

- Линн Бейли: «Изучаем SQL» — моя любимая серия книг, с кучей картиночек и подробным разжевыванием материала!
- Ресурсы:
  - **http://www.sql-ex.ru/** — теория + практика, там есть задачки;
  - **https://www.w3schools.com/sql/** — очень хороший учебник как по языкам программирования, так и по SQL.

API

Эрик Рэй: «Изучаем XML» — в XML возвращаются все SOAP-запросы и некоторые REST.

Других книг я пока, к сожалению, не знаю. Чтобы именно по тестированию или близко к этому. Но планирую исправить этот недостаток и написать книги по всем моим курсам. В том числе и про особенности тестирования SOAP и REST API ;-)

Linux

- Уильям Шоттс: «Командная строка Linux» — простой язык, хорошие примеры.
- Скотт Граннеман: «Linux. Карманный справочник» — маленькая книжечка, но сколько интересного внутри!

Остальное

Регулярные выражения

- Бен Форта: «Регулярные выражения. 10 минут на урок» — ЛУЧШАЯ КНИГА по регуляркам. Интересно и доходчиво!
- Эрик и Элизабет Фримен: «Изучаем HTML, XHTML и CSS» — для автоматизации веба HTML надо знать.

Регулярные выражения нужны для фильтрации логов или небольшой автоматизации разных рутинных задач на уровне Блокнота. Примеры вы можете найти в моем блоге по названию «Автоматизация в блокноте».

Версионный контроль

- Eric Sink: «Version Control by Example» — Сравнение SVN, Mercurial и Git на одних и тех же примерах, очень доходчиво! Но на английском...
- Дэн Пилон, Расс Майлз: «Управление разработкой ПО» — тут всё обо всем про разработку ПО. В том числе про контроль версий. Отличная книга для не-ИТишника, чтобы понять, что да как.

С системой версионного контроля вы обязательно столкнетесь на любой работе. Сейчас уже даже фрилансеры осознали ее преимущества.

По логам, увы, книг пока нет. Ждите мою ;-) Хотя, возможно, я просто чего-то не знаю? Напишите мне на **ok.molechka@gmail.com**, посоветуйте отличную книгу, и я внесу ее в список для чтения с отсылкой на вас ;-)

Автоматизатор

О профессии

Основные задачи автоматизатора:

1. Автоматизировать тесты.
2. Разбираться в падениях автотестов.
3. Автоматизировать рутину.
4. Ревьюить код (unit-тесты разработчиков, автотесты коллег...).

Соответственно, для автоматизации вам нужно изучать языки программирования. Это может быть один язык или разные.

Постоянно возникают холивары: какой язык учить? Если вы придете в чат тестировщиков с таким вопросом, то получите кардинально разные ответы. Но есть общий совет, который работает в большинстве случаев: лучше всего начинать учить тот язык, на котором написано ваше приложение.

Какие это дает плюсы:

- вам помогут разработчики! Это самый главный плюс — вы все равно соберете кучу граблей, будете сидеть и думать: «А как это сделать? А тут почему не работает, вроде все верно?». Можно пойти к разработчикам и попросить помощи;
- вы будете понимать код приложения, хотя бы «простые» его части.

Разумеется, если приложение древнее, написанное на дельфи, не стоит писать тесты на нем же. Лучше изучить что-то поактуальнее.

На самом деле совершенно неважно, какой язык вы выучите. Всегда можно тестировать приложение черным ящиком, словно доступа к коду у вас и нет. И тогда совсем неважно, на каком языке пишете вы. Я рекомендую выбирать самый популярный на рынке, по нему и сообщество большое. Будет куда обратиться, если что-то не получается, а разработчики помочь не могут.

Ну и стоит определиться, хотите вы начинать с ООП (объектно-ориентированное программирование) или с процедурных языков (Python, PHP):

- ООП сложнее. Чтобы начать что-то писать на языках ООП, нужно кучу всего создать. Подумать об архитектуре, создать разные классы... Но они популярнее, потому что на них написаны программы, их знают разработчики;
- процедурный язык сильно проще в обучении, на нем можно сразу начинать писать код и тесты. Есть фреймворки для автотестов, тут проблем тоже не будет. А еще на нем хорошо автоматизировать рутину!

В общем, выбирайте и копайте уже эту тему.

Книги

Непосредственно по языкам программирования рекомендую книги серии Head First O`Really:

- Кэти Сьерра и Берт Бейнс: «Изучаем Java».
- Эрик Фримен, Элизабет Робсон: «Изучаем программирование на JavaScript».

Еще про языки программирования:

Ник Морган: «JavaScript для детей»— очень классно изложен материал, не только детям можно читать! При этом есть и сложные темы.

Плюс автоматизатору обязательно уметь работать в командной строке, базово общаться с линуксом и знать про системы версионного контроля (иначе где вы будете хранить свои автотесты?). Книги по этим темам:

- Скотт Граннеман: «Linux. Карманный справочник».
- Бен Форта: «Регулярные выражения. 10 минут на урок» — без них бывает сложно.
- «Version Control by Example» — сравнение SVN, Mercurial и Git на одних и тех же примерах, очень доходчиво! Но на английском...

И еще полезные:

- Дэн Пилон, Расс Майлз: «Управление разработкой ПО» — в целом о процессах.
- Эрик и Элизабет Фримен: «Изучаем HTML, XHTML и CSS» — для автоматизации веба HTML надо знать.

Нагрузочник

О профессии

Инженер по нагрузочному тестированию изучает реакцию программы на большую нагрузку. Это может быть:

1. Много пользователей веб-сайта — если он мегапопулярен, как Фейсбук, YouTube и похожие. Туда будет действительно МНОГО заходов каждую секунду.
2. Много запросов по API — если в системе есть это самое API, нужно проверить, какую нагрузку оно держит, сколько запросов в секунду. И как быстро работает программа при нагрузке по API.

Основной инструмент на первом этапе — JMeter. Разберитесь, как он работает, а потом нагружайте свое приложение. Если это нужно по ТЗ и если руководство одобряет!

Важное правило: чужие сайты тестировать на нагрузку НЕЛЬЗЯ! Только с разрешения автора!!!

Поэтому, если вы новичок и хотите просто «ради себя» попробовать изучить JMeter, установите себе какое-нибудь приложение локально и нагружайте его. Сайты в Интернете: интернет-магазины, почтовики, даже тестовый Users — не трогаем!

Если вы уже работаете и хотите для себя поизучать нагрузку, то, опять же, спросите разрешения у коллег. Вдруг у вас в конторе только одна тестовая платформа, а от вашей нагрузки она ляжет и заблокирует работу команды?

А вот если вы свою версию продукта поднимаете локально, то нагружайте сколько влезет!

Сейчас отдельная категория «нагрузочников» есть только в больших компаниях. Которым прям важно это и у которых это реальная проблема, а не просто хочушка клиента. Реальная задача: у Фейсбука или Wildberries, на которые заходят сотнями.

А просто хочушка — это когда приходит клиент, говорит, что хочет интернет-магазинчик. Но чтобы проверили качественно! А нагрузочное проводили? А сколько он выдержит? Ой, 100 запросов в секунду — это слишком мало, переделывайте!

Простой пользователь не понимает, что 100 запросов в секунду — это ОЧЕНЬ много. Редко когда заходят паралелльно столько людей. Но если про нагрузочное тестирование слышал, то просит провести (желательно бесплатно). Хотя если оценит трудозатраты, может и отказаться от этой идеи.

В любом случае начинать нагрузочное ради нагрузочного смысла нет. Если у вас на работе оно пока не надо — значит, время еще не пришло. Можно ради развития «для себя» потыкать, но и всё.

А вот если нужно нагружать, то стоит обсудить стратегию с разработчиками и вместе решать, кто и как это будет делать: они на уровне кода или вы через JMeter.

Из собственного опыта...

У меня на одном проекте нагрузочный тест написал разработчик. В самой системе он сделал отдельный триггер по подготовке данных. Нужно просто зайти в админку и нажать на кнопку «запустить триггер». И всё, тот пойдет по всем задачам:

1. Подготовит данные.
2. Выгрузит их в буферную таблицу.
3. Запустит стандартную процедуру забора инкремента.

Паралельно открываешь уже прописанный файлик в JMeter, чтобы проверить сразу две операции:

- загрузку данных из БД;
- загрузку данных через SOAP.

Ведь нужно проверить скорость работы, когда эти два сложных процесса идут в параллель!

Книги

К сожалению, не могу ничего посоветовать. Книг по нагрузке не знаю.

Безопасник

О профессии

Инженер по безопасности — это почти как хакер, только добрый. Ломает не чтобы воспользоваться «дырой» в безопасности, а чтобы на нее указать до выхода системы в продакшен. Ручной хакер ;-)

Про эту профессию я тоже мало чего могу рассказать. Просто потому, что в эту область надо закапываться. Она сложная, это не просто «раз, и уже умею». И если тест-дизайном занимается любой тестировщик, автоматизацию он тоже может пощупать, а вот нагрузку с безопасностью надо изучать. Причем на каких-то специализированных курсах.

Что «простого» можно изучить:

1. SQL-инъекции.
2. XSS-атаки.

Для первого надо знать SQL, а это нынче даже ручники должны уметь. А для второго — HTML, что, опять же, нынче полезно знать — чтобы тестировать и автоматизировать веб.

Но это простые атаки, а есть еще сложные, когда злоумышленник может:

- влезть в чужой аккаунт на почте и прочитать личную переписку;
- влезть в чужой аккаунт, на котором лежат деньги, и потратить их;

- снять в банкомате «чужие» деньги;
- ...

Вот чтобы это было невозможно, нужно внимательно тестировать на безопасность все операции, связанные с деньгами. А если вы — простой интернет-магазинчик, то для оплаты надо использовать проверенные средства, а не писать своё на коленке, это плохо закончится.

Книги

- Тобиас Клейн: «Дневник охотника за ошибками» — очень интересная книга от профессионального охотника на уязвимости, который выбрал 7 самых интересных из найденных им ошибок за последние несколько лет и подробно описал: как нашел, что делал, в какую сторону мыслил!
- Майкл Саттон, Адам Грин и Педрам Амини : «Fuzzing. Исследование уязвимостей методом грубой силы» — о фаззинге от первопроходцев. Авторы создали отдельный сайт, куда выложили фаззеры, написанные для книги. Бери да юзай! А в книге описано, зачем и как.

Тестировщик usability

О профессии

Основные задачи тестировщика юзабилити:

1. Проверять документацию на «понятность» простому пользователю.
2. Проверять тексты на сайте на понятность.
3. Проверять продукт на интуитивность, понятность, обучаемость.

Все мы немного тестировщики юзабилити. Ведь такой тестировщик должен думать о конечном пользователе: что будет ему неудобно или непонятно? А все мы являемся пользователями: интернет-магазин, оплата картой, банк онлайн...

И если не каждый день, то хоть иногда напарываемся на дико неудобную штуку. Например, чтобы заплатить за квартиру, нужно пройти пять разных экранов приложения — и так каждый раз! Или в игрушке, чтобы просмотреть рекламу и получить за это плюшку, надо нажать OK в двух всплывающих окнах подряд...

Как тестировщики, мы просто обязаны сообщать о таких «багах». Это называется *баг юзабилити*. Но будьте осторожны с ними — если что-то не нравится лично вам, это еще не значит, что проблема действительно есть.

Ошибка новичка: узнав о существовании юзабилити, тут же находит кучу «мегакритичных» проблем в продукте. И очень возмущается, когда задачи закрываются как «не баг и исправляться не будут», по крайней мере, у нас на курсе:

— Но МНЕ же неудобно! Значит, всем неудобно, пользователи уйдут искать конкурентов, и вы потеряете клиентов!

Помните про антипаттерн «обиженные зайчики»? Вот он особенно применим к багам юзабилити. Нет, конечно же, *нужно* сообщать о таких проблемах. Только без пафоса в стиле: «Да это точно баг, еще и критикал!». И лучше сначала устно обсудить с командой. Они могут вам объяснить, почему сделано именно так, а не иначе.

Я рекомендую сначала изучить книги типа «Психбольницы» Купера, а потом только заводить баги юзабилити.

А еще важное правило для юзабилити: самому работать со своим продуктом. Только как пользователь, а не как тестировщик. Это меняет угол зрения.

Из собственного опыта...

Мне казалось, что форма выверки дубликатов очень удобная. Я ее часто тестировала, в том числе и «как пользователь», — ну все хорошо же!

А потом поехала в банк к заказчику и стала решать его задачи. У них операционисты просматривают эти формы по 100 в день, вот и я пару дней принимала решения. И тут же выявилась куча замечаний о том, что именно неудобно.

А ведь до этого казалось, что я смотрю как пользователь! Но нет, чтобы смотреть как пользователь, нужно делать реальную задачу на реальных данных, иначе всё не то.

Книги

Самые шикарные и рекомендуются к прочтению любому тестировщику:

- Алан Купер: «Психбольница в руках пациентов» — лучшая книга про usability. Стоит читать даже тем, кто этим видом тестирования пока не занимался.
- Дэвид Платт: «Софт — отстой! И что с этим делать?» — офигительная книжка по usability. Ничуть не хуже «Психбольницы», при этом в два раза веселее ;-)
- Стив Круг: «Не заставляйте меня думать» — шикарная книга, читается легко и быстро!

И еще парочка, где вопросы юзабилити рассматриваются чуть глубже:

- Дональд Норман: «Дизайн привычных вещей» — просто о сложном. Автор на примере бытовых вещей показывает ужасный дизайн. Книге куча лет, но она до сих пор актуальна!
- Джеф Раскин: «Интерфейс» — показывает, какие интерфейсы делать не надо и почему, например, бесполезны подтверждающие диалоги.

Тест-менеджер

О профессии

Что нужно понимать о работе менеджера? — это НЕ тестирование!

Ваши приоритеты:

1. Найти таланты.
2. Развить их.
3. Распределять задачи.
4. Не мешать! ;-)

Основные задачи тест-менеджера:

- помогать новичку влиться в процесс;
- курировать развитие людей своего отдела;
- стыковать нужных людей (если, например, нужен разработчик — к кому обратиться);
- переводить баги в разработку или закрывать как «не баг»;
- мотивировать команду;
- набирать команду;
- улучшать процессы;
- отвечать за результаты тестирования.

Да-да, менеджер не тестирует. По крайней мере, в идеальном мире не должен. У него просто не будет оставаться времени на простую летучку.

Менеджеру важно уметь замотивировать сотрудников на работу. Вовремя заметить потухший огонек в глаза, чтобы дело не дошло до увольнения или процессионального выгорания. Он должен помогать команде развиваться — обсуждать с каждым, чего тот хочет достичь, куда расти, какие шаги следует предпринять в этом направлении.

Если компания большая, менеджер может выступать связующим звеном между отделами:

1. Тестировщик заводит задачу на тест-менеджера.
2. Тот со своим знанием продукта ее оценивает (стоит ли делать или нет), потом переводит на старшего разработчика.
3. А старший разработчик уже решает, кто конкретно ее сделает.

Когда людей много, такой процесс оправдан. Когда людей мало, ты сам знаешь, на кого повесить задачку.

А еще очень важно в работе аутсорс-команды — чтобы кто-то отвечал за результат. Потому что заказчику неинтересно бегать по двум-трем-пяти тестировщикам и выяснять, кому задать вопрос. Ему нужна точка входа, которую и выполняет менеджер. Так что на аутсорсе менеджер нужен даже для небольшой команды.

