Министерство науки и высшего образования Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

Ю. В. Морозова

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Томск
2019

УДК 004.415.53(075.8)
ББК 32.973.26-018.2я73
М 801

Рецензенты:

В. В. Герасименко, канд. техн. наук, заместитель начальника отдела информационных технологий ООО «КДВ Групп»;

П. В. Сенченко, канд. техн. наук, доцент кафедры автоматизации обработки информации Томского государственного университета систем управления и радиоэлектроники

Морозова Ю. В.

М 801 Тестирование программного обеспечения : учебное пособие / Ю. В. Морозова. – Томск : Эль Контент, 2019. – 120 с.

ISBN 978-5-4332-0279-5

В настоящее время программное обеспечение (ПО) используется во всех сферах человеческой деятельности, и сегодня тестирование – это обязательная часть процесса разработки программного продукта. Цель профессии «тестировщик» – помочь создать качественный продукт. Работа тестировщика – предотвратить дефекты и, следовательно, обеспечить высокое качество процесса разработки и его результатов. Для этого необходимо обладать: знаниями о видах тестирования; об инструментах и библиотеках для автоматизации тестирования, умением пользоваться специальным ПО для автоматизированного тестирования и регистрации ошибок, навыками тест-дизайна. Данное пособие поможет получить эти знания.

Для студентов направлений «Программная инженерия» и «Бизнес-информатика», а также новичков в области тестирования программного обеспечения и программирования.

ISBN 978-5-4332-0279-5

© Морозова Ю. В., 2019
© Оформление.
ООО «Эль Контент», 2019

Оглавление

Введение ... **4**
1 Что такое тестирование и зачем нам все это надо ... **6**
2 Дефекты и их жизненный цикл ... **23**
2.1 Классификация дефектов ... 24
2.2 Жизненный цикл дефекта ... 29
3 Место тестирования в жизненном цикле разработки ПО ... **38**
3.1 Модели жизненного цикла разработки ПО ... 39
3.2 Этапы тестирования ... 44
3.3 Методы проектирования тестов ... 47
3.4 Тестовая документация ... 61
3.4.1 Тест-кейсы ... 62
3.4.2 Чек-листы ... 68
4 Классификация видов тестирования ... **73**
4.1 По знанию системы ... 74
4.2 По позитивности ... 77
4.3 По целям (или объекту) ... 78
4.4 По исполнителям (субъектам) ... 91
4.5 По времени проведения ... 92
4.6 По степени автоматизации ... 94
4.7 По состоянию системы (по исполнению кода) ... 96
5 Особенности тестирования мобильных и веб-приложений ... **99**
5.1 Особенности тестирования веб-приложений ... 101
5.2 Особенности тестирования мобильных приложений ... 106
Заключение ... **111**
Литература ... **112**
Глоссарий ... **114**

Введение

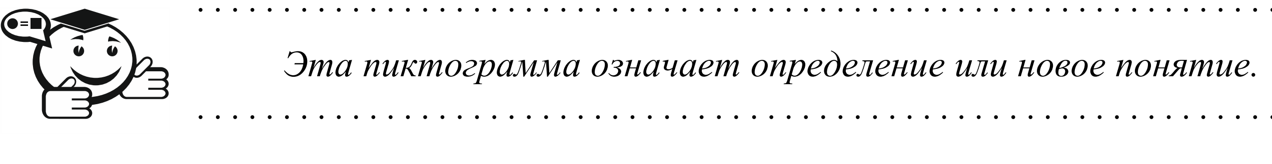
Компьютерные технологии захватили нашу повседневную жизнь. Программное обеспечение управляет работой множества окружающих нас вещей – от мобильных телефонов и компьютеров до швейных машин и чайников. Многие программисты ненавидят тестировать свой код. Для них это не важно, они считают свой код идеальным и не видят цели тестирования, им кажется, что это огромная трата времени и сил. Тестирование необходимо, так как все мы совершаем ошибки. Некоторые из них могут быть незначительными, в то время как другие – иметь самые разрушительные последствия. В любом случае, все мы сталкивались с теми или иными ошибками в программах: текстовый редактор, намертво зависший при работе с важным документом; банкомат, «съевший» карточку, или просто сайт, который никак не загрузится – все это встречается в нашей жизни и раздражает нас.

Данное учебное пособие призвано раскрыть основные понятия тестирования программного обеспечения, стратегии и техники. Оно будет полезно начинающим тестировщикам, а также программистам, которые желают писать чистый код.

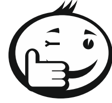
В пособии рассмотрены этапы, уровни и виды тестирования ПО, техники тестирования черного и белого ящиков, особенности тестирования веб- и мобильных приложений. Приведен ряд полезных инструментов и подходов для автоматизации тестирования.

Соглашения, принятые в учебном пособии

Для улучшения восприятия материала в данном учебном пособии используются пиктограммы и специальное выделение важной информации.



Эта пиктограмма означает определение или новое понятие.



Эта пиктограмма означает совет. В данном блоке указаны более простые или иные способы выполнения определенной задачи.

Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Выводы

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по главе

1 Что такое тестирование и зачем нам все это надо

> Результатом работы тестировщика является счастье конечного пользователя.
>
> *Роман Савин*

Давайте начнем с понятия программного обеспечения. Что же такое программный продукт? Это все, что сейчас нас окружает. Это онлайн-сервисы, интернет-магазины, социальные сети, игры, мобильные приложения, чайники и мультиварки с дистанционным управлением посредством мобильных приложений и т. д. и т. п. Они окружают нас каждый день, но иногда с ними случается беда:

- В 1990 г. абоненты американского оператора сотовой связи AT&T не могли пользоваться дальней связью в течение 9 часов. Причиной стало обновление программного обеспечения. Убытки компании составили 60 млн долл.
- Баг американской системы противовоздушной обороны Patriot, выпущенной в 1991 г., унес жизни 28 солдат. Точность расчетов системы падала при использовании более 14 часов. В реальных условиях она была задействована около 100 часов.
- В 1974 г. один программист написал систему расчета платежей, где года записывались в виде двух последних цифр, что влекло за собой проблемы с вычислениями в 2000-х гг. В 1995 г. компания, которой принадлежала программа, потеряла несколько сотен миллиардов долларов в процессе замены ПО на компьютерах по всему миру.
- В 1994 г. компании Disney пришлось иметь дело с большим числом разгневанных покупателей. Ее первая компьютерная игра, The Lion King Animated Storybook, запускалась только на небольшом числе компьютеров, таких, которые использовали разработчики, когда писали код. История попала в новости многих газет и телеканалов.
- В 1994 г. рейс № 140 компании China Airlines закончился гибелью 271 человека. Причиной стал баг в программном обеспечении, из-за которого самолет потерял управление и упал при посадке.

- В 1983 г. чуть не случилась третья мировая война. Почти в результате бага: сбой в системе раннего реагирования СССР привел к ложному сообщению о запуске пяти баллистических ракет с территории США. К счастью, дежурный офицер Станислав Петров почувствовал неладное – если бы США атаковали, решил он, они бы запустили больше ракет. Тревога была внесена в отчеты как ложная. Причина: ошибка в ПО не позволяла отличить запуск ракеты от отблесков солнца на вершине облаков.
- В 2005 г. компания Toyota объявила об отзыве 160 000 гибридных автомобилей Prius из-за необъяснимого включения света и остановки бензиновых двигателей. Однако, в отличие от множества отзывов последних лет, проблема Prius'а заключалась вовсе не в железе, это была программная ошибка в программном обеспечении автомобиля. Prius имел *программный баг* [1].
- В 2012 г. случился крах Knight Capital Group из-за бага: потеряв 440 млн долл. за 30 минут, компания была вынуждена согласиться на слияние. Один из крупнейших игроков трейдерского рынка в США потерял 75% своих акций в одночасье. Причина: ошибка в торговом алгоритме заставила ПО проводить ошибочные сделки, покупая дорого и продавая дешево – как можно догадаться, это совсем не то, что обычно называют успешной торговой стратегией [2].
- 2 августа 2016 г. Samsung представила миру свой новый фаблет Galaxy Note 7 на мероприятии в Нью-Йорке. Менее чем через две недели после релиза в сети стали появляться первые сообщения о взрывах этого устройства (рис. 1.1). Менее чем через два месяца после выхода флагмана компании производство и продажи Galaxy Note 7 были окончательно прекращены. Samsung зафиксировал всего 35 случаев, когда оставленный на зарядке Galaxy Note 7 нагревался до такого состояния, что это приводило к взрыву аккумулятора. По официальным данным, их подвел производитель аккумулятора. На тестирование прислал один, а в общей поставке – другой. В любом случае, это был провал.

Рис. 1.1 – Бюллетень, предупреждающий о запрете на использование Galaxy Note 7 в автобусах в штате Флорида, США

- В 2000 г. мы успешно преодолели Баг тысячелетия (Millennium Bug), или Проблему 2000 года, самый известный баг в этом списке и многие из нас слышали о нем в то время. Не все компьютеры одинаково работают с датами, и после 19 января 2038 г. у нас опять могут появиться проблемы.

Какие можно сделать выводы?

Итак, тестирование не нужно, если продукт:

- делается для себя;
- не нацелен на продажи;
- не предусматривает особых финансовых вложений;
- не используется другими людьми или не создает им существенных проблем.

Тестирование необходимо, если:

- продукт делается с целью получения прибыли;

- он призван решать важные задачи;
- плохое качество выпущенного продукта нанесет удар по репутации компании и понизит ее авторитет среди пользователей.

Конечно, исправить можно многое: дефекты дизайна, неудобно расположенные элементы, скорость работы, уязвимости в системе безопасности, но вот вернуть доверие пользователей и репутацию IT-компании крайне сложно. Раньше компаний на IT -рынке было мало, и пользователи программных продуктов были продвинутыми и заменяли тестировщиков. Если в программе обнаруживались баги или какие-то дефекты, то пользователь звонил или отправлял письмо в компанию, где ошибку исправляли, и по почте отправляли дискетку со свежим релизом. Но начиная с 1990 г., согласно статистике, продажи персональных компьютеров с каждым годом удваивались. И появилась армия пользователей, которая не готова была что-то тестировать. Если программа не выполняет поставленных задач или наносит ущерб, пользователи уходят к конкурентам. Если что-то не устраивало или не работало, проще обменять на другой софт, т. к. число компаний, производящих ПО, тоже увеличивалось с каждым годом. И у пользователей появился выбор, что покупать и чем пользоваться. И сегодня тестирование – это обязательная часть процесса разработки программного обеспечения.

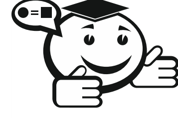
Таким образом, тестирование ушло вглубь компаний и появилась профессия *тестировщик*. Несмотря на то что профессия очень актуальна и есть множество вакансий, в вузах такой специальности нет. Поэтому чаще тестировщиками становятся выпускники специальностей, так или иначе связанных с программированием.

В большинстве случаев объявление с вакансией на должность тестировщика выглядит так: «Требуется QA (QA Tester, тестировщик, тестер, QA-инженер)». И возникает вопрос: «Кто такой QA Tester?». Тот, который тестирует того, кто делает QA?

Для начала давайте рассмотрим, что такое *Quality Control* и чем он отличается от *Quality Assurance* (*QA*).

Контроль качества (***quality control***) – *рабочие методы и активности, нацеленные на выполнение требований к качеству, являющиеся частью управления качеством* [3].

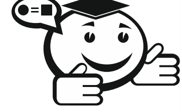
Задачей контроля качества является поддержка качества продукта в текущий момент времени.



Обеспечение качества (*quality assurance – QA*) *– это совокупность мероприятий, охватывающих все этапы разработки ПО, включая эксплуатацию и релиз, которые предпринимаются на разных стадиях жизненного цикла ПО, главной целью которого является обеспечение качества выпускаемого продукта* [3].

Таким образом, обеспечение качества (Quality Assurance) включает в себя контроль качества (Quality Control) наряду с другими процессами по улучшению качества работы компании.

Давайте разберемся: что же такое качество?



Качество программного обеспечения (*software quality*) *– это способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям* (*ISO 9126*).

Обобщая определение, приведенное в ISO-стандарте, можно сказать, что качество – это отсутствие дефектов.

Давайте рассмотрим, в чем разница между тестированием и обеспечением качества. Часто используют фразу «обеспечение качества» (или просто QA) для обозначения тестирования, тем не менее, обеспечение качества и тестирование не одно и тоже, но эти понятия связаны. Их объединяет более крупное понятие – управление качеством. Это понятие появилось в процессе эволюции представления о тестировании.

Самый первый компьютерный баг в истории был обнаружен 9 сентября 1947 г. Согласно фольклору, это произошло после случая в Гарвардском университете. После того как на реле прадедушки персонального компьютера Mark II Aiken Relay Calculator присел отдохнуть мотылек, один из контактов слегка коротнуло и весь 35-тонный агрегат со скрежетом остановился (рис. 1.2).

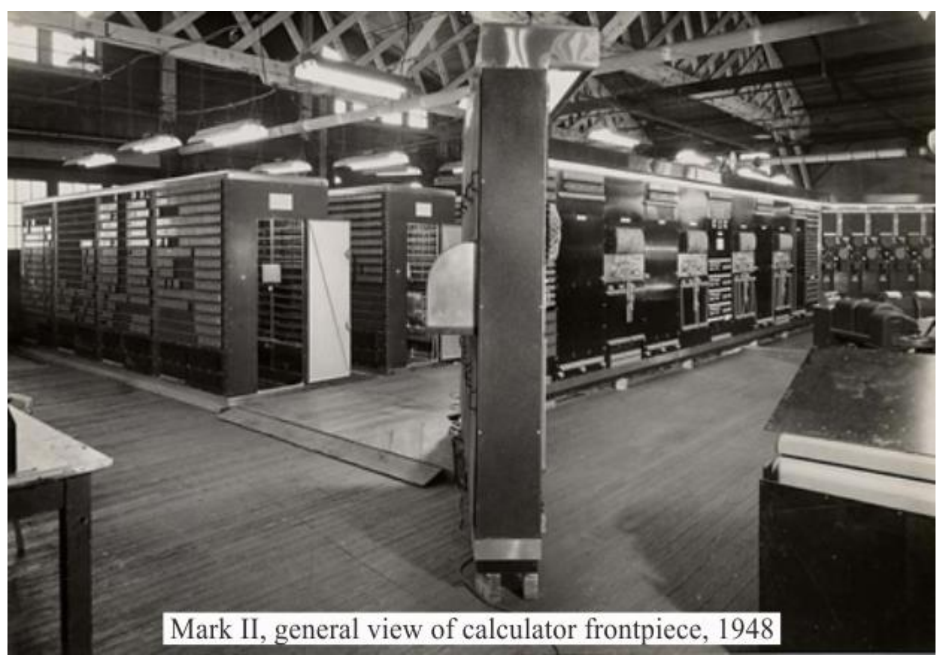


Рис. 1.2 – Mark II Aiken Relay Calculator

Инженеры извлекли мотылька, после чего аккуратно зафиксировали его скотчем в журнале испытаний с комментарием «Первый фактический случай найденного жука» (First actual case of bug being found) (рис. 1.3). Именно поэтому компьютерные дефекты принято называть *багами* (*bugs* – жуки).

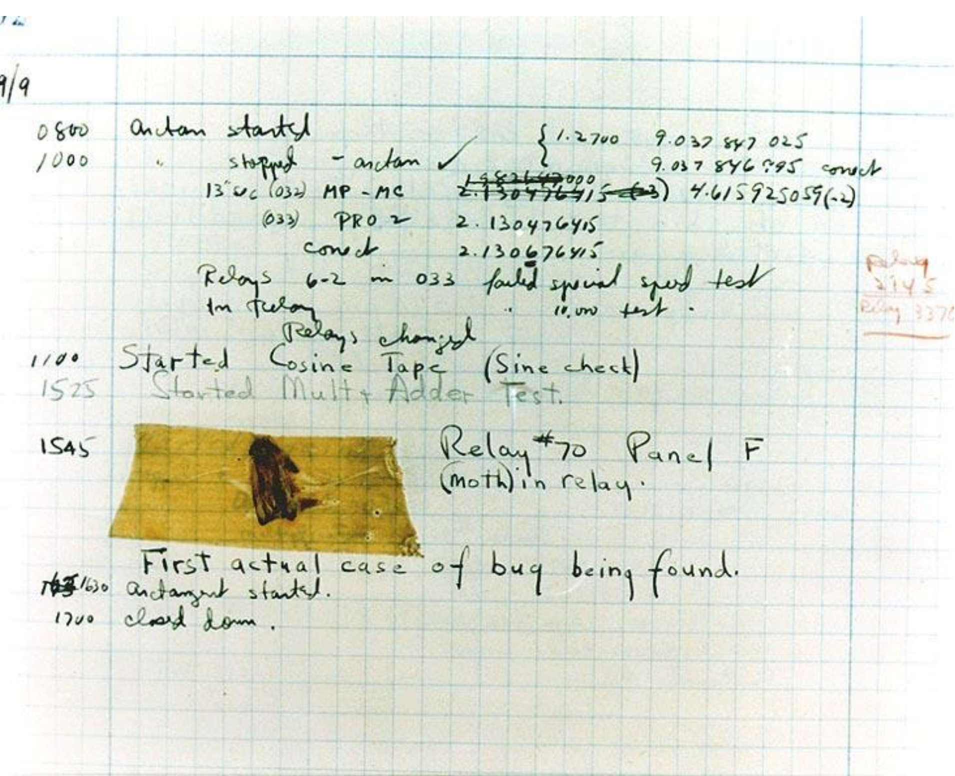
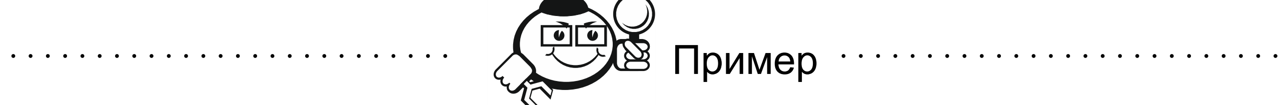


Рис. 1.3 – Первый фактический случай найденного жука

С тех пор 9 сентября принято отмечать как день тестировщика программного обеспечения. Хотя этот праздник и неофициальный, однако наличие у тестировщиков собственного профессионального праздника говорит о том, насколько важна эта профессия в мире.

В 1950–1960-х гг. процесс тестирования был предельно формализован, отделен от процесса непосредственной разработки ПО. Фактически тестирование представляло собой скорее *отладку программ* (*debugging*). Существовала концепция так называемого *исчерпывающего тестирования* (*exhaustive testing*), при котором осуществлялась проверка всех путей выполнения кода со всеми возможными входными данными.



Рассмотрим в качестве простого примера калькулятор. Сколько комбинаций входных параметров нужно, чтобы протестировать одну только операцию сложения двух чисел по 8 знаков? Получается $10^8 \times 10^8 = 10^{16}$ комбинаций. Если прикинуть, что на каждый тест мы потратим около 1 секунды, то время полного тестирования займет у нас 10^{16} секунд, то есть больше 317 097 919 лет непрерывных тестов.

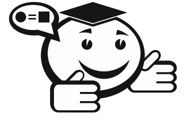
Таким образом, исчерпывающее тестирование невозможно, т. к. количество возможных путей и входных данных очень велико, а также при таком подходе сложно найти проблемы в документации. По этим причинам исчерпывающее тестирование было отклонено и признано теоретически невозможным. Поэтому одна из целей, которая стоит перед разработчиком тестов, – сократить количество тестов, чтобы укладываться в адекватные сроки тестирования, но при этом не пропускать серьезных ошибок.

В начале 1970-х гг. тестирование ПО обозначалось как «процесс, направленный на демонстрацию корректности продукта» или как «деятельность по подтверждению правильности работы ПО». Хотя концепция была теоретически перспективной, на практике она требовала много времени и была недостаточной. Было решено, что доказательство правильности – неэффективный метод тестирования ПО. Однако в некоторых случаях демонстрация правильной работы используется и в наши дни, например, во время приемо-сдаточных испытаний.

В 1980-х гг. тестирование стало включать предупреждение дефектов. Появилось проектирование тестов – наиболее эффективный из известных методов

предупреждения ошибок. В это же время стали высказываться мысли, что необходима методология тестирования, в частности, что тестирование должно включать проверки на всем протяжении цикла разработки, и это должен быть управляемый процесс. В ходе тестирования надо проверить не только собранную программу, но и требования, код, архитектуру, сами тесты.

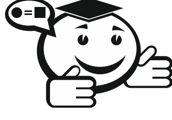
В 1980 г. компьютерный инженер и ученый Гленфорд Майерс дал следующее определение тестирования.



Тестирование ПО (*software testing*) *– это процесс выполнения программы с намерением найти ошибки* [4].

В начале 1990-х гг. в понятие «тестирование» стали включать планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений, и это означало переход от тестирования к обеспечению качества, охватывающему весь цикл разработки ПО. В это время начинают появляться различные программные инструменты для поддержки процесса тестирования: более продвинутые среды для автоматизации с возможностью создания скриптов и генерации отчетов, системы управления тестами, ПО для проведения нагрузочного тестирования.

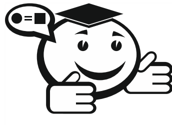
В 1990 г. ученый Борис Бейзер дал новое определение тестирования.



Тестирование ПО *– это не действие, это интеллектуальная дисциплина, имеющая целью получение надежного программного обеспечения без излишних усилий на его проверку* [5].

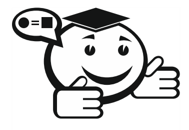
В 2000-х гг. появилось еще более широкое определение тестирования, в него было добавлено понятие «оптимизация бизнес-технологий». Основной подход заключается в оценке и максимизации значимости всех этапов жизненного цикла разработки ПО для достижения необходимого уровня качества, производительности и доступности.

В 2004 г. было выработано новое определение, которое включено в международный стандарт ISO/IEC TR 19759 SWEBOK (Software Engineering Body of Knowledge) 2015 г., в котором аккумулированы актуальные знания по программной инженерии.



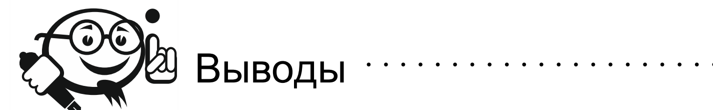
***Тестирование ПО** – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранном определенным образом* [6].

Главное в этом определении то, что тестирование позиционируется как конечный процесс. Далее определение позволило определить понятие дефекта (бага).



***Дефект** (**баг**, **bug**) – это отклонение фактического результата (actual result) от ожидаемого результата (expected result).*

Давайте подведем итоги.



Тестирование (*testing*) – это самый низкий уровень, забота о качестве в виде обнаружения багов до того, как их найдут пользователи.

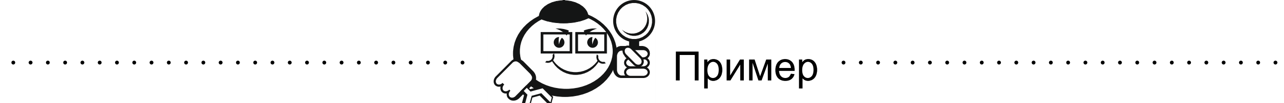
Контроль качества (*quality control*, *QC*) – следующий уровень, анализ результатов тестирования и качества сборки в процессе разработки. Это измерение качества продукта.

Обеспечение качества (*quality assurance*, *QA*) – это измерение и управление качеством процесса, который используется для создания качественного продукта, это забота о качестве в виде превентирования появления багов.

Таким образом, мы можем построить модель иерархии процессов обеспечения качества: тестирование – часть QC, QC – часть QA (рис. 1.4).

Рис. 1.4 – Модель иерархии процессов обеспечения качества

Рассмотрим эти процессы на примере велосипеда.



За что отвечает тестирование и обеспечение качества при конструировании велосипеда?

С помощью тестирования мы определяем, работают ли все детали и сам велосипед. Из нужных ли материалов сделаны все части.

QA отвечает за обеспечение соответствия всех этапов конструирования велосипеда обязательным стандартам качества, т. е. внимание качеству велосипеда уделяется еще до его конструирования и сборки.

Очень часто в процессе тестирования ПО встречаются еще два термина – это *валидация* и *верификация*. Эти два понятия тесно связаны с процессами тестирования и обеспечения качества. Несмотря на кажущуюся схожесть, термины «тестирование», «верификация» и «валидация» тоже означают разные уровни проверки корректности работы ПО.

На рисунке 1.5 валидация представлена наружным кольцом диаграммы и полностью включает в себя верификацию и тестирование.

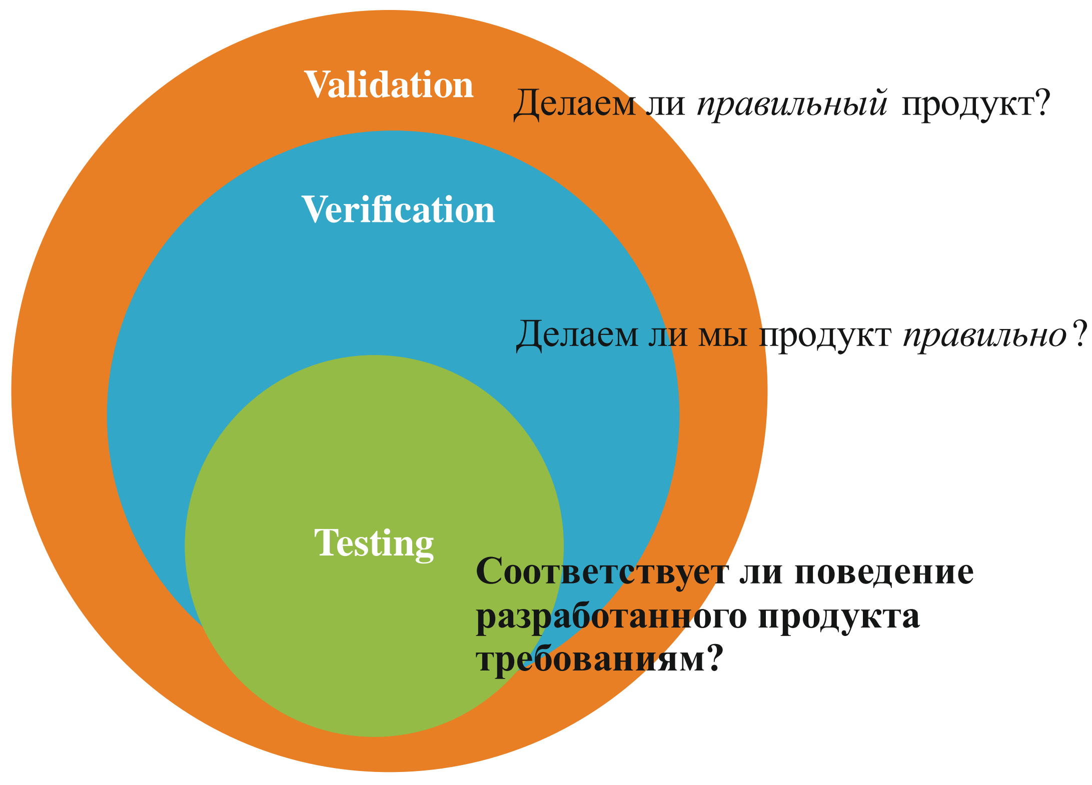


Рис. 1.5 – Диаграмма отношения валидации, верификации и тестирования

Валидация – проверка правильности продукта. *Верификация* подтверждает, что продукт делался правильно на каждом этапе жизненного цикла и включает в себя тестирование. *Тестирование* устанавливает, что требования были правильно реализованы.

Верификация (*verification*) *– проверка того, что продукт делался правильно, т. е. проверка того, что он разрабатывался в соответствии со всеми требованиями по отношению к процессу и этапам разработки.*

Валидация (*validation*) *– проверка того, что сам продукт правилен, т. е. установление того, что он удовлетворяет требованиям, ожиданиям пользователя, заказчика и других заинтересованных сторон* [7].

Если посмотреть на эти три процесса с точки зрения вопроса, на который они дают ответ, то тестирование отвечает на вопрос «как это сделано?» или «соответствует ли поведение разработанной программы требованиям?», верификация – «что сделано?» или «соответствует ли разработанная система требованиям?», а валидация – «сделано ли то, что нужно?» или «соответствует ли разработанная система ожиданиям заказчика?».

Вернемся к примеру о велосипеде.

Пример

За что отвечает верификация и валидация на примере велосипеда?

Верификация: Педали есть? – Есть. Все ли заявленные детали есть? – Есть.

Валидация: Едет велосипед? Не едет. От того, что все детали у нас есть, толку от велосипеда нет, а значит, не выполняется главное заявленное требование к велосипеду.

Давайте поговорим о требованиях, которые предъявляются к ПО.

Помимо требований, установленных контрактом / техническим заданием, существует ряд общих требований, записанных в стандарте ISO, которым должен соответствовать любой программный продукт.

