Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения
информационных технологий

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

*Допущено Министерством образования Республики Беларусь
в качестве учебного пособия для студентов учреждений
высшего образования по специальностям
«Программное обеспечение информационных технологий»,
«Инженерно-психологическое обеспечение информационных
технологий» и по направлениям специальностей
«Информационные системы и технологии
(в обеспечении промышленной безопасности)»,
«Информационные системы и технологии
(в бизнес-менеджменте)»,
«Профессиональное обучение (информатика)»*

Минск БГУИР 2019

УДК 004.415.53(076)
ББК 32.972.1я73
Т36

Авторы:

С. С. Куликов, Г. В. Данилова, О. Г. Смолякова, М. М. Меженная

Рецензенты:

кафедра информационных систем управления
Белорусского государственного университета
(протокол №1 от 29.08.2018);

доцент кафедры экономической информатики
учреждения образования «Белорусский государственный экономический университет» кандидат социологических наук, доцент З. В. Пунчик

Тестирование программного обеспечения : учеб. пособие / Т36 С. С. Куликов [и др.]. – Минск : БГУИР, 2019. – 276 с. : ил.
ISBN 978-985-543-462-8.

Содержит материалы к лекционным и практическим занятиям.

УДК 004.415.53(076)
ББК 32.972.1я73

ISBN 978-985-543-462-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время, характеризующееся повсеместным использованием цифровых технологий, особое внимание уделяется созданию качественных, а значит, привлекательных для конечного пользователя, конкурентоспособных и эффективных с точки зрения бизнеса программных продуктов. В этой связи неотъемлемым этапом разработки программного обеспечения (ПО) является **тестирование – процесс анализа программного средства и сопутствующей документации с целью поиска дефектов и повышения качества продукта.** В свою очередь знание принципов и практические навыки тестирования являются обязательными компетенциями специалиста в области информационных технологий.

Учебное пособие «Тестирование программного обеспечения» объединяет фундаментальные знания в области анализа, планирования, проведения тестовых испытаний и оценки качества программных приложений на всех стадиях их жизненного цикла.

В основу данного учебного пособия положен десятилетний опыт проведения тренингов для начинающих тестировщиков. За это время накопилось огромное количество вопросов от слушателей, и стали отчётливо видны типичные для многих начинающих проблемы и сложности. Обощение знаний и опыта по тестированию программного обеспечения поможет начинающим тестировщикам быстрее погрузиться в профессию и избежать многих досадных ошибок.

Материал будет полезен как «совсем начинающим», так и имеющим опыт в тестировании. Кроме того, данное учебное пособие – своего рода «карта», в которой есть ссылки на множество внешних источников информации (которые могут оказаться полезными даже опытным тестировщикам), а также много примеров с пояснениями. Оригинальные (англоязычные) определения приведены в сносках.

Прежде чем приступить к изучению основного материала, ознакомьтесь с условными обозначениями.

	Определения и иная важная для запоминания информация. Часто будет встречаться рядом с расположенным далее знаком.
	Дополнительные сведения или отсылка к соответствующим источникам. Всё то, что полезно знать.
	Предостережения и частые ошибки. Недостаточно показать, «как правильно», часто большую пользу приносят примеры того, как поступать не стоит.
	Задания для самостоятельной проработки. Настоятельно рекомендуется выполнять их (даже если вам кажется, что всё очень просто).

1 ПРОЦЕССЫ ТЕСТИРОВАНИЯ И РАЗРАБОТКИ ПО

1.1 Модели разработки ПО

Чтобы лучше разобраться в том, как тестирование соотносится с программированием и иными видами проектной деятельности, для начала рассмотрим саму основу – модели разработки (lifecycle model[1]) ПО (как часть жизненного цикла (software lifecycle[2]) ПО). При этом сразу подчеркнём, что разработка ПО является лишь частью жизненного цикла ПО, и здесь мы говорим именно о **разработке**.

Материал данного раздела относится скорее к дисциплине «управление проектами», поэтому он приведён сжато; не следует воспринимать его как исчерпывающее руководство.

Модель разработки ПО (Software Development Model, SDM) – структура, систематизирующая различные виды проектной деятельности, их взаимодействие и последовательность в процессе разработки ПО. Выбор той или иной модели зависит от масштаба и сложности проекта, предметной области, доступных ресурсов и множества других факторов.

Выбор модели разработки ПО серьёзно влияет на процесс тестирования, определяя выбор стратегии, расписание, необходимые ресурсы и т. д.

Моделей разработки ПО много, но классическими можно считать водопадную, v-образную, итерационную инкрементальную, спиральную и гибкую.

Перечень моделей разработки ПО (с кратким описанием), рекомендуемых к изучению тестировщиками, можно найти в статье «What are the Software Development Models?»[3]

Знать и разбираться в моделях разработки ПО тестировщику необходимо затем, чтобы уже с первых дней работы понимать, что происходит вокруг, что, зачем и почему вы делаете. Многие начинающие тестировщики отмечают, что ощущают бессмысленность происходящего, даже если текущие задания интересны. Чем более полным будет ваше представление картины происходящего в проекте, тем яснее для вас будет ваш собственный вклад в общее дело и смысл того, чем вы занимаетесь.

Ещё одна важная вещь, которую следует понимать, состоит в том, что никакая модель не является догмой или универсальным решением. Нет идеальной модели – есть та, которая лучше или хуже подходит для конкретного проекта, конкретной команды, конкретных условий.

[1] **Lifecycle model.** A partitioning of the life of a product or project into phases. ISTQB Glossary.

[2] **Software lifecycle.** The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note these phases may overlap or be performed iteratively. ISTQB Glossary.

[3] What are the Software Development Models? URL: http://istqbexamcertification.com/what-are-the-software-development-models/.

Часто встречающаяся ошибка! Единственное, от чего стоит предостеречь заранее, так это от фривольной трактовки модели и перекраивания её «на свой вкус» без чёткого понимания, что и зачем вы делаете. Об ошибках, которые бывают при нарушении логики модели, прекрасно сказал в своём слайдкасте «Scrum Tailoring»[4] Максим Дорофеев.

Водопадная модель (waterfall model[5]) сейчас представляет скорее исторический интерес, т. к. в современных проектах практически неприменима. Она предполагает однократное выполнение каждой из фаз проекта, которые, в свою очередь, строго следуют друг за другом (рисунок 1. а). Очень упрощённо можно сказать, что в рамках этой модели в любой момент времени команде «видны» лишь предыдущая и следующая фазы. В реальной же разработке ПО необходимо «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.

К недостаткам водопадной модели принято относить тот факт, что участие пользователей ПО в ней либо не предусмотрено вообще, либо предусмотрено лишь косвенно на стадии однократного сбора требований. С точки зрения же тестирования эта модель плоха тем, что тестирование в явном виде появляется здесь лишь с середины развития проекта, достигая своего максимума в самом конце.

Тем не менее водопадная модель часто интуитивно применяется при выполнении относительно простых задач, а её недостатки послужили отправным пунктом для создания новых моделей. Также эта модель в усовершенствованном виде используется на крупных проектах, в которых требования очень стабильны и могут быть хорошо сформулированы в начале проекта (аэрокосмическая область, медицинское ПО и т. д.).

Относительно краткое и действительно хорошее описание водопадной модели можно найти в статье «What is Waterfall model advantages, disadvantages and when to use it?»[6].
Отличное описание истории развития и заката водопадной модели было создано Максимом Дорофеевым в виде слайдкаста «The Rise And Fall Of Waterfall», который можно посмотреть[7] в его ЖЖ.

[4] Дорофеев М. Scrum Tailoring. URL: http://cartmendum.livejournal.com/10862.html.
[5] In a **waterfall model**, each phase must be completed fully before the next phase can begin. This type of model is basically used for the project which is small and there are no uncertain requirements. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. URL: http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/.
[6] What is Waterfall model advantages, disadvantages and when to use it? URL: http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/.
[7] ЖЖ Максима Дорофеева. URL: http://cartmendum.livejournal.com/44064.html.

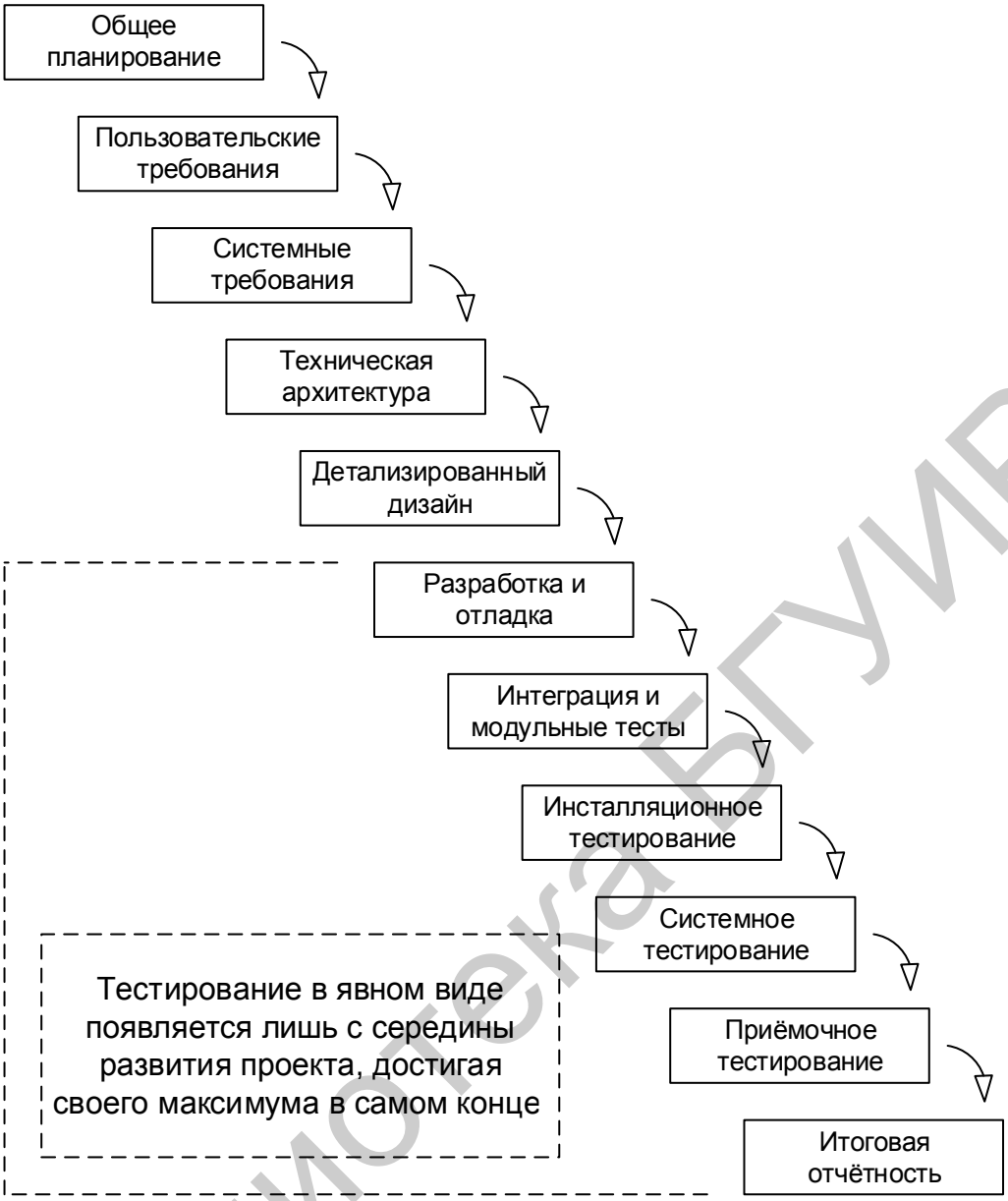


Рисунок 1.a – Водопадная модель разработки ПО

V-образная модель (V-model[8]) является логическим развитием водопадной. Можно заметить (рисунок 1.b), что в общем случае как водопадная, так и v-образная модели жизненного цикла ПО могут содержать один и тот же набор стадий, но принципиальное отличие заключается в том, как эта информация используется в процессе реализации проекта.

Очень упрощённо можно сказать, что при использовании v-образной модели на каждой стадии «на спуске» нужно думать о том, что и как будет происходить на соответствующей стадии «на подъёме». 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.

[8] **V-model.** A framework to describe the software development lifecycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development lifecycle. ISTQB Glossary.

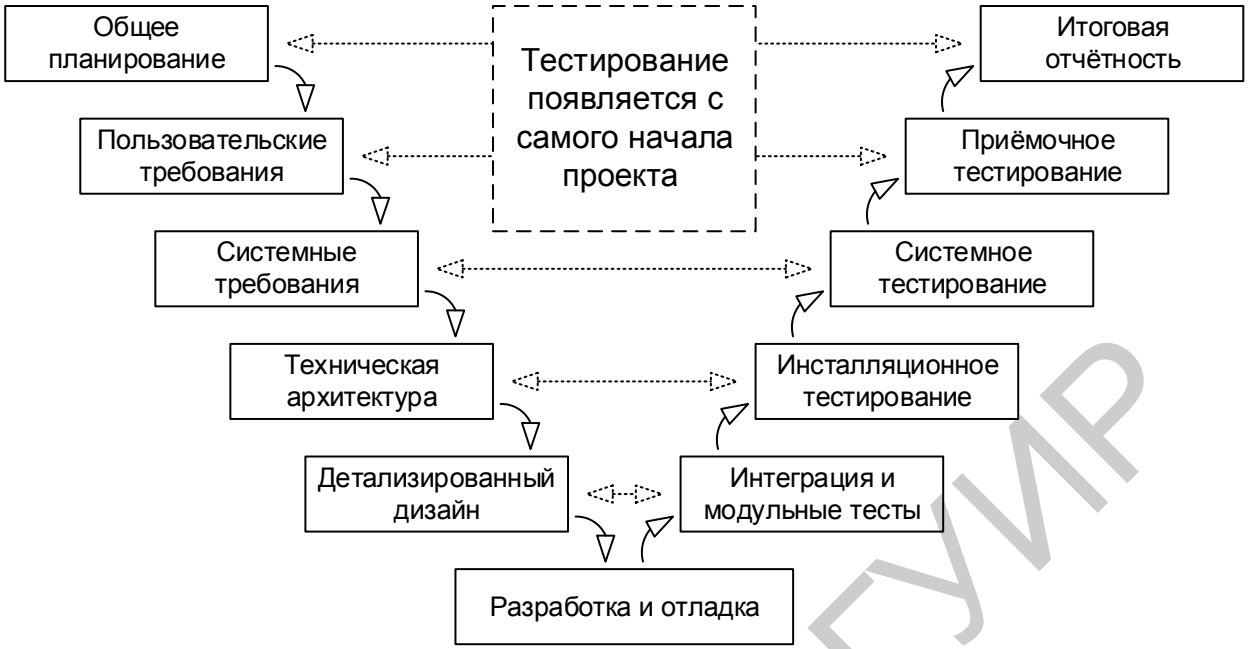


Рисунок 1.b – V-образная модель разработки ПО

Краткое описание v-образной модели можно найти в статье «What is V-model advantages, disadvantages and when to use it?»[9]. С пояснениями к использованию v-образной модели в тестировании можно ознакомиться в статье «Using V Models for Testing»[10].

Итерационная инкрементальная модель (iterative model[11], incremental model[12]) является фундаментальной основой современного подхода к разработке ПО. Как следует из названия модели, ей свойственна определённая двойственность (а ISTQB-глоссарий даже не приводит единого определения, разбивая его на отдельные части):

- с точки зрения жизненного цикла модель является **итерационной**, т. к. подразумевает многократное повторение одних и тех же стадий;
- с точки зрения развития продукта (приращения его полезных функций) модель является **инкрементальной**.

Ключевой особенностью данной модели является разбивка проекта на относительно небольшие промежутки (итерации), каждый из которых в общем случае может включать в себя все классические стадии, характерные водопадной и

[9] What is V-model advantages, disadvantages and when to use it? URL: http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/.
[10] Firesmith D. Using V-Models for Testing. URL: http://blog.sei.cmu.edu/post.cfm/using-v-models-testing-315.
[11] **Iterative development model.** A development lifecycle where a project is broken into a usually large number of iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product. ISTQB Glossary.
[12] **Incremental development model.** A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a 'mini V-model' with its own design, coding and testing phases. ISTQB Glossary.

v-образной моделям (рисунок 1.c). Итогом итерации является приращение (инкремент) функциональности продукта, выраженное в промежуточном билде (build[13]).

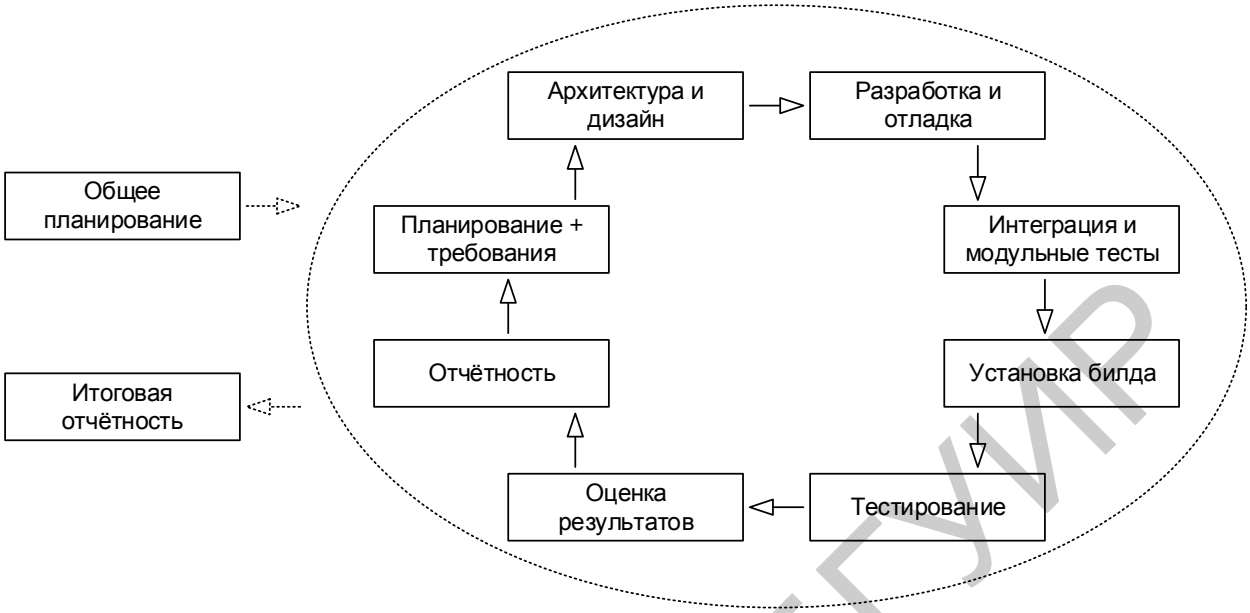


Рисунок 1.c – Итерационная инкрементальная модель разработки ПО

Длина итераций может меняться в зависимости от множества факторов, однако сам принцип многократного повторения позволяет гарантировать, что и тестирование, и демонстрация продукта конечному заказчику (с получением обратной связи) будут активно применяться с самого начала и на протяжении всего времени разработки проекта.

Во многих случаях допускается распараллеливание отдельных стадий внутри итерации и активная доработка с целью устранения недостатков, обнаруженных на любой из (предыдущих) стадий.

Итерационная инкрементальная модель очень хорошо зарекомендовала себя в объёмных и сложных проектах, выполняемых большими командами на протяжении длительных сроков.

Относительно краткие и содержательные описания итерационной инкрементальной модели можно найти в статьях «What is Iterative model advantages, disadvantages and when to use it?»[14] и «What is Incremental model advantages, disadvantages and when to use it?»[15]

[13] **Build.** A development activity whereby a complete system is compiled and linked, so that a consistent system is available including all latest changes. На основе определения термина «daily build» из ISTQB Glossary.

[14] What is Iterative model advantages, disadvantages and when to use it? URL: http://istqbexamcertification.com/what-is-iterative-model-advantages-disadvantages-and-when-to-use-it/.

[15] What is Incremental model advantages, disadvantages and when to use it? URL: http://istqbexamcertification.com/what-is-incremental-model-advantages-disadvantages-and-when-to-use-it/.

Спиральная модель (spiral model[16]) представляет собой частный случай итерационной инкрементальной модели, в котором особое внимание уделяется управлению рисками, в особенности влияющими на организацию процесса разработки проекта и контрольные точки.

Схематично суть спиральной модели представлена на рисунке 1.d. Обратите внимание на то, что здесь явно выделены четыре ключевые фазы:

- проработка целей, альтернатив и ограничений;
- анализ рисков и прототипирование;
- разработка (промежуточной версии) продукта;
- планирование следующего цикла.

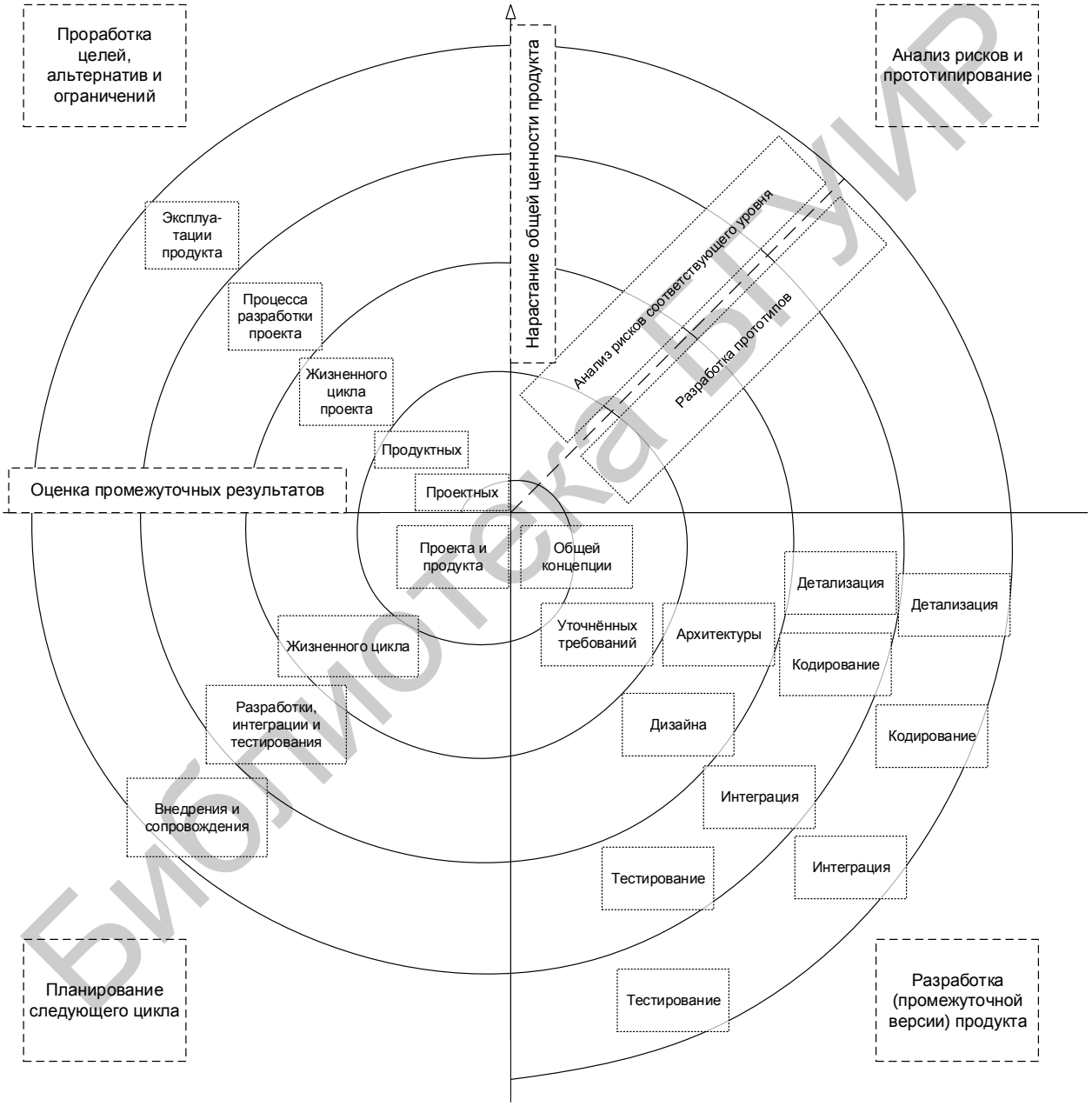


Рисунок 1.d – Спиральная модель разработки ПО

[16] **Spiral model.** A software lifecycle model which supposes incremental development, using the waterfall model for each step, with the aim of managing risk. In the spiral model, developers define and implement features in order of decreasing priority. URL: http://dictionary.reference.com/browse/spiral+model.

С точки зрения тестирования и управления качеством повышенное внимание рискам является ощутимым преимуществом при использовании спиральной модели для разработки концептуальных проектов, в которых требования естественным образом являются сложными и нестабильными (могут многократно меняться по ходу выполнения проекта).

Автор модели Barry Boehm в своих публикациях[17, 18] подробно раскрывает эти вопросы и приводит множество рассуждений и рекомендаций, как применять спиральную модель с максимальным эффектом.

Относительно краткие и очень хорошие описания спиральной модели можно найти в статьях «What is Spiral model- advantages, disadvantages and when to use it?»[19] и «Spiral Model»[20].

Гибкая модель (agile model[21]) представляет собой совокупность различных подходов к разработке ПО и базируется на так называемом «agile-манифесте»[22]:

- Люди и взаимодействие важнее процессов и инструментов.
- Работающий продукт важнее исчерпывающей документации.
- Сотрудничество с заказчиком важнее согласования условий контракта.
- Готовность к изменениям важнее следования первоначальному плану.

Данная тема является настолько обширной, что знаний со статей по ссылкам явно недостаточно, по этой причине стоит ознакомиться с книгами:

- Crispin L., Gregory J. « Agile Testing»;
- Rubin K. S. «Essential Scrum».

Несложно догадаться, что положенные в основу гибкой модели подходы являются логическим развитием и продолжением созданных и опробованных за десятилетия разработок в водопадной, v-образной, итерационной инкрементальной, спиральной и иных моделях. Причём впервые был достигнут ощутимый результат в снижении бюрократической составляющей и максимальной адаптации процесса разработки ПО к мгновенным изменениям рынка и требованиям заказчика.

Если описывать поверхностно, то можно сказать, что гибкая модель представляет собой облегчённую с точки зрения документации версию итерационной инкрементальной и спиральной моделей (рисунок 1.е[23]); при этом следует помнить об «agile-манифесте» и всех вытекающих из него преимуществах и недостатках.

Главным недостатком гибкой модели считается сложность её применения к крупным проектам, а также частое ошибочное внедрение её подходов, вызванное недопониманием фундаментальных принципов модели.

---

[17] Boehm B. A Spiral Model of Software Development and Enhancement. URL: http://csse.usc.edu/csse/TECHRPTS/1988/usccse88-500/usccse88-500.pdf.

[18] Boehm B. Spiral Development: Experience, Principles, and Refinements. URL: http://www.sei.cmu.edu/reports/00sr008.pdf.

[19] What is Spiral model- advantages, disadvantages and when to use it? URL: http://istqbexamcertification.com/what-is-spiral-model-advantages-disadvantages-and-when-to-use-it/.

[20] Ghahrai A. Spiral Model. URL: http://www.testingexcellence.com/spiral-model/.

[21] **Agile software development.** A group of software development methodologies based on EITP iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. ISTQB Glossary.

[22] Agile-манифест. URL: http://agilemanifesto.org/iso/ru/manifesto.html.

[23] Оригинал рисунка и исходный материал. URL: http://www.gtagile.com/GT_Agile/Home.html.

Тем не менее можно утверждать, что количество проектов, использующих гибкую модель разработки, становится всё больше.

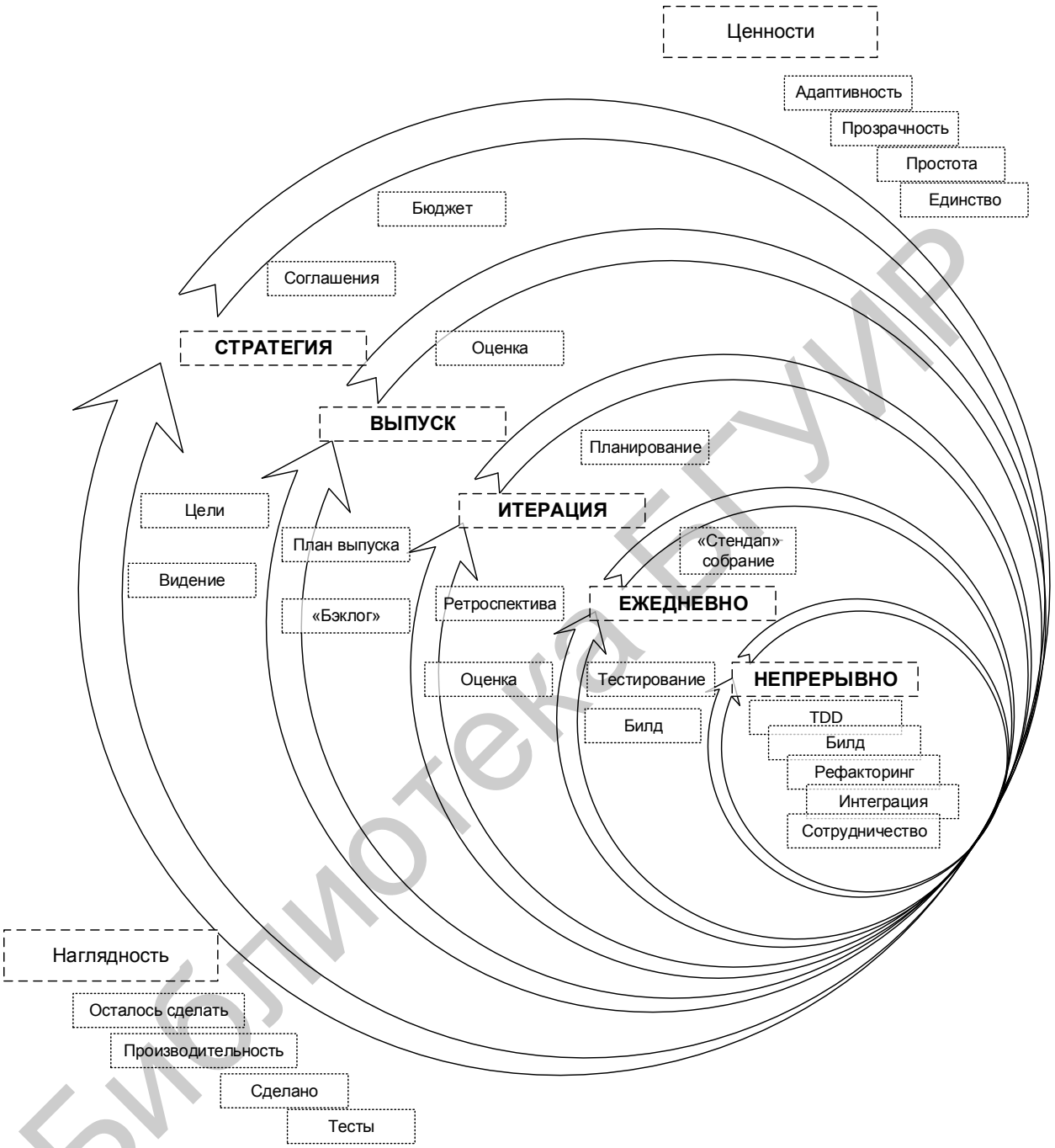


Рисунок 1.е – Суть гибкой модели разработки ПО

	Очень подробное и «элегантное» изложение принципов применения гибкой модели разработки ПО можно найти в статье «The Agile System Development Life Cycle»[24].

Кратко суть моделей разработки ПО отражена в таблице 1.а.

[24] The Agile System Development Life Cycle. URL: http://www.ambysoft.com/essays/agileLifecycle.html.

Таблица 1.а – Сравнительная характеристика моделей разработки ПО

Модель	Преимущества	Недостатки	Тестирование
Водопадная	• У каждой стадии есть чёткий проверяемый результат. • В каждый момент времени команда выполняет один вид работы. • Хорошо работает для небольших задач	• Полная неспособность адаптировать проект к изменениям в требованиях. • Крайне позднее создание работающего продукта	• С середины проекта
V-образная	• У каждой стадии есть чёткий проверяемый результат. • Внимание тестированию уделяется с первой же стадии. • Хорошо работает для проектов со стабильными требованиями	• Недостаточная гибкость и адаптируемость. • Отсутствует раннее прототипирование. • Сложность устранения проблем, пропущенных на ранних стадиях развития проекта	• На переходах между стадиями
Итерационная инкрементальная	• Достаточно раннее прототипирование. • Простота управления итерациями. • Декомпозиция проекта на управляемые итерации	• Недостаточная гибкость внутри итераций. • Сложность устранения проблем, пропущенных на ранних стадиях развития проекта	• В определённые моменты итераций. • Повторное тестирование (после доработки) уже проверенного ранее
Спиральная	• Глубокий анализ рисков. • Подходит для крупных проектов. • Достаточно раннее прототипирование	• Высокие накладные расходы. • Сложность применения для небольших проектов. • Высокая зависимость успеха от качества анализа рисков	
Гибкая	• Максимальное вовлечение заказчика. • Большой объём работы с требованиями. • Тесная интеграция тестирования и разработки. • Минимизация документации	• Сложность реализации для больших проектов. • Сложность построения стабильных процессов	• В определённые моменты итераций и в любой необходимый момент

Ещё два кратких и информативных сравнения моделей жизненного цикла ПО можно найти в статьях «Project Lifecycle Models: How They Differ and When to Use Them»[25] и «Блок-схема выбора оптимальной методологии разработки ПО»[26]. А общий обзор всех моделей в контексте тестирования ПО представлен в статье «What are the Software Development Models?»[27].

Задание 1.а: Представьте, что на собеседовании вас попросили назвать основные модели разработки ПО, перечислить их преимущества и недостатки с точки зрения тестирования. Ответьте на этот вопрос сейчас, а ответ запишите.

1.2 Жизненный цикл тестирования

Следуя общей логике итеративности, превалирующей во всех современных моделях разработки ПО, жизненный цикл тестирования также выражается замкнутой последовательностью действий (рисунок 1.g).

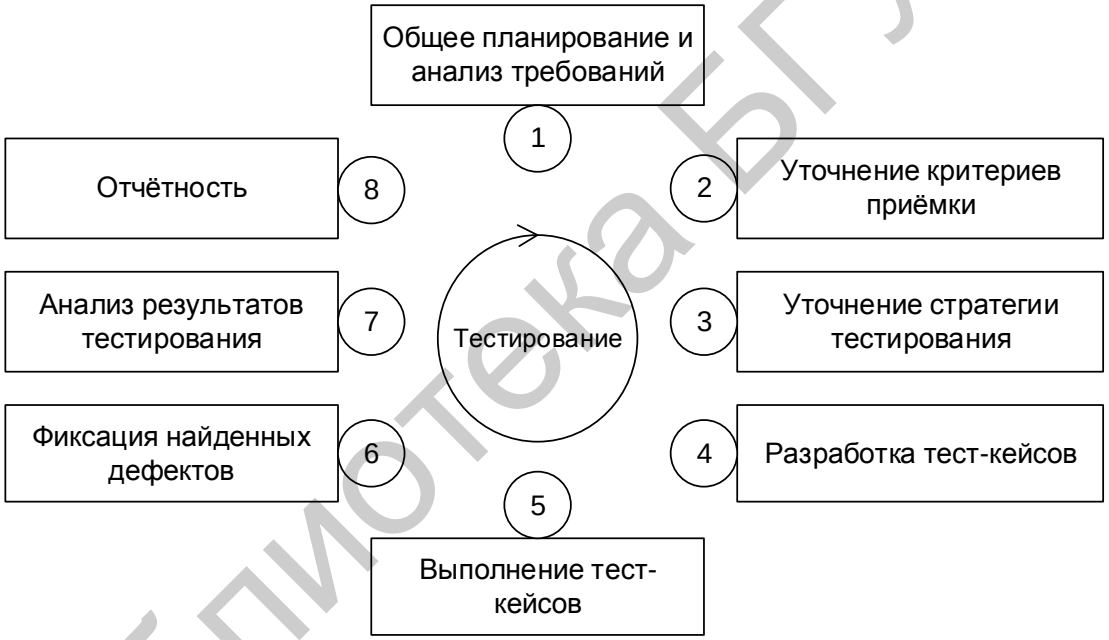


Рисунок 1.g – Жизненный цикл тестирования

Важно понимать, что длина такой итерации может варьироваться в широком диапазоне – от единиц часов до десятков месяцев (и, соответственно, подробности каждой стадии вариативны). Как правило, если речь идёт о длительном промежутке времени, он разбивается на множество относительно коротких итераций, но сам при этом «тяготеет» к той или иной стадии в каждый момент времени (например, в начале проекта больше планирования, в конце – больше отчётности).

Ещё раз подчеркнём, что приведённая схема – не догма, и вы легко можете

[25] Project Lifecycle Models: How They Differ and When to Use Them. URL: http://www.business-esolutions.com/islm.html.
[26] Блок-схема выбора оптимальной методологии разработки ПО. URL: http://megamozg.ru/post/23022/.
[27] What are the Software Development Models? URL: http://istqbexamcertification.com/what-are-the-software-development-models/.

найти альтернативу (например, по ссылкам[28,29]), но общая суть и ключевые принципы остаются неизменными. Их и рассмотрим.

Стадия 1 – общее планирование и анализ требований – объективно необходима как минимум для того, чтобы иметь ответ на такие вопросы, как: что нам предстоит тестировать; как много будет работы; какие есть сложности; всё ли необходимое у нас есть и т. п. Как правило, получить ответы на эти вопросы невозможно без анализа требований, т. к. именно требования являются первичным источником ответов.

Стадия 2 – уточнение критериев приёмки – позволяет сформулировать или уточнить метрики и признаки возможности или необходимости начала тестирования (entry criteria[30]), приостановки (suspension criteria[31]) и возобновления (resumption criteria[32]) тестирования, завершения или прекращения тестирования (exit criteria[33]).

Стадия 3 – уточнение стратегии тестирования – представляет собой ещё одно обращение к планированию, но уже на локальном уровне: рассматриваются и уточняются те части стратегии тестирования (test strategy[34]), которые актуальны для текущей итерации.

Стадия 4 – разработка тест-кейсов – посвящена разработке, пересмотру, уточнению, доработке, переработке и прочим действиям с тест-кейсами, наборами тест-кейсов, тестовыми сценариями и иными артефактами, которые будут использоваться при непосредственном выполнении тестирования.

Стадия 5 – выполнение тест-кейсов – и **стадия 6** – фиксация найденных дефектов – тесно связаны между собой и фактически выполняются параллельно: дефекты фиксируются сразу по факту их обнаружения в процессе выполнения тест-кейсов. Однако зачастую после выполнения всех тест-кейсов и написания всех отчётов о найденных дефектах проводится явно выделенная стадия уточнения, на которой все отчёты о дефектах рассматриваются повторно с целью формирования единого понимания проблемы и уточнения таких характеристик дефекта, как важность и срочность.

Стадия 7 – анализ результатов тестирования – и **стадия 8** – отчётность – также тесно связаны между собой и выполняются практически параллельно. Формулируемые на стадии анализа результатов выводы напрямую зависят от плана тестирования, критериев приёмки и уточнённой стратегии, полученных на стадиях 1, 2 и 3. Полученные выводы оформляются на стадии 8 и служат основой для стадий 1, 2 и 3 следующей итерации тестирования. Таким образом, цикл замыкается.

---

[28] Software Testing Life Cycle. URL: http://softwaretestingfundamentals.com/software-testing-life-cycle/.

[29] Software Testing Life Cycle. URL: http://www.softwaretestingmentor.com/stlc/software-test-life-cycle/.

[30] **Entry criteria.** The set of generic and specific conditions for permitting a process to go forward with a defined task, e.g. test phase. The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria. ISTQB Glossary.

[31] **Suspension criteria.** The criteria used to (temporarily) stop all or a portion of the testing activities on the test items. ISTQB Glossary.

[32] **Resumption criteria.** The criteria used to restart all or a portion of the testing activities that were suspended previously. ISTQB Glossary.

[33] **Exit criteria.** The set of generic and specific conditions, agreed upon with the stakeholders for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing. ISTQB Glossary.

[34] **Test strategy.** A high-level description of the test levels to be performed and the testing within those levels for an organization or program (one or more projects). ISTQB Glossary.

В жизненном цикле тестирования пять из восьми стадий так или иначе связаны с управлением проектами. Материал по планированию и отчётности кратко изложен в шестом разделе данного учебного пособия «Оценка трудозатрат, планирование и отчётность». А сейчас мы переходим к ключевым навыкам и основным видам деятельности тестировщиков и начнём с работы с документацией.

1.3 Контрольные вопросы и задания

- Что такое модель разработки ПО?
- Как модель разработки ПО связана с жизненным циклом ПО?
- Перечислите модели разработки ПО.
- Охарактеризуйте каждую из моделей разработки ПО.
- Какие факторы влияют на выбор модели разработки ПО?
- Сделайте сравнительный анализ моделей разработки.
- Назовите основную цель тестирования.
- Какие этапы включает в себя жизненный цикл тестирования?
- Охарактеризуйте каждый этап.
- На каком этапе появляется тестирование в явном виде?
- Приведите своё описание жизненного цикла ПО.

2 ТЕСТИРОВАНИЕ ДОКУМЕНТАЦИИ И ТРЕБОВАНИЙ

2.1 Что такое «требование»

Мы убедились в первом разделе, посвящённом жизненному циклу тестирования, что работа по тестированию ПО начинается с документации и требований.

!!!	**Требование** (requirement[35]) – описание того, какие функции и с соблюдением каких условий приложение должно выполнять в процессе решения полезной для пользователя задачи.

....	Небольшое «историческое отступление»: если ознакомиться с определениями требований в литературе десяти-, двадцати-, тридцатилетней давности, то можно заметить, что изначально о пользователях, их задачах и полезных для них свойствах приложения в определении «требование» не было сказано. Пользователь выступал некоей абстрактной фигурой, не имеющей отношения к приложению. В настоящее время такой подход недопустим, т. к. он не только приводит к коммерческому провалу продукта на рынке, но и многократно повышает затраты на разработку и тестирование.

....	Хорошим кратким предисловием ко всему тому, что будет рассмотрено в данном разделе, можно считать небольшую статью «What is documentation testing in software testing»[36].

2.2 Важность требований

Требования являются отправной точкой для определения того, что проектная команда будет проектировать, реализовывать и тестировать. Элементарная логика заключается в том, что если в требованиях что-то «не то», то и реализовано будет «не то», т. е. колоссальная работа множества людей будет выполнена впустую. Эту мысль иллюстрирует рисунок 2.а.

Брайан Хэнкс, описывая важность требований[37], подчёркивает в них следующее:

- Позволяют понять, что и с соблюдением каких условий система должна делать.
- Предоставляют возможность оценить масштаб изменений и управлять изменениями.
- Являются основой для формирования плана проекта (в том числе плана тестирования).
- Помогают предотвращать или разрешать конфликтные ситуации.
- Упрощают расстановку приоритетов в наборе задач.
- Позволяют объективно оценить степень прогресса в разработке проекта.

[35] **Requirement.** A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. ISTQB glossary.

[36] What is documentation testing in software testing. URL: http://istqbexamcertification.com/what-is-documentation-testing/.

[37] Hanks B. Requirements in the Real World. URL: https://classes .soe.ucsc.edu/cmps109/Winter 02/notes/ requirementsLecture.pdf.

Вне зависимости от того, какая модель разработки ПО используется на проекте, чем позже будет обнаружена проблема, тем сложнее и дороже будет её решение. А в самом начале («водопада», «спуска по букве v», «итерации», «витка спирали») идёт планирование и работа с требованиями.

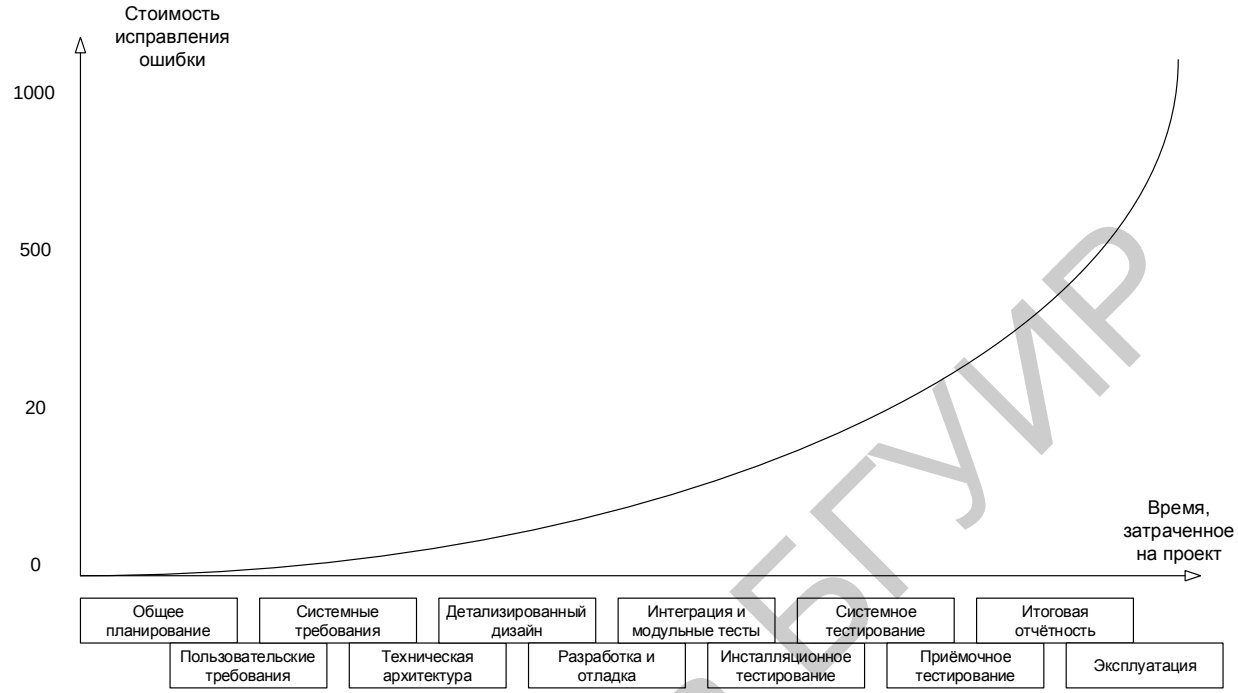


Рисунок 2.а – Стоимость исправления ошибки в зависимости от момента её обнаружения

Если проблема в требованиях будет выяснена на этой стадии, её решение может свестись к исправлению пары слов в тексте, в то время как недоработка, вызванная пропущенной проблемой в требованиях и обнаруженная на стадии эксплуатации, может даже полностью уничтожить проект.

Если графики вас не убеждают, попробуем проиллюстрировать ту же мысль на простом примере. Допустим, вы с друзьями составляете список покупок перед поездкой в гипермаркет. Вы поедете покупать, а друзья ждут вас дома. Сколько «стоит» дописать, вычеркнуть или изменить пару пунктов, пока вы только-только составляете список? Нисколько. Если мысль о несовершенстве списка настигла вас по пути в гипермаркет, уже придётся звонить (дёшево, но небесплатно). Если вы поняли, что в списке «что-то не то» в очереди на кассу, придётся возвращаться в торговый зал и тратить время. Если проблема выяснилась по пути домой или даже дома, придётся вернуться в гипермаркет. И, наконец, «клинический» случай: в списке изначально было что-то совсем искажённое (например, «100 кг конфет – и всё»), поездка совершена, все деньги потрачены, конфеты привезены, и только тут выясняется, что это была шутка.

Задание 2.а: Представьте, что ваш с друзьями бюджет ограничен, и в списке требований появляются приоритеты (что-то купить надо обязательно, что-то, если останутся деньги, и т. п.). Как это повлияет на риски, связанные с ошибками в списке?

Ещё одним аргументом в пользу тестирования требований является то, что, по разным оценкам, в них зарождается от ½ до ¾ всех проблем с программным обеспе-

чением. В итоге есть риск, что проект получится несоответствующим действительности, как показано на рисунке 2.b.

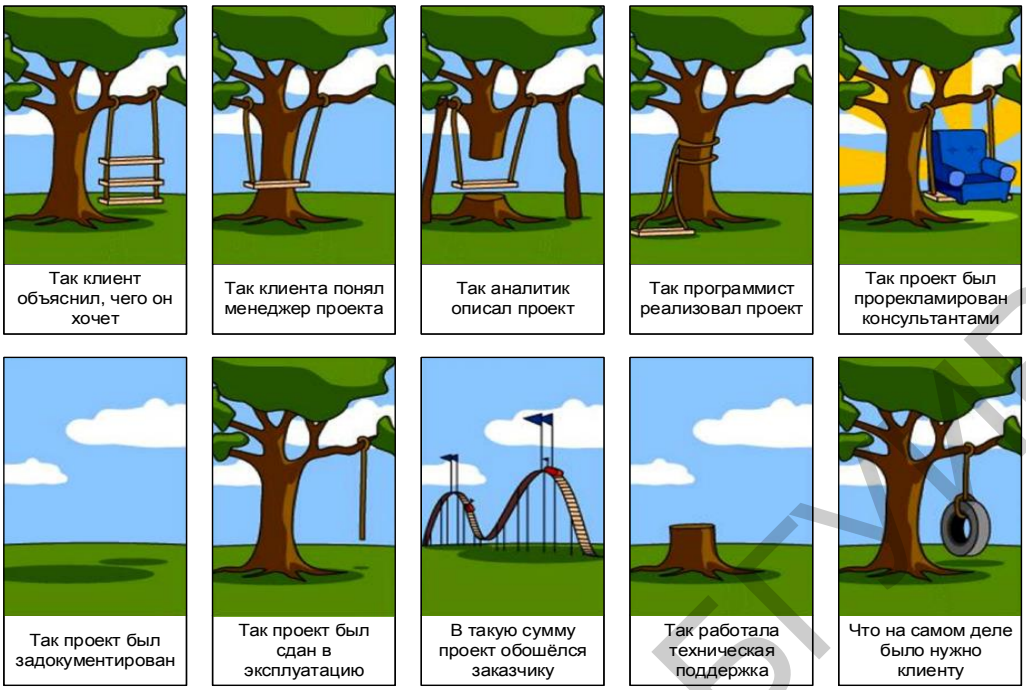


Рисунок 2.b – Типичный проект с плохими требованиями

Поскольку мы постоянно употребляем выражение «документация и требования», а не просто «требования», то стоит рассмотреть перечень документации, которая должна подвергаться тестированию в процессе разработки ПО (хотя далее мы будем концентрироваться именно на требованиях).

В общем случае документацию можно разделить на два больших вида в зависимости от времени и места её использования (здесь приведём сноски с определениями, т. к. по видам документации очень часто возникает множество вопросов и потому придётся рассмотреть некоторые моменты подробнее).

- **Продуктная документация** (product documentation, development documentation[38]) используется проектной командой во время разработки и поддержки продукта. Она включает:
  - план проекта (project management plan[39]), и в том числе тестовый план (test plan[40]);

[38] Development documentation. Development documentation comprises those documents that propose, specify, plan, review, test, and implement the products of development teams in the software industry. Development documents include proposals, user or customer requirements description, test and review reports (suggesting product improvements), and self-reflective documents written by team members, analyzing the process from their perspective. Barker T. Documentation for software and IS development.

[39] Project management plan. A formal, approved document that defines how the project is executed, monitored and controlled. It may be summary or detailed and may be composed of one of more subsidiary management plans and other planning documents. PMBOK. – 3rd ed.

[40] Test plan. A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process. ISTQB Glossary.

- требования к программному продукту (product requirements document, PRD[41]) и функциональные спецификации (functional specifications[42] document, FSD[43]; software requirements specification, SRS[44]);
- архитектуру и дизайн (architecture and design[45]);
- тест-кейсы и наборы тест-кейсов (test cases[46], test suites[47]);
- технические спецификации (technical specifications[48]), такие как схемы баз данных, описания алгоритмов, интерфейсов и т. д.

- **Проектная документация** (project documentation[49]) включает в себя как продуктную документацию, так и некоторые дополнительные виды документации и используется не только на стадии разработки, но и на более ранних и поздних стадиях (например, на стадии внедрения и эксплуатации). Она включает:
  - пользовательскую и сопроводительную документацию (user and accom-

[41] Product requirements document, PRD. The PRD describes the product your company will build. It drives the efforts of the entire product team and the company's sales, marketing and customer support efforts. The purpose of the product requirements document (PRD) or product spec is to clearly and unambiguously articulate the product's purpose, features, functionality, and behavior. The product team will use this specification to actually build and test the product, so it needs to be complete enough to provide them the information they need to do their jobs. Cagan M. How to write a good PRD.