И, конечно, нужно уметь подбирать людей! Проводить собеседования, искать таких тестировщиков, которые усилят твою команду, и прочая...

Книги

Лучшие книги для руководителя:

- Том Демарко: «Deadline. Роман об управлении проектами» — шикарный роман!
- Серия книг Элияху Голдратта: «Цель», «Цель-2», «Цель-3». Все его книги по-своему прекрасны, и руководителю их читать обязательно!
- Фредерик Лалу: «Открывая организации будущего» — титанический труд, но его нереально интересно читать! О том, как выглядят организации будущего на примерах тех, кто уже использует принципы самоорганизации.

О процессах, причем именно в ИТ-области:

- Джефф Каролло, Джеймс Уиттакер, Джейсон Арбон: «Как тестируют в Google?» — о процессе разработки в компании Google. Стоит читать не новичкам, будет больше пользы.
- Джефф Сазерленд: «Scrum. Революционный метод управления проектами» — отличное введение в Scrum. Подробно: что это такое, зачем его внедрять и как именно это делать.

- Джез Хамбл, Дейвид Фарли: «Непрерывное развертывание ПО» — техническая книга, но говорит о правильных вещах. Как заменить ручное развертывание на автоматизированное.

И не только о процессах, просто полезное от умных людей:

- Джоэл Спольски: «Джоэл о программировании» — там не только о программировани, но и о тестировании, и о багах, и много о чем еще!
- Джоэл Спольски: «Джоэл. И снова о программировании» —— вторая часть, не менее замечательная ;-)

О мотивации сотрудников:

- Маркус Бакингем, Курт Коффман: «Сначала нарушьте все правила» — лучшая книжка! И рассказывает о том, как и почему люди хорошо выполняют свою работу.
- Светлана Иванова: «Я слышу, что вы думаете на самом деле» — еще одна отличная книга-тренинг Светланы Ивановой. Тонкая, но емкая, про подтекст в речи собеседника.
- Дэниел Пинк: «Драйв! Что на самом деле нас мотивирует» — вопросы мотивации в ярких примерах.

О том, как проводить собеседования:

- Светлана Иванова: «Мотивация на 100%» — потрясающая книга-тренинг. Все доступно и интересно описано + задания на самопроверку в конце каждой главы.
- Маркус Бакингем, Дональд Клифтон: «Добейся максимума» — клевая! Заставляет задуматься о своих талантах и сильных сторонах. Это очень полезно. А перед собеседованием можно задуматься, человек с набором каких талантов тебе нужен.

О лидерстве:

- Патрик Ленсиони: «Пять пороков команды» — отличный бизнес-роман про лидерство!
- Максим Недякин: «Искренний сервис» — тоже ВАУ-книга, она для руководителей, но реально вдохновляет на то, чтобы делать чуть больше, чем надо.

О психологии:

- Роберт Чалдини: «Психология влияния» — просто ВАУ! Читать всем! О психолсгических приемах влияния на наше сознание.

Об обучении новичков:

- Джули Дирксен: «Искусство обучать» — чумовая книжка! Яркая, интересная и мегаполезная! Всем тренерам читать обязательно!!

- Ли Лефевер: «Искусство объяснять» — тоже шедевр! Книга рассказывает, как преодолеть «проклятье знания» и объяснять сложные вещи простыми словами.
- Джордж Леонард: «Мастерство. Путешествие длиною в жизнь» — шикарная книга о том, как учить, и о том, как учиться.
- James Marcus Bach: «Secrets of a buccaneer-scholar» — о том, как учиться. От известного тестировщика. Не обязательно ходить на платные курсы или учиться в школе. Можно быть «пиратом» и учиться самому!

Аналитик

О профессии

Основные задачи аналитика:
1. Поговорить с заказчиком.
2. Понять его боль.
3. Предложить решение.
4. Сформулировать его.
5. Донести до разработчика.

Аналитику очень важно уметь внятно излагать свои мысли и писать понятно. Но самое важное умение — понять, что именно заказчик хочет! Потому что иногда заказчик сам придумает решение своей проблемы и несет его вам, готовое. И тут есть два варианта:

1. Сделать, как он просил, без вопросов «а что, мы это можем?» Можем, оценили, денег запросили и вперед! Вот только решение может быть неоптимальным и его проблемы не решит.
2. Уточнить, зачем это нужно и как будет использовано — а вот это правильный подход. Возможно, его проблема уже решается существующими методами, и мы подскажем бесплатное решение. А возможно, знание «зачем?» поможет вскрыть те проблемы, о которых сам заказчик не подумал, но которые надо предусмотреть.

Надеюсь, и так понятно, где уровень новичка, а где — мастера? ;-)

Книги

Как выяснять требования и выкидывать лишнее:

- «Интерком о продакт менеджменте» — книга про ИТ, что самое ценное! Как сказать фиче «нет» и грамотно составить roadmap.

О том, как хорошо писать:

- Людмила Сарычева, Максим Ильяхов: «Пиши, сокращай» — прочитайте сами и аналитику подарите!
- Пол Смит: «Мастер историй» — суперкнига! Набор из сотни историй, которые можно использовать в работе.
- Карл Вигерс: «Разработка требований к программному обеспечению» — в русском переводе очень сложно читать, но аналогов мало.

Оратор

О профессии

«Что?» — спросите вы. — «Какой еще оратор??».

А вы знаете, тестировщику это тоже бывает полезно! По разным причинам:

1. Если вы будете общаться с заказчиком или вырастете в аналитика, или у вас в компании вы «и швец, и жнец». Нужно уметь продвигать свои идеи!
2. Если вы будете обучать персонал заказчика работе со своим ПО — у меня на одном проекте это делал тестировщик, например.
3. Если вы будете обучать новичков, мастерство спикера тоже пригодится.

4. Если вы будете выступать на конференциях — чтобы поделиться с аудиторией своими достижениями, создать себе имя или обсудить насущную проблему.

Так что подумайте над этим. И если что надумаете, то далее мой список рекомендаций к прочтению!

Книги

Топ-3 авторов:

1. Нэнси Дуарте.
2. Радислав Гандапас.
3. Гарр Рейнольдс.

Читать в таком порядке! У каждого автора есть, как минимум, пара книг, так что теперь мой список вау-книг:

- «Slide:ology» — обожаю Нэнси Дуарте! Считаю ее книги лучшими!
- «Resonate. Захвати аудиторию своей яркой историей!» — и снова Нэнси Дуарте. Когда придумываю тему доклада, так всегда и вижу руку, рисующую линию жизни и вопрос: »Хочешь ли ты, чтобы твоя идея жила?».
- Гарр Рейнольдс: «Презентация в стиле дзен» — после Нэнси Дуарте лучшая книга. Много примеров, ссылок и доп. материалов. А еще классный дизайн и интересное чтение!
- Радислав Гандапас: «Камасутра для оратора» — легко читается, много полезного!
- Радислав Гандапас: «К выступлению готов» — отличный конструктор для выступающего!
- «iПрезентация» — рассказ на примере выступлений Стива Джобса. Когда делаю презентацию, передо мной теперь все время стоит Стив Джобс, «крутящий» пальцами свою идею «3 в 1» при рассказе про iPhone.

В книге Игоря Манна и Рената Шагабутдинова: «Бизнесхак на каждый день» — также можно почерпнуть всякое интересное, в том числе, как делать презентации и выступать.

Мои бизнес-хаки

1. Купите личный презентер

Только Logitech, вот такой:

Он самый лучший. Легкий, удобно держать в руке, легко переключать слайды и показывать на что-то указкой. Его используют на конференции SQA Days, и в свое время мне рекомендовали именно его.

Я купила себе такой лет... восемь (?) назад. Ни о чем не жалею — он однозначно стоит своих денег.

Зачем нужен личный, если на конференции дадут «общий»? А если не дадут? Не на всех конференциях есть презентеры. А еще они могут сломаться. У меня всегда с собой личный «на всякий случай».

Да и репетировать проще сразу с презентером. Так ты можешь отрабатывать свои движения, а не быть прикованным к компу и мышке.

Из собственного опыта...

Вот мы с моим коллегой Арсением ездили на конференцию DUMP: он выступает в mobile-секции, я — в тестировании. Репетировали вместе на работе. Исходно должен был быть общий доклад, но его в итоге не приняли, а репетировать вместе мы продолжили. Подтверждаем — с презентером хорошо, без презентера плохо!

У нас в офисе раньше был «офисный» презентер. Для всяких внутренних конференций, обучений и т. д. И вот мы с Арсением собираемся на репетицию, включаем слайды... А презентера нету! o_O И в итоге я стою около компа и уныло клацаю на «→». Бееееe ;-(

Так что лучше иметь свой личный. В том числе для репетиций.

В общем, штука классная и нужная, если вы готовитесь выступать! А если будете выступать не один раз, то личный презентер must have.

2. Сделайте визитки с фото

Когда возвращаешься домой с SQA Days, смотришь на полученные визитки и недоумеваешь: кто все эти люди? Да, если какой-то докладчик был живой и яркий, он запомнится. Но помнить сразу 5–10–20 людей нереально. Добавьте фотографию на личную визитку — позвольте легко вас узнать даже спустя полгода.

На визитке должно быть:

- Фото — вспомнить, кто ты вообще такой.
- ФИО — как к тебе обращаться.
- Кто ты? Разработчик? Тестировщик? Аналитик?
- Как с тобой связаться?

Вот пример моей первой визитки. Не лучший вариант, но хотя бы с фото, можно вспомнить «кто это такой» :-)

3. Приезжайте пораньше

Осмотрите зал. Где вы будете стоять? Где находится ноутбук с презентацией? Сколько мест в зале? Лучше узнать все заранее, это придаст уверенности в себе.

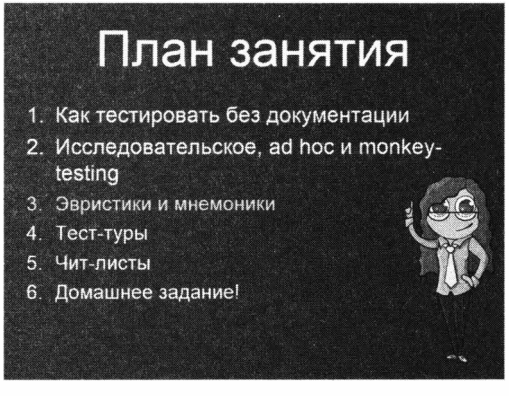
Загрузите презентацию на компьютер с утра. Пролистайте ее всю и посмотрите, как она отображается на экране. Может, какое-то слово уедет за пределы экрана — можно быстро подправить. Может, шрифты не поддерживаются. Видео не загружается, презентация битая...

Если обнаружить проблемы перед выступлением, будет fail и паника. Если приехать заранее, то успеете все не спеша поправить. И не придется краснеть перед зрителями.

Главные хаки: презентер да визитки. Остальное за вами. Репетируем, репетируем и еще раз репетируем. Но это не мелкий хак, а большой и глобальный совет.

4. На каком мы этапе?

Зрителям нравится осознавать, на каком этапе выступления они находятся. Поэтому встраивайте в слайды разделы. Сделайте общий слайд «о чем мы сегодня поговорим» и периодически его показывайте: вот мы прошли первый пункт и теперь говорим о втором. Из шести.



Этапы — для зрителей. Но и для докладчика есть лайфхаки — как помнить о времени. Вот вам бизнес-хак от Максима Дорофеева: сделать внизу слайда небольшие кружочки (видимые только вам) с анимацией исчезновения через 15 секунд и запуском таймера после исчезновения последнего кружочка. Если слайд на одну минуту, в углу видите четыре кружочка. Осталось только два — у вас 30 секунд.

Крутой бизнес-хак! С этапами для зрителей я давно знакома, а вот находка Максима очень познавательна. Правда, я не смотрю на экран ;-) Но для лекций такой может пригодиться.

5. Используйте реквизит

Принесите что-то, связанное с темой выступления. Положите на видном месте. Профит! Зрители заинтригованы и будут весь доклад ждать, когда же вы используете «эту штуку».

Эта мысль не дает мне покоя. Очень хочется попробовать! Но пока нет идей, что можно принести ;-) Однако на будущее учту однозначно!

Сервисы

Самые главные — с бесплатными фоточками для ваших слайдов:

- Free Photos: http://www.freedigitalphotos.net.
- Free Images: http://www.freeimages.com.
- Freerange: https://www.freerangestock.com.
- Free Photos Bank: http://www.freephotosbank.com.
- iStock: http://www.istockphoto.com/ru.

Швец, жнец, на дуде игрец

О профессии

Развиваться в каком-то одном направлении — это хорошо. Но бывает и так, что тестировщик на работе и швец, и жнец, и на дуде игрец. Например, в компании может не быть аналитика, поэтому ТЗ пишет тот, кто первый освободится для задачи. Или как у нас, например, в компании развит горизонтальный рост, и тестировщик делает кучу всего!

И тогда ваш список задач может существенно расшириться:

1. Тестировать функционал.
2. Писать автотесты.
3. Проводить регрессию.
4. Модерировать планирование или рестроспективу.
5. Отвечать на вопросы заказчика (первая линия поддержки).

6. Писать ТЗ.
7. Оценивать риски.
8. Проводить обучение.
9. ...

Простой пример: даже если вы не собираетесь писать автотесты, вам все равно полезно уметь читать код. Вы можете провести ревью unit-тестов разработчиков, вы можете вручную что-то «подкорячить» в билде... А потом продолжить тестировать вручную!

Или команда только начнет внедрять всякие гибкие процессы, и будет нужен scrum-мастер, куратор планирований и ретроспектив. И тут уж кто вызовется, потому что исходно таких нет.

Или аналитик занят, а доработка от заказчика очень простая: можно самому написать ТЗ, а можно сидеть и ныть, что все тебя игнорируют.

Мне нравится подход швеца-жнеца, а вот понравится ли он вам — думайте сами. Если нет, то лучше идти работать в банки, страховые или другие крупные компании с четко отлаженным процессом. И на собеседовании заранее уточнять, что будет входить в ваши обязанности.

Но зато у вас будет быстрый горизонтальный рост. Ведь вы всегда узнаете что-то новое и делаете задачи, которые вам ранее даже не снились ;-)

Прокачка мощная, особенно для джуниоров. Более крутые просто берут себе задачки посложнее и больше «трогают» код.

Книги

Тут уже общее развитие. Даже если вы ручник, поинтересуйтесь языком программирования. Посмотрите в сторону тестирования API и прочая.

Например, технические книги:

- Эрик Рэй: «Изучаем XML» — очень крутая и полезная книга! Про XML с нуля, поймет любой.
- Уильям Шоттс: «Командная строка Linux» — все, что надо знать о линуксе.
- Плюс все книги из серии Head First O`Really по разработке ПО.

«Мне некуда расти в компании!»

Первую работу особо не выбирают. Куда взяли, там и копошусь. Для опыта подойдет любое место, даже если вы просто будете прогонять чужие чек-листы раз за разом.

Но потом, конечно, работа перестает приносить радость... Когда нет интересных задач, когда каждый день одно и то же, кажется, что это тупик.

А ведь хочется, чтобы на работе было УХ! Чтобы вставать утром и сразу туда бежать. Потому что там интересно. Потому что ты важен, на тебя рассчитывают. И, вообще, работа вдохновляет!

Что же делать в такой ситуации? Самый простой путь — менять работу. И иногда именно он и оправдан, потому что тут развивайся, не развивайся — всем начхать.

Но лично меня немного коробит нытье по поводу «Ах, мне некуда расти! Но работу сменить пока не могу, поэтому буду тут. Но как тут плохо ах-ах-ах». Прям-таки некуда? Прям-таки ты за полгода уже достиг своего потолка?