В его рамках имеются следующие характеристики:

- *Надежность* (*reliability*) – способность программного обеспечения сохранять свою функциональность (с определенным удовлетворительным уровнем качества) при установленных условиях за установленный период времени. Например, интернет-магазин должен быть доступен в любое время и работать без сбоев.
- *Практичность* (*usability*) – это свойство системы, характеризующее ее привлекательность для пользователя и полезность в применении для решения поставленных перед пользователем задач. Понятие практичности сложно формализуемо и, как следствие, сложно измеримо. Например, онлайн-банк должен поддерживать перевод денежных средств на счета разных банков и для оплаты различных услуг, а также возможность быстрого выполнения множества операций.
- *Эффективность* (*efficiencies*) – это способность продукта обеспечивать заданный уровень качества функционирования программного обеспечения и объем используемых ресурсов при установленных условиях. Под ресурсами могут пониматься и другие программные продукты, аппаратные средства и т. д.; т. е. насколько рационально программа относится к ресурсам (память, процессор) при выполнении своих задач. Например, способность программы обеспечивать пользователю максимальное достижение целей даже при минимальных условиях: в онлайн-банке можно выполнять автоплатеж.
- *Сопровождаемость* (*maintainability*) – это свойство продукта, характеризующее его способность к модификации и изменению конфигурации, т. е. насколько легко возможно внести изменения в существующую конфигурацию продукта. Например, зачастую бывают такие ситуации, когда проект разрабатывается одной компанией, а поддерживается или дорабатывается уже другой, поэтому особенно важно, чтобы система была разработана в соответствии с требованиями к коду и была снабжена необходимым объемом сопроводительной документации.
- *Мобильность* (*portability*) – способность программного обеспечения быть перенесенным из одного окружения в другое. Например, интернет-магазин должен открываться во всех современных браузерах, желательно наличие мобильной версии.

- *Завершенность* (*completeness*) – готовность программного обеспечения к непосредственной работе. Например, одна из наиболее распространенных просьб программистов перед началом тестирования – выполнить очистку базы либо ряд скриптов, чтобы привести базу в порядок, и т. д. Все эти действия говорят о том, что система не готова к непосредственному использованию.
- *Расширяемость* (*expansibility*) – возможность доработки и развития программного продукта, добавления новой функциональности, если такая потребность возникнет. Например, очень распространен такой случай, когда уже существующую систему дополняют новыми функциями. Архитектура сайта должна быть создана таким образом, чтобы позволять программистам делать новые «надстройки» функциональности.
- *Возможность повторного использования* (*reusability*) – возможность использования продукта или отдельных его компонент в других системах и комплексах. Например, некоторые модули в разрабатываемых системах могут быть повторно использованы в будущих или настоящих проектах. Создав эффективный движок для игры, можно использовать его в других играх с похожей логикой.
- *Простота и удобство использования* (*user friendliness*) – наличие дружественного интуитивно понятного интерфейса. Например, при посещении онлайн-банка для пользователя должно быть понятно и интуитивно очевидно, как сделать перевод, куда ввести реквизиты.

Каждому из этих требований соответствует свой вид тестирования. Основные виды тестирования мы рассмотрим позже.

Давайте рассмотрим принципы тестирования. Эти принципы являются общим руководством для тестирования в целом, они были сформулированы профессором программной инженерии Высшей политехнической школы (Цюрих, Швейцария) Бертраном Мейером в статье [8].

Принцип 1. Тестирование демонстрирует наличие дефектов.

На практике данный принцип означает, что тестировщик никогда не может гарантировать 100% отсутствия дефектов, даже если ни одного не было найдено.

Принцип 2. Исчерпывающее тестирование недостижимо.

В этом мы убедились на простом примере: насколько хорошо бы ни были спроектированы наши тесты и насколько максимально бы мы ни учитывали все

возможные входные условия – все возможные варианты протестировать невозможно.

Принцип 3. Раннее тестирование.

Здесь все просто: чем раньше мы найдем дефект, тем дешевле нам обойдется его исправление. Понятно, что исправить строчку в спецификации или коде проще и дешевле, чем вносить изменения в готовый продукт.

Принцип 4. Скопление дефектов.

Практика показывает, что, как правило, большее количество дефектов концентрируется в небольшом количестве модулей. Поэтому если в каком-то модуле были обнаружены дефекты, то тестировать его надо с особенной тщательностью.

Принцип 5. Парадокс пестицида.

В 1990 г. Борис Бейзер ввел термин *pesticide paradox*, чтобы описать тот факт, что ПО становится устойчивым к одним и тем же тестам. То же самое происходит с насекомыми и пестицидами. Если постоянно травить вредителей одним и тем же пестицидом, у них появляется к ним иммунитет. Данный принцип означает, что после определенного момента одни и те же тесты перестают находить ошибки в программном обеспечении. Поэтому необходимо периодически менять набор тестов и входных данных.

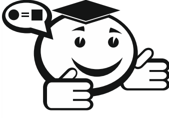
Принцип 6. Тестирование зависит от контекста.

Ясно, что выбор вида тестирования в той или иной ситуации зависит от того, какие задачи мы перед собой ставим в настоящий момент. Например, набор тестов при тестировании нагрузки будет принципиально отличаться от тактики тестирования простоты использования.

Принцип 7. Заблуждение об отсутствии ошибок.

Данный принцип частично повторяет принцип 1, т. к. утверждает, что отсутствие найденных дефектов не гарантирует их фактическое отсутствие.

Знание данных принципов облегчает задачи, стоящие перед тестировщиком, и помогает сделать правильный выбор в использовании той или иной стратегии тестирования. Но не надо забывать про здравый смысл! Главная аксиома тестировщика – все ошибаются: программисты, аналитики, составители требований и документации. Всех нужно проверять! В этом тестирование очень похоже на экспериментальную науку. Таким образом, тестировщики – это люди, которые ставят эксперименты.



Тестировщик (*software tester*) *– специалист, который моделирует различные ситуации, которые могут возникнуть в процессе использования предмета тестирования, управляет выполнением программы, создает искусственные ситуации, наблюдает за поведением программы и сравнивает наблюдаемое поведение с ожидаемым, составляет отчет о проведенном тестировании, в котором должен быть указан анализ и причины возникших проблем.*

Деятельность тестировщика обычно подразумевает как минимум три модели поведения:

- Обычный пользователь, который не читал инструкций или не способен их прочитать. По этой модели выявляется несоответствие интерфейса программы существующим стереотипам.
- Добросовестный пользователь, который действует в строгом соответствии с инструкциями. По этой модели происходит поиск ошибок как в логике работы программы, так и в документации на программу.
- Злонамеренный пользователь, который стремится использовать программу непредусмотренным способом. Именно при негативном тестировании чаще выявляются дефекты.

Что нужно знать и уметь, чтобы стать тестировщиком?

Несмотря на относительную молодость профессии тестировщика, сегодня сформирован набор требований, которые нужны для старта. Существует шутка, что главное, чему нужно научить тестировщика, – обнаружить дефект и убедить программиста, что это действительно дефект, который необходимо устранить. Программисты, как правило, очень любят свой код и очень не любят его исправлять, и иногда это выливается в настоящие войны.

Все навыки тестировщика можно условно разделить на три группы:

- профессиональные – это ключевые навыки, отличающие тестировщика от других IT-специалистов;
- технические – это общие навыки в области IT, которыми тем не менее должен обладать и тестировщик;
- личностные.

Профессиональные навыки:

1. Знание видов тестирования.
2. Знание инструментов и библиотек для автоматизации тестирования.

3. Умение пользоваться специальным ПО для автоматизированного тестирования и регистрации ошибок.
4. Навыки тест-дизайна и тест-анализа.

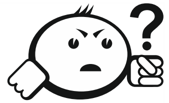
Технические навыки:

1. Знание иностранных языков. В сфере IT знание английского необходимо.
2. Программирование. Это упрощает жизнь тестировщику. Можно ли тестировать без знания программирования? Да, можно. Можно ли это делать по-настоящему хорошо? Нет.
3. Базы данных и язык SQL. Здесь от тестировщика тоже не требуется квалификация на уровне узких специалистов, но минимальные навыки работы с наиболее распространенными СУБД и умение писать простые запросы можно считать обязательными.
4. Понимание принципов работы сетей и операционных систем. Хотя бы на минимальном уровне, позволяющем провести диагностику проблемы и решить ее своими силами, если это возможно.
5. Понимание принципов работы веб-приложений и мобильных приложений. В наши дни почти все пишется именно в виде таких приложений, и понимание соответствующих технологий становится обязательным для тестирования.

В завершение также отметим *личностные качества*, позволяющие тестировщику быстрее стать отличным специалистом:

1) повышенная ответственность и исполнительность;
2) хорошие коммуникативные навыки, способность ясно, быстро, четко выражать свои мысли;
3) терпение, усидчивость, внимательность к деталям, наблюдательность;
4) хорошее абстрактное и аналитическое мышление;
5) способность ставить нестандартные эксперименты, склонность к исследовательской деятельности.

Легче всего войти в IT-сферу в качестве тестировщика. Проработав несколько лет, довольно просто перейти в другую область, например стать разработчиком, аналитиком или даже руководителем.



Контрольные вопросы по главе 1

1. Что включает обеспечение качества?
2. Что такое качество ПО?
3. Чем отличается валидация от верификации?
4. Что такое дефект?
5. Назовите общие требования, которым должен соответствовать любой программный продукт.
6. Перечислите и поясните 7 основных принципов тестирования.
7. Зачем нужны тестировщики ПО?
8. Чем отличаются профессиональные навыки тестировщика от технических?
9. Каким личными качествами должен обладать тестировщик?
10. Назовите карьерные перспективы тестировщиков. Определите для себя лично, каких профессиональных целей Вам хотелось бы достигнуть в области контроля качества ПО, и какие шаги для этого следует совершать.

2 Дефекты и их жизненный цикл

Думай как баг, действуй как баг, и ты найдешь баг.

Народная мудрость

Многие утверждают, что любое тестирование – это поиск багов. Однако есть и те, кто не согласен с этим утверждением, они считают, что тестирование – это процесс исследования ПО с целью получения информации о качестве продукта. Ну а качество – это отсутствие дефектов. Поэтому эти утверждения взаимосвязаны.

Давайте рассмотрим, что же такое дефект или баг. Приведем пример из книги [9] Романа Савина.

1. Соседка по комнате рекламирует себя как хорошую, на все руки хозяйку, а выясняется, что она даже яичницу пожарить не в состоянии.
2. Вы купили книгу по программированию, а в ней кулинарные рецепты.
3. Заставили соседку почитать книгу по кулинарии, но она прочитала книгу по программирования.

Мы видим во всех случаях отклонение фактического от ожидаемого:

1. Ожидаемый результат – девушка умеет готовить.

Фактический результат – остались без обеда.

2. Ожидаемый результат – знания по программированию.

Фактический результат – знания по кулинарии.

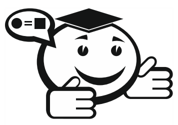
3. Ожидаемый результат – обед будет приготовлен.

Фактический результат – еще один день без обеда.

Таком образом, *дефект* – это несоответствие продукции заявленным требованиям. *Ожидаемый результат* – поведение системы, описанное в требованиях. *Фактический результат* – поведение системы, наблюдаемое в процессе тестирования.

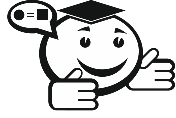
2.1 Классификация дефектов

Дефект в тестировании ПО – достаточно широкое понятие. Существует несколько видов дефектов. В литературе, посвященной программной инженерии, встречается множество терминов, описывающих нарушение функционирования программных систем: *недостатки* (*faults*), *отказы* (*failures*), *ошибки* (*errors*) и др. Соответствующая терминология описана в IEEE Std. 610-90 и также обсуждается в области знаний SWEBOK «Качество программного обеспечения». Рассмотрим некоторые из них.



Ошибка (*error*) *– это дефект, заложенный в логике системы, в результате которого ПО не соответствует требованиям.*

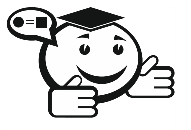
Например, неточность округления в бухгалтерской программе (до десятых единиц вместо сотых), заложенная программистом при кодировании функционала, может вызвать серьезные проблемы при использовании. Такие ошибки заслуживают наибольшего внимания со стороны специалиста по тестированию.



Недочет (*fault*) *– это дефект системы, при котором она все же может выполнять свои основные функции.*

Например, неверное отображение кнопки, конечно, досадное недоразумение, но в принципе оно не мешает нажать на данную кнопку и увидеть результат действия, поэтому недочет не блокирует работу пользователя. Опечатки в тексте тоже можно назвать недочетом, но бывают и критические недочеты. Организаторы масштабного мероприятия «ВолгаФест» в 2017 г. допустили досадную опечатку, перепутав имя первого космонавта Юрия Гагарина. Если верить фотографии, которая была вывешена в виде заставки на главной сцене фестиваля, легендарного летчика звали Алексеем (рис. 2.1). Организаторы «ВолгаФеста» сразу признали свою оплошность. «Такие недочеты случаются из-за больших объемов работы. Надеемся, что эта ошибка никого не расстроила», – написали тогда в комментарии пресс-службы.

Рис. 2.1 – Опечатка в макете, которая дорого стоит



. . . .

Отказ (*failure*) *– это неспособность системы выполнять свои функции.*

. . . .

Отказы, как правило, выявляются еще на стадии дымового тестирования (о котором будет рассказано позже), т. е. практически при первичном осмотре ПО. Конечно же, отказы требуют скорейшего устранения и повторного тестирования в кратчайшие сроки.

Например, для сайта отказом будет отсутствие нужной страницы после нажатия на ссылку перехода, знаменитая ошибка *error 404* (рис. 2.2).

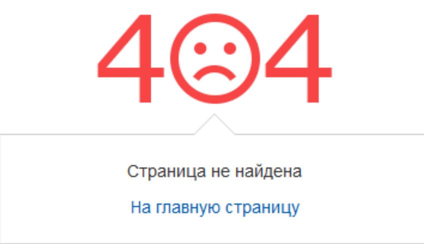


Рис. 2.2 – Ошибка error 404

Все ошибки, которые возникают в программах, принято подразделять на следующие классы:

- логические и функциональные ошибки;
- ошибки вычислений и времени выполнения;
- ошибки ввода-вывода и манипулирования данными;
- ошибки интерфейсов;
- ошибки объема данных и др.

Логические ошибки являются причиной нарушения логики алгоритма, внутренней несогласованности переменных и операторов, а также правил программирования.

Функциональные ошибки – следствие неправильно определенных функций, нарушения порядка их применения или отсутствия полноты их реализации и т. д.

Ошибки вычислений возникают по причине неточности исходных данных и реализованных формул, погрешностей методов, неправильного применения операций вычислений или операндов. Ошибки времени выполнения связаны с необеспечением требуемой скорости обработки запросов или времени восстановления программы.

Ошибки ввода-вывода и манипулирования данными являются следствием некачественной подготовки данных для выполнения программы, сбоев при занесении их в базы данных или при выборке из нее.

Ошибки интерфейса относятся к ошибкам взаимосвязи отдельных элементов друг с другом, что проявляется при передаче данных между ними, а также при взаимодействии со средой функционирования.

Ошибки объема относятся к данным и являются следствием того, что реализованные методы доступа и размеры баз данных не удовлетворяют реальным объемам информации системы или интенсивности их обработки.

Существует еще такое понятие, как *фича*. Это сленговое обозначение каких-либо необычных признаков какого-либо свойства ПО. Фичей могут выступать необычные программные возможности, особые функции; что-либо, что привлекает особое внимание. Случается так, что найденные баги, т. е. непредвиденные ошибки, выдаются за особенность, включенную в программное обеспечение специально. Отсюда и пошла шутливая фраза, облетевшая весь Интернет: «Это не баг, это фича!».

Пример бага, ставшего фичей, – крутой нрав Ганди в игре Civilization (культовая пошаговая глобальная стратегия). Алгоритм первой части не мог придать агрессии отрицательное значение, поэтому из-за жесткой экономии памяти уровень агрессивности мог принимать значение от 0 до 255. Махатма Ганди, лидер игровой Индии, имел агрессивность в 1. Из-за игровой ошибки 1 в итоге приравнивалась к 255, и сверхагрессивная Индия начинала ковровые бомбардировки всех стран мира ядерным оружием [11]. Ошибка обернулась шуткой и прижилась в игре.

На рисунке 2.3 приведено процентное соотношение ошибок при разработке ПО [10].

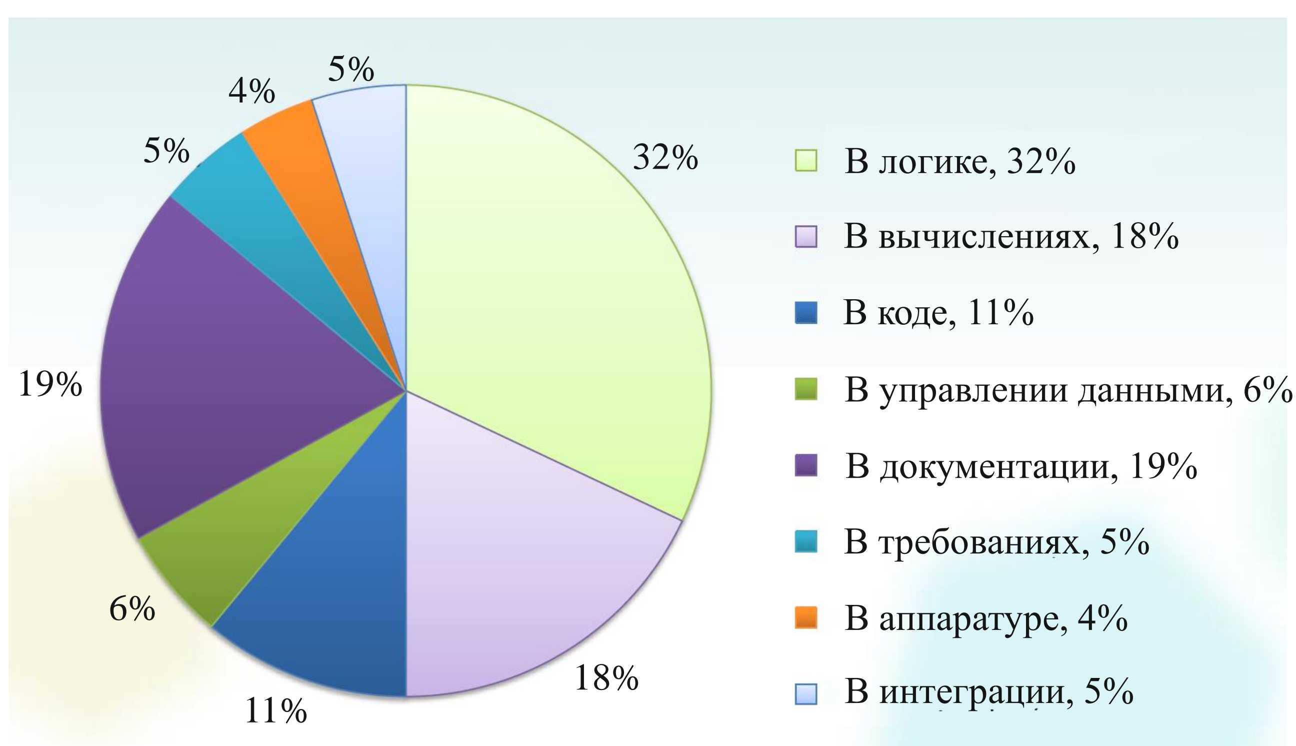


Рис. 2.3 – Процентное соотношение ошибок в ПО

Как видим, большее количество ошибок допускается в логике программы.

Пример

Примеры функциональных ошибок:

1. Не сохраняются изменения данных в профиле.

2. Не работает добавление комментария.
3. Не работает удаление товара из корзины.
4. Не работает поиск.

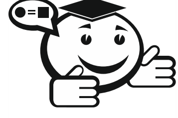
Примеры ошибок интерфейса:

1. Текст выходит за границы поля.
2. Элемент управления сайтом наслаивается на нижестоящий элемент.
3. Не отображается картинка.

Примеры логических ошибок

1. Можно поставить дату рождения в будущем. 31 февраля, 31 июня и т. д.
2. Можно сделать заказ, не указав адрес доставки.

Найти дефект – самое простое, главное – его локализовать!



Локализация бага (*localization bug*) *– это процедура нахождения и описания условий, при котором дефект повторяется.*

То есть необходимо найти причины возникновения бага. Необходимо его проанализировать, задокументировать и воспроизвести.

Серьезность (*severity*) *– это атрибут, характеризующий влияние дефекта на работоспособность приложения.*

Выставляя серьезность дефекта, тестировщик оценивает его влияние на работоспособность приложения. Чем выше серьезность, тем масштабнее негативные последствия данного дефекта.

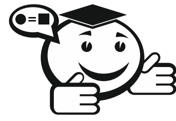
Принята следующая градация серьезности дефекта:

- *Блокирующая* (*Blocker*). Блокирующая ошибка приводит приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.
- *Критическая* (*Critical*). Критическая ошибка – неправильно работающая ключевая бизнес-логика, дыра в системе безопасности; проблема, влекущая временное падение сервера или приводящая в нерабочее состояние некоторую часть системы без возможности обойти проблему,

используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системы.

- *Значительная* (*Major*). Значительная ошибка – ошибка, при которой часть основной бизнес-логики работает некорректно. Ошибка некритична или есть возможность для работы с тестируемой функцией, используя другие входные точки.
- *Незначительная* (*Minor*). Незначительная ошибка не нарушает бизнес-логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.
- *Тривиальная* (*Trivial*). Тривиальная ошибка не касается бизнес-логики приложения, это плохо воспроизводимая проблема пользовательского интерфейса; проблема сторонних библиотек или сервисов; проблема, не оказывающая никакого влияния на общее качество продукта.

Также полезно указать приоритет исправления каждого дефекта.



Приоритет (*priority*) *– это атрибут, указывающий на очередность выполнения задачи или устранения дефекта.*

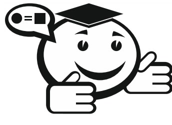
Чем выше приоритет, тем быстрее нужно исправить дефект. Например, слово, напечатанное с ошибкой, может иметь самый низкий уровень серьезности, но перед релизом эта ошибка может иметь наивысший приоритет и должна быть экстренно исправлена.

Градация приоритета дефекта может быть следующая:

- *Высокий* (*High*). Ошибка должна быть исправлена как можно быстрее, т. к. ее наличие является критической для ПО.
- *Средний* (*Medium*). Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.
- *Низкий* (*Low*). Ошибка должна быть исправлена, ее наличие не является критичной и не требует срочного решения.

2.2 Жизненный цикл дефекта

За время своей жизни дефекты проходят определенные стадии, которые характеризуются их статусами.



Жизненный цикл бага (*bug workflow*) *– последовательность этапов, которые проходит баг на своем пути с момента его создания до окончательного закрытия.*

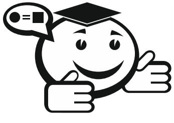
Самый простой жизненный цикл (*workflow*) без применения инструментальных средств: тестировщик приходит к разработчику и просто сообщает об ошибке, но здесь есть риск, что разработчик может забыть про ошибку и не исправить ее. Поэтому обычно в IT-компаниях дефекты документируют в системах учета дефектов (баг-трекинговые системы, bug tracking system, BTS, bug-tracker) с возможностью общего доступа к ним. Существует множество таких систем, которые позволяют не только создавать задачи, менять их статусы, но и создавать отчеты об инцидентах (баг-репорты, *bug report*).

На рынке ПО предложено огромное количество bug-tracker, среди них есть и свободно распространяемые и платные.

Например:

- *Redmine* – открытое серверное веб-приложение для управления проектами и задачами (в том числе для отслеживания ошибок).
- *Bugzilla* – свободная система отслеживания ошибок (баг-трекинга) с веб-интерфейсом.
- *Jira* – коммерческая система отслеживания ошибок, предназначена для организации взаимодействия с пользователями, хотя в некоторых случаях используется и для управления проектами.
- *Mantis* – свободно распространяемая система отслеживания ошибок в программных продуктах. Обеспечивает взаимодействие разработчиков с пользователями, позволяет пользователям создавать сообщения об ошибках и отслеживать дальнейший процесс работы над ними со стороны разработчиков. Система имеет гибкие возможности конфигурирования, что позволяет настраивать ее не только для работы над программными продуктами, но и в качестве системы учета заявок.

В каждой BTS наименования важности (серьезности), приоритеты и статусы дефектов могут отличаться от приведенных и зависят от конкретной системы, но суть от этого не меняется. Рассмотрим распространенные атрибуты жизненного цикла.



. . . .

Статус (*status*) *– основной атрибут, определяющий текущее состояние дефекта.*

. . . .

Статус отражает меру активности бага от начального состояния, когда он еще не подтвержден как баг, до завершения, когда баг исправлен/решен. Набор *атрибутов* зависит от конкретной BTS, но чаще встречаются следующие:

- *NEW* (*Новый*). Дефект только что зарегистрирован, в зависимости от анализа дефекта его статус меняется на *Отказ* или *Назначен*.
- *ASSIGNED* (*Назначен*) или *OPEN* (*Открыт*). Дефект просмотрен и открыт (можно сказать, признан) для исправления и может быть назначен на другого сотрудника или переведен в состояние *NEW*.
- *REOPENED* (*Открыт заново*). Дефект был решен ранее, однако возникла необходимость вернуться к нему (решение было неверным либо неокончательным).
- *CLOSED* (*Закрыт*). В результате определенного количества циклов баг все-таки окончательно устранен и больше не потребует внимания команды – он объявляется закрытым.

Резолюция – очень важный атрибут, напрямую связанный со статусом, т. е. резолюция – это детализация статуса.

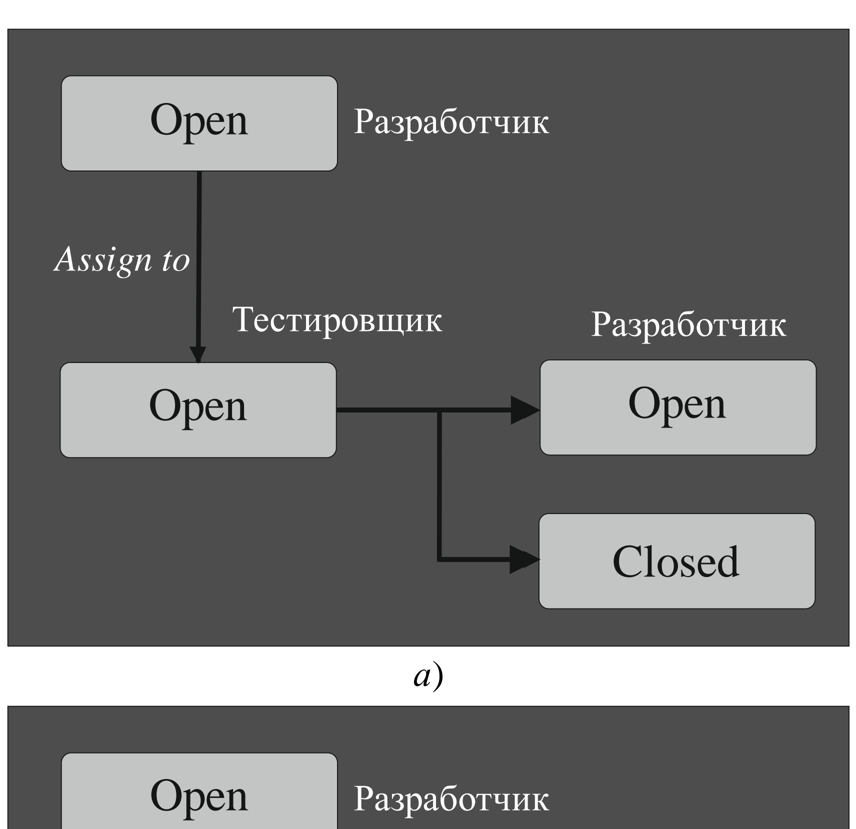
- *FIXED* (*Решено*) – стандартная резолюция, означающая, что задание выполнено или баг исправлен. После дефект переходит обратно в сферу ответственности тестировщика. Как правило, сопровождается резолюцией, например:
  1. Исправлено (исправления включены в версию XXX).
  2. Дубль (повторяет дефект, уже находящийся в работе).
  3. Не исправлено (работает в соответствии со спецификацией, имеет слишком низкий приоритет, исправление отложено до следующей версии и т. п.).
  4. Невоспроизводимо (запрос дополнительной информации об условиях, в которых дефект проявляется).
- *INVALID* (*Некорректно*) – неправильная или некорректная постановка задачи.
- *WONTFIX* (*Проблема есть, но решаться не будет*) – такая резолюция может быть вынесена в отношении *request for enhancement* (просьб об

усовершенствовании), которые хоть и имеют смысл, являются слишком трудновыполнимыми и не являются обязательными.