[42] Specification. A document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system, and, often, the procedures for determining whether these provisions have been satisfied. ISTQB Glossary.

[43] Functional specifications document, FSD. См. «Software requirements specification, SRS».

[44] Software requirements specification, SRS. SRS describes as fully as necessary the expected behavior of the software system. The SRS is used in development, testing, quality assurance, project management, and related project functions. People call this deliverable by many different names, including business requirements document, functional spec, requirements document, and others. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[45] **Architecture. Design.** A software *architecture* for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them. … *Architecture is design*, but not all design is architecture. That is, there are many design decisions that are left unbound by the architecture, and are happily left to the discretion and good judgment of downstream designers and implementers. The architecture establishes constraints on downstream activities, and those activities must produce artifacts (finer-grained designs and code) that are compliant with the architecture, but architecture does not define an implementation. Documenting Software Architectures. Clements. P. Очень подробное описание различия архитектуры и дизайна можно найти в статье Eden A., Kazman R. Architecture, Design, Implementation. URL: http://www.eden-study.org/articles/2003/icse03.pdf.

[46] **Test case.** A set of input values, execution preconditions, expected results and execution post-conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. ISTQB Glossary.

[47] **Test suite.** A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one. ISTQB Glossary.

[48] **Technical specifications.** Scripts, source code, data definition language, etc. PMBOK. – 3rd ed. Также см. «Specification».

[49] **Project documentation.** Other expectations and deliverables that are not a part of the software the team implements, but that are necessary to the successful completion of the project as a whole. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

panying documentation[50]), такую как встроенная помощь, руководство по установке и использованию, лицензионные соглашения и т. д;

- маркетинговую документацию (market requirements document, MRD[51]), которую представители разработчика или заказчика используют как на начальных этапах (для уточнения сути и концепции проекта), так и на финальных этапах развития проекта (для продвижения продукта на рынке).

В некоторых классификациях часть документов из продуктной документации может быть причислена к проектной документации – это совершенно нормально, т. к. понятие проектной документации по определению является более широким. Поскольку с этой классификацией связано очень много вопросов и непонимания, отразим её суть ещё раз – графически (рисунок 2.c) – и напомним, что мы договорились классифицировать документацию по признаку того, где (для чего) она является наиболее востребованной.

Рисунок 2.c – Соотношение понятий «продуктная документация» и «проектная документация»

Степень важности и глубина тестирования того или иного вида документации и даже отдельного документа определяется большим количеством факторов, но неизменным остаётся общий принцип: всё, что мы создаём в процессе разработки проекта (даже рисунки маркером на доске, письма, переписку в Skype), можно считать документацией и так или иначе подвергать тестированию (например, вычитывание письма перед отправкой – это тоже своего рода тестирование документации).

[50] **User documentation.** User documentation refers to the documentation for a product or service provided to the end users. The user documentation is designed to assist end users to use the product or service. This is often referred to as user assistance. The user documentation is a part of the overall product delivered to the customer. На основе статей doc-department.com.

[51] **Market requirements document, MRD.** An MRD goes into details about the target market segments and the issues that pertain to commercial success. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

2.3 Источники и пути выявления требований

Требования исходят от заказчика. Их сбор (gathering) и выявление (elicitation) осуществляются с помощью следующих основных техник[52] (рисунок 2.d).

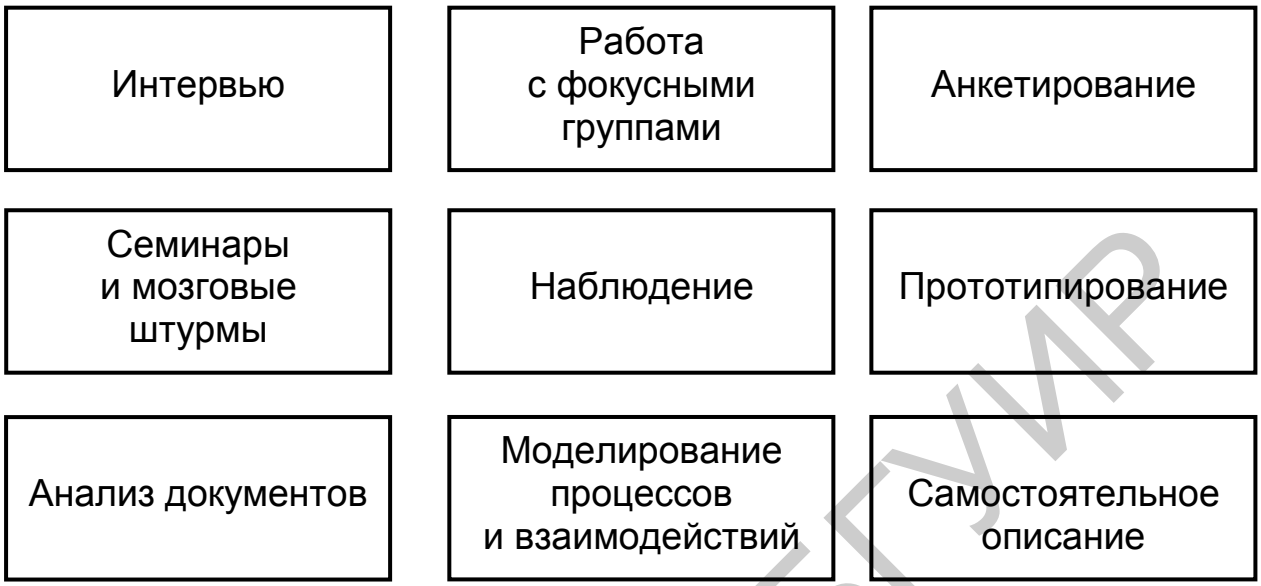


Рисунок 2.d – Основные техники сбора и выявления требований

Интервью. Самый универсальный путь выявления требований, заключающийся в общении проектного специалиста (как правило, специалиста по бизнес-анализу) и представителя заказчика (или эксперта, пользователя и т. д.) Интервью может проходить в классическом понимании этого слова (беседа в виде «вопрос – ответ»), в виде переписки и т. п. Главным здесь является то, что ключевыми фигурами выступают двое – интервьюируемый и интервьюер (хотя это и не исключает наличия «аудитории слушателей», например, в виде лиц, включённых в копию переписки).

Работа с фокусными группами. Может выступать как вариант «расширенного интервью», где источником информации является не одно лицо, а группа лиц (как правило, представляющих собой целевую аудиторию, и/или обладающих важной для проекта информацией, и/или уполномоченных принимать важные для проекта решения).

Анкетирование. Этот вариант выявления требований вызывает много споров, т. к. при неверной реализации может привести к нулевому результату при объёмных затратах. В то же время при правильной организации анкетирование позволяет автоматически собрать и обработать огромное количество ответов от огромного количества респондентов. Ключевым фактором успеха является правильное составление анкеты, правильный выбор аудитории и правильное преподнесение анкеты.

Семинары и мозговой штурм. Семинары позволяют группе людей очень быстро обменяться информацией (и наглядно продемонстрировать те или иные идеи), а также хорошо сочетаются с интервью, анкетированием, прототипированием и моделированием, в том числе для обсуждения результатов и формирования выводов и решений. Мозговой штурм может проводиться и как часть семинара, и как отдельный вид деятельности. Он позволяет за минимальное время сгенерировать

[52] Здесь можно почитать подробнее о том, в чём разница между сбором и выявлением требований: Brandenburg L. Requirements Gathering vs. Elicitation. URL: http://www.bridging-the-gap.com/requirements-gathering-vs-elicitation/.

большое количество идей, которые в дальнейшем можно не спеша рассмотреть с точки зрения их использования для развития проекта.

Наблюдение. Может выражаться как в буквальном созерцании некоторых процессов, так и во включении проектного специалиста в эти процессы в качестве участника. С одной стороны, наблюдение позволяет увидеть то, о чём (по совершенно различным соображениям) могут умолчать интервьюируемые, анкетируемые и представители фокусных групп, но с другой – отнимает очень много времени и чаще всего позволяет увидеть лишь часть процессов.

Прототипирование. Состоит в демонстрации и обсуждении промежуточных версий продукта (например, дизайн страниц сайта может быть сначала представлен в виде картинок, и лишь затем свёрстан). Это один из лучших путей поиска единого понимания и уточнения требований, однако он может привести к серьёзным дополнительным затратам при отсутствии специальных инструментов (позволяющих быстро создавать прототипы) и слишком раннем применении (когда требования ещё не стабильны, и высока вероятность создания прототипа, имеющего мало общего с пожеланиями заказчика).

Анализ документов. Хорошо работает тогда, когда эксперты в предметной области (временно) недоступны, а также в предметных областях, имеющих общепринятую устоявшуюся регламентирующую документацию. Также к этой технике относится и просто изучение документов, регламентирующих бизнес-процессы в предметной области заказчика или в конкретной организации, что позволяет приобрести необходимые для лучшего понимания сути проекта знания.

Моделирование процессов и взаимодействий. Может применяться как к «бизнес-процессам и взаимодействиям» (например: «*Договор на закупку формируется отделом закупок, визируется бухгалтерией и юридическим отделом…*»), так и к «техническим процессам и взаимодействиям» (например: «*Платёжное поручение генерируется модулем "Бухгалтерия", шифруется модулем "Безопасность" и передаётся на сохранение в модуль "Хранилище"*»). Данная техника требует высокой квалификации специалиста по бизнес-анализу, т. к. сопряжена с обработкой большого объёма сложной (и часто плохо структурированной) информации.

Самостоятельное описание. Является не столько техникой выявления требований, сколько техникой их фиксации и формализации. Очень сложно (и категорически запрещено) пытаться самому «придумать требования за заказчика», но в спокойной обстановке можно самостоятельно обработать собранную информацию и аккуратно оформить её для дальнейшего обсуждения и уточнения.

Часто специалисты по бизнес-анализу приходят в свою профессию из сферы тестирования. Если вас заинтересовало это направление, стоит ознакомиться со следующими книгами:

- Корнипаев И. «Требования для программного обеспечения: рекомендации по сбору и документированию»;
- Cadle J., Paul D., Turner P. «Business Analysis Techniques. 72 Essential Tools for Success»;
- Paul D., Yeates D., Cadle J. «Business Analysis». – 2nd ed.

2.4 Уровни и типы требований

Форма представления, степень детализации и перечень полезных свойств требований зависят от уровней и типов требований[53], которые схематично представлены на рисунке 2.е.

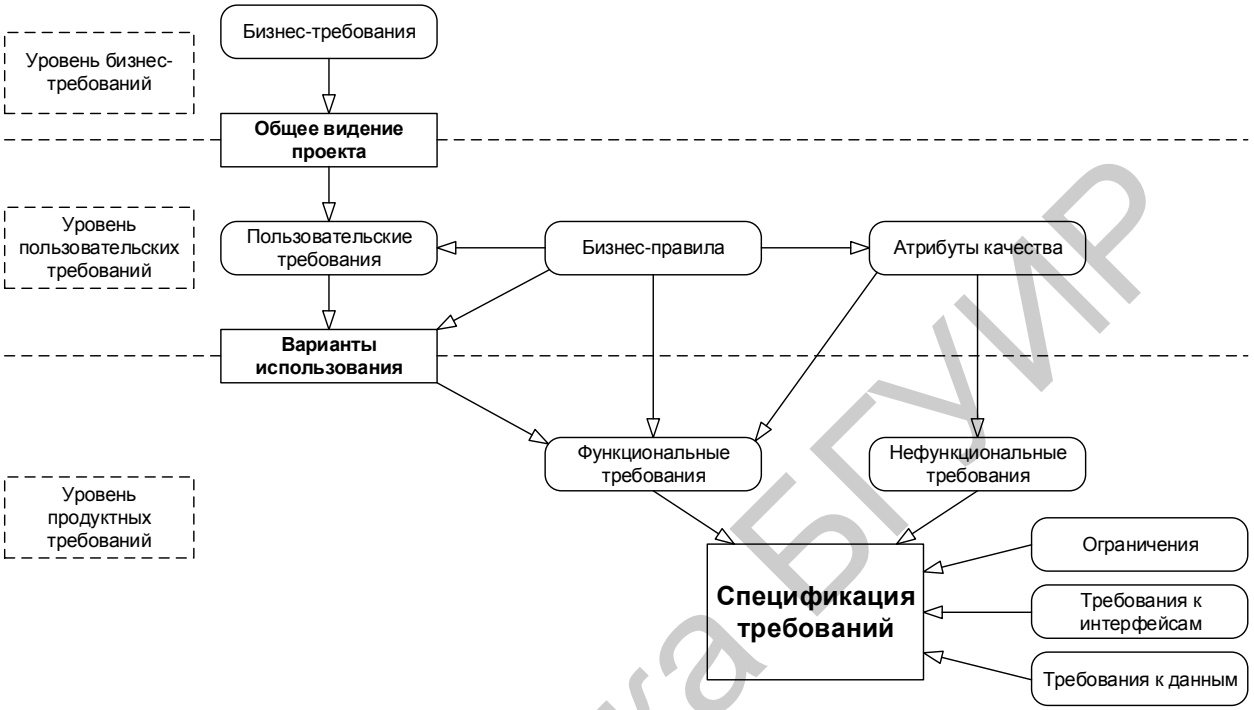


Рисунок 2.е – Уровни и типы требований

Бизнес-требования (business requirements[54]) выражают цель, ради которой разрабатывается продукт (зачем вообще он нужен, какая от него ожидается польза). Результатом выявления требований на этом уровне является общее видение (vision and scope[55]) – документ, который, как правило, представлен простым текстом и таблицами. Здесь нет детализации поведения системы и иных технических характеристик, но вполне могут быть определены приоритеты решаемых бизнес-задач, риски и т. п.

Несколько простых, изолированных от контекста и друг от друга примеров бизнес-требований:

- *Нужен инструмент, в реальном времени отображающий наиболее выгодный курс покупки и продажи валюты.*
- *Необходимо в два-три раза повысить количество заявок, обрабатываемых одним оператором за смену.*

[53] Очень подробное описание уровней и типов требований (а также их применения) можно найти в статье Westfall L. Software Requirements Engineering: What, Why, Who, When, and How. URL: http://www.westfallteam.com/Papers/The_Why_What_Who_When_and_How_Of_Software_Requirements.pdf.

[54] **Business requirement.** Anything that describes the financial, marketplace, or other business benefit that either customers or the developing organization wish to gain from the product. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[55] **Vision and scope.** The vision and scope document collects the business requirements into a single deliverable that sets the stage for the subsequent development work. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

- *Нужно автоматизировать процесс выписки товарно-транспортных накладных на основе договоров.*

Пользовательские требования (user requirements[56]) описывают задачи, которые пользователь может выполнять с помощью разрабатываемой системы (реакцию системы на действия пользователя, сценарии работы пользователя). Поскольку здесь уже появляется описание поведения системы, требования этого уровня могут быть использованы для оценки объёма работ, стоимости проекта, времени разработки и т. д. Пользовательские требования оформляются в виде вариантов использования (use cases[57]), пользовательских историй (user stories[58]), пользовательских сценариев (user scenarios[59]). (Также см. создание пользовательских сценариев в подразделе 4.6 «Наборы тест-кейсов»).

Несколько простых, изолированных от контекста и друг от друга примеров пользовательских требований:

- *При первом входе пользователя в систему должно отображаться лицензионное соглашение.*
- *Администратор должен иметь возможность просматривать список всех пользователей, работающих в данный момент в системе.*
- *При первом сохранении новой статьи система должна выдавать запрос на сохранение в виде черновика или публикацию.*

Бизнес-правила (business rules[60]) описывают особенности принятых в предметной области и/или непосредственно у заказчика процессов, ограничений и иных правил. Эти правила могут относиться к бизнес-процессам, правилам работы сотрудников, нюансам работы ПО и т. д.

Несколько простых, изолированных от контекста и друг от друга примеров бизнес-правил:

- *Никакой документ, просмотренный посетителями сайта хотя бы один раз, не может быть отредактирован или удалён.*
- *Публикация статьи возможна только после утверждения главным редактором.*
- *Подключение к системе извне офиса запрещено в нерабочее время.*

Атрибуты качества (quality attributes[61]) расширяют собой нефункциональные требования и на уровне пользовательских требований могут быть представлены в виде описания ключевых для проекта показателей качества (свойств продукта, не

[56] **User requirement.** User requirements are general statements of user goals or business tasks that users need to perform. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[57] **Use case.** A sequence of transactions in a dialogue between an actor and a component or system with a tangible result, where an actor can be a user or anything that can exchange information with the system. ISTQB Glossary.

[58] **User story.** A high-level user or business requirement commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria. ISTQB Glossary.

[59] A scenario is a hypothetical story, used to help a person think through a complex problem or system. Kaner C. An Introduction to Scenario Testing. URL: http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf.

[60] **Business rule.** A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business. A business rule expresses specific constraints on the creation, updating, and removal of persistent data in an information system. David H. et al. Defining Business Rules – What Are They Really.

[61] **Quality attribute.** A feature or characteristic that affects an item's quality. ISTQB Glossary.

связанных с функциональностью, но являющихся важными для достижения целей создания продукта: производительности, масштабируемости, восстанавливаемости). Атрибутов качества большое количество[62], но для любого проекта реально важными являются лишь некоторые их подмножества.

Несколько простых, изолированных от контекста и друг от друга примеров атрибутов качества:

- *Максимальное время готовности системы к выполнению новой команды после отмены предыдущей не может превышать одну секунду.*
- *Внесённые в текст статьи изменения не должны быть утеряны при нарушении соединения между клиентом и сервером.*
- *Приложение должно поддерживать добавление произвольного количества неиероглифических языков интерфейса.*

Функциональные требования (functional requirements[63]) описывают поведение системы, т. е. её действия (вычисления, преобразования, проверки, обработку и т. д.) В контексте проектирования функциональные требования в основном влияют на дизайн системы.

Несколько простых, изолированных от контекста и друг от друга примеров функциональных требований:

- *В процессе инсталляции приложение должно проверять остаток свободного места на целевом носителе.*
- *Система должна автоматически выполнять резервное копирование данных ежедневно в указанный момент времени.*
- *Электронный адрес пользователя, вводимый при регистрации, должен быть проверен на соответствие требованиям RFC822.*

Нефункциональные требования (non-functional requirements[64]) описывают свойства системы (удобство использования, безопасность, надёжность, расширяемость и т. д.), которыми она должна обладать при реализации своего поведения. В контексте проектирования нефункциональные требования в основном влияют на архитектуру системы.

Несколько простых, изолированных от контекста и друг от друга примеров нефункциональных требований:

- *При одновременной непрерывной работе с системой 1000 пользователей, минимальное время между возникновением сбоев должно быть более или равно 100 ч.*
- *Ни при каких условиях общий объём используемой приложением памяти не может превышать 2 Гбайт.*
- *Размер шрифта для любой надписи на экране должен поддерживать настройку в диапазоне от 5 до 15 пт.*

Ограничения, требования к интерфейсам и требования к данным в общем случае могут быть отнесены к нефункциональным, однако их часто выделяют в отдельные подгруппы (здесь для простоты рассмотрены лишь три таких подгруппы, но их может быть и гораздо больше; как правило, они возникают из атрибутов качества,

[62] Даже в Википедии их список огромен. URL: http://en.wikipedia.org/wiki/ List_of_system_quality_attributes.

[63] **Functional requirement.** A requirement that specifies a function that a component or system must perform. ISTQB Glossary. Functional requirements describe the observable behaviors the system will exhibit under certain conditions and the actions the system will let users take. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[64] **Non-functional requirement.** A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability. ISTQB Glossary.

но высокая степень детализации позволяет отнести их к уровню требований к продукту).

Ограничения (limitations, constraints[65]) представляют собой факторы, ограничивающие выбор способов и средств (в том числе инструментов) реализации продукта.

Несколько простых, изолированных от контекста и друг от друга примеров ограничений:

- *Все элементы интерфейса должны отображаться без прокрутки при разрешениях экрана от 800×600 до 1920×1080.*
- *Не допускается использование Flash при реализации клиентской части приложения.*
- *Приложение должно сохранять способность реализовывать функции с уровнем важности «критический» при отсутствии у клиента поддержки JavaScript.*

Требования к интерфейсам (external interfaces requirements[66]) описывают особенности взаимодействия разрабатываемой системы с другими системами и операционной средой.

Несколько простых, изолированных от контекста и друг от друга примеров требований к интерфейсам:

- *Обмен данными между клиентской и серверной частями приложения при осуществлении фоновых AJAX-запросов должен быть реализован в формате JSON.*
- *Протоколирование событий должно вестись в журнале событий операционной системы.*
- *Соединение с почтовым сервером должно выполняться согласно RFC3207 («SMTP over TLS»).*

Требования к данным (data requirements[67]) описывают структуры данных (и сами данные), являющиеся неотъемлемой частью разрабатываемой системы. Часто сюда относят описание базы данных и особенностей её использования.

Несколько простых, изолированных от контекста и друг от друга примеров требований к данным:

- *Все данные системы, за исключением пользовательских документов, должны храниться в БД под управлением СУБД MySQL; пользовательские документы должны храниться в БД под управлением СУБД MongoDB.*
- *Информация о кассовых транзакциях за текущий месяц должна храниться в операционной таблице, а по завершении месяца переноситься в архивную.*
- *Для ускорения операций поиска по тексту статей и обзоров должны быть предусмотрены полнотекстовые индексы на соответствующих полях таблиц.*

[65] **Limitation, constraint.** Design and implementation constraints legitimately restrict the options available to the developer. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[66] **External interface requirements.** Requirements in this category describe the connections between the system and the rest of the universe. They include interfaces to users, hardware, and other software systems. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[67] **Data requirements.** Data requirement describe the format, data type, allowed values, or default value for a data element; the composition of a complex business data structure; or a report to be generated. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

Спецификация требований (software requirements specification, SRS[68]) объединяет в себе описание всех требований уровня продукта и может представлять собой весьма объёмный документ (сотни и тысячи страниц).

Поскольку требований может быть очень много, а их приходится не только единожды написать и согласовать между собой, но и постоянно обновлять, работу проектной команды по управлению требованиями значительно облегчают соответствующие инструментальные средства (requirements management tools[69, 70]).

Для более глубокого понимания принципов создания, организации и использования набора требований рекомендуется ознакомиться с фундаментальной работой Карла Вигерса «Разработка требований к программному обеспечению» Wiegers K., Beatty J. Software Requirements. – 3rd ed. (Developer Best Practices). В этой же книге (в приложениях) приведены наглядные учебные примеры документов, описывающих требования различного уровня.

2.5 Свойства качественных требований

В процессе тестирования требований проверяется их соответствие определённому набору свойств (рисунок 2.f).

Завершённость (completeness[71]). Требование является полным и законченным с точки зрения представления в нём всей необходимой информации, ничто не пропущено по соображениям «это и так всем понятно».

Типичные проблемы с завершённостью:

- Отсутствуют нефункциональные составляющие требования или ссылки на соответствующие нефункциональные требования (например: «*Пароли должны храниться в зашифрованном виде*» – каков алгоритм шифрования?).
- Указана лишь часть некоторого перечисления (например: «*Экспорт осуществляется в форматы PDF, PNG и т. д.*» – что мы должны понимать под «и т. д.»?).
- Приведённые ссылки неоднозначны (например: «*см. выше*» вместо «*см. раздел 123.45.b*»).

[68] **Software requirements specification, SRS.** SRS describes as fully as necessary the expected behavior of the software system. The SRS is used in development, testing, quality assurance, project management, and related project functions. People call this deliverable by many different names, including business requirements document, functional spec, requirements document, and others. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[69] **Requirements management tool.** A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to predefined requirements rules. ISTQB Glossary.

[70] Обширный список инструментальных средств управления требованиями можно найти здесь: URL: http://makingofsoftware.com/resources/list-of-rm-tools.

[71] Each requirement must contain all the information necessary for the reader to understand it. In the case of functional requirements, this means providing the information the developer needs to be able to implement it correctly. No requirement or necessary information should be absent. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

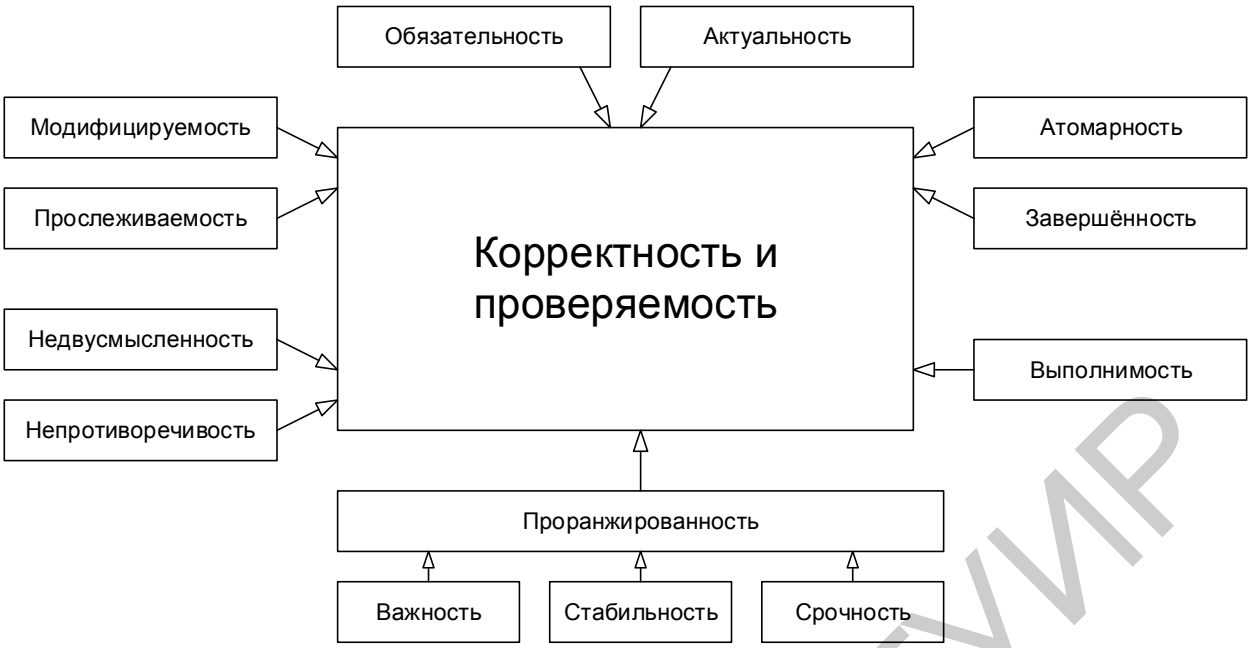


Рисунок 2.f – Свойства качественного требования

Атомарность, единичность (atomicity[72]). Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершённости и оно описывает одну и только одну ситуацию.

Типичные проблемы с атомарностью:

- В одном требовании, фактически, содержится несколько независимых (например: «*Кнопка "Restart" не должна отображаться при остановленном сервисе, окно "Log" должно вмещать не менее двадцати записей о последних действиях пользователя*» – здесь в одном предложении описаны совершенно разные элементы интерфейса в совершенно разных контекстах).
- Требование допускает разночтение в силу грамматических особенностей языка (например: «*Если пользователь подтверждает заказ и редактирует заказ или откладывает заказ, должен выдаваться запрос на оплату*» – здесь описаны три разных случая, и это требование стоит разбить на три отдельных во избежание путаницы). Такое нарушение атомарности часто влечёт за собой возникновение противоречивости.
- В одном требовании объединено описание нескольких независимых ситуаций (например: «*Когда пользователь входит в систему, для него должно отображаться приветствие; когда пользователь вошёл в систему, должно отображаться имя пользователя; когда пользователь выходит из системы, должно отображаться прощание*» – все эти три ситуации заслуживают того, чтобы быть описанными отдельными и куда более детальными требованиями).

Непротиворечивость, последовательность (consistency[73]). Требование не

[72] Each requirement you write represents a single market need that you either satisfy or fail to satisfy. A well written requirement is independently deliverable and represents an incremental increase in the value of your software. Blain T. Writing Good Requirements – The Big Ten Rules. URL: http://tynerblain.com/blog/2006/05/25/writing-good-requirements-the-big-ten-rules/.

[73] Consistent requirements don't conflict with other requirements of the same type or with higher-level business, user, or system requirements. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

должно содержать внутренних противоречий и противоречий другим требованиям и документам.

Типичные проблемы с непротиворечивостью:

- Противоречия внутри одного требования (например: «*После успешного входа в систему пользователя, не имеющего права входить в систему…*» – тогда как он успешно вошёл в систему, если не имел такого права?) .
- Противоречия между двумя и более требованиями, между таблицей и текстом, рисунком и текстом, требованием и прототипом и т. д. (например: «*712.a. Кнопка "Close" всегда должна быть красной*» и «*36452.x. Кнопка "Close" всегда должна быть синей*» – так всё же красной или синей?).
- Использование неверной терминологии или использование разных терминов для обозначения одного и того же объекта или явления (например: «*В случае если разрешение окна составляет менее 800×600…*» – разрешение есть у экрана, у окна – размер).

Недвусмысленность (unambiguousness[74], clearness). Требование описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок и допускает только однозначное объективное понимание. Требование атомарно в плане невозможности различной трактовки сочетания отдельных фраз.

Типичные проблемы с недвусмысленностью:

- Использование терминов или фраз, допускающих субъективное толкование (например: «*Приложение должно поддерживать передачу больших объёмов данных*» – насколько «больших»?). Вот лишь небольшой перечень слов и выражений, которые можно считать верными признаками двусмысленности: адекватно (adequate), быть способным (be able to; be capable of; capability of), легко (easy), обеспечивать (provide for), как минимум (as a minimum), эффективно (effectively), своевременно (timely), применимо (as applicable), если возможно (if possible), будет определено позже (to be determined, TBD), по мере необходимости (as appropriate), если это целесообразно (if practical), но не ограничиваясь (but not limited to), иметь возможность (capability to), нормально (normal), минимизировать (minimize), максимизировать (maximize), оптимизировать (optimize), быстро (rapid), удобно (user-friendly), просто (simple), часто (often), обычно (usual), большой (large), гибкий (flexible), устойчивый (robust), по последнему слову техники (state-of-the-art), улучшенный (improved), результативно (efficient). Вот утрированный пример требования, звучащего очень красиво, но совершенно нереализуемого и непонятного: «*В случае необходимости оптимизации передачи больших файлов система должна эффективно использовать минимум оперативной памяти, если это возможно*».
- Использование неочевидных или двусмысленных аббревиатур без расшифровки (например: «*Доступ к ФС осуществляется посредством системы прозрачного шифрования*» и «*ФС предоставляет возможность фиксировать сообщения в их текущем состоянии с хранением истории всех изменений*» – ФС здесь обозначает файловую систему? Точно? А не какой-нибудь «фиксатор сообщений»?).
- Формулировка требований из соображений, что нечто должно быть всем очевидно (например: «*Система конвертирует входной файл из формата PDF в выходной файл формата PNG*» – и при этом автор считает совершенно оче-

[74] Natural language is prone to two types of ambiguity. One type I can spot myself, when I can think of more than one way to interpret a given requirement. The other type of ambiguity is harder to catch. That's when different people read the requirement and come up with different interpretations of it. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

видным, что имена файлов система получает из командной строки, а многостраничный PDF конвертируется в несколько PNG-файлов, к именам которых добавляется «page-1», «page-2» и т. д.). Эта проблема перекликается с нарушением корректности.

Выполнимость (feasibility[75]). Требование технологически выполнимо и может быть реализовано в рамках бюджета и сроков разработки проекта.

Типичные проблемы с выполнимостью:

- Так называемое «озолочение» (gold plating) – требования, которые крайне долго и/или дорого реализуются и при этом практически бесполезны для конечных пользователей (например: «*Настройка параметров для подключения к базе данных должна поддерживать распознавание символов из жестов, полученных с устройств трёхмерного ввода*»).
- Технически нереализуемые на современном уровне развития технологий требования (например: «*Анализ договоров должен выполняться с применением искусственного интеллекта, который будет выносить однозначное корректное заключение о степени выгоды от заключения договора*»).
- В принципе нереализуемые требования (например: «*Система поиска должна заранее предусматривать все возможные варианты поисковых запросов и кэшировать их результаты*»).

Обязательность, нужность (obligation[76]) и **актуальность** (up-to-date). Если требование не является обязательным к реализации, оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета (см. «проранжированность по…»). Также должны быть исключены (или переработаны) требования, утратившие актуальность.

Типичные проблемы с обязательностью и актуальностью:

- Требование было добавлено «на всякий случай», хотя реальной потребности в нём не было и нет.
- Требованию выставлены неверные значения приоритета по критериям важности и/или срочности.
- Требование устарело, но не было переработано или удалено.

Прослеживаемость (traceability[77, 78]). Прослеживаемость бывает вертикальной (vertical traceability[79]) и горизонтальной (horizontal traceability[80]). Вертикальная позволяет соотносить между собой требования на различных уровнях требований,

---

[75] It must be possible to implement each requirement within the known capabilities and limitations of the system and its operating environment, as well as within project constraints of time, budget, and staff. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[76] Each requirement should describe a capability that provides stakeholders with the anticipated business value, differentiates the product in the marketplace, or is required for conformance to an external standard, policy, or regulation. Every requirement should originate from a source that has the authority to provide requirements. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[77] **Traceability.** The ability to identify related items in documentation and software, such as requirements with associated tests. ISTQB Glossary.

[78] A traceable requirement can be linked both backward to its origin and forward to derived requirements, design elements, code that implements it, and tests that verify its implementation. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[79] **Vertical traceability.** The tracing of requirements through the layers of development documentation to components. ISTQB Glossary.

[80] **Horizontal traceability.** The tracing of requirements for a test level through the layers of test documentation (e.g. test plan, test design specification, test case specification and test procedure specification or test script). ISTQB Glossary.

горизонтальная позволяет соотносить требование с тест-планом, тест-кейсами, архитектурными решениями и т. д.

Для обеспечения прослеживаемости часто используются специальные инструменты по управлению требованиями (requirements management tool[81]) и/или матрицы прослеживаемости (traceability matrix[82]).

Типичные проблемы с прослеживаемостью:

- Требования не пронумерованы, не структурированы, не имеют оглавления, не имеют работающих перекрёстных ссылок.
- При разработке требований не были использованы инструменты и техники управления требованиями.
- Набор требований неполный, носит обрывочный характер с явными «пробелами».

Модифицируемость (modifiability[83]). Это свойство характеризует простоту внесения изменений в отдельные требования и в набор требований. Можно говорить о наличии модифицируемости в том случае, если при доработке требований искомую информацию легко найти, а её изменение не приводит к нарушению иных описанных в этом перечне свойств.

Типичные проблемы с модифицируемостью:

- Требования неатомарны (см. «атомарность») и непрослеживаемы (см. «прослеживаемость»), а потому их изменение с высокой вероятностью порождает противоречивость (см. «непротиворечивость»).
- Требования изначально противоречивы (см. «непротиворечивость»). В такой ситуации внесение изменений (не связанных с устранением противоречивости) только усугубляет ситуацию, увеличивая противоречивость и снижая прослеживаемость.
- Требования представлены в неудобной для обработки форме (например, не использованы инструменты управления требованиями, и в итоге команде приходится работать с десятками огромных текстовых документов).

Проранжированность по важности, стабильности, срочности (ranked[84] for importance, stability, priority). Важность характеризует зависимость успеха проекта от успеха реализации требования. Стабильность характеризует вероятность того, что в ближайшем будущем в требование не будет внесено никаких изменений. Срочность

[81] **Requirements management tool.** A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to predefined requirements rules. ISTQB Glossary.

[82] **Traceability matrix.** A two-dimensional table, which correlates two entities (e.g., requirements and test cases). The table allows tracing back and forth the links of one entity to the other, thus enabling the determination of coverage achieved and the assessment of impact of proposed changes. ISTQB Glossary.

[83] To facilitate modifiability, avoid stating requirements redundantly. Repeating a requirement in multiple places where it logically belongs makes the document easier to read but harder to maintain. The multiple instances of the requirement all have to be modified at the same time to avoid generating inconsistencies. Cross-reference related items in the SRS to help keep them synchronized when making changes. Wiegers K., Beatty J. Software Requirementsю – 3rd ed.

[84] Prioritize business requirements according to which are most important to achieving the desired value. Assign an implementation priority to each functional requirement, user requirement, use case flow, or feature to indicate how essential it is to a particular product release. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

определяет распределение во времени усилий проектной команды по реализации того или иного требования.

Типичные проблемы с проранжированностью состоят в её отсутствии или неверной реализации и приводят к следующим последствиям.

- Проблемы с проранжированностью по важности повышают риск неверного распределения усилий проектной команды, направления усилий на второстепенные задачи и конечного провала проекта из-за неспособности продукта выполнять ключевые задачи с соблюдением ключевых условий.
- Проблемы с проранжированностью по стабильности повышают риск выполнения бессмысленной работы по совершенствованию, реализации и тестированию требований, которые в самое ближайшее время могут претерпеть кардинальные изменения (вплоть до полной утраты актуальности).
- Проблемы с проранжированностью по срочности повышают риск нарушения желаемой заказчиком последовательности реализации функциональности и ввода этой функциональности в эксплуатацию.

Корректность (correctness[85]) и **проверяемость** (verifiability[86]). Фактически эти свойства вытекают из соблюдения всех вышеперечисленных (или можно сказать, что они не соблюдаются, если нарушено хотя бы одно из вышеперечисленных). В дополнение можно отметить, что проверяемость подразумевает возможность создания объективного тест-кейса (тест-кейсов), однозначно показывающего, что требование реализовано верно и поведение приложения в точности соответствует требованию.

К типичным проблемам с корректностью также можно отнести:

- Опечатки (особенно опасны опечатки в аббревиатурах, превращающие одну осмысленную аббревиатуру в другую, также осмысленную, но не имеющую отношения к некоему контексту; такие опечатки крайне сложно заметить).
- Наличие неаргументированных требований к дизайну и архитектуре.
- Плохое оформление текста и сопутствующей графической информации, грамматические, пунктуационные и иные ошибки в тексте.
- Неверный уровень детализации (например, слишком глубокая детализация требования на уровне бизнес-требований или недостаточная детализация на уровне требований к продукту).
- Требования к пользователю, а не к приложению (например: «Пользователь должен быть в состоянии отправить сообщение». – увы, мы не можем влиять на состояние пользователя).

Хорошее краткое руководство по написанию качественных требований представлено в статье «Writing Good Requirements – The Big Ten Rules»[87].

[85] Each requirement must accurately describe a capability that will meet some stakeholder's need and must clearly describe the functionality to be built. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[86] If a requirement isn't verifiable, deciding whether it was correctly implemented becomes a matter of opinion, not objective analysis. Requirements that are incomplete, inconsistent, infeasible, or ambiguous are also unverifiable. Wiegers K., Beatty J. Software Requirements. – 3rd ed.

[87] Blain T. Writing Good Requirements – The Big Ten Rules. URL: http://tynerblain.com/blog/2006/05/25/writing-good-requirements-the-big-ten-rules/.

2.6 Техники тестирования требований

Тестирование документации и требований относится к разряду нефункционального тестирования (non-functional testing[88]). Основные техники такого тестирования в контексте требований таковы.

Взаимный просмотр (peer review[89]). Взаимный просмотр («рецензирование») является одной из наиболее активно используемых техник тестирования требований и может быть представлен в одной из трёх следующих форм (по мере нарастания его сложности и цены):

- **Беглый просмотр** (walkthrough[90]) может выражаться как в показе автором своей работы коллегам с целью достижения общего понимания и получения обратной связи, так и в простом обмене результатами работы между двумя и более авторами с тем, чтобы коллега высказал свои вопросы и замечания. Это самый быстрый и наиболее широко используемый вид просмотра. Для запоминания: аналог беглого просмотра – это ситуация, когда вы в школе с одноклассниками проверяли перед сдачей сочинения друг друга, чтобы найти описки и ошибки.
- **Технический просмотр** (technical review[91]) выполняется группой специалистов. В идеальной ситуации каждый специалист должен представлять свою область знаний. Просматриваемый продукт не может считаться достаточно качественным, пока хотя бы у одного из просматривающих остаются замечания. Для запоминания: аналог технического просмотра – это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т. д.
- **Формальная инспекция** (inspection[92]) представляет собой структурированный, систематизированный и документируемый подход к анализу документации. Для его выполнения привлекается большое количество специалистов, само выполнение занимает достаточно много времени, и по той причине этот вариант просмотра используется достаточно редко (как правило, при получении на сопровождение и доработку проекта, созданием которого ранее занималась другая компания). Для запоминания: аналог формальной инспекции – это ситуация генеральной уборки квартиры (включая содержимое всех шкафов, холодильника, кладовки и т. д.)

Вопросы. Следующей очевидной техникой тестирования и повышения качества требований является (повторное) использование техник выявления требований, а также (как отдельный вид деятельности) – задавание вопросов. Если хоть что-то в требованиях вызывает у вас непонимание или подозрение, задавайте вопросы. Можно адресовать вопросы представителям заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам

[88] **Non-functional testing.** Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability. ISTQB Glossary.

[89] **Peer review.** A review of a software work product by colleagues of the producer of the product for the purpose of identifying defects and improvements. Examples are inspection, technical review and walkthrough. ISTQB Glossary.

[90] **Walkthrough.** A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. ISTQB Glossary.

[91] **Technical review.** A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. ISTQB Glossary.

[92] **Inspection.** A type of peer review that relies on visual examination of documents to detect defects, e.g. violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. ISTQB Glossary.

при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы получить исчерпывающий ответ, который позволит улучшить требования.

Поскольку здесь начинающие тестировщики допускают много ошибок, рассмотрим подробнее, как следует задавать вопросы. В таблице 2.а приведены несколько плохо сформулированных требований, а также примеры неудачных и удачных вопросов. Отметим, что неудачные вопросы провоцируют на бездумные ответы, не содержащие полезной информации.

Таблица 2.а – Примеры неудачных и удачных вопросов к требованиям

Неудовлетворительное требование	**Неудачные вопросы**	**Удачные вопросы**
«Приложение должно быстро запускаться»	«Насколько быстро?» (Сформулировав вопрос таким образом, вы получите ответ в стиле «Очень быстро», «Максимально быстро», «Нууу… просто быстро».) «А если не получится быстро?» (Так вы рискуете просто удивить или даже разозлить заказчика.) «Всегда?» («Да, всегда. Хм, а вы ожидали другого ответа?»)	«Каково максимально допустимое время запуска приложения, на каком оборудовании и при какой загруженности этого оборудования операционной системой и другими приложениями?» «На достижение каких целей влияет скорость запуска приложения?» «Допускается ли фоновая загрузка отдельных компонентов приложения?» «Что является критерием того, что приложение закончило запуск?»
«Опционально должен поддерживаться экспорт документов в формат PDF»	«Любых документов?» (Ответы «Да, любых» или «Нет, только открытых» вам всё равно не помогут.) «В PDF какой версии должен производиться экспорт?» (Сам по себе вопрос хорош, но остаётся непонятным, что имелось в виду под «опционально».) «Зачем?» («Нужно!» Именно так хочется ответить, если вопрос не раскрыт полностью.)	«Насколько возможность экспорта в PDF важна?» «Как часто, кем и с какой целью она будет использоваться?» «Является ли PDF единственным допустимым форматом для этих целей или есть альтернативы?» «Допускается ли использование внешних утилит (например, виртуальных PDF-принтеров) для экспорта документов в PDF?»

Продолжение таблицы 2.а

Неудовлетворительное требование	Неудачные вопросы	Удачные вопросы
«Если дата события не указана, она выбирается автоматически»	«А если указана?» (То она указана. Логично, не так ли?) «А если дату невозможно выбрать автоматически?» (Сам вопрос интересен, но без пояснения причин невозможности звучит как насмешка.) «А если у события нет даты?» (Тут автор вопроса, скорее всего, хотел уточнить, обязательно ли это поле для заполнения. Но из самого требования видно, что обязательно: если оно не заполнено человеком, его должен заполнить компьютер.)	«Возможно, имелось в виду, что дата **генерируется** автоматически, а не **выбирается**? Если да, то по какому алгоритму она генерируется? Если нет, то из какого набора выбирается дата и как генерируется этот набор?» P.S. «Возможно, стоит использовать текущую дату?»

Тест-кейсы и чек-листы. Мы помним, что хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет нам определить, насколько требование проверяемо. Если вы можете быстро придумать несколько пунктов чек-листа, это ещё не признак того, что с требованием всё хорошо (например, оно может противоречить каким-то другим требованиям). Но если никаких идей по тестированию требования не появляется – это тревожный знак.

Рекомендуется для начала убедиться, что вы понимаете требование (в том числе прочесть смежные требования, обратиться за помощью к коллегам и т. д.) Также можно пока отложить работу с данным конкретным требованием и вернуться к нему позднее – возможно, анализ других требований позволит вам лучше понять и это «конкретное». Но если ничто не наводит на мысль, скорее всего, с требованием что-то не так.

Справедливости ради надо отметить, что на начальном этапе проработки требований такие случаи встречаются очень часто – требования сформированы очень поверхностно, расплывчато и явно нуждаются в доработке, т. е. здесь нет необходимости проводить сложный анализ, чтобы констатировать непроверяемость требования.

На стадии же, когда требования уже хорошо сформулированы и протестированы, вы можете продолжать использовать эту технику, совмещая разработку тест-кейсов и дополнительное тестирование требований.

Исследование поведения системы. Эта техника логически вытекает из предыдущей (продумывания тест-кейсов и чек-листов), но отличается тем, что здесь тестированию подвергается, как правило, не одно требование, а целый набор. Тестировщик мысленно моделирует процесс работы пользователя с системой, созданной по тестируемым требованиям, и ищет неоднозначные или вовсе неописанные варианты поведения системы. Этот подход сложен, требует достаточной квалификации тестировщика, но способен выявить нетривиальные недоработки, которые почти невозможно заметить, тестируя требования по отдельности.

Рисунки (графическое представление). Чтобы увидеть общую картину требований целиком, имеет смысл использовать рисунки, схемы, диаграммы, интеллект-карты[93] и т. д. Графическое представление удобно одновременно своей наглядностью и краткостью (например, UML-схема базы данных, занимающая один экран, может быть описана несколькими десятками страниц текста). На рисунке очень легко заметить, что какие-то элементы «не стыкуются», что где-то чего-то не хватает и т. д. Если вы для графического представления требований будете использовать общепринятую нотацию (например, уже упомянутый UML), вы получите дополнительные преимущества: вашу схему смогут без труда понимать и дорабатывать коллеги, а в итоге может получиться хорошее дополнение к текстовой форме представления требований.

Прототипирование. Можно сказать, что прототипирование часто является следствием создания графического представления и анализа поведения системы. С использованием специальных инструментов можно очень быстро сделать наброски пользовательских интерфейсов, оценить применимость тех или иных решений и даже создать не просто «прототип ради прототипа», а заготовку для дальнейшей разработки, если окажется, что реализованное в прототипе (возможно, с небольшими доработками) устраивает заказчика.

2.7 Пример анализа и тестирования требований

Поскольку наша задача состоит в том, чтобы сформировать понимание логики анализа и тестирования требований, мы будем рассматривать предельно краткий и простой их набор.

Отличный подробный пример требований можно найти в приложениях к книге Карла Вигерса «Разработка требований к программному обеспечению» Wiegers K., Beatty J. Software Requirements. – 3 rd ed. (Developer Best Practices).

Допустим, что у некоего клиента есть проблема: поступающие в огромном количестве его сотрудникам текстовые файлы приходят в разных кодировках, и сотрудники тратят много времени на перекодирование («ручной подбор кодировки»). Соответственно, он хотел бы иметь инструмент, позволяющий автоматически приводить кодировки всех текстовых файлов к какой-то одной. Итак, на свет появляется проект с кодовым названием «Конвертер файлов».

Уровень бизнес-требований. Бизнес-требования (см. подраздел 2.4 «Уровни и типы требований») изначально могут выглядеть так: «Необходим инструмент для автоматического приведения кодировок текстовых документов к одной».

Здесь мы можем задать большое количество вопросов. Для удобства приведём как сами вопросы, так и предполагаемые ответы клиента.

Задание 2.b: Прежде чем ознакомиться с приведённым ниже списком вопросов, сформулируйте собственный.

- В каких форматах представлены текстовые документы (обычный текст, HTML, MD, что-то иное)? ***Понятия не имею, я в этом не разбираюсь.***
- В каких кодировках приходят исходные документы? ***В разных.***
- В какую кодировку нужно преобразовать документы? ***В самую удобную и универсальную.***

[93] Mind map. URL: http://en.wikipedia.org/wiki/Mind_map.

- На каких языках написан текст в документах? ***Русский и английский.***
- Откуда и как поступают текстовые документы (по почте, с сайтов, по сети, както иначе)? ***Это неважно. Поступают разными способами, но мы их складываем в одну папку на диске, нам так удобно.***
- Каков максимальный объём документа? ***Два десятка страниц.***
- Как часто появляются новые документы (например, какой максимум документов может поступить за час)? ***200–300 в час.***
- С помощью чего сотрудники просматривают документы? ***Notepad++.***

Даже таких вопросов и ответов достаточно, чтобы переформулировать бизнес-требования следующим образом (обратите внимание, что многие вопросы были заданы с учётом будущего и не привели к появлению в бизнес-требованиях лишней технической детализации):

Суть проекта: разработка инструмента, устраняющего проблему множественности кодировок в текстовых документах, расположенных в локальном дисковом хранилище.

Цели проекта:

- Исключение необходимости ручного подбора кодировок текстовых документов.
- Сокращение времени работы с текстовым документом на величину, необходимую для ручного подбора кодировки.

Метрики достижения целей:

- Полная автоматизация определения и преобразования кодировки текстового документа к заданной.
- Сокращение времени обработки текстового документа в среднем на одну-две минуты на документ за счёт устранения необходимости ручного подбора кодировки.

Риски:

- Высокая техническая сложность безошибочного определения исходной кодировки текстового документа.

Почему мы решили, что среднее время на подбор кодировки составляет одну-две минуты? Мы провели наблюдение. Также мы помним ответы заказчика на вопросы об исходных форматах документов, исходных и конечной кодировках (заказчик честно сказал, что не имеет ответа), а потому мы попросили его предоставить нам доступ к хранилищу документов и выяснили:

- Исходные форматы: plain text, HTML, MD.
- Исходные кодировки: CP1251, UTF8, CP866, KOI8R.
- Целевая кодировка: UTF8.

На данном этапе мы принимаем решение заняться детализацией требований на более низких уровнях, т. к. появившиеся там вопросы позволят нам вернуться к бизнес-требованиям и улучшить их, если в этом возникнет необходимость.

Уровень пользовательских требований. Пришло время заняться уровнем пользовательских требований (см. подраздел 2. 4 «Уровни и типы требований»). Проект у нас несколько специфичный – результатами работы программного средства будет пользоваться большое количество людей, но само программное средство при этом они использовать не будут (оно, запущенное на сервере с хранилищем документов, будет просто выполнять свою работу «само по себе»). По этой причине под пользователем здесь мы будем понимать человека, настраивающего работу приложения на сервере.

Сначала мы создадим небольшую диаграмму вариантов использования, представленную на рисунке 2.g (да, иногда её создают **после** текстового описания требо-

ваний, но иногда и **до** – в данном случае удобнее сделать это сначала). В реальных проектах подобные схемы могут быть на несколько порядков более сложными и требующими подробной детализации каждого варианта использования. У нас же проект миниатюрный, потому схема получилась элементарной, и мы сразу переходим к описанию требований.