Ну и что, что тебе не дают других заданий? Это как-то мешает *тебе* развиваться? Есть куча вариантов саморазвития!

Бывает так

А хочется вот так

Применяйте техники тест-дизайна

На самом деле в большинстве случаев даже применение простейших граничных значений да классов эквивалентности может дать много новой информации о проекте.

Я лично проверяла курс по тест-дизайну у Алексея Баранцева. И читала домашние задания людей с 5-летним опытом, которые не в состоянии протестировать формочку на 2–3 поля. Не могут проверить границы для нечислового поля, да и для числового тоже.

А ведь это простейшие проверки! Что уж говорить об интересных и заковыристых багах? Чтобы их находить, надо развивать фантазию и уметь применять хотя бы базовые техники. И студенты потом говорят: «Вау, я на своем проекте баги нашел!». Хотя вроде и так все это знали, просто на курсе тренер на примере ДЗ показал, что «вот тут и тут еще надо проверить». И ой, оно сломалось!

Вы не поверите, какие заковыристые баги можно найти на классах «ноль–не ноль», «нижний регистр–разный регистр», «русская раскладка–английская раскладка» и других. А ведь казалось бы!

Поэтому я не верю в то, что вы прям так хорошо тестируете уже сейчас. Тем более, что вы всего ничего в компании работаете. Да за полгода от человека, особенно новичка, только только начинается отдача для проекта! Ведь, чтобы понимать, ЧТО именно нужно проверять, надо знать проект: его взаимосвязи между модулями, его разработчиков, его фишки, его функционал...

Попробуйте изучить технику тест-дизайна и применить к своему проекту. Вот прям взять и пойти применять, а не просто почитать и покивать головой «Да-да, это логично, я это и так делаю». Не делаете. Или делаете, но не везде. Поэтому берем технику и сосредоточенно пихаем везде, где только можем. Это приносит результат!

Создавайте чит-листы

Еще вариант нытья: у нас нет тестов, документации и вообще полный хаос!

Ну так вперед — создайте тесты. Что вас останавливает? Это же ваша работа как тестировщика, тесты придумывать и проводить! А еще их можно записать — для будущих поколений. Что? Времени нет? Ну так не надо расписывать каждое нажатие на кнопочку, никто не требует подробнейших тест-кейсов. Пишите чек-листы!

Или чит-листы — так называют чек-листы по конкретному функционалу. Своего рода подсказка, что смотреть. Называйте как хотите, но смысл один. Если на вашем проекте нет нормального описания тестов, надо не ныть, а исправлять ситуацию.

Пишите документацию

Проблема: на проекте нет нормальной документации.

Решение: начните сами ее составлять. Да-да, вы! Не Вася из соседнего отдела, а вы.

Это принесет кучу плюсов:

- коллеги начнут использовать ваш труд, поймут, что с документацией все же лучше, — сами начнут ее писать или задумаются об аналитике;
- вы поймете, что это такое — писать документацию. Какие моменты при этом пропускаешь, о чем забываешь. А это поможет вам лучше ее тестировать.

Я не предлагаю сесть и полгода скурпулезно описывать каждую кнопочку. Выдали задачу на тестирование? Вам все равно нужно узнать, как она работает. Если документации нет — то узнаем устно у знающих людей. А потом записываем. Кратенько. И это уже остается на будущее. И процесс становится немножечко лучше.

Улучшите процесс

Почему баги на проде находятся? Почему релиз вовремя закрыть не успели? Что мешало лично вам? Подумайте об этом. И попробуйте предложить решение.

Конечно, в идеале думать о минусах релиза должна вся команда на ретроспективе. Но, может, у вас вообще нет ретроспектив. А может, они проводятся «на отвали» или просто очень сумбурно. Или коллеги зажатые и не хотят обсуждать проблемы: ведь атмосфера должна быть открытой, чтобы можно было говорить о своих косяках. А если тебя будут за них гнобить, то и рассказывать не хочется.

В общем, если ретроспективы не помогают, подумайте сами о том, что могло бы сделать вашу жизнь лучше. И сделайте! Или попросите разработчиков вам в этом помочь.

Учитесь автоматизации

Ну вам же скучно на текущей работе, так? Вы хотите новые интересные задачи и вот это вот всё. А на работе с кучей интересных задач требования к тестировщикам совсем другие. Там недостаточно просто тыкать на кнопочки. Там могут требовать уметь читать код, писать автотесты, владеть регулярными выражениями, самому настраиваить виртуалку... Так что учитесь этому заранее!

Что вы можете начать делать в качестве автоматизации:

1. **Читать код своей программы** — смотреть в системе версионного контроля изменения (если вас пустят к коду). Это иногда помогает замечать мелкие косяки разработчика. А если ребята исходно пишут хороший код, там будут и понятные новичку места.
2. **Писать автотесты на свой проект** — если вас не пускают в код, то и ладно! Автотесты методом черного ящика (через GUI) еще никто не отменял. Учитесь этому, потому что в вакансиях автоматизаторов это востребовано.
3. **Автоматизировать рутину** — у вас есть задачки, которые бесят? Автоматизирйте их! И вам интересно будет сделать задачку, и больше не придется страдать

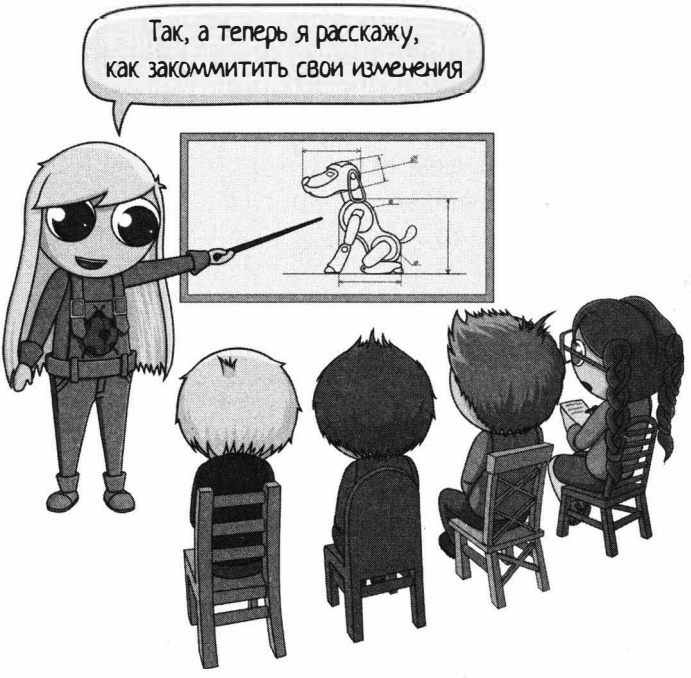
Ну а теперь скажите мне, кто может помешать вам автоматизировать рутину? Или GUI? Правильно, никто! Так что хватит ныть «меня все равно в код не пускают», для изучения автоматизации это просто не нужно.

Организуйте внутреннее обучение

Обучите своих новичков:

- как работать именно с вашей программой — кто, кроме вас, знает все ее фишки и места скопления багов?
- прочитайте лекцию на выбранную тему: то, чего явно не хватает команде. Или то, что им интересно. Или то, что вам интересно. Все что угодно!

Хочешь разобраться в теме? Изучи ее сам и объясни другому! Обучая коллег, вы сами лучше узнаете тему (ища ответы на каверзные вопросы, например). А еще это новая интересная задача, которая поможет отвлечься от текущей рутины.



Из собственного опыта...

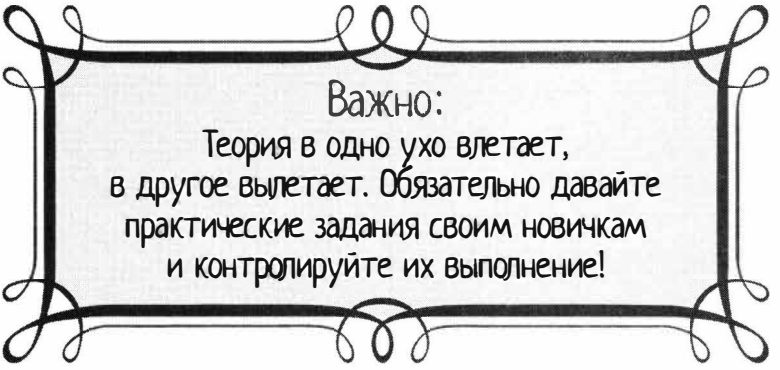
Когда у нас был костяк из нескольких тестировщиков, то знания передавались из уст в уста. Просто «старший» коллега садился рядом и всё показывал-рассказывал. Но потом мы стали набирать сразу по 2–3 человека. Рассказывать каждому?

Мы стали проводить обучения для новичков. Что интересно, послушать про наш проект приходили и опытные товарищи, и коллеги с других проектов: им было интересно. А уж общие вещи типа «как работать в Mercurial» или «основы Linux» собирали аншлаг!

Попросите команду поучаствовать! Пусть разработчик расскажет, как правильно собрать проект или как работать с системой контроля версий. А аналитик — как правильно написать ТЗ. Ведь, может быть, завтра вам придется написать ТЗ, потому что все будут заняты. На что обращать внимание? Что не забыть учесть? Расскажите об этом!

Если сотрудники выезжают к заказчику — расскажите, как себя там вести. Если проводят внедрения — расскажите про этот процесс. Ведь совсем скоро ваши новички заменят вас в этой работе. Сохраните свои знания — расскажите коллегам и запишите на видео для будущих поколений.

Важно: теория в одно ухо влетает, в другое вылетает. Обязательно давайте практические задания своим новичкам и контролируйте их выполнение.



А еще можно проводить свои мини-конференции! Это когда коллеги выступают только друг перед другом. Так меньше паники, все же свои, да и народу мало. Зато можно попрактиковаться выступать + рассказать и услышать что-то полезное.

Из собственного опыта...

Мы проводили конференцию EducationDay. На ней любой мог выступить с интересной ему темой. Рассказать коллегам что-то новое и интересное.

Формат

Конференция длится целый день. Посещение докладов — по желанию, необязательное. НО! Участвуют только те, кто делает доклады. Этот пункт подстегнул выступить даже тех коллег, которые не очень любят общаться публично ;-)

Участники выкладывают в confluence (наша корпоративная википедия) список тем, на которые они готовы выступить. Остальные участники (только те, кто тоже будет выступать, что важно!) голосуют за доклады. Какой набрал больше всего голосов — на ту тему и готовится выступление.

Доклад должен быть коротким: 15 минут рассказ, 5 минут отводится на вопросы. После каждых трех докладов идет перерыв на 15 минут. В середине устраивается обед. Дело происходит в офисе.

Список тем

Программа первой конференции оказалась неожиданной!

- Проектирование интерфейсов: кому, зачем и как.
- Матчасть киберпанка (public key infrastructure).
- Практическое применение различных инструментов в работе с большими и очень большими данными.
- Настройка гетерогенных сервисов между Oracle и mssql/excel/dbf.
- Анализ данных (логов) с помощью grep, awk, R.
- Про rspec и Cucumber.
- Rocket Science или etl средствами bash.
- ...

Выступали и аналитики, и тестировщики, и разработчики, и даже админ. И это было круто и интересно! А главное: никуда не надо ехать. Вот разве что начальство должно одобрить конференцию на целый день для скольки-то коллег.

Ваша задача — убедить начальство, что это окупится. К тому же это дешевле, чем отправить сотрудников на внешнюю конференцию.

Соберите книжный клуб

Почему бы и нет? Если в команде есть люди, которые любят читать, но не ведут при этом блог, — отличный повод пообщаться с ними. Можно читать одну книгу в параллель, а потом обсуждать ее. Кто что вынес для себя интересного?

Это прикольно, потому что ты вынес одно, а коллега — другое. И слушая его рассказ, внезапно осознаешь, что сам об этой идее даже не подумал при чтении. А она интересная!

Можно собираться за обедом, тогда не придется тратить послерабочее время. И поели, и новостями обменялись — профит!

Выступите на конференции

И снова куча плюшек:

- вы прокачиваете ораторские навыки;
- вы рассказываете, как у вас в компании классно, что помогает найти работников (если у вас правда классно);

- вы рассказываете, какие у вас есть проблемы и как вы их решали — а потом получаете полезный фидбек, что еще можно сделать;
- вы «продаете» себя — полезно тренерам или людям, которые сами ищут работу;
- и да, вы можете найти работу на конференции ;-)
- а еще это разгрузка, другой вид деятельности, возможность отвлечься.

Я рекомендую идти именно спикером. Так вы сможете завязать больше контактов. Но если рассказывать вам нечего или вы стесняетесь, то просто езжайте на конференцию и слушайте «а как у других».

И если хотите сменить работу, идите туда, где классные процессы. А это видно по докладам. Ведь ребята в докладах часто говорят «а как у нас».

Подведем итоги

- Нет тестов? Создай!
- Продалбываются задачи? Реши, что делать, продай идею команде.
- Надоел поиск? Поменяйся функционалом с коллегой.
- Автоматизируй рутину.
- Создай книжный клуб.
- Учи новичков.
- Съеди на конференцию.
- Выступи на конференции.
- ...

Возможностей для роста целая куча. Даже внутри такой компании, где «всем на тебя плевать и задачи унылые, и в код не пускают». Нарабатывайте опыт, развивайтесь для себя. А потом да, меняйте работу.

Домашнее задание

Подумайте о том, какое направление развития вам наиболее интересно.

1. Выпишите навыки для развития.
2. Расставьте приоритеты
3. По трем первым по приоритету укажите ближайший шаг (с датой).

Шаг — это небольшое действие, которое можно выполнить относительно быстро.

- **Плохой шаг** — «научиться автоматизации», он не по SMART ;-)
- **Хороший шаг** — прочитать первую главу в книге «Изучаем Java» и выполнить задания в ее конце к вечеру пятницы.

Глава 15
ВСЁ ОБО ВСЁМ

Точнее, чуть-чуть о всяких страшных словах. Вы не научитесь SQL-у после прочтения этой главы, но хотя бы будете знать, что это такое.

В этой главе расскажу:

- что такое API;
- что такое база данных;
- что такое CSV (система контроля версий);
- что такое сборщик продукта;
- что такое командная строка;
- что такое Linux;
- что такое regexp.

Несколько вступительных слов

Здесь я кратко расскажу о таких страшных словах, как API, SQL, Maven... Просто чтобы вы знали, что это такое. Это не значит, что в одну главу я впихаю все знания по этим темам. Нет. Каждую тему нужно гуглить и изучать отдельно.

Более того, когда я дописала эту главу, у меня получилось в ней 303 страницы! Напомню, это еще кратенько, без глубокого копания тем. Потому что по каждой теме можно написать отдельную книгу (и по некоторым они уже написаны).

В этой книге уже хватает объемных глав, поэтому я решила вынести эти 300+ страниц в отдельную книгу: «Сложные ИТ-термины на простом языке»[175]. А еще — в цикл статей на Хабре, ссылки на которые вы можете найти на странице книги.