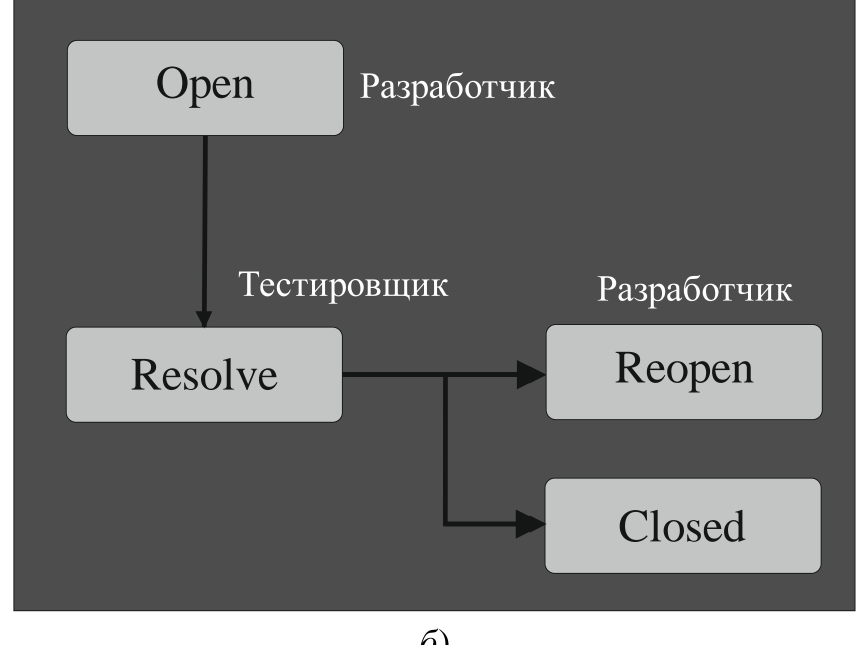
- *DUPLICATE* (*Дублирует*) – описанная проблема уже зарегистрирована в другом баге.
- *WORKSFORME* (*А у меня работает…*) – не удалось воспроизвести описанную проблему ни эмуляцией сценария, ни анализом кода.
- *MOVED* (*Перенесено*) – проблема перенесена в иную систему-tracker.
- *REJECTED* (*Отказ*) сопровождается комментарием программиста или менеджера о причине reject'а (отклонения): некачественное описание дефекта, дублирование эффекта, невозможность воспроизвести дефект. После этого тестер или закрывает дефект (*Closed*), или дополняет комментарии данного дефекта и переводит дефект заново в состояние *Назначен* (*Open*).
- *PENDING RETEST* (*Ожидает повторного тестирования*). На этом этапе баг-репорт ожидает повторного тестирования.
- *RETEST* (*Повторное тестирование*). На этом этапе тестировщики проверяют изменения, внесенные разработчиками, и повторно тестируют их.
- *NOT A BUG* (*Это не дефект*). Баг-репорт может иметь такой статус, например, если клиент попросил внести небольшие изменения в продукт: изменить цвет, шрифт и т. д.
- *VERIFIED* (*Проверен*). Тестировщик проверяет, действительно ли ответственный разработчик исправил дефект. Если бага больше нет, он получает данный статус.

Самые простые варианты жизненного цикла багов схематически представлены на рисунке 2.4, *а* и *б*.

Каждый дефект имеет ряд обязательных свойств, которые нужно внести в BTS при добавлении данного дефекта. Для этого необходимо написать отчет о дефекте (отчет об инциденте, баг-репорт).



а)



б)

Рис. 2.4 – Жизненный цикл дефекта

. . . .

Отчет о дефекте (*баг-репорт, bug report*) *– это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.*

. . . .

В идеале описание дефекта должно иметь следующую структуру:

1. Уникальный номер (*ID*): присваивается автоматически в BTS.

2. Краткое название (*Title*, *Summary* или *Short Description*): короткий текст, который помогает сразу понять, что это за дефект.
3. Описание (*Description* или *Summary*): полное описание дефекта, включая шаги для воспроизведения.
4. Платформа (*Platform*): указывается, что используется для запуска программного обеспечения, в частности имя и версия операционной системы; в случае веб-приложения – имя и версия веб-браузера.
5. Шаги по воспроизведению (*Steps to reproduce*): шаги для воспроизведения дефекта разработчиком.
6. Ожидаемый результат (*Expected Result*). Что должно произойти, когда вы делаете какое-нибудь действие? Что вы ожидаете?
7. Фактический результат (*Actual Result*). Результат ошибки. Что случилось на самом деле?
8. Приложения (*Attachments*): некоторые дефекты сложно описать, поэтому для наглядности к описанию ошибки прилагаются скриншоты, видео или лог-файлы.
9. Серьезность (*Severity*) описывает влияние ошибки.
10. Приоритет (*Priority*). С помощью данного свойства определяется очередность исправления данного дефекта программистом.
11. Статус (*Status*) – состояние отчета об ошибке в любой системе отслеживания багов. Первоначальный статус отчета об ошибке – *New*. После этого статус может измениться на резолюции *Fixed*, *Verified*, *Reopen*, *Won't Fix* и т. д.
12. Комментарии (*Comments*) – комментарии к багу.

Часто шаги воспроизведения (*Steps to Reproduce*), фактический результат (*Result*) и ожидаемый результат (*Expected Result*) записывают в описание.

Если дефект описан согласно данной схеме, то он вызовет меньше всего вопросов, и разработчик, не теряя время на дополнительные разъяснения, приступит к его исправлению. Рассмотрим пример баг-репорта в системе *Mantis*.

Пример

0000159

Отсутствует контент на странице сайта «Кинополис» на вкладке «Сеансы».

Описание

Отсутствие нужной информации на странице сайта https://kino-polis.ru/ приводит к блокировке множества функций сайта, а именно: невозможно выбрать сеанс, забронировать кресло в зале и произвести оплату, т. к. все эти функции недоступны из-за ошибки. Также на странице не прогружается рекламный видеоролик из-за ошибки: *Error loading media: File could not be played!*.

Шаги по воспроизведению

1. Открыть страницу сайта https://kino-polis.ru/.
2. Перейти на вкладку «Сеансы».

Дополнительные сведения

Фактический результат: отсутствие нужной информации на странице, необходимой для выполнения дальнейших действий.

Ожидаемый результат: открытие страницы с сеансами, где можно посмотреть информацию о фильме, описание и продолжительность, характеристики зала; выбрать зал и места, произвести оплату выбранных мест.

Приоритет

Неотложный

Влияние

Блокирующее

Attachments (рис. 2.5).

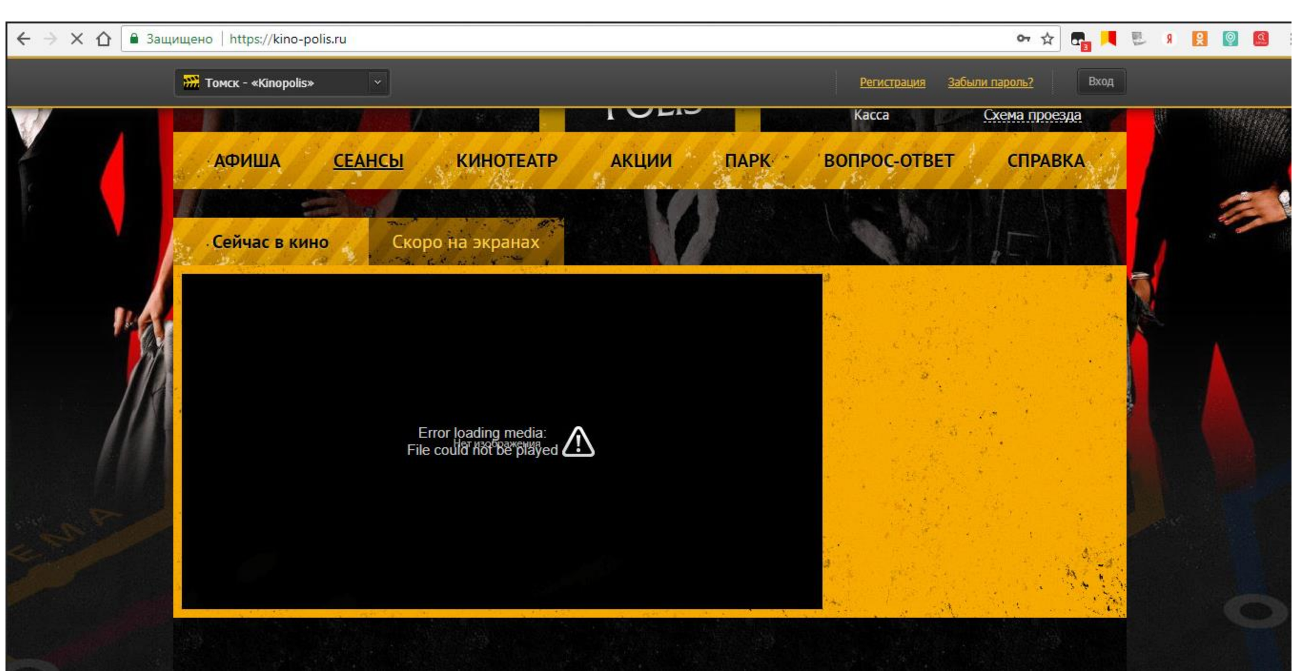
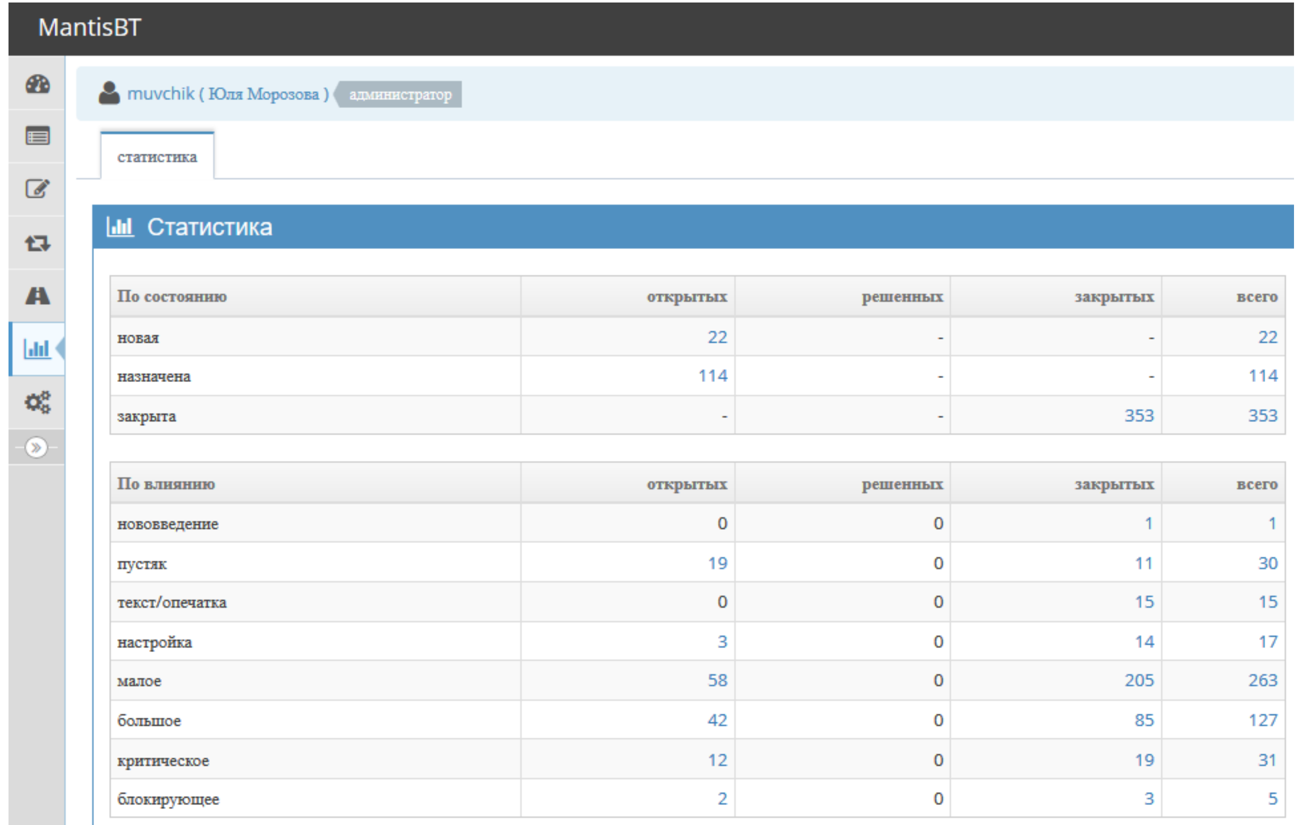


Рис. 2.5 – Дефект на странице сайта «Кинополис» на вкладке «Сеансы»

. . . .

На базе состояний дефектов в системе учета может быть построено множество отчетов, позволяющих оценивать эффективность как команды разработки, так и команды тестирования (рис. 2.6). Отчет представляет собой некий информационный срез о текущем состоянии базы зарегистрированных багов.



По состоянию	открытых	решенных	закрытых	всего
новая	22	-	-	22
назначена	114	-	-	114
закрыта	-	-	353	353

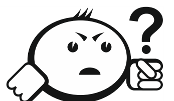
По влиянию	открытых	решенных	закрытых	всего
нововведение	0	0	1	1
пустяк	19	0	11	30
текст/опечатка	0	0	15	15
настройка	3	0	14	17
малое	58	0	205	263
большое	42	0	85	127
критическое	12	0	19	31
блокирующее	2	0	3	5

Рис. 2.6 – Статистика в BTS Mantis

Грамотно написанный баг-репорт увеличивает шансы, что ошибка будет исправлена. Придерживайтесь нескольких простых правил:

1. «Одна ошибка – один баг-репорт». Один баг в репорте может помочь избежать дублирования и путаницы.
2. Указывайте сначала глагол. Все глаголы ставьте в неопределенной форме: нажать, открыть, перейти.
3. Пишите без лишних слов, используйте простые конструкции. Давайте скриншотам говорящие заголовки, например «Сортировка.png».
4. Придерживайтесь принципа «Что-Где-Когда». Обязательно приводите ссылку, максимально подробную, на тот раздел, где баг. Но при этом не стоит полагаться только на ссылку, она может сломаться. Пишите и путь к нужному месту по шагам. Если у вас мобильное приложение, то приводите максимум скриншотов на шаги.

5. Если баг только для авторизованных пользователей, то обязательно указывайте данные для входа: логин и пароль.
6. Все скриншоты или упоминания о них приводите в нужных местах с названием (см. скрин Сортировка.png) или прямо встраивайте в баг-рапорт с помощью !Сортировка.png! или #Сортировка.png (так будет сразу ссылка на баг).
7. Не забывайте указывать серьезность и приоритет дефекта.
8. Обезличенность. Когда сообщаете об ошибке, то сообщаете о дефекте программного обеспечения, а не о дефекте разработчика.



Контрольные вопросы по главе 2

1. Назовите виды дефектов и дайте им определения.
2. Какие ошибки чаще всего встречаются в ПО?
3. Чем отличается серьезность от приоритета?
4. После переключения языка сайта на английский текст баннеров на главной странице не переведен. Опишите дефект.
5. Что такое локализация дефекта?
6. Расскажите, как должен быть оформлен дефект и для чего нужна такая четкая последовательность.
7. Опишите жизненный цикл дефекта.
8. Перечислите известные вам системы учета дефектов.
9. Для чего нужен баг-репорт?
10. Что необходимо описать в баг-репорте?

3 Место тестирования в жизненном цикле разработки ПО

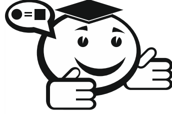
Жизнь – это движение! А тестирование – это жизнь ☺

Ольга Назина (Киселева), автор портала Testbase

Как известно, тестирование занимает особое положение в жизненном цикле программного обеспечения и в то же время является самостоятельным процессом. Чем раньше дефект обнаружен, тем дешевле обходится его исправление, поэтому тестирование должно проводиться на самом раннем этапе разработки ПО. Обычно программный продукт проходит следующие стадии:

- анализ требований к проекту;
- проектирование;
- реализация;
- тестирование продукта;
- внедрение и поддержка.

Говоря другими словами, это время от начального момента создания какого-либо программного продукта до конца его разработки и внедрения.



Жизненный цикл программного обеспечения (*software life cycle*) *– совокупность итерационных процедур, связанных с последовательным изменением состояния программного обеспечения: от формирования исходных требований к нему до окончания его эксплуатации конечным пользователем.*

В жизненном цикле программного обеспечения процесс тестирования в основном стоит за этапом разработки, перед эксплуатацией продукта пользователями. Психология разработчика, а также финансирование создания ПО, как правило, не позволяют начать процесс тестирования пока разработка программного продукта не подойдет к завершению.

Жизненный цикл тестирования – часть жизненного цикла разработки программного обеспечения. Они должны запускаться одновременно!

3.1 Модели жизненного цикла разработки ПО

Выбор модели разработки ПО серьезно влияет на процесс тестирования, определяя выбор стратегии, расписание, необходимые ресурсы и т. д. Моделей разработки ПО много, но в общем случае классическими можно считать водопадную, v-образную, итерационную.

Каскадная, или водопадная (*waterfall*), *модель* жизненного цикла ПО сейчас представляет скорее исторический интерес, т. к. в современных проектах практически не применима (рис. 3.1).

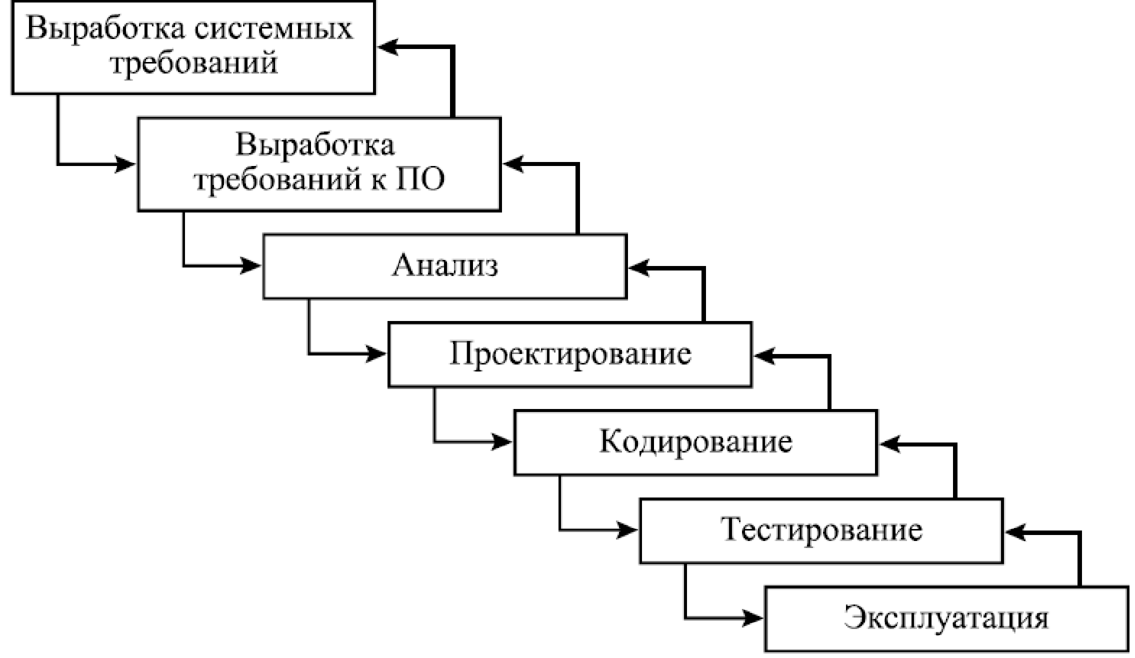


Рис. 3.1 – Водопадная модель

Данная модель разработки программного обеспечения предполагает, что все этапы жизненного цикла проходят последовательно и работы по этапу не начинаются, пока не закончены работы на предыдущем этапе. Тестированию в данной методологии отводится четко заданное время, и по завершении этапа тестирования к вопросу контроля качества команда больше не возвращается. Очень упрощенно можно сказать, что в рамках этой модели в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть» весь проект целиком и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить. Тем не менее водопадная модель часто интуитивно применяется при выполнении относительно простых задач, а ее недостатки послужили прекрасным отправным пунктом для создания новых моделей.

V-образная модель является развитием водопадной (рис. 3.2). Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования.

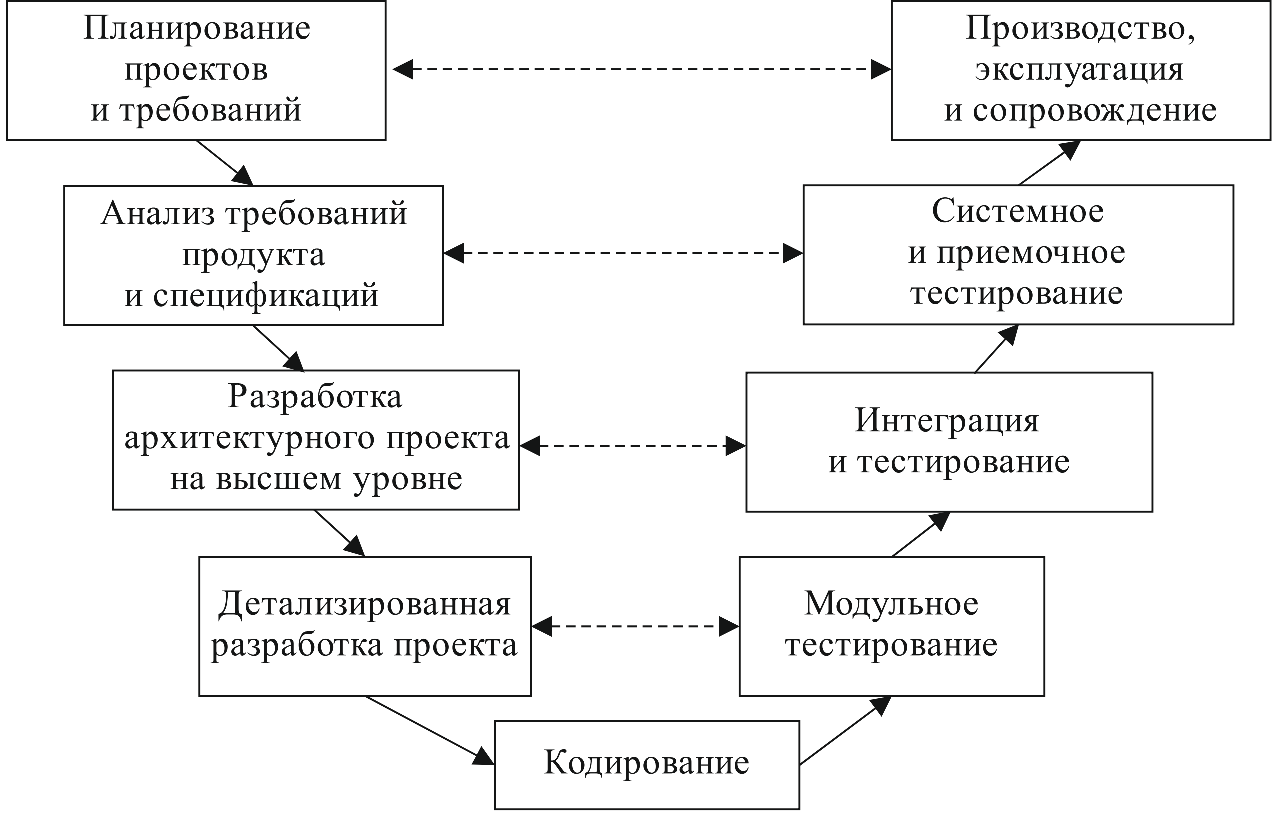


Рис. 3.2 – V-образная модель

При использовании v-образной модели на каждой стадии «на спуске» нужно думать о том, что и как будет происходить на соответствующей стадии «на подъеме». Тестирование здесь появляется уже на самых ранних стадиях развития проекта, что позволяет минимизировать риски, а также обнаружить и устранить множество потенциальных проблем до того, как они станут проблемами реальными. Можно отметить следующие преимущества этой модели: у каждой стадии есть четкий проверяемый результат; с первой же стадии уделяется внимание тестированию; модель хорошо работает для проектов со стабильными требованиями. Однако v-образная модель обладает недостаточной гибкостью и адаптируемостью, что повышает сложность устранения проблем, пропущенных на ранних стадиях развития проекта.

Естественное развитие каскадной и v-образной модели привело к их сближению и появлению современных подходов – методологий, которые по суще-

ству представляет собой рациональное сочетание вышеописанных моделей. Различные варианты эволюционного подхода реализованы в большинстве современных технологий и методов: Rational Unified Process (RUP), Agile и др. Далее рассмотрим наиболее распространенные модели, ориентированные на итеративный процесс разработки.

Один из самых известных процессов, использующих *итеративную модель* разработки, – *Rational Unified Process* (*RUP*). Он был создан во второй половине 1990-х гг. в компании Rational Software. Термин RUP обозначает как методологию, так и продукт компании IBM (ранее Rational) для управления процессом разработки. Методология RUP описывает абстрактный общий процесс, на основе которого организация или проектная команда должна создать специализированный процесс, ориентированный на ее потребности (рис. 3.3).

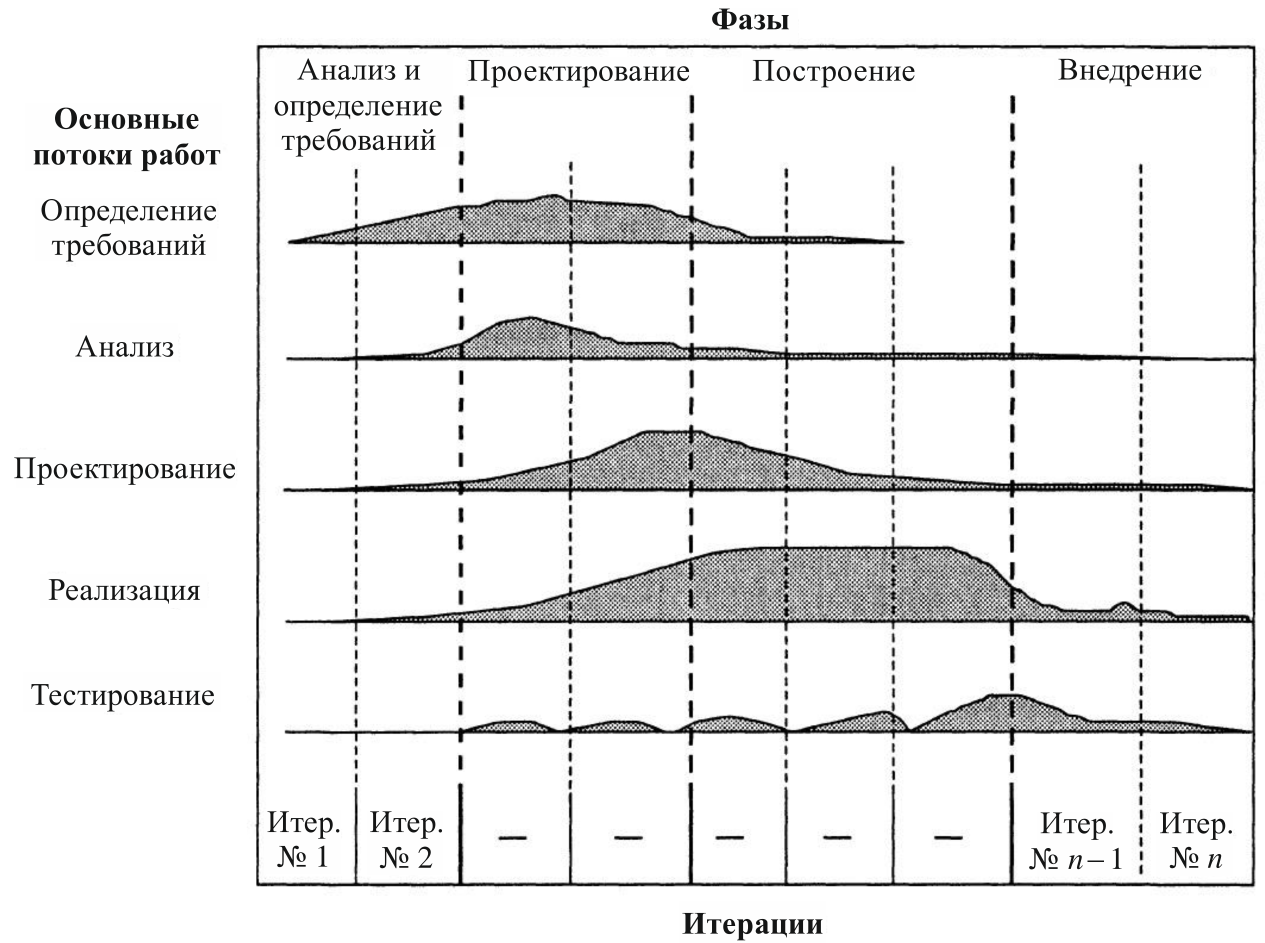


Рис. 3.3 – Модель RUP

Жизненный цикл программного продукта состоит из серии относительно коротких итераций. Весь процесс разработки в данной модели делится на части:

анализ и определение требований, проектирование, построение и внедрение. Однако соотношение этих задач может существенно меняться. Тестирование в RUP обычно проводится циклами, каждый из которых имеет конкретный список задач и целей. Цикл тестирования может совпадать с итерацией или соответствовать ее определенной части. Как правило, цикл тестирования проводится для конкретной сборки системы.

Agile – методология *гибкой разработки* программного обеспечения, предполагающая большое количество итераций. Документ Agile Manifesto описывает 4 идеи и 12 принципов гибкого подхода [12].

Это наиболее современный неформализованный подход к созданию ПО, в процессе которого реагирование на изменения ценятся выше строгого следования плану. Используется для молодых стремительно развивающихся проектов, которые с каждой итерацией программного обеспечения по сути готовы к его релизу. В гибкой методологии разработки после каждой итерации заказчик может наблюдать результат и понимать, удовлетворяет он его или нет. Это одно из преимуществ гибкой модели. К ее недостаткам относят то, что из-за отсутствия конкретных формулировок результатов сложно оценить трудозатраты и стоимость, требуемые на разработку.

Agile стала фундаментальной концепцией для разработки ПО и нашла отражение в других методологиях. Все больше и больше проектов начинают использовать гибкую модель разработки. Многие ведущие IT-компании используют эту модель, новые методологии, такие как SCRUM, Kanban, XP, – лишь распространенные комбинации Agile (рис. 3.4).