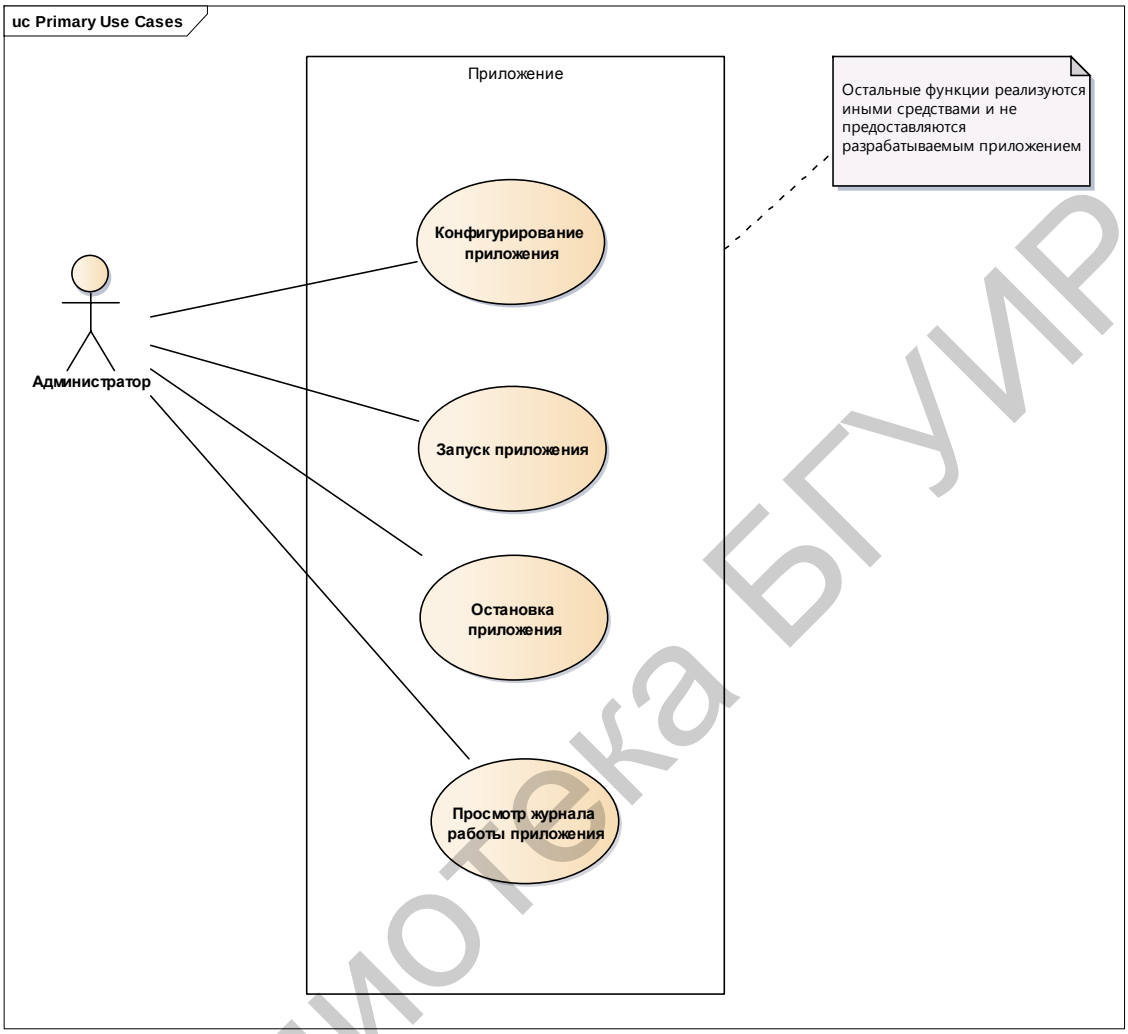


Рисунок 2.g – Диаграмма вариантов использования

Внимание! Это – неудовлетворительные требования. И мы далее будем их улучшать.

Системные характеристики:

- СХ-1: Приложение является консольным.
- СХ-2: Для работы приложение использует интерпретатор PHP.
- СХ-3: Приложение является кроссплатформенным.

Пользовательские требования:

- Также см. диаграмму вариантов использования.
- ПТ-1: Запуск и остановка приложения:
  - ПТ-1.1: запуск приложения производится из консоли командой PHP converter.php параметры;
  - ПТ-1.2: остановка приложения производится выполнением команды Ctrl+C.
- ПТ-2: Конфигурирование приложения:
  - ПТ-2.1: конфигурирование приложения сводится к указанию путей в файловой системе;

- ПТ-2.2: целевой кодировкой является UTF8;
- ПТ-3: Просмотр журнала работы приложения:
  - ПТ-3.1: в процессе работы приложение должно выводить журнал своей работы в консоль и лог-файл;
  - ПТ-3.2: при первом запуске приложения лог-файл создаётся, а при последующих – дописывается.

Бизнес-правила:

- БП-1: Источник и приёмник файлов:
  - БП-1.1: каталоги, являющиеся источником исходных и приёмником конечных файлов, не должны совпадать;
  - БП-1.2: каталог, являющийся приёмником конечных файлов, не может быть подкаталогом источника.

Атрибуты качества:

- АК-1: Производительность:
  - АК-1.1: приложение должно обеспечивать скорость обработки данных 5 Мбайт/с.
- АК-2: Устойчивость к входным данным:
  - АК-2.1: приложение должно обрабатывать входные файлы размером до 50 Мбайт включительно;
  - АК-2.2: если входной файл не является текстовым, приложение должно произвести обработку.

В подразделе 2.8 «Типичные ошибки при анализе и тестировании требований», сказано, что не стоит изменять исходный формат файла и форматирование документа, потому мы используем встроенные средства Word для отслеживания изменений и добавления комментариев. Примерный результат показан на рисунке 2.h.

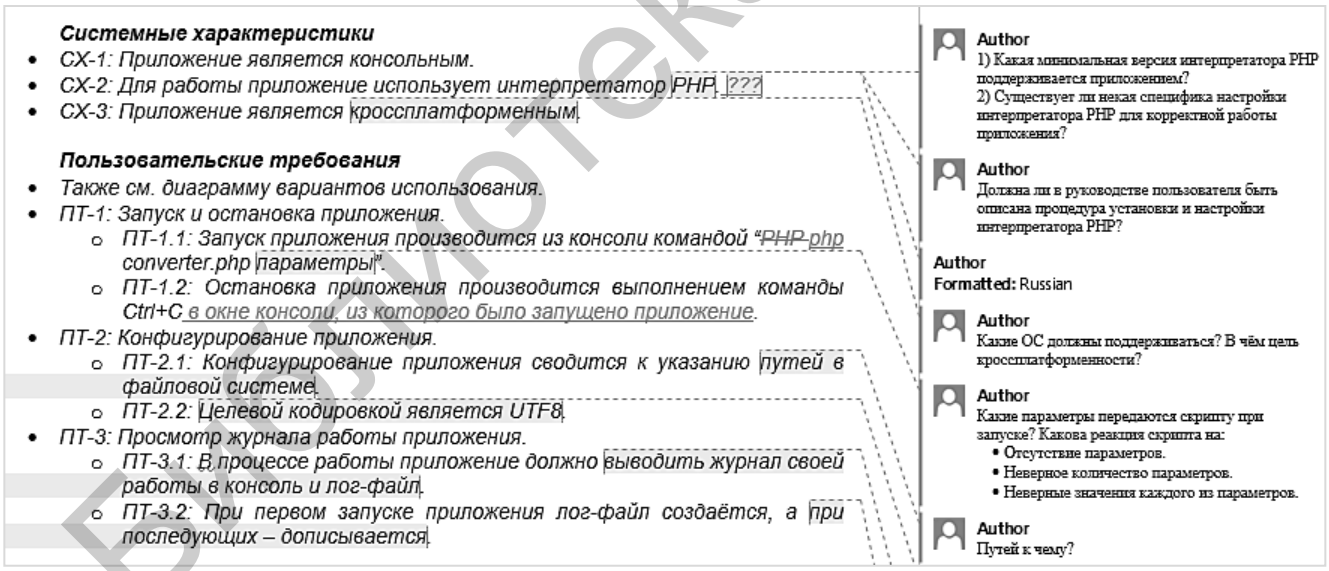


Рисунок 2.h – Использование средств Word для работы с требованиями

К сожалению, мы не можем в данном тексте применить эти средства (результат будет отображаться некорректно, т. к. вы сейчас, скорее всего, читаете этот текст не в виде DOCX-документа), а поэтому применим второй классический способ, заключающийся во вписывании своих вопросов и комментариев прямо внутрь текста требований.

Проблемные места требований отмечены подчёркиванием, наши вопросы отмечены *курсивом*, предполагаемые ответы заказчика (если точнее, технического

специалиста-заказчика) – **жирным**. В процессе анализа текст требований примет вот такой вид.

Системные характеристики:

- СХ-1: Приложение является консольным.
- СХ-2: Для работы приложение использует интерпретатор PHP:
  - *Какая минимальная версия интерпретатора PHP поддерживается приложением?* **5.5.x.**
  - *Существует ли некая специфика настройки интерпретатора PHP для корректной работы приложения?* **Наверное, должен работать mbstring.**
  - *Настаиваете ли вы на реализации приложения именно на PHP? Если да, то почему?* **Да, только PHP. У нас есть сотрудник, который его знает.**
  - *Должна ли в руководстве пользователя быть описана процедура установки и настройки интерпретатора PHP?* **Нет.**
- СХ-3: Приложение является кроссплатформенным:
  - *Какие ОС должны поддерживаться?* **Любая, где работает PHP.**
  - *В чём вообще цель кроссплатформенности?* **Мы ещё не знаем, на чём будет работать сервер.**

Пользовательские требования:

- Также см. диаграмму вариантов использования.
- ПТ-1: Запуск и остановка приложения:
  - ПТ-1.1: запуск приложения производится из консоли командой «PHP *(возможно, здесь опечатка*: *должно быть php (в нижнем регистре))*: converter.php параметры». **Да.**
    - *Какие параметры передаются скрипту при запуске?* **Каталог с исходными файлами, каталог с конечными файлами.**
    - *Какова реакция скрипта на следующие моменты*:
      - *на отсутствие параметров?* **Пишет Help;**
      - *неверное количество параметров?* **Пишет Help и поясняет, что не так;**
      - *неверные значения каждого из параметров?* **Пишет Help и поясняет, что не так.**
  - ПТ-1.2: остановка приложения производится выполнением команды Ctrl+C *(предлагаем дополнить это выражение фразой «в окне консоли, из которого было запущено приложение»).* **Да.**
- ПТ-2: Конфигурирование приложения:
  - ПТ-2.1: конфигурирование приложения сводится к указанию путей в файловой системе:
    - *Путей к чему?* **Каталог с исходными файлами, каталог с конечными файлами.**
  - ПТ-2.2: целевой кодировкой является UTF8:
    - *Предполагается ли указание иной целевой кодировки, или UTF8 используется в качестве целевой всегда?* **Только UTF8, других не надо.**
- ПТ-3: Просмотр журнала работы приложения:
  - ПТ-3.1: в процессе работы приложение должно выводить журнал своей работы в консоль и лог-файл:
    - *Каков формат журнала?* **Дата-время, что и с чем делали, что получилось. Загляните в лог Apache, там корректно написано.**

        - *Различаются ли форматы журнала для консоли и лог-файла?* **Нет.**
        - *Как определяется имя лог-файла?* **Третий параметр при запуске. Если не указан, то пусть будет converter.log рядом с php-скриптом.**
    - ПТ-3.2: при первом запуске приложения лог-файл создаётся, а при последующих – дописывается:
        - *Как приложение различает свой первый и последующие запуски?* **Никак не различает.**
        - *Какова реакция приложения на отсутствие лог-файла в случае, если это не первый запуск?* **Приложение его создаёт. Идея в том, чтобы оно не перезаписывало старый лог – и всё.**

Бизнес-правила:

- БП-1: Источник и приёмник файлов:
    - БП-1.1: каталоги, являющиеся источником исходных и приёмником конечных файлов, не должны совпадать: **Да.**
        - *Какова реакция приложения в случае совпадения этих каталогов?* **Пишет Help и поясняет, что не так.**
    - БП-1.2: каталог, являющийся приёмником конечных файлов, не может быть подкаталогом источника *(предлагаем заменить слово «источника» на фразу «каталога, являющегося источником исходных файлов»).* **Не надо, становится непонятно.**

Атрибуты качества:

- АК-1: Производительность:
    - АК-1.1: приложение должно обеспечивать скорость обработки данных 5 Мбайт/с:
        - *При каких технических характеристиках системы?* **i7, 4GB RAM.**
- АК-2: Устойчивость к входным данным:
    - АК-2.1: приложение должно обрабатывать входные файлы размером до 50 Мбайт включительно:
        - *Какова реакция приложения на файлы, размер которых превышает 50 Мбайт?* **Не обрабатывает.**
    - АК-2.2: если входной файл не является текстовым, приложение должно произвести обработку:
        - *Обработку чего должно произвести приложение?* **Этого файла. Неважно, что станет с файлом, лишь бы после этого скрипт функционировал.**

Задание 2.c: Проанализируйте предложенный набор требований с точки зрения свойств качественных требований (см. подраздел 2.5 «Свойства качественных требований»), сформулируйте свои вопросы заказчику, которые позволят улучшить этот набор требований.

Здесь есть несколько важных моментов, на которые стоит обратить внимание:

- Ответы заказчика могут быть менее структурированными и последовательными, чем наши вопросы. Это нормально. Он может позволить себе такое, мы – нет.
- Ответы заказчика могут содержать противоречия (в нашем примере сначала заказчик писал, что параметрами, передаваемыми из командной строки, являются только два имени каталога, а потом дополнил, что там же указывается имя лог-файла). Это тоже естественно, т. к. заказчик мог что-то забыть или перепутать. Наша задача – свести эти противоречивые данные воедино (если это возможно) и задать уточняющие вопросы (если это необходимо).

- В случае если с нами общается технический специалист, в его ответах вполне могут встречаться технические жаргонизмы (как «Help» в нашем примере). Не надо переспрашивать его о том, что это такое, если жаргонизм имеет однозначное общепринятое значение, но при доработке текста наша задача – написать то же самое строгим техническим языком. Если жаргонизм всё же непонятен, тогда лучше уточнить его значение (так, «Help» – это всего лишь краткая помощь, выводимая консольными приложениями как подсказка о том, как их использовать).

Уровень продуктных требований (см. подраздел 2.4 «Уровни и типы требований»). Применим так называемое «самостоятельное описание» (см. подраздел 2.3 «Источники и пути выявления требований») и улучшим требования. Поскольку мы уже получили много специфической технической информации, можно параллельно писать полноценную спецификацию требований. Во многих случаях, когда для оформления требований используется простой текст, для удобства формируется единый документ, который интегрирует в себе как пользовательские требования, так и детальные спецификации. Теперь требования принимают следующий вид.

Системные характеристики:

- СХ-1: Приложение является консольным.
- СХ-2: Приложение разрабатывается на языке программирования PHP (причина выбора языка PHP отражена в пункте О-1 раздела «Ограничения», особенности и важные настройки интерпретатора PHP отражены в пункте ДС-1 раздела «Детальные спецификации»).
- СХ-3: Приложение является кроссплатформенным с учётом пункта О-4 раздела «Ограничения».

Пользовательские требования:

- Также см. диаграмму вариантов использования.
- ПТ-1: Запуск и остановка приложения:
  - ПТ-1.1: запуск приложения производится из консоли командой «php converter.php SOURCE_DIR DESTINATION_DIR [LOG_FILE_NAME]» (описание параметров приведено в разделе ДС-2.1, реакция на ошибки при указании параметров приведена в разделах ДС-2.2, ДС-2.3, ДС-2.4);
  - ПТ-1.2: остановка приложения производится выполнением команды Ctrl+C в окне консоли, из которого было запущено приложение.
- ПТ-2: Конфигурирование приложения:
  - ПТ-2.1: конфигурирование приложения сводится к указанию параметров командной строки (см. ДС-2);
  - ПТ-2.2: целевой кодировкой преобразования текстов является кодировка UTF8 (также см. О-5).
- ПТ-3: Просмотр журнала работы приложения:
  - ПТ-3.1: в процессе работы приложение должно выводить журнал своей работы в консоль и лог-файл (см. ДС-4), имя которого определяется правилами, указанными в ДС-2.1;
  - ПТ-3.2: формат журнала работы и лог файла указан в ДС-4.1, а реакция приложения на наличие или отсутствие лог-файла указана в ДС-4.2 и ДС-4.3 соответственно.

Бизнес-правила:

- БП-1: Источник и приёмник файлов:
  - БП-1.1: каталоги, являющиеся источником исходных и приёмником конечных файлов, не должны совпадать (см. также ДС-2.1 и ДС-3.2);
  - БП-1.2: каталог, являющийся приёмником конечных файлов, не может

находиться внутри каталога, являющегося источником исходных файлов или его подкаталогов (см. также ДС-2.1 и ДС-3.2).

Атрибуты качества:

- АК-1: Производительность:
  - АК-1.1: приложение должно обеспечивать скорость обработки данных не менее 5 Мбайт/с на аппаратном обеспечении, эквивалентном следующему: процессор i7, 4 Гбайт оперативной памяти, средняя скорость чтения/записи на диск – 30 Мбайт/с. Также см. О-6.
- АК-2: Устойчивость к входным данным:
  - АК-2.1: требования относительно форматов обрабатываемых файлов изложены в ДС-5.1;
  - АК-2.2: требования относительно размеров обрабатываемых файлов изложены в ДС-5.2;
  - АК-2.3: поведение приложения в ситуации обработки файлов с нарушениями формата определено в ДС-5.3.

Ограничения:

- О-1: Приложение разрабатывается на языке программирования PHP, использование которого обусловлено возможностью заказчика осуществлять поддержку приложения силами собственного IT-отдела.
- О-2: Ограничения относительно версии и настроек интерпретатора PHP отражены в пункте ДС-1 раздела «Детальные спецификации».
- О-3: Процедуры установки и настройки интерпретатора PHP выходят за рамки данного проекта и **не описываются** в документации.
- О-4: Кроссплатформенные возможности приложения сводятся к способности работать под ОС семейства Windows и Linux, поддерживающих работу интерпретатора PHP версии, указанной в ДС-1.1.
- О-5: Целевая кодировка UTF8 является жёстко заданной, и её изменение в процессе эксплуатации приложения не предусмотрено.
- О-6: Допускается невыполнение АК-1.1 в случае, если невозможность обеспечить заявленную производительность обусловлена объективными внешними причинами (например, техническими проблемами на сервере заказчика).

Созданные на основе таких пользовательских требований детальные спецификации имеют следующий вид.

Детальные спецификации:

- **ДС-1: Интерпретатор PHP:**
  - ДС-1.1: минимальная версия – 5.5;
  - ДС-1.2: для работы приложения должно быть установлено и включено расширение mbstring.
- **ДС-2: Параметры командной строки:**
  - ДС-2.1: при запуске приложения оно получает из командной строки три параметра:
    - SOURCE_DIR – обязательный параметр, определяющий путь к каталогу с файлами, которые необходимо обработать;
    - DESTINATION_DIR – обязательный параметр, определяющий путь к каталогу, в который необходимо поместить обработанные файлы (этот каталог не может находиться внутри каталога SOURCE_DIR или в его подкаталогах (см. БП-1.1 и БП-1.2));
    - LOG_FILE_NAME – необязательный параметр, определяющий полное имя лог-файла (по умолчанию лог-файл с именем «converter.log» размещается по тому же пути, по которому находится файл скрипта converter.php).

- ДС-2.2: при указании недостаточного количества параметров командной строки приложение должно завершить работу, выдав сообщение об использовании (ДС-3.1);
- ДС-2.3: при указании излишнего количества параметров командной строки приложение должно игнорировать все параметры командной строки, кроме указанных в пункте ДС-2.1;
- ДС-2.4: при указании неверного значения любого из параметров командной строки приложение должно завершить работу, выдав сообщение об использовании (ДС-3.1), а также сообщив имя неверно указанного параметра, его значение и суть ошибки (см. ДС-3.2).

- **ДС-3: Сообщения:**
  - ДС-3.1: сообщение об использовании «USAGE converter.php SOURCE_DIR DISTINATION_DIR LOG_FILE_NAME»;
  - ДС-3.2: сообщения об ошибках:
    - Directory not exists or inaccessible;
    - Destination dir may not reside within source dir tree;
    - Wrong file name or inaccessible path;
- **ДС-4: Журнал работы:**
  - ДС-4.1: формат журнала работы одинаков для отображения в консоли и записи в лог-файл: YYYY-MM-DD HH:II:SS имя_операции параметры_операции результат_операции;
  - ДС-4.2: в случае если лог-файл отсутствует, должен быть создан новый пустой лог-файл;
  - ДС-4.3: в случае если лог-файл уже существует, должно происходить добавление новых записей в его конец.
- **ДС-5: Форматы и размеры файлов:**
  - ДС-5.1: приложение должно обрабатывать текстовые файлы на русском и английском языках в следующих исходных кодировках: WIN1251, CP866, KOI8R.
    Обрабатываемые файлы могут быть представлены в следующих форматах, определяемых расширениями файлов:
    - Plain Text (TXT);
    - Hyper Text Markup Language Document (HTML);
    - Mark Down Document (MD).
  - ДС-5.2: приложение должно обрабатывать файлы размером до 50 Мбайт (включительно), игнорируя любой файл, размер которого превышает 50 Мбайт;
  - ДС-5.3: если файл с расширением из ДС-5.1 содержит внутри себя данные, не соответствующие формату файла, допускается повреждение таких данных.

Задание 2.d: Заметили ли вы, что в исправленном варианте требований «потерялась» диаграмма вариантов использования (равно как и активная ссылка на неё)? (Просто тест на внимательность, не более.)

Итак, мы получили набор требований, с которым уже вполне можно работать. Он не идеален (и никогда вы не встретите идеальных требований), но он вполне пригоден для того, чтобы разработчики смогли реализовать приложение, а тестировщики – протестировать его.

Задание 2.e: Протестируйте этот набор требований и найдите в нём хотя бы 3–5 ошибок и неточностей, задайте соответствующие вопросы заказчику.

2.8 Типичные ошибки при анализе и тестировании требований

Для лучшего понимания и запоминания материала рассмотрим типичные ошибки, совершаемые в процессе анализа и тестирования требований.

Изменение формата файла и документа. По какой-то непонятной причине очень многие начинающие тестировщики стремятся полностью уничтожить исходный документ, заменив текст таблицами (или наоборот), перенося данные из Word в Excel и т. д. Это можно сделать только в одном случае: если вы предварительно договорились о подобных изменениях с автором документа. В противном случае вы полностью уничтожаете чью-то работу, делая дальнейшее развитие документа весьма затруднительным.

Самое худшее, что можно сделать с документом, – это сохранить его в итоге в некотором формате, предназначенном скорее для чтения, чем для редактирования (PDF, наборе картинок и т. п.).

Если требования изначально создаются в некоей системе управления требованиями, этот вопрос неактуален, но высокоуровневые требования большинство заказчиков привыкли видеть в обычном DOCX-документе, а Word предоставляет такие прекрасные возможности работы с документом, как отслеживание изменений (рисунок 2.i) и комментарии (рисунок 2.j).

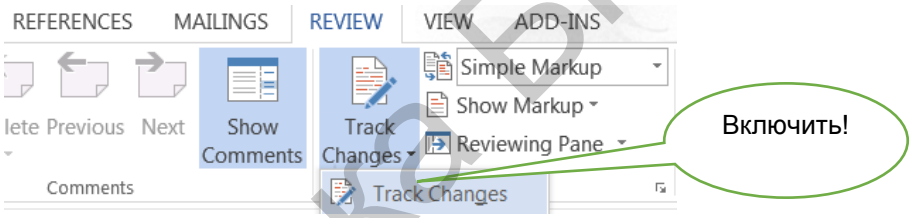


Рисунок 2.i – Активация отслеживания изменений в Word

В итоге получаем результат, представленный на рисунке 2.j: исходный формат сохраняется (а автор к нему уже привык), все изменения хорошо видны и могут быть приняты или отклонены несколькими щелчками мыши, а типичные часто повторяющиеся вопросы вы можете помимо указания в комментариях вынести в отдельный список и поместить его в том же документе.

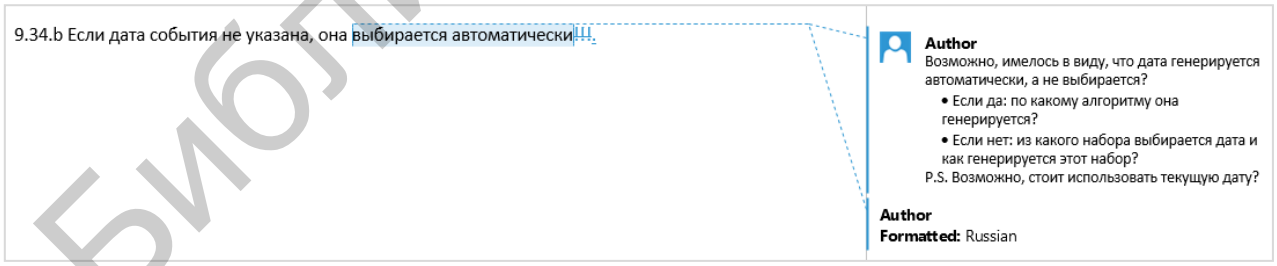


Рисунок 2.j – Вид документа с правками

Следует учесть ещё два небольших, но неприятных момента относительно таблиц:

- Выравнивание ВСЕГО текста в таблице по центру. Да, выравнивание по центру хорошо смотрится в заголовках и ячейках с парой-тройкой слов, но если так выровнять весь текст, читать его становится сложно.
- Отключение границ ячеек. Такая таблица намного хуже читается.

Отметка того факта, что с требованием всё в порядке. Если у вас не возникло вопросов и/или замечаний к требованию, то не надо об этом писать. Любые пометки в документе подсознательно воспринимаются как признак проблемы, и такое «одобрение требований» только отвлекает и затрудняет работу с документом – менее заметными становятся пометки, относящиеся к проблемам.

Описание одной и той же проблемы в нескольких местах. Помните, что ваши пометки, комментарии, замечания и вопросы тоже должны обладать свойствами хороших требований (настолько, насколько эти свойства к ним применимы). Если вы много раз в разных местах пишете одно и то же об одном и том же, вы нарушаете как минимум свойство модифицируемости. Постарайтесь в таком случае вынести ваш текст в конец документа, укажите в нём же (в начале) перечень пунктов требований, к которым он относится, а в комментариях к самим требованиям просто ссылайтесь на этот текст.

Написание вопросов и комментариев без указания места требования, к которым они относятся. Если ваше инструментальное средство позволяет указать часть требования, к которому вы пишете вопрос или комментарий, сделайте это (например, Word позволяет выделить для комментирования любую часть текста, хоть один символ). Если это невозможно, цитируйте соответствующую часть текста. В противном случае вы порождаете неоднозначность или вовсе делаете вашу пометку бессмысленной, т. к. невозможно понять, о чём вообще идёт речь.

Задать неудачно сформулированные вопросы. Эта ошибка была подробно рассмотрена выше (см. подраздел 2.6 «Техники тестирования требований» и таблицу 2.а). Однако обратим внимание на ещё три вида неудачных вопросов:

- Вопросы первого вида возникают из-за того, что автор вопроса не знает общепринятой терминологии или типичного поведения стандартных элементов интерфейса (например, «Что такое чек-бокс?», «Как в списке можно выбрать несколько пунктов?», «Как подсказка может всплывать?»).
- Второй вид неудачных вопросов похож на первый из-за формулировок: вместо того, чтобы написать «что вы имеете в виду под {чем-то}?», автор вопроса пишет «что такое {что-то}?». То есть вместо вполне логичного уточнения создаётся ситуация, очень похожая на рассмотренную в предыдущем пункте.
- Третий вид сложно привязать к причине возникновения, но его суть в том, что к некорректному и/или невыполнимому требованию задаётся вопрос наподобие «Что будет, если мы это сделаем?». Ничего не будет, т. к. мы это точно не сделаем. И вопрос должен быть совершенно иным, правильно сформулированным (каким именно – зависит от конкретной ситуации, но точно не этот).

И ещё раз напомним о точности формулировок: иногда одно-два слова могут изначально уничтожить отличную идею, превратив удачный вопрос в неудачный. Сравните: «Что такое формат даты по умолчанию?» и «Каков формат даты по умолчанию?». Первый вариант просто показывает некомпетентность автора вопроса, тогда как второй позволяет получить полезную информацию.

К этой же проблеме относится непонимание контекста. Часто можно увидеть вопросы: «О каком приложении идёт речь?», «Что такое система?» и т. п. Чаще всего автор таких вопросов просто вырвал требование из контекста, по которому было совершенно ясно, о чём идёт речь.

Написание очень длинных комментариев и/или вопросов. История знает случаи, когда одна страница исходных требований превращалась в 20–30 страниц текста анализа и вопросов. Это неправильный подход. Все те же мысли можно выразить более кратко, чем сэкономить как своё время, так и время автора исходного документа. Тем более стоит учитывать, что на начальных стадиях работы с требованиями они весьма нестабильны, и может получиться так, что ваши

5–10 страниц комментариев относятся к требованию, которое просто удалят или изменят до неузнаваемости.

Критика текста или даже его автора. Помните, что ваша задача – сделать требования лучше, а не показать их недостатки (или недостатки автора). Поэтому комментарии типа «Неудовлетворительное требование», «Неужели вы не понимаете, как глупо это звучит», «Надо переформулировать» – неуместны и недопустимы.

Категоричные заявления без обоснования. Как продолжение ошибки «Критика текста или даже его автора» можно отметить и просто категоричные заявления наподобие «это невозможно», «мы не будем этого делать», «это не нужно». Даже если вы понимаете, что требование бессмысленно или невыполнимо, эту мысль стоит сформулировать в корректной форме и дополнить вопросами, позволяющими автору документа самому принять окончательное решение. Например, «это не нужно» можно переформулировать так: «Мы сомневаемся в том, что данная функция будет востребована пользователями. Какова важность этого требования? Уверены ли вы в его необходимости?»

Указание проблемы с требованиями без пояснения её сути. Помните, что автор исходного документа может быть непрофессионалом по тестированию или бизнес-анализу. Потому просто пометка в стиле «неполнота», «двусмысленность» и т. д. могут ничего ему не сказать. Поясняйте свою мысль.

К этой же ошибке можно отнести небольшую, но досадную недоработку, касающуюся противоречивости: если вы обнаружили некие расхождения, сделайте соответствующие пометки во всех противоречащих друг другу местах, а не только в одном из них. Например, вы обнаружили, что требование 20 противоречит требованию 30. Тогда в требовании 20 отметьте, что оно противоречит требованию 30, и наоборот. И поясните суть противоречия.

Плохое оформление вопросов и комментариев. Старайтесь сделать ваши вопросы и комментарии максимально простыми для восприятия. Помните не только о краткости формулировок, но и об оформлении текста (см., например, как на рисунке 2.j вопросы структурированы в виде списка – такая структура воспринимается намного легче, чем сплошной текст). Перечитайте свой текст, исправьте опечатки, грамматические и пунктуационные ошибки и т. д.

Описание проблемы не в том месте, к которому она относится. Классическим примером может быть неточность в сноске, приложении или рисунке, которая почему-то описана не там, где она находится, а в ссылке на соответствующий элемент. Исключением может считаться противоречивость, при которой описать проблему нужно в обоих местах.

Ошибочное восприятие требования как «требования к пользователю». Ранее (см. «Корректность» в «Свойства качественных требований») мы говорили, что требования в стиле «пользователь должен быть в состоянии отправить сообщение» являются некорректными. И это так. Но бывают ситуации, когда проблема малоопасная и состоит только в формулировке. Например, фразы в стиле «пользователь может нажать любую из кнопок», «пользователю должно быть видно главное меню» на самом деле означают, что «все отображаемые кнопки должны быть доступны для нажатия» и «главное меню должно отображаться». Да, эту недоработку тоже стоит исправить, но не следует относить её к критической проблеме.

Скрытое редактирование требований. Эту ошибку можно смело отнести к разряду крайне опасных. Её суть состоит в том, что тестировщик произвольно вносит правки в требования, никак не отмечая этот факт. Соответственно, автор документа, скорее всего, не заметит такой правки, а потом будет очень удивлён, когда в продукте что-то будет реализовано совсем не так, как когда-то было описано в требованиях. Поэтому придерживаемся простой рекомендации: если вы что-то правите, обяза-

тельно отмечайте это (средствами вашего инструмента или явно в тексте). И ещё лучше отмечать правку как предложение по изменению, а не как свершившийся факт, т. к. автор исходного документа может иметь совершенно иной взгляд на ситуацию.

Анализ, не соответствующий уровню требований. При тестировании требований следует постоянно помнить, к какому уровню они относятся, т. к. в противном случае появляются следующие типичные ошибки:

- добавление в бизнес-требования мелких технических подробностей;
- дублирование на уровне пользовательских требований части бизнес-требований (если вы хотите увеличить прослеживаемость набора требований, имеет смысл просто использовать ссылки);
- недостаточная детализация требований уровня продукта (общие фразы, допустимые, например, на уровне бизнес-требований, здесь уже должны быть предельно детализированы, структурированы и дополнены подробной технической информацией).

2.9 Контрольные вопросы и задания

- Сформулируйте определение требования.
- Зачем нужны требования?
- Кто является основным источником, а кто потребителем требований?
- Какова связь требований и архитектуры проекта?
- Чем отличается проектная документация от продуктной?
- Перечислите основные источники требований.
- Назовите основные пути выявления требований.
- Каковы преимущества и недостатки анкетирования как способа определения требований?
- Каковы преимущества и недостатки наблюдения как способа определения требований?
- Каковы преимущества и недостатки самостоятельного определения требований на основе документов?
- Каковы преимущества и недостатки семинаров как способа определения требований?
- Каковы преимущества и недостатки прототипирования как способа определения требований?
- Сделайте сравнительный анализ основных техник выявления требований.
- Какие типы требований вы знаете?
- Опишите уровни требований: как они выражаются и что описывают?
- Какие требования относятся к группе нефункциональных требований?
- Какие требования относятся к группе функциональных требований?
- Опишите, что содержит спецификация.
- Перечислите свойства качественного требования.
- Приведите примеры нарушения такого свойства качественного требования, как завершенность.
- Приведите примеры нарушения такого свойства качественного требования, как атомарность.
- Приведите примеры нарушения такого свойства качественного требования, как непротиворечивость.
- Приведите примеры нарушения такого свойства качественного требования, как недвусмысленность.

- Приведите примеры нарушения такого свойства качественного требования, как выполнимость.
- Приведите примеры нарушения такого свойства качественного требования, как актуальность.
- Приведите примеры нарушения такого свойства качественного требования, как прослеживаемость.
- Приведите примеры нарушения такого свойства качественного требования, как модифицируемость.
- Приведите примеры нарушения такого свойства качественного требования, как корректность.
- Какими свойствами должны обладать наборы требований?
- Назовите основные проблемы с наборами требований.
- Какие виды документации можно тестировать? Перечислите.
- Назовите основные техники тестирования требований.
- Приведите примеры удачных и неудачных вопросов к требованиям.
- Какие проблемы чаще всего возникают при работе с требованиями? В чём их суть?

3 ВИДЫ И НАПРАВЛЕНИЯ ТЕСТИРОВАНИЯ

3.1 Упрощённая классификация тестирования

Тестирование можно классифицировать по большому количеству признаков, и практически в каждой серьёзной книге о тестировании автор представляет свой (безусловно имеющий право на существование) взгляд по этому вопросу.

Текущий материал достаточно объёмный и сложный, а глубокое понимание каждого пункта в классификации требует определённого опыта, поэтому мы разделим данную тему на две: в этом разделе рассмотрим самый простой, минимальный набор информации, необходимый начинающему тестировщику, а в следующем – приведём подробную классификацию.

Используйте нижеприведённый рисунок 3.а как очень краткую «памятку для запоминания».

Рисунок 3.а – Упрощённая классификация тестирования

Итак, тестирование можно классифицировать:

- По запуску кода на исполнение:
  - статическое тестирование – без запуска;
  - динамическое тестирование – с запуском.
- По доступу к коду и архитектуре приложения:
  - метод белого ящика – доступ к коду есть;
  - метод чёрного ящика – доступа к коду нет;
  - метод серого ящика – к части кода доступ есть, к части – нет.
- По степени автоматизации:
  - ручное тестирование – тест-кейсы выполняет человек;
  - автоматизированное тестирование – тест-кейсы частично или полностью выполняет специальное инструментальное средство.
- По уровню детализации приложения (по уровню тестирования):
  - модульное (компонентное) тестирование – проверяются отдельные небольшие части приложения;
  - интеграционное тестирование – проверяется взаимодействие между несколькими частями приложения;
  - системное тестирование – приложение проверяется как единое целое.
- По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):
  - дымовое тестирование (обязательно изучите этимологию термина, хотя бы в Википедии[94]) – проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования данного приложения;

[94] Smoke test. Wikipedia. URL: http://en.wikipedia.org/wiki/Smoke_testing_(electrical).

- тестирование критического пути – проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности;
- расширенное тестирование – проверка всей (остальной) функциональности, заявленной в требованиях.

- По принципам работы с приложением:
  - позитивное тестирование – все действия с приложением выполняются строго по инструкции без недопустимых действий, некорректных данных и т. д. Можно образно сказать, что приложение исследуется в «тепличных условиях»;
  - негативное тестирование – в работе с приложением выполняются некорректные операции и используются данные, потенциально приводящие к ошибкам (классика жанра – деление на нуль).

Внимание! Очень часто встречающаяся ошибка! Негативные тесты НЕ предполагают возникновения в приложении ошибки. Напротив, они предполагают, что верно работающее приложение даже в критической ситуации поведёт себя правильным образом (в примере с делением на нуль, например, отобразит сообщение «Делить на нуль запрещено»).

Часто возникает вопрос о том, чем различаются «тип тестирования», «вид тестирования», «способ тестирования», «подход к тестированию» и т. д. и т. п. Если вас интересует строгий формальный ответ, посмотрите в направлении таких вещей как «таксономия[95]» и «таксон[96]», т. к. сам вопрос выходит за рамки тестирования как такового и относится уже к области науки.

Но исторически так сложилось, что как «тип тестирования» (testing type) и «вид тестирования» (testing kind) давно стали синонимами.

3.2 Подробная классификация тестирования

Теперь мы рассмотрим классификацию тестирования максимально подробно. Настоятельно рекомендуется прочесть не только текст этого раздела, но и все дополнительные источники, на которые будут приведены ссылки.

На рисунках 3.b и 3.c приведена схема, на которой все способы классификации показаны одновременно. Многие авторы, создававшие подобные классификации, использовали интеллект-карты, однако такая техника не позволяет в полной мере отразить тот факт, что способы классификации пересекаются (т. е. некоторые виды тестирования можно отнести к разным способам классификации). На рисунках 3.b и 3.c самые яркие случаи таких пересечений отмечены границей блоков в виде набора точек. Если вы видите на схеме подобный блок – ищите одноимённый где-то в другом виде классификации.

Почему важно обобщать и систематизировать сведения о тестировании в рамках единой классификации? Классификация позволяет упорядочить знания и значительно ускоряет процессы планирования тестирования и разработки тест-кейсов, а также позволяет оптимизировать трудозатраты за счёт того, что тестировщику не приходится изобретать очередной велосипед.

При этом ничто не мешает создавать собственные классификации – как вообще придуманные с нуля, так и представляющие собой комбинации и модификации представленных ниже классификаций.

[95] Таксономия. Wikipedia. URL: https://ru.wikipedia.org/wiki / Таксономия.
[96] Таксон. Wikipedia. URL: https://ru.wikipedia.org/wiki / Таксон.

	Настоятельно рекомендуется в дополнение к материалу этого раздела прочесть: • статью «Классификация видов тестирования» [97]; • книгу Ли Коупленда «Практическое руководство по разработке тестов» Copeland L. «A Practitioner's Guide to Software Test Design»; • заметку «Types of Software Testing: List of 100 Different Testing Types» [98].

	Если вас интересует «эталонная классификация», то… её не существует. Можно сказать, что в материалах[99] ISTQB приведена наиболее обобщённая и общепринятая точка зрения на этот вопрос, но и там нет единой ситемы, которая объединяла бы все варианты классификации. Так что, если вас просят классифицировать тестирование, стоит уточнить, согласно какому автору или источнику спрашивающий ожидает услышать ваш ответ.

Сейчас вы приступите к изучению одного из самых сложных разделов этого пособия. Если вы уже имеете достаточный опыт в тестировании, можете отталкиваться от схемы, чтобы систематизировать и расширить свои знания. Если вы только начинаете заниматься тестированием, рекомендуется сначала прочесть информацию, приведенную за схемой.

	По поводу схем, которые вы сейчас увидите на рисунках 3.b и 3.c, часто поступают вопросы, почему функциональное и нефункциональное тестирования не связаны с соответствующими подвидами. Есть две причины: 1) Несмотря на то что те или иные виды тестирования принято причислять к функциональному или нефункциональному тестированию, в них всё равно присутствуют обе составляющие (как функциональная, так и нефункциональная), пусть и в разных пропорциях. Более того, часто проверить нефункциональную составляющую невозможно, пока не будет реализована соответствующая функциональная составляющая. 2) Схема превратилась бы в непроглядную паутину линий. Поэтому было решено оставить рисунки 3.b и 3.c в том виде, в каком они представлены на следующих двух страницах. Полноразмерный вариант этих рисунков можно скачать здесь[100].

Итак, тестирование можно классифицировать следующим образом (см. рисунки 3.b и 3.c).

[97] Классификация видов тестирования. URL: http://habrahabr.ru/company/npo-comp/blog/223833/.
[98] Types of Software Testing: List of 100 Different Testing Types. URL: http://www.guru99.com/types-of-software-testing.html.
[99] International Software Testing Qualifications Board, Downloads. URL: http://www.istqb.org/downloads.html.
[100] Полноразмерный вариант рисунка 3.b. URL: http://svyatoslav.biz/wp-pics/software_testing_classification_ru.png. Полноразмерный вариант рисунка 3.c. URL: http://svyatoslav.biz/wp-pics/software_testing_classification_en.png.

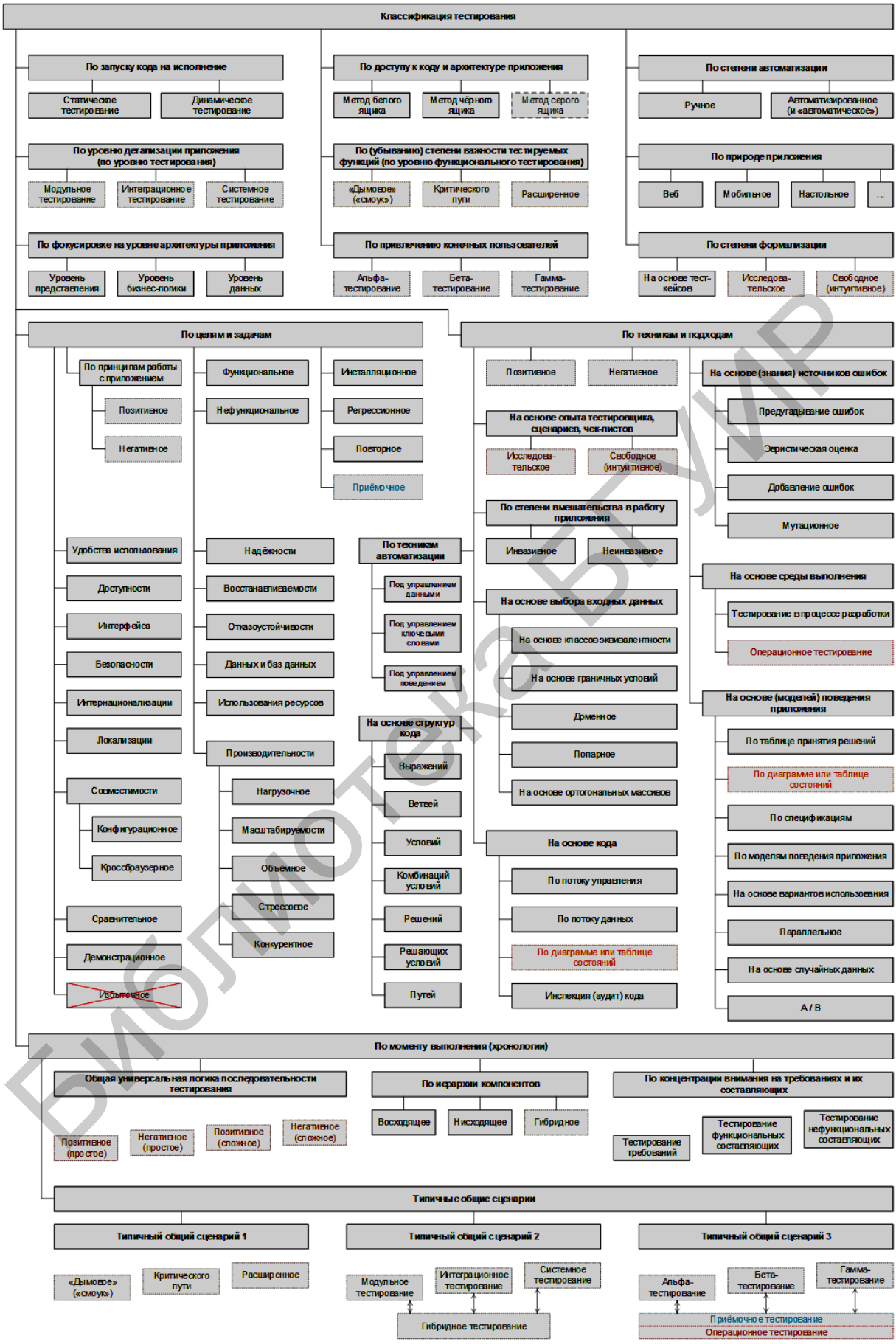


Рисунок 3.b – Классификация тестирования (русскоязычный вариант)

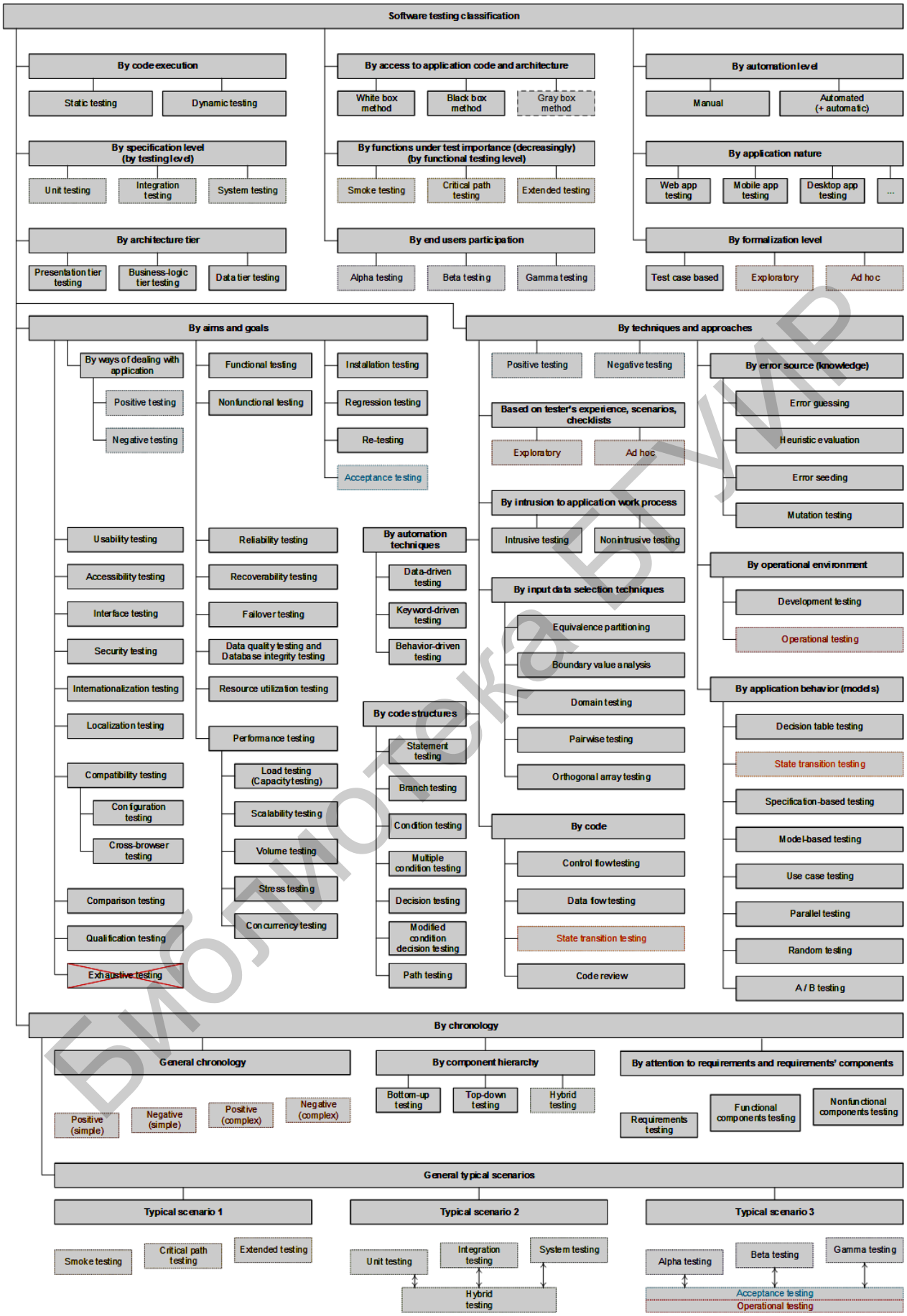


Рисунок 3.с – Классификация тестирования (англоязычный вариант)

Далеко не всякое тестирование предполагает взаимодействие с работающим приложением. Поэтому в рамках **классификации по запуску кода на исполнение** выделяют:

- **Статическое тестирование** (static testing[101]) – тестирование без запуска кода на исполнение. В рамках этого подхода тестированию могут подвергаться:
  - документы (требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т. д.);
  - графические прототипы (например, эскизы пользовательского интерфейса);
  - код приложения (что часто выполняется самими программистами в рамках аудита кода (code review[102]), являющегося специфической вариацией взаимного просмотра (см. подраздел 2.6 «Техники тестирования требований») в применении к исходному коду). Код приложения также можно проверять с использованием техник тестирования на основе структур кода (см. тестирование на основе структур кода дальше в этом разделе);
  - параметры (настройки) среды исполнения приложения;
  - подготовленные тестовые данные.
- **Динамическое тестирование** (dynamic testing[103]) – тестирование с запуском кода на исполнение. Запускаться на исполнение может как код всего приложения целиком (системное тестирование), так и код нескольких взаимосвязанных частей (интеграционное тестирование), отдельных частей (модульное или компонентное тестирование) и даже отдельные участки кода. Основная идея этого вида тестирования состоит в том, что проверяется реальное поведение (части) приложения.

Классификация по доступу к коду и архитектуре приложения:

- **Метод белого ящика** (white box testing[104], open box testing, clear box testing, glass box testing) – у тестировщика есть доступ к внутренней структуре и коду приложения, а также он имеет достаточно знаний для их понимания. Выделяют даже сопутствующую тестированию по методу белого ящика глобальную технику – тестирование на основе дизайна (design-based testing[105]). Для более глубокого изучения сути метода белого ящика рекомендуется ознакомиться с техниками исследования потока управления или потока данных, использования диаграмм состояний. Некоторые авторы склонны тесно связывать этот метод со статическим тестированием, но ничто не мешает тестировщику запустить код на выполнение и при этом периодически обращаться к самому коду (а модульное тестирование и вовсе предполагает запуск кода на исполнение и при этом работу именно с кодом, а не с «приложением целиком»).

[101] **Static testing.** Testing of a software development artifact, e.g. requirements, design or code, without execution of these artifacts, e.g. reviews or static analysis. ISTQB Glossary.
[102] Cohen J. Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.). URL: http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf.
[103] **Dynamic testing.** Testing that involves the execution of the software of a component or system. ISTQB Glossary.
[104] **White box testing.** Testing based on an analysis of the internal structure of the component or system. ISTQB Glossary.
[105] **Design-based Testing.** An approach to testing in which test cases are designed based on the architecture and/or detailed design of a component or system (e.g. tests of interfaces between components or systems). ISTQB Glossary.

- **Метод чёрного ящика** (black box testing[106], 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. При этом абсолютное большинство перечисленных на рисунках 3.b и 3.c видов тестирования работают по методу чёрного ящика, идею которого в альтернативном определении можно сформулировать так: тестировщик оказывает на приложение воздействия (и проверяет реакцию) тем же способом, каким при реальной эксплуатации приложения на него воздействовали бы пользователи или другие приложения. В рамках тестирования по методу чёрного ящика основной информацией для создания тест-кейсов выступает документация (особенно – требования (requirements-based testing[107])) и общий здравый смысл (для случаев, когда поведение приложения в некоторой ситуации не регламентировано явно; иногда это называют «тестированием на основе неявных требований», но определения-канона у этого подхода нет).
- **Метод серого ящика** (gray box testing[108]) – это комбинация методов белого и чёрного ящиков, состоящая в том, что к части кода и архитектуры у тестировщика доступ есть, а к части – нет. На рисунках 3.b и 3.c этот метод обозначен особым пунктиром потому, что его явное упоминание – крайне редкий случай: обычно говорят о методах белого или чёрного ящиков в применении к тем или иным частям приложения, при этом понимая, что «приложение целиком» тестируется по методу серого ящика.

Важно! Некоторые авторы[109] определяют метод серого ящика как противопоставление методам белого и чёрного ящиков, особо подчёркивая, что при работе по методу серого ящика внутренняя структура тестируемого объекта известна частично и выясняется по мере исследования. Этот подход, бесспорно, имеет право на существование, но в своём предельном случае он вырождается до состояния «часть системы мы знаем, часть – не знаем», т. е. до всё той же комбинации белого и чёрного ящиков.