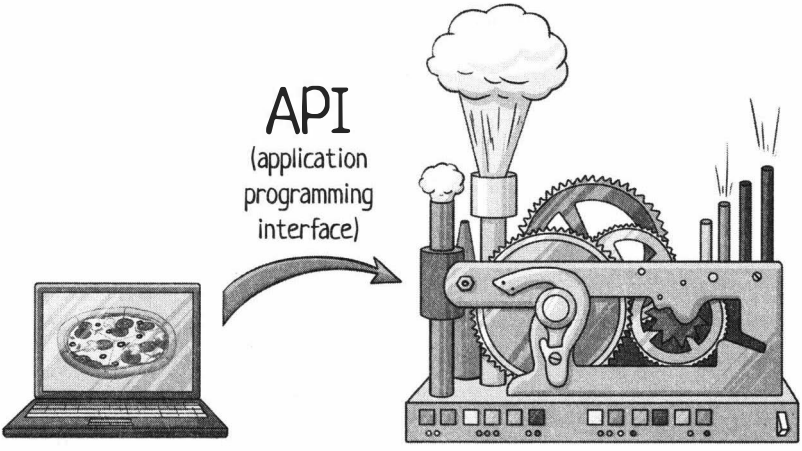
Здесь же, в заключительной главе для начинающих тестировщиков, я оставлю только самое основное. То, что поможет вам перестать пугаться «страшных» слов ;-)

Это может пригодиться на собеседовании. По крайней мере, вы поймете, о чем идет речь. И честно скажете: «Эту тему не копал глубоко, знаю только то и это». Но если есть общее понимание — это уже лучше, чем ничего!

Что такое API?

Вот, допустим, решила я заказать себе пиццу. Открыла интернет-магазин, выбрала пиццу и нажала «Купить». Так вот, интернет-страничка магазина — размер картинки с пиццей, цвет кнопочки «Купить», ее месторасположение — это всё GUI, Graphical User Interface. Или графический пользовательский интерфейс. Интерфейс, с которым работают пользователи.

Но как только я нажала на кнопку в GUI, в работу вступает другой интерфейс — API, Application Programming Interface. С ним работают программы.



Это то, что происходит под капотом. Тут как с машиной: вы нажимаете кнопочки, поворачиваете рычаги, а под капотом там ого-го какая система! Так что вы всегда тестируете через API, даже если никогда не слышали этого слова. Потому что API скрывается под любым графическим интерфейсом.

Но при этом, если вы в вакансии видите «умение тестировать API», то тут надо различать варианты:

- автоматизировать на уровне API — для этого надо знать язык программирования;
- вызывать внешнее API — обычно SOAP или REST.

API может вызвать программа, но можете сделать это и вы. С помощью таких инструментов, как SoapUI, Postman, curl... Давайте посмотрим, чем различаются самые популярные версии API.

SOAP & REST API

SOAP

SOAP используется больше в enterprise-системах — больших банках или страховых. У него есть ряд преимуществ:

- он работает по более защищенному протоколу HTTPS;
- он формализованный, подчиняется ряду правил;
- может быть полностью описан через WSDL (это документ на языке XML, который рассказывает про доступные SOAP-методы).

Есть и минусы:

- SOAP отправляет много метаданных;
- непопулярен для разработки Web и Mobile;
- работает только с XML.

Вручную его вызывают обычно через инструмент SoapUI.

REST

REST активно используется в веб-разработке. Он:

- простой в поддержке;
- прозрачно разделяет клиент и сервер;
- не давит стандартами — здесь всё на усмотрение разработчика, есть только рекомендации «как лучше»;
- работает с разными форматами (XML, JSON...).

Но есть и минусы:

- работает только по HTTP (незащищенный протокол);
- сложно настроить безопасный обмен данными.

Форматы передачи данных по API

Наиболее популярные форматы:

- XML;
- JSON;
- простой текст.

Вот, допустим, вы сохраняете нового клиента с именем «Ольга» и паролем «1». Чтобы передать клиента системе, можно открыть окно авторизации в графическом интерфейсе, а можно послать запрос через REST API.

REST API умеет работать как с XML, так и с JSON. Вот как будет выглядеть один и тот же запрос для разных форматов (табл. 15.1).

Таблица 15.1. *Один и тот же запрос для форматов XML и JSON*

XML	JSON

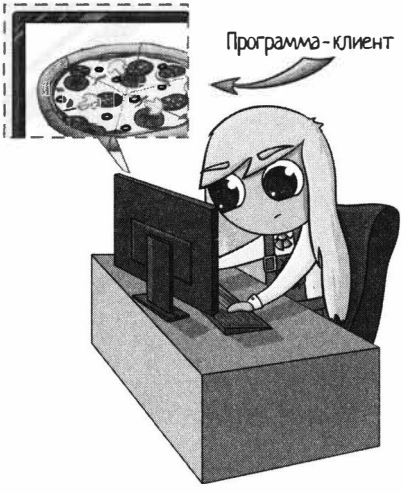
<client>
  <name>Ольга</name>
  <password>1</password>
</client>
	client: {
	  name: "Ольга";
	  password: "1"
	}

В XML каждый параметр оборачивается в теги, то есть в угловые скобки <>. И дублируется дважды: первый тег открывающий, второй закрывающий, чтобы система поняла, что «здесь имя закончилось».

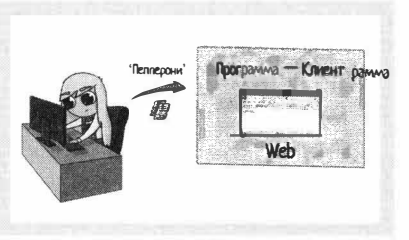
JSON выглядит намного проще, в нем как минимум нет дублирования названий параметров. Его проще читать, и он меньше «весит», что бывает важно с кучей данных и плохим Интернетом.

Что такое клиент-серверная архитектура?

Вернемся к примеру заказа пиццы в интернет-магазине. Когда вы нажимаете кнопки в браузере, вы работаете с клиентом в клиент-серверной архитектуре. Клиент — это та программа, с которой «общается» пользователь.



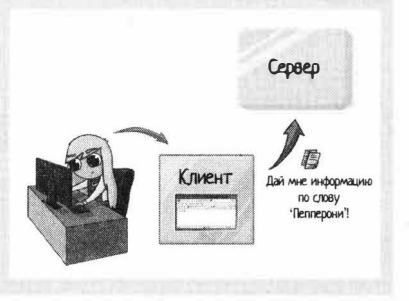
Допустим, девочка Катя ищет пиццу «Пепперони». Вводит в поисковую строку: Пепперони и видит результат — все варианты с таким названием. Но что происходит в потрохах приложения?



Катя вводит данные в клиент

Катя ввела данные на клиенте. Но когда она нажала «найти», клиент отправил запрос на сервер:

— Дай мне информацию по слову «Пепперони»!

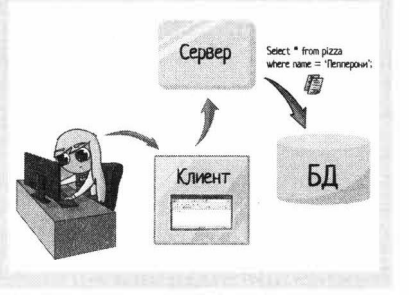


Клиент передаёт данные на сервер

Сервер отправил запрос в базу данных (БД):

```
Select * from pizza where name = "Пепперони"
```

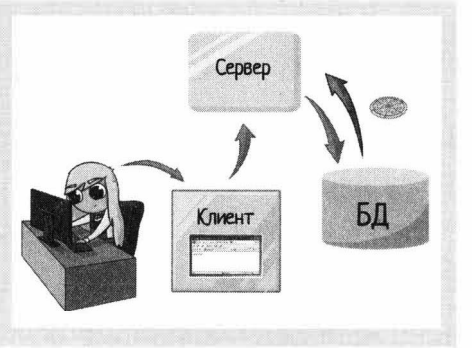
Что и означает: дай мне всю информацию по пиццам с названием «Пепперони».



Сервер запрашивает данные в БД

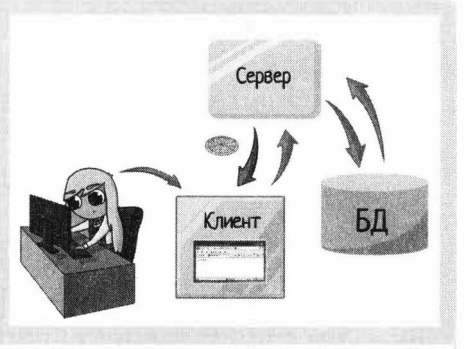
База ответила:

— Вот тебе все, что нашла.



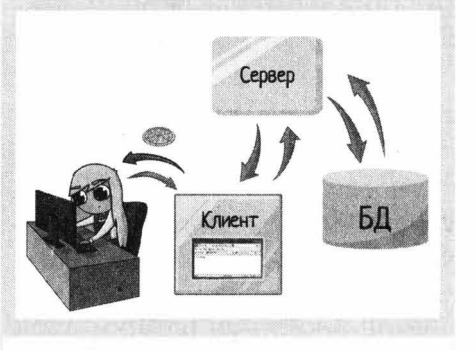
БД возвращает инфу серверу

Сервер вернул эту информацию клиенту:



Сервер передаёт её клиенту

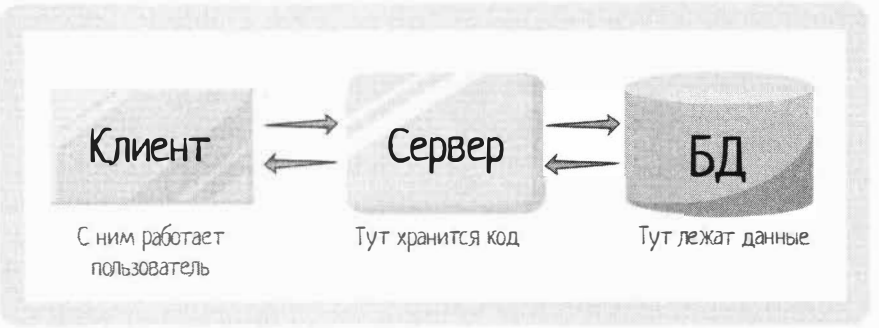
А клиент уже отрисовал ее для Кати:



Клиент показывает инфу Кате

Катя смотрит:
— Ага, возьму большую!

Катя даже не догадывается, какой путь проделали данные в программе, когда она вбила туда название пиццы. А мы разобрались:



- **Клиент** — та программа, с которой работает пользователь. Он знать не знает, что у него на компьютере программа целиком или где-то за ней прячутся сервер с базой, а то и целый RAID. Он работает в браузере или с desktop-приложением. И всё что ему нужно знать — это «куда тут тыкать».

Клиенту не нужно много памяти, места на диске и других ресурсов. Поэтому клиентские рабочие места относительно дешево стоят. А это именно то, что нам нужно.

- **Сервер** — компьютер, на котором хранится само приложение. Весь код, вся логика, все дополнительные материалы и справочники. Например, справочник адресов (ФИАС) или справочник юридических лиц (ЕГРЮЛ) — они тоже занимают место, как сами по себе, так и в памяти приложения.

Иногда говорят «сервер приложения» и «сервер БД». Это нормально, ведь фактически сервер — это просто машина, компьютер. А базу и сервер приложения обычно хранят на разных машинах — ради безопасности. В таком случае, если говорят «сервер приложения», — речь о втором звене нашей схемы.

Приложения бывают самые разные. Есть ресурсоемкие, им нужно много памяти и места на диске. Есть «легкие», которые можно развернуть даже на домашнем компьютере.

- **БД (база данных)** — хранилище данных. Тут вы можете легко поискать информацию и можете быть уверены в том, что она сохранится, даже если в приложении что-то сломается.

Сколько места нужно под базу, зависит от количества данных. Есть огромные базы в банках, где и одного терабайта будет мало. А есть совсем небольшие, которые вы можете установить на своей машине, например, XAMPP. И вряд ли вы напихаете туда столько данных, что у вас не останется под них места ;-)

Отдельной базы может не быть, тогда структура станет двузвенной: клиент-сервер. И всё!

Цикл создания приложения

- Где хранится код?
- Кто собирает его воедино?
- Кто «деплоит» приложение?
- Как это автоматизировать?

Что такое система контроля версий?

Система контроля версий (VCS, Version Control System) — это как бы Dropbox для кода.

Разработчик пишет код на языке программирования и сохраняет его. Но куда? Оставить на компьютере? А если жесткий диск умрет, то что? Мы весь код потеряем!

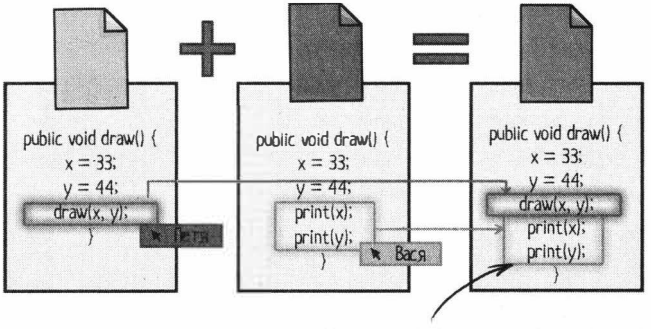
Чтобы не потерять работу, надо делать резервные копии. Или хранить код в облаке. Можно использовать «общее» решение типа Dropbox или Яндекс-диска. Но в нём есть свои минусы, главный из которых — невозможность совместной работы.

Что если разработчики Вася и Петя решили затронуть один класс с кодом? Вася успел первым — в итоге он редактирует спокойно, а Петины правки создают новую конфликтующую версию. И кому-то потом придется вручную проверять эти конфликты.

Другой вариант — сидеть и ждать, пока файлик освободится. Но над большими продуктами трудятся целые команды разработчиков. И совместное редактирование одного файла кода — дело обычное.

Система контроля версий создана для хранения кода, хотя в ней можно хранить и картинки, и документацию. Но главная её фишка — это как раз работа с изменениями. То есть возможность совместно редактировать один и тот же файл. А потом система посмотрит: были ли конфликты? Если разработчики меняли разные места, она автоматически объединит изменения.

Если затрагивалось одно место, решение будет принимать человек. И всё же это намного лучше конфликтов в дропбоксе. Система красиво подсвечивает, кто что менял, а дальше уже коллеги разбираются, как это лучше всего совместить.



Система контроля версий автоматически объединила изменения в коде

Что такое сборщик продукта?

Разработчик написал код — это просто набор текстовых файликов с расширением java, если он пишет на языке программирования Java, или каким-то другим, если он пишет код на другом языке.

Потом он этот код сохранил — для этого используется система контроля версий. Пока всё хорошо, но как файлики с кодом становятся красивой картинкой в браузере?

Для этого нужно как-то запустить код, написанный в файликах. В небольшом приложении достаточно запустить один класс, например, `Main.java` — и вот оно, ваше приложение! Но чем сложнее ПО, тем сложнее процесс сборки. Он может выглядеть примерно так:

1. Скомпилировать проект.
2. Объединить классы в файл JAR или другую библиотеку.
3. Установить набор зависимостей.
4. Запустить конкретный класс или несколько классов.

И только после всех этих манипуляций у нас появляется ПО. Однако стоит заметить, что все эти манипуляции нужны для языков типа Java. Если вы используете интерпретируемый язык (например, PHP), то сборщик вам не будет нужен.

Набор манипуляций зависит от конкретного проекта, но чем сложнее проект, тем больше действий надо сделать. Вручную выполнять все эти действия смысла нет — скука, да и только. К тому же человек легко может ошибиться, а в итоге получится неработающее приложение.

Поэтому эту работу автоматизируют. Можно написать скрипт сборки на коленке, но зачем, если уже есть стандартные сборщики? Для Java это Ant, Maven, Gradle... Для других языков программирования есть свои сборщики.

Но общий принцип у них один: разработчик настраивает все зависимости в коде, прописывая их в сборщике. И выкидывает из головы. А дальше уже запускает одну команду:

1. Собери мне проект.
2. Прогони все unit-тесты.
3. ...