Методология XP, разработанная Кентом Беком, является сегодня одной из самых популярных гибких методологий. Она описывается как набор практик: игра в планирование, короткие релизы, метафоры, простой дизайн, переработки кода (refactoring), разработка «тестами вперед», парное программирование, коллективное владение кодом, 40-часовая рабочая неделя, постоянное присутствие заказчика и стандарты кода. Очень большое внимание в методологии уделяется тестированию. Как правило, для каждого нового метода сначала пишется тест, а потом уже разрабатывается собственно код метода до тех пор, пока тест не начнет выполняться успешно. Эти тесты сохраняются в наборах, которые автоматически выполняются после любого изменения кода. Методология подходит для больших или нацеленных на длительный жизненный цикл проектов, постоянно адаптируемых к условиям рынка.

Рис. 3.4 – Гибкие модели

SCRUM – система разработки ПО, которая основана на делении всего процесса на итерации, где в конце каждой из них команда готова предоставить очередной релиз продукта. Вся разработка делится на спринты (зачастую 2–3 недели). Так как в этой системе ставка сделана на общение, то присутствует большое количество митингов:

- *Stand-up* – короткий митинг, проводится каждый день, принимают участие все члены команды и каждый участник отвечает на 3 вопроса: что делал? что будет делать? какие есть блокеры?
- *Плэннинг* проводится в начале спринта, на этом собрании определяют, какие задачи должны быть выполнены за следующий спринт.
- *Ретроспектива* проводится в конце спринта, ее суть – выяснить, что было сделано хорошо и что можно было улучшить.

Эта модель, конечно, обладает рядом преимуществ: заказчик может наблюдать результат в процессе разработки, происходит ежедневный контроль за процессом разработки, есть возможность вносить коррективы во время разработки и, что самое важное, налажена коммуникация со всеми членами команды. Но при таком подходе сложно оценить трудозатраты и стоимость, требуемые на разработку, определить самые узкие места до начала разработки.

Методология *Kanban* впервые появилась в Японии. Эту методику изобрела в 1959 г. компания Toyota. В 1962 г. она начала использоваться для производства автомобилей. В Kanban всего три простых базовых принципа, на которых строится все остальное:

- *Визуализация* процесса разработки. Визуальное представление процесса разработки помогает вам быстро определить, на какой стадии выполнения находится каждая задача. Это может быть крайне полезно в случае большого проекта, состоящего из множества задач. Наиболее подходящий инструмент для этого – канбан-доска. *Канбан-доска* – это таблица с несколькими столбцами. Внутри столбцов находятся стикеры с задачами.
- *Минимизация* WIP. Ограничение числа задач, выполняемых на каждой стадии разработки, помогает эффективно распределять имеющиеся ресурсы и не вызывать простоев в работе.
- *Измерение* и *оптимизация* жизненного цикла разработки. Возможность вносить изменения в процесс разработки является одним из важных компонентов Agile-методологии.

Преимуществом гибких моделей является то, что они позволяют включать тестирование уже на этапе планирования [13].

3.2 Этапы тестирования

В более широком смысле тестирование – это один из этапов разработки ПО, включающий в себя активности (рис. 3.5):

- планирование работ (Test Planning);
- анализ и проектирование тестов (Test Analysis & Test Design);
- выполнение тестирования (Test Execution);
- анализ полученных результатов (Test results evaluation).

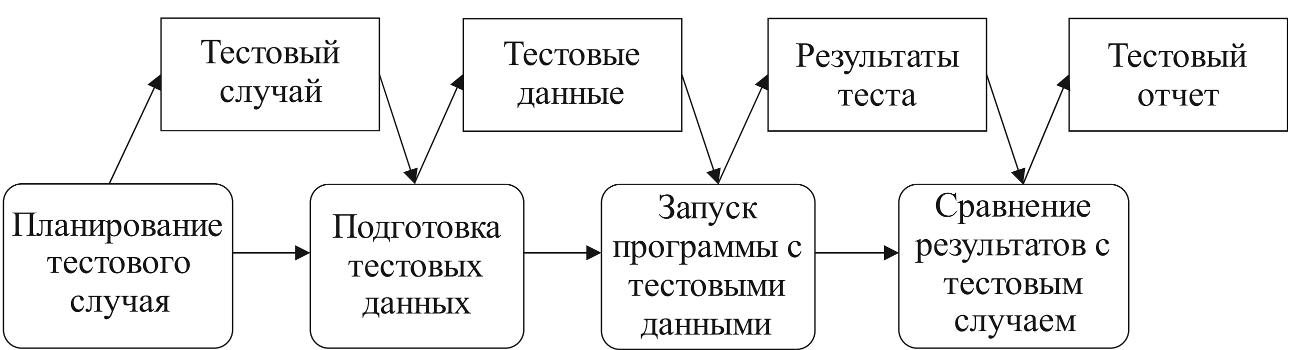
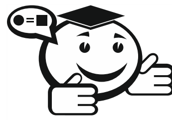


Рис. 3.5 – Этапы тестирования ПО



***Планирование тестирования** – это действия, направленные на определение целей и описание задач тестирования.*

Планирование учитывает данные, полученные при проверке и управлении.

1. Планирование тестирования:

- планирование момента начала тестирования;
- планирование критерия завершения тестирования;
- планирование объема тестирования;
- планирование стратегии тестирования: определяются виды тестирования и их применение по отношению к объекту тестирования;
- планирование тестовой среды.

К артефактам, создаваемым на стадии планирования, можно отнести:

- тестовый план;
- матрицу конфигураций, которая может быть включена в тестовый план как отдельный раздел;
- запрос на выделение тестового оборудования.

2. Анализ и разработка тестовых сценариев.

***Тест-анализ** – это деятельность, во время которой осуществляется анализ имеющихся проектных артефактов: документации (спецификаций, требований, планов), моделей, исполняемого кода и т. д.*

***Тест дизайн** – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы), в соответствии с определенными ранее критериями качества и целями тестировани (IEEE 829).*

Для *анализа* и *проектирования тестов* поставлены следующие основные задачи:

- оценка тестируемости базиса тестирования и объектов тестирования;
- идентификация и расстановка приоритетов условий тестирования, основанных на анализе элементов тестирования, спецификации, поведении и структуре программного обеспечения;

- разработка и расстановка приоритетов тестовых сценариев высокого уровня;
- выявление необходимых данных для поддержки тестовых условий и тестовых сценариев;
- проектирование и установка тестового окружения и выявление необходимой инфраструктуры и инструментов;
- создание двунаправленной трассируемости между тестовым базисом и тестовым сценарием.

Результат тест-анализа – точные метрики и цели, к которым стремится процесс тестирования. Они входят в выходные критерии тестирования, должны быть неразрывно связаны с базисом тестирования и стратегическими целями проекта и служат основой для тест-дизайна.

Тест-дизайн (разработка тестовых сценариев) включает в себя:

- составление пошаговой инструкции по выполнению тестовых сценариев;
- составление критериев успешности прохождения теста.

Тест-анализ отвечает на вопрос «Что тестировать?», а тест-дизайн отвечает на вопрос «Как тестировать?».

. . . .

Выполнение тестирования *– это деятельность, где процедуры тестирования или автоматизированные сценарии задаются последовательностью тестовых сценариев, собирается любая информация, необходимая для выполнения тестов, разворачивается окружающая среда и запускаются тесты.*

. . . .

3. Подготовка тестовой среды:

- составление набора входных данных;
- развертывание тестовой среды и т. д.

4. Выполнение тестов:

- ввод входных значений тестирования;
- анализ выходных значений;
- выполнение тестовых сценариев и т. д.

5. Создание отчетов:

- анализ успешности прохождения тестирования;
- описание найденных дефектов и т. д.

6. Действия по завершению тестирования:

- свертывание тестовой среды;
- очистка базы данных от тестовых значений и т. д.

Действия по завершению тестирования собирают данные о завершенных испытаниях для объединения опыта, тестового обеспечения, фактов и цифр. Действия по завершению тестирования происходят на тех этапах проекта, когда система программного обеспечения выпущена, тестирование завершено (или прервано), этап завершен или релиз по сопровождению был закончен. Для действий по завершению тестирования поставлены следующие основные задачи:

- проверка того, достигнуты ли запланированные результаты;
- закрытие отчетов об инцидентах или внесение изменений в записи по каждому из открытых инциденто;
- документирование приемки системы;
- использование собранной информации для повышения зрелости процесса тестирования.

3.3 Методы проектирования тестов

Рассмотрим методы проектирования тестов. Все их можно условно свести к трем группам:

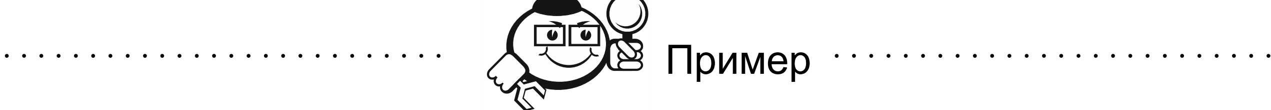
1. На основе опыта

Исследовательское тестирование (*Exploratory testing*). Данный вид тестирования заключается в том, что инженер по качеству изучает/исследует программное обеспечение точно так же, как этот делал бы обычный пользователь, с поправкой на опыт тестировщика. Подход, который предполагает одновременное выполнение и разработку тестов, а также изучение продукта. Опытный тестировщик-исследователь записывает идеи тестов и успешно применяет их к последующим циклам испытаний.

Интуитивное (*ad-hoc testing*) – вид тестирования, который выполняется без подготовки к тестам, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Оно не требует никакой документации, планирования, процессов, которых следует придерживаться при выполнении. Также на данный вид тестирования не пишутся тест-кейсы, что в свою очередь может вызвать определенные затруднения в попытках воспроизвести дефект в системе. Часто интуитивное тестирова-

ние путают с исследовательским. Если говорить об интуитивном и исследовательском тестировании, то Ad-hoc testing – это более интуитивное и беспорядочное тестирование, когда тестировщик просто идет и проверяет, что ему хочется. У него нет определенной цели, структуры тестов в голове, какой-то системы. В свою очередь исследовательское тестирование более структурированное. Обычно тестировщик знает, что ему нужно проверить, у него есть цель и какая-то система проведения тестов, хотя тесты в этом случае не обязательно должны быть оформлены в виде тест-кейсов.

Предугадывание ошибки (*error guessing*, *EG*) – метод проектирования тестов, когда опыт тестировщика используется для предугадывания того, какие дефекты могут быть в тестируемом компоненте или системе в результате сделанных ошибок, а также для разработки тестов специально для их выявления. Подход к этой технологии состоит в составлении списка возможных дефектов и багов для такого типа ПО, а также в разработке тестов для их проверки. Конечно, успешность применения этой технологии напрямую зависит от опытности аналитиков в команде.



Допустим, перед тестировщиком стоит задача тестирования регистрации на интернет-сайте.

1. Вводим корректные данные в графах электронной почты и домашнего адреса, т. к. они являются самыми главными для службы доставки (рис. 3.6).

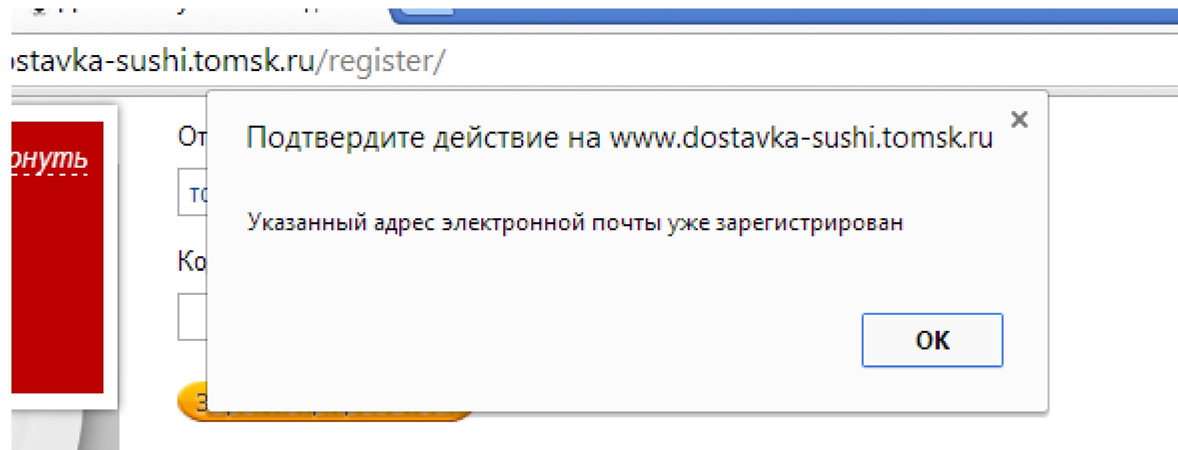


Рис. 3.6 – Тестирование формы регистрации

Пользователь не получает подтверждение регистрации, но при этом после повторного нажатия на кнопку «Зарегистрироваться» выдается сообщение о том,

что учетная запись с данным e-mail уже зарегистрирована. Это дефект, т. к. пользователю никак не было сообщено, что регистрация прошла успешно.

2. Вводим некорректные данные (негативное тестирование) (рис. 3.7).

Домашний телефон*
12633726472364
Мобильный телефон*
23623647326472
Адрес*
уганарнпаврпар 32746 234
Электронная почта*
dsjhfusdf
Пароль*

Рис. 3.7 – Негативное тестирование формы регистрации

Жмем кнопку «Зарегистрироваться». Система абсолютно корректно среагировала на неправильный адрес электронной почты, выдав пользователю соответствующее сообщение (рис. 3.8).

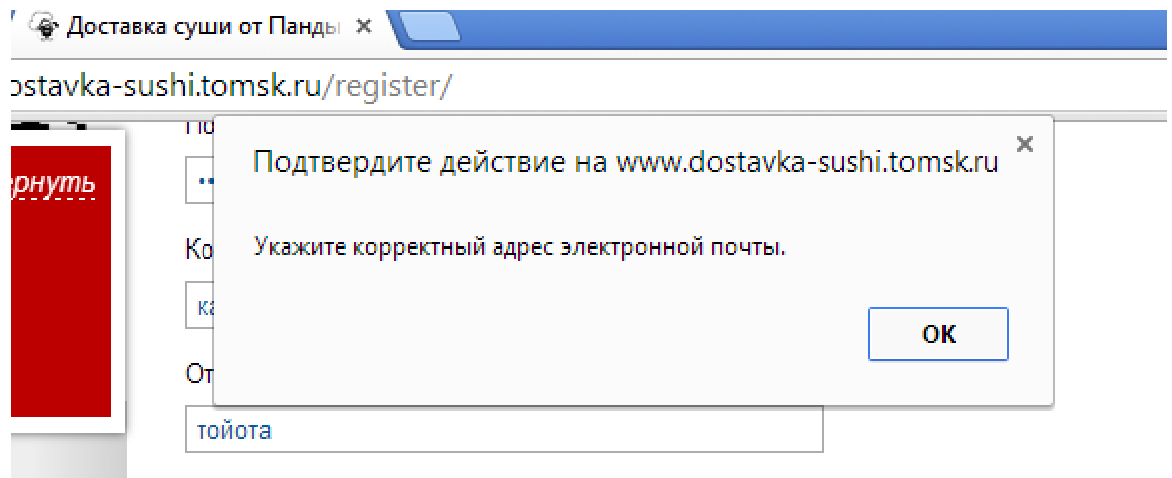


Рис. 3.8 – Реакция на неправильный адрес электронной почты

Таким образом, исследуя систему, специалист по тестированию находит дефекты.

. . . .

Разновидностью метода предугадывания ошибки является атака на недочет.

Атака на недочет (*fault attack*). Направленная и нацеленная попытка оценить качество, главным образом надежность, объекта тестирования за счет попыток вызвать определенные дефекты.

Например, исходя из опыта, можно сказать, что многие сайты не адаптированы под маленькие мониторы, т. е. если открыть сайт на маленьком экране (например, планшете), то не будет полосы прокрутки. Чтобы убедиться, открываем сайт и действительно видим, что полосы прокрутки нет (рис. 3.9).

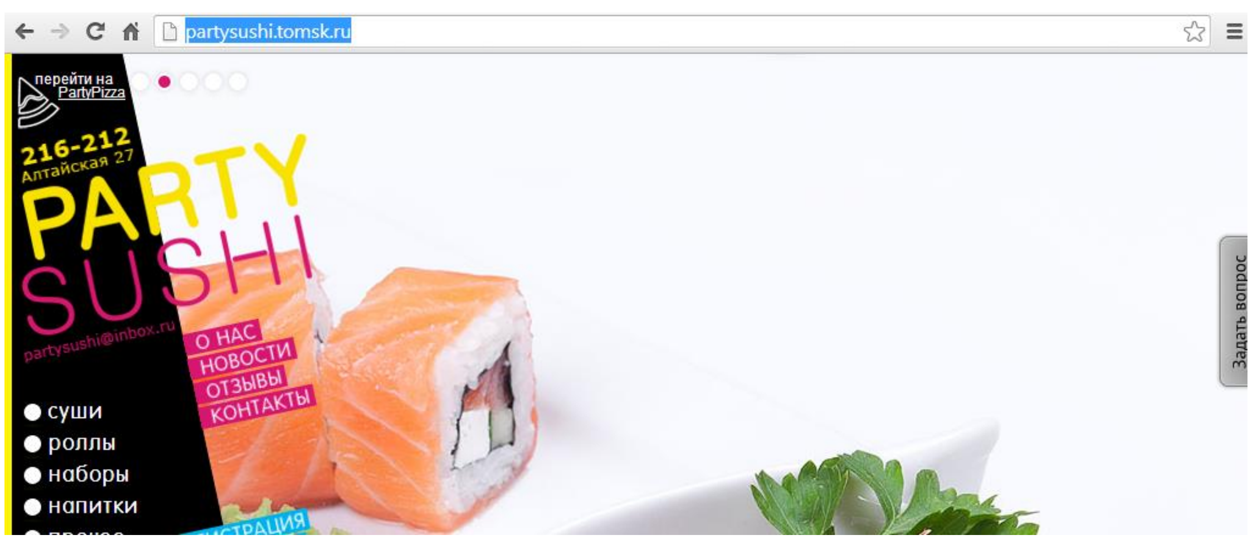


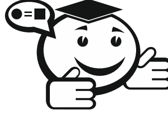
Рис. 3.9 – Дефект графического интерфейса

Таким образом, основываясь на опыте, можно находить дефекты быстрее и эффективнее.

Общие признаки методов на основе опыта:

- для определения тестовых сценариев используются человеческие знания и опыт;
- знания тестировщиков, разработчиков, пользователей и заинтересованных лиц о программном продукте, его использовании и окружении являются одним из источников информации;
- знания о вероятных дефектах и их распределении являются другим источником информации.

2. На основе спецификации (стратегия черного ящика)



Метод черного ящика (*black box testing, closed box testing, specification-based testing*) *– у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.*

Разработка тестов методом черного ящика (*black box test design technique*) – процедура создания и/или выбора тестовых сценариев, основанная на анализе функциональной или нефункциональной спецификации компонента или системы без знания внутренней структуры.

Техники черного ящика:

- эквивалентное разбиение;
- анализ граничных значений;
- попарное тестирование;
- таблицы решений;
- сценарии использования.

Эквивалентное разбиение (*equivalence partitioning*). Данная техника основана на том принципе, что входные данные разбиваются на подмножества, атрибуты которого обладают схожими свойствами и тестируются по схожим сценариям. Как правило, тестовые сценарии разрабатываются для покрытия каждой области как минимум один раз.

При использовании этой техники тестировщик должен помнить о том, что

- слишком большое количество эквивалентных классов увеличивает вероятность того, что множество тестов будет лишним (избыточным);
- слишком малое число эквивалентных классов увеличивает вероятность того, что ошибки продукта будут пропущены.

Поэтому стоит вопрос: какие тесты считать эквивалентными? Два теста считаются эквивалентными, если

- они тестируют одну и ту же функцию, модуль, часть системы;
- один из тестов находит ошибку, а другой скорее всего тоже ее поймает;
- один из них не находит ошибку, а другой скорее всего тоже не поймает.

Примерный алгоритм использования:

1. *Определить классы эквивалентности*. Это главный шаг техники. От него во многом зависит эффективность ее применения.
2. *Выбрать одного представителя от каждого класса*. На этом шаге из каждого эквивалентного набора тестов выбирается один тест.
3. *Выполнить тесты*. На этом шаге выполняются тесты от каждого класса эквивалентности.

Пример

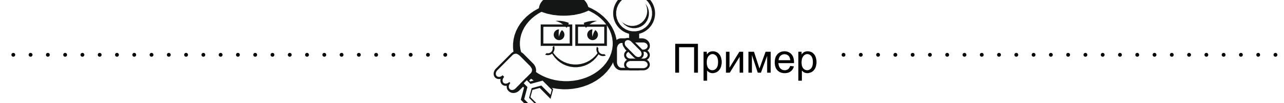
Если нам необходимо проверить работу калькулятора на всевозможных входных данных, совсем необязательно перебором проверять все значения. Достаточно разбить входные значения на 3 части: отрицательные значения, ноль, положительные значения (в зависимости от задачи), а затем выбрать из каждого подмножества (части) по несколько значений и проверить работу только на них.

Анализ граничных значений (*boundary value analysis*). Разработка тестов методом черного ящика, при котором тестовые сценарии проектируются на основании граничных значений. Статистика показывает, что зачастую большое количество дефектов скапливается в граничных значениях множества входных данных, поэтому именно граничные значения следует проверять особенно тщательно. *Граничное значение* (*boundary value*) – входное значение или выходные данные, которые находятся на грани эквивалентной области или на наименьшем расстоянии от обеих сторон грани, например, минимальное или максимальное значение области.

Анализ граничных значений может применяться на всех уровнях тестирования. Если техника анализа классов эквивалентности ориентирована на тестовое покрытие, то эта техника основана на рисках. Эта техника начинается с идеи о том, что программа может сломаться в области граничных значений.

Примерный алгоритм использования техники анализа граничных значений:

1. Выделить классы эквивалентности. Это очень важный шаг, и от правильности разбиения на классы эквивалентности зависит эффективность тестов граничных значений.
2. Определить граничные значения этих классов.
3. Определить, к какому классу будет относиться каждая граница.



Например, есть функция, вычисляющая сумму чисел *a* и *b*:

```
byte sum(int a, int b)
{
        return (byte) a + b;
}
```

Если передадим значения $a = -129$ и $b = 128$, то в случае отсутствия специальной обработки ситуации переполнения, сумма будет вычислена неверно, т. к. наименьший по размеру целочисленный тип *byte* имеет диапазон допустимых значений (технологическую границу) от -128 до 127.

Проанализируем приграничные значения. При вводе значений 128 и 1 результатом будет -127. При вводе значений -128 и -1 результатом будет 127. Результат неверный.

Тестирование по сценариям использования (*use case testing*). Разработка тестов методом черного ящика, при котором тестовые сценарии создаются для выполнения сценариев использования. Данный вид тестирования возможен, если аналитиками был составлен документ с перечнем вариантов использования (пошаговые инструкции возможных операций с системой).

Попарное тестирование (*pairwise testing*) – это современная и эффективная методика тестирования, основанная на том предположении, что большинство дефектов возникает при взаимодействии не более двух факторов. Тестовые наборы, генерируемые при использовании данной методики, охватывают все уникальные пары комбинаций факторов, что считается достаточным для обнаружения большего числа дефектов.

Для попарного тестирования используются алгоритмы, основанные на построении ортогональных массивов или на алгоритме всех пар, которые опираются на теоретические исследования в области комбинаторных алгоритмов, алгоритмов дискретной математики.

Алгоритм всех пар (*All-Pairs Algorithm*) – это комбинаторная методика, которая была специально создана для попарного тестирования. В ее основе лежит выбор возможных комбинаций значений всех переменных, в которых содер-

жатся все возможные значения для каждой пары переменных. Исходя из определения при этом будет получено меньшее число комбинаций, чем при использовании ортогональных массивов.

Для тестирования с использованием алгоритма всех пар выполняют следующие шаги:

1. Как и для ортогональных массивов, определяют таблицу всех переменных и их значений.
2. Оставляют в таблице только все возможные уникальные комбинации пар значений переменных.

Согласно данным сайта pairwise.org, есть множество программ для получения пар, есть и онлайновые сервисы: hexawise, testcover.

Каковы преимущества попарного тестирования?

- Данный тип проверки уменьшает количество тест-кейсов, необходимых для проверки продукта.
- Таким образом, попарное тестирование ускоряет выполнение самого процесса контроля качества продукта.
- Для проведения этого типа проверки требуется на 50 процентов меньше сил в сравнении с другими тактиками выполнения функционального тестирования.
- Практика показывает, что количество багов, обнаруженных с помощью попарного тестирования, будет больше, чем при проверке всех значений для каждого параметра ввода.

Давайте рассмотрим пример применения попарного тестирования.

Проанализируем параметры для сортировки приставок, которые часто используются в интернет-магазинах:

- Цена от 1 490 до 35 190 руб.
- Бренды: DeepSilver, DENDY, MadCatz, Microsoft, New Game, Nintendo, SEGA, Sony.
- Платформа: Sony Playstation 4, Microsoft Xbox One, другие консоли.
- Объем жесткого диска (ОБЖ) от 16 до 1024 ГБ.
- Производитель видеочипа: AMD, ATI, NVIDIA.
- Оптический привод: Blu-ray, нет.
- Беспроводная связь: Bluetooth, Wi-Fi, нет, радиоканал 2.4 ГГц.

Не рекомендуется включать негативные тесты в таблицу составления пар. Негативными тестами проверяют реакцию системы на некорректный ввод (для этого все-таки предназначены негативные функциональные тесты), и по сути в каждом таком кейсе проверяется только один параметр – некорректный. А попарное тестирование должно применяться в случае необходимости проверки взаимодействующих значений и реакции системы на это взаимодействие. Так как с негативным вводом система и остальные данные взаимодействовать не должны (это выявляется ранее проведенными функциональные тестами), то и включение этих значений для составления пар не имеет смысла. Также попарное тестирование не призвано тестировать и граничные значения, т. к. для этого должны быть отдельные тесты.

Поэтому в выборку войдут только валидные значения. Например:

- Цена: 15 000 руб.
- ОБЖ: 64 ГБ.

Входные значения запишем в текстовый документ (рис. 3.10).

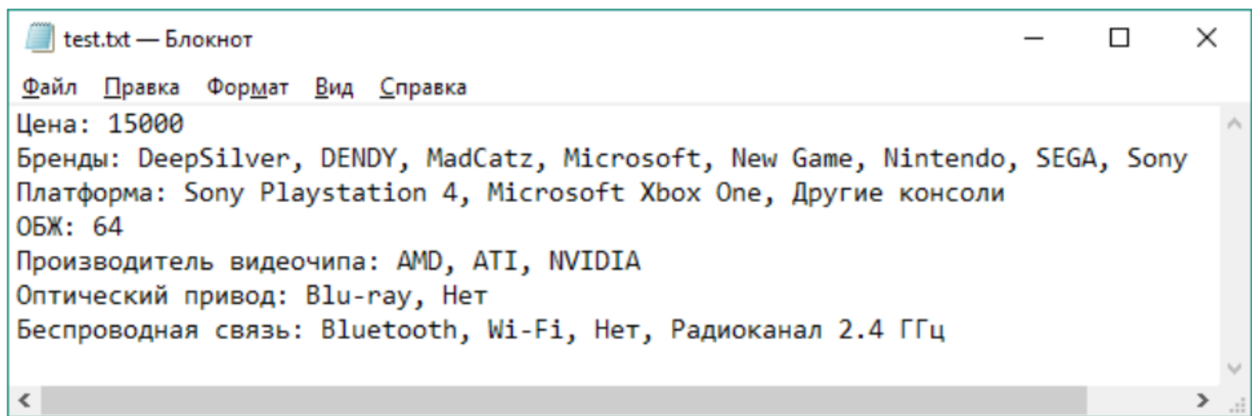
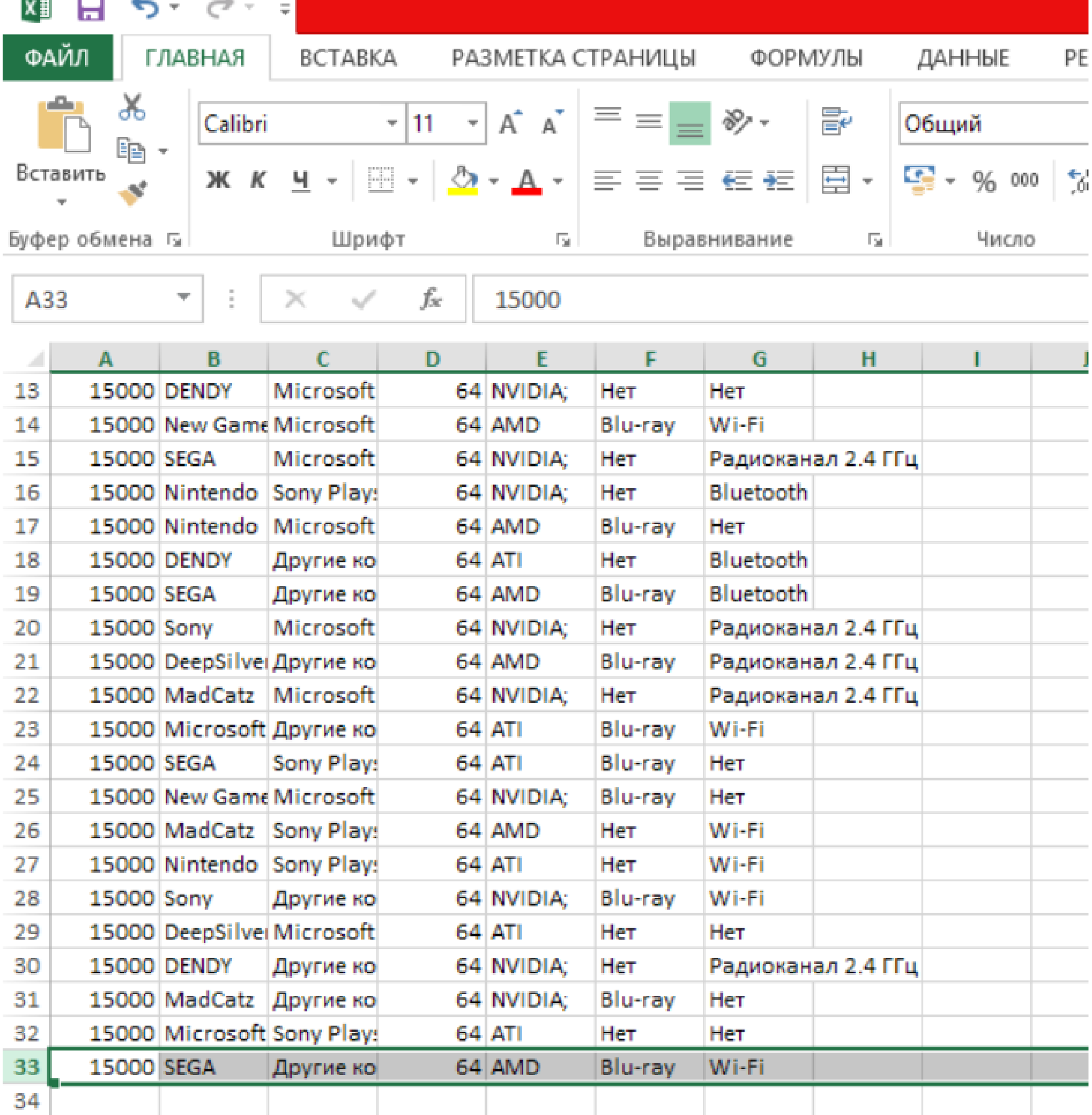


Рис. 3.10 – Текстовый документ с входными значениями

Передав этот файл PICT (свободный инструмент для попарного тестирования, разработанный Microsoft), сгенерируем и запишем возможные пары в Excel-файл (рис. 3.11).