Если сравнить основные преимущества и недостатки перечисленных методов, получается следующая картина (таблица 3.a).

[106] **Black box testing.** Testing, either functional or non-functional, without reference to the internal structure of the component or system. ISTQB Glossary.
[107] **Requirements-based Testing.** An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g. tests that exercise specific functions or probe non-functional attributes such as reliability or usability. ISTQB Glossary.
[108] **Gray box testing** is a software testing method, which is a combination of Black Box Testing method and White Box Testing method. … In Gray Box Testing, the internal structure is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Gray Box Testing Fundamentals.URL: http://softwaretestingfundamentals.com/gray-box-testing.
[109] Gray box testing (gray box) definition. Rouse. URL: http://searchsoftwarequality.techtarget.com /definition/gray-box.

Таблица 3.а – Преимущества и недостатки методов белого, чёрного и серого ящиков

Название метода	Преимущества	Недостатки
Метод белого ящика	• Показывает скрытые проблемы и упрощает их диагностику. • Допускает достаточно простую автоматизацию тест-кейсов и их выполнение на самых ранних стадиях развития проекта. • Обладает развитой системой метрик, сбор и анализ которых легко автоматизируются. • Стимулирует разработчиков к написанию качественного кода. • Многие техники этого метода являются проверенными, хорошо зарекомендовавшими себя решениями, базирующимися на строгом техническом подходе	• Не может выполняться тестировщиками, не обладающими достаточными знаниями в области программирования. • Тестирование сфокусировано на реализованной функциональности, что повышает вероятность пропуска нереализованных требований. • Поведение приложения исследуется в отрыве от реальной среды выполнения и не учитывает её влияние. • Поведение приложения исследуется в отрыве от реальных пользовательских сценариев (см. подраздел 4.6 «Наборы тест-кейсов»)
Метод чёрного ящика	• Тестировщик не обязан обладать (глубокими) знаниями в области программирования. • Поведение приложения исследуется в контексте реальной среды выполнения и учитывает её влияние. • Поведение приложения исследуется в контексте реальных пользовательских сценариев. • Тест-кейсы можно создавать уже на стадии появления стабильных требований. • Процесс создания тест-кейсов позволяет выявить дефекты в требованиях. • Допускает создание тест-кейсов, которые можно многократно использовать на разных проектах	• Возможно повторение части тест-кейсов, уже выполненных разработчиками. • Высока вероятность того, что часть возможных вариантов поведения приложения останется непротестированной. • Для разработки высокоэффективных тест-кейсов необходима качественная документация. • Диагностика обнаруженных дефектов более сложна в сравнении с техниками метода белого ящика. • В связи с широким выбором техник и подходов затрудняется планирование и оценка трудозатрат. • В случае автоматизации могут потребоваться сложные дорогостоящие инструментальные средства
Метод серого ящика	Сочетает преимущества и недостатки методов белого и чёрного ящиков	

Методы белого и чёрного ящика не являются конкурирующими или взаимоисключающими, напротив, они гармонично дополняют друг друга, компенсируя таким образом имеющиеся недостатки.

Классификация по степени автоматизации:

- **Ручное тестирование** (manual testing[110]) – тестирование, в котором тест-кейсы выполняются человеком вручную без использования средств автоматизации. Несмотря на то что это звучит очень просто, от тестировщика в те или иные моменты времени требуются такие качества, как терпеливость, наблюдательность, креативность, умение ставить нестандартные эксперименты, а также умение видеть и понимать, что происходит «внутри системы», т. е. как внешние воздействия на приложение трансформируются в его внутренние процессы.
- **Автоматизированное тестирование** (automated testing, test automation[111]) – набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования. Тест-кейсы частично или полностью выполняет специальное инструментальное средство, однако разработка тест-кейсов, подготовка данных, оценка результатов выполнения, написания отчётов об обнаруженных дефектах – всё это и многое другое по-прежнему исполняет человек.

Некоторые авторы говорят отдельно о «полуавтоматизированном» тестировании как варианте ручного с частичным использованием средств автоматизации и отдельно об «автоматизированном» тестировании (включая туда области тестирования, в которых компьютер выполняет ощутимо больший процент задач). Но т. к. без участия человека всё равно не обходится ни один из этих видов тестирования, не станем увеличивать количество терминов и ограничимся одним понятием «автоматизированное тестирование».

Для автоматизированного тестирования есть много как сильных, так и слабых сторон (таблица 3.b).

Таблица 3.b – Преимущества и недостатки автоматизированного тестирования

Преимущества	Недостатки
• Скорость выполнения тест-кейсов может в разы и на порядки превосходить возможности человека. • Отсутствие влияния человеческого фактора в процессе выполнения тест-кейсов (усталости, невнимательности и т. д.)	• Необходим высококвалифицированный персонал в силу того факта, что автоматизация – это «проект внутри проекта» (со своими требованиями, планами, кодом и т. д.). • Высокие затраты на сложные средства автоматизации, разработку и сопровождение кода тест-кейсов

[110] **Manual testing** is performed by the tester who carries out all the actions on the tested application manually, step by step and indicates whether a particular step was accomplished successfully or whether it failed. Manual testing is always a part of any testing effort. It is especially useful in the initial phase of software development, when the software and its user interface are not stable enough, and beginning the automation does not make sense. Smart Bear Test Complete user manual. URL: http://support.smartbear.com/viewarticle/55004/.

[111] **Test automation** is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Commonly, test automation involves automating a manual process already in place that uses a formalized testing process. Hooda R. V. An Automation of Software Testing: A Foundation for the Future.

Продолжение таблицы 3.b

Преимущества	Недостатки
• Минимизация затрат при многократном выполнении тест-кейсов (участие человека здесь требуется лишь эпизодически).Способность средств автоматизации выполнить тест-кейсы, в принципе непосильные для человека в силу своей сложности, скорости или иных факторов. • Способность средств автоматизации собирать, сохранять, анализировать, агрегировать и представлять в удобной для восприятия человеком форме колоссальные объёмы данных. • Способность средств автоматизации выполнять низкоуровневые действия с приложением, операционной системой, каналами передачи данных и т. д.	• Автоматизация требует более тщательного планирования и управления рисками, т. к. в противном случае проекту может быть нанесён серьёзный ущерб. • Средств автоматизации крайне много, что усложняет проблему выбора того или иного средства и может повлечь за собой финансовые затраты (и риски), необходимость обучения персонала (или поиска специалистов). • В случае ощутимого изменения требований, смены технологического домена, переработки интерфейсов (как пользовательских, так и программных) многие тест-кейсы становятся безнадёжно устаревшими и требуют создания их заново

Если же выразить все преимущества и недостатки автоматизации тестирования одной фразой, то получается, что она позволяет ощутимо увеличить тестовое покрытие (test coverage[112]), но при этом столь же ощутимо увеличивает риски.

Задание 3.a: Составьте аналогичную таблицу с преимуществами и недостатками ручного тестирования. Подсказка: здесь недостаточно просто поменять заголовки колонок с преимуществами и недостатками автоматизации.

Классификация по уровню детализации приложения (по уровню тестирования).

Внимание! Возможна путаница, вызванная тем, что единого общепринятого набора классификаций не существует, а две из них имеют очень схожие названия:

- «По уровню детализации приложения», что эквивалентно «по уровню тестирования»;
- «По (убыванию) степени важности тестируемых функций», что эквивалентно «по уровню **функционального** тестирования».

- **Модульное (компонентное) тестирование** (unit testing, module testing, component testing[113]) направлено на проверку отдельных небольших частей приложения, которые (как правило) можно исследовать изолированно от других подобных частей. При выполнении данного тестирования могут проверяться отдельные функции или методы классов, сами классы, взаимодействие классов, небольшие библиотеки, отдельные части приложения. Часто данный вид

[112] **Coverage, Test coverage.** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite. ISTQB Glossary.

[113] **Module testing, Unit testing, Component testing.** The testing of individual software components. ISTQB Glossary.

тестирования реализуется с использованием специальных технологий и инструментальных средств автоматизации тестирования, значительно упрощающих и ускоряющих разработку соответствующих тест-кейсов.

При переводе на русский язык понятия «модульное тестирование» теряются нюансы степени детализации: «unit-тестирование», как правило, направлено на тестирование атомарных участков кода, «модульное» – на тестирование классов и небольших библиотек, «компонентное» – на тестирование библиотек и структурных частей приложения. Но эта классификация не стандартизирована, и у различных авторов можно встретить совершенно разные взаимоисключающие трактовки.

- **Интеграционное тестирование** (integration testing[114], component integration testing[115], pairwise integration testing[116], system integration testing[117], incremental testing[118], interface testing[119], thread testing[120]) направлено на проверку взаимодействия между несколькими частями приложения (каждая из которых, в свою очередь, проверена отдельно на стадии модульного тестирования). К сожалению, даже если мы работаем с очень качественными отдельными компонентами, «на стыке» их взаимодействия часто возникают проблемы. Именно эти проблемы и выявляет интеграционное тестирование. (См. также техники восходящего, нисходящего и гибридного тестирования в хронологической классификации по иерархии компонентов.)
- **Системное тестирование** (system testing[121]) направлено на проверку всего приложения как единого целого, собранного из частей, проверенных на двух предыдущих стадиях. Здесь не только выявляются дефекты «на стыках» компонентов, но и появляется возможность полноценно взаимодействовать с приложением с точки зрения конечного пользователя, применяя множество других видов тестирования, перечисленных в данном разделе.

С классификацией по уровню детализации приложения связан досадный факт: если предыдущая стадия обнаружила проблемы, то на следующей стадии эти проблемы точно нанесут удар по качеству; если же предыдущая стадия не обнаружила проблем, это ещё не гарантия защиты от проблем на следующей стадии.

---

[114] **Integration testing.** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. ISTQB Glossary.

[115] **Component integration testing.** Testing performed to expose defects in the interfaces and interaction between integrated components. ISTQB Glossary.

[116] **Pairwise integration testing.** A form of integration testing that targets pairs of components that work together, as shown in a call graph. ISTQB Glossary.

[117] **System integration testing.** Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet). ISTQB Glossary.

[118] **Incremental testing.** Testing where components or systems are integrated and tested one or some at a time, until all the components or systems are integrated and tested. ISTQB Glossary.

[119] **Interface testing.** An integration test type that is concerned with testing the interfaces between components or systems. ISTQB Glossary.

[120] **Thread testing.** An approach to component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy. ISTQB Glossary.

[121] **System testing.** The process of testing an integrated system to verify that it meets specified requirements. ISTQB Glossary.

Для лучшего запоминания степени детализации в модульном, интеграционном и системном тестированиях показаны схематично на рисунке 3.d.

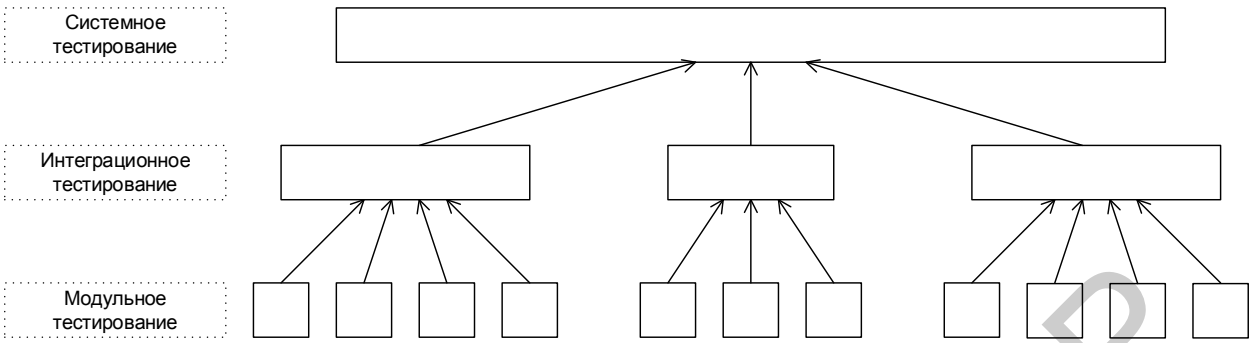


Рисунок 3.d – Схематичное представление классификации тестирования по уровню детализации приложения

Если обратиться к словарю ISTQB за определением уровня тестирования (test level[122]), то можно увидеть, что аналогичное разбиение на модульное, интеграционное и системное тестирования, к которым добавлено ещё и приёмочное тестирование, используется в контексте разделения областей ответственности на проекте. Но такая классификация больше относится к вопросам управления проектом, чем к тестированию в чистом виде, а потому выходит за рамки рассматриваемых нами вопросов.

	С самым полным вариантом классификации тестирования по уровню тестирования можно ознакомиться в статье «What are Software Testing Levels?[123]». Для удобства восприятия отразим эту концепцию на рисунке 3.e, но отметим, что это скорее общий теоретический взгляд.

[122] **Test level.** A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test. ISTQB Glossary.

[123] What are Software Testing Levels? URL: http://istqbexamcertification.com/what-are-software-testing-levels/.

Рисунок 3.e – Самый полный вариант классификации тестирования по уровню тестирования

Классификация по (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):

В некоторых источниках эту разновидность классификации также называют «по глубине тестирования».

- **Дымовое тестирование** (smoke test[124], intake test[125], build verification test[126]) направлено на проверку самой главной, самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения (или иного объекта, подвергаемого дымовому тестированию).

[124] **Smoke test, Confidence test, Sanity test.** A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details. ISTQB Glossary.

[125] **Intake test.** A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase. ISTQB Glossary.

[126] **Build verification test.** A set of automated tests which validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g. agile projects) and it is run on every new build before the build is released for further testing. ISTQB Glossary.

Внимание! Очень распространённая проблема! Из-за особенности перевода на русский язык под термином «приёмочное тестирование» часто может пониматься как «smoke test» (см. подраздел 2.3 «Подробная классификация тестирования»), так и «acceptance test», которые изначально не имеют между собою ничего общего. По этой причине многие тестировщики почти не используют русский перевод «дымовое тестирование», а так и говорят – «смоук-тест».

Дымовое тестирование проводится после выхода нового билда, чтобы определить общий уровень качества приложения и принять решение о (не)целесообразности выполнения тестирования критического пути и расширенного тестирования. Поскольку тест-кейсов на уровне дымового тестирования относительно немного, а сами они достаточно просты, но при этом очень часто повторяются, они являются хорошими кандидатами на автоматизацию. В связи с высокой важностью тест-кейсов на данном уровне пороговое значение метрики их прохождения часто выставляется равным 100 % или близким к 100 %.

Очень часто сталкиваемся с вопросом о том, чем «smoke test» отличается от «sanity test». В глоссарии ISTQB сказано просто: «sanity test: See smoke test». Но некоторые авторы утверждают[127], что разница[128] есть, и отразим её на рисунке 3.f.

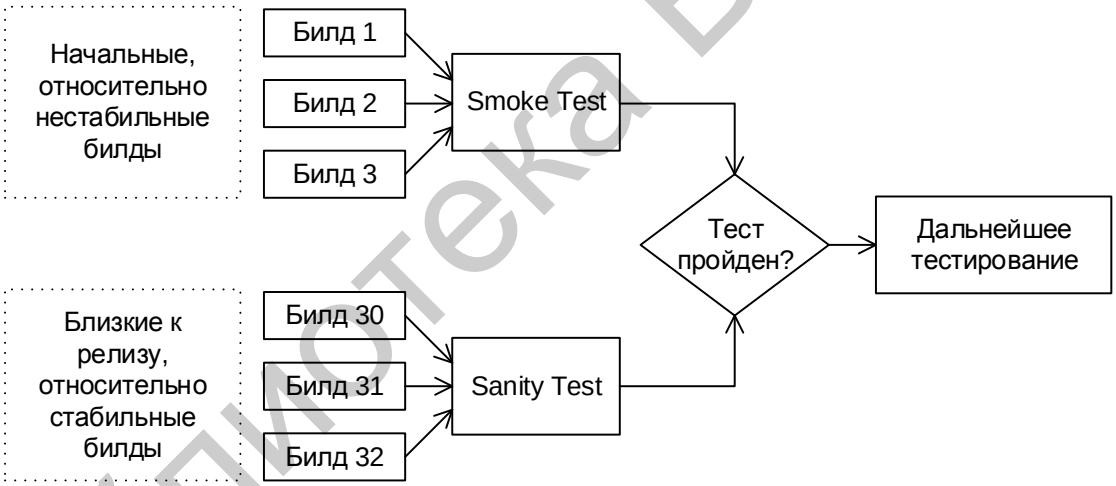


Рисунок 3.f – Трактовка разницы между smoke test и sanity test

- **Тестирование критического пути** (critical path[129] test) направлено на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. Как видно из определения в сноске к англоязычной версии термина, сама идея заимствована из управления проектами и трансформирована в контексте тестирования в следующую: существует

[127] Smoke Vs Sanity Testing – Introduction and Differences. URL: http://www.guru99.com/smoke-sanity-testing.html.

[128] Smoke testing and sanity testing – Quick and simple differences. URL: http://www.softwaretestinghelp.com/smoke-testing-and-sanity-testing-difference/.

[129] **Critical path.** Longest sequence of activities in a project plan which must be completed on time for the project to complete on due date. An activity on the critical path cannot be started until its predecessor activity is complete; if it is delayed for a day, the entire project will be delayed for a day unless the activity following the delayed activity is completed a day earlier. URL: http://www.businessdictionary.com/definition/critical-path.html.

большинство пользователей, которые чаще всего используют некое подмножество функций приложения (рисунок 3.g). Именно эти функции и нужно проверить, как только мы убедились, что приложение «в принципе работает» (дымовой тест прошёл успешно). Если по каким-то причинам приложение не выполняет эти функции или выполняет их некорректно, очень многие пользователи не смогут достичь множества своих целей. Пороговое значение метрики успешного прохождения «теста критического пути» уже немного ниже, чем в дымовом тестировании, но всё равно достаточно высоко (как правило, порядка 70–90 % – в зависимости от сути проекта).

- **Расширенное тестирование** (extended test[130]) направлено на исследование всей заявленной в требованиях функциональности – даже той, которая низко проранжирована по степени важности. При этом здесь также учитывается, какая функциональность является более важной, а какая – менее. Но при наличии достаточного количества времени и иных ресурсов тест-кейсы этого уровня могут затронуть даже самые низкоприоритетные требования.

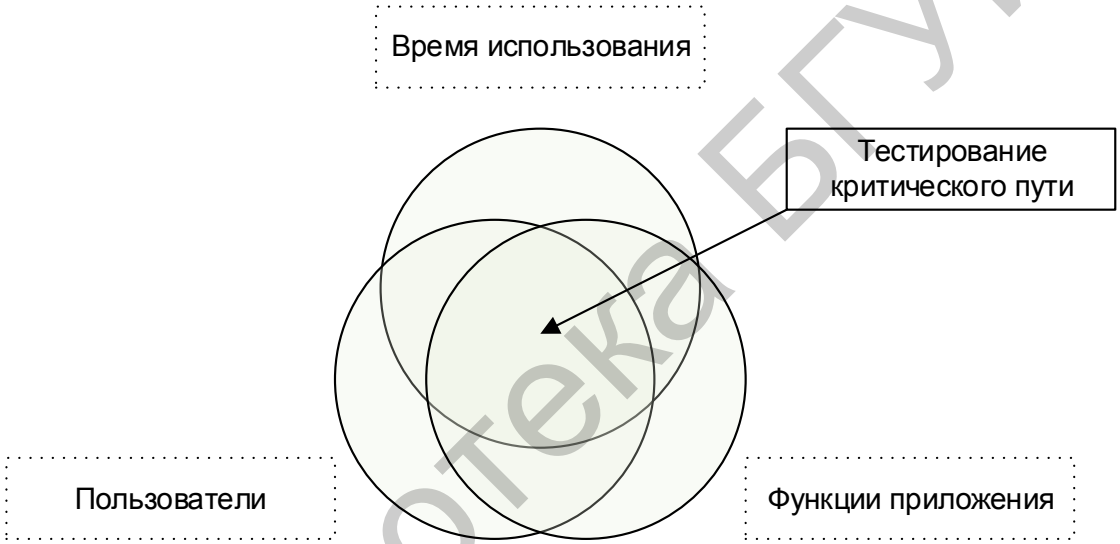


Рисунок 3.g – Суть тестирования критического пути

Ещё одним направлением исследования в рамках данного тестирования являются нетипичные, маловероятные, экзотические случаи и сценарии использования функций и свойств приложения, затронутых на предыдущих уровнях. Пороговое значение метрики успешного прохождения расширенного тестирования существенно ниже, чем в тестировании критического пути (иногда можно увидеть даже значения в диапазоне 30–50 %, т. к. подавляющее большинство найденных здесь дефектов не представляет угрозы для успешного использования приложения большинством пользователей).

[130] **Extended test.** The idea is to develop a comprehensive application system test suite by modeling essential capabilities as extended use cases. By Kuijt R. Extended Use Case Test Design Pattern.

	К сожалению, часто можно встретить мнение, что дымовое тестирование, тестирование критического пути и расширенное тестирование напрямую связаны с позитивным тестированием и негативным тестированием, и негативное появляется только на уровне тестирования критического пути. Это не так. Как позитивные, так и негативные тесты могут (а иногда и обязаны) встречаться на всех перечисленных уровнях. Например, деление на нуль в калькуляторе явно должно относиться к дымовому тестированию, хотя это яркий пример негативного тест-кейса.

Для лучшего запоминания отразим эту классификацию графически:

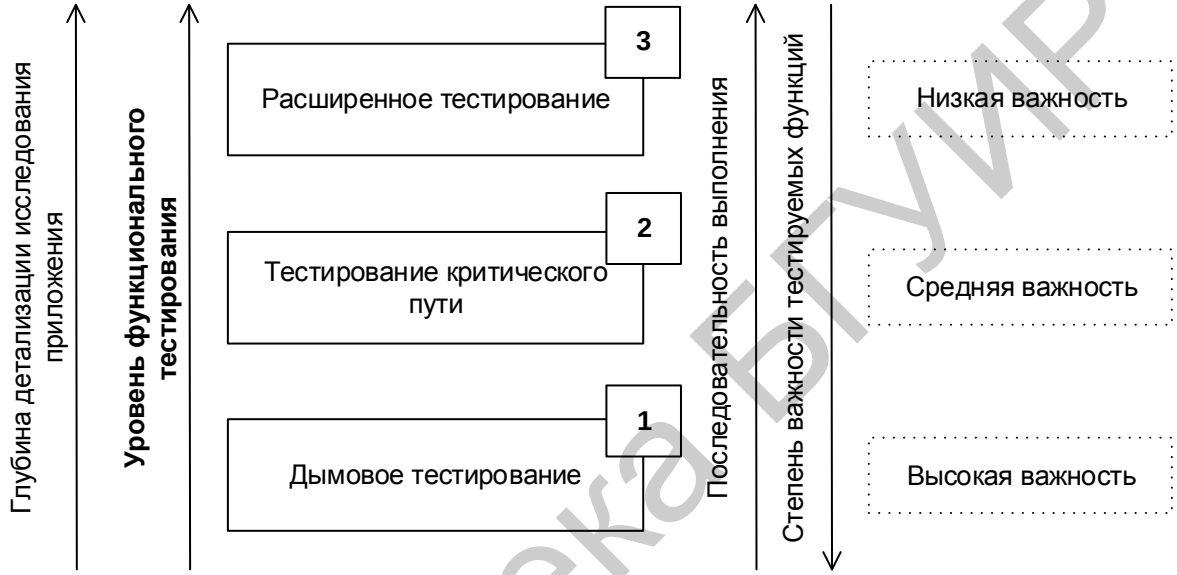


Рисунок 3.h – Классификация тестирования по (убыванию) степени важности тестируемых функций (по уровню функционального тестирования)

Классификация по принципам работы с приложением:

- **Позитивное тестирование** (positive testing[131]) направлено на исследование приложения в ситуации, когда все действия выполняются строго по инструкции без каких либо ошибок, отклонений, ввода неверных данных и т. д. Если позитивные тест-кейсы завершаются ошибками – это тревожный признак, значит приложение работает неверно даже в идеальных условиях (и можно предположить, что в неидеальных условиях оно работает ещё хуже). Для ускорения тестирования несколько позитивных тест-кейсов можно объединять (например, перед отправкой заполнить все поля формы верными значениями) – иногда это может усложнить диагностику ошибки, но существенная экономия времени компенсирует этот риск.

[131] **Positive testing** is testing process where the system validated against the valid input data. In this testing tester always check for only valid set of values and check if application behaves as expected with its expected inputs. The main intention of this testing is to check whether software application not showing error when not supposed to & showing error when supposed to. Such testing is to be carried out keeping positive point of view & only execute the positive scenario. Positive Testing always tries to prove that a given product and project always meets the requirements and specifications. URL: http://www.softwaretestingclass.com/positive-and-negative-testing-in-software-testing/.

- **Негативное тестирование** (negative testing[132], invalid testing[133]) направлено на исследование работы приложения в ситуациях, когда с ним выполняются (некорректные) операции и/или используются данные, потенциально приводящие к ошибкам (классика жанра – деление на нуль). Поскольку в реальной жизни таких ситуаций значительно больше (пользователи допускают ошибки, злоумышленники осознанно «ломают» приложение, в среде работы приложения возникают проблемы и т. д.), негативных тест-кейсов оказывается значительно больше, чем позитивных (иногда – в разы или даже на порядки). В отличие от позитивных негативные тест-кейсы не стоит объединять, т. к. подобное решение может привести к неверной трактовке поведения приложения и пропуску (улучшению) дефектов.

Классификация по природе приложения:

Данный вид классификации является искусственным, поскольку «внутри» речь будет идти об одних и тех же видах тестирования, отличающихся в данном контексте лишь концентрацией на соответствующих функциях и особенностях приложения, использованием специфических инструментов и отдельных техник.

- **Тестирование веб-приложений** (web-applications testing) сопряжено с интенсивной деятельностью в области тестирования совместимости (см. подраздел 3.1 «Упрощённая классификация тестирования») (в особенности – кросс-браузерного тестирования), тестирования производительности, автоматизации тестирования с использованием широкого спектра инструментальных средств.
- **Тестирование мобильных приложений** (mobile applications testing) также требует повышенного внимания к тестированию совместимости, оптимизации производительности (в том числе клиентской части с точки зрения снижения энергопотребления), автоматизации тестирования с применением эмуляторов мобильных устройств.
- **Тестирование настольных приложений** (desktop applications testing) является самым классическим среди всех перечисленных в данной классификации, и его особенности зависят от предметной области приложения, нюансов архитектуры, ключевых показателей качества и т. д.

Эту классификацию можно продолжать очень долго. Например, можно отдельно рассматривать тестирование консольных приложений (console applications testing) и приложений с графическим интерфейсом (GUI-applications testing), серверных приложений (server applications testing) и клиентских приложений (client applications testing) и т. д.

Классификация по фокусировке на уровне архитектуры приложения (данный вид классификации, как и предыдущий, также является искусственным и отражает лишь концентрацию внимания на отдельной части приложения):

- **Тестирование уровня представления** (presentation tier testing) сконцентрировано на той части приложения, которая отвечает за взаимодействие с «внешним миром» (как пользователями, так и другими приложениями). Здесь исследуются вопросы удобства использования, скорости отклика интерфейса, совместимости с браузерами, корректности работы интерфейсов.
- **Тестирование уровня бизнес-логики** (business logic tier testing) отвечает за проверку основного набора функций приложения и строится на базе ключевых

[132] **Negative testing.** Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique, e.g. testing with invalid input values or exceptions. ISTQB Glossary.

[133] **Invalid testing.** Testing using input values that should be rejected by the component or system. ISTQB Glossary.

требований к приложению, бизнес-правил и общей проверки функциональности.

- **Тестирование уровня данных** (data tier testing) сконцентрировано на той части приложения, которая отвечает за хранение и некоторую обработку данных (чаще всего – в базе данных или ином хранилище). Здесь особый интерес представляет тестирование данных, проверка соблюдения бизнес-правил, тестирование производительности.

Если вы не знакомы с понятием многоуровневой архитектуры приложений, ознакомьтесь с ним хотя бы по материалу[134] из Википедии.

Классификация по привлечению конечных пользователей (все три перечисленных ниже вида тестирования относятся к операционному тестированию):

- **Альфа-тестирование** (alpha testing[135]) выполняется внутри организации-разработчика с возможным частичным привлечением конечных пользователей. Может являться формой внутреннего приёмочного тестирования. В некоторых источниках отмечается, что это тестирование должно проводиться без привлечения команды разработчиков, но другие источники не выдвигают такого требования. Суть этого вида: продукт уже можно периодически показывать внешним пользователям, но он ещё достаточно «сырой», потому основное тестирование выполняется организацией-разработчиком.
- **Бета-тестирование** (beta testing[136]) выполняется вне организации-разработчика с активным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования. Суть этого вида: продукт уже можно открыто показывать внешним пользователям, он уже достаточно стабилен, но проблемы всё ещё могут встречаться, и для их выявления нужна обратная связь от реальных пользователей.
- **Гамма-тестирование** (gamma testing[137]) – финальная стадия тестирования перед выпуском продукта, направленная на исправление незначительных дефектов, обнаруженных в бета-тестировании. Как правило, также выполняется с максимальным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования. Суть этого вида: продукт уже почти готов, и сейчас обратная связь от реальных пользователей используется для устранения последних недоработок.

Классификация по степени формализации:

[134] Multitier architecture. Wikipedia. URL: http://en.wikipedia.org/wiki/Multitier_architecture.

[135] **Alpha testing.** Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing. ISTQB Glossary.

[136] **Beta testing.** Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing for off-the-shelf software in order to acquire feedback from the market. ISTQB Glossary.

[137] **Gamma testing** is done when software is ready for release with specified requirements, this testing done directly by skipping all the in-house testing activities. The software is almost ready for final release. No feature development or enhancement of the software is undertaken and tightly scoped bug fixes are the only code. Gamma check is performed when the application is ready for release to the specified requirements and this check is performed directly without going through all the testing activities at home. URL: http://www.360logica.com/blog/2012/06/what-are-alpha-beta-and-gamma-testing.html.

- **Тестирование на основе тест-кейсов** (scripted testing[138], test case based testing) – формализованный подход, в котором тестирование производится на основе заранее подготовленных тест-кейсов, наборов тест-кейсов и иной документации. Это самый распространённый способ тестирования, который также позволяет достичь максимальной полноты исследования приложения за счёт строгой систематизации процесса, удобства применения метрик и широкого набора выработанных за десятилетия и проверенных на практике рекомендаций.
- **Исследовательское тестирование** (exploratory testing[139]) – частично формализованный подход, в рамках которого тестировщик выполняет работу с приложением по выбранному сценарию, который, в свою очередь, дорабатывается в процессе выполнения с целью более полного исследования приложения. Ключевым фактором успеха при выполнении исследовательского тестирования является именно работа по сценарию, а не выполнение разрозненных бездумных операций. Существует даже специальный сценарный подход, называемый сессионным тестированием (session-based testing[140]). В качестве альтернативы сценариям при выборе действий с приложением иногда могут использоваться чек-листы, и тогда этот вид тестирования называют тестированием на основе чек-листов (checklist-based testing[141]).

Дополнительную информацию об исследовательском тестировании можно получить из статьи Джеймса Баха «Что такое исследовательское тестирование?[142]»

- **Свободное (интуитивное) тестирование** (ad hoc testing[143]) – полностью неформализованный подход, в котором не предполагается использования ни тест-кейсов, ни чек-листов, ни сценариев – тестировщик полностью опирается на свой профессионализм и интуицию (experience-based testing[144]) для спонтанного выполнения с приложением действий, которые, по его мнению, могут обнаружить ошибку. Этот вид тестирования используется редко и исключительно как дополнение к полностью или частично формализованному тестированию в случаях, когда для исследования некоторого аспекта поведения приложения (пока?) нет тест-кейсов.

[138] **Scripted testing.** Test execution carried out by following a previously documented sequence of tests. ISTQB Glossary.
[139] **Exploratory testing.** An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. ISTQB Glossary.
[140] **Session-based Testing.** An approach to testing in which test activities are planned as uninterrupted sessions of test design and execution, often used in conjunction with exploratory testing. ISTQB Glossary.
[141] **Checklist-based Testing.** An experience-based test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified. ISTQB Glossary.
[142] Bach J. What is Exploratory Testing? URL: http://www.satisfice.com/articles/what_is_et.shtml.
[143] **Ad hoc testing.** Testing carried out informally; no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and arbitrariness guides the test execution activity. ISTQB Glossary.
[144] **Experience-based Testing.** Testing based on the tester's experience, knowledge and intuition. ISTQB Glossary.

Ни в коем случае не стоит путать исследовательское и свободное направления тестирования. Это разные техники исследования приложения с разной степенью формализации с разными задачами и областями применения.

Классификация тестирования по целям и задачам:

- **Позитивное тестирование** (см. с. 53, с. 67).
- **Негативное тестирование** (см. с. 53, с. 68).
- **Функциональное тестирование** (functional testing[145]) – вид тестирования, направленный на проверку корректности работы функциональности приложения (корректность реализации функциональных требований (см. подраздел 2.4 «Уровни и типы требований»). Часто функциональное тестирование ассоциируют с тестированием по методу чёрного ящика, однако и методом белого ящика вполне можно проверять корректность реализации функциональности.

Часто возникает вопрос, в чём разница между функциональным тестированием (functional testing) и тестированием функциональности (functionality testing[146]). Подробнее о функциональном тестировании изложено в статье «What is Functional testing (Testing of functions) in software?»[147], а о тестировании функциональности в статье «What is functionality testing in software?»[148].
Конспективно:

- функциональное тестирование (как антоним нефункционального) направлено на проверку того, какие функции приложения реализованы и что они работают верным образом;
- тестирование функциональности направлено на те же задачи, но акцент смещён в сторону исследования приложения в реальной рабочей среде, после локализации и в тому подобных ситуациях.

- **Нефункциональное тестирование** (non-functional testing[149]) – вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований (см. подраздел 2.4 «Уровни и типы требований»)), таких как удобство использования, совместимость, производительность, безопасность и т. д.
- **Инсталляционное тестирование** (installation testing, installability testing[150]) – тестирование, направленное на выявление дефектов, влияющих на протекание стадии инсталляции (установки) приложения. В общем случае такое тестирование проверяет множество сценариев и аспектов работы инсталлятора в таких ситуациях, как:

[145] **Functional testing.** Testing based on an analysis of the specification of the functionality of a component or system. ISTQB Glossary.
[146] **Functionality testing.** The process of testing to determine the functionality of a software product (the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions). ISTQB Glossary.
[147] What is Functional testing (Testing of functions) in software? URL: http://istqbexamcertification.com/what-is-functional-testing-testing-of-functions-in-software/.
[148] What is functionality testing in software? URL: http://istqbexamcertification.com/what-is-functionality-testing-in-software/.
[149] **Non-functional testing.** Testing the attributes of a component or system that do not relate to functionality, e. g. reliability, efficiency, usability, maintainability and portability. ISTQB Glossary.
[150] **Installability testing.** The process of testing the installability of a software product. Installability is the capability of the software product to be installed in a specified environment. ISTQB Glossary.

    - новая среда исполнения, в которой приложение ранее не было инсталлировано;
    - обновление существующей версии («апгрейд»);
    - изменение текущей версии на более старую («даунгрейд»);
    - повторная установка приложения с целью устранения возникших проблем («переинсталляция»);
    - повторный запуск инсталляции после ошибки, приведшей к невозможности продолжения инсталляции;
    - удаление приложения;
    - установка нового приложения из семейства приложений;
    - автоматическая инсталляция без участия пользователя.
- **Регрессионное тестирование** (regression testing[151]) – тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования. Фредерик Брукс в своей книге «Мифический человеко-месяц»[152] писал: «Фундаментальная проблема при сопровождении программ состоит в том, что исправление одной ошибки с большой вероятностью (20–50 %) влечёт появление новой». Потому регрессионное тестирование является неотъемлемым инструментом обеспечения качества и активно используется практически в любом проекте.
- **Повторное тестирование** (re-testing[153], confirmation testing) – выполнение тест-кейсов, которые ранее обнаружили дефекты, с целью подтверждения устранения дефектов. Фактически этот вид тестирования сводится к действиям на финальной стадии жизненного цикла отчёта о дефекте (см. подраздел 5.2 «Отчёт о дефекте и его жизненный цикл»), направленным на то, чтобы перевести дефект в состояние «проверен» и «закрыт».
- **Приёмочное тестирование** (acceptance testing[154]) – формализованное тестирование, направленное на проверку приложения с точки зрения конечного пользователя/заказчика и вынесения решения о том, принимает ли заказчик работу у исполнителя (проектной команды). Можно выделить следующие подвиды приёмочного тестирования (хотя упоминают их крайне редко, ограничиваясь в основном общим термином «приёмочное тестирование»):
    - **производственное приёмочное тестирование** (factory acceptance testing[155]) – выполняемое проектной командой исследование полноты и качества реализации приложения с точки зрения его готовности к передаче заказчику. Этот вид тестирования часто рассматривается как синоним альфа-тестирования;

[151] **Regression testing.** Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed. ISTQB Glossary.

[152] Brooks F. The Mythical Man-Month.

[153] **Re-testing, Confirmation testing.** Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions. ISTQB Glossary.

[154] **Acceptance Testing.** Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. ISTQB Glossary.

[155] **Factory acceptance testing.** Acceptance testing conducted at the site at which the product is developed and performed by employees of the supplier organization, to determine whether or not a component or system satisfies the requirements, normally including hardware as well as software. ISTQB Glossary.

    - **операционное приёмочное тестирование** (operational acceptance testing[156], production acceptance testing) – операционное тестирование (см. далее в этом подразделе), выполняемое с точки зрения выполнения инсталляции, потребления приложением ресурсов, совместимости с программной и аппаратной платформой и т. д.;
    - **итоговое приёмочное тестирование** (site acceptance testing[157]) – тестирование конечными пользователями (представителями заказчика) приложения в реальных условиях эксплуатации с целью вынесения решения о том, требует ли приложение доработок или может быть принято в эксплуатацию в текущем виде.
- **Операционное тестирование** (operational testing[158]) – тестирование, проводимое в реальной или приближенной к реальной операционной среде (operational environment[159]), включающей операционную систему, системы управления базами данных, серверы приложений, веб-серверы, аппаратное обеспечение и т. д.
- **Тестирование удобства использования** (usability[160] testing) – тестирование, направленное на исследование того, насколько конечному пользователю понятно, как работать с продуктом (understandability[161], learnability[162], operability[163]), а также на то, насколько ему нравится использовать продукт (attractiveness[164]). И это не оговорка – очень часто успех продукта зависит именно от эмоций, которые он вызывает у пользователей. Для эффективного проведения этого вида тестирования требуется реализовать достаточно серьёзные исследования с привлечением конечных пользователей, проведением маркетинговых исследований и т. д.
- **Тестирование интерфейса** (interface testing[165]) – тестирование, направленное на проверку интерфейсов приложения или его компонентов. По определению ISTQB-глоссария этот вид тестирования относится к интеграционному тести-

[156] **Operational acceptance testing, Production acceptance testing.** Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or systems administration staff focusing on operational aspects, e.g. recoverability, resource-behavior, installability and technical compliance. ISTQB Glossary.

[157] **Site acceptance testing.** Acceptance testing by users/customers at their site, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes, normally including hardware as well as software. ISTQB Glossary.

[158] **Operational testing.** Testing conducted to evaluate a component or system in its operational environment. ISTQB Glossary.

[159] **Operational environment.** Hardware and software products installed at users' or customers' sites where the component or system under test will be used. The software may include operating systems, database management systems, and other applications. ISTQB Glossary.

[160] **Usability.** The capability of the software to be understood, learned, used and attractive to the user when used under specified conditions. ISTQB Glossary.

[161] **Understandability.** The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. ISTQB Glossary.

[162] **Learnability.** The capability of the software product to enable the user to learn its application. ISTQB Glossary.

[163] **Operability.** The capability of the software product to enable the user to operate and control it. ISTQB Glossary.

[164] **Attractiveness.** The capability of the software product to be attractive to the user. ISTQB Glossary.

[165] **Interface Testing.** An integration test type that is concerned with testing the interfaces between components or systems. ISTQB Glossary.

рованию, и это вполне справедливо для таких его вариаций, как тестирование интерфейса прикладного программирования (API testing[166]) и интерфейса командной строки (CLI testing[167]), хотя последнее может выступать и как разновидность тестирования пользовательского интерфейса, если через командную строку с приложением взаимодействует пользователь, а не другое приложение. Однако многие источники предлагают включить в состав тестирования интерфейса и тестирование непосредственно интерфейса пользователя (GUI testing[168]).

Важно! Тестирование интерфейса пользователя (GUI testing[168]) и тестирование удобства использования (usability[160] testing) – разные параметры, не одно и то же! Например, удобный интерфейс может работать некорректно, а корректно работающий интерфейс может быть неудобным.

- **Тестирование доступности** (accessibility testing[169]) – тестирование, направленное на исследование пригодности продукта к использованию людьми с ограниченными возможностями (слабым зрением и т. д.).
- **Тестирование безопасности** (security testing[170]) – тестирование, направленное на проверку способности приложения противостоять злонамеренным попыткам получения доступа к данным или функциям, права на доступ к которым у злоумышленника нет.

Подробнее про этот вид тестирования изложено в статье «What is Security testing in software testing?»[171].

- **Тестирование интернационализации** (internationalization testing, i18n testing, globalization[172] testing, localizability[173] testing) – тестирование, направленное на проверку готовности продукта к работе с использованием различных языков и с учётом различных национальных и культурных особенностей. Этот вид тестирования не подразумевает проверки качества соответствующей адаптации (этим занимается тестирование локализации, см. следующий пункт), оно сфо-

[166] **API testing.** Testing performed by submitting commands to the software under test using programming interfaces of the application directly. ISTQB Glossary.
[167] **CLI testing.** Testing performed by submitting commands to the software under test using a dedicated command-line interface. ISTQB Glossary.
[168] **GUI testing.** Testing performed by interacting with the software under test via the graphical user interface. ISTQB Glossary.
[169] **Accessibility testing.** Testing to determine the ease by which users with disabilities can use a component or system. ISTQB Glossary.
[170] **Security testing.** Testing to determine the security of the software product. ISTQB Glossary.
[171] What is Security testing in software testing? URL: http://istqbexamcertification.com/what-is-security-testing-in-software/.
[172] **Globalization.** The process of developing a program core whose features and code design are not solely based on a single language or locale. Instead, their design is developed for the input, display, and output of a defined set of Unicode-supported language scripts and data related to specific locales. «Globalization Step-by-Step». URL: https://msdn.microsoft.com/en-us/goglobal/bb688112.
[173] **Localizability.** The design of the software code base and resources such that a program can be localized into different language editions without any changes to the source code. «Globalization Step-by-Step». URL: https://msdn.microsoft.com/en-us/goglobal/bb688112.

кусировано именно на проверке возможности такой адаптации (например: что будет, если открыть файл с иероглифом в имени; как будет работать интерфейс, если всё перевести на японский; может ли приложение искать данные в тексте на корейском и т. д.).

- **Тестирование локализации** (localization testing[174], l10n) – тестирование, направленное на проверку корректности и качества адаптации продукта к использованию на том или ином языке с учётом национальных и культурных особенностей. Это тестирование следует за тестированием интернационализации и проверяет корректность перевода и адаптации продукта, а не готовность продукта к таким действиям.
- **Тестирование совместимости** (compatibility testing, interoperability testing[175]) – тестирование, направленное на проверку способности приложения работать в указанном окружении. Здесь, например, может проверяться:
  - совместимость с аппаратной платформой, операционной системой и сетевой инфраструктурой (конфигурационное тестирование, configuration testing[176]);
  - совместимость с браузерами и их версиями (кросс-браузерное тестирование, cross-browser testing[177]);
  - совместимость с мобильными устройствами (mobile testing[178]) и т. д.

  В некоторых источниках к тестированию совместимости добавляют (хоть и подчёркивая, что это не его часть) так называемое тестирование соответствия (compliance testing[179], conformance testing, regulation testing).

[174] **Localization testing** checks the quality of a product's localization for a particular target culture/locale. This test is based on the results of globalization testing, which verifies the functional support for that particular culture/locale. Localization testing can be executed only on the localized version of a product. Localization Testing. URL: https://msdn.microsoft.com/en-us/library/aa292138%28v=vs.71%29.aspx.

[175] **Compatibility Testing, Interoperability Testing.** The process of testing to determine the interoperability of a software product (the capability to interact with one or more specified components or systems). ISTQB Glossary.

[176] **Configuration Testing, Portability Testing.** The process of testing to determine the portability of a software product (the ease with which the software product can be transferred from one hardware or software environment to another). ISTQB Glossary.

[177] **Cross-browser testing** helps you ensure that your web site or web application functions correctly in various web browsers. Typically, QA engineers create individual tests for each browser or create tests that use lots of conditional statements that check the browser type used and execute browser-specific commands. URL: http://support.smartbear.com/viewarticle/55299/.

[178] **Mobile testing** is a testing with multiple operating systems (and different versions of each OS, especially with Android), multiple devices (different makes and models of phones, tablets, phablets), multiple carriers (including international ones), multiple speeds of data transference (3G, LTE, Wi-Fi), multiple screen sizes (and resolutions and aspect ratios), multiple input controls (including BlackBerry's eternal physical keypads), and multiple technologies – GPS, accelerometers – that web and desktop apps almost never use. URL: http://smartbear.com/all-resources/articles/what-is-mobile-testing/.

[179] **Compliance testing, Conformance testing, Regulation testing.** The process of testing to determine the compliance of the component or system (the capability to adhere to standards, conventions or regulations in laws and similar prescriptions). ISTQB Glossary.

Рекомендуется ознакомиться с дополнительным материалом по тестированию совместимости с мобильными платформами в статьях «What is Mobile Testing?»[180] и «Beginner's Guide to Mobile Application Testing»[181].

- **Тестирование данных** (data quality[182] testing) и баз данных (database integrity testing[183]) – два близких по смыслу вида тестирования, направленных на исследование таких характеристик данных, как полнота, непротиворечивость, целостность, структурированность и т. д. В контексте баз данных исследованию может подвергаться адекватность модели предметной области, способность модели обеспечивать целостность и консистентность данных, корректность работы триггеров, хранимых процедур и т. д.
- **Тестирование использования ресурсов** (resource utilization testing[184], efficiency testing[185], storage testing[186]) – совокупность видов тестирования, проверяющих эффективность использования приложением доступных ему ресурсов и зависимость результатов работы приложения от количества доступных ему ресурсов. Часто эти виды тестирования прямо или косвенно примыкают к техникам тестирования производительности.
- **Сравнительное тестирование** (comparison testing[187]) – тестирование, направленное на сравнительный анализ преимуществ и недостатков разрабатываемого продукта по отношению к его основным конкурентам.
- **Демонстрационное тестирование** (qualification testing[188]) – формальный процесс демонстрации заказчику продукта с целью подтверждения, что продукт соответствует всем заявленным требованиям. В отличие от приёмочного тестирования этот процесс более строгий и всеобъемлющий, но может проводиться и на промежуточных стадиях разработки продукта.

[180] What is Mobile Testing? URL: http://smartbear.com/all-resources/articles/what-is-mobile-testing/.

[181] Beginner's Guide to Mobile Application Testing URL: http://www.softwaretestinghelp.com/beginners-guide-to-mobile-application-testing/.

[182] **Data quality.** An attribute of data that indicates correctness with respect to some pre-defined criteria, e.g., business expectations, requirements on data integrity, data consistency. ISTQB Glossary.

[183] **Database integrity testing.** Testing the methods and processes used to access and manage the data(base), to ensure access methods, processes and data rules function as expected and that during access to the database, data is not corrupted or unexpectedly deleted, updated or created. ISTQB Glossary.

[184] **Resource utilization testing, Storage testing.** The process of testing to determine the resource-utilization of a software product. ISTQB Glossary.

[185] **Efficiency testing.** The process of testing to determine the efficiency of a software product (the capability of a process to produce the intended outcome, relative to the amount of resources used). ISTQB Glossary.

[186] **Storage testing.** This is a determination of whether or not certain processing conditions use more storage (memory) than estimated. Pusuluri N. R. Software Testing Concepts And Tools.

[187] **Comparison testing.** Testing that compares software weaknesses and strengths to those of competitors' products. Malhotra J. J., Tiple B. S. Software Testing and Quality Assurance.

[188] **Qualification testing.** Formal testing, usually conducted by the developer for the consumer, to demonstrate that the software meets its specified requirements. Pusuluri N. R. Software Testing Concepts And Tools.

- **Избыточное тестирование** (exhaustive testing[189]) – тестирование приложения со всеми возможными комбинациями всех возможных входных данных во всех возможных условиях выполнения. Для сколь бы то ни было сложной системы нереализуемо, но может применяться для проверки отдельных крайне простых компонентов.
- **Тестирование надёжности** (reliability testing[190]) – тестирование способности приложения выполнять свои функции в заданных условиях на протяжении заданного времени или заданного количества операций.
- **Тестирование восстанавливаемости** (recoverability testing[191]) – тестирование способности приложения восстанавливать свои функции и заданный уровень производительности, а также восстанавливать данные в случае возникновения критической ситуации, приводящей к временной (частичной) утрате работоспособности приложения.
- **Тестирование отказоустойчивости** (failover testing[192]) – тестирование, заключающееся в эмуляции или реальном создании критических ситуаций с целью проверки способности приложения задействовать соответствующие механизмы, предотвращающие нарушение работоспособности, производительности и повреждения данных.
- **Тестирование производительности** (performance testing[193]) – 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[194], capacity testing[195]) – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»);
    - **тестирование масштабируемости** (scalability testing[196]) – исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов;

[189] **Exhaustive testing.** A test approach in which the test suite comprises all combinations of input values and preconditions. ISTQB Glossary.
[190] **Reliability Testing.** The process of testing to determine the reliability of a software product (the ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations). ISTQB Glossary.
[191] **Recoverability Testing.** The process of testing to determine the recoverability of a software product (the capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure). ISTQB Glossary.
[192] **Failover Testing.** Testing by simulating failure modes or actually causing failures in a controlled environment. Following a failure, the failover mechanism is tested to ensure that data is not lost or corrupted and that any agreed service levels are maintained (e.g. function availability or response times). ISTQB Glossary.
[193] **Performance Testing.** The process of testing to determine the performance of a software product. ISTQB Glossary.
[194] **Load Testing.** A type of performance testing conducted to evaluate the behavior of a component or system with increasing load, e.g. numbers of parallel users and/or numbers of transactions, to determine what load can be handled by the component or system. ISTQB Glossary.
[195] **Capacity Testing.** Testing to determine how many users and/or transactions a given system will support and still meet performance goals. URL: https://msdn.microsoft.com/en-us/library/bb924357.aspx.
[196] **Scalability Testing.** Testing to determine the scalability of the software product (the capability of the software product to be upgraded to accommodate increased loads). ISTQB Glossary.

- **объёмное тестирование** (volume testing[197]) – исследование производительности приложения при обработке различных (как правило, больших) объёмов данных;
- **стрессовое тестирование** (stress testing[198]) – исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчётный 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[199]) и представляет собой крайнюю форму негативного тестирования;
- **конкурентное тестирование** (concurrency testing[200]) – исследование поведения приложения в ситуации, когда ему приходится обрабатывать большое количество одновременно поступающих запросов, что вызывает конкуренцию между запросами за ресурсы (базу данных, память, канал передачи данных, дисковую подсистему и т. д.) Иногда под конкурентным тестированием понимают также исследование работы многопоточных приложений и корректность синхронизации действий, производимых в разных потоках.

В качестве отдельных или вспомогательных техник в рамках тестирования производительности могут использоваться тестирование использования ресурсов, тестирование надёжности, тестирование восстанавливаемости, тестирование отказоустойчивости и т. д.

Подробное рассмотрение нескольких видов тестирования производительности приведено в статье «Автоматизация тестирования производительности: основные положения и области применения»[201].

Классификация тестирования по техникам и подходам:

- **Позитивное тестирование** (см. с. 53, с. 67).
- **Негативное тестирование** (см. с. 53, с. 68).

Классификация тестирования на основе опыта тестировщика, сценариев, чек-листов:

- **Исследовательское тестирование** (см. с. 70).
- **Свободное (интуитивное)** тестирование (см. с. 70).

---

[197] **Volume Testing.** Testing where the system is subjected to large volumes of data. ISTQB Glossary.

[198] **Stress testing.** A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as access to memory or servers. ISTQB Glossary.