А сборщик идет по прописанным инструкциям и выполняет команды.

Разумеется, сборщик — это тоже код. Он включает прописанные инструкции, которые меняются. Так что инструкцию по сборке нужно хранить вместе с кодом в системе контроля версий!

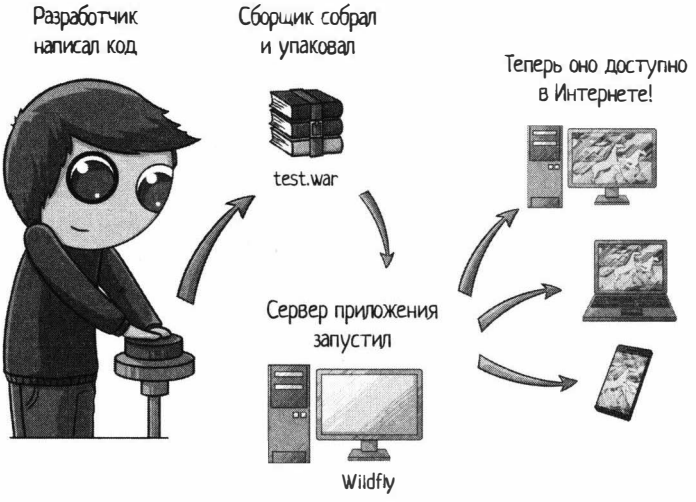
Что такое сервер приложения?

Сервер обеспечивает возможность обращаться с приложением по HTTP-протоколу. То есть разместить ваше приложение в Сети, чтобы оно стало доступно через браузер в Интернете.

Оглянемся назад:

1. Разработчик написал код — это просто набор текстовых файликов с расширением java или каким-то другим.
2. Потом он этот код сохранил — для этого используется система контроля версий.
3. Потом он этот код собрал — если это было нужно. Приложение уже работает, но на компьютере разработчика.

Чтобы приложение стало доступным в Сети, нужен сервер. Он берет на себя скучную инфраструктурную работу. А разработчик может сконцентрироваться на бизнес-логике, не отвлекаясь на детали обеспечения транспортного пути.

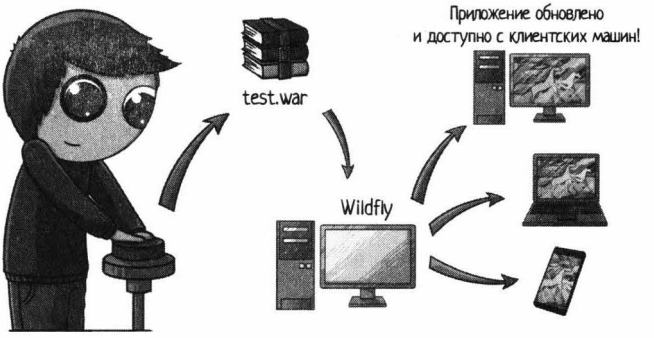


Что такое CI (Continuous Integration)?

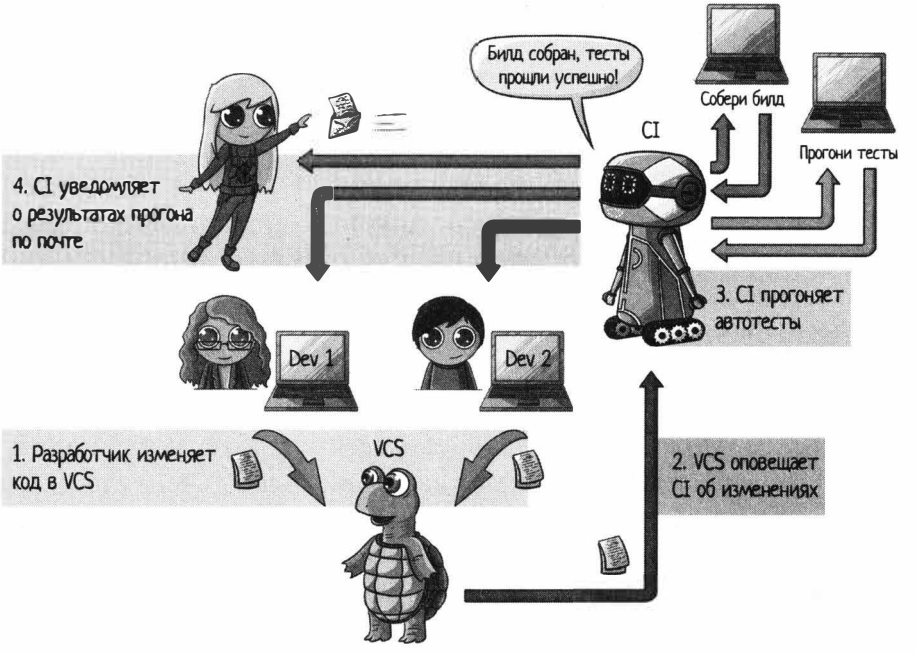
Как выглядит процесс запуска программы:

1. Разработчик написал код — это просто набор текстовых файликов.
2. Потом он этот код сохранил — для этого используется система контроля версий.
3. Потом он этот код собрал — если это было нужно. Приложение уже работает, но на компьютере разработчика.
4. Потом он его запустил — с помощью сервера приложения.

Это уже хорошо — одна кнопка вместо десяти телодвижений на каждый пункт. Но это пока еще полуавтоматизация — все равно нужен человек, который введет команду и соберет билд «ручками». Допустим, этим занимаются разработчики. Когда тестировщик просит новую сборку на тестирование, разработчик нажимает кнопку и ждет сборки. Затем нажимает другую, чтобы запустить приложение...



А вот если убрать из этой схемы человека — мы получим CI! CI — это сборка, деплой[176] и тестирование приложения без участия человека.



Приложение CI само забирает изменения из репозитория с кодом. Само! А ещё оно само автотесты прогонит и оповестит всех заинтересованных о результатах сборки и тестирования. И все это — автоматически, без вмешательства человека! То есть один раз настроили, а дальше оно само...

Что такое Docker?

Docker — это контейнер для приложения. В котором уже все настроено: и операционная система, и сервер приложения, и вся инфраструктура. Бери да используй!

Это как если бы вы решили купить велосипед. Сравните две ситуации:

1. Вам привезли все детали отдельно. Причем велосипед хитрый, так что деталей много, в том числе мелких.

А инструкция по сборке местами неполная, местами устаревшая — детали выглядят уже по-другому, поди угадай, о чем речь!

2. Вам привезли готовый велосипед. Садись и используй!

Его *уже* собрали. За вас. Это сделали опытные мастера, которые легко найдут нужную деталь среди исходников и соберут все в правильной последовательности. Ведь они знают, как это делается.

Именно в этом фишка докера. Вам не надо настраивать окружение, его уже настроили до вас — те люди, которые разбираются в этом и умеют это делать. Вам как пользователю — очень удобно! Тык-тык, и оно работает!

Докер решает проблему «пользователь забыл что-то настроить», уменьшает количество запросов в техподдержку и облегчает поставку продукта — когда мы сами всё настроили, то спим спокойно, не ошибутся теперь.

Командная строка

Что такое командная строка? Это окно в Windows, откуда вы можете отдавать системе различные команды.

Самый простой способ открыть это окно — нажать комбинацию клавиш <Win>+<R> и ввести в открывшемся поле команду `cmd`.

Что вы должны уметь тут делать? План-минимум:

- перемещаться по папкам;
- копировать файлики;
- запускать приложение;
- архивировать файлы.

Так что гуглим, как это делается, и тренируемся!

Что такое Linux?

Linux — это операционная система. Как винда (Windows), только более защищенная. В винде легко подхватить вирус, в линуксе это практически невозможно. А еще линукс бесплатный (часть дистрибутивов)! Правда, разобраться в нем немного посложнее...

Чтобы работать в Linux, вам надо уметь работать в консоли. Да, сейчас есть и графические интерфейсы (WinSCP очень сильно облегчает жизнь). Но без консоли никуда. А это вам не мышкой в иконочки тыкать!

Зато продвинутых пользователей система этим и подкупает. Ведь можно самому собрать ядро линукса! То есть полностью контролировать установку системы на компьютер, ух! А вот простого человека такая мысль только пугает...

Но в любом случае, в XXI веке с линукс-системами стоит уметь работать. Серверы приложений обычно делают именно на Linux-системах. Так что даже от тестировщиков просят «знания Linux», это можно увидеть в вакансиях. Но что входит в эти знания, что надо уметь?

Никто не ждет от вас (давайте я напомню, что вы читаете книгу для начинающих тестировщиков) глубоких знаний UNIX-систем — как они внутри устроены, как их администрировать и прочее.

Вам нужно просто уметь выполнить простые операции:

- посмотреть логи через командную строку;
- забрать их к себе;
- создать архив;
- скопировать файлик;
- переместить файлик;
- запустить службу;
- ...

Где тренироваться?

Можно «поднять» виртуальную машину. Правда, тут сначала придется разбираться, как «поднимать» виртуалку ;-) Можно установить докер, там немного попроще будет.

А можно купить облачную машину. Когда мне надо было поиграться с линуксом, я пошла на SimpleCloud (он мне в гугле одним из первых выпал, и у него дружелюбный интерфейс. Но можно выбрать любой аналог) и купила самую дешевую машину — за 150 руб в месяц. Месяца вам за глаза хватит, чтобы «пощупать-потыркать», и этой машины с минимумом памяти тоже.

Чтобы подключиться к машине, используйте инструменты:

- Putty — командная строка;
- WinSCP — графический интерфейс.

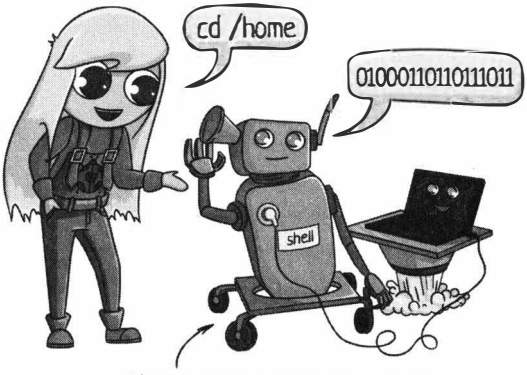
Что такое bash / shell?

И то, и другое — интерпретаторы командной строки в линуксе. То есть если вы откроете командную строку и введете любую команду, да хоть:

```
cd /home
```

то именно интерпретатор ее расшифрует и скажет компьютеру: «Он хочет перейти в каталог /home». Компьютер ведь не понимает команды на русском / английском языках. Ему нужны байтики. Этим и занимается интерпретатор — переводом с «нашего» на «компьютерный» язык.

Так что `cd /home` — это shell-команда! Или bash. Смотря какой интерпретатор установлен в вашей системе. В каждой операционной системе установлен интерпретатор по умолчанию. У них есть какие-то различия, но есть и набор базовых команд, которые понимают все: `cd`, `mv`, `cp`, `ls`... (в винде эти команды немного другие).



Интерпретатор командной строки

Что такое regexp?

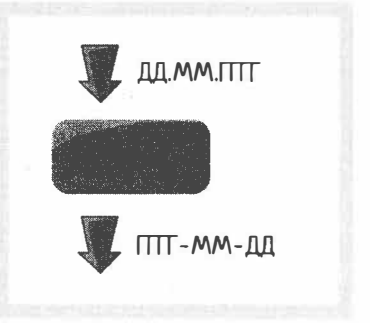
Регулярные выражения (от англ. regular expressions) — это механизм для поиска и замены текста. В строке, файле, нескольких файлах... Их используют разработчики в коде приложения, тестировщики в автотестах, да и все прочие просто при работе в командной строке!

Чем это лучше простого поиска? Тем, что позволяет задать шаблон.

Например, на вход приходит дата рождения в формате ДД.ММ.ГГГГ. Вам надо передать ее дальше, но уже в формате ГГГГ-ММ-ДД. Как это сделать с помощью простого поиска? Вы же не знаете заранее, какая именно дата будет.

А регулярное выражение позволяет задать шаблон «найди мне цифры в таком-то формате».

Замена — на входе одно, на выходе другое.



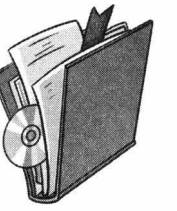
Регулярные выражения — очень полезная вещь для тестировщика. Применений у них много, даже если вы не автоматизатор и не спешите им стать:

- найти все нужные файлы в папке;
- grep-нуть логи — отсечь все лишнее и найти только ту информацию, которая вам сейчас интересна;
- проверить по базе, нет ли явно некорректных записей: не остались ли тестовые данные в продакшене? Не присылает ли смежная система какую-то фигню вместо нормальных данных?
- проверить данные чужой системы, если она выгружает их в файл;
- выверить файлик текстов для сайта — нет ли там дублирования слов?
- подправить текст для статьи;
- ...

Если вы знаете, что в коде вашей программы есть регулярное выражение, вы можете его протестировать. Вы также можете использовать регулярки внутри ваших автотестов. Хотя тут стоит быть осторожным.

Не забывайте о шутке: «У разработчика была одна проблема, и он стал решать ее с помощью регулярных выражений. Теперь у него две проблемы». Бывает и так, безусловно. Как и с любым другим кодом.

Портфолио



Что можно почерпнуть из этой главы и взять себе для портфолио?

- Работа с API.

Возьмите любой бесплатный проект и напишите автотесты на его API-методы. В портфолио вставьте ссылку на коллекцию в Postman (если тесты на REST-методы) или на файл для SoapUI (SOAP-методы).

Файл с коллекцией тестов можно и в системе контроля версий сохранить. Это тоже отличный вариант — так вы покажете, что и с VCS научились работать. Создайте бесплатный репозиторий и сохраните туда свои автотесты.

Да, про написание автотестов для API-методов придется отдельно гуглить, в этой книге я эту тему не затрагивала ;-) Но вы можете посмотреть видео на моем YouTube-канале[177] — я выкладываю туда отрывки лекций из курса по автоматизации в Postman.

Бесплатные проекты можно найти на Testbase в разделе «Test IT»[178]. Например, система Users, тут вам и SOAP, и REST.

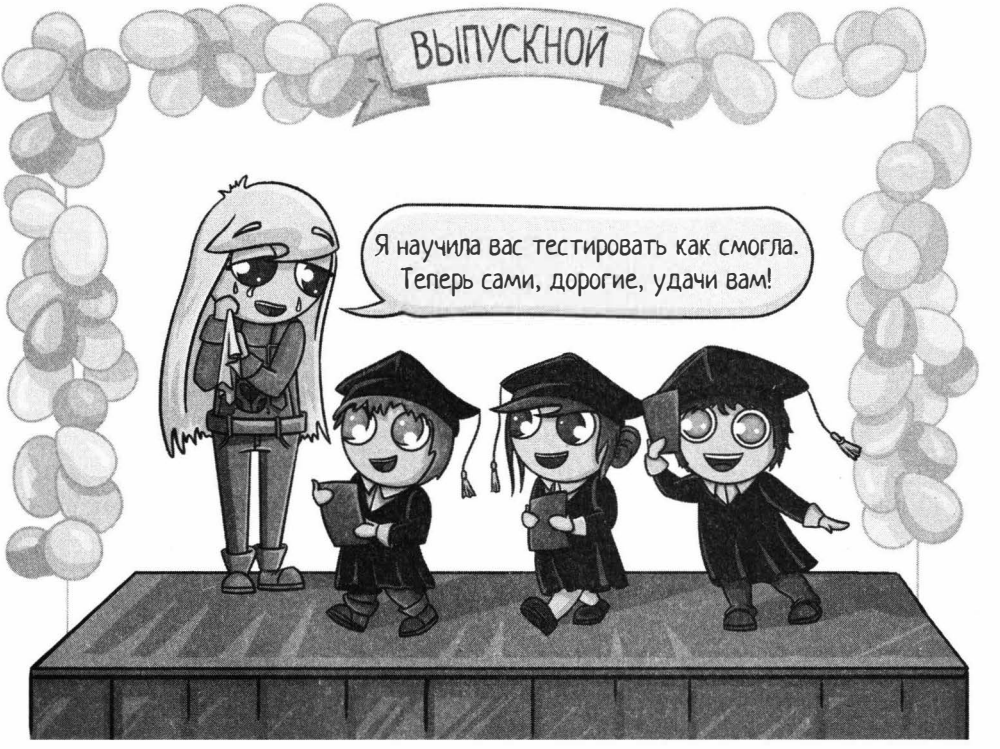
○ SQL.