	A	B	C	D	E	F	G
13	15000	DENDY	Microsoft	64	NVIDIA;	Нет	Нет
14	15000	New Game	Microsoft	64	AMD	Blu-ray	Wi-Fi
15	15000	SEGA	Microsoft	64	NVIDIA;	Нет	Радиоканал 2.4 ГГц
16	15000	Nintendo	Sony Play:	64	NVIDIA;	Нет	Bluetooth
17	15000	Nintendo	Microsoft	64	AMD	Blu-ray	Нет
18	15000	DENDY	Другие ко	64	ATI	Нет	Bluetooth
19	15000	SEGA	Другие ко	64	AMD	Blu-ray	Bluetooth
20	15000	Sony	Microsoft	64	NVIDIA;	Нет	Радиоканал 2.4 ГГц
21	15000	DeepSilver	Другие ко	64	AMD	Blu-ray	Радиоканал 2.4 ГГц
22	15000	MadCatz	Microsoft	64	NVIDIA;	Нет	Радиоканал 2.4 ГГц
23	15000	Microsoft	Другие ко	64	ATI	Blu-ray	Wi-Fi
24	15000	SEGA	Sony Play:	64	ATI	Blu-ray	Нет
25	15000	New Game	Microsoft	64	NVIDIA;	Blu-ray	Нет
26	15000	MadCatz	Sony Play:	64	AMD	Нет	Wi-Fi
27	15000	Nintendo	Sony Play:	64	ATI	Нет	Wi-Fi
28	15000	Sony	Другие ко	64	NVIDIA;	Blu-ray	Wi-Fi
29	15000	DeepSilver	Microsoft	64	ATI	Нет	Нет
30	15000	DENDY	Другие ко	64	NVIDIA;	Нет	Радиоканал 2.4 ГГц
31	15000	MadCatz	Другие ко	64	NVIDIA;	Blu-ray	Нет
32	15000	Microsoft	Sony Play:	64	ATI	Нет	Нет
33	15000	SEGA	Другие ко	64	AMD	Blu-ray	Wi-Fi
34							

Рис. 3.11 – Таблица тестов, сгенерированная PICT

Как видим, для полного тестирования всех параметров необходимо провести 33 теста.

. . . .

Тестирование таблицы решений (*decision table testing*). Разработка тестов методом черного ящика, при котором тестовые сценарии проектируются для проверки комбинаций входных данных и/или причин, отраженных в таблице решений. *Таблица решений* (*decision table*) – таблица, отражающая комбинации входных данных и/или причин с соответствующими выходными данными и/или

действиям (следствиям), которая может быть использована для проектирования тестовых сценариев.

Таблицы решений – хороший метод для сбора системных требований, содержащих логические условия, и документирования внутреннего дизайна системы. Они могут использоваться для записи сложных бизнес-правил, которые должна реализовывать система. Анализируются спецификации и определяются условия и действия системы. Входные условия и действия чаще всего формулируются таким образом, чтобы они могли принимать логические значения «истина» или «ложь». Этот метод может быть применен ко всем ситуациям, в которых действие программного продукта зависит от нескольких логических альтернатив.

Таблица принятия решений, как правило, разделяется на 4 квадранта:

Условия	Варианты выполнения действий
Действия	Необходимость действий

Условия – список возможных условий.

Варианты выполнения действий – комбинация из выполнения и/или невыполнения условий этого списка.

Действия – список возможных действий.

Необходимость действий – указание на то, надо или не надо выполнять соответствующее действие для каждой из комбинаций условий.

Рассмотрим *таблицу принятия решений* на примере.

Работникам компании выплачиваются бонусы, если они проработали больше года и достигли согласованных целей. Следующая таблица решений была разработана для тестирования системы (табл. 3.1).

Таблица 3.1 – Таблица решений для расчета бонусов

		T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
Условия									
Усл1	Работает больше 1 года?	YES	NO	YES	NO	YES	NO	YES	NO

		T1	T2	T3	T4	T5	T6	T7	T8
Условия									
Усл2	Согласована цель?	NO	NO	YES	YES	NO	NO	YES	YES
Усл3	Достигнута цель?	NO	NO	NO	NO	YES	YES	YES	YES
Действие									
	Выплата бонуса?	NO	NO	NO	NO	NO	NO	YES	NO

Какие тесты могут быть удалены, т. к. не смогут происходить в реальной жизни?

1. T1 и T2.
2. T3 и T4.
3. T5 и T6.
4. T7 и T8.

T5 и T6 могут быть удалены, так как цель не может быть достигнута как несогласованная.

. . . .

Каковы преимущества тестирования таблицы решений?

- Каждая вертикальная колонка («правило») является схематическим изображением тест-кейса, где «условие» определяет параметры входных данных, а «действие» – ожидаемый результат.
- С помощью таблицы решений можно установить одно и больше действий для каждого правила.
- Специалист может объединять принципы тестирования таблицы решений и тестирования граничных значений.
- Даже если система выполняет сложный комплекс действий, данный тип проверки все равно можно использовать.

3. На основе структуры (стратегия белого ящика)

. . . .

Метод белого ящика (*white box testing, open box testing, clear box testing, glass box testing*) – *тестирование, при котором у тестировщика есть доступ к внутренней структуре и коду приложения;*

основано на анализе внутренней структуры компонента или системы.

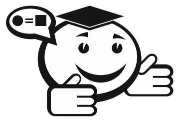
. . . .

Техники белого ящика:

- тестирование операторов (*statement testing*) – это тестирование, направленное на то, чтобы тестовой сценарий прошел через каждый оператор кода программы;
- тестирование альтернатив (*decision testing*) – это тестирование, направленное на то, чтобы тестовой сценарий прошел через каждую альтернативу кода программы.

Оценить качество тестирования на основе структуры можно посредством подсчета покрытия кода.

. . . .



Покрытие (*coverage*) *– уровень, выражаемый в процентах, на который определенный элемент покрытия был проверен набором тестов.*

. . . .

Покрытие альтернатив (*decision coverage*) – процент результатов альтернативы, который был проверен набором тестов. Стопроцентное покрытие решений подразумевает стопроцентное покрытие ветвей и операторов.

Покрытие операторов (*statement coverage*) – процентное отношение операторов, исполняемых набором тестов, к их общему количеству.

Покрытие альтернатив, связанное с *тестированием ветвей*, – это доля результатов альтернатив (например, вариантов «Истина» и «Ложь» для оператора «Если»), проверенных набором сценариев тестирования. В методе тестирования альтернатив тестовые сценарии создаются для выполнения определенных результатов альтернатив. Ветви исходят из точек альтернатив в программном коде и показывают передачу управления различным участкам кода. Покрытие альтернатив определяется отношением числа всех результатов альтернатив, покрытых разработанными или выполненными тестовыми сценариями, к числу всех возможных результатов альтернатив в тестируемом коде. Покрытие альтернатив более строгое, чем покрытие операторов: 100%-ное покрытие альтернатив обеспечивает 100%-ное покрытие операторов, но не наоборот.

Граф потока управления представлен на рисунке 3.12. Следующие 3 теста были выполнены следующим образом:

1. Тест A покрывает путь: A, B, D, E, G.
2. Тест B покрывает путь: A, B, D, E, F, G.
3. Тест C покрывает путь: A, C, F, C, F, C, F, G.

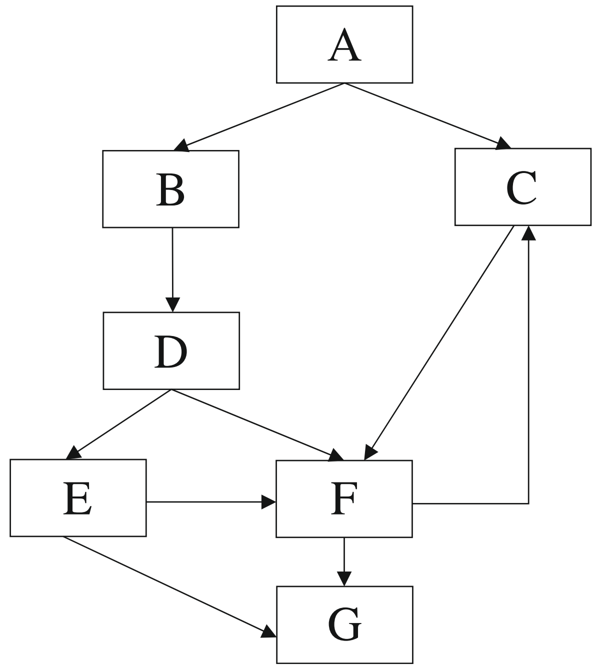


Рис. 3.12 – Граф потока управления

На диаграмме есть 4 условия: A, D, E, F. Тест A покрывает $A \to B$, $D \to E$ и $E \to G$. Тест B покрывает $A \to B$, $D \to E$, $E \to F$ и $F \to G$. Тест C покрывает $A \to C$, $F \to C$ и $F \to G$. Поэтому условие A покрыто ($A \to B$ и $A \to C$), условие E покрыто ($E \to G$ и $E \to F$), условие F покрыто ($F \to C$ и $F \to G$). Условие D не покрыто, покрыто только $D \to E$, а $D \to F$ не покрыто. Как видим, условие D не было протестировано полностью.

Для подсчета покрытия кода существуют различные программные средства, многие интегрированные системы имеют встроенные анализаторы покрытия кода тестами, например:

1) Java:
   - Jcov;
   - *EclEmma – Java Code Coverage for Eclipse*;

   - Cobertura;
2) C++:
   - Gcov;
   - Tcov;
3) C#:
   - OpenCover;
4) встроенный в Visual Studio Test Coverage.

Общие признаки подходов, основанных на структуре:

- тестовые сценарии выводятся на основе информации о том, как спроектировано программное обеспечение (например, на основе программного кода и подробного описания проектного решения);
- для программного обеспечения может быть измерена величина покрытия для имеющихся тестовых сценариев, и последующие тестовые сценарии могут разрабатываться для систематического увеличения покрытия.

3.4 Тестовая документация

Очевидно, что процесс тестирования ПО должен быть документированным. Выявленные в ходе тестирования дефекты также должны быть полностью описаны и документированы.

Хорошая документация обладает тремя важнейшими преимуществами:

- облегчает тестирование;
- помогает организовать взаимодействие между персоналом;
- представляет собой удобную структуру для организации, планирования и управления тестовым проектом.

Существует международный стандарт написания тестовой документации – IEEE 829. IEEE (Institute of Electrical and Electronic Engineers) – организация, созданная в США в 1963 г., которая является разработчиком стандартов для локальных вычислительных систем, в том числе по кабельной системе, физической топологии и методам доступа к среде передачи данных. Стандарт также определяет форму и содержание тестовых документов. Этот стандарт разрабатывался с 1977 г. и был утвержден в 1983 г., а затем вновь подтвержден в 1991, 1998, 2008 гг. Несмотря на свою зрелость, он до сих пор актуален. Данный стандарт содержит в себе разделы информации по всем необходимым документам процесса тестирования.

Наиболее распространенный комплект документации по тестированию выглядит следующим образом:

- тест-план (test plan);
- тест-кейсы (test cases);
- чек-лист (check-list);
- отчет о тестировании (test report);
- отчет о дефектах (bug report);
- программа и методика испытаний (ПИМ).

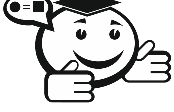
Теперь более детально рассмотрим тест-кейсы и чек-листы, так как они являются основными документами тестировщика на каждый день.

3.4.1 Тест-кейсы

Во главе всего тестирования лежит термин «тест».

Тест (***test***) *– набор из одного или нескольких тестовых сценариев.*

Любой тестировщик проводит тестирование того или иного продукта, руководствуясь специальной профессиональной документацией – *тест-кейсами* (*test case*). Тест-кейсы в свою очередь составляют *тест-комплекты* (*test suite*).



Тестовый сценарий*, или* ***тестовая ситуация****,* ***тест-кейс*** (***test case***) *– набор входных значений, предусловий выполнения, ожидаемых результатов и постусловий выполнения, разработанный для определенной цели или тестового условия, таких как выполнение определенного пути программы, либо для проверки соответствия определенному требованию* (*IEEE 610*).

Тест-кейс – это самая маленькая часть тест-документации, это ситуация, которая проверяет конкретно взятое условие из требований. Это как инструкция из «Икеи», все должно быть расписано очень подробно.

Данный документ – самый важный в процессе выполнения тестов, и его следует составлять для каждого проекта:

- Не тратится время на «вспоминание» и формулировку шагов теста, можно просто следовать заранее написанной инструкции.

- Тест-кейсы являются тем документом, который удобно продемонстрировать заказчику, чтобы показать, что именно тестировалось и каким образом.
- Тест-кейсы облегчают ввод в процесс новых специалистов, ранее не знакомых с тестируемой системой.
- Тест-кейсы служат хорошей обучающей базой для неопытных специалистов по тестированию.
- Наличие тест-кейсов значительно ускоряет регрессионное тестирование.

Недостатки тест-кейсов:

1. Очень много копирования. Тест-кейсы очень похожи друг на друга, первые шаги одинаковые.
2. Сложно поддерживать. Чтобы актуализировать тест-кейсы, надо внести изменения в сотни сценариев, поэтому тест-кейсы должны быть независимы.
3. Неактуальное состояние. Тест-кейсы копируются друг с друга, и часто в них остаются неактуальные части из исходного кейса, которые забыли изменить.

При написании тест-кейсов следует придерживаться основных правил:

- Начинайте с коротких тест-кейсов.
- Перед написанием детализированных тест-кейсов запишите все, что можно протестировать в вашем приложении в вольной форме.
- Используйте активный залог («open», «paste», «click»). В русском языке используйте безличную форму: «открыть» (вместо «откройте»).
- Описывайте поведение системы: «появляется окно», «приложение закрывается».
- Используйте простой технический стиль.
- Обязательно указывайте точные названия всех элементов приложения.
- Не объясняйте базовые понятия работы с операционной системой.
- Избегайте лишней информации в тест-кейсах, все должно быть коротко и ясно.
- Ни в коем случае нельзя связывать тест-кейсы между собой, так как это создаст дополнительные неудобства при роботе с кейсом. Тест-кейс, к которому вы сделали привязку, могут удалить из базы данных или провести в нем коррективы, т. к. это независимый документ.

Тест-кейс должен обязательно содержать хотя бы ожидаемый результат (даже без описания действий, которые к нему ведут). Кроме ожидаемого результата необходимо пошаговое описание действий, которые позволят прийти к фактическому результату и сравнить его с ожидаемым. Краткое описание тест-кейса имеет смысл вынести в заголовок.

Что должно быть в тест-кейсе?

Приведенный список – это лишь *рекомендация*, каждый тестировщик пишет так, как ему удобнее или как это принято в IT-компании.

Уникальное краткое название (*ID*). В это поле записывается номер кейса или номер вместе с определенной аббревиатурой, к примеру «1-П» служит для уникальной идентификации среди других кейсов.

Предусловие (*Pre Conditions*). Список действий или критерии, которые приводят систему к состоянию, пригодному для проведения основной проверки. Сюда можно записать и предварительные шаги, что не относится к самому тесту. На них можно ссылаться и из других тестов, но сам тест-кейс должен быть независим.

Описание (*Summary*). Это краткое описание проблемы. Описание должно содержать ответ на вопрос, что произошло и при каких условиях работает неверно.

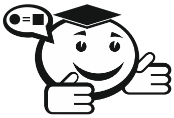
Шаги (*степы*) (*Steps*). Здесь описывают шаги, для того чтобы воспроизвести баг. Степы рекомендуют максимально сокращать, то есть найти кратчайший путь для воспроизведения бага. Очень важно, чтобы они оставались максимально понятными для разработчиков.

Ожидаемый результат (*Expected Result*). В этом поле описывают ожидаемый результат после выполнения всех шагов или, возможно, после конкретных шагов, что бывает реже. Ожидаемые результаты можно группировать.

Статус (*Pass/Fail*). Поле служит для проставления статуса каждого тест-кейса. Если ожидаемый результат совпадает с реальным, то проставляют *pass* (прошел), в противном случае – *fail* (ошибка). Возможно еще несколько статусов в зависимости от процессов и правил в IT-компании.

Постусловие (*Post Conditions*). Список действий, переводящих систему в первоначальное состояние. Не является обязательной частью. Это скорее правило хорошего тона. Использование постусловия особенно актуально при автоматизированном тестировании, когда за один прогон можно наполнить базу данных сотней или даже тысячей некорректных документов.

Тест-кейсы бывают *позитивные* и *негативные*.



Позитивный тест-кейс *использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.*

Негативный тест-кейс *оперирует как корректными, так и некорректными данными (минимум хотя бы один некорректный параметр) и ставит целью проверку исключительных ситуаций.*

Запомните, что сначала проводят позитивное тестирование, а только потом негативное! Ведь на тестирование всегда не хватает времени, и поэтому в первую очередь необходимо проверить корректную работу ПО, а лишь потом провести обработку исключений.

Пример

П-1

«Проверка того, что программа умеет показывать файлы формата PNG».

Шаг 1. Нажать кнопку «Выбрать файл».

Шаг 2. Выбрать файл с расширением PNG.

Шаг 3. Нажать кнопку «Открыть».

Ожидаемый результат: содержимое файла показано в графическом виде, в полноэкранном режиме.

Здесь сравнение ожидаемого результата и фактического осуществить довольно просто, и критерий соответствия не нужен.

Рассмотрим позитивный тест-кейс (табл. 3.2).

Таблица 3.2 – Позитивный тест-кейс

ID	**Название сценария**	**Шаги**	**Ожидаемый результат**
П-2	Переключение на русскоязычную версию сайта	1. Открыть сайт http://epayservices.com/	1. Открылась главная страница сайта http://epayservices.com/
		2. Кликнуть на меню смены языка	2. Выпал список для смены локализации страницы

ID	Название сценария	Шаги	Ожидаемый результат
		3. Выбрать в выпадающем списке элемент RU и нажать	3. По нажатию на элемент списка RU загрузилась русскоязычная версия сайта http://epayservices.com/ru/index.html

Рассмотрим негативный тест-кейс (табл. 3.3).

Таблица 3.3 – Негативный тест-кейс

ID	Название сценария	Шаги	Ожидаемый результат
Н-3	Проверка имени на кириллице в регистрационной форме	1. Открыть сайт	1. Открылся сайт
		2. Перейти по ссылке Sign Up	2. Открылась страница регистрации нового пользователя
		3. Перейти по ссылке New user registration	3. После загрузки регистрационной формы по ссылке New user registration заполнили поле First Name «Линда»
		4. Ввести в поле First Name «Линда»	4. После активации кнопки Register появилась надпись под полем First Name: may consist of: Latin letters, spaces
		5. Активировать кнопку Register	

Проверим обработку ошибки (рис. 3.13).

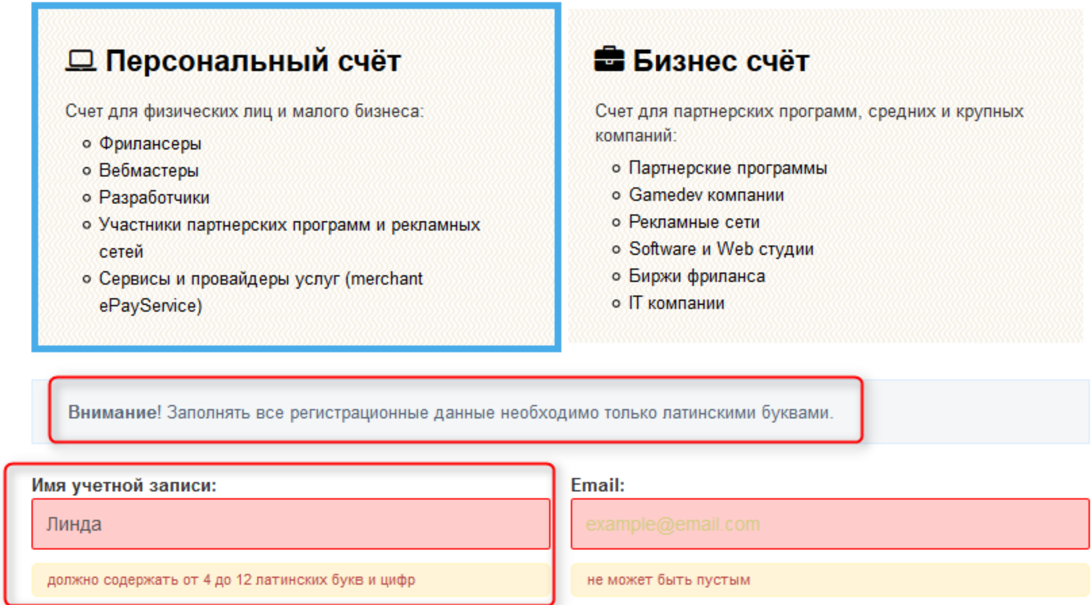


Рис. 3.13 – Выполнение негативного тест-кейса Н-3

. . . .

Существует несколько специальных инструментов для написания тест-кейсов, среди них:

1. Testlink – инструмент управления тест-кейсами (test case management), распространяемый по лицензии GPL (устаревший и неудобный интерфейс, непродуманная система ссылок на конкретные кейсы или сьюты. Плюс: бесплатный.
2. TestRail – программное обеспечение для управления данными, полученными в результате тестирования. Данный инструмент помогает отслеживать процессы, управлять программным обеспечением и организовывать команду. Можно создавать тест-кейсы, управлять тестовыми наборами и координировать весь процесс тестирования программного обеспечения. TestRail предоставляет возможность повысить производительность и получить полный обзор хода процесса тестирования. Минус: платный.

В 2016 г. на форуме Software-Testing.Ru [14] был проведен опрос среди профессиональных тестировщиков по поводу использования инструментов для

работы с тест-кейсами. На рисунке 3.14 представлены результаты опроса. На первом месте оказались программные продукты *MS Excel* и *MS Word*, которые не являются специализированными системами для работы с тест-кейсами.

Популярность инструментов

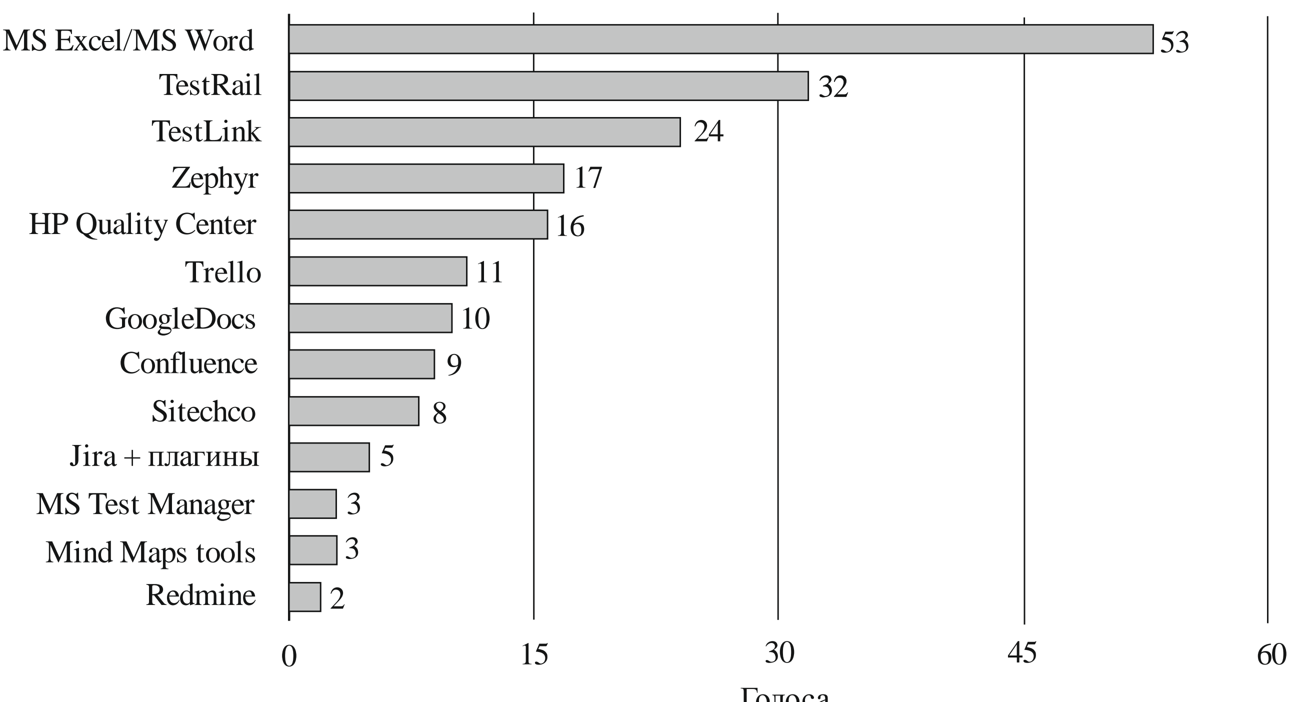


Рис. 3.14 – Рейтинг популярности инструментов
для работы с тест-кейсами

3.4.2 Чек-листы

Чек-листы – один из фундаментальных инструментов тестирования. Они позволяют не забывать о важных тестах, фиксировать результаты своей работы и отслеживать статистику о статусе программного продукта. Данный документ очень удобно иметь в своем арсенале.

. . . .

Чек-лист (**check-list**) *– набор идей тестов, описывающий, что должно быть протестировано.*

. . . .

Чек-лист отличается от тест-кейса степенью подробности, в нем вы не встретите подробных шагов. Для использования чек-листа при тестировании очень много информации нужно держать в голове в момент прогона тестов и знать логику работы приложения на отлично. Главный принцип чек-листов за-

ключается в том, что каждый тестировщик по-своему проходит их, расширяя тестовый набор на основе своего опыта, поэтому начинающему тестировщику сложно начать работать с чек-листами.

Чек-лист чаще всего представляет собой обычный и привычный нам список. Он содержит информацию о компонентах, которые являются общими для целого класса проектов, например, чек-лист для веб-проектов или мобильных приложений.

Для того чтобы чек-лист был действительно полезным инструментом, он должен обладать рядом важных свойств.

Логичность. Чек-лист пишется не просто так, а на основе целей и для того, чтобы помочь в достижении этих целей. К сожалению, одной из самых частых и опасных ошибок при составлении чек-листа является превращение его в свалку мыслей, которые никак не связаны друг с другом.

Последовательность и структурированность. Со структурированностью все достаточно просто – она достигается за счет оформления чек-листа в виде многоуровневого списка. Что до последовательности, то даже в том случае, когда пункты чек-листа не описывают цепочку действий, человеку все равно удобнее воспринимать информацию в виде неких небольших групп идей, переход между которыми является понятным и очевидным (например, сначала можно прописать идеи простых позитивных тест-кейсов, потом идеи простых негативных тест-кейсов, потом постепенно повышать сложность тест-кейсов, но не стоит писать эти идеи вперемешку).

Полнота и неизбыточность. Чек-лист должен представлять собой аккуратную «сухую выжимку» идей, в которых нет дублирования (часто появляется из-за разных формулировок одной и той же идеи), и в то же время ничто важное не упущено.

Пример 1

Чек-лист с результатами и комментариями тестирования можно структурировать по функционалу ПО. Чек-листы можно расширять. Простой чек-лист показан в таблице 3.4.