[199] **Destructive software testing** assures proper or predictable software behavior when the software is subject to improper usage or improper input, attempts to crash a software product, tries to crack or break a software product, checks the robustness of a software product. Akingbehin K. Towards Destructive Software Testing.

[200] **Concurrency testing**. Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is handled by the component or system. ISTQB Glossary.

[201] Автоматизация тестирования производительности: основные положения и области применения. URL: http://svyatoslav.biz/technologies/performance_testing/.

Классификация тестирования по степени вмешательства в работу приложения:

- **Инвазивное тестирование** (intrusive testing[202]) – тестирование, выполнение которого может повлиять на функционирование приложения в силу работы инструментов тестирования (например, будут искажены показатели производительности) или в силу вмешательства (level of intrusion[203]) в сам код приложения (например, для анализа работы приложения было добавлено дополнительное протоколирование, включён вывод отладочной информации и т. д.) Некоторые источники рассматривают[204] инвазивное тестирование как форму негативного или даже стрессового тестирования.
- **Неинвазивное тестирование** (nonintrusive testing[205]) – тестирование, выполнение которого незаметно для приложения и не влияет на процесс его обычной работы.

Классификация тестирования по техникам автоматизации:

- **Тестирование под управлением данными** (data-driven testing[206]) – способ разработки автоматизированных тест-кейсов, в котором входные данные и ожидаемые результаты выносятся за пределы тест-кейса и хранятся вне его – в файле, базе данных и т. д.
- **Тестирование под управлением ключевыми словами** (keyword-driven testing[207]) – способ разработки автоматизированных тест-кейсов, в котором за пределы тест-кейса выносится не только набор входных данных и ожидаемых результатов, но и логика поведения тест-кейса, которая описывается ключевыми словами (командами).
- **Тестирование под управлением поведением** (behavior-driven testing[208]) –

---

[202] **Intrusive testing.** Testing that collects timing and processing information during program execution that may change the behavior of the software from its behavior in a real environment. Intrusive testing usually involves additional code embedded in the software being tested or additional processes running concurrently with software being tested on the same processor. URL: http://encyclopedia2.thefreedictionary.com/intrusive+testing/.

[203] **Level of intrusion.** The level to which a test object is modified by adjusting it for testability. ISTQB Glossary.

[204] Intrusive testing can be considered a type of interrupt testing, which is used to test how well a system reacts to intrusions and interrupts to its normal workflow. URL: http://www.techopedia.com/definition/7802/intrusive-testing.

[205] **Nonintrusive Testing.** Testing that is transparent to the software under test, i.e., does not change its timing or processing characteristics. Nonintrusive testing usually involves additional hardware that collects timing or processing information and processes that information on another platform. URL: http://encyclopedia2.thefreedictionary.com/nonintrusive+testing.

[206] **Data-driven Testing (DDT).** A scripting technique that stores test input and expected results in a table or spreadsheet, so that a single control script can execute all of the tests in the table. Data-driven testing is often used to support the application of test execution tools such as capture/playback tools. ISTQB Glossary.

[207] **Keyword-driven Testing (KDT).** A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script or the test. ISTQB Glossary.

[208] **Behavior-driven Testing (BDT).** Behavior-driven Tests focuses on the behavior rather than the technical implementation of the software. If you want to emphasize on business point of view and requirements then BDT is the way to go. BDT are Given-when-then style tests written in natural language which are easily understandable to non-technical individuals. Hence these tests allow business analysts and management people to actively participate in test creation and review process. Rangaiah J. URL: http://www.womentesters.com/behaviour-driven-testing-an-introduction/.

способ разработки автоматизированных тест-кейсов, в котором основное внимание уделяется корректности работы бизнес-сценариев, а не отдельным деталям функционирования приложения.

Классификация тестирования на основе (знания) источников ошибок:

- **Тестирование предугадыванием ошибок** (error guessing[209]) – техника тестирования, в которой тесты разрабатываются на основе опыта тестировщика и его знаний о том, какие дефекты типичны для тех или иных компонентов или областей функциональности приложения. Может комбинироваться с техникой так называемого «ошибкоориентированного» тестирования (failure-directed testing[210]), в котором новые тесты строятся на основе информации о ранее обнаруженных в приложении проблемах.
- **Эвристическая оценка** (heuristic evaluation[211]) – техника тестирования удобства использования, направленная на поиск проблем в интерфейсе пользователя, представляющих собой отклонение от общепринятых норм.
- **Мутационное тестирование** (mutation testing[212]) – техника тестирования, в которой сравнивается поведение нескольких версий одного и того же компонента, причём часть таких версий может быть специально разработана с добавлением ошибок (что позволяет оценить эффективность тест-кейсов – качественные тесты обнаружат эти специально добавленные ошибки). Может комбинироваться со следующим в этом списке видом тестирования (тестированием добавлением ошибок).
- **Тестирование добавлением ошибок** (error seeding[213]) – техника тестирования, в которой в приложение специально добавляются заранее известные, специально продуманные ошибки с целью мониторинга их обнаружения и устранения и, таким образом, формирования более точной оценки показателей процесса тестирования. Может комбинироваться с предыдущим в этом списке видом тестирования (мутационным тестированием).

Классификация тестирования на основе выбора входных данных:

- **Тестирование на основе классов эквивалентности** (equivalence partitioning[214]) – техника тестирования, направленная на сокращение количества разрабатываемых и выполняемых тест-кейсов при сохранении достаточного тестового покрытия. Суть техники состоит в выявлении наборов эквивалентных тест-кейсов (каждый из которых проверяет одно и то же поведение приложения) и выборе

[209] **Error Guessing.** A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them. ISTQB Glossary.

[210] **Failure-directed Testing.** Software testing based on the knowledge of the types of errors made in the past that are likely for the system under test. URL: http://dictionary.reference.com/browse/failure-directed+testing.

[211] **Heuristic Evaluation.** A usability review technique that targets usability problems in the user interface or user interface design. With this technique, the reviewers examine the interface and judge its compliance with recognized usability principles (the «heuristics»). ISTQB Glossary.

[212] **Mutation Testing, Back-to-Back Testing.** Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies. ISTQB Glossary.

[213] **Error seeding.** The process of intentionally adding known faults to those already in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of faults remaining in the program. ISTQB Glossary.

[214] **Equivalence partitioning.** A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once. ISTQB Glossary.

из таких наборов небольшого подмножества тест-кейсов, с наибольшей вероятностью обнаруживающих проблему.

- **Тестирование на основе граничных условий** (boundary value analysis[215]) – инструментальная техника тестирования на основе классов эквивалентности, позволяющая выявить специфические значения исследуемых параметров, относящиеся к границам классов эквивалентности. Эта техника значительно упрощает выявление наборов эквивалентных тест-кейсов и выбор таких тест-кейсов, которые обнаружат проблему с наибольшей вероятностью.
- **Доменное тестирование** (domain analysis[216], domain testing) – техника тестирования на основе классов эквивалентности и граничных условий, позволяющая эффективно создавать тест-кейсы, затрагивающие несколько параметров (переменных) одновременно (в том числе с учётом взаимозависимости этих параметров). Данная техника также описывает подходы к выбору минимального множества показательных тест-кейсов из всего набора возможных тест-кейсов.
- **Попарное тестирование** (pairwise testing[217]) – техника тестирования, в которой тест-кейсы строятся по принципу проверки пар значений параметров (переменных) вместо того, чтобы пытаться проверить все возможные комбинации всех значений всех параметров. Эта техника является частным случаем N-комбинационного тестирования (n-wise testing[218]) и позволяет существенно сократить трудозатраты на тестирование (а иногда и вовсе сделать возможным тестирование в случае, когда количество «всех комбинаций всех значений всех параметров» измеряется миллиардами).

Попарное тестирование (pairwise testing[217]) – это НЕ парное тестирование (pair testing[219])!

- **Тестирование на основе ортогональных массивов** (orthogonal array testing[220]) – инструментальная техника попарного и N-комбинационного тестирования, основанная на использовании так называемых «ортогональных массивов» (двумерных массивов, обладающих следующим свойством: если взять две любые колонки такого массива, то получившийся «подмассив» будет содержать все возможные попарные комбинации значений, представленных в исходном массиве).

[215] **Boundary value analysis.** A black box test design technique in which test cases are designed based on boundary values (input values or output values which are on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum or maximum value of a range). ISTQB Glossary.

[216] **Domain analysis.** A black box test design technique that is used to identify efficient and effective test cases when multiple variables can or should be tested together. It builds on and generalizes equivalence partitioning and boundary values analysis. ISTQB Glossary.

[217] **Pairwise testing.** A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters. ISTQB Glossary.

[218] **N-wise testing.** A black box test design technique in which test cases are designed to execute all possible discrete combinations of any set of n input parameters. ISTQB Glossary.

[219] **Pair testing.** Two persons, e.g. two testers, a developer and a tester, or an end-user and a tester, working together to find defects. Typically, they share one computer and trade control of it while testing. ISTQB Glossary.

[220] **Orthogonal array testing.** A systematic way of testing all-pair combinations of variables using orthogonal arrays. It significantly reduces the number of all combinations of variables to test all pair combinations. See also combinatorial testing, n-wise testing, pairwise testing. ISTQB Glossary.

Ортогональные массивы – это НЕ ортогональные матрицы. Это совершенно разные понятия! Сравните их описания исходя из статей «Orthogonal array»[221] и «Orthogonal matrix»[222].

Также рассмотрим комбинаторные техники тестирования, которые расширяют и дополняют только что рассмотренный список видов тестирования на основе выбора входных данных.

Очень подробное описание некоторых видов тестирования, относящихся к данной классификации, можно найти в книге Ли Копленда «Практическое руководство по разработке тестов» (Copeland L. «A Practitioner's Guide to Software Test Design»), в частности:

- тестирование на основе классов эквивалентности – в главе 3;
- тестирование на основе граничных условий – в главе 4;
- доменное тестирование – в главе 8;
- попарное тестирование и тестирование на основе ортогональных массивов – в главе 6.

Важно! Большинство рассмотренных выше техник тестирования входит в «джентльменский набор любого тестировщика», поэтому их понимание и умение применять можно считать обязательным.

Классификация тестирования на основе среды выполнения:

- **Тестирование в процессе разработки** (development testing[223]) – тестирование, выполняемое непосредственно в процессе разработки приложения и/или в среде выполнения, отличной от среды реального использования приложения. Как правило, выполняется самими разработчиками.
- **Операционное тестирование** (см. с. 73).

Классификация тестирования на основе кода (code based testing). Следующую технику в различных источниках называют по-разному (чаще всего – тестированием на основе структур, причём некоторые авторы смешивают в один набор тестирования по потоку управления и по потоку данных, а некоторые строго разделяют эти стратегии). Подвиды этой техники также организуют в разные комбинации, но наиболее универсально их можно классифицировать так:

- **Тестирование по потоку управления** (control flow testing[224]) – семейство техник тестирования, в которых тест-кейсы разрабатываются с целью активации и проверки выполнения различных последовательностей событий, которые определяются посредством анализа исходного кода приложения. Дополнительное подробное пояснение приведено далее в этом разделе (см. тестирование на основе структур кода).

[221] Orthogonal array. Wikipedia. URL: http://en.wikipedia.org/wiki/Orthogonal_array.
[222] Orthogonal matrix. Wikipedia. URL: http://en.wikipedia.org/wiki/Orthogonal_matrix.
[223] **Development testing.** Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers. ISTQB Glossary.
[224] **Control Flow Testing.** An approach to structure-based testing in which test cases are designed to execute specific sequences of events. Various techniques exist for control flow testing, e.g. decision testing, condition testing, and path testing, that each have their specific approach and level of control flow coverage. ISTQB Glossary.

- **Тестирование по потоку данных** (data-flow testing[225]) – семейство техник тестирования, основанных на выборе отдельных путей из потока управления с целью исследования событий, связанных с изменением состояния переменных. Дополнительное подробное пояснение приведено далее в этом разделе (в части, где тестирование по потоку данных пояснено с точки зрения стандарта ISO/IEC/IEEE 29119-4).
- **Тестирование по диаграмме или таблице состояний** (state transition testing[226]) – техника тестирования, в которой тест-кейсы разрабатываются для проверки переходов приложения из одного состояния в другое. Состояния могут быть описаны диаграммой состояний (state diagram[227]) или таблицей состояний (state table[228]).

Хорошее подробное пояснение по данному виду тестирования можно найти в статье «What is State transition testing in software testing?»[229].

Иногда эту технику тестирования также называют «тестированием по принципу конечного автомата» (finite state machine[230] testing). Важным преимуществом этой техники является возможность применения в ней теории конечных автоматов (которая хорошо формализована), а также возможность использования автоматизации для генерации комбинаций входных данных.

- **Инспекция (аудит) кода** (code review, code inspection[231]) – семейство техник повышения качества кода за счёт того, что в процессе создания или совершенствования кода участвуют несколько человек. Степень формализации аудита кода может варьироваться от достаточно беглого просмотра до тщательной формальной инспекции. В отличие от техник статического анализа кода (по потоку управления и потоку данных) аудит кода также улучшает такие его характеристики, как понятность, поддерживаемость, соответствие соглашениям об оформлении и т. д. Аудит кода выполняется в основном самими программистами.

Классификация тестирования на основе структур кода (structure-based techniques) предполагает возможность исследования логики выполнения кода в зависимости от различных ситуаций и включает в себя:

[225] **Data Flow Testing.** A white box test design technique in which test cases are designed to execute definition-use pairs of variables. ISTQB Glossary.
[226] **State Transition Testing.** A black box test design technique in which test cases are designed to execute valid and invalid state transitions. ISTQB Glossary.
[227] **State Diagram.** A diagram that depicts the states that a component or system can assume, and shows the events or circumstances that cause and/or result from a change from one state to another. ISTQB Glossary.
[228] **State Table.** A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions. ISTQB Glossary.
[229] What is State transition testing in software testing? URL: http://istqbexamcertification.com/what-is-state-transition-testing-in-software-testing.
[230] **Finite State Machine.** A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions. ISTQB Glossary.
[231] **Inspection.** A type of peer review that relies on visual examination of documents to detect defects, e.g. violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. ISTQB Glossary.

- **Тестирование на основе выражений** (statement testing[232]) – это техника тестирования (по методу белого ящика), в которой проверяется корректность (и сам факт) выполнения отдельных выражений в коде.
- **Тестирование на основе ветвей** (branch testing[233]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение отдельных ветвей кода (под ветвью понимается атомарная часть кода, выполнение которой происходит или не происходит в зависимости от истинности или ложности некоторого условия).
- **Тестирование на основе условий** (condition testing[234]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение отдельных условий (условием считается выражение, которое может быть вычислено до значения «истина» или «ложь»).
- **Тестирование на основе комбинаций условий** (multiple condition testing[235]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение сложных (составных) условий.
- **Тестирование на основе отдельных условий, порождающих ветвление («решающих условий»)** (modified condition decision coverage testing[236]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение таких отдельных условий в составе сложных условий, которые в одиночку определяют результат вычисления всего сложного условия.
- **Тестирование на основе решений** (decision testing[237]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение сложных ветвлений (с двумя и более возможными вариантами). Несмотря на то что «два варианта» сюда также подходят, формально такую ситуацию стоит отнести к тестированию на основе условий.
- **Тестирование на основе путей** (path testing[238]) – это техника тестирования (по методу белого ящика), в которой проверяется выполнение всех или некоторых специально выбранных путей в коде приложения.

[232] **Statement Testing.** A white box test design technique in which test cases are designed to execute statements (statement is an entity in a programming language, which is typically the smallest indivisible unit of execution). ISTQB Glossary.

[233] **Branch Testing.** A white box test design technique in which test cases are designed to execute branches (branch is a basic block that can be selected for execution based on a program construct in which one of two or more alternative program paths is available, e.g. case, jump, go to, if-then-else.). ISTQB Glossary.

[234] **Condition Testing.** A white box test design technique in which test cases are designed to execute condition outcomes (condition is a logical expression that can be evaluated as True or False, e.g. A > B). ISTQB Glossary.

[235] **Multiple Condition Testing.** A white box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement). ISTQB Glossary.

[236] **Modified Condition Decision Coverage Testing.** Technique to design test cases to execute branch condition outcomes that independently affect a decision outcome and discard conditions that do not affect the final outcome. Hass A. M. Guide to Advanced Software Testing. – 2rd ed.

[237] **Decision Testing.** A white box test design technique in which test cases are designed to execute decision outcomes (decision is program point at which the control flow has two or more alternative routes, e.g. a node with two or more links to separate branches). ISTQB Glossary.

[238] **Path testing.** A white box test design technique in which test cases are designed to execute paths. ISTQB Glossary.

С точки зрения науки определения большинства видов тестирования на основе структур кода должны звучать несколько иначе, т. к. в программировании условием считается выражение без логических операторов, а решением – выражение с логическими операторами. Но глоссарий ISTQB не ставит на этом акцент, а поэтому приведённые выше определения можно считать корректными. Однако, если вам интересно, рекомендуется ознакомиться с заметкой «What is the difference between a Decision and a Condition?»[239].

Сравнительная характеристика видов тестирования на основе структур кода кратко показана в таблице 3.c.

Таблица 3.c – Виды тестирования на основе структур кода

Русскоязычное название	**Англоязычное название**	**Суть (что проверяется)**
Тестирование на основе выражений	Statement testing	Отдельные атомарные участки кода, например, x = 10
Тестирование на основе ветвей	Branch testing	Проход по ветвям выполнения кода
Тестирование на основе условий	Condition testing, Branch Condition Testing	Отдельные условные конструкции, например, if (a == b)
Тестирование на основе комбинаций условий	Multiple condition testing, Branch Condition Combination Testing	Составные условные конструкции, например, if ((a == b) || (c == d))
Тестирование на основе отдельных условий, порождающих ветвление («решающих условий»)	Modified Condition Decision Coverage Testing	Отдельные условия, в одиночку влияющие на итог вычисления составного условия, например, в условии if ((x == y) && (n == m)) ложное значение в каждом из отдельных условий само по себе приводит к результату false вне зависимости от результата вычисления второго условия
Тестирование на основе решений	Decision testing	Сложные ветвления, например, оператор switch
Тестирование на основе путей	Path testing	Все или специально выбранные пути

[239] What is the difference between a Decision and a Condition? URL: http://www-01.ibm.com/support/docview.wss?uid=swg21129252.

Классификация тестирования на основе (моделей) поведения приложения (application behavior/model-based testing):

- **Тестирование по таблице принятия решений** (decision table testing[240]) – техника тестирования (по методу чёрного ящика), в которой тест-кейсы разрабатываются на основе так называемой таблицы принятия решений, в которой отражены входные данные (и их комбинации) и воздействия на приложение, а также соответствующие им выходные данные и реакции приложения.
- **Тестирование по диаграмме или таблице состояний** (см. с. 83).
- **Тестирование по спецификациям** (specification-based testing, black box testing) (см. с. 58).
- **Тестирование по моделям поведения приложения** (model-based testing[241]) – это техника тестирования, в которой исследование приложения (и разработка тест-кейсов) строится на некой модели: таблице принятия решений, таблице или диаграмме состояний, пользовательских сценариев, модели нагрузки и т. д.
- **Тестирование на основе вариантов использования** (use case testing[242]) – это техника тестирования (по методу чёрного ящика), в которой тест-кейсы разрабатываются на основе вариантов использования. Варианты использования выступают в основном источником информации для шагов тест-кейса, в то время как наборы входных данных удобно разрабатывать с помощью техник выбора входных данных. В общем случае источником информации для разработки тест-кейсов в этой технике могут выступать не только варианты использования, но и другие пользовательские требования в любом их виде. В случае если методология разработки проекта подразумевает использование пользовательских историй, этот вид тестирования может быть заменён тестированием на основе пользовательских историй (user story testing[243]).
- **Параллельное тестирование** (parallel testing[244]) – техника тестирования, в которой поведение нового (или модифицированного) приложения сравнивается с поведением эталонного приложения (предположительно работающего верно). Термин «параллельное тестирование» также может использоваться для обозначения способа проведения тестирования, когда несколько тестировщиков или систем автоматизации выполняют работу одновременно, т. е. параллельно. Очень редко (и не совсем верно) под параллельным тестированием понимают мутационное тестирование.
- **Тестирование на основе случайных данных** (random testing[245]) – техника

[240] **Decision Table Testing.** A black box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table (a table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases). ISTQB Glossary.

[241] **Model-based Testing.** Testing based on a model of the component or system under test, e.g. reliability growth models, usage models such as operational profiles or behavioral models such as decision table or state transition diagram. ISTQB Glossary.

[242] **Use case testing.** A black box test design technique in which test cases are designed to execute scenarios of use cases. ISTQB Glossary.

[243] **User story testing.** A black box test design technique in which test cases are designed based on user stories to verify their correct implementation. ISTQB Glossary.

[244] **Parallel testing.** Testing a new or an altered data processing system with the same source data that is used in another system. The other system is considered as the standard of comparison. ISTQB Glossary.

[245] **Random testing.** A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile. This technique can be used for testing non-functional attributes such as reliability and performance. ISTQB Glossary.

тестирования (по методу чёрного ящика), в которой входные данные, действия или даже сами тест-кейсы выбираются на основе случайных или псевдослучайных значений так, чтобы соответствовать операционному профилю (operational profile[246]) – подмножеству действий, соответствующих некоей ситуации или сценарию работы с приложением. Не стоит путать этот вид тестирования с так называемым «обезьяньим тестированием» (monkey testing[247]).

- **A/B-тестирование** (A/B testing, split testing[248]) – техника тестирования, в которой исследуется влияние на результат выполнения операции изменения одного из входных параметров. Однако куда чаще можно встретить трактовку A/B-тестирования как технику тестирования удобства использования, в которой пользователям случайным образом предлагаются разные варианты элементов интерфейса, после чего оценивается разница в реакции пользователей.

Очень подробное описание некоторых видов тестирования, относящихся к данной классификации, можно найти в книге Ли Коупленда «Практическое руководство по разработке тестов» (Copeland L. «A Practitioner's Guide to Software Test Design»):

- тестирование по таблице принятия решений – в главе 5;
- тестирование по диаграмме или таблице состояний – в главе 7;
- тестирование на основе вариантов использования – в главе 9.

Несмотря на многочисленные попытки создать единую хронологию тестирования, предпринятые многими авторами, по-прежнему можно утверждать, что общепринятого решения, которое в равной степени подходило бы для любой методологии управления проектами, любого отдельного проекта и любой его стадии, просто не существует.

Если попытаться описать хронологию тестирования общей фразой, то можно сказать, что происходит постепенное наращивание сложности самих тест-кейсов и сложности логики их выбора.

Классификация тестирования по моменту выполнения (хронологии):

- **Общая универсальная логика последовательности тестирования** состоит в том, чтобы начинать исследование каждой задачи с простых позитивных тест-кейсов, к которым постепенно добавлять негативные (но тоже достаточно простые). Лишь после того, как наиболее типичные ситуации покрыты простыми тест-кейсами, следует переходить к более сложным (опять же, начиная с позитивных). Такой подход – не догма, но к нему стоит прислушаться, т. к. углубление на начальных этапах в негативные (к тому же сложные) тест-кейсы может привести к ситуации, в которой приложение отлично справляется с целым рядом исключительных ситуаций, но не работает при решении элементарных повседневных задач. Итак, суть универсальной последовательности:

[246] **Operational profile.** The representation of a distinct set of tasks performed by the component or system, possibly based on user behavior when interacting with the component or system, and their probabilities of occurrence. A task is logical rather that physical and can be executed over several machines or be executed in non-contiguous time segments. ISTQB Glossary.

[247] **Monkey testing.** Testing by means of a random selection from a large range of inputs and by randomly pushing buttons, ignorant of how the product is being used. ISTQB Glossary.

[248] **Split testing** is a design for establishing a causal relationship between changes and their influence on user-observable behavior. Kohav R. Controlled experiments on the web: survey and practical guide.

- простое позитивное тестирование;
- простое негативное тестирование;
- сложное позитивное тестирование;
- сложное негативное тестирование.

- **Последовательность тестирования, построенная по иерархии компонентов:**
  - **восходящее тестирование** (bottom-up testing[249]) – инкрементальный подход к интеграционному тестированию, в котором в первую очередь тестируются низкоуровневые компоненты, после чего процесс переходит на всё более и более высокоуровневые компоненты;
  - **нисходящее тестирование** (top-down testing[250]) – инкрементальный подход к интеграционному тестированию, в котором в первую очередь тестируются высокоуровневые компоненты, после чего процесс переходит на всё более и более низкоуровневые компоненты;
  - **гибридное тестирование** (hybrid testing[251]) – комбинация восходящего и нисходящего тестирований, позволяющая упростить и ускорить получение результатов оценки приложения.

Поскольку термин «гибридное» является синонимом «комбинированное», под «гибридным тестированием» может пониматься практически любое сочетание двух и более видов техник или подходов к тестированию. Всегда уточняйте, о гибриде чего именно идёт речь.

- **Последовательность тестирования, построенная по концентрации внимания на требованиях и их составляющих:**
  - тестирование требований, которое может варьироваться от беглой оценки в стиле «всё ли нам понятно» до весьма формальных подходов, в любом случае первично по отношению к тестированию того, как эти требования реализованы;
  - тестирование реализации функциональных составляющих требований логично проводить до тестирования реализации нефункциональных составляющих, т. к. если что-то просто не работает, то проверять производительность, безопасность, удобство и прочие нефункциональные составляющие бессмысленно, а чаще всего и вовсе невозможно;
  - тестирование реализации нефункциональных составляющих требований часто становится логическим завершением проверки того, как реализовано то или иное требование.
- **Типичные общие сценарии** используются в том случае, когда не существует явных предпосылок к реализации иной стратегии. Такие сценарии могут видоизменяться и комбинироваться (например, весь «типичный общий сценарий 1»

[249] **Bottom-up testing.** An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. ISTQB Glossary.

[250] **Top-down testing.** An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested. ISTQB Glossary.

[251] **Hybrid testing, Sandwich testing.** First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. Parmar K. Integration testing techniques.

можно повторять на всех шагах «типичного общего сценария 2»):

- типичный общий сценарий 1:
    1) дымовое тестирование;
    2) тестирование критического пути;
    3) расширенное тестирование;
- типичный общий сценарий 2:
    1) модульное тестирование;
    2) интеграционное тестирование;
    3) системное тестирование;
- типичный общий сценарий 3:
    1) альфа-тестирование;
    2) бета-тестирование;
    3) гамма-тестирование.

Ещё раз подчеркнём, что рассмотренные здесь классификации тестирования не являются каноном и чем-то незыблемым. Они приведены с целью упорядочить большой объём информации о различных видах деятельности тестировщиков и упростить запоминание соответствующих фактов.

3.3 Альтернативные и дополнительные классификации тестирования

Для полноты картины остаётся лишь показать альтернативные взгляды на классификацию тестирования. Одна из них (рисунки 3.i и 3.j) представляет не более чем иную комбинацию ранее рассмотренных видов и техник. Вторая (рисунки 3.k и 3.l) содержит много новых определений, но их подробное изучение выходит за рамки данного пособия, и потому будут даны лишь краткие пояснения (при необходимости можно ознакомиться с первоисточниками, которые указаны для каждого определения в сносках).

Ещё раз подчеркнём, что в данном подразделе приведены лишь определения. Соответствующим видам и техникам тестирования посвящены отдельные книги.

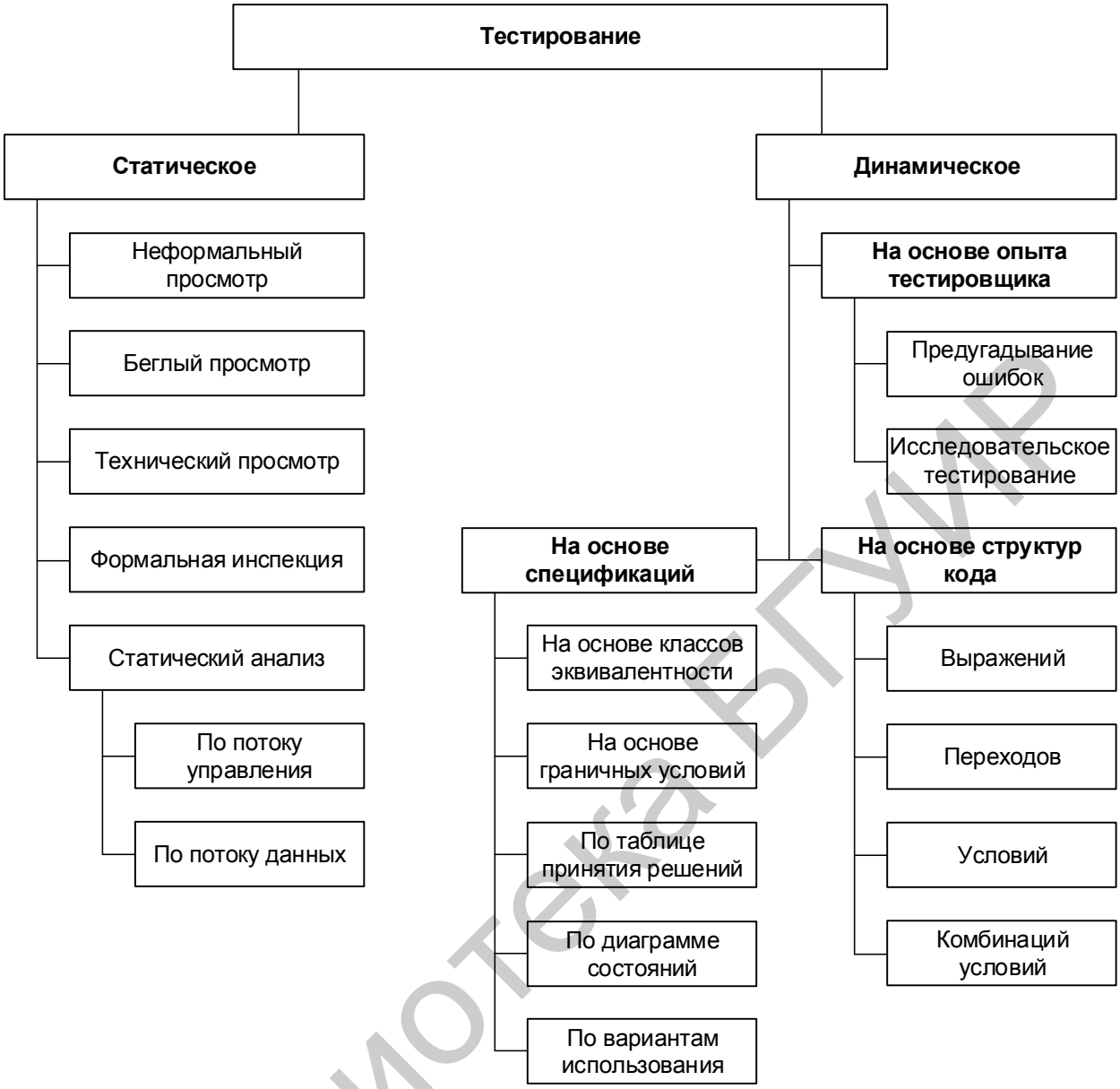


Рисунок 3.i – Классификация тестирования согласно книге Veenendaal E., Evans I. «Foundations of Software Testing: ISTQB Certification» (русскоязычный вариант)

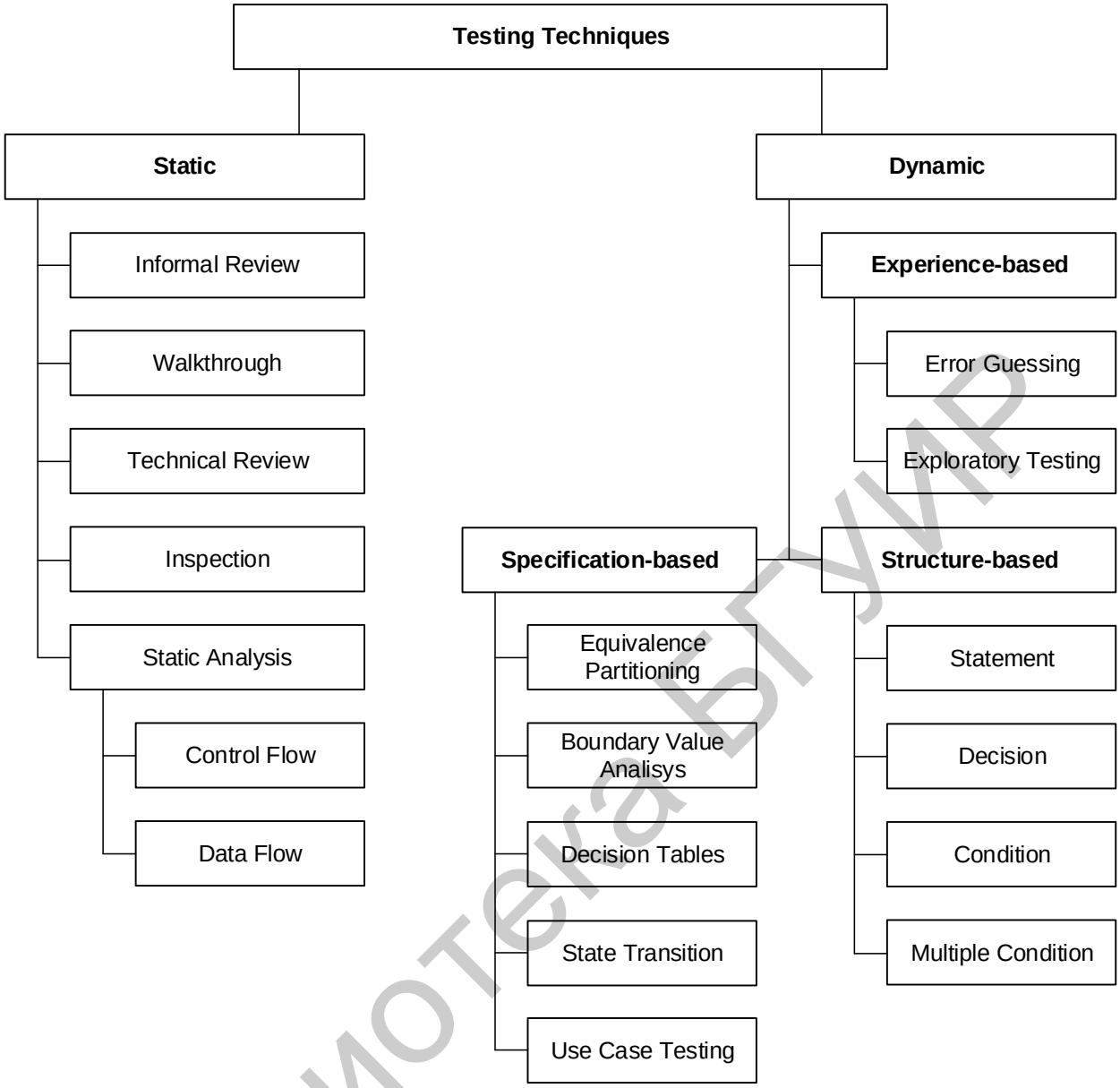


Рисунок 3.j – Классификация тестирования согласно книге Veenendaal E., Evans I. «Foundations of Software Testing: ISTQB Certification» (англоязычный вариант)

В следующей классификации встречаются как уже рассмотренные пункты, так и ранее не рассмотренные (отмечены пунктирной линией). Краткие определения нерассмотренных ранее видов тестирования даны после рисунков 3.k и 3.l.

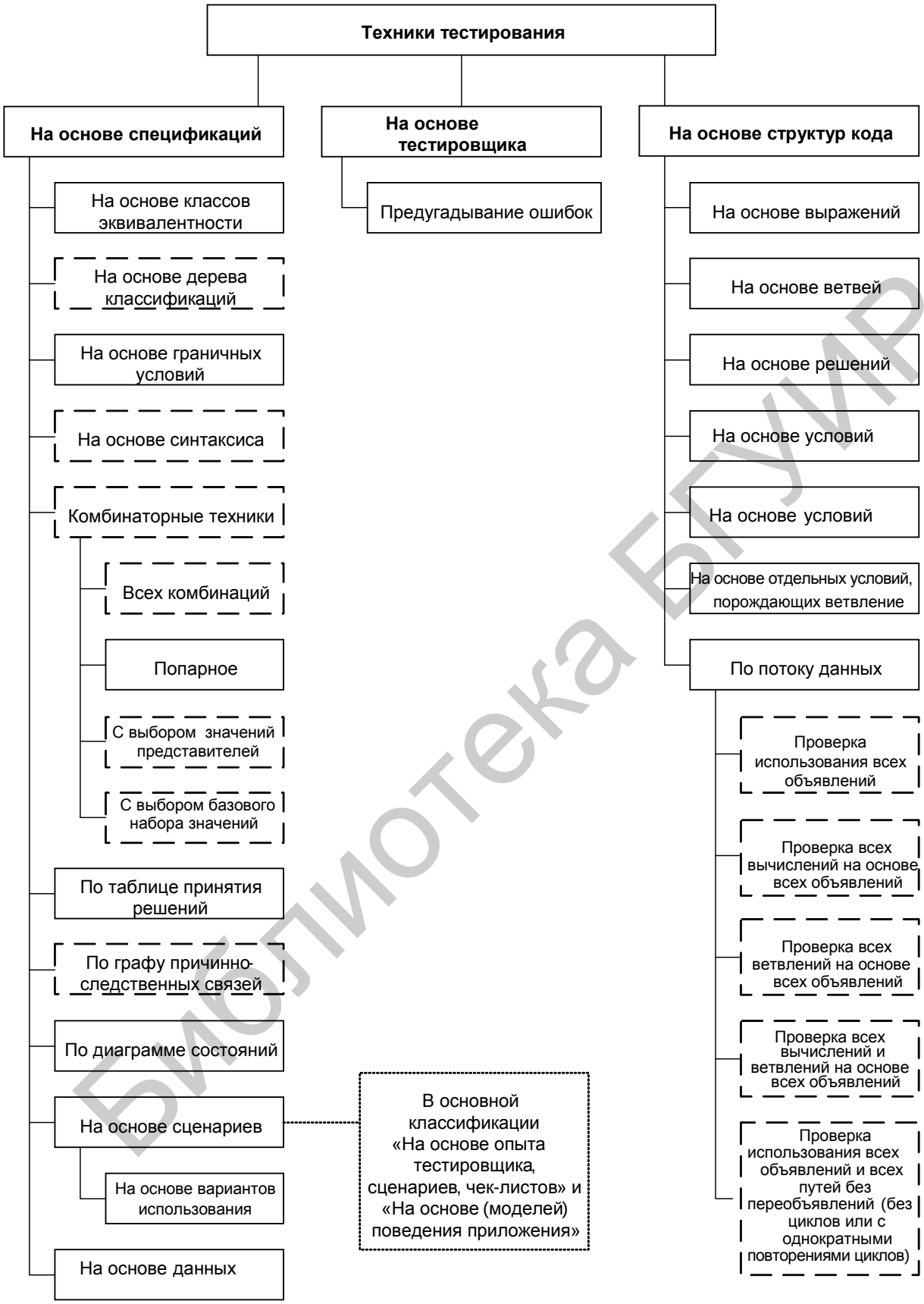


Рисунок 3.k – Классификация тестирования согласно ISO/IEC/IEEE 29119-4 (русскоязычный вариант)

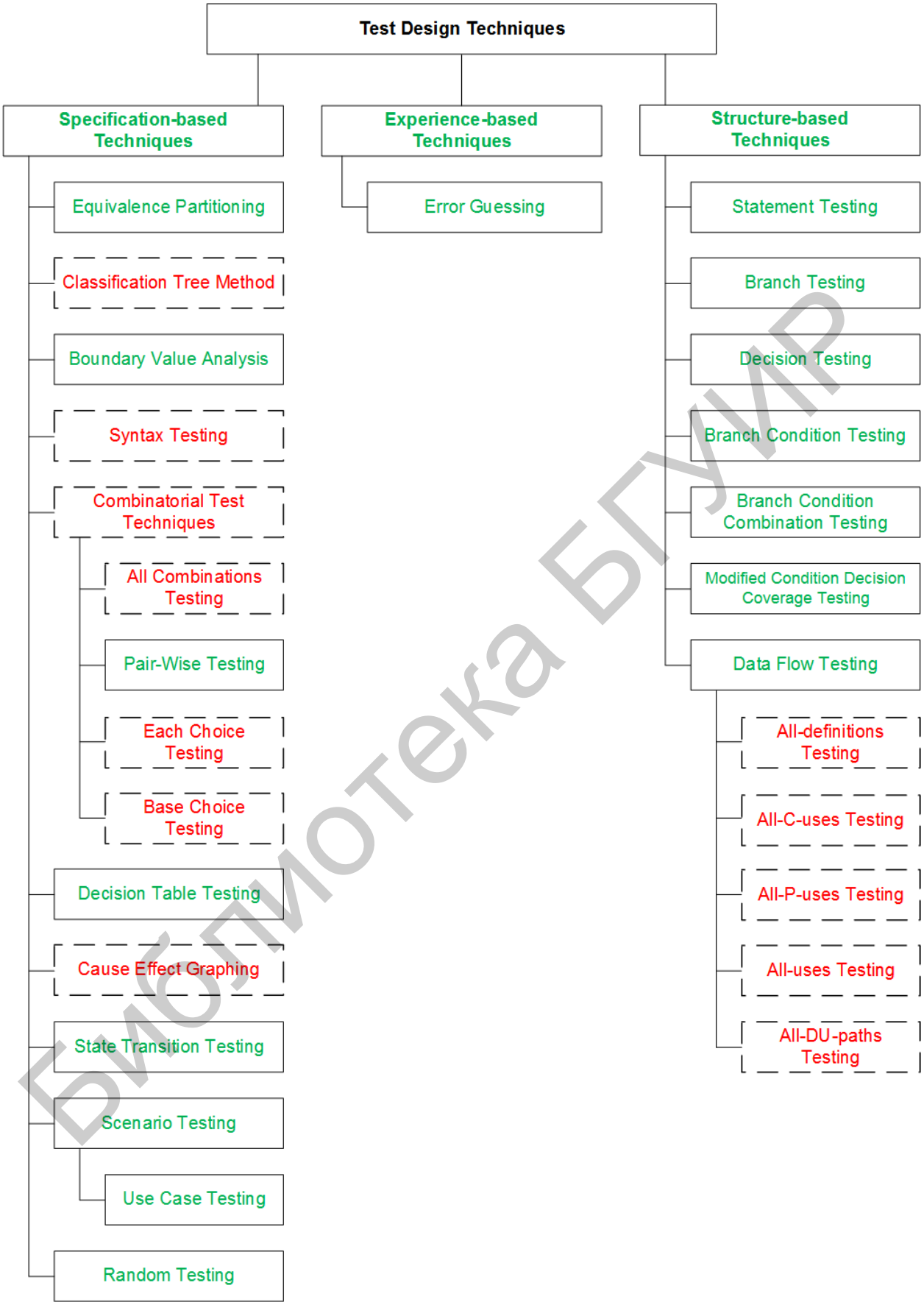


Рисунок 3.I – Классификация тестирования согласно ISO/IEC/IEEE 29119-4 (англоязычный вариант)

- **Тестирование на основе дерева классификаций** (classification tree[252] method[253]) – техника тестирования (по методу чёрного ящика), в которой тест-кейсы создаются на основе иерархически организованных наборов эквивалентных входных и выходных данных.
- **Тестирование на основе синтаксиса** (syntax testing[254]) – техника тестирования (по методу чёрного ящика), в которой тест-кейсы создаются на основе определения наборов входных и выходных данных.
- **Комбинаторные техники или комбинаторное тестирование** (combinatorial testing[255]) – способ выбрать подходящий набор комбинаций тестовых данных для достижения установленного уровня тестового покрытия в случае, когда проверка всех возможных наборов значений тестовых данных невозможна за имеющееся время. Существуют следующие комбинаторные техники:
    - **тестирование всех комбинаций** (all combinations testing[256]) – тестирование всех возможных комбинаций всех значений всех тестовых данных (например, всех параметров функции);
    - **попарное тестирование** (см. с. 81);
    - **тестирование с выбором значений-представителей** (each choice testing[257]) – тестирование, при котором по одному значению из каждого набора тестовых данных должно быть использовано хотя бы в одном тест-кейсе;
    - **тестирование с выбором базового набора значений** (base choice testing[258]) – тестирование, при котором выделяется набор значений (базовый набор), который используется для проведения тестирования в первую очередь, а далее тест-кейсы строятся на основе выбора всех базовых значений, кроме одного, которое заменяется значением, не входящим в базовый набор;

  Также учитываем классификацию тестирования на основе выбора входных данных, которая расширяет и дополняет данный список (см. с. 80).

[252] **Classification tree.** A tree showing equivalence partitions hierarchically ordered, which is used to design test cases in the classification tree method. ISTQB Glossary.
[253] **Classification tree method.** A black box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains. ISTQB Glossary.
[254] **Syntax testing.** A black box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain. ISTQB Glossary.
[255] **Combinatorial testing.** A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple parameters and where those parameters themselves each have several values, which gives rise to more combinations than are feasible to test in the time allowed. ISTQB Glossary.
[256] **All combinations testing.** Testing of all possible combinations of all values for all parameters. Hass A. M. Guide to advanced software testing. – 2nd ed.
[257] **Each choice testing.** One value from each block for each partition must be used in at least one test case. Introduction to Software Testing. Chapter 4. Input Space Partition Testing, P. Ammann,J. Offutt.
[258] **Base choice testing.** A base choice block is chosen for each partition, and a base test is formed by using the base choice for each partition. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other parameter. Ammann P., Offutt J. Introduction to Software Testing. Chapter 4. Input Space Partition Testing.

- **Тестирование по графу причинно-следственных связей** (cause-effect graphing[259]) – техника тестирования (по методу чёрного ящика), в которой тест-кейсы разрабатываются на основе графа причинно-следственных связей (графического представления входных данных и воздействий со связанными с ними выходными данными и эффектами).
- **Тестирование по потоку данных** (data-flow testing[260]) – семейство техник тестирования, основанных на выборе отдельных путей из потока управления с целью исследования событий, связанных с изменением состояния переменных. Эти техники позволяют обнаружить следующие ситуации: переменная определена, но нигде не используется; переменная используется, но не определена; переменная определена несколько раз до того, как она используется; переменная удалена до последнего случая использования.

  Приведём теоретический материал.

  Над переменной в общем случае может выполняться несколько действий (покажем на примере переменной **x**):
  - объявление (declaration): int **x**;
  - определение (definition, d-use): **x** = 99;
  - использование в вычислениях (computation use, c-use): z = **x** + 1;
  - использование в условиях (predicate use, p-use): if (**x** > 17) { … };
  - удаление (kill, k-use): **x** = null.

  Теперь можно рассмотреть техники тестирования на основе потока данных. Они подробно описаны в книге Бориса Бейзера «Техники тестирования ПО» (Beizer B. Software Testing Techniques. – 2nd ed), мы же приведём очень краткие пояснения:
  - **проверка использования всех объявлений** (all-definitions testing[261]) – тестовым набором проверяется, что для каждой переменной существует путь от её определения к её использованию в вычислениях или условиях;
  - **проверка всех вычислений на основе всех объявлений** (all-c-uses testing[262]) – тестовым набором проверяется, что для каждой переменной существует путь от каждого её определения к её использованию в вычислениях;
  - **проверка всех ветвлений на основе всех объявлений** (all-p-uses testing[263]) – тестовым набором проверяется, что для каждой переменной существует путь от каждого её определения к её использованию в условиях;

[259] **Cause-effect graphing.** A black box test design technique in which test cases are designed from cause-effect graphs (a graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases). ISTQB Glossary.
[260] **Data flow testing.** A white box test design technique in which test cases are designed to execute definition-use pairs of variables. ISTQB Glossary.
[261] **All-definitions strategy.** Test set requires that every definition of every variable is covered by at least one use of that variable (c-use or p-use). Beizer B. Software Testing Techniques. – 2nd ed.
[262] **All-computation-uses strategy.** For every variable and every definition of that variable, include at least one definition-free path from the definition to every computation use. Beizer B. Software Testing Techniques. – 2nd ed.
[263] **All-predicate-uses strategy.** For every variable and every definition of that variable, include at least one definition-free path from the definition to every predicate use. Beizer B. Software Testing Techniques. – 2nd ed.

- **проверка всех вычислений и ветвлений на основе всех объявлений** (all-uses testing[264]) – тестовым набором проверяется, что для каждой переменной существует хотя бы один путь от каждого её определения к каждому её использованию в вычислениях и в условиях;
- **проверка использования всех объявлений и всех путей без переобъявлений (без циклов или с однократными повторениями циклов)** (all-du-paths testing[265]) – тестовым набором для каждой переменной проверяются все пути от каждого её определения к каждому её использованию в вычислениях и в условиях (самая мощная стратегия, которая в то же время требует наибольшего количества тест-кейсов).

Для лучшего понимания и запоминания приведём оригинальную схему из книги Бориса Бейзера «Техники тестирования ПО» (под названием Figure 5.7. Relative Strength of Structural Test Strategies), показывающую взаимосвязь стратегий тестирования на основе потока данных (рисунок 3.m).

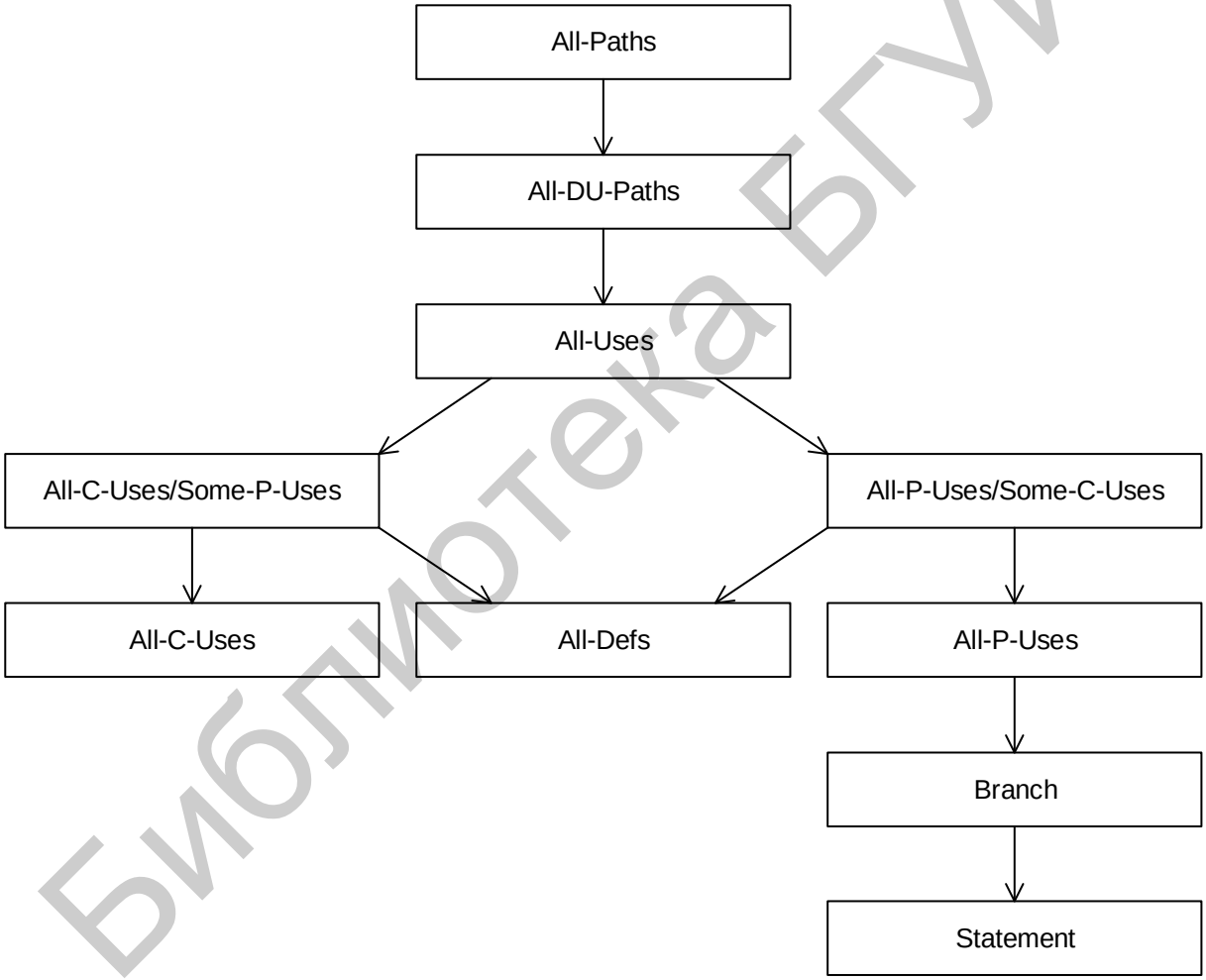


Рисунок 3.m – Взаимосвязь и относительная мощность стратегий тестирования на основе потока данных (по книге Бориса Бейзера «Техники тестирования ПО»)

[264] **All-uses strategy.** Test set includes at least one path segment from every definition to every use that can be reached by that definition. Beizer B. Software Testing Techniques. – 2nd ed.