Самый часто спрашиваемый на собеседовании навык! Но по нему портфолио делать не надо. Просто потренируйтесь и пропишите в резюме самое сложное, что умеете делать. А потом не провалитесь на собеседовании, решая задачки ;-)

○ Linux.

Аналогично SQL — требуют часто, но в портфолио писать примеры не надо. Просто потренируйтесь выполнять обычные действия, которые могут вам пригодиться в дальнейшем: создать файл, каталог, скопировать, переместить и так далее.

ЗАКЛЮЧЕНИЕ



Ну вот, я и рассказала вам всё, что хотела. И даже немножечко больше ;-) Последнюю, 15-ю, главу я добавила спонтанно, уже когда почти закончила книгу. А в итоге еще полгода работы. Но оно того стоило!

Давайте под конец подведем небольшой итог. С чего все-таки начинать тестирование? Вот вы пришли на работу и получили задачу: «Проверить такую-то функцию». Что делать?

Давайте сначала рассмотрим ситуацию «требования есть».

Когда есть требования

Изучить требования

Иногда это прям подробное ТЗ с вариантами использования и описанием всех возможных путей развития событий. А иногда это кратенькая фраза:

> Сделать умный поиск как в <Популярный интернет-магазин>, который будет исправлять опечатки ввода.

И тот, и другой варианты — нормальные.

Протестировать их!

Не забывайте, что чем раньше мы найдем ошибку, тем проще и дешевле ее будет исправить. Лучше всего найти проблему еще на стадии ТЗ! Поэтому требования надо не просто читать, их надо читать вдумчиво и сразу прикидывать план тестирования. Или писать чек-листы для будущих проверок. Это поможет вам найти нестыковки, которые вы не заметили бы, если бы просто читали.

Здорово, когда у задачи есть бизнес-обоснование. Если мы понимаем исходную проблему пользователя, которую должны решить, нам будет проще проверять ТЗ. А точно ли мы решаем эту проблему? Нет ли способа проще и лучше?

Если мы создаем новый функционал — то для кого? С какой целью? Зная цель, можно понять, насколько хорошо она достигается.

Не забывайте проверять требования, а не просто изучать их. Держите в голове список проверок:

- полнота;
- однозначность;
- непротиворечивость;
- необходимость;
- осуществимость;
- тестируемость.

Если вы тестируете не создание простенького сайта типа интернет-магазина, функционал будет сложный. И даже если он хорошо описан, иногда сложно упорядочить у себя в голове «ху из ху». В таком случае используйте любые инструменты для визуализации требований:

- State & Transition Diagramm (схема состояний и переходов);
- Mind map;
- любую другую схему.

Также может помочь, если просто переписать ТЗ немного по-другому. В таком случае вспоминаем инструменты:

- вариант использования;
- Decision Table (таблица решений).

Написать чек-лист проверок

Ещё раз: если есть требования, чек-лист лучше попробовать написать ДО того, как мы увидим реализацию. В идеале, конечно, аналитик пишет требования и передает команде. Пока разработчик пишет код, тестировщик пишет тесты.

Да, когда вы начнете «щупать» руками систему, чек-лист придется корректировать. Где-то вы ожидали другой ОР, но и этот нормальный. А где-то забыли о проверках. Когда начинаешь тестировать, всегда приходят в голову новые идеи.

И вот здесь уже, на этом этапе, вступают в силу все основные техники тестировщика:

- оформление тест-кейсов и чек-листов;
- тест-дизайн и тест-анализ;
- классы эквивалентности;
- граничные значения.

Почему я включила в название раздела именно «чек-лист»? Потому что чаще выбирают именно его. Но оформление — это далеко не самая важная его часть. Самое главное — это наполнение ваших проверок!

Вы еще долго будете учиться этому — видеть важные проверки. Видеть неочевидную границу (если речь не просто о поле ввода чисел), полностью проверять хитрое взаимодействие двух систем...

Классы эквивалентности — это не то, что учится за 2–3–4 часа. Это придет с опытом, но уже сейчас вы можете начать применять эти техники хотя бы на базовом уровне. Что и выделит вас среди других джуниоров.

Протестировать систему

Теперь, когда у вас есть набор проверок, остается только применить его и протестировать систему. Разумеется, не стоит идти строго по исходному чек-листу. Как только вы начнете «щупать» реальную систему, у вас обязательно появятся новые идеи и мысли по поводу тестов — так проводите их! И пополняйте свой чек-лист. Это нормально.

При тестировании не забываем проверять документацию. Да, ТЗ мы уже проверили, но помним про:

- сообщения об ошибках;
- предупреждения «что вводить»;
- примеры;
- письма от системы;
- поп-ап сообщения.

Оформить результат

Закончили тестирование — оформляем результат. Что сюда входит:

- баг-репорты, если при тестировании найдены баги;
- отчет «Что я успел проверить» (но он нужен не всегда).

Баг-репорты

С баг-репортами всё понятно, мы подробно их рассматривали в *главе 5*. Если вы уже работаете, то заносите баги в баг-трекинговую систему по всем правилам. Не ленитесь, ограничиваясь лишь заголовком. Добавьте шаги и скриншот.

Поверьте моему опыту, раз исправят «тут же, при тебе», два, три... Но рано или поздно такую «сейчас поправим» задачу отложат в долгий ящик. А потом никто уже не сможет вспомнить, о чем там была речь. Поэтому оформлять надо так, чтобы всегда можно было вспомнить, что там было, и легко воспроизвести. При этом помня про правило «кратко, но емко!» Я совсем не призываю вас к куче лишнего текста ;-)

Если вы делаете тестовое задание, то оформлять каждый баг обычно не надо. Если вам дают систему с кучей багов, то обычно просят 1–2 оформить «нормально», а остальные записать списком. Фактически просто название оставить.

В этом случае такое оформление будет нормальным. Оно поможет показать, сколько багов вы нашли, и при этом не тратить кучу времени на оформление. Вы ведь уже показали навык оформления на одном-двух примерах.

Отчет

Иногда от тестировщика просят отчет о проделанной работе. Сразу уточните, в каком виде от вас ожидают этот отчет.

Иногда отчет генерится автоматически. Это если вы работаете с системой, в которой уже есть готовые тест-кейсы или чек-листы. Тогда вы просто их выполняете и помечаете в системе галочки: «Этот тест прошел успешно, а этот упал». И всё, отчет готов!

Когда у нас не было такой системы и мы писали чек-листы в экселе, я просто раскрашивала ячейки тестов разными цветами:

- зеленый — тест прошел успешно;
- красный — тест упал + даю ссылку на баг;
- белый — не успела проверить.

Иногда отчет пишется в свободной форме из серии «Что я успел проверить». Это может быть как подробное изложение, так и перечисление проверенного функционала со ссылками на чек-листы проверок.

Иногда в компании есть вполне конкретная форма отчета. Тогда придерживайтесь её.

Если у вас просят отчет о тестировании во время тестового задания, то напишите его в свободной форме. Если функционала было мало, то можно предоставить чек-лист проверок. Он покажет, что вы вообще тестировали. И именно сами проверки имеют значение.

Если вам дали кучу функционала, в отчете достаточно лишь обозначить его:

- Проверил:
  - регистрацию;
  - авторизацию через емейл / телефон / соцсети;
  - создание пользователя;
  - редактирование пользователя;
  - удаление пользователя;
  - создание заявки;
  - ...
- Во время тестирования нашел баги:
  - раз
  - два
  - три
  - ...
- Пример оформления багов (пара штук описана подробно).

Хотя лично я против объемных тестовых заданий. Чтобы реально проверить ваши навыки, достаточно дать условную формочку с одним полем. Этого хватит! А вот если просят сделать задание на пару дней, то ну их! Найдите компанию без таких дурацких требований. Хотя если хочется ради опыта сделать что-то такое, то вперед ;-)

В любом случае, если от вас хотят видеть отчет, для начала поинтересуйтесь, в каком виде. А то, может, вы что-то сделаете, а получите совершенно не то, что заказчик планировал. Лучше уточнить.

Когда требований нет

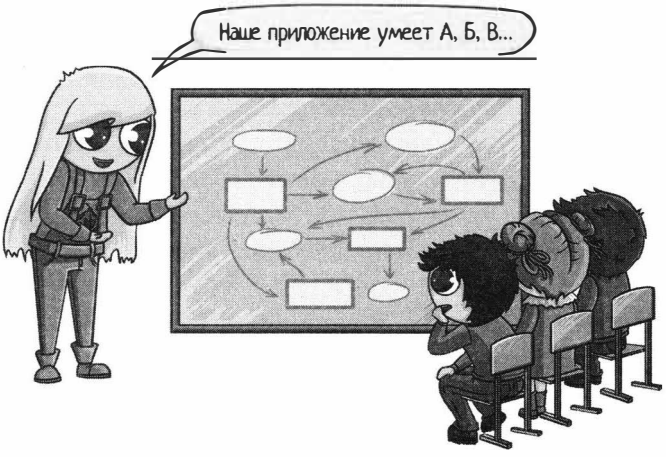
На самом деле в этом случае мало что меняется. Ведь обычно требования все же есть, просто неявные. Тогда расспрашиваем аналитика, разработчика или другого носителя знаний и фиксируем результат.

Если же вы для портфолио взяли реальный сайт, у вас не будет требований. Только общедоступная информация, но это не требования, а, скорее, FAQ. Что делать в таком случае?

Изучить систему

Почитайте, что она умеет делать. Попробуйте использовать её возможности. Пощупайте ее!

А потом можете зарисовать mind map, чтобы ничего не забыть.



Написать чек-лист проверок

Самое важное при тестировании — не тыкать систему наобум, а действовать системно. Поэтому логичнее сначала подумать о том, что вы хотите проверить. И потом уже идти проверять.

Можно, конечно, думать «в голове», но лучше записать. Так больше идей в голову придет, и при проверке вы ничего не забудете!

И снова этот пункт — самый важный и самый сложный. Да, накидать хоть какой-нибудь чек-лист обычно не проблема. 2–5–10 пунктов... Но вот чтобы сделать его качественно, надо применять:

- классы эквивалентности;
- граничные значения;
- логику: «А что будет, если сделать так?» в нужных местах (особенно — на стыках двух систем или двух частей одной системы).

Этому вы еще долго будете учиться, так что не стоит воспринимать этот пункт как самый простой. Придумать качественные проверки — это сложная задача. Ее можно и нужно изучать!

Протестировать

Ну всё, план действий (чек-лист) есть, теперь мы готовы тестировать!

Важно, чтобы вы понимали — это третий пункт, а не первый. Не надо бросаться на систему с шашкой наголо: «Вижу поле ввода? Фигачу туда все возможные строковые значения!».

А потом такие тестировщики очень удивляются функционалу системы — а что, так тоже можно было? А что, после авторизации что-то меняется?!!

Сначала познакомьтесь со своей системой, изучите ее. Что она вообще умеет делать? Это касается любой системы, которую вы тестируете: будь то тестовое задание, работа или курсы повышения квалификации.

Оформить результат

Тут всё то же самое, что и в *разд. «Требования есть»*. По результатам тестирования оформляем:

- баг-репорты, если при тестировании найдены баги;
- отчет «Что я успел проверить» (но он нужен не всегда).

А что потом?

А потом вы прокачиваете свои навыки в том числе на собственных ошибках.

Пропустили баг? Обязательно проведите его ретроспективу (мы говорили об этом в *главе 5*). Так вы на будущее запомните, куда нужно тщательнее копать. А иногда сможете найти связанную проблему, которую не видели раньше.

Еще можно в принципе изучить баг-трекер для вашей системы — какие баги в ней встречаются? Учитесь у коллег!

Ну а точечно навыки качайте по видео, статьям, курсам в Интернете. Но это вы уже на работе сможете сказать, что именно вам сейчас для работы нужно. Это и учите!

Несколько слов в завершение

Ну вот, пожалуй, и всё. Я рассказала всё, что хотела.

Конечно, сейчас от джуниоров требуют уметь «и то, и то, и вот это еще». О том, что вас могут спросить на собеседовании, мы говорили в *главе 13*. Да, по верхам, но тут уж извините. На каждую из предложенных тем (Devtools, Postman, API, SQL) можно написать отдельную книгу. И они уже даже написаны!

Но нельзя выучить сразу всё. Поэтому начните с основ, которые абсолютно точно пригодятся любому тестировщику. Мы рассматривали их в этой книге:

- составление чек-листов;
- оформление тест-кейсов;
- заведение баг-репортов.

Это — основа основ. И я надеюсь, что моя книга поможем вам с нею.

Помните, что это — книга-тренинг. Не ленитесь выполнять задания, приведенные в конце глав. Особенно из раздела «портфолио»! Ведь оно выгодно подчеркнет вас среди других резюме новичков. А еще придаст уверенности в своих силах.

Если вы читали книгу без остановок, то сделайте задания сейчас:

1. Откройте конец *главы 1.*
2. Выполните задания оттуда.
3. Откройте конец *главы 2.*
4. Выполните задания оттуда.
5. ...

Это поможет увидеть,что «не всё так просто, как казалось». Прочитали главу, прочитали задания, они показались легкими. А теперь? Когда вы успели забыть то, что читали ранее? Вот и будет повод освежить знания.

Конечно, еще более крутой вариант — это сделать задание, и чтобы тебе ментор сказал: «Вот тут неверно и тут, переделывай». Так будет больше пользы. Если у вас есть друзья-тестировщики, обратитесь к ним за помощью.

Или приходите ко мне на курс **http://testbase.ru/learn/beginner**. Да, теорию вы теперь всю знаете из моей книги. Но ведь самое важное в ней — это практика!