Таблица 3.4 – Пример чек-листа

№	Проверка	Результат	Комментарий
1	Регистрация в личный профиль		
1.1	Регистрация на сайте	Pass	
1.2	Редактирование профиля	Fail	Bug#1054
2	Поиск		
2.1	По названию	Pass	
2.2	По ссылкам	Pass	

Пример 2

Чек-лист без проверки детально показан в таблице 3.5.

Таблица 3.5 – Пример детального чек-листа

#	Компонент	Тест/ожидаемый результат
1	Ссылка	1. Проверка нажатия (кликабельности). 2. Проверка перехода. 3. Цвет ссылки должен поменяться, если по ней был совершен переход. 4. Название страницы, на которую происходит переход, должно соответствовать названию ссылки
2	Текстовое поле	1. Должно быть доступно для редактирования в соответствии с логикой. 2. Печатный текст должен отображаться корректно и сохраняться при изменении фокуса мыши. 3. Длина поля должна соответствовать длине предполагаемого содержимого. 4. При необходимости должен контролироваться тип вводимых данных (текст, число, дата). 5. При необходимости должно контролироваться значение вводимых данных (минимум, максимум)

Пример 3

Пример чек-листа тестирования удобства использования:

- Содержание веб-страницы верное, без грамматических и орфографических ошибок.
- Все шрифты соответствуют требованиям.

- Все тексты правильно выровнены.
- Все сообщения об ошибках верные, без орфографических и грамматических ошибок, и соответствуют заголовку окна.
- Подсказки существуют для всех полей.
- Все поля правильно выровнены.

Пример 4

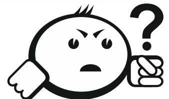
Пример чек-листа для тестирования мобильного приложения по характеристикам устройства (табл. 3.6).

Таблица 3.6 – Пример чек-листа для мобильных приложений

#	**Описание**	**Pass/fail**	**Примечания**
1.1	Можно ли установить приложение?		
1.2	Ведет ли оно себя правильно при входящем звонке?		
1.3	Ведет ли оно себя правильно при входящем SMS?		
1.4	Ведет ли оно себя правильно при подключении зарядного устройства?		
1.5	Ведет ли оно себя правильно при отключении зарядного устройства?		
1.6	Ведет ли оно себя правильно, если устройство переведено в спящий режим?		

Существует несколько специальных инструментов для написания чек-листов.

- таблицы Excel/OpenOffice для самостоятельной работы;
- таблицы GoogleDocs для распределения в команде;
- Sitechco для командной работы и версионности;
- Xmind.



Контрольные вопросы по главе 3

1. Какие этапы включается в себя жизненный цикл разработки ПО?
2. Чем отличается Agile от водопадной и v-образной модели?

3. Назовите современный методологии разработки ПО.
4. Распишите этапы тестирования.
5. Что такое тест-дизайн?
6. Назовите основные методы проектирования тестов.
7. Назовите техники метода черного ящика.
8. В чем суть попарного тестирования?
9. Что должен в себе содержать тест-кейс?
10. Приведите чек-листы в своей повседневной жизни.

4 Классификация видов тестирования

Тестирование можно классифицировать по очень большому количеству признаков, и практически в каждой серьезной книге о тестировании автор показывает свой (безусловно, имеющий право на существование) взгляд на этот вопрос. Зачем вообще нужна классификация тестирования? Согласно определению ISTQB (некоммерческая организация, занимающаяся определением различных принципов развития сферы тестирования ПО), виды тестирования являются *средством четкого определения цели определенного уровня для программы или проекта*. Классификация позволяет упорядочить знания и значительно ускоряет процессы планирования тестирования и разработки тест-кейсов; способствует оптимизации трудозатрат (тестировщику каждый раз не приходится изобретать велосипед). Опытные тестировщики редко используют эту классификацию. В реальной жизни тестировщик работает на основе опыта и интуиции.

Для каждого уровня тестирования могут быть определены: цели, артефакты процесса разработки, на основании которых будут разработаны тестовые сценарии; объекты тестирования, типичные дефекты и отказы, которые могут быть найдены во время тестирования.

Тестирование ПО можно классифицировать по следующим признакам (рис. 4.1):

1) по знанию системы;
2) по позитивности;
3) по целям (объекту);
4) по исполнителям (субъекту);
5) по времени проведения (тестирование изменений);
6) по степени автоматизации;
7) по состоянию (по исполнению кода).

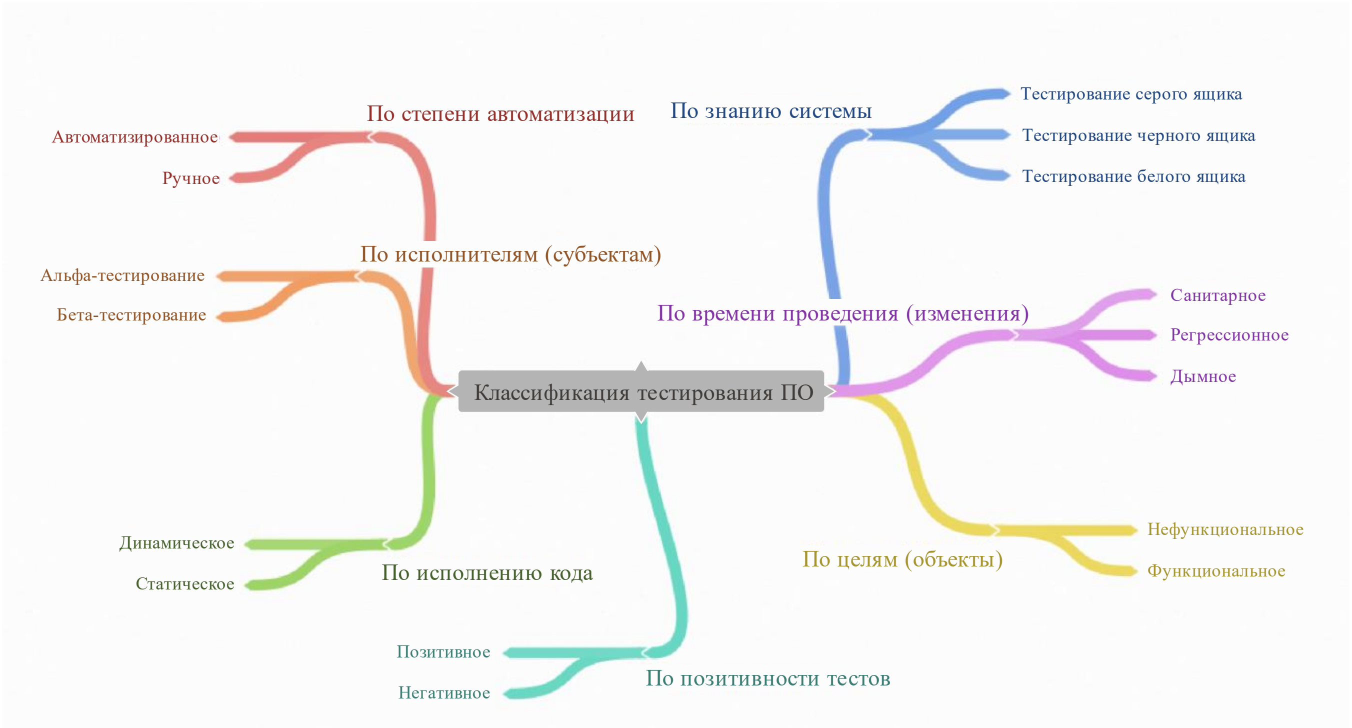


Рис. 4.1 – Классификация тестирования ПО

4.1 По знанию системы

По знанию системы выделяют:

- тестирование черного ящика (*black box testing*);
- тестирование белого ящика (*white box testing*);
- тестирование серого ящика (*grey box testing*).

. . . .

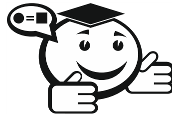
Тестирование черного ящика (*black box testing, closed box testing, specification-based testing*) *– у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.*

. . . .

Тестирование методом черного ящика, также известное как тестирование, основанное на спецификации, или тестирование поведения – техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы. Нужно концентрироваться на том, *что* программа делает, а не на том, *как* она это делает.

Поскольку это *тип* тестирования, по определению он может включать другие его виды. Тестирование черного ящика может быть как *функциональным*, так

и *нефункциональным*. Техника черного ящика применима на всех *уровнях тестирования* (от модульного до приемочного), для которых существует спецификация. Например, при осуществлении системного или интеграционного тестирования требования или функциональная спецификация будут основой для написания тест-кейсов.



*Согласно ISTQB, **тест-дизайн, основанный на технике черного ящика**, – это процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.*

Целью этой техники является поиск ошибок в следующих категориях:

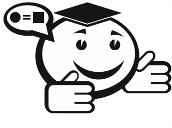
- неправильно реализованные или недостающие функции;
- ошибки интерфейса;
- ошибки в структурах данных или организации доступа к внешним базам данных;
- ошибки поведения или недостаточная производительности системы.

Преимущества:

- тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
- тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;
- можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

- тестируется только очень ограниченное количество путей выполнения программы;
- без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
- некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования.



Тестирование белого ящика (*white box testing, open box testing, clear box testing, glass box testing*) *– у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.*

Тестирование белого ящика еще часто называют структурным тестированием или тестированием на основе структуры. Разработка тестовых сценариев происходит на основании анализа внутренней структуры компонента или системы.

Техники, основанные на структуре, или белого ящика:

- тестирование операторов;
- тестирование альтернатив.

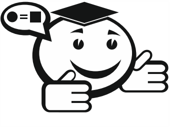
Преимущества:

- показывает скрытые проблемы и упрощает их диагностику;
- допускает достаточно простую автоматизацию тест-кейсов и их выполнение на самых ранних стадиях развития проекта;
- мотивирует разработчиков к написанию качественного (чистого) кода.

Недостатки:

- не может выполняться тестировщиками, не обладающими достаточными знаниями в области программирования;
- поведение ПО исследуется в отрыве от реальной среды выполнения и не учитывает ее влияние.

Написать автотесты – это еще полдела, необходимо проверить, а весь ли код покрыт тестами. Автоматические тесты должны покрывать 100% функционала, нужно стремиться к тому, чтобы каждая строчка кода была исполнена в результате исполнения хотя бы одного теста. Данная характеристика называется *code coverage* и буквально означает степень покрытия кода тестами.



Покрытие кода (***code coverage***) *– метод анализа, определяющий, какие части программного обеспечения были проверены* (*покрыты*) *набором тестов, а какие – нет, например, покрытие операторов, покрытие альтернатив или покрытие условий.*

Тестирование белого ящика применимо на разных уровнях тестирования – от модульного до системного, но главным образом применяется именно для реализации модульного тестирования компонента его разработчиком.

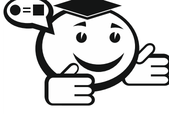
Метод серого ящика сочетает преимущества и недостатки методов белого и черного ящика.

- с одной стороны, тестирование ориентировано на пользователя, а значит, используется имитация поведения пользователя, т. е. применяется тестирование черного ящика;
- с другой – информированное тестирование, т. е. тестировщик знает, как устроена хотя бы часть тестируемого ПО, и активно использует это знание.

4.2 По позитивности

По критерию позитивности тестов выделяют:

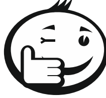
- позитивное тестирование (*positive testing*);
- негативное тестирование (*negative testing*).



Позитивное тестирование *проводится по сценариям, предполагающим нормальную, «правильную» работу системы в заведомо корректных условиях.*

Негативное тестирование *использует сценарии, проверяющие ситуации, связанные с потенциальными дефектами в системе.*

Негативное тестирование нацелено на демонстрацию того, что система или функция не работают. Негативное тестирование относится в большей степени к позиции тестировщика, нежели к определенному подходу к тестированию или методу проектирования тестов, например тестирование с некорректными входными значениями или тестирование обработки исключений.



При наличии позитивных и негативных тестов приоритет исполнения имеют позитивные тесты. Это связано с тем, что время на тестирование ограничено и в первую очередь необходимо проверить правильную работу системы, а только потом обработку исключительных ситуаций. При негативном тестировании чаще выявляются дефекты.

4.3 По целям (или объекту)

По целям (или объекту) разделяют (рис. 4.2):

- нефункциональное тестирование (*non-functional testing*);
- функциональное тестирование (*functional testing*).

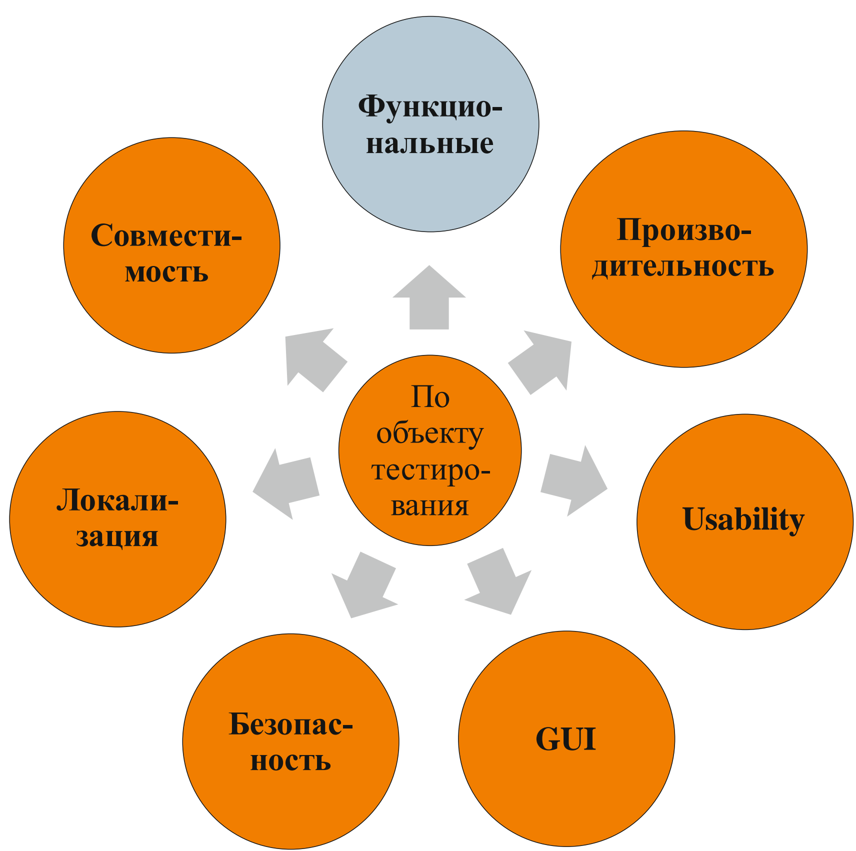


Рис. 4.2 – Классификация тестирования ПО по целям (объекту)

Нефункциональное тестирование (*non-functional testing*) – *вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований), таких как удобство использования, совместимость, производительность, безопасность и т. д.*

Рассмотрим некоторые виды нефункционального тестирования более подробно.

Тестирование производительности (*performance testing*) – исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке.

В рамках тестирования производительности выделяют следующие подвиды:

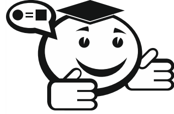
- *Нагрузочное тестирование* (*load testing*, *capacity testing*) – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).
- *Стрессовое тестирование* (*stress testing*) – исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчетный уровень, или в ситуациях недоступности значительной части необходимых приложению ресурсов. Стрессовое тестирование может выполняться и вне контекста нагрузочного тестирования: тогда оно, как правило, называется «тестированием на разрушение» (*destructive testing*) и представляет собой крайнюю форму негативного тестирования.
- *Тестирование масштабируемости* (*scalability testing*) – исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов.
- *Объемное тестирование* (*volume testing*) – исследование производительности приложения при обработке различных (как правило, больших) объемов данных.

Цель тестирования производительности – выявить «узкие места» в системе или архитектуре. Пользователям приложения нужны две вещи: быстрый ответ и высокая отказоустойчивость. С помощью тестирования производительности возможно выявлять эти узкие места в архитектуре и независимо масштабировать, конфигурировать и регулировать сервисы для достижения такого быстродействия конечными пользователями.

Нагрузочное тестирование – это проверка нашей системы с помощью огромного количества одновременных пользователей или подключений, которые и являются нашей *нагрузкой*. Это количество мы постоянно увеличиваем для достижения максимального количества задач, с которыми система может справиться. Нагрузочное испытание наиболее актуально при релизе нового сервиса, когда нужно проверить, выдержит ли он ожидаемый поток трафика. Моделирование падения системы от нагрузки – это цель стресс-тестирования. Наиболее очевидный способ сделать это – прогонять тест, наращивая количество запросов, пока система не перестанет отвечать.

Для имитации различных вызовов клиентского приложения существует большое количество различных инструментов: Apache JMeter, HP LoadRunner, Silk Performer from Micro Focus, WebLOAD и Gatling, которые часто используются для выполнения различных видов тестирования производительности.

. . . .



Графический интерфейс пользователя (***Graphical user interface***, ***GUI***) *– разновидность пользовательского интерфейса, в котором элементы интерфейса* (*меню, кнопки, значки, списки и т. п.*)*, представленные пользователю на дисплее, исполнены в виде графических изображений.*

Тестирование интерфейса пользователя (***GUI***) *– это тестирование, при котором проверяются элементы интерфейса пользователя.*

. . . .

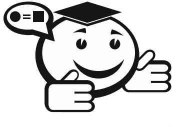
Задачей тестирования графического интерфейса пользователя является обнаружение ошибок следующего характера:

- ошибки в функциональности посредством интерфейса;
- необработанные исключения при взаимодействии с интерфейсом;
- потеря или искажение данных, передаваемых через элементы интерфейса;
- ошибки в интерфейсе (несоответствие проектной документации, отсутствие элементов интерфейса).

Пример бага графического интерфейса: при открытии сайта и/или нажатии кнопки «Главная» из любого раздела картинка основной части экрана отображается не сразу, вместо нее на несколько секунд выдается код (рис. 4.3).

Самые популярные специальные инструменты и рамки для тестирования пользовательского интерфейса: FitNesse, iMacros, Coded UI, Jubula, LoadUI.

. . . .



Тестирование удобства использования (*usability testing*) *выполняется с целью определения того, насколько органично используется пользовательский интерфейс целевыми пользователями, т. е. проверяется интуитивность интерфейса.*

. . . .

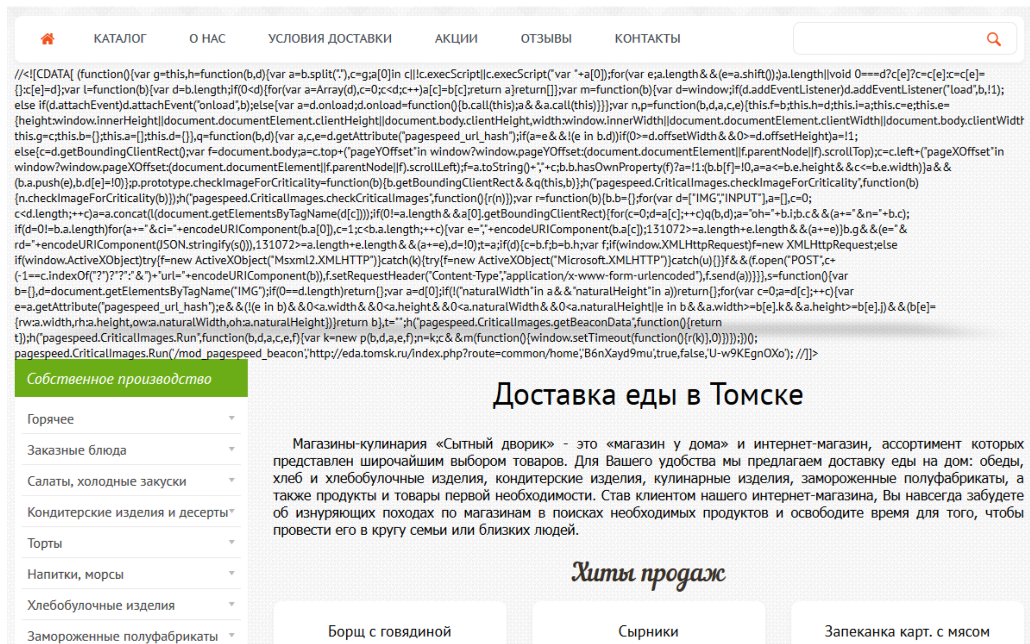


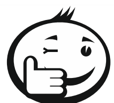
Рис. 4.3 – Пример бага графического интерфейса

Юзабилити-тестирование часто проводится путем привлечения группы потенциальных пользователей (фокус-группы) с целью собрать впечатления от работы с системой. Часто такой подход используют компании, разрабатывающие игры. Выпуская на рынок демоверсию игры, собирают и анализируют отзывы пользователей и учитывают их при следующем релизе.

Проверка юзабилити приложения заключается:

- в оценке соответствия дизайна приложения его функциональности, заданной заказчиком;
- анализе используемых графических элементов, цветового оформления с точки зрения восприятия;
- оценке удобства навигации и ссылочной структуре;
- анализе текстового наполнения сайта;
- оценке удобства использования функциями приложения (сервисами, если это сайт);
- анализе шрифтового оформления текста.

При тестировании юзабилити было бы полезно учитывать 10 правил проектирования пользовательского интерфейса, составленных Якобом Нильсеном (один из основателей компании Nielsen Norman Group, которая занимается проектированием пользовательских интерфейсов) [15]:



1. Информативность системы. Пользователь всегда должен знать текущий статус приложения.

2. Приближенность приложения к реальному миру. Диалог с пользователем должен вестись на понятном ему языке, стоит остерегаться использования непонятной терминологии.

3. Система должна иметь выходы. Приложение всегда должно иметь «запасные выходы» из любой функциональности, которые пользователь по ошибке запустил.

4. Однозначность. Все термины, функции и понятия должны описываться в едином толковании – у пользователя не должно возникнуть путаницы.

5. Предусмотрительность. Система должна всячески «оберегать» пользователя от возможных ошибок.

6. Наглядность. Пользователь не должен ломать голову в попытках понять, что ему нужно делать, или пытаясь вспомнить, как он достиг того или иного состояния системы. Возможные манипуляции с программой должны быть постоянно наглядными.

7. Гибкость и эффективность. Необходимо предоставить опытным пользователям возможность избегать рутинных действий, и в то же самое время скрывать расширение функционала от неопытных.

8. Лаконичность и точность. Диалоги должны содержать только ту информацию, которую необходимо донести до пользователя, ничего лишнего.

9. Лояльность к ошибкам. Информация об ошибках должна быть понятной и содержать подсказки к дальнейшим действиям.

10. Постоянная справка. Как бы информативно ни была спроектирована система, она всегда должна содержать раздел справки и документации.

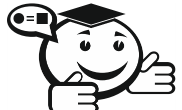
. . . .

Тестирование удобства использования и тестирование интерфейса пользователя не одно и то же! Например, корректно работающий интерфейс может быть неудобным, а удобный может работать некорректно.

Пример «неюзабилити» интерфейса (перенасыщенность информацией главной страницы) показан на рисунке 4.4.

Популярные инструменты для тестирования юзабилити: User Zoom, Reflector, Loop, Usabilla, GTMetrix.

Рис. 4.4 – Пример «неюзабилити» интерфейса



. . . .

Тестирование интернационализации (*internationalisation testing*) *– вид тестирования, при котором проверяется готовность приложения к работе с различными языковыми настройками, в частности способность корректно отображать шрифты, пункты меню, производить поиск, сортировку, способность приложения обрабатывать файлы, поименованные на различных языках.*

Тестирование локализации (*localization testing*) *проверяет, насколько корректно продукт адаптирован к работе на том или ином языке: все ли переведено и переведено правильно, не нарушилась ли логика построения интерфейса и обработки данных и т. д.*

. . . .

Языковая локализация – это процесс адаптации продукта, который ранее был переведен на несколько языков, для определенной страны или региона. Цель проводимого тестирования локализации – проверить мультиязычный интерфейс приложения или сайта на наличие ошибок перевода, правильность почтовых адресов, имени и фамилии, валют, формата даты и времени и пр. Этот вид тестирования также дает возможность узнать, насколько понятным конечному пользователю стал продукт, соответствует ли он ожиданиям и выполнит ли возложенные на него задачи.

Локализация часто затрагивает как техническую, так и культурную часть. Этим и объясняется наличие столь разных проверок и подходов, которые не все-

гда можно подогнать под какие-либо шаблоны. Чем более разносторонними будут тесты, тем больше шансов, что продукт пройдет всестороннюю проверку и в итоге полностью удовлетворит как заказчика, так и пользователей.

Данный процесс локализации включает в себя:

- перевод пользовательского интерфейса;
- перевод документации;
- контроль формата даты и времени;
- внимание к денежным единицам;
- внимание к правовым особенностям;
- раскладку клавиатуры пользователя;
- контроль символики и цветов;
- толкование текста, символов, знаков.

Интернационализация – более обобщенное понятие, подразумевающее проектирование и реализацию программного продукта или документации таким образом, который максимально упростит локализацию приложения.

Интернационализация включает в себя:

- создание продукта с учетом возможности кодировки Unicode (стандарт кодирования, поддерживающий практически все языки мира);
- создание в приложении возможности поддержки элементов, которые невозможно локализовать обычным образом (вертикальный текст азиатских стран, чтение LTR (left-to-right), RTL (right-to-left) для азиатских языков, TTB (top-to-bottom));
- возможность загрузки локализированных элементов в будущем при желании пользователя.

Интернационализация не предполагает перевод текста программ или документации на другой язык, она подразумевает разработку приложений таким образом, который сделает локализацию максимально простой и удобной, а также позволит избежать проблем при интеграции продукта для стран с отличающейся культурой.

Примеры локализации представлены на рисунке 4.5.

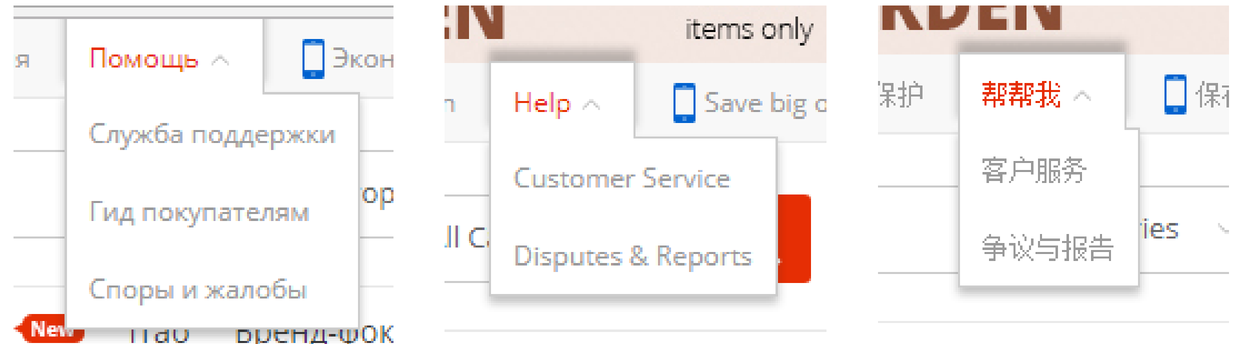


Рис. 4.5 – Пример локализации

К ошибкам локализации относятся неподдерживаемые шрифты, неверные размеры кнопок и выпадающих списков, некорректные переносы в элементах управления, названиях колонок в таблицах и т. д. (рис. 4.6).

Рис. 4.6 – Ошибки локализации

Локализация обычно осуществляется с использованием некоторой комбинации внутренних ресурсов, независимых подрядчиков и полномасштабных услуг компании локализации. Некоторые инструменты для обеспечения тестирования локализации: eggPlant, Babylon.NET by Redpin и smartCAT. Для проверки шрифтов используются встроенные в браузер «инструменты разработчика» и/или расширения для браузеров: Какой шрифт; Typ.io Peek; csscan; WhatFont; typ.io; FonFace Ninja; Type Sample; Fount.

__Тестирование безопасности__ (security and access control testing) производится для оценки уязвимости программного обеспечения к различным вредоносным воздействиям на уровне приложения, операционной системы, сети и т. д.

Тестирование безопасности – это множество вещей, суть которых заключается в том, чтобы усложнить условия для кражи данных, денег и информации. Является одним из видов тестирования ПО, выполняемого специализированной

группой тестировщиков. Например, в одной из систем интернет-платежей есть специальный отдел, который профессионально занимается взламыванием своего же веб-сайта и получает премии за каждую найденную ошибку в системе обеспечения безопасности.

В ходе тестирования тестировщик чаще всего играет роль взломщика и начинает манипулировать приложением:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- обрушение системы;
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
- просмотр несекретных данных в надежде найти ключ для входа в систему.

Тестирование безопасности может включать в себя *тестирование уязвимости*, оно заключается в выявлении уязвимости ПО, оборудования или сети, которая может быть использована хакерами и другими вредоносными программами, похожими на вирусы или червей. Тестирование на уязвимость является ключом к обеспечению безопасности и доступности ПО. С ростом числа хакеров и вредоносных программ тестирование уязвимостей имеет решающее значение для успеха бизнеса.