[265] **All-DU-path strategy.** Test set includes every du path from every definition of every variable to every use of that definition. Beizer B. Software Testing Techniques. – 2nd ed.

3.4 Классификация по принадлежности к тестированию по методу белого и чёрного ящиков

Типичным вопросом на собеседовании для начинающих тестировщиков является просьба перечислить техники тестирования по методу белого и чёрного ящиков. Ниже представлена таблица 3.d, в которой все вышерассмотренные виды тестирования соотнесены с соответствующим методом. Эту таблицу можно использовать также как справочник по видам тестирования (они представлены в той же последовательности, в какой описаны в данном разделе).

Важно! В источниках наподобие ISTQB-глоссария многие виды и техники тестирования жёстко соотнесены с методами белого или чёрного ящиков. Но это не значит, что их невозможно применить в другом, неотмеченном методе. Так, например, тестирование на основе классов эквивалентности отнесено к методу чёрного ящика, но оно подходит и для написания модульных тест-кейсов, являющихся ярчайшими представителями тестирования по методу белого ящика.
Воспринимайте данные из предстваленной ниже таблицы не как «этот вид тестирования может применяться только для…», а как «чаще всего этот вид тестирования применяется для…»

Таблица 3.d – Виды и техники тестирования в контексте методов белого и чёрного ящиков

Вид тестирования (русскоязычное название)	**Вид тестирования (англоязычное название)**	**Белый ящик**	**Чёрный ящик**
Статическое тестирование	Static testing	Да	Нет
Динамическое тестирование	Dynamic testing	Изредка	Да
Ручное тестирование	Manual testing	Мало	Да
Автоматизированное тестирование	Automated testing	Да	Да
Модульное (компонентное) тестирование	Unit testing, Module testing, Component testing	Да	Нет
Интеграционное тестирование	Integration testing	Да	Да
Системное тестирование	System testing	Мало	Да
Дымовое тестирование	Smoke test, Intake test, Build verification test	Мало	Да
Тестирование критического пути	Critical path test	Мало	Да
Расширенное тестирование	Extended test	Мало	Да
Позитивное тестирование	Positive testing	Да	Да
Негативное тестирование	Negative testing, Invalid testing	Да	Да
Тестирование веб-приложений	Web-applications testing	Да	Да

Продолжение таблицы 3.d

Вид тестирования (русскоязычное название)	**Вид тестирования (англоязычное название)**	**Белый ящик**	**Чёрный ящик**
Тестирование мобильных приложений	Mobile applications testing	Да	Да
Тестирование настольных приложений	Desktop applications testing	Да	Да
Тестирование уровня представления	Presentation tier testing	Мало	Да
Тестирование уровня бизнес-логики	Business logic tier testing	Да	Да
Тестирование уровня данных	Data tier testing	Да	Мало
Альфа-тестирование	Alpha testing	Мало	Да
Гамма-тестирование	Gamma testing	Почти никогда	Да
Тестирование на основе тест-кейсов	Scripted testing, Test case based testing	Да	Да
Исследовательское тестирование	Exploratory testing	Нет	Да
Свободное (интуитивное) тестирование	Ad hoc testing	Нет	Да
Функциональное тестирование	Functional testing	Да	Да
Нефункциональное тестирование	Non-functional testing	Да	Да
Инсталляционное тестирование	Installation testing	Изредка	Да
Регрессионное тестирование	Regression testing	Да	Да
Повторное тестирование	Re-testing, Confirmation testing	Да	Да
Приёмочное тестирование	Acceptance testing	Крайне редко	Да
Операционное тестирование	Operational testing	Крайне редко	Да
Тестирование удобства использования	Usability testing	Крайне редко	Да
Тестирование доступности	Accessibility testing	Крайне редко	Да
Тестирование интерфейса	Interface testing	Да	Да
Тестирование безопасности	Security testing	Да	Да

Продолжение таблицы 3.d

Вид тестирования (русскоязычное название)	**Вид тестирования (англоязычное название)**	**Белый ящик**	**Чёрный ящик**
Тестирование интернационализации	Internationalization testing	Мало	Да
Тестирование локализации	Localization testing	Мало	Да
Тестирование совместимости	Compatibility testing	Мало	Да
Конфигурационное тестирование	Configuration testing	Мало	Да
Кроссбраузерное тестирование	Cross-browser testing	Мало	Да
Тестирование данных и баз данных	Data quality testing and Database integrity testing	Да	Мало
Тестирование использования ресурсов	Resource utilization testing	Крайне редко	Да
Сравнительное тестирование	Comparison testing	Нет	Да
Демонстрационное тестирование	Qualification testing	Нет	Да
Избыточное тестирование	Exhaustive testing	Крайне редко	Нет
Тестирование надёжности	Reliability testing	Крайне редко	Да
Тестирование восстанавливаемости	Recoverability testing	Крайне редко	Да
Тестирование отказоустойчивости	Failover testing	Крайне редко	Да
Тестирование производительности	Performance testing	Крайне редко	Да
Нагрузочное тестирование	Load testing, Capacity testing	Крайне редко	Да
Тестирование масштабируемости	Scalability testing	Крайне редко	Да
Объёмное тестирование	Volume testing	Крайне редко	Да
Стрессовое тестирование	Stress testing	Крайне редко	Да
Конкурентное тестирование	Concurrency testing	Крайне редко	Да
Инвазивное тестирование	Intrusive testing	Да	Да
Неинвазивное тестирование	Nonintrusive testing	Да	Да

Продолжение таблицы 3.d

Вид тестирования (русскоязычное название)	**Вид тестирования (англоязычное название)**	**Белый ящик**	**Чёрный ящик**
Тестирование под управлением данными	Data-driven testing	Да	Да
Тестирование под управлением ключевыми словами	Keyword-driven testing	Да	Да
Тестирование предугадыванием ошибок	Error guessing	Крайне редко	Да
Эвристическая оценка	Heuristic evaluation	Нет	Да
Мутационное тестирование	Mutation testing	Да	Да
Тестирование добавлением ошибок	Error seeding	Да	Да
Тестирование на основе классов эквивалентности	Equivalence partitioning	Да	Да
Тестирование на основе граничных условий	Boundary value analysis	Да	Да
Доменное тестирование	Domain testing, Domain analysis	Да	Да
Попарное тестирование	Pairwise testing	Да	Да
Тестирование на основе ортогональных массивов	Orthogonal array testing	Да	Да
Тестирование в процессе разработки	Development testing	Да	Да
Тестирование по потоку управления	Control flow testing	Да	Нет
Тестирование по потоку данных	Data flow testing	Да	Нет
Тестирование по диаграмме или таблице состояний	State transition testing	Да	Нет
Инспекция (аудит) кода	Code review, code inspection	Да	Нет
Тестирование на основе выражений	Statement testing	Да	Нет
Тестирование на основе ветвей	Branch testing	Да	Нет
Тестирование на основе условий	Condition testing	Да	Нет
Тестирование на основе комбинаций условий	Multiple condition testing	Да	Нет

Продолжение таблицы 3.d

Вид тестирования (русскоязычное название)	**Вид тестирования (англоязычное название)**	**Белый ящик**	**Чёрный ящик**
Тестирование на основе отдельных условий, порождающих ветвление («решающих условий»)	Modified condition decision coverage testing	Да	Нет
Тестирование на основе решений	Decision testing	Да	Нет
Тестирование на основе путей	Path testing	Да	Нет
Тестирование по таблице принятия решений	Decision table testing	Да	Да
Тестирование по моделям поведения приложения	Model-based testing	Да	Да
Тестирование на основе вариантов использования	Use case testing	Да	Да
Параллельное тестирование	Parallel testing	Да	Да
Тестирование на основе случайных данных	Random testing	Да	Да
A/B-тестирование	A/B testing, Split testing	Нет	Да
Восходящее тестирование	Bottom-up testing	Да	Да
Нисходящее тестирование	Top-down testing	Да	Да
Гибридное тестирование	Hybrid testing	Да	Да
Тестирование на основе дерева классификаций	Classification tree method	Да	Да
Тестирование на основе синтаксиса	Syntax testing	Да	Да
Комбинаторные техники (комбинаторное тестирование)	Combinatorial testing	Да	Да
Тестирование всех комбинаций	All combinations testing	Да	Нет
Тестирование с выбором значений-представителей	Each choice testing	Да	Нет
Тестирование с выбором базового набора значений	Base choice testing	Да	Нет
Тестирование по графу причинно-следственных связей	Cause-effect graphing	Мало	Да
Проверка использования всех объявлений	All-definitions testing	Да	Нет
Проверка всех вычислений на основе всех объявлений	All-c-uses testing	Да	Нет

Продолжение таблицы 3.d

Вид тестирования (русскоязычное название)	Вид тестирования (англоязычное название)	Белый ящик	Чёрный ящик
Проверка всех ветвлений на основе всех объявлений	All-p-uses testing	Да	Нет
Проверка всех вычислений и ветвлений на основе всех объявлений	All-uses testing	Да	Нет
Проверка использования всех объявлений и всех путей без переобъявлений (без циклов или с однократными повторениями циклов)	All-du-paths testing	Да	Нет

3.5 Контрольные вопросы и задания

- По каким признакам можно классифицировать тестирование?
- Перечислите виды тестирования по запуску кода на исполнение.
- Перечислите виды тестирования по доступу к коду и архитектуре приложения. Дайте им сравнительную характеристику.
- Перечислите виды тестирования по степени автоматизации. Дайте им сравнительную характеристику.
- Перечислите виды тестирования по уровню детализации приложения.
- Что такое дымовое тестирование?
- Что такое тестирование критического пути?
- Что такое расширенное тестирование?
- Что такое позитивное тестирование?
- Что такое негативное тестирование?
- Каким должно быть поведение системы в ответ на негативный тест?
- Дайте определение и приведите примеры статического тестирования.
- Дайте определение и приведите примеры динамического тестирования.
- Перечислите виды тестирования по привлечению конечных пользователей.
- Опишите процесс тестирования на основе тест-кейсов.
- Опишите процесс исследовательского тестирования.
- Опишите процесс интуитивного тестирования.
- Что такое функцинальное тестирование?
- Что такое нефункцинальное тестирование?
- Что такое регрессионное тестирование?
- Что такое тестирование удобства использования?
- Что такое тестирование доступности?
- Что такое тестирование интерфейса?
- Что такое тестирование безопасности?
- Что такое тестирование интернационализации?
- Что такое тестирование локализации?
- Что такое тестирование совместимости?
- Что такое тестирование производительности?
- Какие существуют виды тестирования производительности?
- Какие существуют виды тестирования на основе структур кода?

4 ЧЕК-ЛИСТЫ, ТЕСТ-КЕЙСЫ, НАБОРЫ ТЕСТ-КЕЙСОВ

4.1 Чек-листы

Тестировщику приходится работать с огромным количеством информации, выбирать из множества вариантов решения задач и изобретать новые. В процессе этой деятельности объективно невозможно удержать в голове все мысли, а потому продумывание и разработку тест-кейсов рекомендуется выполнять с использованием так называемых «чек-листов».

Чек-лист (checklist[266]) – набор идей [тест-кейсов]. Последнее слово не зря взято в скобки, т. к. в общем случае чек-лист – это просто набор идей: идей по тестированию, идей по разработке, идей по планированию и управлению – **любых** идей.

Чек-лист чаще всего представляет собой обычный и привычный нам список, который может быть:

- списком, в котором последовательность пунктов не имеет значения (например, список значений некоего поля);
- списком, в котором последовательность пунктов важна (например, шаги в краткой инструкции);
- структурированным (многоуровневым) списком (вне зависимости от учёта последовательности пунктов), что позволяет отразить иерархию идей.

Важно понять, что нет и не может быть никаких запретов и ограничений при разработке чек-листов – главное, чтобы они помогали в работе. Иногда чек-листы могут даже выражаться графически (например, с использованием ментальных карт[267] или концепт-карт[268]), хотя традиционно их составляют в виде многоуровневых списков.

Поскольку в разных проектах встречаются однотипные задачи, хорошо продуманные и аккуратно оформленные чек-листы могут использоваться повторно, чем достигается экономия сил и времени.

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 груду мыслей, которые никак не связаны друг с другом.

Последовательность и структурированность. Со структурированностью всё достаточно просто – она достигается за счёт оформления чек-листа в виде многоуровневого списка. Что до последовательности, то даже в том случае, когда пункты чек-листа не описывают цепочку действий, человеку всё равно удобнее воспринимать информацию в виде некоторых небольших групп идей, переход между которыми является понятным и очевидным (например, сначала можно прописать идеи простых по-

[266] Понятие «чек-листа» не завязано на тестирование как таковое – это совершенно универсальная техника, которая может применяться в любой без исключения области жизни. В русском языке вне контекста информационных технологий чаще используется понятное и привычное слово «список» (например, «список покупок», «список дел» и т. д.), но в тестировании прижилась калькированная с английского версия – «чек-лист».

[267] Mind map. URL: http://en.wikipedia.org/wiki/Mind_map.

[268] Concept map. URL: http://en.wikipedia.org/wiki/Concept_map.

зитивных тест-кейсов, потом идеи простых негативных тест-кейсов, затем 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.

Правильно создавать и оформлять чек-листы также помогает восприятие их не только как хранилища наборов идей, но как «требования для составления тест-кейсов». Эта мысль приводит к пересмотру и переосмыслению свойств качественных требований (см. подраздел 2.5 «Свойства качественных требований») в применении к чек-листам.

Задание 4.а: Перечитайте подраздел 2.5 «Свойства качественных требований» и подумайте, какие свойства качественных требований можно также считать и свойствами качественных чек-листов.

Итак, рассмотрим процесс создания чек-листа.

Поскольку мы не можем сразу «протестировать всё приложение» (это слишком большая задача), нам уже сейчас нужно выбрать какую-либо логику построения чек-листов: в общем случае чек-листов будет несколько; в итоге их можно будет структурированно объединить в один, но это не обязательно.

Типичными вариантами такой логики является создание отдельных чек-листов:

- для различных уровней функционального тестирования;
- отдельных частей (модулей и подмодулей) приложения (см. «Модуль и подмодуль приложения» в подразделе 4.3 «Атрибуты (поля) тест-кейса»);
- отдельных требований, групп требований, уровней и типов требований (см. подраздел 2.4 «Уровни и типы требований»);
- типичных пользовательских сценариев (см. «Пользовательские сценарии (сценарии использования)» в подразделе 4.6 «Наборы тест-кейсов»);
- частей или функций приложения, наиболее подверженных рискам.

Этот список можно расширять и дополнять, можно комбинировать его пункты, получая, например, чек-листы для проверки наиболее типичных сценариев, затрагивающих некоторую часть приложения.

Чтобы проиллюстрировать принципы построения чек-листов, мы воспользуемся логикой разбиения функций приложения по степени их важности (классификацией по убыванию степени важности функций приложения) на три категории:

- Базовые функции, без которых существование приложения теряет смысл (т. е. самые важные – то, ради чего приложение вообще создавалось), или нарушение работы которых создаёт объективные серьёзные проблемы для среды исполнения.
- Функции, востребованные большинством пользователей в их повседневной работе.
- Остальные функции (разнообразные «мелочи», проблемы с которыми несильно повлияют на ценность приложения для конечного пользователя).

Рассмотрим в качестве примера приложение «Конвертер файлов» (см. подраздел 2.7 «Пример анализа и тестирования требований»). Выделим функции, без которых существование данного приложения теряет смысл. Сначала приведём целиком весь чек-лист для дымового тестирования, а потом рассмотрим его подробнее:

- Конфигурирование и запуск.
- Обработка файлов (таблица 4.a).

Таблица 4.a – Обработка файлов

Кодировки входных файлов	Форматы входных файлов		
	TXT	HTML	MD
WIN1251	+	+	+
CP866	+	+	+
KOI8R	+	+	+

- Остановка.

Да, и всё. Здесь перечислены все ключевые функции приложения.

Конфигурирование и запуск. Если приложение невозможно настроить для работы в пользовательской среде, оно бесполезно. Если приложение не запускается, оно бесполезно. Если на стадии запуска возникают проблемы, они могут негативно отразиться на функционировании приложения и потому также заслуживают пристального внимания.

Примечание – В нашем примере мы столкнулись со скорее нетипичным случаем – приложение конфигурируется параметрами командной строки, а поэтому разделить операции «конфигурирования» и «запуска» не представляется возможным; в реальной жизни для подавляющего большинства приложений эти операции выполняются раздельно.

Обработка файлов. Приложение разрабатывалось для обработки файлов, поэтому здесь, даже на стадии создания чек-листа, мы не поленились создать матрицу, отражающую все возможные комбинации допустимых форматов и допустимых кодировок входных файлов, чтобы ничего не забыть и подчеркнуть важность соответствующих проверок.

Остановка. С точки зрения пользователя эта функция не кажется столь уж важной, но остановка (и запуск) любого приложения связаны с большим количеством системных операций, проблемы с которыми могут привести к множеству серьёзных последствий (вплоть до невозможности повторного запуска приложения или нарушения работы операционной системы).

Рассмотрим функции, востребованные большинством пользователей. Следующим шагом мы будем выполнять проверку того, как приложение ведёт себя в обычной повседневной жизни, пока не затрагивая «экзотические» ситуации. Актуальным вопросом является допустимость дублирования проверок на разных уровнях функционального тестирования – можно ли так делать? Одновременно и «нет», и «да». «Нет» – в том смысле, что не допускается (не имеет смысла) повторение тех же проверок, которые только что были выполнены. «Да» – в том смысле, что любую проверку можно детализировать и снабдить дополнительными деталями:

- Конфигурирование и запуск:
  - с верными параметрами:
    - значения SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME указаны и содержат пробелы и кириллические символы (повторить для форматов путей в Windows- и *nix-файловых системах, обратить внимание на имена логических дисков и разделители имён каталогов (“/” и “\”));
    - значение LOG_FILE_NAME не указано;
  - без параметров;
  - с недостаточным количеством параметров;

    - с неверными параметрами:
        - недопустимый путь SOURCE_DIR;
        - недопустимый путь DESTINATION_DIR;
        - недопустимое имя LOG_FILE_NAME;
        - DESTINATION_DIR находится внутри SOURCE_DIR;
        - значения DESTINATION_DIR и SOURCE_DIR совпадают.
- Обработка файлов:
    - разные форматы, кодировки и размеры (таблица 4.b).

Таблица 4.b – Разные форматы и кодировки входных файлов

Кодировки входных файлов	Форматы входных файлов		
	TXT	HTML	MD
WIN1251	100 Кбайт	50 Мбайт	10 Мбайт
CP866	10 Мбайт	100 Кбайт	50 Мбайт
KOI8R	50 Мбайт	10 Мбайт	100 Кбайт
Любая	0 байт		
Любая	50 Мбайт + 1 байт	50 Мбайт + 1 байт	50 Мбайт + 1 байт
—	Любой недопустимый формат		
Любая	Повреждения в допустимом формате		

    - недоступные входные файлы:
        - нет прав доступа;
        - файл открыт и заблокирован;
        - файл с атрибутом «только для чтения».
- Остановка:
    - закрытием окна консоли.
- Журнал работы приложения:
    - автоматическое создание (при отсутствии журнала);
    - продолжение (дополнение журнала) при повторных запусках.
- Производительность:
    - элементарный тест с грубой оценкой.

Обратите внимание, что чек-лист может содержать не только «предельно сжатые тезисы», но и вполне развёрнутые комментарии. Если это необходимо, лучше пояснить суть идеи подробнее, чем потом гадать, что же имелось в виду.

Также обратите внимание, что многие пункты чек-листа носят весьма высокоуровневый характер, и это нормально. Например, «повреждения в допустимом формате» (см. матрицу с кодировками, форматами и размерами) звучит расплывчато, но этот недочёт будет устранён уже на уровне полноценных тест-кейсов.

Рассмотрим остальные функции и особые сценарии. Пришло время обратить внимание на разнообразные мелочи и «хитрые» нюансы, проблемы с которыми едва ли сильно озаботят пользователя, но формально всё же будут считать ошибками:

- Конфигурирование и запуск:
    - значения SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME:
        - в разных стилях (Windows-пути + *nix-пути) – одно в одном стиле, другое – в другом;
        - с использованием UNC-имён;
        - LOG_FILE_NAME внутри SOURCE_DIR;
        - LOG_FILE_NAME внутри DESTINATION_DIR;

- размер LOG_FILE_NAME на момент запуска:
  - 2–4 Гбайт;
  - 4+ Гбайт;
- запуск двух и более копий приложения:
  - с одинаковыми параметрами SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME;
  - одинаковыми SOURCE_DIR и LOG_FILE_NAME, но разными DESTINATION_DIR;
  - одинаковыми DESTINATION_DIR и LOG_FILE_NAME, но разными SOURCE_DIR.

- Обработка файлов:
  - файл верного формата, в котором текст представлен в двух и более поддерживаемых кодировках одновременно;
  - размер входного файла:
    - 2–4 Гбайт;
    - 4+ Гбайт.

Задание 4.b: Возможно, вам захотелось что-то изменить в приведённых выше чек-листах – это совершенно нормально и справедливо: нет и не может быть некоего «единственно верного идеального чек-листа» и ваши идеи вполне имеют право на существование, поэтому составьте свой собственный чек-лист или отметьте недостатки, которые вы заметили в приведённом выше чек-листе.

Как мы увидим в подразделе 4.2, создание качественного тест-кейса может потребовать длительной кропотливой и достаточно монотонной работы, которая при этом не требует от опытного тестировщика высокого уровня интеллектуальных способностей, а по этой причине переключение между работой над чек-листами (творческая составляющая) и расписыванием их в тест-кейсы (механическая составляющая) позволяют разнообразить рабочий процесс и снизить утомляемость. Хотя, конечно, написание сложных и притом качественных тест-кейсов может оказаться ничуть не менее творческой работой, чем продумывание чек-листов.

4.2 Тест-кейс и его жизненный цикл

Терминология и общие сведения

Для начала определимся с терминологией, поскольку здесь есть много путаницы, вызванной разными переводами англоязычных терминов на русский язык и разными традициями в тех или иных странах, фирмах и отдельных командах.

Во главе всего лежит термин «тест». Официальное определение звучит так.

Тест (test[269]) – набор из одного или нескольких тест-кейсов.

Поскольку среди всех прочих терминов этот легче и быстрее всего произносить, в зависимости от контекста под ним могут понимать и отдельный пункт чек-листа, и отдельный шаг в тест-кейсе, и сам тест-кейс, и набор тест-кейсов и т. д. Главное здесь одно: если вы слышите или видите слово «тест», воспринимайте его в контексте.

[269] **Test.** A set of one or more test cases. ISTQB Glossary.

Теперь рассмотрим самый главный для нас термин – «тест-кейс».

Тест-кейс (test case[270]) – набор входных данных, условий выполнения и ожидаемых результатов, разработанный с целью проверки того или иного свойства или поведения программного средства.
Под тест-кейсом также можно понимать соответствующий документ, представляющий формальную запись тест-кейса.

Мы ещё вернёмся к этой мысли (подраздел 4.7 «Логика создания эффективных проверок»), но уже сейчас критически важно понять и запомнить: если у тест-кейса не указаны входные данные, условия выполнения и ожидаемые результаты, и/или не ясна цель тест-кейса – это плохой тест-кейс (иногда он не имеет смысла, иногда его и вовсе невозможно выполнить).

Примечание – Иногда термин test case на русский язык переводят как «тестовый случай». Это точный перевод, но из-за того что «тест-кейс» удобнее произносить, наибольшее распространение получил именно этот вариант.

Остальные термины, связанные с тестами, тест-кейсами и тестовыми сценариями, на данном этапе можно изучить в ознакомительных целях. Если вы откроете ISTQB-глоссарий на букву «Т», то увидите огромное количество терминов, тесно связанных друг с другом перекрёстными ссылками: на начальном этапе изучения тестирования нет необходимости глубоко рассматривать их все, однако некоторые всё же заслуживают внимания. Они представлены ниже.

Высокоуровневый тест-кейс (high level test case[271]) – тест-кейс без конкретных входных данных и ожидаемых результатов.

Как правило, ограничивается общими идеями и операциями, схож по своей сути с подробно описанным пунктом чек-листа. Достаточно часто встречается в интеграционном тестировании и системном тестировании, а также на уровне дымового тестирования. Может служить отправной точкой для проведения исследовательского тестирования или для создания низкоуровневых тест-кейсов.

Низкоуровневый тест-кейс (low level test case[272]) – тест-кейс с конкретными входными данными и ожидаемыми результатами.

Представляет собой «полностью готовый к выполнению» тест-кейс и вообще является наиболее классическим видом тест-кейсов. Начинающих тестировщиков чаще всего учат писать именно такие тесты, т. к. прописать все данные подробно намного проще, чем понять, какой информацией можно пренебречь, при этом не занижая ценность тест-кейса.

[270] **Test case.** A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. ISTQB Glossary.
[271] **High level test case (logical test case).** A test case without concrete (implementation level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available. ISTQB Glossary.
[272] **Low level test case.** A test case with concrete (implementation level) values for input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators. ISTQB Glossary.

Спецификация тест-кейса (test case specification[273]) – документ, описывающий набор тест-кейсов (включая их цели, входные данные, условия и шаги выполнения, ожидаемые результаты) для тестируемого элемента (test item[274], test object[275]).

Спецификация теста (test specification[276]) – документ, состоящий из спецификации тест-дизайна (test design specification[277]), спецификации тест-кейса (test case specification[273]) и/или спецификации тест-процедуры (test procedure specification[278]).

Тест-сценарий (test scenario[279], test procedure specification, test script) – документ, описывающий последовательность действий по выполнению теста (также известен как «тест-скрипт»).

Внимание! Из-за особенностей перевода очень часто под термином «тест-сценарий» («тестовый сценарий») имеют в виду набор тест-кейсов.

Цель написания тест-кейсов

Тестирование можно проводить и без тест-кейсов (не нужно, но можно; да, эффективность такого подхода варьируется в очень широком диапазоне в зависимости от множества факторов). Наличие же тест-кейсов позволяет:

- Структурировать и систематизировать подход к тестированию (без чего крупный проект почти гарантированно обречён на провал).
- Вычислять метрики тестового покрытия (test coverage[280] metrics) и принимать меры по его увеличению (тест-кейсы здесь являются главным источником информации, без которого существование подобных метрик теряет смысл).
- Отслеживать соответствие текущей ситуации плану (сколько примерно понадобится тест-кейсов, сколько уже есть, сколько выполнено из запланированного на данном этапе количества и т. д.).
- Уточнить взаимопонимание между заказчиком, разработчиками и тестировщиками (тест-кейсы зачастую намного более наглядно показывают поведение приложения, чем это отражено в требованиях).
- Хранить информацию для длительного использования и обмена опытом между сотрудниками и командами (или как минимум – не пытаться удержать в голове сотни страниц текста).
- Проводить регрессионное тестирование и повторное тестирование (которые без тест-кейсов было бы вообще невозможно выполнить).
- Повышать качество требований (мы это уже рассматривали («Тест-кейсы и

[273] **Test case specification.** A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item. ISTQB Glossary.
[274] **Test item.** The individual element to be tested. There usually is one test object and many test items. ISTQB Glossary.
[275] **Test object.** The component or system to be tested. ISTQB Glossary.
[276] **Test specification.** A document that consists of a test design specification, test case specification and/or test procedure specification. ISTQB Glossary.
[277] **Test design specification.** A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases. ISTQB Glossary.
[278] **Test procedure specification (test procedure).** A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. ISTQB Glossary.
[279] **Test scenario.** A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. ISTQB Glossary.
[280] **Coverage (test coverage).** The degree, expressed as a percentage, to which a specified coverage item (an entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements) has been exercised by a test suite. ISTQB Glossary.

чек-листы» в подразделе 1.6 «Техники тестирования требований»): написание чек-листов и тест-кейсов – хорошая техника тестирования требований).

- Быстро вводить в курс дела нового сотрудника, недавно подключившегося к проекту.

Жизненный цикл тест-кейса

В отличие от отчёта о дефекте, у которого есть полноценный развитый жизненный цикл (подраздел 5.2 «Отчёт о дефекте и его жизненный цикл»), для тест-кейса речь скорее идёт о наборе состояний (рисунок 4.а), в которых он может находиться.

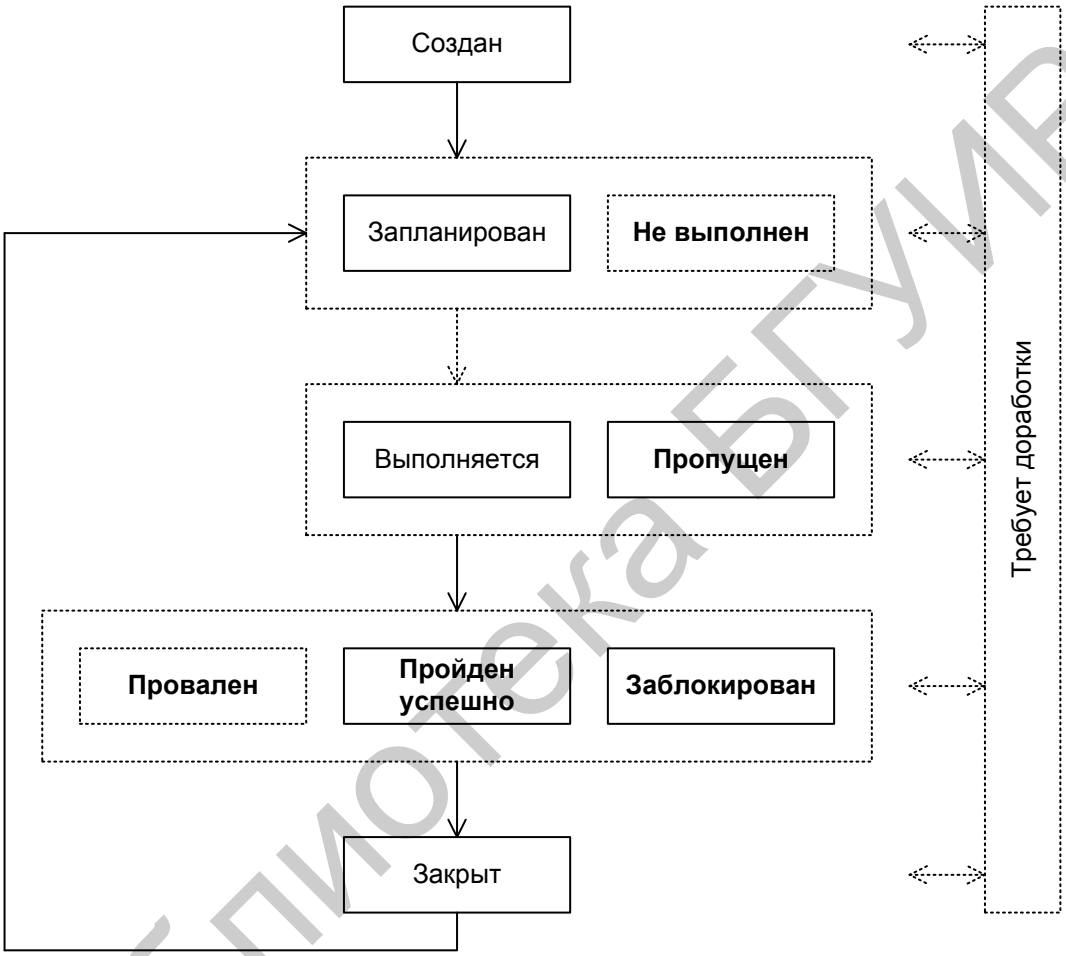


Рисунок 4.а – Жизненный цикл (набор состояний) тест-кейса

Примечание – Жирным шрифтом отмечены наиболее важные состояния.

Рассмотрим жизненный цикл тест-кейса подробно.

- **Создан (new)** – типичное начальное состояние практически любого артефакта. Тест-кейс автоматически переходит в это состояние после создания.
- **Запланирован (planned, ready for testing)** – в этом состоянии тест-кейс находится, когда он или явно включён в план ближайшей итерации тестирования, или как минимум готов для выполнения.
- **Не выполнен (not tested)** – в некоторых системах управления тест-кейсами это состояние заменяет собой предыдущее («запланирован»). Нахождение тест-кейса в данном состоянии означает, что он готов к выполнению, но ещё не был выполнен.
- Выполняется (work in progress) – если тест-кейс требует длительного времени на выполнение, он может быть переведён в это состояние для подчёркивания

того факта, что работа идёт, и скоро можно ожидать её результатов. Если выполнение тест-кейса занимает мало времени, это состояние, как правило, пропускается, а тест-кейс сразу переводится в одно из трёх следующих состояний – «провален», «пройден успешно» или «заблокирован».

- **Пропущен (skipped)** – бывают ситуации, когда выполнение тест-кейса отменяется по соображениям нехватки времени или изменения логики тестирования.
- **Провален (failed)** – данное состояние означает, что в процессе выполнения тест-кейса был обнаружен дефект, заключающийся в том, что ожидаемый результат как минимум по одному шагу тест-кейса не совпадает с фактическим результатом. Если в процессе выполнения тест-кейса был «случайно» обнаружен дефект, никак не связанный с шагами тест-кейса и их ожидаемыми результатами, тест-кейс считается пройденным успешно (при этом, естественно, по обнаруженному дефекту создаётся отчёт о дефекте).
- **Пройден успешно (passed)** – данное состояние означает, что в процессе выполнения тест-кейса не было обнаружено дефектов, связанных с расхождением ожидаемых и фактических результатов его шагов.
- **Заблокирован (blocked)** – данное состояние означает, что по какой-то причине выполнение тест-кейса невозможно (как правило, такой причиной является наличие дефекта, не позволяющего реализовать некий пользовательский сценарий).
- Закрыт (closed) – очень редкий случай, т. к. тест-кейс, как правило, оставляют в состояниях «провален / пройден успешно / заблокирован / пропущен». В некоторых системах управления тест-кейс переводят в данное состояние, чтобы подчеркнуть тот факт, что на данной итерации тестирования все действия с ним завершены.
- Требует доработки (not ready) – как видно из схемы, в это состояние (и из него) тест-кейс может быть преведён в любой момент времени, если в нём будет обнаружена ошибка, если изменятся требования, по которым он был написан, или наступит иная ситуация, не позволяющая считать тест-кейс пригодным для выполнения и перевода в иные состояния.

Ещё раз подчеркнём, что в отличие от жизненного цикла дефекта, который достаточно стандартизирован и формализован, для тест-кейса описанное выше носит общий рекомендательный характер, рассматривается скорее как разрозненный набор состояний (а не строгий жизненный цикл) и может сильно отличаться в разных кампаниях (в связи с имеющимися традициями и/или возможностями систем управления тест-кейсами).

4.3 Атрибуты (поля) тест-кейса

Как уже было сказано выше, термин «тест-кейс» может относиться к формальной записи тест-кейса в виде технического документа. Эта запись имеет общепринятую структуру, компоненты которой называются атрибутами (полями) тест-кейса.

В зависимости от инструмента управления тест-кейсом внешний вид их записи может немного отличаться, могут быть добавлены или убраны отдельные поля, но концепция остаётся неизменной.

Общий вид всей структуры тест-кейса представлен на рисунке 4.b.

UG_U1.12	A	R97	Галерея	Загрузка файла	**Галерея, загрузка файла, имя со спецсимволами** Приготовления: создать непустой файл с именем #$%^&.jpg. 1. Нажать кнопку «Загрузить картинку». 2. Нажать кнопку «Выбрать». 3. Выбрать из списка приготовленный файл. 4. Нажать кнопку «OK». 5. Нажать кнопку «Добавить в галерею»	1. Появляется окно загрузки картинки. 2. Появляется диалоговое окно браузера выбора файла для загрузки. 3. Имя выбранного файла появляется в поле «Файл». 4. Диалоговое окно файла закрывается, в поле «Файл» появляется полное имя файла. 5. Выбранный файл появляется в списке файлов галереи

Рисунок 4.b – Общий вид тест-кейса

Теперь рассмотрим каждый атрибут подробно.

Идентификатор (identifier) представляет собой уникальное значение, позволяющее однозначно отличить один тест-кейс от другого и используемое во всевозможных ссылках. В общем случае идентификатор тест-кейса может представлять собой просто уникальный номер, но (если позволяет инструментальное средство управления тест-кейсами) может быть и куда сложнее: включать префиксы, суффиксы и иные осмысленные компоненты, позволяющие быстро определить цель тест-кейса и часть приложения (или требований), к которой он относится.

Приоритет (priority) показывает важность тест-кейса. Он может быть выражен буквами (A, B, C, D, E), цифрами (1, 2, 3, 4, 5), словами («крайне высокий», «высокий», «средний», «низкий», «крайне низкий») или иным удобным способом. Количество градаций также не фиксировано, но чаще всего лежит в диапазоне от трёх до пяти.

Приоритет тест-кейса может коррелировать:

- с важностью требования, пользовательского сценария (см. «Пользовательские сценарии (сценарии использования)» в подразделе 4.6 «Наборы тест-кейсов») или функции, с которыми связан тест-кейс;
- потенциальной важностью дефекта (см. «Важность (severity)» в подразделе 4.6 «Наборы тест-кейсов»), на поиск которого направлен тест-кейс;
- степенью риска, связанного с проверяемым тест-кейсом требованием, сценарием или функцией.

Основная задача этого атрибута – упрощение распределения внимания и усилий команды (более высокоприоритетные тест-кейсы получают их больше), а также упрощение планирования и принятия решения о том, чем можно пожертвовать в некоторой форс-мажорной ситуации, не позволяющей выполнить все запланированные тест-кейсы.

Связанное с тест-кейсом требование (requirement) показывает то основное требование, проверке выполнения которого посвящён тест-кейс (основное потому, что один тест-кейс может затрагивать несколько требований). Наличие этого поля улучшает такое свойство тест-кейса, как прослеживаемость (см. «Прослеживаемость» в подразделе 4.6 «Наборы тест-кейсов»).

Часто задаваемые вопросы, связанные с заполнением этого поля, таковы:

- Можно ли его оставить пустым? Да. Тест-кейс вполне мог разрабатываться вне прямой привязки к требованиям, и (пока) значение этого поля определить сложно. Хотя такой вариант и не считается безупречным, он достаточно распространён.
- Можно ли в этом поле указывать несколько требований? Да, но чаще всего стараются выбрать одно самое главное или «более высокоуровневое» (например, вместо того, чтобы перечислять R56.1, R56.2, R56.3 и т. д., можно просто написать R56). Чаще всего в инструментах управления тестами это поле представляет собой выпадающий список, где можно выбрать только одно значение, и этот вопрос становится неактуальным. К тому же многие тест-кейсы всё же направлены на проверку строго одного требования, и для них этот вопрос также неактуален.

Модуль и подмодуль приложения (module and submodule) указывают на части приложения, к которым относится тест-кейс, и позволяют лучше понять его цель.

Идея деления приложения на модули и подмодули проистекает из того, что в сложных системах практически невозможно охватить взглядом весь проект целиком, и вопрос «как протестировать это приложение» становится недопустимо сложным. Тогда приложение логически разделяется на компоненты (модули), а те, в свою очередь, – на более мелкие компоненты (подмодули). И вот уже для таких небольших частей приложения придумать чек-листы и создать хорошие тест-кейсы становится намного проще.

Как правило, иерархия модулей и подмодулей создаётся как единый набор для всей проектной команды, чтобы исключить путаницу из-за того, что разные люди будут использовать разные подходы к такому разделению или даже просто разные названия одних и тех же частей приложения.

Теперь о самом сложном: как выбираются модули и подмодули. В реальности проще всего отталкиваться от архитектуры и дизайна приложения. Например, в уже знакомом нам приложении можно выделить такую иерархию модулей и подмодулей:

- Механизм запуска:
  - механизм анализа параметров;
  - механизм сборки приложения;
  - механизм обработки ошибочных ситуаций.
- Механизм взаимодействия с файловой системой:
  - механизм обхода дерева SOURCE_DIR;
  - механизм обработки ошибочных ситуаций.
- Механизм преобразования файлов:
  - механизм определения кодировок;
  - механизм преобразования кодировок;
  - механизм обработки ошибочных ситуаций.

- Механизм ведения журнала:
  - механизм записи журнала;
  - механизм отображения журнала в консоли;
  - механизм обработки ошибочных ситуаций.

Согласитесь, что такие длинные названия с постоянно повторяющимся словом «механизм» читать и запоминать сложно. Перепишем:

- Стартер:
  - анализатор параметров;
  - сборщик приложения;
  - обработчик ошибок.
- Сканер:
  - обходчик;
  - обработчик ошибок.
- Преобразователь:
  - детектор;
  - конвертер;
  - обработчик ошибок.
- Регистратор:
  - дисковый регистратор;
  - консольный регистратор;
  - обработчик ошибок.

Но что делать, если мы не знаем «внутренностей» приложения (или не очень разбираемся в программировании)? Модули и подмодули можно выделять на основе графического интерфейса пользователя (крупные области и элементы внутри них), на основе решаемых приложением задач и подзадач и т. д. Главное, чтобы эта логика была одинаковым образом применена ко всему приложению.

Внимание! Часто встречающаяся ошибка! Модуль и подмодуль приложения – это НЕ действия, это именно структурные части, «куски» приложения. В заблуждение вас могут ввести такие названия, как, например, «печать, настройка принтера» (но здесь имеются в виду именно части приложения, отвечающие за печать и за настройку принтера (и названы они отглагольными существительными), а не процесс печати или настройки принтера).
Сравните (на примере человека): «дыхательная система, лёгкие» – это модуль и подмодуль, а «дыхание», «сопение», «чихание» – нет; «голова, мозг» – это модуль и подмодуль, а «кивание», «мышление» – нет.

Наличие полей «Модуль и подмодуль приложения» улучшает такое свойство тест-кейса, как прослеживаемость (см. «Прослеживаемость» в подразделе 4.6 «Наборы тест-кейсов»).

Заглавие (суть) тест-кейса (title) призвано упростить быстрое понимание основной идеи тест-кейса без обращения к его остальным атрибутам.

Заглавие тест-кейса может быть полноценным предложением, фразой, набором словосочетаний – главное, чтобы выполнялись следующие условия:

- Информативность.
- Уникальность (хотя бы относительная, чтобы не путать разные тест-кейсы).

Именно заглавие является наиболее информативным полем при просмотре списка тест-кейсов. Сравните (таблица 4.c).

Таблица 4.с – Сравнение заглавий тест-кейсов

Плохо	Хорошо
Тест 1	Запуск, одна копия, верные параметры
Тест 2	Запуск одной копии с неверными путями
Тест 78 (улучшенный)	Запуск, много копий, без конфликтов
Остановка	Остановка по Ctrl+C
Закрытие	Остановка закрытием консоли
…	…

Внимание! Часто встречающаяся ошибка! Если инструмент управления тест-кейсами не требует писать заглавие, его **всё равно надо писать**. Тест-кейсы без заглавий превращаются в путанную информацию, использование которой сопряжено с колоссальными и совершенно бессмысленными затратами.

И ещё одно небольшое уточнение, которое может помочь лучше формулировать заглавия. В дословном переводе с английского «test case» обозначает «тестовый случай (ситуация)». Так вот, заглавие как раз и описывает этот случай (ситуацию), т. е. что происходит в тест-кейсе, какую ситуацию он проверяет.

Исходные данные, необходимые для выполнения тест-кейса (precondition, preparation, initial data, setup), позволяют описать всё то, что должно быть подготовлено до начала выполнения тест-кейса, например:

- Состояние базы данных.
- Состояние файловой системы и её объектов.
- Состояние серверов и сетевой инфраструктуры.

Всё, что описывается в этом поле, готовится БЕЗ использования тестируемого приложения, и таким образом, если здесь возникают проблемы, нельзя писать отчёт о дефекте в приложении. Эта мысль очень и очень важна, поэтому поясним её простым жизненным примером. Представьте, что вы дегустируете конфеты. В поле «исходные данные» можно прописать «купить конфеты таких-то сортов в таком-то количестве». Если таких конфет нет в продаже, закрыт магазин, не хватило денег – всё это НЕ проблемы вкуса конфет, и нельзя писать отчёт о дефекте конфет вида «конфеты невкусные потому, что закрыт магазин».

Некоторые авторы не следуют этой логике и допускают в приготовлениях работу с тестируемым приложением. И здесь нет «правильного варианта» – просто в одной традиции решено одним образом, в другой – другим. Во многом это ещё и терминологическая проблема: «preparation», «initial data» и «set up» вполне логично выполнять без участия тестируемого приложения, в то время как «precondition» по смыслу ближе к описанию состояния тестируемого приложения. В реальной рабочей обстановке вам достаточно будет прочитать несколько тест-кейсов, уже созданных вашими коллегами, чтобы понять, какой точки зрения по данному вопросу они придерживаются.

Шаги тест-кейса (steps) описывают последовательность действий, которые необходимо реализовать в процессе выполнения тест-кейса. Общие рекомендации по написанию шагов таковы:

- Начинайте с понятного и очевидного места, не пишите лишних начальных шагов (запуск приложения, очевидные операции с интерфейсом и т. п.).

- Даже если в тест-кейсе всего один шаг, нумеруйте его (иначе возрастает вероятность в будущем случайно «приклеить» описание этого шага к новому тексту).
- Если вы пишете на русском языке, используйте безличную форму (например, «открыть», «ввести», «добавить» вместо «откройте», «введите», «добавьте»).
- Соотносите степень детализации шагов и их параметров с целью тест-кейса, его сложностью, уровнем и т. д. – в зависимости от этих и многих других факторов степень детализации может варьироваться от общих идей до предельно чётко прописанных значений и указаний.
- Ссылайтесь на предыдущие шаги и их диапазоны для сокращения объёма текста (например, «повторить шаги 3–5 со значением…»).
- Пишите шаги последовательно, без условных конструкций вида «если…, то…».

Внимание! Часто встречающаяся ошибка! Категорически запрещено ссылаться на шаги из других тест-кейсов и другие тест-кейсы целиком: если те, другие тест-кейсы будут изменены или удалены, ваш тест-кейс начнёт ссылаться на неверные данные или в пустоту, а если в процессе выполнения те, другие тест-кейсы или шаги приведут к возникновению ошибки, вы не сможете закончить выполнение вашего тест-кейса.

Ожидаемые результаты (expected results) по каждому шагу тест-кейса описывают реакцию приложения на действия, описанные в поле «шаги тест-кейса». Номер шага соответствует номеру результата.

По написанию ожидаемых результатов можно порекомендовать следующее:

- Описывайте поведение системы так, чтобы исключить субъективное толкование (например, «приложение работает верно» – плохо, «появляется окно с надписью…» – хорошо).
- Пишите ожидаемый результат по всем шагам без исключения, если у вас есть хоть малейшие сомнения в том, что результат какого-либо шага будет совершенно тривиальным и очевидным (если вы всё же пропускаете ожидаемый результат для какого-то тривиального действия, лучше оставить в списке ожидаемых результатов пустую строку – это облегчает восприятие).
- Пишите кратко, но не в ущерб информативности.
- Избегайте условных конструкций вида «если…, то…».

Внимание! Часто встречающаяся ошибка! В ожидаемых результатах ВСЕГДА описывается КОРРЕКТНАЯ работа приложения. Нет и не может быть ожидаемого результата в виде «приложение вызывает ошибку в операционной системе и аварийно завершается с потерей всех пользовательских данных».

При этом корректная работа приложения вполне может предполагать отображение сообщений о неверных действиях пользователя или неких критических ситуациях. Так, сообщение «Невозможно сохранить файл по указанному пути: на целевом носителе недостаточно свободного места» – это не ошибка приложения, это его совершенно нормальная и правильная работа. Ошибкой приложения (в этой же ситуации) было бы отсутствие такого сообщения, и/или повреждение, или потеря записываемых данных.

4.4 Инструментальные средства управления тестированием

Инструментальных средств управления тестированием (test management tool[281]) очень много[282], к тому же многие компании разрабатывают свои внутренние средства решения этой задачи.

Не имеет смысла заучивать, как работать с тест-кейсами в том или ином инструментальном средстве – принцип везде един, и соответствующие навыки нарабатываются буквально за пару дней. Что на текущий момент важно понимать, так это общий набор функций, реализуемых такими инструментальными средствами (конечно, то или иное средство может не реализовывать какую-то функцию из этого списка и/или реализовывать не вошедшие в список функции):

- создание тест-кейсов и наборов тест-кейсов;
- контроль версий документов с возможностью определить, кто внёс те или иные изменения, и отменить эти изменения, если требуется;
- формирование и отслеживание реализации плана тестирования, сбор и визуализация разнообразных метрик, генерирование отчётов;
- интеграция с системами управления дефектами, фиксация взаимосвязи между выполнением тест-кейсов и созданными отчётами о дефектах;
- интеграция с системами управления проектами;
- интеграция с инструментами автоматизированного тестирования, управление выполнением автоматизированных тест-кейсов.

Иными словами, хорошее инструментальное средство управления тестированием берёт на себя все рутинные технические операции, которые объективно необходимо выполнять в процессе реализации жизненного цикла тестирования (см. подраздел 1.2 «Жизненный цикл тестирования»). Огромным преимуществом также является способность таких инструментальных средств отслеживать взаимосвязи между различными документами и иными артефактами, взаимосвязи между артефактами и процессами и т. д., при этом обеспечивая разграничение прав доступа, сохранность и корректность информации.

Для общего развития и лучшего закрепления темы об оформлении тест-кейсов мы сейчас рассмотрим несколько картинок с формами из разных инструментальных средств.

Здесь вполне сознательно не приведено никакого сравнения или подробного описания – подобных обзоров достаточно в Интернете, и они стремительно устаревают по мере выхода новых версий обозреваемых продуктов.

Но интерес представляют отдельные особенности интерфейса, на которые мы обратим внимание в примере (рисунок 4.c). Важно: если вас интересует подробное описание каждого поля, связанных с ним процессов и т. д., обратитесь к официальной документации, а здесь даны лишь краткие пояснения:

[281] Test management tool. A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting. ISTQB Glossary.
[282] Test management tools. URL: http://www.opensourcetesting.org/testmgt.php.

QAComplete[283]

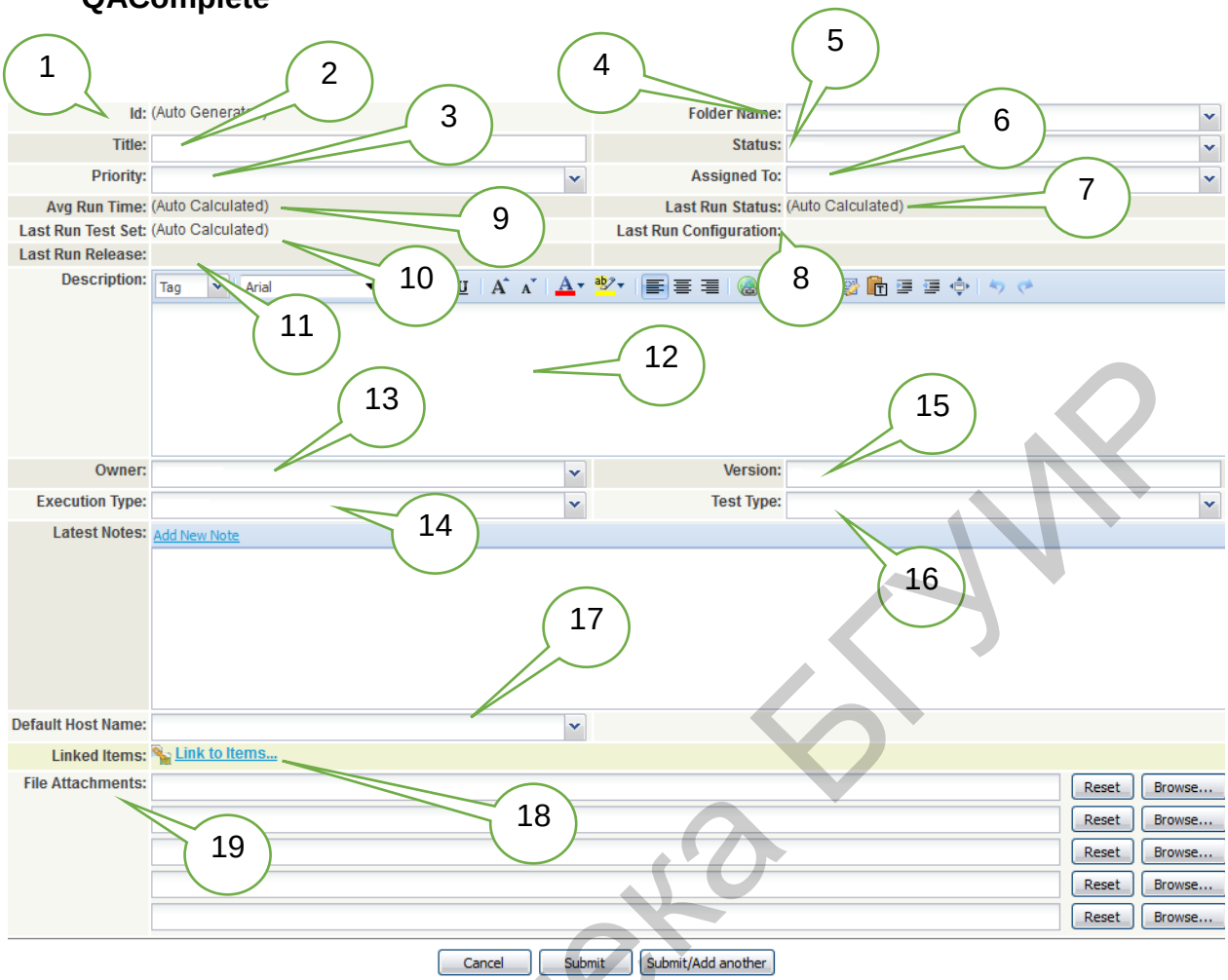


Рисунок 4.c – Создание тест-кейса в QAComplete

1. Id (идентификатор), как видно из соответствующей надписи, автогенерируемый.
2. Title (заглавие), как и в большинстве систем, является обязательным для заполнения.
3. Priority (приоритет) предлагает значения: high (высокий), medium (средний), low (низкий).
4. Folder name (расположение) является аналогом полей «Модуль» и «Подмодуль» и позволяет выбрать из выпадающего древовидного списка соответствующее значение, описывающее, к чему относится тест-кейс.
5. Status (статус) показывает текущее состояние тест-кейса: new (новый), approved (утверждён), awaiting approval (на рассмотрении), in design (в разработке), outdated (устарел), rejected (отклонён).
6. Assigned to (исполнитель) указывает, кто в данный момент является «основной рабочей силой» по данному тест-кейсу (или кто должен принять решение о, например, утверждении тест-кейса).
7. Last Run Status (результат последнего запуска) показывает, прошёл ли тест успешно (passed) или завершился неудачей (failed).
8. Last Run Configuration (конфигурация, использованная для последнего запус-

[283] QAComplete. URL: http://smartbear.com/product/test-management-tool/qacomplete/.