Где еще можно найти практику, смотрите в моей статье «Где начинающим тестировщикам получать опыт?»[179]. Я желаю вам удачи в новом деле. Надеюсь, у нас теперь станет больше хороших, «качественных» джуниоров! ;-)

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

А

Абстракция 108
  в чек-листах 131
Автоматизатор 373
Автоматизация 384, 412, 511, 540
Автоматизация рутины 374, 409, 541
Автоматизированное тестирование 373, 380
Автотесты 331
Альтернативные сценарии 266
Альфа-тестирование 360
Анализ 230
Аналитик 24, 25, 46, 294, 514, 530
Антипаттерны обоснования бага 218
Артефакты тестирования 304
Архитектура кода 392
Аттач 220
  в ожидаемом результате тест-кейса 109

Б

Баг 26, 53, 163, 191, 571
Багред 209
Баг-репорт 304, 363, 443, 571
Баг-трекер 304, 576
Баг-трекинг 188, 221
БД, база данных 552
Белый ящик 331, 374
Бета-тестирование 361
Бизнес-логика 240
Бизнес-смысл 165
Блок-схема 259
Брутфорсинг 353
Бюрократия 422, 428, 430

В

Валидация 155
Вариант использования 149, 258, 314, 569
Веб-приложение 359
Версия 222
Виды документации 290, 301
Визуализация 320
Вопросы 58, 293
Вопросы, которые стоит задать 41
Входные данные 254

Г

Главный недостаток тест-кейсов 104
Главный функционал 43
Границы 166
Граничные значения 149, 158, 172, 442, 570
График 311
Графический интерфейс 400
Графический пользовательский интерфейс GUI 548

Д

Данные 245
Десктопное приложение 359
Диаграмма 311
  состояний и переходов 149, 259
Динамическое тестирование 376
Документ 309
Документация 36, 51, 108, 228, 258, 261, 263, 314

Е

Единообразие 219

З

Заглушка 267
Заголовок бага 193
Задача тестировщика 36, 68
Заказчик 25, 42
Закрытый вопрос 42

И

Идеи для тестов 316
Идеи тестов 83
Инструкция 285, 360
Инструменты
  автоматизации тестов 415
  баг-трекинга 189
  для награзучного тестирования 521
  для оформления бага 209
  для поиска границ 163
  для снятия ограничений в вебе 163, 271
  для pairwise 181
  для рисования mind map 57
  исследования продукта 55
  кроссбраузерности 225
  по автозаполнению полей 164
  полуавтоматизации 413
  снятия ограничений в вебе 226
Интеграционные тесты 402
Интеллектуальная карта 56
Интерфейс 177, 251, 263, 296, 331, 387, 548
Исполнитель 222
Исследовательские туры 243, 376
Исследовательское тестирование 240, 376, 380, 515
ИТ 455, 501

К

Какие вопросы не задавать 45
Карта сценариев 320
Киллер фича 244
Классификация 326, 327
Классы эквивалентности 149, 152, 156, 172, 178, 330, 442, 515, 570
Клиент 226, 244, 551
Клиент-серверная архитектура 226, 551
Код 50
Конкретика
  в ожидаемом результате тест-кейса 110
  в чек-листах 133, 138
Контекст 48
Конфигурации 362
Конфлюенс 141
Коридорный тест 350
Кратко, но емко! 93, 105, 126, 196, 296, 311, 458
Кроссбраузерность 200, 224

Л

Лайфхак 301
Ложноположительные срабатывания теста 390
Локализация 193, 202, 213, 311
Локализация багов 394, 399, 404

М

Майнд карта 304
Майнд карты 242
Мануальное тестирование 371
Менеджер 22, 24, 28
Ментальная ловушка 205
Метод 5 почему 230
Метод бисекционного деления 207
Минимальные данные для воспроизведения бага 207
Минимальные шаги для воспроизведения 214
Мнемоника 162, 242
Мобильное приложение 359
Мобильные приложения 278, 289, 340, 344, 358
Мозговой штурм 176

Н

Набор тестов 122, 305
Нагрузочное 329
  тестирование 340, 342, 380, 521
Название бага 208
Направления развития в тестировании 508
Негативное тестирование 43, 68, 71, 179, 335, 380
Нефункциональное тестирование 335, 514
Ноль как класс эквивалентности 161
Нумерация
  в тест-кейсах 115
НФТ 372

О

Обоснование бага 197, 218, 339
Образец 264
Обучение 286
Ограничение 271
Ожидаемый результат 67, 217
  в баге 211
Онлайн-тренажеры 439
Определение бага 186, 209
Ортогональная матрица 149, 172
Основной сценарий 68
Основные сценарии использования 58
Открытый вопрос 42
Отчет 201, 571
Отчет о тестировании 304
Оформление багов 364
Ошибка 29, 191
Ошибки локализации 357
Ошибки названия бага 194

П

Параллельная работа 225
Перехват трафика 354
Пирамида автоматизации 393
Письма от системы 267, 359
Плагины 164
Плейсхолдер 267
ПО 20, 21, 24, 26, 29, 247, 585
Пограничные значения 159
Позитивное тестирование 68, 69, 333, 380
Поиск 297
Полный перебор 153, 353
Полуавтоматизация 410, 412
Пользователь 61, 71
Пользовательский интерфейс 119, 351, 390
Пользовательский сценарий 258
Попарное тестирование 149, 172
Портфолио 455
Правило 20 минут 55, 205
Правило минимальных чернил 97
Презентация 287, 534
Приёмочное тестирование 367
Примеры
  в чек-листах 128
  оформления багов 222
Принцип лопаты 204
Приоритет 222
  бага 199
  в задаче 194
  задачи 207
  проверки в чек-листе 177
  тестирования 76
  тестов 329
  функций приложения 59
Приоритизация 481
Программа 20, 28, 37, 549
Продукт 36
Проект для практики 331
Производительность 338, 340
Профессиональный рост 55
Процесс 420

Р

Разработчик 25
Ревью
  автотестов 395
Регистронезависимость 275
Регресс 264, 299, 387
Регрессионное тестирование 248, 369
Регулярные выражения 564
Результат
  в баге 217
  в чек-листе 133
  проверки 76
Резюме 436
Ретроспектива 540
Рефакторинг 299
Рисунок 320
Ручное тестирование 371, 385, 510

С

Санитарное тестирование 366
Сборщик 557
Сборщик продукта 509
Сервер 226, 552, 558
Сервер приложения 509, 559

Сервисы для подготовки презентаций 535
Серебряная пуля 242
Сертификат 174
Серый ящик 332
Система контроля версий 509, 555, 557, 559
Скриншот 195, 216, 220, 224, 259, 263
Скриптовое тестирование 380
СМС 358
Собеседование 174, 328, 492
Сообщение об ошибке 134, 159
Сопроводительное письмо 437, 463
Ссылка 211
Статическое тестирование 331, 375
Стресс-тестирование 342, 380
Схема 260, 569
  состояний и переходов 318, 515, 569
Сценарий 298
  использования 198
Сценарное тестирование 241

Т

Таблица решений 149, 315, 569
Тест 40, 66, 152, 157, 175
Тест-анализ 172, 179, 570
Тест-дизайн 148, 172, 178, 330, 384, 398, 538, 570
Тест-дизайнер 372, 511, 515
Тестирование
  безопасности 352, 380, 523
  графического интерфейса 380
  документации 258, 276
  защищенности 514
  локализации 356, 380
  надежности 344, 380
  по готовым тестам 376
  попарных значений 181
  производительности 342, 380, 514
  совместимости 359, 380
  стабильности 344
  требований 258
  удобства использования 346, 380, 514
Тестировщик 24, 25, 26, 27
Тест-кейс 58, 77, 141, 143, 191, 240, 315
Тест-кейсы 304, 363, 371, 376, 387, 423, 440, 455
Тест-менеджер 511, 527
Тестовая документация 304
Тестовое задание 306, 474, 499
Тестовые данные 212, 228
Тестовый стенд 98
Тест-план 122, 304, 329, 479
Техники тест-дизайна 150
Технологическая граница 163, 165
ТЗ 36, 42, 49, 159, 160, 165, 258, 260, 263, 292, 312, 320, 330, 331, 351, 422, 536, 568
ТЗ (техническое задание) 30
Типовые ошибки
  при оформлении бага 224
Типы границ 162
Требования 29, 46, 258, 567
  в вакансиях 384

У

Улучшение 191
Умение гуглить 45, 53, 129
Уточнение информации 45

Ф

Фидбек на тестовое 481
Фреймворк 299, 400
Фриланс биржи 446
Функционал 115, 155, 177, 292, 296, 299, 337, 346, 368, 379, 387, 538
Функциональное тестирование 335, 442
Функция 58, 152, 244, 252, 260, 396

Х

Хороший код 177

Ц

Цель пользователя 58

Ч

Чек-лист 58, 124, 141, 143, 228, 230, 241, 304, 371, 376, 423, 440, 568
  закрытия задачи 228

Черный ящик 329, 330, 374
Чит-лист 144

Ш

Шаблон
  бага 210
  для резюме 462
  компании 314
  улучшения 219
Шаги
  бага 211
  воспроизведения бага 196
  для воспроизведения бага 216

Э

Эвристика 150, 241
Эталон 106
Эффект
  лентяя 206
  пестицида 156, 376

A

Acceptance тестирование 367
Ad-hoc тестирование 240
API 266, 398, 401, 416, 509, 516, 521, 537, 549
API-тесты 395, 405

B

Bash 564
Buddy testing 241

C

CI 509, 560
Cmd 562
Concurrency 225
CRUD 225

D

Decision Table 317, 515, 569
Docker 561

G

GUI 351, 541, 548

H

HeadHunter (HH) 436
HR 422, 455, 477, 490
HTML 523
HTTP-протокол 558

I

IE 224
Integer 159
ISTQB 174, 490
IT 40

J

JIRA 54
JSON 509, 550

K

KPI 202

L

Legacycode 247
LinkedIn 436
Linux 516, 562

M

Maxlenght 226
Mind map 57, 569
Monkey testing 241

N

Notepad 54

P

Pair testing 241
Pairwise 180, 412
PM 25, 514
Pop-up сообщения 277
Postman 399
PROD 118, 199, 242
PRODUCTION, PROD 98
Putty 563

R

RecordAndPlay 388, 403
ReleaseNotes 310
REST 509, 516, 550
RESTAPI 398

S

Sanitytesting 366, 368
Shell 564
Smoke-тесты 365
SOAP 51, 509, 550
SOAPAPI 398
SQADays 201
SQL 487, 509, 516, 523
SQL-инъекции 355, 523
State & Transition 314
State-Transition 515
State&TransitionDiagramm 569

T

TestSuite 122, 305

U

UI 119
UI-тесты 400, 405
Unit-тесты 393, 405
Usability 336
Usability тестирование 346, 442, 524

V

VCS 555

W

WinSCP 562

X

XML 509, 550
XSS-атаки 355, 523

ПРИМЕЧАНИЯ

[1] Говард Шульц. Как чашка за чашкой строилась Starbucks: **http://okiseleva.blogspot.ru/2017/02/starbucks.html**. Все ссылки, используемые в книге, можно посмотреть на **http://testbase.ru/book-beginner/links**.

[2] См. **https://tilda.cc/page/?pageid=211634&projectid=63957**.

[3] Тильда — это такой интернет-ресурс, помогающий создавать веб-страницы.

[4] См. **http://testbase.ru/**.

[5] См. **http://okiseleva.blogspot.ru/2014/03/google.htmlhttp://okiseleva.blogspot.ru/2014/03/google.html**.

[6] См. **http://okiseleva.blogspot.ru/2013/01/dot-com.html**.

[7] Фикс — исправление бага, жаргон.

[8] Софт — ПО, программное обеспечение, тоже жаргон, привыкайте ;-).

[9] Взял Samsung Galaxy Note 7 в самолет — заплати штраф: **https://3dnews.ru/941053**. Samsung до сих пор ищет причину возгорания Galaxy Note 7: **https://3dnews.ru/940888**. Видеоролики: **http://tv.mk.ru/video/2016/10/19/vse-vzryvy-galaxy-note-7-shokiruyushhie-kadry.html** или **https://www.youtube.com/watch?v=RLiXGjFQz68**.

[10] Desktop — переводится как «поверхность стола». Это то, что установлено у вас на компьютере и запускается с него, а не через браузер: Microsoft Word, Skype, Paint, World of Warcraft...

[11] См. **http://www.amazon.com/Practitioners-Guide-Software-Test-Design/dp/158053791X**.

[12] См. *разд. «Чем занимается тестировщик?» главы «Введение в тестирование ПО»*.

[13] См. **http://testbase.ru/**.

[14] Бэкап (от англ. backup) — резервная копия программы.

[15] IT — ИТ, информационные технологии, наша отрасль.

[16] JIRA — коммерческая система отслеживания ошибок (баг-трекер), мы ещё поговорим о таких системах позднее, в *главе 5*, посвященной баг-трекингу.

[17] Confluence — корпоративная вики-система для внутреннего использования организациями с целью создания единой базы знаний.

[18] Postman — инструмент для работы с REST API, позволяющий отправлять запросы вручную или писать автотесты.

[19] Rocket-science (Рокет саенс) — своеобразный иронический фразеологизм, означающий сложную умственную работу, требующую ощутимых интеллектуальных способностей.

[20] В 2021 году.

[21] См. **https://testbase.atlassian.net/wiki/spaces/STUDENTS/pages/436109314/**.

[22] См. *разд. «Чем занимается тестировщик?» главы «Введение в тестирование ПО»*.

[23] Релизимся (сленг) — выпускаемся.

[24] Классы эквивалентности — области, внутри которых значения эквивалентны друг другу. Тестирование одного значения приводит к тому же результату, что и тестирование другого (см. *главу 3*).

[25] Integer — тип данных «целое число», от −2 147 483 648 до 2 147 483 647.

[26] Краш (сленг) — серьезное «падение» приложения, из серии знаменитого синего экрана смерти в винде.

[27] См. раздел про позитивное и негативное тестирование.

[28] См. **https://dadata.ru/**.

[29] См. **https://dadata.userecho.com/forums/4-baza-znanij/topics/2901-iyun-2017-umnaya-transliteratsiya-fio/**.

[30] См. **https://dadata.ru/**.

[31] Подробнее про дымовое тестирование (smoketest) рассказано в *главе 9. Классификация тестирования*.

[32] См. **https://www.say7.info/cook/recipe/401-SHarlotka.html**.

[33] См. **https://dadata.ru/**.

[34] Это как двойная авторизация в почте или при оплате: сначала ты заходишь по паролю в аккаунт, а потом тебе приходит СМС и нужно ввести данные оттуда. На сайтах вместо СМС делают всплывающее окошко с дополнительным паролем. Оно не имеет никакого отношения к функционалу самого сайта и нужно, чтобы в принципе открыть ссылку. Пример: **http://buggy.bugred.ru/**. Логин и пароль здесь надо ввести в систему не для авторизации ВНУТРИ Багги, а для того, чтобы вообще попасть на сайт.

[35] Testlink: **http://okiseleva.blogspot.com/2018/09/testlink.html**; Confluence: **http://okiseleva.blogspot.com/2017/07/jira-confluence.html**. Ссылки можно найти на странице: **http://testbase.ru/test-it**.

[36] Это из примера чек-листа без результата. Ищите полную версию в блог-посте «Какой результат писать в чек-листе»: **https://okiseleva.blogspot.ru/2015/03/blog-post_33.html**.

[37] API — программный интерфейс, API-метод — метод, по которому две программы общаются между собой. Это вы видите в интерфейсе красивые окошечки, а сами программы общаются по непонятным простому обывателю символам (JSON, SOAP).

[38] ASAP — As Soon As Possible, как можно скорее.

[39] См. **https://dadata.ru/**.

[40] Пример такого чек-листа: **http://akkaparallel.blogspot.ru/2013/06/email.html.**

[41] См. **https://dadata.ru/.**

[42] Брутфорсом называется метод взлома учетных записей путем подбора паролей к ним (от англ. brute force, грубая сила).

[43] См. **https://docs.google.com/spreadsheets/d/1veMQd8JtncxZe_zJ5CiFrg0M6uPNWDZexxJXUPdQCNg/edit#gid=0**.

[44] См. **https://sitechco.ru**.

[45] См. **https://okiseleva.blogspot.com/2019/08/tms-test-it.html.**

[46] См. **https://ru.wikipedia.org/wiki/TeamCity** — инструмент, позволяющий прогонять автотесты на ваших тестовых серверах и хранить информацию о каждом запуске.