Всемирная некоммерческая благотворительная организация OWASP (https://www.owasp.org/), которая занимается повышением безопасности ПО, составила список из 10 самых опасных уязвимостей, которым могут быть подвержены интернет-ресурсы. Сообщество обновляет и пересматривает этот список раз в три года, поэтому он содержит актуальную информацию. Последнее обновление включает в себя следующие уязвимости:

- внедрение кода;
- некорректная аутентификация и управление сессией;
- утечка чувствительных данных;
- внедрение внешних XML-сущностей (XXE);
- нарушение контроля доступа;
- небезопасная конфигурация;
- межсайтовый скриптинг;
- небезопасная десериализация;

- использование компонентов с известными уязвимостями;
- отсутствие журналирования и мониторинга.

Даже проведя полный цикл тестирования безопасности, нельзя быть на 100% уверенным, что система по-настоящему обезопасена. Количество сетевых атак неуклонно растет. Это наглядно иллюстрирует интерактивная карта кибер-атак в режиме онлайн [16], которая показана на рисунке 4.7.

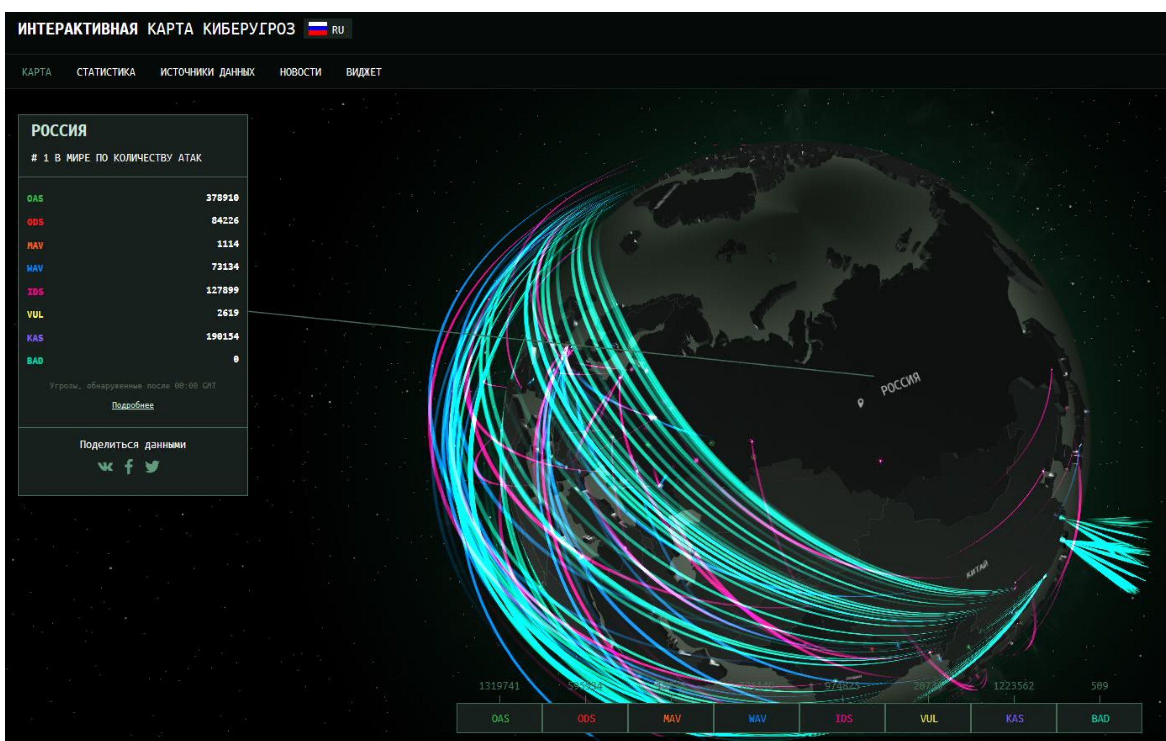
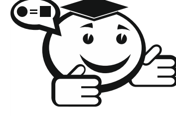


Рис. 4.7 – Интерактивная карта кибератак в режиме онлайн

К сожалению, самое слабое звено в работе ПО – это человек. Человек – существо несовершенное. Ему свойственно ошибаться, поэтому в тестировании безопасности сегодня актуально такое направление, как социальная инженерия.

В сфере информационной безопасности термин «социальная инженерия» используется для описания науки и искусства психологической манипуляции. По статистике, 55% убытков, связанных с нарушениями информационной безопасности, возникают по вине сотрудников, подвергшихся влиянию социальных инженеров. Для снижения рисков, связанных с этим обстоятельством, используются различные технические и административные механизмы защиты. Один из них – повышение осведомленности в области информационной безопасности. Поэтому важное направление тестирования на проникновение (*penetration testing*) – метод оценки безопасности компьютерных систем или сетей средствами моделирования атаки злоумышленника.

В тестировании на проникновение могут использоваться инструменты: Nmap; Nessus; Metasploit; Wireshark; OpenSSL; Cain & Abel; THC Hydra.



Тестирование совместимости (*compatibility testing*) – *это вид тестирования, основной целью которого является проверка качественной работы разрабатываемого программного средства с другим ПО* (*окружением, операционными системами, браузерами и т. д.*).

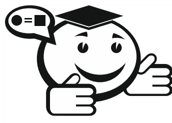
Тестирование с разными браузерами ***называется кросс-браузерным тестированием*** (*cross-browser testing*).

Тестирование с разными операционными системами называется ***кросс-платформенным тестированием*** (*cross-platform testing*).

Окружение может включать в себя следующие элементы:

- аппаратная платформа;
- сетевые устройства;
- периферия (принтеры, CD/DVD-приводы, веб-камеры и т. д.);
- операционная система (Unix, Windows, MacOS…);
- базы данных (Oracle, MS SQL, MySQL…);
- системное программное обеспечение (веб-сервер, антивирус и др.);
- браузеры (Internet Explorer, Firefox, Opera, Chrome, Safari).

Основные инструменты, которые часто используются для тестирования всех конфигураций: BrowserStack, CrossBrowserTesting by Smart Bear, Litmus, Browsera, Rational Clearcase by IBM, Ghostlab.



Функциональное тестирование (*functional testing*) – *тестирование, основанное на анализе спецификации функциональности компонента или системы.*

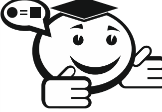
Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (*модульном*, *интеграционном*, *системном*, *приемочном*). *Функциональное тестирование* проводится с целью выявления ошибок в функционировании программы путем сопоставления

фактического (*actualt*) и ожидаемого (*expected*) результатов. Основным источником ожидаемого результата являются спецификации функциональных требований (*requirements*) к программному продукту.

Модульное тестирование (*компонентное, юнит-тестирование, unit testing*) *занимается поиском дефектов и верификацией функционирования программных модулей, программ, объектов, классов и т. п., которые можно протестировать изолированно.*

Цель модульного тестирования – изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Обычно модульное тестирование производится с доступом к тестируемому коду и с поддержкой рабочего окружения, такого как фреймворк модульного тестирования или утилиты отладки. На практике такое тестирование обычно производится разработчиками, которые пишут код. Дефекты обычно исправляются сразу после того, как становятся известны, без занесения их в базу дефектов. В процессе могут быть использованы заглушки, прототипы, драйверы и эмуляторы.

Один из подходов к модульному тестированию – составить автоматизированные тест-кейсы до кодирования. Это называется разработкой через тестирование. В 1999 г. разработка через тестирование была связана с концепцией «сначала тест» (*test-first*), применяемой в экстремальном программировании, однако позже она выделилась как независимая методология. Кент Бек, считающийся изобретателем этой техники, предлагает принцип «подделай, пока не сделаешь» (*fake it till you make it*).

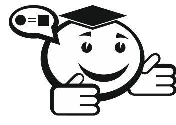


Разработка через тестирование (*test-driven development, TDD*) *– техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.*

TDD требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста

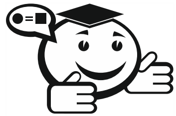
подтверждает поведение, предполагаемое программистом. Разработчики часто пользуются библиотеками для тестирования (фреймворками, *testing frameworks*) для создания и автоматизации запуска наборов тестов. Существуют различные инструменты (*JUnit*, *PHPUnit*, *TestNG*, *PyTest*), которые позволяют создавать и поддерживать качественные юнит-тесты.

Следующий за модульным уровень – интеграционное тестирование.



Интеграционное тестирование (*integration testing*) *предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы* (*операционной системой, оборудованием*) *либо связи между различными системами.*

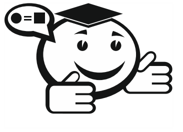
Основная задача интеграционного тестирования – поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями. Инструменты, используемые для интеграционного тестирования такие же, как и те, которые используются для модульного тестирования, хотя и занимают больше времени. С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (*stub*) на месте отсутствующих модулей.



Заглушка (*stub*) – *это имитация вызываемой функции, возвращающая те же данные, но ничего больше не делающая.*

Заглушки используются для получения данных из внешней зависимости, подменяя ее. При этом игнорируются все данные, могущие поступать из тестируемого объекта в заглушку. Это один из самых популярных видов тестовых объектов.

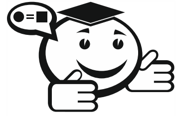
Следующий уровень – системное тестирование.



Системное тестирование (*system testing*) – *процесс тестирования системы в целом с целью проверки того, соответствует ли она установленным требованиям.*

Системному тестированию с помощью метода черного ящика подвергается проект в целом. Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация. Системное тестирование чаще всего выполняет независимая тестовая команда. Также к системному тестированию можно отнести альфа-тестирование и бета-тестирование, которые будут рассмотрены позже.

Еще один уровень тестирования, который часто применяется, – это приемочное тестирование.

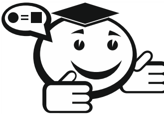


Приемочное тестирование (*acceptance testing*) *– формальное тестирование по отношению к потребностям, требованиям и бизнес-процессам пользователя, проводимое с целью определить соответствие системы критериям приемки и дать возможность пользователям, заказчикам или иным авторизированным лицам определить, принимать систему или нет* (*IEEE 610*).

4.4 По исполнителям (субъектам)

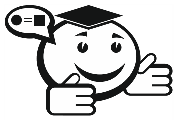
По исполнителям (субъектам) разделяют:

- тестирование разработчиками или специалистами до передачи пользователю, или альфа-тестирование (*alpha testing*);
- тестирование пользователями после передачи пользователю, или бета-тестирование (*beta testing*).



Альфа-тестирование (*alpha testing*) *– имитация реальной работы с системой штатными разработчиками либо реальная работа с системой потенциальными пользователями/заказчиком.*

Альфа-тестирование выполняется отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться ПО. Альфа-тестирование часто применяется к коробочному программному обеспечению в качестве внутреннего приемочного тестирования.



Бета-тестирование (*beta testing*) *– распространение предварительной версии* (*иногда с ограничениями по функциональности или времени работы*) *для некоторой большой группы лиц, с тем чтобы убедиться, что продукт содержит достаточно мало ошибок.*

Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей. Бета-тестировщики – это ограниченная группа пользователей из целевой аудитории. Они работают с новой системой до того, как она станет доступна всем пользователям. Компании, выпускающие онлайн-игры, нередко начинают привлекать к тестированию своих будущих пользователей еще на стадии альфа-тестов. Надо сказать, что разработчики не испытывают недостатка в желающих принять участие в такой работе. Производитель обычно сохраняет бета-тестерам после перехода на финальную версию их достижения и регалии, заработанные при тестировании.

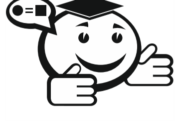
Бета-тестирование часто проводится как форма внешнего приемочного тестирования готового программного обеспечения, для того чтобы получить отзывы рынка.

4.5 По времени проведения

По времени проведения выделяют следующие виды тестирования:

- дымовое (*smoky test*);
- санитарное (*sanity*);
- регрессионное (*regression*).

Дымовое тестирование – проверка самой важной, ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения.



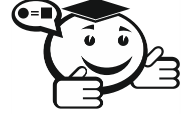
Дымовое (***дымное***) ***тестирование*** (*smoky testing*) *– первый прогон программы* (*после написания или после внесения существенных изменений*).

Как правило, используется для определения, готова ли программа для проведения более обширного тестирования. Дымовое тестирование позволяет обна-

руживать критичные ошибки заранее, их правка начнется незамедлительно после их обнаружения. Автоматизация дымового тестирования позволяется собрать все ошибки на текущей версии собранного продукта и передать их в отдел разработки.

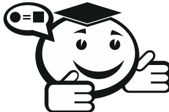
Ежедневная сборка и тест «на дым» являются передовыми практическими методами.

Санитарное тестирование относится к виду тестирования, которое используется с целью доказательства работоспособности конкретной функции или модуля согласно заявленным техническим требованиям.



Санитарное тестирование (***санити-тестирование***) (*sanity testing*) *– это узконаправленное тестирование, достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям.*

Зачастую санитарное тестирование используют для проверки какой-либо части программы или приложения в результате внесенных изменений. Выполнение обычно происходит в ручном режиме.



Регрессионное тестирование (*regression testing*) *– тестирование уже протестированной программы, проводящееся после модификации для уверенности в том, что процесс модификации не внес или не активизировал ошибки в областях, не подвергавшихся изменениям.*

Процесс возникновения ошибок в таких ситуациях выглядит следующим образом: в определенный модуль существующего функционала вносятся изменения, но т. к. данный модуль может быть взаимосвязан с другими модулями, то изменения в одном модуле могут коснуться нескольких модулей. Соответственно, следует продумать регрессионное тестирование после подобных изменений.

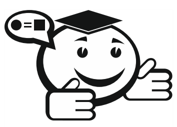
Виды регрессионного тестирования:

- регрессия дефектов – тестирование старых дефектов с целью узнать, были ли они исправлены;

- регрессия исправленных дефектов – тестирование дефектов, исправленных в прошлом, с целью узнать, не были ли они повторены после внесения изменений;
- регрессия побочного эффекта – тестирование выборочного функционала с попыткой узнать, не «задели» ли новые изменения уже существующий функционал.

Регрессионное тестирование может выполняться на всех уровнях тестирования и включает функциональное, нефункциональное и структурное тестирование. Такие тесты запускаются множество раз и меняются медленно, поэтому регрессионное тестирование является хорошим кандидатом на автоматизацию.

Отдельно можно выделить ре-тест (*re-test*).



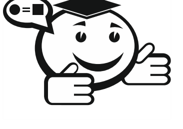
Ре-тест (*re-test*) *перепроверяет и подтверждает факт того, что ранее тест-кейсы, которые не прошли, проходят после того, как дефекты исправлены.*

При ре-тесте используется тот же самый тест-кейс, который выявил дефект. Он проверяет, исправлен ли дефект.

4.6 По степени автоматизации

По степени автоматизации тестирование бывает:

- ручное (*manual testing*);
- автоматизированное (*automated testing*).



Ручное тестирование (*manual testing*) *– это исполнение тестов без помощи каких-либо программ, автоматизирующих работу тестировщика.*

Ручное тестирование – очень трудоемкий процесс, но инструменты (программы) не могут чувствовать, как конечные пользователи, они не могут рассчитывать на предыдущий опыт, интуицию, как это делают люди. Только люди могут полностью оценить программу, инструмент же просто определяет, проходят тесты или нет. Один из фундаментальных принципов тестирования гласит: *100%-ная автоматизация невозможна*. Поэтому ручное тестирование – необходимость.

К тому же не все виды тестирования можно автоматизировать:

- *Исследовательское тестирование*. Тест-кейсы выбираются исходя из опыта тестировщика и основываются на логических умозаключениях и интуиции. При этом у тестировщика отсутствует качественная документация, а тестирование должно быть завершено в сжатые сроки. Данный вид тестирования помогает за короткое время обнаружить наиболее критичные дефекты.
- *Тестирование юзабилити*. При проведении данного вида тестирования тестировщику важно определить, насколько удобным будет продукт для конечного пользователя. Естественно, тесты должны проводиться и анализироваться человеком.
- *Интуитивное тестирование* (*ad-hoc testing*). Данный вид тестирования выполняется без заранее разработанного сценария и определенных результатов. Выполняя проверки, тестировщик импровизирует и полагается на здравый смысл, свой опыт и знание продукта.

Автоматизированное тестирование (*automated testing*) *– вид тестирования, при котором используются специальные программные средства для выполнения тестов, сверки полученных фактических результатов с ожидаемыми значениями, установки предусловий для запуска тестов, а также для других функций, включая генерацию отчетов.*

Основные преимущества автоматизации: скорость и отсутствие человеческого фактора, но написание скрипта автоматизации требует от тестировщика знания сферы деятельности, автоматизации тест-кейсов и возможности выбрать соответствующий фреймворк. При этом автоматизация, как и разработка приложений, нуждается в тестировании. Скрипты, написанные с помощью автоматизации, необходимо тщательно проверить, используя все тестовые данные и негативное тестирование.

Инструменты автоматизации полезны для многих видов деятельности, они позволяют сэкономить массу времени, выполняя различные задачи:

- регрессионное тестирование;
- дымовое тестирование;
- нагрузочное тестирование;
- тестирование производительности;

- тестирование методом белого ящика (с помощью инструментов модульного тестирования);
- тестирование локализации;
- анализ покрытия кода.

Автоматизация была и остается ключевой тенденцией в тестировании на протяжении более 15 лет. В 2019 г. 68% респондентов World Quality Report (Мировой доклад о качестве) отметили, что благодаря автоматизации улучшилось тестовое покрытие продуктов по сравнению с предыдущим годом, когда процент был ниже на 17% [17]. Некоторые специалисты утверждают, что автоматизация полностью вытеснит работу ручных тестировщиков в ближайшем будущем. Однако этого не произойдет, потому что в некоторых аспектах ни один инструмент не сравнится с человеком. Поэтому чаще используют оба подхода. Комбинация обоих видов – идеальный способ получить от тестирования максимальный результат. В будущем все QA-инженеры должны будут обладать навыками ручного и автоматизированного тестирования.

4.7 По состоянию системы (по исполнению кода)

Все виды тестирования можно условно разделить по *состоянию системы* на две большие группы:

- статическое тестирование (*static testing*);
- динамическое тестирование (*dynamic testing*).

Статическое тестирование (*static testing*) *– это процесс анализа самой разработки программного обеспечения, т. е. тестирование без запуска программы.*

Данный вид тестирования осуществляется в основном программистами.

Тестирование артефактов разработки программного обеспечения, таких как требования, дизайн или программный код, проводят без исполнения этих артефактов. Большинство статических техник могут быть использованы для тестирования любых форм документации, включая вычитку кода, инспекцию проектной документации, функциональной спецификации и требований.

Статический анализ кода (*static code analysis*) – это анализ исходного кода, производимый без его исполнения.

Равноправный анализ (*peer review*) – рецензирование разрабатываемого программного продукта, проводящееся сотрудниками компании-разработчика с

целью нахождения дефектов и внесение улучшений. Примерами рецензирования являются: инспекция, технический анализ и разбор.

Инспекция (*inspection*) – тип равноправного анализа, основанный на визуальной проверке документов для поиска ошибок. Например, нарушение стандартов разработки и несоответствие документации более высокого уровня. Это наиболее формальная методика рецензирования, поэтому она всегда основывается на документированной процедуре (IEEE 610, IEEE 1028).

Формальное рецензирование (*formal review*) – рецензирование, характеризирующееся задокументированными процедурами и требованиями, например инспекция.

Даже статическое тестирование может быть автоматизировано, например можно использовать автоматические средства проверки синтаксиса программного кода.

Статические анализаторы: Lint, FindBugs, PVS-Studio, Coverity. PVS-Studio – это инструмент для выявления ошибок и потенциальных уязвимостей в исходном коде программ, написанных на языках C, C++, C# и Java. Работает в среде Windows, Linux и macOS.

Проведем статический анализ кода:

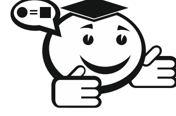
```
void m()
        {
                String src="kjlghkljfhdf";
                int srclen=src.length();
                 int k;
                 int i=0;
            for (k=i; i<srclen; k++)
            {
                 System.out.print(src.charAt(k));
              if (src.charAt(k)=='>')
                break;
            }

        }
```

Анализатор нашел подозрительный цикл: It is likely that a wrong variable is being compared inside the 'for' operator. Consider reviewing 'i'.

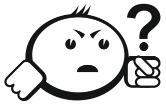
Этот код предрасположен к нарушению доступа (Access Violation). Цикл должен продолжаться, пока не найдется символ '>' или не закончится строка длиной в 'srclen' символов. Для сравнения ошибочно использована переменная 'i', а не 'k'. Если символ '>' найден не будет, то все закончится ошибкой доступа.

Статическое тестирование позволяет обнаружить дефекты, которые являются результатом ошибки и привести к сбоям в программном обеспечении, а динамическое тестирование позволяет продемонстрировать непосредственно сбои в программном обеспечении.



Динамическое тестирование (*dynamic testing*) *– это тестовая деятельность, предусматривающая эксплуатацию* (*запуск*) *программного продукта.*

Динамическое тестирование предполагает запуск программы, выполнение всех функциональных модулей и сравнение фактического поведения с ожидаемым. При этом динамическое тестирование может включать разные виды тестирования: тестирование черным, белым и серым ящиками; модульное интеграционное, системное и приемочное тестирование, функциональное и нефункциональное тестирование.



Контрольные вопросы по главе 4

1. Дайте определение термину «покрытие кода».
2. Назовите виды нефункционального тестирования и охарактеризуйте каждый из них.
3. Охарактеризуйте позитивное и негативное тестирование.
4. Чем отличается дымовое тестирование от санитарного?
5. Что такое регрессионное тестирование?
6. Почему ошибки возникают повторно?
7. Какие существуют виды регрессионного тестирования?
8. Что такое автоматизированное тестирование?
9. Какие тесты – лучшие претенденты на автоматизацию?
10. Как проводят статическое тестирование?

5 Особенности тестирования мобильных и веб-приложений

> Если вы хотите найти «легкий» баг в веб-приложении, попробуйте открыть его на телефоне и проверить, все ли там нормально отображается.
>
> *Народная мудрость*

Веб-ориентированные приложения стали широко популярными в конце 1990-х – начале 2000-х гг. Сегодня веб-приложения становятся более актуальными, распространенными и все более сложными. Практически любой специалист по тестированию программного обеспечения с опытом более года даст утвердительный ответ на этот вопрос, ведь существуют вполне объективные причины такого положения дел: аналитическое агентство We Are Social и крупнейшая SMM-платформа Hootsuite совместно подготовили пакет отчетов о глобальном цифровом рынке Global Digital 2019. По представленным в отчетах данным, сегодня во всем мире Интернетом пользуется более 4 миллиардов человек [18]. В 2019 г. *аудитория Интернета насчитывает 4,39 миллиарда человек*, что на 366 миллионов (9%) больше, чем в январе 2018 г. (рис. 5.1).

Рис. 5.1 – Статистика отчетов
о глобальном цифровом рынке Global Digital в мире

Во всем мире количество людей, которые пользуются мобильными телефонами, превысило 5,1 миллиарда. Благодаря этому уровень проникновения мобильной связи во всем мире поднялся до 67% – мобильный телефон есть у двух третей населения Земного шара (рис. 5.2).

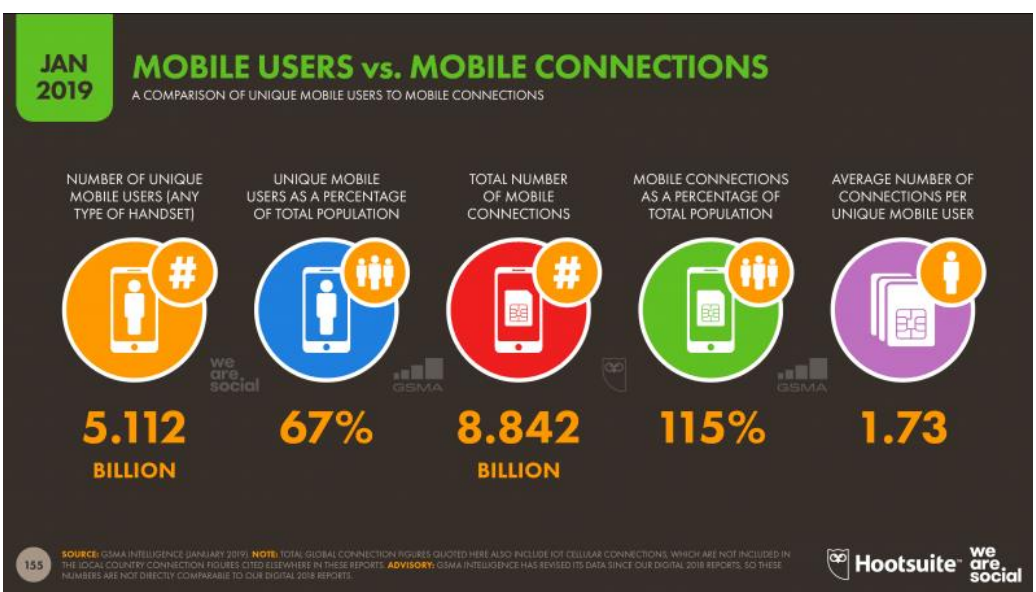


Рис. 5.2 – Статистика отчетов
о глобальном цифровом рынке Global Digital по мобильной связи

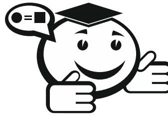
Люди во всем мире предпочитают выходить в интернет со смартфонов. Они генерируют больше веб-трафика, чем все прочие устройства суммарно.

В России насчитывается 109,6 миллионов интернет-пользователей (рис. 5.3).

Рис. 5.3 – Статистика отчетов
о глобальном цифровом рынке Global Digital в России

При взгляде на эти огромные цифры становится понятным, почему в мире разрабатывается так много новых мобильных и веб-приложений. Этот процесс приводит к необходимости привлечения большого количества специалистов в IT-сферу.

5.1 Особенности тестирования веб-приложений



Веб-приложение (*web apps*) *– это клиент-серверное приложение, в котором клиентом выступает браузер, а сервером – веб-сервер, что уже по сути является двумя различными программами, которые необходимо тестировать как отдельно, так и в связке.*

Тестировать надо все стороны приложения, используя различные виды тестирования ПО:

- тестирование удобства использования;
- функциональное тестирование;
- тестирование совместимости;
- тестирование баз данных;
- тестирование безопасности;
- тестирование производительности.

Самая трудоемкая часть тестирования – *функциональное тестирование*. На этом этапе тестируются все функциональные требования приложения: работа ссылок, поиска, кнопок; целостность базы данных, подгрузка файлов на сервер, работа счетчиков на страницах портала и пользовательских форм.

Функциональное тестирование веб-приложений включает:

- проверку внешних и внутренних ссылок;
- тестирование интерфейса (UI) (верстка, локализация);
- тестирование бизнес-логики;
- тестирование навигации;
- кросс-браузерное тестирование, включая тестирование на мобильных устройствах.

Задача функционального тестирования веб-приложения состоит в проверке доступности и корректности работы всех функциональных возможностей приложения в как можно более полном множестве клиентских браузеров и их версий.

При тестировании веб-приложений в помощь тестировщику было разработано множество расширений для браузеров и дополнительных вспомогательных инструментов.

Валидаторы проверяют html-код, как заданный с помощью ссылки на страницу, так и просто в виде загруженного файла или скопированного текста; обнаруживают ссылки, приводящие к ошибке 404 (не должно быть ссылок, ведущих на несуществующую или неправильно написанную страницу); дают список замечаний с рекомендациями по их исправлению. Пример валидатора: http://validator.w3.org/.

Тестирование верстки включает в себя:

- тестирование внешнего вида;
- тестирование адаптированности страницы.

***Верстка** – этап дизайна страницы сайта.*

***Адаптивная верстка** сайта – верстка, при которой CSS-стили меняются динамически для разной ширины окна браузера; позволяет веб-страницам автоматически подстраиваться под экраны планшетов и смартфонов*

При тестирование внешнего вида необходимо проверить:

- выделяются ли все элементы;
- кликабельны ли элементы (кнопки/ссылки);
- реакцию активных/неактивных элементов на наведение;
- наличие и соответствие подсказок (*tooltip*) у кликабельных элементов, назначение которых не очевидно;
- ввод и удаление данных;
- дизайн/шрифты/цветовую гамму;
- изображения и некорректное отображение текста вокруг изображений;
- соответствие макету.

И тут у тестировщика есть выбор инструментов. Проверить измерение расстояния на экране позволяет Screen Calipers (http://www.iconico.com/caliper/); проверить соответствие дизайна макета: Photoshop или PerfectPixel (плагин для браузера Chrome). Разобраться со шрифтами поможет, например, What Font. Проверить многообразие цветов – Color Zilla.

В процессе тестирования полезно все документировать, для этого используются инструменты для снятия скриншотов и записи экрана. Расширение Chrome Lightshot – легкий и удобный инструмент для скриншотов, тоже позволяющий их аннотировать. Скриншоты можно сохранять на жесткий диск или загружать в облако, чтобы поделиться ссылкой с кем-то еще. Инструмент Screencastify дает возможность делать запись экрана. Для снятия скриншотов и записи действий на экране можно использовать и Screenpresso.

Сайт должен корректно отображаться на любых ПК, планшетах или мобильных телефонах. И эту проблему решает адаптивная верстка, которую надо протестировать на всем множестве версий браузеров и гаджетов. И тут опять поможет автоматизация.

Существуют расширения Chrome – эмуляторы размеров экрана. Resolution Test поможет при тестировании веб-приложений на разных разрешениях и размерах экрана. Это расширение изменяет размеры браузера и эмулирует ваше приложение в нужном разрешении экрана. Window Resizer – еще одно полезное для веб-тестирования расширение. Оно отличается от Resolution Test возможностью устанавливать горячие клавиши, а также экспортировать свои настройки и переносить их на другой компьютер (рис. 5.4). Есть плагин и у Firefox – Firesizer, который изменяет размеры окна браузера, эмулируя различные разрешения экрана.