ка) показывает, на какой аппаратно-программной платформе тест-кейс выполнялся в последний раз.

9. Avg Run Time (среднее время выполнения) содержит вычисленное автоматически среднее время, необходимое на выполнение тест-кейса.
10. Last Run Test Set (в последний раз выполнялся в наборе) содержит информацию о наборе тест-кейсов, в рамках которого тест-кейс выполнялся последний раз.
11. Last Run Release (последний раз выполнялся на выпуске) содержит информацию о выпуске (билде) программного средства, на котором тест-кейс выполнялся последний раз.
12. Description (описание) позволяет добавить любую полезную информацию о тест-кейсе (включая особенности выполнения, приготовления и т. д.).
13. Owner (владелец) указывает на владельца тест-кейса (как правило – его автора).
14. Execution Type (способ выполнения) по умолчанию предлагает только значение manual (ручное выполнение), но при соответствующих настройках и интеграции с другими продуктами список можно расширить (как минимум добавить automated (автоматизированное выполнение)).
15. Version (версия) содержит номер текущей версии тест-кейса (фактически, счётчик того, сколько раз тест-кейс редактировали). Вся история изменений сохраняется, предоставляя возможность вернуться к любой из предыдущих версий.
16. Test Type (вид теста) по умолчанию предлагает такие варианты, как negative (негативный), positive (позитивный), regression (регрессионный), smoke-test (дымный).
17. Default host name (имя хоста по умолчанию) в основном используется в автоматизированных тест-кейсах и предлагает выбрать из списка имя зарегистрированного компьютера, на котором установлен специальный клиент.
18. Linked Items (связанные объекты) представляют собой ссылки на требования, отчёты о дефектах и т. д.
19. File Attachments (вложения) могут содержать тестовые данные, поясняющие изображения, видеоролики и т. д.

При необходимости можно добавить и настроить дополнительные поля, значительно расширяющие исходные возможности системы.

4.5 Свойства качественных тест-кейсов

Даже правильно оформленный тест-кейс может оказаться некачественным, если в нём нарушено одно из следующих свойств.

Правильный технический язык. Это свойство в равной мере относится и к требованиям, и к тест-кейсам, и к отчётам о дефектах – к любой документации. Общие требования:

- Пишите лаконично, но понятно.
- Используйте безличную форму глаголов (например, «открыть» вместо «откройте»).
- Обязательно указывайте точные имена и технически верные названия элементов приложения.
- Не объясняйте базовые принципы работы с компьютером (предполагается, что ваши коллеги знают, что такое, например, «пункт меню» и как с ним работать).

Баланс между специфичностью и общностью. Тест-кейс считается тем бо-

лее специфичным, чем более детально в нём расписаны конкретные действия, конкретные значения и т. д., т. е. чем в нём больше конкретики. Соответственно, тест-кейс считается тем более общим, чем в нём меньше конкретики.

Рассмотрим поля «шаги» и «ожидаемые результаты» двух тест-кейсов (подумайте, какой тест-кейс вы бы посчитали хорошим, а какой – плохим и почему).

Тест-кейс 1 предствлен в таблице 4.d, а тест-кейс 2 – в таблице 4.e.

Таблица 4.d – Тест-кейс 1

Шаги	Ожидаемые результаты
Конвертация из всех поддерживаемых кодировок Приготовления: • Создать папки C:/A, C:/B, C:/C, C:/D. • Разместить в папке C:/D файлы 1.html, 2.txt, 3.md из прилагаемого архива. 1. Запустить приложение, выполнив команду «php converter.php c:/a c:/b c:/c/converter.log». 2. Скопировать файлы 1.html, 2.txt, 3.md из папки C:/D в папку C:/A. 3. Остановить приложение нажатием Crtl+C	1. Отображается консольный журнал приложения с сообщением «текущее_время started, source dir c:/a, destination dir c:/b, log file c:/c/converter.log», в папке C:/C появляется файл converter.log, в котором появляется запись «текущее_время started, source dir c:/a, destination dir c:/b, log file c:/c/converter.log». 2. Файлы 1.html, 2.txt, 3.md появляются в папке C:/A, затем пропадают оттуда и появляются в папке C:/B. В консольном журнале и файле C:/C/converter.log появляются сообщения (записи) «текущее_время processing 1.html (KOI8-R)», «текущее_время processing 2.txt (CP-1251)», «текущее_время processing 3.md (CP-866)». 3. В файле C:/C/converter.log появляется запись «текущее_время closed». Приложение завершает работу

Таблица 4.e – Тест-кейс 2

Шаги	Ожидаемые результаты
Конвертация из всех поддерживаемых кодировок 1. Выполнить конвертацию трёх файлов допустимого размера в трёх разных кодировках всех трёх допустимых форматов	1. Файлы перемещаются в папку-приёмник, кодировка всех файлов меняется на UTF-8

Если вернуться к вопросу, какой тест-кейс вы бы посчитали хорошим, а какой – плохим, и почему, то ответ таков: оба тест-кейса плохие, потому что первый является слишком специфичным, а второй – слишком обобщённым. Можно сказать, что здесь до абсурда доведены идеи низкоуровневых (см. «Низкоуровневый тест-кейс» в подразделе 4.6 «Наборы тест-кейсов») и высокоуровневых (см. «Высокоуровневый тест-кейс» в подразделе 4.6 «Наборы тест-кейсов») тест-кейсов.

Проблемы, связанные с излишней специфичностью (см. тест-кейс 1):

- При повторных выполнениях тест-кейса всегда будут выполняться строго одни и те же действия со строго одними и теми же данными, что снижает вероятность обнаружения ошибки.
- Возрастает время написания, доработки и даже просто прочтения тест-кейса.

- В случае выполнения тривиальных действий опытные специалисты тратят дополнительные мыслительные ресурсы в попытках понять, что же они упустили из виду, т. к. они привыкли, что так описываются только самые сложные и неочевидные ситуации.

Проблемы, связанные с излишней общностью (см. тест-кейс 2):

- Тест-кейс сложен для выполнения начинающими тестировщиками или даже опытными специалистами, лишь недавно подключившимися к проекту.
- Недобросовестные сотрудники склонны халатно относиться к таким тест-кейсам.
- Тестировщик, выполняющий тест-кейс, может понять его иначе, чем было задумано автором (и в итоге будет выполнен фактически совсем другой тест-кейс).

Выход из этой ситуации состоит в том, чтобы придерживаться золотой середины (хотя, конечно же, какие-то тесты будут чуть более специфичными, какие-то – чуть более общими). Вот пример такого срединного подхода: тест-кейс 3 (таблица 4.f).

Таблица 4.f – Тест-кейс 3

Шаги	Ожидаемые результаты
Конвертация из всех поддерживаемых кодировок Приготовления: • Создать в корне любого диска четыре отдельные папки для входных файлов, выходных файлов, файла журнала и временного хранения тестовых файлов. • Распаковать содержимое прилагаемого архива в папку для временного хранения тестовых файлов. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать файлы из папки для временного хранения в папку для входных файлов. 3. Остановить приложение	1. Приложение запускается и выводит сообщение о своём запуске в консоль и файл журнала. 2. Файлы из папки для входных файлов перемещаются в папку для выходных файлов, в консоли и файле журнала отображаются сообщения о конвертации каждого из файлов с указанием его исходной кодировки. 3. Приложение выводит сообщение о завершении работы в файл журнала и завершает работу

В этом тест-кейсе есть всё необходимое для понимания и выполнения, но при этом он стал короче и проще для выполнения, а отсутствие строго указанных значений приводит к тому, что при многократном выполнении тест-кейса (особенно – разными тестировщиками) конкретные параметры будут менять свои значения, что увеличивает вероятность обнаружения ошибки.

Ещё раз выразим главную мысль: сами по себе специфичность или общность тест-кейса не являются чем-то плохим, но резкий перегиб в ту или иную сторону снижает качество тест-кейса.

Баланс между простотой и сложностью. Здесь не существует академических определений, но принято считать, что простой тест-кейс оперирует одним объектом (или в нём явно виден главный объект), а также содержит небольшое количество тривиальных действий; сложный тест-кейс оперирует несколькими равноправными объектами и содержит много нетривиальных действий.

Преимущества простых тест-кейсов:

- Их можно быстро прочесть, легко понять и выполнить.
- Они понятны начинающим тестировщикам и новым людям на проекте.
- Они делают наличие ошибки очевидным (как правило, в них предполагается выполнение повседневных тривиальных действий, проблемы с которыми видны невооружённым взглядом и не вызывают дискуссий).
- Они упрощают начальную диагностику ошибки, т. к. сужают круг поиска.

Преимущества сложных тест-кейсов:

- При взаимодействии многих объектов повышается вероятность возникновения ошибки.
- Пользователи, как правило, используют сложные сценарии, а потому сложные тесты более полноценно эмулируют работу пользователей.
- Программисты редко проверяют такие сложные случаи (и это совершенно не входит в их обязанности).

Рассмотрим примеры.

Слишком простой тест-кейс представлен в таблице 4.g.

Таблица 4.g – Простой тест-кейс

Шаги	Ожидаемые результаты
Запуск приложения 1. Запустить приложение	1. Приложение запускается

Слишком сложный тест-кейс представлен в таблице 4.h.

Таблица 4.h – Сложный тест-кейс

Шаги	Ожидаемые результаты
Повторная конвертация Приготовления: • Создать в корне любого диска три отдельные папки для входных файлов, выходных файлов, файла журнала. • Подготовить набор из нескольких файлов максимального поддерживаемого размера поддерживаемых форматов с поддерживаемыми кодировками, а также нескольких файлов допустимого размера, но недопустимого формата. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать в папку для входных файлов несколько файлов допустимого формата. 3. Переместить сконвертированные файлы из папки с результирующими файлами в папку для входных файлов.	1. Приложение запускается. 2. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов. 3. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов. 5. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов допустимого формата и сообщения об игнорировании файлов недопустимого формата. 6. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов допустимого формата и сообщения об игнорировании файлов недопустимого формата

Продолжение таблицы 4.h

Шаги	Ожидаемые результаты
4. Переместить сконвертированные файлы из папки с результирующими файлами в папку с набором файлов для теста. 5. Переместить все файлы из папки с набором файлов для теста в папку для входных файлов. 6. Переместить сконвертированные файлы из папки с результирующими файлами в папку для входных файлов	

Этот тест-кейс одновременно является слишком сложным по избыточности действий и по спецификации лишних данных и операций.

Задание 4.d: Перепишите этот тест-кейс, устранив его недостатки, но сохранив общую цель (проверку повторной конвертации уже ранее сконвертированных файлов).

Примером хорошего простого тест-кейса может служить тест-кейс 3 из пункта про специфичность и общность.

Пример хорошего сложного тест-кейса приведён в таблице 4.i.

Таблица 4.i – Хороший сложный тест-кейс

Шаги	Ожидаемые результаты
Много копий приложения, конфликт файловых операций Приготовления: • Создать в корне любого диска три отдельные папки для входных файлов, выходных файлов, файла журнала. • Подготовить набор из нескольких файлов максимального поддерживаемого размера поддерживаемых форматов с поддерживаемыми кодировками. 1. Запустить первую копию приложения, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Запустить вторую копию приложения с теми же параметрами (см. шаг 1). 3. Запустить третью копию приложения с теми же параметрами (см. шаг 1).	3. Все три копии приложения запускаются, в файле журнала появляются последовательно три записи о запуске приложения. 5. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов, а также (возможно) сообщения вида: a. «source file inaccessible, retrying»; b. «destination file inaccessible, retrying»; c. «log file inaccessible, retrying». Ключевым показателем корректной работы является успешная конвертация всех файлов, а также появление в консоли и файле журнала сообщений об успешной конвертации каждого файла (от одной до трёх записей на каждый файл).

Продолжение таблицы 4.i

Шаги	Ожидаемые результаты
4. Изменить приоритет процессов второй («high») и третей («low») копий. Скопировать подготовленный набор исходных файлов в папку для входных файлов	Сообщения (предупреждения) о недоступности входного файла, выходного файла или файла журнала также являются показателем корректной работы приложения, однако их количество зависит от многих внешних факторов и не может быть спрогнозировано заранее

Иногда более сложные тест-кейсы являются также и более специфичными, но это лишь общая тенденция, а не закон. Также нельзя по сложности тест-кейса однозначно судить о его приоритете (в нашем примере хорошего сложного тест-кейса он явно будет иметь очень низкий приоритет, т. к. проверяемая им ситуация является искусственной и крайне маловероятной, но бывают и сложные тесты с самым высоким приоритетом).

Как и в случае специфичности и общности, сами по себе простота или сложность тест-кейсов не являются чем-то плохим (более того – рекомендуется начинать разработку и выполнение тест-кейсов с простых, а затем переходить ко всё более и более сложным), однако излишняя простота и излишняя сложность также снижают качество тест-кейса.

«Показательность» (высокая вероятность обнаружения ошибки). Начиная с уровня тестирования критического пути можно утверждать, что тест-кейс является тем более хорошим, чем он более показателен (с большей вероятностью обнаруживает ошибку). Именно поэтому мы считаем непригодными слишком простые тест-кейсы – они непоказательны.

Пример непоказательного (плохого) тест-кейса представлен в таблице 4.j.

Таблица 4.j – Непоказательный тест-кейс

Шаги	Ожидаемые результаты
Запуск и остановка приложения 1. Запустить приложение с корректными параметрами. 2. Завершить работу приложения	1. Приложение запускается. 2. Приложение завершает работу

Пример показательного (хорошего) тест-кейса представлен в таблице 4.k.

Таблица 4.k – Показательный (хороший) тест-кейс

Шаги	Ожидаемые результаты
Запуск с некорректными параметрами, несуществующие пути 1. Запустить приложение со всеми тремя параметрами (SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME), значения которых указывают на несуществующие в файловой системе пути (например: z:\src\, z:\dst\ при условии, что в системе нет логического диска z)	1. В консоли отображаются нижеуказанные сообщения, приложение завершает работу. Сообщения: a. SOURCE_DIR: directory not exists or inaccessible. b. DESTINATION_DIR: directory not exists or inaccessible. c. LOG_FILE_NAME: wrong file name or inaccessible path

Обратите внимание, что показательный тест-кейс по-прежнему остался достаточно простым, но он проверяет ситуацию, возникновение ошибки в которой несравненно более вероятно, чем в ситуации, описываемой плохим непоказательным тест-кейсом.

Также можно сказать, что показательные тест-кейсы часто выполняют какие-то «интересные действия», т. е. такие, которые едва ли будут выполнены просто в процессе работы с приложением (например: «сохранить файл» – это обычное тривиальное действие, которое явно будет выполнено не одну сотню раз даже самими разработчиками, а вот «сохранить файл на носитель, защищённый от записи», «сохранить файл на носитель с недостаточным объёмом свободного пространства», «сохранить файл в папку, к которой нет доступа» – это уже гораздо более интересные и нетривиальные действия).

Последовательность в достижении цели. Суть этого свойства выражается в том, что все действия в тест-кейсе направлены на следование единой логике и достижение единой цели и не содержат никаких отклонений.

Примерами правильной реализации этого свойства могут служить представленные в этом подразделе в избытке примеры хороших тест-кейсов. А нарушение может выглядеть так (таблица 4.l).

Таблица 4.l – Ошибки в тест-кейсах

Шаги	Ожидаемые результаты
Конвертация из всех поддерживаемых кодировок Приготовления: • Создать в корне любого диска четыре отдельные папки для входных файлов, выходных файлов, файла журнала и временного хранения тестовых файлов. • Распаковать содержимое прилагаемого архива в папку для временного хранения тестовых файлов. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать файлы из папки для временного хранения в папку для входных файлов. *3. Остановить приложение.* *4. Удалить файл журнала.* *5. Повторно запустить приложение с теми же параметрами.* 6. Остановить приложение	1. Приложение запускается и выводит сообщение о своём запуске в консоль и файл журнала. 2. Файлы из папки для входных файлов перемещаются в папку для выходных файлов, в консоли и файле журнала отображаются сообщения о конвертации каждого из файлов с указанием его исходной кодировки. *3. Приложение выводит сообщение о завершении работы в файл журнала и завершает работу.* *5. Приложение запускается и выводит сообщение о своём запуске в консоль и заново созданный файл журнала.* *6.* Приложение выводит сообщение о завершении работы в файл журнала и завершает работу

Шаги 3–5 никак не соответствуют цели тест-кейса, состоящей в проверке корректности конвертации входных данных, представленных во всех поддерживаемых кодировках.

Отсутствие лишних действий. Чаще всего это свойство подразумевает, что не нужно в шагах тест-кейса долго и по пунктам расписывать то, что можно заменить одной фразой (таблица 4.m).

Таблица 4.m – Лишние действия в тест-кейсах

Плохо	Хорошо
1. Указать в качестве первого параметра приложения путь к папке с исходными файлами. 2. Указать в качестве второго параметра приложения путь к папке с конечными файлами. 3. Указать в качестве третьего параметра приложения путь к файлу журнала. 4. Запустить приложение	1. Запустить приложение со всеми тремя корректными параметрами (например: c:\src\, c:\dst\, c:\log.txt при условии, что соответствующие папки существуют и доступны приложению)

Вторая по частоте ошибка заключается в том, что начало каждого тест-кейса с запуска приложения и подробного описания по приведению его в то или иное состояние. В наших примерах мы рассматриваем каждый тест-кейс как существующий в единственном виде в изолированной среде, и потому вынуждены осознанно допускать эту ошибку (иначе тест-кейс будет неполным), но в реальной жизни на запуск приложения будут свои тесты, а длинный путь из многих действий можно описать как одно действие, из контекста которого понятно, как это действие выполнить. Следующий пример тест-кейса не относится к нашему «Конвертеру файлов», но очень хорошо иллюстрирует эту мысль (таблица 4.n).

Таблица 4.n – Частые ошибки в тест-кейсах

Плохо	Хорошо
1. Запустить приложение. 2. Выбрать в меню пункт «Файл». 3. Выбрать подпункт «Открыть». 4. Перейти в папку, в которой находится хотя бы один файл формата DOCX с тремя и более страницами	1. Открыть DOCX-файл с тремя и более страницами

К этому же свойству можно отнести ошибку с повторением одних и тех же приготовлений во множестве тест-кейсов (да, по описанным выше причинам в примерах мы снова вынужденно выполняем то, что на практике делать не надо). Куда удобнее объединить тесты в набор и указать приготовления один раз, подчеркнув, нужно или нет их выполнять перед каждым тест-кейсом в наборе.

Проблема с подготовительными (и финальными) действиями идеально решена в автоматизированном модульном тестировании[284] с использованием фреймворков наподобие JUnit или TestNG – там существует специальный «механизм фиксаций» (fixture), автоматически выполняющий указанные действия перед каждым отдельным тестовым методом или после него.

Неизбыточность по отношению к другим тест-кейсам. В процессе создания множества тест-кейсов очень легко оказаться в ситуации, когда два и более тест-кейса фактически выполняют одни и те же проверки, преследуют одни и те же цели, направлены на поиск одних и тех же проблем.

[284] **Unit testing (component testing).** The testing of individual software components. ISTQB Glossary.

Существуют разные способы минимизации количества таких тест-кейсов, такие как использование классов эквивалентности и граничных условий.

Если вы обнаруживаете несколько тест-кейсов, дублирующих задачи друг друга, лучше всего или удалить все, кроме одного, самого показательного, или перед удалением остальных на их основе доработать этот выбранный самый показательный тест-кейс.

Демонстративность (способность демонстрировать обнаруженную ошибку очевидным образом). Ожидаемые результаты должны быть подобраны и сформулированы таким образом, чтобы любое отклонение от них сразу же бросалось в глаза и становилось очевидным, что произошла ошибка. Сравните выдержки из двух тест-кейсов.

Выдержка из недемонстративного тест-кейса представлена в таблице 4.o.

Таблица 4.o – Недемонстративный тест-кейс

Шаги	Ожидаемые результаты
1. Разместить в файле текст «Пример длинного текста, содержащего символы русского и английского алфавита вперемешку» в кодировке KOI8-R (в слове «Пример» буквы «р» – английские). 2. Сохранить файл под именем «test. txt» и отправить файл на конвертацию. 3. Переименовать файл в «test.txt»	1. Приложение игнорирует файл. 2. Текст принимает корректный вид в кодировке UTF-8 с учётом английских букв

Выдержка из демонстративного тест-кейса представлена в таблице 4.p.

Таблица 4.p – Демонстративный тест-кейс

Шаги	Ожидаемые результаты
1. Разместить в файле текст «╨Я╤А╨╗╨╛╨╤А ╤В╨╡╨║╤Б╤В╨▒ ╨▓.» (Эти символы представляют собой словосочетание «Пример текста.» в кодировке KOI8-R). 2. Отправить файл на конвертацию	1. Текст принимает вид: «Пример текста.» (кодировка UTF8)

В первом случае тест-кейс не подходит не только из-за неясности формулировки «корректный вид в кодировке UTF-8 с учётом английских букв», там также очень легко допустить следующие ошибки при выполнении:

- Забыть сконвертировать вручную входной текст в KOI8-R.
- Не заметить, что в первый раз расширение файла начинается с пробела.
- Забыть заменить в слове «Пример» букву «р» на английскую.
- Из-за расплывчатости формулировки ожидаемого результата принять ошибочное, но выглядящее правдоподобно поведение за верное.

Второй тест-кейс чётко ориентирован на свою цель по проверке конвертации (не содержит странной проверки с игнорированием файла с неверным расширением) и описан так, что его выполнение не представляет никаких сложностей, а любое отклонение фактического результата от ожидаемого будет сразу же заметно.

Прослеживаемость. Из содержащейся в качественном тест-кейсе информации должно быть понятно, какую часть приложения, какие функции и какие требования он проверяет. Частично это свойство достигается через заполнение соответствующих полей тест-кейса («Ссылка на требование», «Модуль», «Подмодуль»), но и сама логика тест-кейса играет не последнюю роль, т. к. в случае серьёзных наруше-

ний этого свойства можно долго выяснять, например, на какое требование ссылается тест-кейс, и пытаться понять, как же они друг с другом связаны.

Пример непрослеживаемого тест-кейса представлен в таблице 4.q.

Таблица 4.q – Непрослеживаемый тест-кейс

Требование	**Модуль**	**Подмодуль**	**Шаги**	**Ожидаемые результаты**
ПТ-4	Приложение		**Совмещение кодировок** Приготовления: файл с несколькими допустимыми и недопустимыми кодировками. 1. Передать файл на конвертацию	1. Допустимые кодировки конвертируются верно, недопустимые остаются без изменений

Да, этот тест-кейс не подходит сам по себе (в качественном тест-кейсе сложно получить ситуацию непрослеживаемости), но в нём есть и особые недостатки, затрудняющие прослеживаемость:

- Ссылка на несуществующее требование (убедитесь сами, что требования ПТ-4 нет).
- В поле «Модуль» указано значение «Приложение» (по большому счёту можно было оставлять это поле пустым – это было бы столь же информативно), поле «Подмодуль» не заполнено.
- По заглавию и шагам можно предположить, что этот тест-кейс ближе всего к ДС-5.1 и ДС-5.3, но сформулированный ожидаемый результат не следует явно из этих требований.

  Пример прослеживаемого тест-кейса представлен в таблице 4.r.

Таблица 4.r – Прослеживаемый тест-кейс

Требование	**Модуль**	**Подмодуль**	**Шаги**	**Ожидаемые результаты**
ДС-2.4, ДС-3.2	Стартер	Обработчик ошибок	**Запуск с некорректными параметрами, несуществующие пути** 1. Запустить приложение со всеми тремя параметрами, значения которых указывают на несуществующие в файловой системе пути	1. В консоли отображаются нижеуказанные сообщения, приложение завершает работу. Сообщения: a. SOURCE_DIR: directory not exists or inaccessible; b. DESTINTION_DIR: directory not exists or inaccessible; c. LOG_FILE_NAME: wrong file name or inaccessible path

Можно подумать, что этот тест-кейс затрагивает ДС-2 и ДС-3 целиком, но в поле «Требование» есть вполне чёткая конкретизация, к тому же указанные модуль, подмодуль и сама логика тест-кейса устраняют оставшиеся сомнения.

Возможность повторного использования. Это свойство редко выполняется для низкоуровневых тест-кейсов (см. «Низкоуровневый тест-кейс» в подразделе 2.5 «Наборы тест-кейсов»), но при создании высокоуровневых тест-кейсов (см. «Высокоуровневый тест-кейс» в подразделе 2.5 «Наборы тест-кейсов») можно добиться таких формулировок, при которых тест-кейс практически без изменений можно будет использовать для тестирования аналогичной функциональности в других проектах или других областях приложения.

Примером тест-кейса, который тяжело использовать повторно, может являться практически любой тест-кейс с высокой специфичностью.

Не самым идеальным, но очень наглядным примером тест-кейса, который может быть легко использован в разных проектах, может служить следующий тест-кейс, представленный в таблице 4.s.

Таблица 4.s – Наглядный тест-кейс

Шаги	Ожидаемые результаты
Запуск, все параметры некорректны 1. Запустить приложение, указав в качестве всех параметров заведомо некорректные значения	1. Приложение запускается, после чего выводит сообщение с описанием сути проблемы с каждым из параметров и завершает работу

Соответствие принятым шаблонам оформления и традициям. С шаблонами оформления, как правило, проблем не возникает: они строго определены имеющимся образцом или вообще экранной формой инструментального средства управления тест-кейсами. Что же касается традиций, то они отличаются даже в разных командах в рамках одной компании, и тут невозможно дать иной совет, кроме как «почитайте уже готовые тест-кейсы перед тем, как писать свои».

4.6 Наборы тест-кейсов

Терминология и общие сведения

Набор тест-кейсов (test case suite[285], test suite, test set) – совокупность тест-кейсов, выбранных с некоторой общей целью или по некоторому общему признаку. Иногда в такой совокупности результаты завершения одного тест-кейса становятся входным состоянием приложения для следующего тест-кейса.

Внимание! Из-за особенностей перевода очень часто вместо «набор тест-кейсов» говорят «тестовый сценарий». Формально это можно считать ошибкой, но это явление приобрело настолько широкое распространение, что стало вариантом нормы.

Как мы только что убедились на примере множества отдельных тест-кейсов, крайне неудобно (более того, это ошибка!) каждый раз писать в каждом тест-кейсе одни и те же приготовления и повторять одни и те же начальные шаги.

Намного удобнее объединить несколько тест-кейсов в набор или последовательность. И здесь мы приходим к классификации наборов тест-кейсов.

[285] **Test case suite (test suite, test set).** A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one. ISTQB Glossary.

В общем случае наборы тест-кейсов можно разделить на свободные (порядок выполнения тест-кейсов не важен) и последовательные (порядок выполнения тест-кейсов важен).

Преимущества свободных наборов:

- Тест-кейсы можно выполнять в любом удобном порядке, а также создавать «наборы внутри наборов».
- Если какой-то тест-кейс завершился ошибкой, это не повлияет на возможность выполнения других тест-кейсов.

Преимущества последовательных наборов:

- Каждый следующий в наборе тест-кейс в качестве входного состояния приложения получает результат работы предыдущего тест-кейса, что позволяет сильно сократить количество шагов в отдельных тест-кейсах.
- Длинные последовательности действий куда лучше имитируют работу реальных пользователей, чем отдельные воздействия на приложение.

Пользовательские сценарии (сценарии использования)

	В данном случае речь НЕ идёт о use cases (вариантах использования), представляющих собой форму требований (см. подраздел 2.4 «Уровни и типы требований»). Пользовательские сценарии как техника тестирования куда менее формализованы, хотя и могут строиться на основе вариантов использования.

К отдельному подвиду последовательных наборов тест-кейсов (или даже неоформленных идей тест-кейсов, таких, как пункты чек-листа) можно отнести пользовательские сценарии[286] (или сценарии использования), представляющие собой цепочки действий, выполняемых пользователем в определённой ситуации для достижения определённой цели.

Поясним это сначала на примере, не относящемся к «Конвертеру файлов». Допустим, пользователь хочет распечатать табличку на дверь кабинета с текстом «Идёт работа, не стучать!» Для этого ему нужно:

1) Запустить текстовый редактор.
2) Создать новый документ (*если редактор не делает это самостоятельно*).
3) Набрать в документе текст.
4) Отформатировать текст должным образом.
5) Отправить документ на печать.
6) Сохранить документ (*спорно, но допустим*).
7) Закрыть текстовый редактор.

Вот мы и получили пользовательский сценарий, пункты которого могут стать основой для шагов тест-кейса или целого набора отдельных тест-кейсов.

Сценарии могут быть достаточно длинными и сложными, могут содержать внутри себя циклы и условные ветвления, но при всём этом они обладают рядом весьма интересных преимуществ:

- Сценарии показывают реальные и понятные примеры использования продукта (в отличие от обширных чек-листов, где смысл отдельных пунктов может теряться).
- Сценарии понятны конечным пользователям и хорошо подходят для обсуждения и совместного улучшения.

[286] A scenario is a hypothetical story, used to help a person think through a complex problem or system. Kaner C. An Introduction to Scenario Testing. URL: http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf.

- Сценарии и их части легче оценивать с точки зрения важности, чем отдельные пункты (особенно низкоуровневых) требований.
- Сценарии отлично показывают недоработки в требованиях (если становится непонятно, что делать в том или ином пункте сценария, – с требованиями явно что-то не то).
- В предельном случае (нехватка времени и прочие форс-мажоры) сценарии можно даже не прописывать подробно, а просто именовать – и само наименование уже подскажет опытному специалисту, что делать.

Последний пункт проиллюстрируем на примере. Классифицируем потенциальных пользователей нашего приложения (напомним, что в нашем случае «пользователь» – это администратор, настраивающий работу приложения) по степени квалификации и склонности к экспериментам, а затем дадим каждому «виду пользователя» запоминающееся имя (таблица 4.t).

Таблица 4.t – Степень классификации пользователей

Вид пользователя	**Низкая квалификация**	**Высокая квалификация**
Не склонен к экспериментам	«Осторожный»	«Консервативный»
Склонен к экспериментам	«Отчаянный»	«Изощрённый»

Уже на этой стадии не составляет труда представить себе отличия в логике работы с приложением, например, «консервативного» и «отчаянного» пользователей.

Но мы пойдём дальше и озаглавим для них сами сценарии, например, в ситуациях, когда такой пользователь позитивно и негативно относится к идее внедрения нашего приложения (таблица 4.u).

Таблица 4.u – Сценарии поведения на основе классификации пользователей

Отношение пользователя	**«Осторожный»**	**«Консервативный»**	**«Отчаянный»**	**«Изощрённый»**
Позитивно	«А можно вот так?»	«Начнём с инструкции!»	«Гляньте, что я придумал!»	«Я всё оптимизирую!»
Негативно	«Я ничего не понимаю.»	«У вас вот здесь несоответствие…»	«Всё равно поломаю!»	«А я говорил!»

Проявив даже немного воображения, можно представить, что и как будет происходить в каждой из получившихся восьми ситуаций. Причём на создание пары таких таблиц уходит всего несколько минут, а эффект от их использования значительно превосходит бездумное «кликанье по кнопкам в надежде найти баг».

Ещё более полное и техничное объяснение того, что такое сценарное тестирование, как его применять и выполнять должным образом, можно прочесть в статье Сэма Канера «An Introduction to Scenario Testing»[287].

Подробная классификация наборов тест-кейсов может быть выражена следующей таблицей (таблица 4.v).

[287] Kaner C. An Introduction to Scenario Testing. URL: http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf.

Таблица 4.v – Подробная классификация наборов тест-кейсов

По образованию тест-кейсами строгой последовательности	По изолированности тест-кейсов друг от друга	
	изолированные	**обобщённые**
Свободные	Изолированные свободные	Обобщённые свободные
Последовательные	Изолированные последовательные	Обобщённые последовательные

Следует отметить следующие особенности:

- Набор изолированных свободных тест-кейсов (рисунок 4.f): действия из раздела «Приготовления» нужно повторить перед каждым тест-кейсом, а сами тест-кейсы можно выполнять в любом порядке.
- Набор обобщённых свободных тест-кейсов (рисунок 4.g): действия из раздела «Приготовления» нужно выполнить один раз (а потом просто выполнять тест-кейсы), а сами тест-кейсы можно выполнять в любом порядке.
- Набор изолированных последовательных тест-кейсов (рисунок 4.h): действия из раздела «Приготовления» нужно повторить перед каждым тест-кейсом, а сами тест-кейсы нужно выполнять в строго определённом порядке.
- Набор обобщённых последовательных тест-кейсов (рисунок 4.i): действия из раздела «Приготовления» нужно выполнить один раз (а потом просто выполнять тест-кейсы), а сами тест-кейсы нужно выполнять в строго определённом порядке.

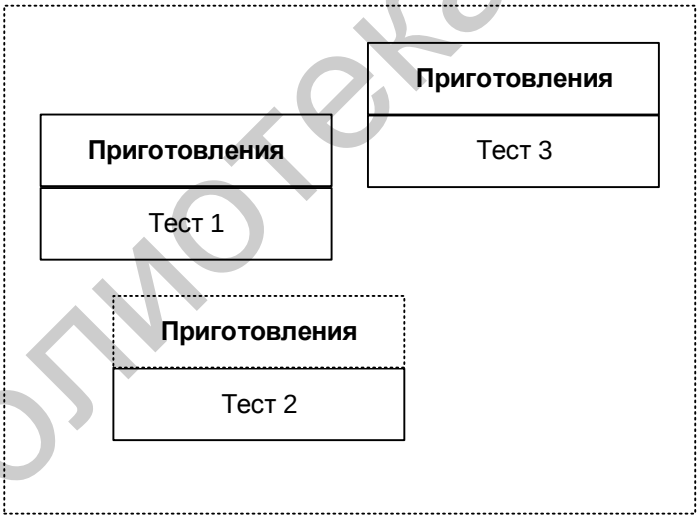


Рисунок 4.f – Набор изолированных свободных тест-кейсов

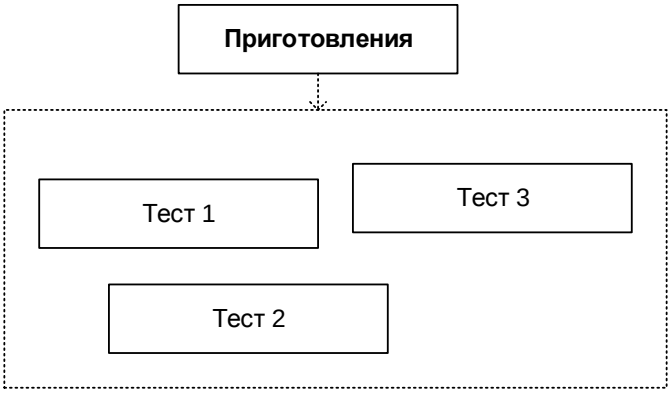


Рисунок 4.g – Набор обобщённых свободных тест-кейсов

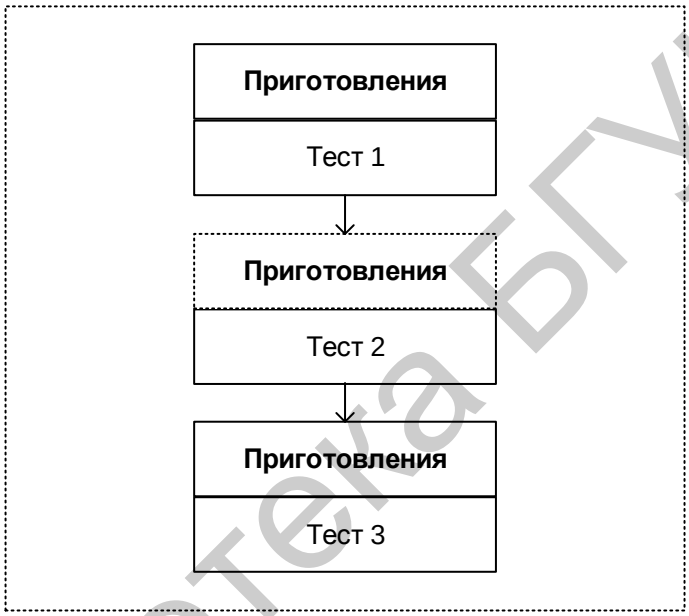


Рисунок 4.h – Набор изолированных последовательных тест-кейсов

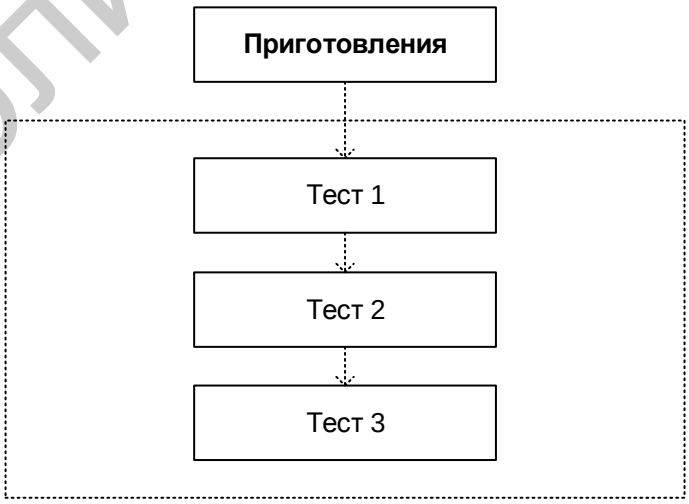


Рисунок 4.i – Набор обобщённых последовательных тест-кейсов

Главное преимущество изолированности: каждый тест-кейс выполняется в «чистой среде», на него не влияют результаты работы предыдущих тест-кейсов.

Главное преимущество обобщённости: приготовления не нужно повторять (экономия времени).

Главное преимущество последовательности: ощутимое сокращение шагов в каждом тест-кейсе, т. к. результат выполнения предыдущего тест-кейса является начальной ситуацией для следующего.

Главное преимущество свободы: возможность выполнять тест-кейсы в любом порядке, а также то, что при провале какого-нибудь тест-кейса (приложение не пришло в ожидаемое состояние) остальные тест-кейсы по-прежнему можно выполнять.

Ниже будут рассмотрены принципы построения наборов тест-кейсов.

Главный вопрос: как формировать наборы тест-кейсов? Правильный ответ звучит очень кратко: логично. И это не шутка. Единственная задача наборов – повысить эффективность тестирования за счёт ускорения и упрощения выполнения тест-кейсов, увеличения глубины исследования некоторой области приложения или функциональности, следования типичным пользовательским сценариям (см. «Пользовательские сценарии (сценарии использования)» в подразделе 2.5 «Наборы тест-кейсов») или удобной последовательности выполнения тест-кейсов и т. д.

Набор тест-кейсов всегда создаётся с какой-то целью, на основе какой-то логики, и по этим же принципам в набор включаются тесты, обладающие подходящими свойствами.

Если же говорить о наиболее типичных подходах к составлению наборов тест-кейсов, то можно обозначить следующие:

- На основе чек-листов. Посмотрите внимательно на примеры чек-листов, которые мы разработали в соответствующем подразделе 4.1 «Чек-листы»: каждый пункт чек-листа может превратиться в несколько тест-кейсов – и вот мы получаем готовый набор.
- На основе разбиения приложения на модули и подмодули (см. «Модуль и подмодуль приложения» в подразделе 4.3 «Атрибуты (поля) тест-кейса»). Для каждого модуля (или его отдельных подмодулей) можно составить свой набор тест-кейсов.
- По принципу проверки самых важных, менее важных и всех остальных функций приложения (именно по этому принципу мы составляли примеры чек-листов в вышеупомянутом подразделе 4.1 «Чек-листы»).
- По принципу группировки тест-кейсов для проверки некоего уровня требований или типа требований (см. подраздел 2.4 «Уровни и типы требований»), группы требований или отдельного требования.
- По принципу частоты обнаружения тест-кейсами дефектов в приложении (например, мы видим, что некоторые тест-кейсы раз за разом завершаются неудачей, значит, мы можем объединить их в набор, условно названный «проблемные места в приложении»).
- По архитектурному принципу: наборы для проверки пользовательского интерфейса и всего уровня представления, для проверки уровня бизнес-логики, для проверки уровня данных.
- По области внутренней работы приложения, например: «тест-кейсы, затрагивающие работу с базой данных», «тест-кейсы, затрагивающие работу с файловой системой», «тест-кейсы, затрагивающие работу с сетью» и т. д.
- По видам тестирования.

Не нужно строго следовать этому списку. Он всего лишь примерный. Важен принцип: если вы видите, что выполнение некоторых тест-кейсов в виде набора принесёт вам пользу, создавайте такой набор.

Примечание – Без хороших инструментальных средств управления тест-кейсами работать с наборами тест-кейсов крайне тяжело, т. к. приходится самостоятельно следить за приготовлениями, «недостающими шагами», изолированностью или обобщённостью, свободой или последовательностью и т. д.

4.7 Логика создания эффективных проверок

Теперь, когда мы рассмотрели принципы построения чек-листов (см. подраздел 4.1 «Чек-листы») и оформления тест-кейсов (см. подраздел 4.3 «Атрибуты (поля) тест-кейса»), свойства качественных тест-кейсов (см. подраздел 4.5 «Свойства качественных тест-кейсов»), а также принципы объединения тест-кейсов в наборы (см. подраздел 4.6 «Наборы тест-кейсов»), настало время перейти к самой сложной, «философской» части, в которой мы будем рассуждать уже не о том, что и как писать, а о том, как думать.

Ранее уже было сказано: если у тест-кейса не указаны входные данные, условия выполнения и ожидаемые результаты, и/или не ясна цель тест-кейса – это плохой тест-кейс. И здесь во главе угла стоит **цель**. Если мы чётко понимаем, что и зачем мы делаем, мы или быстро находим всю остальную недостающую информацию, или столь же быстро формулируем правильные вопросы и адресуем их правильным людям.

Вся суть работы тестировщика в конечном итоге направлена на повышение качества (процессов, продуктов и т. д.) Но что такое качество? Да, существует сухое официальное определение[288], но даже там сказано про «user/customer needs and expectations» (потребности и ожидания пользователя/заказчика).

И здесь проявляется главная мысль: **качество – это некая ценность для конечного пользователя** (заказчика). Человек в любом случае платит за использование продукта – деньгами, своим временем, какими-то усилиями (даже если вы не получаете эту «оплату», человек вполне обоснованно считает, что что-то уже на вас потратил, и он прав). Но получает ли он то, на что рассчитывал? Предположим, что его ожидания – здравы и реалистичны.

Если мы подходим к тестированию формально, то рискуем получить продукт, который по документам (метрикам и т. д.) выглядит идеально, но в реальности никому не нужен.

Поскольку практически любой современный программный продукт представляет собой непростую систему, среди широкого множества её свойств и функций объективно есть самые важные, менее важные и совсем незначительные по важности для пользователей.

Если усилия тестировщиков будут сконцентрированы на первой и второй категориях (самом важном и чуть менее важном), наши шансы создать приложение, удовлетворяющее заказчика, резко увеличиваются.

Есть простая логика:

- Тесты ищут ошибки.
- Но все ошибки найти невозможно.
- Значит, наша задача – найти максимум ВАЖНЫХ ошибок за имеющееся время.

Под важными ошибками здесь мы понимаем такие, которые приводят к нарушению важных для пользователя функций или свойств продукта. Функции и свойства разделены не случайно – безопасность, производительность, удобство и т. д. не относятся к функциям, но играют ничуть не менее важную роль в формировании удовлетворённости заказчика и конечных пользователей.

---

[288] **Quality.** The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations. ISTQB Glossary.

Ситуация усугубляется следующими фактами:

- в силу множества экономических и технических причин мы не можем выполнить «все тесты, что возникли в сознании» (да ещё и многократно) – приходится тщательно выбирать, что и как мы будем тестировать, принимая во внимание только что упомянутую мысль: качество – это некая ценность для конечного пользователя (заказчика);
- никогда в реальной жизни (как бы мы ни старались) мы не получим «совершенного и идеального набора требований» – там всегда будет некоторое количество недоработок, и это тоже надо принимать во внимание.

Однако существует достаточно простой алгоритм, позволяющий нам создавать эффективные проверки даже в таких условиях. Приступая к продумыванию чеклиста, тест-кейса или набора тест-кейсов, задайте себе следующие вопросы и получите чёткие ответы:

- Что перед вами? Если вы не понимаете, что вам предстоит тестировать, вы не пойдёте дальше поверхностных формальных проверок.
- Кому и зачем оно нужно (и насколько это важно)? Ответ на этот вопрос позволит вам быстро придумать несколько характерных сценариев использования (см. подраздел 4.6 «Наборы тест-кейсов») того, что вы собираетесь тестировать.
- Как оно обычно используется? Это уже детализация сценариев и источник идей для позитивного тестирования (их удобно оформить в виде чек-листа).
- Как оно может сломаться, т. е. начать работать неверно? Это также детализация сценариев использования, но уже в контексте негативного тестирования (см. подраздел 3.2 «Подробная классификация тестирования») (их тоже удобно оформить в виде чек-листа).

К этому алгоритму можно добавить ещё небольшой перечень универсальных рекомендаций, которые позволят вам проводить тестирование лучше:

- Начинайте как можно раньше – уже с момента появления первых требований можно заниматься их тестированием и улучшением, можно писать чек-листы и тест-кейсы, можно уточнять план тестирования, готовить тестовое окружение и т. д.
- Если вам предстоит тестировать что-то большое и сложное, разбивайте его на модули и подмодули, функциональность подвергайте функциональной декомпозиции[289] – т. е. добейтесь такого уровня детализации, при котором вы можете без труда удержать в голове всю информацию об объекте тестирования.
- Обязательно пишите чек-листы. Если вам кажется, что вы сможете запомнить все идеи и потом легко их воспроизвести, вы ошибаетесь. Исключений не бывает.
- По мере создания чек-листов, тест-кейсов и т. д. прямо в текст вписывайте возникающие вопросы. Когда вопросов накопится достаточно, соберите их отдельно, уточните формулировки и обратитесь к тому, кто может дать ответы.
- Если используемое вами инструментальное средство позволяет использовать косметическое оформление текста, то используйте (так текст будет лучше читаться), но старайтесь следовать общепринятым традициям и «не раскрашивать» каждое второе слово в свой цвет, шрифт, размер и т. д.
- Используйте технику беглого просмотра (см. подраздел 2.6 «Техники тестирования требований») для получения отзыва от коллег и улучшения созданного вами документа.
- Планируйте время на улучшение тест-кейсов (исправление ошибок, доработку по факту изменения требований и т. д.).

[289] Functional decomposition. Wikipedia. URL: http://en.wikipedia.org/wiki/Functional_decomposition.

- Начинайте проработку (и выполнение) тест-кейсов с простых позитивных проверок наиболее важной функциональности. Затем постепенно повышайте сложность проверок, помня не только о позитивных (см. подраздел 3.2 «Подробная классификация тестирования»), но и о негативных проверках.
- Помните, что в основе тестирования лежит цель. Если вы не можете быстро и просто сформулировать цель созданного вами тест-кейса, вы создали плохой тест-кейс.
- Избегайте избыточных, дублирующих друг друга тест-кейсов. Минимизировать их количество вам помогут техники классов эквивалентности, граничных условий, доменного тестирования (см. подраздел 3.2 «Подробная классификация тестирования»).
- Если показательность (см. подраздел 4.5 «Свойства качественных тест-кейсов») тест-кейса можно увеличить, при этом не сильно изменив его сложность и не отклонившись от исходной цели, сделайте это.
- Помните, что очень многие тест-кейсы требуют отдельной подготовки, которую нужно описать в соответствующем поле тест-кейса.
- Несколько позитивных тест-кейсов можно безбоязненно объединять, но объединение негативных тест-кейсов почти всегда запрещено (см. подраздел 3.2 «Подробная классификация тестирования»).
- Подумайте, как можно оптимизировать созданный вами тест-кейс (набор тест-кейсов и т. д.) так, чтобы снизить трудозатраты на его выполнение.
- Перед тем как отправлять финальную версию созданного вами документа, ещё раз перечитайте написанное (в доброй половине случаев найдёте опечатку или иную недоработку).

Задание 4.e: Дополните этот список идеями, которые вы почерпнули из других книг, статей и т. д.

Пример реализации логики создания эффективных проверок

Ранее мы составили подробный чек-лист для тестирования нашего «Конвертера файлов». Давайте посмотрим на него критически и подумаем, что можно сократить, чем мы в итоге пожертвуем и какой выигрыш получим.

Прежде чем мы начнём оптимизировать чек-лист, важно отметить, что решение о том, что важно и что неважно, стоит принимать на основе ранжирования требований по важности, а также согласовывать с заказчиком.

Что для пользователя САМОЕ важное? Ради чего создавалось приложение? Чтобы конвертировать файлы. Принимая во внимание тот факт, что настройку приложения будет выполнять квалифицированный технический специалист, мы можем даже «отодвинуть на второй план» реакцию приложения на ошибки стадии запуска и завершения. И на первое место выходят обработка файлов, разные форматы, кодировки и размеры (представлены в таблице 4.w).

Таблица 4.w – Форматы, кодировки и размеры файлов

Кодировки входных файлов	**Форматы входных файлов**		
	TXT	HTML	MD
WIN1251	100 Кбайт	50 Мбайт	10 Мбайт
CP866	10 Мбайт	100 Кбайт	50 Мбайт
KOI8R	50 Мбайт	10 Мбайт	100 Кбайт
Любая	0 байт		
Любая	50 Мбайт + 1 байт	50 Мбайт + 1 байт	50 Мбайт + 1 байт
–	Любой недопустимый формат		
Любая	Повреждения в допустимом формате		

Можем ли мы как-то ускорить выполнение этих проверок (ведь их много)? Можем. И у нас даже есть два взаимодополняющих инструмента:

- дальнейшая классификация по важности;
- автоматизация тестирования.

Сначала разделим таблицу на две части: «чуть более важное» и «чуть менее важное».

«Чуть более важное» представлено в таблице 4.x.

Таблица 4.x – Форматы, кодировки и размеры файлов для проверки «чуть более важного» функционала

Кодировки входных файлов	**Форматы входных файлов**		
	TXT	HTML	MD
WIN1251	100 Кбайт	50 Мбайт	10 Мбайт
CP866	10 Мбайт	100 Кбайт	50 Мбайт
KOI8R	50 Мбайт	10 Мбайт	100 Кбайт

Подготовим 18 файлов – 9 исходных и 9 сконвертированных (в любом текстовом редакторе с функцией конвертации кодировок), чтобы в дальнейшем сравнивать работу нашего приложения с эталоном.