[47] См. **http://testbase.ru/411** — раздел «Теория в картинках», статья «Тест-кейс VS чек-лист».

[48] Ориентировочная дата публикации — лето 2022 года.

[49] Погуглите: `чит-листы`.

[50] Эвристика — совокупность исследовательских методов, способствующих открытию ранее неизвестного.

[51] Lee Copeland. A Practitioner's Guide to Software Test Design.: **http://okiseleva.blogspot.ru/2012/03/lee-copeland-practitioners-guide-to.html**.

[52] Подробнее см в посте «Классы эквивалентности: будни Золушки»: **https://okiseleva.blogspot.com/2015/07/blog-post_87.html**.

[53] См. **https://habr.com/ru/post/524784/**.

[54] См. статью в Википедии «Целое (тип данных)».

[55] Подробнее см. в статье «Класс эквивалентности „Ноль-не ноль“»: **https://okiseleva.blogspot.com/2016/12/blog-post_15.html#more**.

[56] Мнемоника — техника для лучшего запоминания. В нашем случае мы используем первые буквы границ и получаем короткую аббревиатуру.

[57] См. **https://habr.com/ru/post/418233/.**

[58] См. **http://www.satisfice.com/tools.shtml**.

[59] См. **http://www.unit-conversion.info/texttools/random-string-generator/**.

[60] Что тестировщику надо знать про панель разработчика: **https://okiseleva.blogspot.com/2016/07/blog-post_52.html**.

[61] Подробнее см в посте: **https://habr.com/ru/post/541202/**.

[62] См. **https://okiseleva.blogspot.com/2016/12/blog-post_15.html**.

[63] См. **https://habr.com/post/418233/**.

[64] См. **https://software-testing.ru/library/testing/functional-testing/1238-number-string-subdomains**.

[65] Бесплатный open-source проект, доступен по ссылке: **https://testbase.atlassian.net/wiki/spaces/FOLKS/overview**.

[66] ISTQB, International Software Testing Qualifications Board (Совет по сертификации тестирования программного обеспечения, который работает на международном уровне). Сертификат, который не ценится в РФ, но ценится в аутсорсе, если вас продают в англоязычные компании.

[67] Ситечко — инструмент для оформления чек-листов, бесплатный, — см. **https://sitechco.ru/2011/08/dobro-pozhalovat-v-sitechko-prostoj-i-u/**.

[68] Ссылки на все инструменты, которые тут упомянуты, вы можете найти в статье «Инструменты для Pairwise» моего блога.

[69] Алистер Коберн, автор книг по разработке требований к ПО.

[70] См. **http://www.satisfice.com/glossary.shtml#Bug** + **http://www.satisfice.com/blog/archives/572**.

[71] Разумеется, это ИМХО (мое мнение). Все люди разные: для меня удобно одно, для вас — другое.

[72] См. http://okiseleva.blogspot.com/2018/08/workflow.html.-

[73] См. **https://megaplan.ru/letters/haters**.

[74] Краш — это когда оно вылетает.

[75] Жаргон, фиксить = исправлять.

[76] См. **https://sqadays.com/en/index** — международная конференция по тестированию. Самая популярная в нашей сфере на русском языке (на 2018 год).

[77] См. **https://habr.com/ru/post/468087/**.

[78] См. **http://bugred.ru/**.

[79] См. **https://okiseleva.blogspot.com/2015/03/blog-post_18.html**.

[80] См. **https://okiseleva.blogspot.com/2016/06/69.html**.

[81] ЕГРЮЛ — Единый государственный реестр юридических лиц.

[82] См. **https://habr.com/ru/post/434522/**.

[83] См. ранее *разд. «Шаблон бага»*.

[84] См. **https://okiseleva.blogspot.com/search/label/%D0%BF%D0%BE%D0%B2%D1%81%D1%8E%D0%B4%D1%83%20%D0%B1%D0%B0%D0%B3%D0%B8**.

[85] См. **https://okiseleva.blogspot.com/2015/07/blog-post_16.html**.

[86] Фейл — «облом» по-нашему.

[87] Exception — необработанное исключение. То есть не красивое сообщение об ошибке, а целые куски кода.

[88] См. **http://users.bugred.ru**.

[89] См. статью «Что тестировщику надо знать про панель разработчика»: **https://okiseleva.blogspot.com/2016/07/blog-post_52.html**.

[90] ASCII — American Standard Code for Information Interchange, Американский стандарт кодирования для передачи информации.

[91] См. **http://software-testing.ru/oldtrainings/about/648**.

[92] См. **https://okiseleva.blogspot.com/2017/10/blog-post.html.**

[93] См. **https://okiseleva.blogspot.com/search/label/%D0%BF%D0%BE%D0%B2%D1%81%D1%8E%D0%B4%D1%83%20%D0%B1%D0%B0%D0%B3%D0%B8**.

[94] См. **https://okiseleva.blogspot.com/search/label/%D0%BF%D0%B0%D0%BD%D0%B1%D0%B0%D0%B3%D0%BE%D0%BD**.

[95] См. **http://okiseleva.blogspot.com/2015/07/blog-post_16.html**.

[96] Взято с Хабра: **https://habr.com/company/redmadrobot/blog/280618/**.

[97] Взято из статьи: **http://training.qatestlab.com/front-page/blog/technical-articles/what-is-ad-hoc-testing/**.

[98] См. Сэм Канер, Джек Фолк, Енг Кек Нгуен, «Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений».

[99] См. *разд. «Типы границ» главы 3.*

[100] См. **http://okiseleva.blogspot.com/2015/02/exploratory-software-testing-james.html**.

[101] См. **http://testbase.ru/bugs**.

[102] Killer-feature — не переводится, жаргонное. По смыслу: главная фича релиза, главный функционал.

[103] См. **http://dump-conf.ru/**.

[104] См. **http://okiseleva.blogspot.com/2015/07/pretty-roses.html**.

[105] Начать можно с **https://okiseleva.blogspot.com/2014/02/blog-post_6.html**, раздел «Usability-тестирование».

[106] О нем мы говорили в *главе 5.*

[107] См. *разд. «Исследовательские туры Уиттакера» главы 6.*

[108] Панбагон. Картинка морского ежа не влезает в отведенное ей место: **http://okiseleva.blogspot.com/2016/03/blog-post_30.html**.

[109] Панбагон. Картинка акции Валентинова дня не влезает в экран iPad mini: **http://okiseleva.blogspot.com/2017/02/mini-ipad.html**.

[110] См. **https://maven.apache.org/install.html**.

[111] См. **https://okiseleva.blogspot.com/2012/07/wink.html**.

[112] См. **https://habr.com/ru/post/346290/**.

[113] Lucene — свободная библиотека для высокопроизводительного полнотекстового поиска.

[114] См. **http://okiseleva.blogspot.com/2015/07/blog-post_16.html**.

[115] См. **http://software-testing.ru/library/around-testing/requirements/2494-purpose-of-test-documentation**.

[116] См. **http://software-testing.ru/library/testing/test-analysis/2405-the-one-page-test-plan**.

[117] См. **https://testbase.atlassian.net/wiki/spaces/STUDENTS/pages/1151278**.

[118] См. **https://testbase.atlassian.net/wiki/spaces/STUDENTS/pages/1192133223/11**.

[119] Как писать Release Notes, чтобы их читали (ВИДЕО): **https://okiseleva.blogspot.com/2017/07/release-notes.html**.

[120] Максим Ильяхов ввел понятие инфостиля. Это то самое «кратко, но емко!» + понятно и без бюрократии. Можно погуглить его блог и книги и почитать подробнее.

[121] См. **https://okiseleva.blogspot.com/2015/11/blog-post_86.html**.

[122] См. **https://habr.com/ru/post/546432/**.

[123] См. **https://testbase.atlassian.net/wiki/spaces/STUDENTS/pages/635732096/State+Transition**.

[124] См. **https://habr.com/ru/post/548192/**.

[125] Блогпост «Пример карты сценариев для визуализации ТЗ»: **https://okiseleva.blogspot.com/2019/03/blog-post_7.html**.

[126] См. **https://testbase.atlassian.net/wiki/spaces/STUDENTS/pages/1326121639**.

[127] См. **https://habr.com/ru/post/550498/**.

[128] Бесплатное Java-приложение с тестами на уровне API: **https://testbase.atlassian.net/wiki/spaces/FOLKS/overviewhttps://testbase.atlassian.net/wiki/spaces/FOLKS/overview**.

[129] Термин используется при поиске работы. Гигиенический фактор не влияет на позитивную мотивацию, когда он есть, но влияет на негативную, когда его нету. Например, наличие туалета в офисе мы не будем считать плюсом (он же везде есть). Но если его нету, это уже минус компании.

[130] То есть месяц без перезагрузки.

[131] Эту главу я писала как раз в 2019 году.

[132] См. **http://testbase.ru/**.

[133] См. *главу 6* про исследовательские туры.

[134] См. **https://okiseleva.blogspot.com/2014/02/blog-post_6.html**.

[135] Скриншот взят из статьи «TEST IT! Тестируем регистрацию на Foodnation.ru»: **http://okiseleva.blogspot.com/2013/04/test-it-foodnationru_25.html**.

[136] Напомню, что эту главу я писала как раз в апреле 2019 года.

[137] Об этом можно послушать в видео «TCP/IP: что это и зачем это тестировщику»: **https://www.youtube.com/watch?v=rLUzYeLdM0k**.

[138] Подробнее см в статье «TEST IT. Тестируем сайт Foodnation.ru»: **http://okiseleva.blogspot.com/2013/04/test-it-foodnationru.html**.

[139] См. статью «Панбагон. Broken Sword 1 вылетает при попытке осмотреть записку в кабинете под Консьержери»: **http://okiseleva.blogspot.com/2016/10/broken-sword-1.html** и решение проблемы в статье «Панбагон. Краш при смене языка, ошибка локализации»: **http://okiseleva.blogspot.com/2016/11/blog-post_3.html**.

[140] Выдержка из **статьи https://okiseleva.blogspot.com/2018/11/blog-post.html**.

[141] Try (англ.) — пробовать. Геймерский сленг: «траить, потраить» — попробовать одолеть нового босса.

[142] Feedback (англ.) — обратная реакция на какое-то действие или продукт.

[143] Loot — лутать, забирать. Победил босса — получил призы. Лут — это те самые призы.

[144] Фича (жаргон) — новый функционал.

[145] Видео есть на моем YouTube-канале «Самописный робот на Watin»: **https://www.youtube.com/watch?v=N6xslAETH70&t=1s**. Но предупреждаю, ватин уже морально устарел, не знаю, поддерживается ли он сейчас!

[146] См. про них подробнее в *главе 10* про автоматизацию.

[147] Зеленый — прошел успешно, красный — какая-то проверка сломалась.

[148] Gherkin — человеко-читаемый язык для описания поведения системы, который использует отступы чтобы задать структуру документа.

[149] Пример взят из статьи с Хабра «Руководство: Cucumber + Java»: **https://habr.com/ru/post/332754/**.

[150] Пирамида Маслоу — это описание потребностей человека: от низменных желаний до возвышенных. Американский психолог Абрахам Маслоу сформулировал эту теорию в 1954 году в работе «Мотивация и личность».

[151] Mock (мок) — это код, имитирующий тот сервис, который вы будете использовать в конечном продукте. Но он легче и проще (в том числе в управлении), чем реальный сервис, который вы использовали бы в производстве.

[152] Users — бесплатное приложение для тестирования: **https://okiseleva.blogspot.com/2017/04/users-soap-rest.html**.

[153] Картинка сделана по аналогии с баянчиком: **https://developerslife.ru/14635**.

[154] Бесплатная система, чтобы «пощупать» API-тесты: **https://testbase.atlassian.net/wiki/spaces/FOLKS/overview**.

[155] Users — напомню, это бесплатное приложение для тестирования: **https://okiseleva.blogspot.com/2017/04/users-soap-rest.html**.

[156] См. **https://testbase.atlassian.net/wiki/spaces/USERS/pages/592511089**.

[157] Selenium — это набор инструментов для автоматизации тестирования.

[158] Это система Continuous Integration (CI). Используется для автоматического запуска тестов.

[159] См. **https://okiseleva.blogspot.com/2013/03/watin.html**.

[160] Митинги — в гибких методологиях так называются ежедневные совещания.

[161] Глава готовилась в 2020 году.

[162] Аутсорсинг — от англ. Outsourcing: Outer Source Using.

[163] HH, HeadHunter — самый популярный на момент подготовки главы (2019 год) сайт для поиска работы.

[164] Эйчар (жаргон) — от англ. Human resources, человеческие ресурсы. Так называют себя специалисты в области управления персоналом.

[165] См. **http://www.sql-ex.ru/** — бесплатный тренажер по SQL.

[166] См. **https://playground.learnqa.ru/puzzle/triangle** — вот такого плана сделайте. Вроде легко, но с особенностями.

[167] См. **http://testbase.ru/test-it**.

[168] На момент подготовки главы — 2019 год.

[169] См. **http://testbase.ru/?post_type=skill&p=51**.

[170] Скилл — сленговое заимствование из английского языка (от слова «skill»). Переводится дословно как «навык», «умение что-либо делать».

[171] Конференция по тестированию: **https://sqadays.com/ru/index**.

[172] Разумеется, в моей школе **http://testbase.ru/learn/beginner** практики дофига, но в предыдущей главе мы обсуждали, как новичку можно получить опыт бесплатно.

[173] Познакомиться с треугольником можно тут: **https://playground.learnqa.ru/puzzle/triangle**.

[174] См. пост «Список книг (по тестированию и не только) с отзывами»: **https://okiseleva.blogspot.com/2014/02/blog-post_6.html**.

[175] См. **http://testbase.ru/books/it-terms**.

[176] Деплой (от англ. deploy, развертывание) — используется в программировании в своем прямом значении, то есть для обозначения развертывания (переноса) программного обеспечения на сервер или устройство, где оно должно функционировать.

[177] См. **https://www.youtube.com/c/okiseleva**.

[178] См. **http://testbase.ru/test-it**.

[179] См. **https://okiseleva.blogspot.com/2014/08/blog-post_12.html**.

Вы держите в руках книгу — тренинг по тестированию программ с домашними заданиями. Выполните их все, и тестовые задания от работодателя вас точно не испугают! В результате вы получите портфолио, которое можно смело показывать при трудоустройстве даже в самую крутую компанию. Книга дает все необходимые базовые знания для специалиста по тестированию ПО и указывает направления для дальнейшего развития.

Ольга Назина — основатель Багреда (справочного ресурса для тестировщиков) и портала Testbase, на котором собраны полезные материалы по тестированию. Автор статей на Хабре и бесплатных уроков на Youtube. 13 лет проработала в тестировании, 6 из которых активно учила новичков. Провела более 110 курсов для начинающих тестировщиков, материал которых послужил основой для этой книги.

За годы работы Ольга протестировала десятки продуктов и обучила тысячи людей делать это правильно. Этот колоссальный опыт она собрала в одной книге, которая в нескучной манере рассказывает о том, как стать отличным тестировщиком. Книга не только развлекает и образовывает читателя, но также даёт все необходимые для специалиста по тестированию знания. От самых азов и до сложных концепций, которые пригодятся на руководящих должностях, — вы найдете в этой книге все, что поможет вам выделиться на фоне других тестировщиков. Книга обязательна к прочтению для русскоязычного QA.

Арсений Батыров, руководитель отдела ручного тестирования в компании InDriver

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6835-7