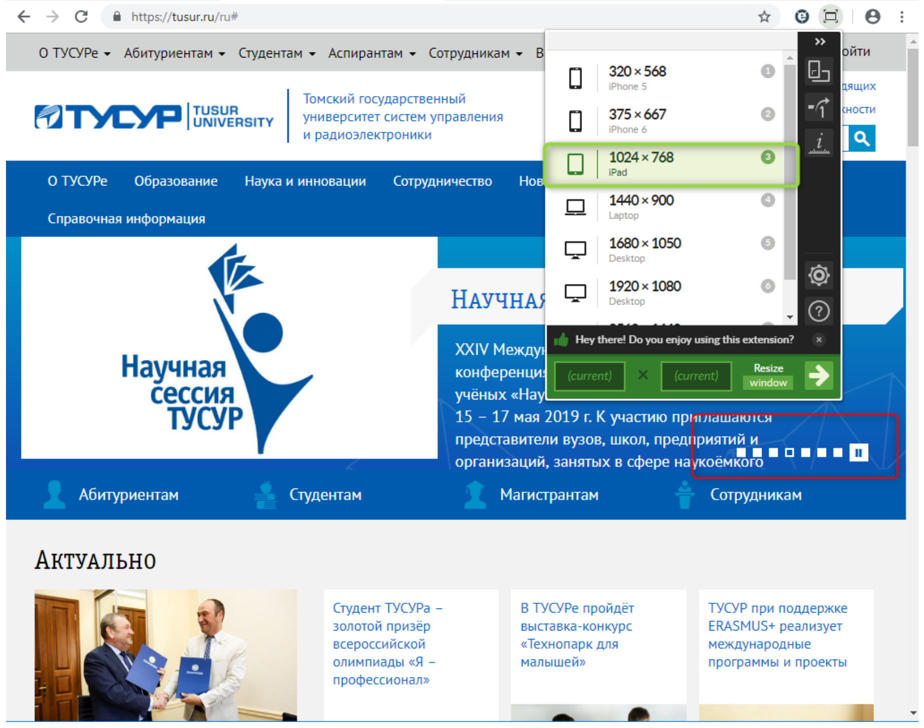


Рис. 5.4 – Дефект адаптивной верстки

Тестирование юзабилити – это анализ взаимодействия пользователя и сайта, поиск ошибок и их устранение. При этом проверяется:

- легкость взаимодействия;
- навигация;
- субъективная удовлетворенность пользователей;
- общий вид.

Сайт должен быть простым в использовании. Главное меню должно быть доступно на каждой странице. Контент должен быть логичным и простым для понимания. Проверьте текст на наличие ошибок. Применение темных цветов раздражает пользователей, не нужно использовать их в теме оформления. Для контента и фона страницы лучше применять общепринятые стандарты, чтобы цвет шрифта, рамок и т. д. не раздражал пользователей (на рисунке 5.5 пример сайта, на котором есть такие дефекты юзабилити, как загруженность сайта, неудачная цветовая гамма, много рекламы, которая закрывает собой контент).

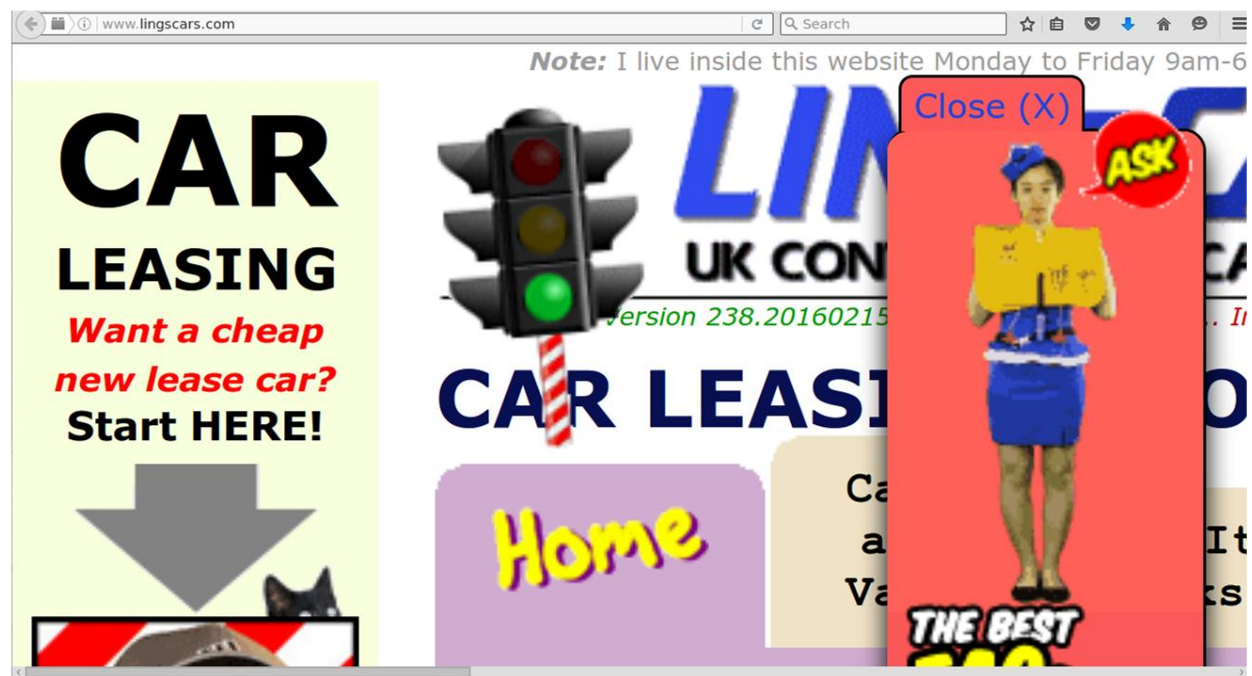


Рис. 5.5 – Пример неудобного сайта

Контент должен быть содержательным, ссылки – работать надлежащим образом, изображения – иметь соответствующий размер. Функция «Поиск по сайту» должна помогать легко находить нужный контент.

Главная цель веб-приложений – обработка сотен и даже тысяч запросов в секунду. Чтобы обеспечить такой трафик, очень важно определить потенциальные узкие места и сделать все возможное, чтобы избежать проблем. Сделать это позволяет *тестирование производительности*.

Тестирование производительности веб-приложения должно включать в себя:

- нагрузочное тестирование;
- стрессовое тестирование.

Нагрузочные испытания:

1. Большое количество пользователей, одновременно посещающих сайт.
2. Работа системы при пиковых нагрузках.
3. Пользователь осуществляет доступ к большому количеству данных.

В реальных ситуациях такое может случаться при массовом всплеске запросов от пользователей, намного превышающем обычный уровень, например спровоцированном большим количеством ссылок на веб-сайт (представьте себе, что ваш веб-сайт упомянули на национальном телевидении). Целью стресс-тестов является изучение порогов нагрузки, при которых веб-сервер выдает ошибки, и того, сможет ли он самостоятельно вернуться к нормальной работе после наплыва запросов или произойдет отказ, и его придется возвращать к работе вручную. Проверьте производительность приложения на различной скорости Интернета.

Самым популярным инструментом нагрузочного тестирования является jMeter, который позволяет определить профили нагрузки и искусственно создать для них нагрузку, выявляющую граничные возможности приложения (или сервера) в условиях работы с ним того или иного количества пользователей.

Неотъемлемой частью комплексного тестирования веб-приложений является *тестирование безопасности*. Данное тестирование направлено на определение путей взлома системы, оценку защищенности веб-приложений, а также анализ рисков доступа к конфиденциальным данным.

Обязательны проверки на наличие уязвимостей согласно классификации OWASP (см. п. 4.3).

Также для автоматизации тестирования безопасности применяются специализированные средства – сканеры безопасности, например OWASP Live CD – специализированный набор инструментов для тестирования безопасности и логики работы веб-приложения.

Многие виды тестирования веб-приложений легко автоматизируются, некоторые требуют тщательного ручного тестирования. Но все они служат одной цели – сделать веб-приложения качественным, удобным, безопасным.

5.2 Особенности тестирования мобильных приложений

Трудно представить современную жизнь без смартфона и, соответственно, без мобильных приложений.

Мобильные приложения (*mobile apps*) *– это программы, которые предназначены для той или иной платформы* (*Android, iOS, Windows 10 Mobile, BlackBerry и многие другие*)*, написаны на языке высокого уровня и позволяют выполнять различные действия в зависимости от своего функционала.*

Мобильное приложение должно быть интуитивно понятным, удобным, работать бесперебойно, безопасно, круглосуточно и достаточно быстро реагировать на действия пользователя.

Неотъемлемой частью разработки таких приложений является их тестирование. Мобильное тестирование – сложный процесс: десятки различных разрешений экрана, аппаратные отличия, несколько версий операционных систем, разные типы подключения к Интернету, внезапные обрывы связи.

Итак, рассмотрим основные моменты и особенности тестирования мобильных приложений, на которые стоит обратить внимание при функциональном и GUI тестировании мобильных приложений.

Все мобильные приложения можно разделить на несколько категорий:

- *Мобильные веб-приложения* – кросс-платформенные приложения позволяют отображать сайты на различных устройствах.
- *Нативные приложения* разработаны только под определенную платформу и по максимуму используют возможность той или иной операционной системы.
- *Гибридные приложения* разрабатываются для обеих платформ одновременно и пишутся на универсальном языке.

Тестовые сценарии отличаются для всех этих типов, кроме того, поведение приложений тоже не совпадает. Наиболее востребованные из них:

- функциональное тестирование;
- внешние события (прерывания);
- тест юзабилити;
- тестирование установки, обновлений и дополнений;
- тестирование безопасности;

- тестирование локализации;
- тестирование производительности;
- тест версий для различных платформ.

Как и при тестировании любого программного обеспечения, в тестировании мобильных приложений используется *функциональное тестирование*, т. е. проверка того, что функционал, заложенный в приложение, работает в соответствии с предъявляемыми функциональными требованиями. Также возможна проверка поддержки горизонтального (*landscape*) и вертикального (*portrait*) положений.

При мобильном тестировании необходимо учитывать то, что мобильные устройства – это в первую очередь устройства для связи, они могут звонить, отправлять СМС, отключаться. И это не должно мешать работе, приложения и наоборот. При тестировании *прерывания* важно проверить работу в следующих ситуациях:

- наличие отличной постоянной связи;
- наличие постоянной неотличной связи;
- отсутствие связи;
- потеря связи;
- входящие/исходящие СМС, ММС, звонки;
- разряд/изъятие батареи;
- отключение сети/Wi-Fi;
- подключение кабеля, карты, зарядки.

Тестирование установки мобильного приложения является важной частью комплексного тестирования приложения. Необходимо тестировать чистую установку с нуля, установку поверх предыдущей версии, прерывания и отмену установки приложения. Также важно тестировать обновления: проверка различных путей установки обновлений, проверка работы установленных изменений, мест, куда они вносились; убедиться в поддерживаемости обновлений более старыми операционными системами, чтобы элементы, которые на новой системе работают хорошо, не падали на более старых версиях.

Тестирование локализации аналогично тестированию на веб-приложениях. Проверка локализации заключается в проверке интерфейса на другом языке на экране, для перевода должно хватить места для текста, даты должны соответствовать формату установленного региона, временные настройки должны быть соблюдены.

Приложение на мобильном устройстве должно быть удобным для использования, так же как и веб-приложение. Здесь очень важно обеспечить для пользователя максимальный комфорт при работе с приложением, который подразумевает выполнение следующих требований:

- быстрота работы;
- простота и понятность интерфейса;
- минимальный ввод данных с клавиатуры;
- наличие индикации (отклика) на действия пользователя.

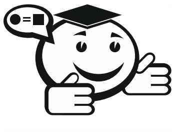
При *тестировании юзабилити* на мобильных устройствах используются те же средства и методы, что и при тестировании веб-приложений. При этом необходимо учитывать размеры экрана и touch-интерфейс, в котором должен быть удобный размер кнопок, чтобы не надо было искать ее на экране и делать много попыток попасть по ней, и высокая скорость отклика элементов (нажатая клавиша должна визуально отличаться).

В ходе *тестирования безопасности* приложение проверяется на наличие уязвимостей, устойчивость к взлому, возможности перехвата трафика с целью получения несанкционированного доступа к передаваемой приложением информации. В настоящее время данный вид тестирования очень важен для приложений, таких как мобильный банк, страховые приложения или мессенджеры. Сотрудники, занимающиеся тестированием безопасности, должны иметь специфические знания и навыки, которые отличаются от навыков, которыми обладают специалисты по другим видам тестирования.

Случайное тестирование (*fuzzy testing*, «*monkey*» *testing*) – тестирование, при котором приложение должно корректно реагировать на возникновение случайных и непредсказуемых событий. Мобильные устройства чаще других попадают в условия, в которых получают хаотичную бесполезную информацию (например, незаблокированный девайс в кармане), потому приложение должно адекватно реагировать на подобные потоки данных.

В мобильной автоматизации использовать реальные устройства – дорого и не всегда эффективно. Удаленные фермы устройств, вроде Browserstack, стоят достаточно дорого. Поддержание локальной фермы стоит еще дороже – администрирование парка устройств отнимает очень много времени.

В такой ситуации спасает тестирование на эмуляторах и симуляторах. Их установка и настройка не занимает много времени, в большинстве своем они бесплатны, а утилиты автоматизации давно умеют подключаться к ним самостоятельно.



Эмулятор (*emulator*) *– программа, полностью или частично копирующая функции и поведение устройства или другой программы.*

Эмуляция выполняет программный код в привычной для этого кода среде, состоящей из тех же компонентов, что и эмулируемый объект.

Некоторые из преимуществ использования эмулятора:

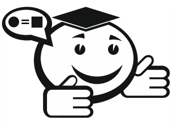
- оперативное тестирование приложения, когда целевой мобильный телефон недоступен (или оказывается в дефиците);
- тестирование сложных или опасных сценариев, которые невозможно или не рекомендуется проверять на реальных мобильных телефонах (например, тесты, которые каким-либо образом могут вывести телефон из строя или нарушить условия соглашения с оператором сотовой связи).

Минусы:

- зачастую эмуляторы очень требовательны к ресурсам, так как наиболее качественные из них эмулируют работу приложения с самых нижних уровней;
- то, что приложение работает на эмуляторе, не значит практически ничего, ведь пользователи будут запускать приложения на реальных мобильных телефонах, которые всегда отличаются даже от самых лучших эмуляторов.

Эмуляторы незаменимы при тестировании верстки и геолокации. При этом нужно понимать, что эмулятор никогда не заменит реальное устройство.

Для всех распространенных мобильных ОС предлагаются бесплатные (для разработчиков) и довольно функциональные эмуляторы. Например, для Android есть официальный Android SDK, который включает в себя эмулятор мобильного устройства, реализующий все аппаратные и программные особенности типичного устройства. Такие же «нативные» решения есть и для iOS, и для Windows Phone.

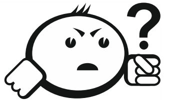


Симулятор (*simulator*) *– модель оригинального ПО, в которой реализуется логика работы (частично или полностью) и имитируется поведение ПО, копируется его интерфейс.*

Симуляция лишь имитирует выполнение кода, а не копирует его, все виртуально на 100%. Поэтому для тестирования чаще используют эмуляторы, т. к. симуляторы просто изображают окружение оригинального устройства и никак не затрагивают его начинку (железо), а ведь это может повлиять на результат.

Тестирование на целевом мобильном устройстве – это самый верный способ убедиться в правильном функционировании приложения, поскольку вы выполняете приложение на том же аппаратном обеспечении, которое будет у ваших пользователей.

Подводя итоги, следует сказать, что при тестировании мобильного приложения необходимо уделить особое внимание выбору парка устройств, на которых будет проводиться тестирование, учесть время на тестирование требований и API, а также осуществить полную качественную проверку поведения приложения в реальных условиях использования мобильного устройства. Самое же главное – обеспечить для пользователя простоту и удобство работы с данным мобильным приложением.



Контрольные вопросы по главе 5

1. Какие виды тестирования можно проводить для веб-приложений?
2. Назовите наиболее распространенные тесты мобильной разработки.
3. Что такое адаптивная верстка?
4. Что необходимо проверить при тестировании верстки?
5. Назовите ошибки при адаптации сайтов для мобильных устройств.

Заключение

Мир информационных технологий сегодня более динамичен, чем когда-либо. Тестирование актуально для всех IT-компаний, которые имеют дело с программным обеспечением. Автоматизация сегодня – это основной способ сокращения цикла тестирования, высвобождения ресурсов и повышения производительности. Количество запросов на автоматизацию тестирования растет с каждым годом, но и от ручного тестирования никуда не уйти. Исследовательское тестирование, тестирование юзабилити и интерфейса пользователя – вот только несколько примеров того, где без ручных проверок не обойтись. Поэтому в будущем все тестировщики должны будут обладать навыками ручного и автоматизированного тестирования.

Особенно востребованы тестировщики в разработке веб- и мобильных приложений, так как увеличивается количество пользователей и устройств. Мобильные и онлайн-игры сохраняют свою популярность, но с развитием AR (дополненной реальности) таких игр станет все больше, точно так же, как и других приложений, использующих данную технологию. А это еще одно направление со своими особенностями для тестировщика. Важно правильно организовать процесс тестирования, это позволит в кратчайшие сроки создать действительно качественный программный продукт.

Литература

1. History's worst software bugs [Электронный ресурс] // Сайт www.wired.com. – URL: https://www.wired.com/2005/11/historys-worst-software-bugs/#cdesc1 (дата обращения: 03.04.2019).
2. Самые ужасные баги в истории [Электронный ресурс] // Сайт software-testing.ru. – URL: http://www.software-testing.ru/library/testing/general-testing/2082-horrible-bugs (дата обращения: 03.04.2019).
3. IEEE Standard 610-90 (Standard Glossary of Software Engineering Terminology) [Электронный ресурс]. – URL: https://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf (дата обращения: 03.04.2019).
4. Майерс, Г. Надежность программного обеспечения / Г. Майерс. – М. : Мир, 1980. – 360 с.
5. Бейзер, Б. Тестирование черного ящика / Б. Бейзер. – СПб. : Питер, 2004. – 318 с.
6. Software Engineering Body of Knowledge (SWEBOK) // Сайт www.computer.org [Электронный ресурс]. – URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering (дата обращения: 03.04.2019).
7. The difference between Verification and Validation // Сайт Serendipity [Электронный ресурс]. – URL: http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/ (дата обращения: 03.04.2019).
8. Meyer, B. Seven principles of software testing / B. Meyer // IEEE Computer. – 2008. – № 41. – P. 99–101.
9. Савин, Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах / Р. Савин. – М. : Дело, 2007. – 312 с.
10. Перемитина, Т. О. Тестирование программного обеспечения : учеб. пособие / Т. О. Перемитина. – Томск : ФДО, ТУСУР, 2015. – 116 с.
11. Серия игр Civilization: весь мир – игра, а люди в ней – цифры в статистике [Электронный ресурс] // Сайт cubiq.ru. – URL: https://cubiq.ru/seriya-igr-civilization/ (дата обращения: 03.04.2019).

12. Agile-манифест разработки программного обеспечения [Электронный ресурс] // Сайт agilemanifesto.org. – URL: https://agilemanifesto.org/iso/ru/manifesto.html (дата обращения: 03.04.2019).
13. Технология разработки программного обеспечения : конспект лекции / сост. И. И. Савенко. – Томск : Изд-во Том. политехн. ун-та, 2014. – 67 с.
14. Инструменты для работы с тест-кейсами. Результаты опроса [Электронный ресурс] // Сайт software-testing.ru. – URL: http://www.software-testing.ru/library/testing/test-analysis/2326-test-case-management-tools (дата обращения: 03.04.2019).
15. Best Application Designs [Электронный ресурс] // Сайт www.nngroup.com. – URL: https://www.nngroup.com/articles/best-application-designs/ (дата обращения: 03.04.2019).
16. Интерактивная карта киберугроз [Электронный ресурс] // Сайт cybermap.kaspersky.com. – URL: https://cybermap.kaspersky.com/ru/ (дата обращения: 03.04.2019).
17. Тренды QA: обзор топ-10 тенденций на 2019 год [Электронный ресурс] // Сайт www.a1qa.ru. – URL: https://www.a1qa.ru/blog/trendy-qa-obzor-top-10-tendentsij-2019/ (дата обращения: 03.04.2019).
18. Digital in 2010 [Электронный ресурс] // Сайт We are social. – https://wearesocial.com/global-digital-report-2019 (дата обращения: 03.04.2019).

Глоссарий

Автоматизированное тестирование (*automated testing*) – вид тестирования, при котором используются специальные программные средства для выполнения тестов, сверки полученных фактических результатов с ожидаемыми значениями, установки предусловий для запуска тестов, а также для других функций, включая генерацию отчетов.

Алгоритм всех пар (*all-pairs algorithm*) – это комбинаторная методика, в основе которой лежит выбор возможных комбинаций значений всех переменных, в которых содержатся все возможные значения для каждой пары переменных.

Альфа-тестирование (*alpha testing*) – имитация реальной работы с системой штатными разработчиками либо реальная работа с системой потенциальными пользователями/заказчиком.

Атака на недочет (*fault attack*) – направленная и нацеленная попытка оценить качество, главным образом надежность, объекта тестирования за счет попыток вызвать определенные дефекты.

Бета-тестирование (*beta testing*) – в некоторых случаях выполняется распространение предварительной версии (иногда с ограничениями по функциональности или времени работы) для некоторой большой группы лиц, с тем чтобы убедиться, что продукт содержит достаточно мало ошибок.

Валидация (*validation*) – проверка того, что сам продукт правилен, т. е. установление того, что он удовлетворяет требованиям, ожиданиям пользователя, заказчика и других заинтересованных сторон.

Верификация (*verification*) – проверка того, что продукт делался правильно, т. е. проверка того, что он разрабатывался в соответствии со всеми требованиями по отношению к процессу и этапам разработки.

Дефект (*баг*, *bug*) – это отклонение фактического результата (*actual result*) от ожидаемого результата (*expected result*).

Динамическое тестирование (*dynamic testing*) – это тестовая деятельность, предусматривающая эксплуатацию (запуск) программного продукта.

Дымовое тестирование (*smoky testing*) – первый прогон программы (после написания или после внесения существенных изменений).

Жизненный цикл бага (*bug workflow*) – последовательность этапов, которые проходит баг на своем пути с момента его создания до окончательного закрытия.

Заглушка (*stub*) – это имитация вызываемой функции, возвращающая те же данные, но ничего больше не делающая.

Интеграционное тестирование (*integration testing*) предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием) либо связи между различными системами).

Исследовательское тестирование (*exploratory testing*) – тестирование, которое предполагает одновременное выполнение и разработку тестов, а также изучение продукта.

Качество программного обеспечения (*software quality*) – это способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям.

Контроль качества (*quality control*) – рабочие методы и активности, нацеленные на выполнение требований к качеству, являющиеся частью управления качеством.

Локализация бага (*localization bug*) – это процедура нахождения и описания условий, при которых дефект повторяется.

Метод белого ящика (*white box testing*, *open box testing*, *clear box testing*, *glass box testing*) – тестирование, при котором у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.

Метод черного ящика (*black box testing*, *closed box testing*, *specification-based testing*) – у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.

Модульное тестирование (*компонентное*, *юнит-тестирование*, *unit testing*) занимается поиском дефектов и верификацией функционирования программных модулей, программ, объектов, классов и т. п., которые можно протестировать изолированно.

Нагрузочное тестирование (*load testing*, *capacity testing*) – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).

Недочет (*fault*) – это дефект системы, при котором она все же может выполнять свои основные функции.

Нефункциональное тестирование (*non-functional testing*) – вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований), таких как удобство использования, совместимость, производительность, безопасность и т. д.

Обеспечение качества (*QA – quality assurance*) – это совокупность мероприятий, охватывающих все этапы разработки ПО, включая эксплуатацию и релиз, которые предпринимаются на разных стадиях жизненного цикла ПО, главной целью которого является обеспечение качества выпускаемого продукта.

Отказ (*failure*) – это неспособность системы выполнять свои функции.

Отчет о дефекте (*баг-репорт*, *bug report*) – это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Ошибка (*error*) – это дефект, заложенный в логике системы, в результате которого ПО не соответствует требованиям.

Покрытие (*coverage*) – уровень, выражаемый в процентах, на который определенный элемент покрытия был проверен набором тестов.

Покрытие альтернатив (*decision coverage*) – процент результатов альтернативы, который был проверен набором тестов. Стопроцентное покрытие решений подразумевает стопроцентное покрытие ветвей и операторов.

Покрытие операторов (*statement coverage*) – процентное отношение операторов, исполняемых набором тестов, к их общему количеству.

Попарное тестирование (*pairwise testing*) – это метод тестирования, основанный на том предположении, что большинство дефектов возникает при взаимодействии не более двух факторов. Тестовые наборы, генерируемые при использовании данной методики, охватывают все уникальные пары комбинаций факторов, что считается достаточным для обнаружения большего числа дефектов.

Предугадывание ошибки (*error guessing – EG*) – метод проектирования тестов, когда опыт тестировщика используется для предугадывания того, какие дефекты могут быть в тестируемом компоненте или системе в результате сделанных ошибок, а также для разработки тестов специально для их выявления.

Приоритет (*priority*) – это атрибут, указывающий на очередность выполнения задачи или устранения дефекта.

Регрессионное тестирование (*regression testing*) – тестирование уже протестированной программы, проводящееся после модификации для уверенности

в том, что процесс модификации не внес или не активизировал ошибки в областях, не подвергавшихся изменениям.

Ручное тестирование (*manual testing*) – это исполнение тестов без помощи каких-либо программ, автоматизирующих работу тестировщика.

Санитарное тестирование (*санити-тестирование*) (*sanity testing*) – это узконаправленное тестирование, достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям.

Серьезность (*severity*) – это атрибут, характеризующий влияние дефекта на работоспособность приложения.

Симулятор (*simulator*) – модель оригинального ПО, в которой реализуется логика работы (частично или полностью) и имитируется поведение ПО, копируется его интерфейс.

Системное тестирование (*system testing*) – процесс тестирования системы в целом с целью проверки того, что она соответствует установленным требованиям.

Статическое тестирование (*static testing*) – это процесс анализа самой разработки программного обеспечения, т. е. тестирование без запуска программы.

Стрессовое тестирование (*stress testing*) – исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчетный уровень, или в ситуациях недоступности значительной части необходимых приложению ресурсов.

Тест-дизайн (*test design*) – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы) в соответствии с определенными ранее критериями качества и целями тестирования.

Тестирование безопасности (*security and access control testing*) производится для оценки уязвимости программного обеспечения к различным вредоносным воздействиям на уровне приложения, операционной системы, сети и т. д.

Тестирование интернационализации (*internationalisation testing*) – вид тестирования, при котором проверяется готовность приложения к работе с различными языковыми настройками, в частности способность корректно отображать шрифты, пункты меню, производить поиск, сортировку; способность приложения обрабатывать файлы, поименованные на различных языках.

Тестирование локализации (*localization testing*) проверяет, насколько корректно продукт адаптирован к работе на том или ином языке: все ли переведено

и правильно ли, не нарушилась ли логика построения интерфейса и обработки данных и т. д.

Тестирование интерфейса пользователя (*GUI testing*) – это тестирование, при котором проверяются элементы интерфейса пользователя.

Тестирование совместимости (*compatibility testing*) – это вид тестирования, основной целью которого является проверка качественной работы разрабатываемого программного средства с другим ПО (окружением, операционными системами, браузерами и т. д.).

Тестовый сценарий или тестовая ситуация (*test case*) – набор входных значений, предусловий выполнения, ожидаемых результатов и постусловий выполнения, разработанный для определенной цели или тестового условия, таких как выполнение определенного пути программы или же для проверки соответствия определенному требованию.

Тестирование ПО (*software testing*) – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранном определенным образом.

Тестирование производительности (*performance testing*) – исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке.

Тестирование удобства использования (*usability testing*) выполняется с целью определения, насколько органично используется пользовательский интерфейс целевыми пользователями, т. е. проверяется интуитивность интерфейса.

Тестировщик (*software tester*) – специалист, который моделирует различные ситуации, которые могут возникнуть в процессе использования предмета тестирования, управляет выполнением программы, создает искусственные ситуации, наблюдает за поведением программы и сравнивает наблюдаемое поведение с ожидаемым, составляет отчет о проведенном тестировании, в котором должен быть указан анализ и причины возникших проблем.

Функциональное тестирование (*functional testing*) – тестирование, основанное на анализе спецификации функциональности компонента или системы.

Чек-лист (*check-list*) – набор идей тестов, описывающий, что должно быть протестировано.

Эмулятор (*emulator*) – программа, полностью или частично копирующая функционал и поведение устройства или другой программы.

Учебное издание

Юлия Викторовна Морозова

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Корректор А. Н. Миронова
Оригинал-макет Г. Д. Дурягиной

Подписано в печать 10.06.2019. Формат 60x84$^1/_{16}$.
Бумага офсетная. Гарнитура Times.
Усл. печ. л. 6,97.
Тираж 150 экз. Заказ № .

Издательство «Эль Контент»
634061, г. Томск, ул. Киевская, д. 57, оф. 27

ISBN 978-5-4332-0279-5