Для «чуть менее важного» осталось:

- файл размером 0 байт (объективно для него не важна такая характеристика, как «кодировка»). *Подготовим один файл размером 0 байт*;
- файл размером 50 Мбайт + 1 байт (для него тоже не важна кодировка). *Подготовим один файл размером 52’428’801 байт*;
- любой недопустимый формат:
    - по расширению (файл с расширением, отличным от .txt, .html, .md). *Берём любой произвольный файл, например, картинку (размер от 1 до 50 Мбайт, расширение .jpg)*;
    - по внутреннему содержанию (например, .jpg переименовали в .txt). *Копии файла из предыдущего пункта даём расширение .txt*;
- повреждения в допустимом формате. *Вычёркиваем. Вообще. Даже крайне сложные дорогие редакторы далеко не всегда способны восстановить повреждённые файлы своих форматов, наше же приложение – это просто миниатюрная утилита конвертации кодировок, и не стоит ждать от неё возможностей профессионального инструментального средства восстановления данных.*

Что мы получили в итоге? Нам нужно подготовить следующие 22 файла. Поскольку у файлов всё равно есть имена, усилим этот набор тестовых данных, представив в именах файлов символы латиницы, кириллицы и спецсимволы (таблица 4.y).

Таблица 4.y – Итоговый набор файлов для тестирования приложения

Имя	Кодировка	Размер
«Мелкий» файл в WIN1251.txt	WIN1251	100 Кбайт
«Средний» файл в CP866.txt	CP866	10 Мбайт
«Крупный» файл в KOI8R.txt	KOI8R	50 Мбайт
«Крупный» файл в win-1251.html	WIN1251	50 Мбайт
«Мелкий» файл в cp-866.html	CP866	100 Кбайт
«Средний» файл в koi8-r.html	KOI8R	10 Мбайт
«Средний» файл в WIN_1251.md	WIN1251	10 Мбайт
«Крупный» файл в CP_866.md	CP866	50 Мбайт
«Мелкий» файл в KOI8_R.md	KOI8R	100 Кбайт
«Мелкий» эталон WIN1251.txt	UTF8	100 Кбайт
«Средний» эталон CP866.txt	UTF8	10 Мбайт
«Крупный» эталон KOI8R.txt	UTF8	50 Мбайт
«Крупный» эталон в win-1251.html	UTF8	50 Мбайт
«Мелкий» эталон в cp-866.html	UTF8	100 Кбайт
«Средний» эталон в koi8-r.html	UTF8	10 Мбайт
«Средний» эталон в WIN_1251.md	UTF8	10 Мбайт
«Крупный» эталон в CP_866.md	UTF8	50 Мбайт
«Мелкий» эталон в KOI8_R.md	UTF8	100 Кбайт
Пустой файл.md	–	0 байт
Слишком большой файл.txt	–	52’428’801 байт
Картинка.jpg	–	~ 1 Мбайт
Картинка в виде TXT.txt	–	~ 1 Мбайт

И только что мы упомянули автоматизацию как способ ускорения выполнения тест-кейсов. В данном случае мы можем обойтись самыми тривиальными командными файлами. В приложении «Командные файлы для Windows и Linux, автоматизирующие выполнение дымового тестирования» приведены скрипты, полностью автоматизирующие выполнение всего уровня дымового тестирования (см. подраздел 3.2 «Подробная классификация тестирования») над представленным выше набором из 22 файлов.

Задание 4.f: Доработайте представленные в приложении командные файлы так, чтобы их выполнение заодно проверяло работу тестируемого приложения с пробелами, кириллическими символами и спецсимволами в путях к входному каталогу, выходному каталогу и файлу журнала. Оптимизируйте получившиеся командные файлы так, чтобы избежать многократного дублирования кода.

Если снова вернуться к чек-листу, то оказывается, что мы уже подготовили проверки для всего уровня дымового тестирования и части уровня тестирования критического пути (см. подраздел 3.2 «Подробная классификация тестирования»).

Продолжим оптимизацию. Большинство проверок не представляет особой сложности, и мы разберёмся с ними по ходу дела, но есть в чек-листе пункт, вызывающий особую тревогу, – производительность.

Тестирование и оптимизация производительности – это отдельный вид тестирования со своими достаточно непростыми правилами и подходами, к тому же разделяющийся на несколько подвидов. Нужно ли оно нам в нашем приложении? Заказчик в АК-1.1 (см. подраздел 2.7 «Пример анализа и тестирования требований»)

определил минимальную производительность приложения как способность обрабатывать входные данные со скоростью не менее 5 Мбайт/с. Грубые эксперименты на указанном в АК-1.1 оборудовании показывают, что даже куда более сложные операции (например, архивирование видеофайла с максимальной степенью сжатия) выполняются быстрее (пусть и ненамного). Вывод? Вычёркиваем. Вероятность встретить здесь проблему ничтожно мала, а соответствующее тестирование требует ощутимых затрат сил и времени, а также наличия соответствующих специалистов.

Вернёмся к чек-листу:

- Конфигурирование и запуск:
  - ~~С верными параметрами:~~
    - ~~значения SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME указаны и содержат пробелы и кириллические символы (повторить для форматов путей в Windows и *nix файловых системах, обратить внимание на имена логических дисков и разделители имён каталогов (“/” и “\”))~~ (**уже учтено при автоматизации проверки работы приложения с 22 файлами**);
    - ~~Значение LOG_FILE_NAME не указано.~~ (**объединить с проверкой ведения самого файла журнала**);
  - без параметров;
  - с недостаточным количеством параметров;
  - с неверными параметрами:
    - недопустимый путь SOURCE_DIR;
    - недопустимый путь DESTINATION_DIR;
    - недопустимое имя LOG_FILE_NAME;
    - DESTINATION_DIR находится внутри SOURCE_DIR;
    - значения DESTINATION_DIR и SOURCE_DIR совпадают.
- Обработка файлов:
  - ~~разные форматы, кодировки и размеры~~ (**уже сделано**).
  - недоступные входные файлы:
    - нет прав доступа;
    - файл открыт и заблокирован;
    - файл с атрибутом «только для чтения».
- ~~остановка:~~
  - ~~закрытием окна консоли~~ (**вычёркиваем, т. к. не настолько важная проверка, а если и будут проблемы, технология PHP не позволит их решить**).
- журнал работы приложения:
  - автоматическое создание (при отсутствии журнала), имя журнала указано явно;
  - продолжение (дополнение журнала) при повторных запусках, имя журнала не указано.
- ~~производительность:~~
  - ~~элементарный тест с грубой оценкой~~ (**ранее решили, что наше приложение явно уложится в весьма демократичные требования заказчика**).

Перепишем компактно то, что у нас осталось от уровня тестирования критического пути (см. подраздел 3.2 «Подробная классификация тестирования»). Внимание! Это НЕ тест-кейс! Это всего лишь ещё одна форма записи чек-листа, более удобная на данном этапе (таблица 4.z).

Таблица 4.z – Чек-лист для уровня критического пути

Суть проверки	Ожидаемая реакция
Запуск без параметров	Отображение инструкции к использованию
Запуск с недостаточным количеством параметров	Отображение инструкции к использованию и указание имён недостающих параметров
Запуск с неверными значениями параметров: ○ недопустимый путь SOURCE_DIR; ○ недопустимый путь DESTINATION_DIR; ○ недопустимое имя LOG_FILE_NAME; ○ DESTINATION_DIR находится внутри SOURCE_DIR; ○ значения DESTINATION_DIR и SOURCE_DIR совпадают	Отображение инструкции к использованию и указание имени неверного параметра, значения неверного параметра и пояснения сути проблемы
Недоступные входные файлы: ○ нет прав доступа; ○ файл открыт и заблокирован; ○ файл с атрибутом «только для чтения»	Отображение сообщения в консоль и файл журнала, дальнейшее игнорирование недоступных файлов
Журнал работы приложения: ○ автоматическое создание (при отсутствии журнала), имя журнала указано явно; ○ продолжение (дополнение журнала) при повторных запусках, имя журнала не указано	Создание или продолжение ведения файла журнала по указанному или вычисленному пути

Наконец, у нас остался уровень расширенного тестирования. И сейчас мы сделаем то, что согласно всем классическим учебникам делать нельзя, – мы откажемся от всего этого набора проверок целиком.

- ~~Конфигурирование и запуск:~~
  - ~~значения SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME:~~
    - ~~в разных стилях (Windows-пути + *nix-пути) – одно в одном стиле, другое – в другом;~~
    - ~~с использованием UNC-имён;~~
    - ~~LOG_FILE_NAME внутри SOURCE_DIR;~~
    - ~~LOG_FILE_NAME внутри DESTINATION_DIR;~~
  - ~~размер LOG_FILE_NAME на момент запуска:~~
    - ~~2–4 Гбайт;~~
    - ~~4+ Гбайт;~~
  - ~~запуск двух и более копий приложения с:~~
    - ~~одинаковыми параметрами SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME;~~
    - ~~одинаковыми SOURCE_DIR и LOG_FILE_NAME, но разными DESTINATION_DIR;~~
    - ~~одинаковыми DESTINATION_DIR и LOG_FILE_NAME, но разными SOURCE_DIR.~~
- ~~Обработка файлов:~~
  - ~~файл верного формата, в котором текст представлен в двух и более поддерживаемых кодировках одновременно;~~
  - ~~размер входного файла:~~

- ~~2–4 Гбайт;~~
- ~~4+ Гбайт.~~

Да, таким образом мы действительно повысили риск пропустить какой-то дефект. Но вероятность возникновения дефекта мала в силу малой вероятности возникновения описанных в этих проверках ситуаций. При этом по самым скромным подсчетам мы на треть сократили общее количество проверок, которые нам нужно будет выполнять, а значит – освободили силы и время для более тщательной проработки типичных каждодневных сценариев использования (см. подраздел 4.6 «Наборы тест-кейсов»).

Весь оптимизированный чек-лист (он же – и черновик для плана выполнения проверок) теперь выглядит так:

1) подготовить файлы (см. таблицу 4.у);
2) пля «дымового теста» использовать командные файлы (см. приложение «Командные файлы для Windows и Linux, автоматизирующие выполнение дымового тестирования»);
3) для основных проверок использовать файлы из пункта 1 и следующие идеи, представленные в таблице 4.α.

Таблица 4.α – Основные проверки для приложения «Конвертер файлов»

Суть проверки	**Ожидаемая реакция**
Запуск без параметров	Отображение инструкции к использованию
Запуск с недостаточным количеством параметров	Отображение инструкции к использованию и указание имён недостающих параметров
Запуск с неверными значениями параметров: ○ недопустимый путь SOURCE_DIR; ○ недопустимый путь DESTINATION_DIR; ○ недопустимое имя LOG_FILE_NAME; ○ DESTINATION_DIR находится внутри SOURCE_DIR; ○ значения DESTINATION_DIR и SOURCE_DIR совпадают	Отображение инструкции к использованию и указание имени неверного параметра, значения неверного параметра и пояснения сути проблемы
Недоступные входные файлы: ○ нет прав доступа; ○ файл открыт и заблокирован; ○ файл с атрибутом «только для чтения»	Отображение сообщения в консоль и файл журнала, дальнейшее игнорирование недоступных файлов
Журнал работы приложения: ○ автоматическое создание (при отсутствии журнала), имя журнала указано явно ○ продолжение (дополнение журнала) при повторных запусках, имя журнала не указано	Создание или продолжение ведения файла журнала по указанному или вычисленному пути

4) в случае наличия времени использовать первоначальную редакцию чек-листа для уровня расширенного тестирования как основу для выполнения исследовательского тестирования.

И почти всё. Стоит ещё раз подчеркнуть, что представленная логика выбора проверок не претендует на то, чтобы считаться единственно верной, но она явно

позволяет сэкономить время и усилия, при этом практически не снизив качество тестирования наиболее востребованной заказчиком функциональности приложения.

Задание 4.g: Подумайте, какие проверки из таблицы 4.α можно автоматизировать с помощью командных файлов. Напишите такие командные файлы.

4.8 Типичные ошибки при разработке чек-листов, тест-кейсов и наборов тест-кейсов

Ошибки оформления и формулировок

Отсутствие заглавия тест-кейса или плохо написанное заглавие. В абсолютном большинстве систем управления тест-кейсами поле для заглавия вынесено отдельно и является обязательным к заполнению – тогда эта проблема отпадает. Если же инструментальное средство позволяет создать тест-кейс без заглавия, возникает риск получить N тест-кейсов, для понимания сути каждого из которых нужно прочитать десятки строк вместо одного предложения. Гарантированно, это напрасная трата рабочего времени и основательное снижение производительности.

Если заглавие тест-кейса приходится вписывать в поле с шагами и инструментальное средство допускает форматирование текста, заглавие стоит писать **жирным шрифтом**, чтобы его было легче отделять от основного текста.

Продолжением этой ошибки является создание одинаковых заглавий, по которым объективно невозможно отличить один тест-кейс от другого. Более того, возникает подозрение, что одинаково озаглавленные тест-кейсы и внутри одинаковы. Потому следует формулировать заглавия по-разному, при этом подчёркивая в них суть тест-кейса и его отличие от других, похожих тест-кейсов.

И, наконец, в заглавии недопустимы «мусорные» слова типа «проверка», «тест» и т. д. Ведь это заглавие тест-кейса, т. е. речь по определению идёт о проверке, и не надо этот факт подчёркивать дополнительно. Более подробное пояснение этой ошибки находится в пункте «Постоянное использование слов «проверить» (и ему подобных) в чек-листах».

Отсутствие нумерации шагов и/или ожидаемых результатов (даже если таков всего лишь один). Наличие этой ошибки превращает тест-кейс в «поток сознания», в котором нет структурированности, модифицируемости и прочих полезных свойств (да, многие свойства качественных требований (см. подраздел 2.5 «Свойства качественных требований») в полной мере применимы и к тест-кейсам), очень легко перепутать, что к чему относится. Даже выполнение такого тест-кейса усложняется, а доработка и вовсе тяжела.

Ссылка на множество требований. Иногда высокоуровневый тест-кейс (см. подраздел 4.2 «Тест-кейс и его жизненный цикл») действительно затрагивает несколько требований, но в таком случае рекомендуется писать ссылку максимум на 2–3 ключевые (наиболее соответствующих цели тест-кейса), а ещё лучше – указывать общий раздел этих требований (т. е. не ссылаться, например, на требования 5.7.1, 5.7.2, 5.7.3, 5.7.7, 5.7.9, 5.7.12, а сразу на подраздел 5.7, включающий в себя все перечисленные пункты). В большинстве инструментальных средств управления тест-кейсами это поле представляет собой выпадающий список, и там эта проблема теряет актуальность.

Использование личной формы глаголов. Если вы пишете требования на русском языке, то используйте неопределенную форму глаголов: пишите «нажать» вместо «нажмите», «ввести» вместо «введите», «перейти» вместо «перейдите» и т. д. В технической документации вообще не рекомендуется обращение к лицу, а также существует мнение, что личная форма глаголов подсознательно воспринимается как

«чужие бессмысленные команды» и приводит к повышенной утомляемости и раздражительности.

Использование прошедшего или будущего времени в ожидаемых результатах. Это не очень серьёзная ошибка, но всё равно «введённое значение отображается в поле» воспринимается лучше, чем «введённое значение отобразилось в поле» или «введённое значение отобразится в поле».

Постоянное использование слов «проверить» (и ему подобных) в чек-листах. Бывает, что почти каждый пункт чек-листа начинается с «проверить …». Но ведь весь чек-лист – это и есть список проверок! Так зачем писать это слово? Сравните (таблица 4.β).

Таблица 4.β – Сравнение названий проверок в чек-листе

Плохо	Хорошо
Проверить запуск приложения.	Запуск приложения.
Проверить открытие корректного файла.	Открытие корректного файла.
Проверить модификацию файла.	Модификация файла.
Проверить сохранение файла.	Сохранение файла.
Проверить закрытие приложения	Закрытие приложения

Подобной ошибкой является типичное слово «попытаться» из негативных тест-кейсов («попытаться поделить на ноль», «попытаться открыть несуществующий файл», «попытаться ввести недопустимые символы»): «деление на нуль», «открытие несуществующего файла», «ввод спецсимволов» намного информативнее. А реакцию приложения, если она не очевидна, можно указать в скобках (так будет даже содержательнее): «деление на нуль» (сообщение «Division by zero detected»), «открытие несуществующего файла» (приводит к автоматическому созданию файла), «ввод спецсимволов» (символы не вводятся, отображается подсказка).

Описание стандартных элементов интерфейса вместо использования их устоявшихся названий. «Маленький крестик справа вверху окна приложения» – это системная кнопка «Закрыть» (system button «Close»), «быстро-быстро дважды щёлкнуть левой клавишей мыши» – двойной щелчок (double click), «маленькое окошечко с надписью появляется, когда наводишь мышь» – всплывающая подсказка (hint).

Пунктуационные, орфографические, синтаксические и им подобные ошибки. Без комментариев.

Логические ошибки

Ссылка на другие тест-кейсы или шаги других тест-кейсов. За исключением случаев написания строго оговорённого, явно обозначенного набора последовательных тест-кейсов это запрещено делать. В лучшем случае вам повезёт, и тест-кейс, на который вы ссылались, будет просто удалён. Повезёт потому, что это будет сразу заметно. Не повезёт в случае, если тест-кейс, на который вы ссылаетесь, будет изменён – ссылка по-прежнему ведёт в некое существующее место, но описано там уже совершенно не то, что было в момент создания ссылки.

Детализация, не соответствующая уровню функционального тестирования. Например, не нужно на уровне дымового тестирования проверять работоспособность каждой отдельной кнопки или прописывать некий крайне сложный, нетривиальный и редкий сценарий – поведение кнопок и без явного указания будет проверено множеством тест-кейсов, объективно задействующих эти кнопки, а сложному сценарию место на уровне тестирования критического пути или даже на уровне расширенного тестирования (в которых, напротив, недостатком можно считать излишнее обобщение без должной детализации).

Расплывчатые двусмысленные описания действий и ожидаемых результатов. Помните, что тест-кейс с высокой вероятностью будете выполнять не вы (автор тест-кейса), а другой сотрудник, и он – не телепат. Попробуйте догадаться по этим примерам, что имел в виду автор:

- «Установить приложение на диск С». (То есть в «С:\»? Прямо в корень? Или как?)
- «Нажать на иконку приложения». (Например, если у меня есть ico-файл с иконкой приложения, и я по нему кликну – это оно? Или нет?)
- «Окно приложения запустится». (Куда?)
- «Работает верно». (Ого! А верно – это, простите, как?)
- «ОК». (И? Что «ОК»?)
- «Количество найденных файлов совпадает». (С чем?)
- «Приложение отказывается выполнять команду». (Что значит «отказывается»? Как это выглядит? Что должно происходить?)

Описание действий в качестве наименований модуля/подмодуля. Например, «запуск приложения» – это НЕ модуль или подмодуль. Модуль или подмодуль – это всегда некоторые части приложения (например, главная страница/авторизация), а не его поведение. Сравните: «дыхательная система» – это модуль человека, но «дыхание» – нет.

Описание событий или процессов в качестве шагов или ожидаемых результатов. Например, в качестве шага сказано: «Ввод спецсимволов в поле X». Это было бы приемлимым заглавием тест-кейса, но не годится в качестве шага, который должен быть сформулирован как «Ввести спецсимволы (перечень) в поле X».

Еще хуже, если подобное встречается в ожидаемых результатах. Например, там написано: «Отображение скорости чтения в панели X». И что? Оно должно начаться, продолжиться, завершиться, не начинаться, каким-то образом измениться (например, измениться должна размерность данных), как-то на что-то повлиять? Тест-кейс становится полностью бессмысленным, т. к. такой ожидаемый результат невозможно сравнить с фактическим поведением приложения.

«Выдумывание» особенностей поведения приложения. Да, часто в требованиях отсутствуют очевидные (без кавычек, они на самом деле очевидные) вещи, но нередко встречаются и некачественные (например, неполные) требования, которые нужно улучшать, а не «телепатически компенсировать».

Например, в требованиях сказано, что «приложение должно отображать диалоговое окно сохранения с указанным по умолчанию каталогом». Если из контекста (соседних требований, иных документов) ничего не удаётся узнать об этом таинственном «каталоге по умолчанию», нужно задать вопрос. Нельзя просто записать в ожидаемых результатах: «Отображается диалоговое окно сохранения с указанным по умолчанию каталогом». (Как мы проверим, что выбран именно указанный по умолчанию каталог, а не какой-то иной?) И уж тем более нельзя в ожидаемых результатах писать: «Отображается диалоговое окно сохранения с выбранным каталогом “C:/SavedDocuments”». (Откуда взялось это «C:/SavedDocuments», не ясно, т. е. оно явно предположено и, скорее всего, предполагается неправильно).

Отсутствие описания приготовления к выполнению тест-кейса. Часто для корректного выполнения тест-кейса необходимо каким-то особым образом настроить окружение. Предположим, что мы проверяем приложение, выполняющее резервное копирование файлов. Если тест выглядит примерно так, то выполняющий его сотрудник будет в замешательстве, т. к. ожидаемый результат абсолютно непонятен. Откуда взялось «~200»? Что это такое? Приведем пример тест-кейса, требущего пояснений посредством описания приготовления к его выполнению, в таблице 4.γ.

Таблица 4.γ – Пример тест-кейса, требующего пояснений посредством описания приготовления к его выполнению

Шаги выполнения	Ожидаемые результаты
1. Нажать на панели «Главная» кнопку «Быстрая дедубликация». 2. Выбрать каталог «C:/MyData»	1. Кнопка «Быстрая дедубликация» переходит в утопленное состояние и меняет цвет с серого на зелёный. 2. На панели «Состояние» в поле «Дубликаты» отображается «~200»

И совсем иначе этот тест-кейс воспринимался бы, если бы в приготовлениях было сказано: «Создать каталог “C:/MyData” с произвольным набором подкаталогов (глубина вложенности не менее пяти). В полученном дереве каталогов разместить 1000 файлов, из которых 200 имеют одинаковые имена и размеры, но НЕ внутреннее содержимое».

Полное дублирование (копирование) одного и того же тест-кейса на уровнях дымового тестирования, тестирования критического пути, расширенного тестирования. Многие идеи естественным образом развиваются от уровня к уровню, но они должны именно развиваться, а не дублироваться (таблица 4.δ). Сравните.

Таблица 4.δ – Пример описания одного и того же тест-кейса на уровнях дымового тестирования, тестирования критического пути, расширенного тестирования

Результат	Дымовое тестирование	Тестирование критического пути	Расширенное тестирование
Плохо	Запуск приложения	Запуск приложения	Запуск приложения
Хорошо	Запуск приложения	Запуск приложения из командной строки. Запуск приложения через ярлык на рабочем столе. Запуск приложения через меню «Пуск»	Запуск приложения из командной строки в активном режиме. Запуск приложения из командной строки в фоновом режиме. Запуск приложения через ярлык на рабочем столе от имени администратора. Запуск приложения через меню «Пуск» из списка часто запускаемых приложений

Слишком длинный перечень шагов, не относящихся к сути (цели) тест-кейса. Например, мы хотим проверить корректность одностороннего режима печати из нашего приложения на дуплексном принтере (таблица 4.ε). Сравните.

Таблица 4.ε – Пример описания шагов тест-кейса

Плохо	Хорошо
Односторонняя печать 1. Запустить приложение. 2. В меню выбрать «Файл» -> «Открыть».	**Односторонняя печать** 1. Открыть любой DOCX-файл, содержащий три и более непустые страницы.

Продолжение таблицы 4.ε

Плохо	Хорошо
3. Выбрать любой DOCX-файл, состоящий из нескольких страниц. 4. Нажать кнопку «Открыть». 5. В меню выбрать «Файл» -> «Печать». 6. В списке «Двусторонняя печать» выбрать пункт «Нет». 7. Нажать кнопку «Печать». 8. Закрыть файл. Закрыть приложение	2. В диалоговом окне «Настройка печати» в списке «Двусторонняя печать» выбрать «Нет». 3. Произвести печать документа на принтере, поддерживающем двустороннюю печать

В левой колонке мы видим огромное количество действий, не относящихся непосредственно к тому, что проверяет тест-кейс. Тем более, что запуск и закрытие приложения, открытие файла, работа меню и прочее или будут покрыты другими тест-кейсами (со своими соответствующими целями), или на самом деле являются очевидными (логично ведь, что нельзя открыть приложением файл, если приложение не запущено) и не нуждаются в описании шагов, которые только создают информационный шум и занимают время на написание и прочтение.

Некорректное наименование элементов интерфейса или их свойств. Иногда из контекста понятно, что автор тест-кейса имел в виду, но чаще это становится истинной проблемой. Например, мы видим тест-кейс с заглавием «Закрытие приложения кнопками "Close" и "Close window"». Уже тут возникает недоумение по поводу того, в чём же различие этих кнопок, да и о чём вообще идёт речь. Ниже (в шагах тест-кейса) автор поясняет: «В рабочей панели внизу экрана нажать "Close window"». Ага! Ясно. Но «Close window» – это НЕ кнопка, это пункт системного контекстного меню приложения в панели задач.

Ещё один отличный пример: «Окно приложения свернётся в окно меньшего диаметра». Хм. Окно круглое? Или должно стать круглым? А, может, тут и вовсе речь про два разных окна, и одно должно будет оказаться внутри второго? Или всё же «размер окна уменьшается» (кстати, насколько?), а его геометрическая форма остаётся прямоугольной?

И, наконец, пример, благодаря которому можно написать отчёт о дефекте на вполне корректно работающее приложение: «В системном меню выбрать "Фиксация расположения"». Казалось бы, что ж тут плохого? Но потом выясняется, что имелось в виду главное меню приложения, а вовсе не системное меню.

Непонимание принципов работы приложения и вызванная этим некорректность тест-кейсов. Классикой жанра является закрытие приложения: тот факт, что окно приложения «исчезло» (например, оно свернулось в область уведомления панели задач (system tray, taskbar notification area), что является сюрпризом), или приложение отключило интерфейс пользователя, продолжив функционировать в фоновом режиме, вовсе не является признаком того, что оно завершило работу.

Проверка типичной «системной» функциональности. Если только ваше приложение не написано с использованием каких-то особенных библиотек и технологий и не реализует какое-то нетипичное поведение, нет необходимости проверять системные кнопки, системные меню, сворачивание-разворачивание окна и т. д. Вероятность встретить здесь ошибку близка к нулю. Если всё же очень хочется, можно вписать эти проверки как уточнения некоторых действий на уровне тестирования критического пути, но создавать для них отдельные тест-кейсы не нужно.

Неверное поведение приложения как ожидаемый результат. Такое не допускается по определению. Не может быть тест-кейса с шагом в стиле «поделить на ноль» с ожидаемым результатом «крах приложения с потерей пользовательских данных». Ожидаемые результаты всегда описывают правильное поведение приложения, даже в самых страшных стрессовых тест-кейсах.

Общая некорректность тест-кейсов. Может быть вызвана множеством причин и выражаться множеством образов, но вот классический пример некорректного описания шагов тест-кейса (таблица 4.ζ).

Таблица 4.ζ – Пример некорректного (неполного) описания шагов тест-кейса

Шаги выполнения	Ожидаемые результаты
… 4. Закрыть приложение нажатием Alt+F4. 5. Выбрать в меню «Текущее состояние»	… 4. Приложение завершает работу. 5. Отображается окно с заголовком «Текущее состояние» и содержимым, соответствующим рисунку 999.99

Здесь или не указано, что вызов окна «Текущее состояние» происходит где-то в другом приложении, или остаётся загадкой, как вызвать это окно у завершившего работу приложения. Запустить заново? Возможно, но в тест-кейсе это не сказано.

Неверное разбиение наборов данных на классы эквивалентности. Действительно, иногда классы эквивалентности могут быть очень неочевидными. Но ошибки встречаются и в довольно простых случаях. Допустим, в требованиях сказано, что размер некоторого файла может быть 10–100 Кбайт (включительно). Разбиение по размеру 0–9 Кбайт, 10–100 Кбайт, 101+ Кбайт **ошибочно**, т. к. килобайт не является неделимой единицей. Такое ошибочное разбиение не учитывает, например, размеры в 9.5 Кбайт, 100.1 Кбайт, 100.7 Кбайт и т. д. Потому здесь стоит применять неравенства: 0 Кбайт ≤ размер < 10 Кбайт, 10 Кбайт ≤ размер ≤ 100 Кбайт, 100 Кбайт < размер. Также можно писать с использованием синтаксиса скобок: [0, 10) Кбайт, [10, 100] Кбайт, (100, ∞) Кбайт, но вариант с неравенствами более привычен большинству людей.

Тест-кейсы, не относящиеся к тестируемому приложению. Например, нам нужно протестировать фотогалерею на сайте. Соответственно, следующие тест-кейсы никак не относятся к фотогалерее (они тестируют браузер, операционную систему пользователя, файловый менеджер и т. д., но НЕ наше приложение, ни его серверную, ни даже клиентскую часть):

- файл с сетевого диска;
- файл со съёмного носителя;
- файл, заблокированный другим приложением;
- файл, открытый другим приложением;
- файл, к которому у пользователя нет прав доступа;
- вручную указать путь к файлу;
- файл из глубоко расположенной поддиректории.

Формальные и/или субъективные проверки. Чаще всего данную ошибку можно встретить в пунктах чек-листа. Возможно, у автора был чёткий и подробный план, но из примеров, приведенных ниже, совершенно невозможно понять, что будет сделано с приложением и какой результат мы должны ожидать:

- «Сконвертировать»;
- «Проверить метод getMessage()»;
- «Некорректная работа в корректных условиях»;

- «Скорость»;
- «Объём данных»;
- «Должно работать быстро».

В отдельных исключительных ситуациях можно возразить, что из контекста и дальнейшей детализации становится понятно, что имелось в виду. Но чаще всего никакого контекста и никакой дальнейшей детализации нет, т. е. приведённые примеры оформлены как отдельные полноправные пункты чек-листа. Так нельзя. Как можно и нужно, разберите по примеру чек-листа в разделе (см. раздел 4.1 «Чек-лист»).

Теперь для лучшего закрепления рекомендуется заново прочитать про оформление атрибутов тест-кейсов, свойства качественных тест-кейсов и логику построения качественных тест-кейсов и качественных наборов тест-кейсов, описанные выше.

4.9 Контрольные вопросы и задания

- Что такое чек-лист?
- Какими свойствами должен обладать качественный чек-лист?
- Что такое тест-кейс?
- Что такое низкоуровневый тест-кейс?
- Что такое высокоуровневый тест-кейс?
- Зачем нужны тест-кейсы?
- Назовите основные состояния тест-кейса в процессе его жизненного цикла.
- Перечислите атрибуты (поля) тест-кейса.
- Дайте характеристику такому атрибуту тест-кейса, как идентификатор.
- Дайте характеристику такому атрибуту тест-кейсасвязанное с тест-кейсом требование.
- Дайте характеристику такому атрибуту тест-кейса, как модуль и подмодуль приложения.
- Дайте характеристику такому атрибуту тест-кейса, как заглавие.
- Каким условиям должно удовлетворять качественное заглавие тест-кейса?
- Дайте характеристику такому атрибуту тест-кейса, как исходные данные, необходимые для выполнения тест-кейса.
- Дайте характеристику такому атрибуту тест-кейса, как шаги тест-кейса.
- Дайте характеристику такому атрибуту тест-кейса, как ожидаемые результаты.
- Какими должны быть качественные ожидаемые результаты?
- Приведите примеры инструментальных средств управления тестированием.
- Перечислите свойства качественных тест-кейсов.
- Что представляют собой наборы тест-кейсов?
- Перечислите принципы построения наборов тест-кейсов.
- Верно ли утверждение, что все ошибки найти невозможно?
- Перечислите типичные ошибки при разработке чек-листов, тест-кейсов и наборов тест-кейсов.

5 ОТЧЁТЫ О ДЕФЕКТАХ

5.1 Ошибки, дефекты, сбои, отказы

Упрощённый взгляд на понятие дефекта

В этом разделе мы будем изучать терминологию (она действительно важна!), а потому начнём с очень простого: упрощённо, дефектом можно считать любое расхождение ожидаемого (свойства, результата, поведения и т. д., которое мы ожидали увидеть) и фактического (свойства, результата, поведения и т. д., которое мы на самом деле увидели). При обнаружении дефекта создаётся отчёт о дефекте.

!!!	**Дефект** – расхождение ожидаемого и фактического результатов. **Ожидаемый результат** – поведение системы, описанное в требованиях. **Фактический результат** – поведение системы, наблюдаемое в процессе тестирования.

	ВАЖНО! Эти три определения приведены в предельно упрощённой (и даже искажённой) форме с целью первичного ознакомления. Полные формулировки см. далее в данном разделе.

Поскольку трактовка в узком смысле не покрывает все возможные формы проявления проблем с программными продуктами, мы сразу же переходим к более подробному рассмотрению соответствующей терминологии.

Попробуем широко взглянуть на терминологию, описывающую проблемы. Разберёмся с широким спектром синонимов, которыми обозначают проблемы с программными продуктами и иными артефактами и процессами, сопутствующими их разработке.

В силлабусе ISTQB написано[290], что человек совершает ошибки, которые приводят к возникновению дефектов в коде, которые, в свою очередь, приводят к сбоям и отказам приложения (однако сбои и отказы могут возникать и из-за внешних условий, таких как электромагнитное воздействие на оборудование и т. д.). Таким образом, упрощённо можно изобразить схему, показанную на рисунке 5.a.

Рисунок 5.a – Ошибки, дефекты, сбои и отказы

Если же посмотреть на англоязычную терминологию, представленную в глоссарии ISTQB и иных источниках, можно построить чуть более сложную схему, отражающую взаимосвязь проблем в разработке программных продуктов (рисунок 5.b).

[290] A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so. Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions. Failures can be caused by environmental conditions as well. For example, radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing the hardware conditions. ISTQB Syllabus.

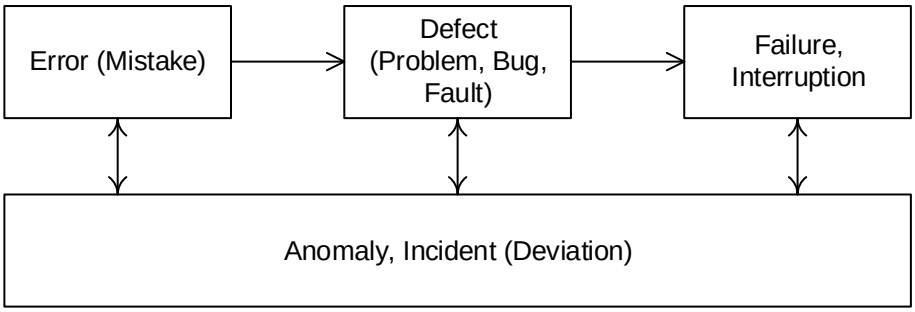


Рисунок 5.b – Взаимосвязь проблем в разработке программных продуктов

Рассмотрим все соответствующие термины.

!!! **Ошибка** (error[291], mistake) – действие человека, приводящее к некорректным результатам.

Этот термин очень часто используют как наиболее универсальный, описывающий любые проблемы («ошибка человека», «ошибка в коде», «ошибка в документации», «ошибка выполнения операции», «ошибка передачи данных», «ошибочный результат» и т. п.) Более того, намного чаще вы сможете услышать выражение «отчёт об ошибке», чем «отчёт о дефекте». И это нормально, так сложилось исторически, к тому же термин «ошибка» на самом деле имеет очень широкий смысл.

!!! **Дефект** (defect[292], bug, problem, fault) – недостаток в компоненте или системе, способный привести к ситуации сбоя или отказа.

Этот термин также приводят в широком смысле, говоря о дефектах в документации, настройках, входных данных и т. д. Почему раздел называется именно «отчёты о дефектах»? Потому что этот термин находится в центре. Бессмысленно писать отчёты о человеческих ошибках, равно как и почти бесполезно просто описывать проявления сбоев и отказов – нужно «докопаться» до их причины, и первым шагом в этом направлении является именно описание дефекта.

!!! **Сбой** (interruption[293]) или **отказ** (failure[294]) – отклонение поведения системы от ожидаемого.
В ГОСТ 27.002-89 даны хорошие и краткие определения сбоя и отказа:
Сбой – самоустраняющийся отказ или однократный отказ, устраняемый незначительным вмешательством оператора.
Отказ – событие, заключающееся в нарушении работоспособного состояния объекта.

[291] **Error, Mistake.** A human action that produces an incorrect result. ISTQB Glossary.
[292] **Defect, Bug, Problem, Fault.** A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system. ISTQB Glossary.
[293] **Interruption.** A suspension of a process, such as the execution of a computer program, caused by an event external to that process and performed in such a way that the process can be resumed. URL: http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=714-22-10.
[294] **Failure.** Deviation of the component or system from its expected delivery, service or result. ISTQB Glossary.

Эти термины скорее относятся к теории надёжности и нечасто встречаются в повседневной работе тестировщика, но именно сбои и отказы являются тем, что тестировщик замечает в процессе тестирования (и отталкиваясь от чего, проводит исследование с целью выявить дефект и его причины).

Аномалия (anomaly[295]) или **инцидент** (incident[296], deviation[84]) – любое отклонение наблюдаемого (фактического) состояния, поведения, значения, результата, свойства от ожиданий наблюдателя, сформированных на основе требований, спецификаций, иной документации или опыта и здравого смысла.

Ошибки, дефекты, сбои, отказы и т. д. являются проявлением аномалий – отклонений фактического результата от ожидаемого. Стоит отметить, что ожидаемый результат действительно может основываться на опыте и здравом смысле, т. к. поведение программного средства никогда не специфицируют до уровня базовых элементарных приёмов работы с компьютером.

Теперь, чтобы окончательно избавиться от путаницы и двусмысленности, договоримся, что именно мы будем считать дефектом в контексте данной книги.

Дефект – отклонение (deviation[296]) фактического результата (actual result[297]) от ожиданий наблюдателя (expected result[298]), сформированных на основе требований, спецификаций, иной документации или опыта и здравого смысла.

Отсюда логически вытекает, что дефекты могут встречаться не только в коде приложения, но и в любой документации, архитектуре и дизайне, настройках – в любой сфере.

Важно понимать, что приведённое определение дефекта позволяет лишь поднять вопрос о том, является ли некое поведение приложения дефектом. В случае если из проектной документации не следует однозначного положительного ответа, обязательно стоит обсудить свои выводы с коллегами и добиться донесения поднятого вопроса до заказчика, если его мнение по обсуждаемому «кандидату в баги» неизвестно.

Хорошее представление о едва-едва затронутой нами теме теории надёжности можно получить, прочитав книгу Рудольфа Стапелберга «Руководство по надёжности, доступности, ремонтопригодности и безопасности в инженерном проектировании» (Stapelberg R. F. «Handbook of Reliability, Availability, Maintainability and Safety in Engineering Design»).

[295] **Anomaly.** Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. See also bug, defect, deviation, error, fault, failure, incident, problem. ISTQB Glossary.

[296] **Incident, Deviation.** Any event occurring that requires investigation. ISTQB Glossary.

[297] **Actual result.** The behavior produced/observed when a component or system is tested. ISTQB Glossary.

[298] **Expected result, Expected outcome, Predicted outcome.** The behavior predicted by the specification, or another source, of the component or system under specified conditions. ISTQB Glossary.

С краткой, но достаточно подробной классификацией аномалий в программных продуктах можно ознакомиться в стандарте «IEEE 1044:2009 Standard Classification For Software Anomalies».

5.2 Отчёт о дефекте и его жизненный цикл

Как было сказано в предыдущем разделе, при обнаружении дефекта тестировщик создаёт отчёт о дефекте.

Отчёт о дефекте (defect report[299]) – документ, описывающий и приоритезирующий обнаруженный дефект, а также содействующий его устранению.

Как следует из самого определения, отчёт о дефекте пишется со следующими основными целями:

- Предоставить информацию о проблеме – уведомить проектную команду и иных заинтересованных лиц о наличии проблемы, описать суть проблемы.
- Приоритезировать проблему – определить степень опасности проблемы для проекта и степень срочности её устранения.
- Содействовать устранению проблемы – качественный отчёт о дефекте не только предоставляет все необходимые подробности для понимания сути случившегося, но также может содержать анализ причин возникновения проблемы и рекомендации по исправлению ситуации.

На последней цели следует остановиться подробнее. Есть мнение, что «хорошо написанный отчёт о дефекте – половина решения проблемы для программиста». И действительно, как мы увидим далее, от полноты, корректности, аккуратности, подробности и логичности отчёта о дефекте зависит очень многое – одна и та же проблема может быть описана так, что программисту останется буквально исправить пару строк кода, а может быть описана и так, что сам автор отчёта на следующий день не сможет понять, что же он имел в виду.

ВАЖНО! «Сверхцель» написания отчёта о дефекте состоит в быстром исправлении ошибки (а в идеале – и в недопущении её возникновения в будущем). Поэтому качеству отчётов о дефекте следует уделять особое, повышенное внимание.

Отчёт о дефекте (и сам дефект вместе с ним) проходит определённые стадии жизненного цикла, которые схематично можно показать так (рисунок 5.c):

- Обнаружен (submitted) – начальное состояние отчёта (иногда называется «Новый» (new)), в котором он находится сразу после создания. Некоторые средства также позволяют сначала создавать черновик (draft) и лишь потом публиковать отчёт.
- Назначен (assigned) – в это состояние отчёт переходит с момента, когда кто-то из проектной команды назначается ответственным за исправление дефекта. Назначение ответственного производится или решением лидера команды разработки, или коллегиально, или по добровольному принципу, или иным принятым в команде способом или выполняется автоматически на основе определённых правил.

[299] **Defect report, Bug report.** A document reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function. ISTQB Glossary.

- Исправлен (fixed) – в это состояние отчёт переводит ответственный за исправление дефекта член команды после выполнения соответствующих действий по исправлению.
- Проверен (verified) – в это состояние отчёт переводит тестировщик, удостоверившийся, что дефект на самом деле был устранён. Как правило, такую проверку выполняет тестировщик, изначально написавший отчёт о дефекте.

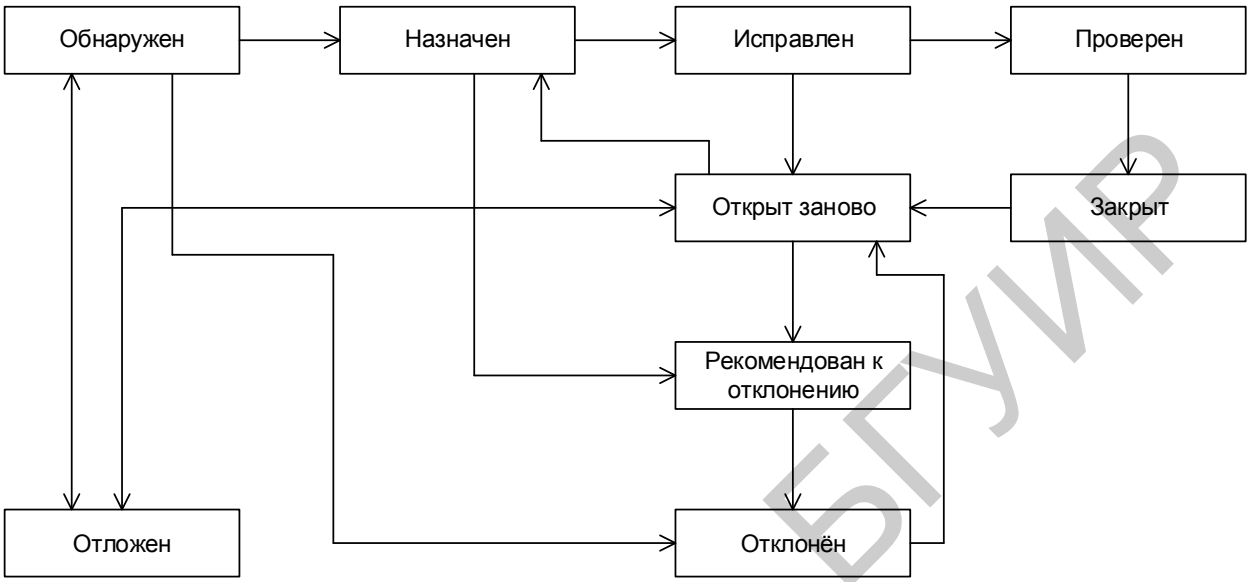


Рисунок 5.c – Жизненный цикл отчёта о дефекте с наиболее типичными переходами между состояниями

Набор стадий жизненного цикла, их наименование и принцип перехода от стадии к стадии может различаться в разных инструментальных средствах управления жизненным циклом отчётов о дефектах. Более того многие такие средства позволяют гибко настраивать эти параметры. На рисунке 5.c показан лишь общий принцип.

- Закрыт (closed) – состояние отчёта, означающее, что по данному дефекту не планируется никаких дальнейших действий. Здесь есть некоторые расхождения в жизненном цикле, принятом в разных инструментальных средствах управления отчётами о дефектах:
  - в некоторых средствах существуют оба состояния – «Проверен» и «Закрыт», чтобы подчеркнуть, что в состоянии «Проверен» ещё могут потребоваться какие-то дополнительные действия (обсуждения, дополнительные проверки в новых билдах и т. д.), в то время как состояние «Закрыт» означает «с дефектом покончено, больше к этому вопросу не возвращаемся»;
  - в некоторых средствах одного из состояний нет (оно поглощается другим);
  - в некоторых средствах в состояние «Закрыт» или «Отклонён» отчёт о дефекте может быть переведён из множества предшествующих состояний с резолюциями наподобие:
    - «Не является дефектом» – приложение так и должно работать, описанное поведение не является аномальным;
    - «Дубликат» – данный дефект уже описан в другом отчёте;

- «Не удалось воспроизвести» – разработчикам не удалось воспроизвести проблему на своём оборудовании;
- «Не будет исправлено» – дефект есть, но по каким-то серьёзным причинам его решено не исправлять;
- «Невозможно исправить» – непреодолимая причина дефекта находится вне области полномочий команды разработчиков, например существует проблема в операционной системе или аппаратном обеспечении, влияние которой устранить разумными способами невозможно;

Как было только что подчёркнуто, в некоторых средствах отчёт о дефекте в подобных случаях будет переведён в состояние «Закрыт», в некоторых – в состояние «Отклонён», в некоторых – часть случаев закреплена за состоянием «Закрыт», часть – за «Отклонён».

- Открыт заново (reopened) – в это состояние (как правило, из состояния «Исправлен») отчёт переводит тестировщик, удостоверившийся, что дефект по-прежнему воспроизводится на билде, в котором он уже должен быть исправлен.
- Рекомендован к отклонению (to be declined) – в это состояние отчёт о дефекте может быть переведён из множества других состояний с целью вынести на рассмотрение вопрос об отклонении отчёта по той или иной причине. Если рекомендация является обоснованной, отчёт переводится в состояние «Отклонён» (см. следующий пункт).
- Отклонён (declined) – в это состояние отчёт переводится в случаях, подробно описанных в пункте «Закрыт», если средство управления отчётами о дефектах предполагает использование этого состояния вместо состояния «Закрыт» для тех или иных резолюций по отчёту.
- Отложен (deferred) – в это состояние отчёт переводится в случае, если исправление дефекта в ближайшее время является нерациональным или не представляется возможным, однако есть основания полагать, что в обозримом будущем ситуация исправится (выйдет новая версия библиотеки, вернётся из отпуска специалист по какой-то конкретной технологии, изменятся требования заказчика и т. д.)

Для полноты рассмотрения данной подтемы приведён пример жизненного цикла, принятого по умолчанию в инструментальном средстве управления отчётами о дефектах JIRA[300] (рисунок 5.d).

[300] What is Workflow. URL: https://confluence.atlassian.com/display/JIRA/What+is+Workflow.

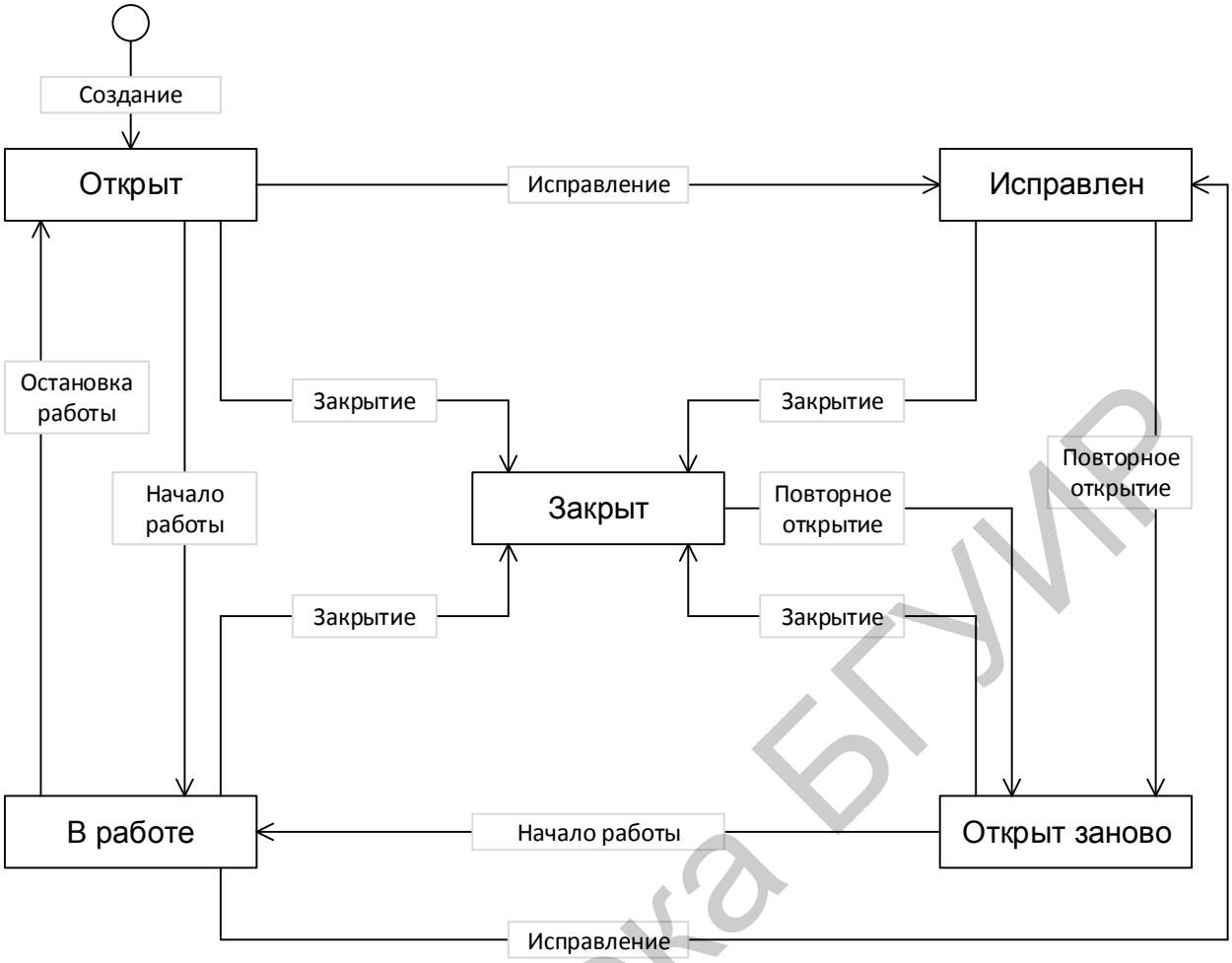


Рисунок 5.d – Жизненный цикл отчёта о дефекте в JIRA

5.3 Атрибуты (поля) отчёта о дефекте

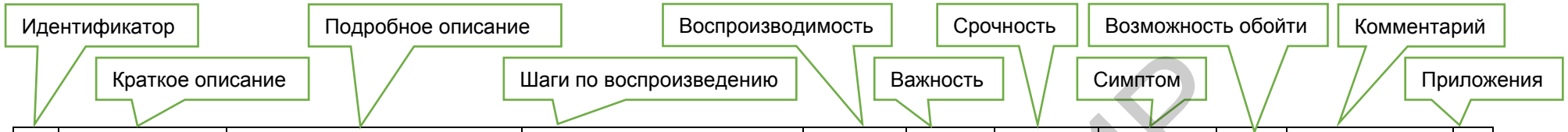
В зависимости от инструментального средства управления отчётами о дефектах внешний вид их записи может немного отличаться, могут быть добавлены или убраны отдельные поля, но концепция остаётся неизменной.

Общий вид всей структуры отчёта о дефекте представлен на рисунке 5.e.

Задание 5.a: Как вы думаете, почему этот отчёт о дефекте можно по формальным признакам отклонить с резолюцией «не является дефектом»?

Теперь рассмотрим каждый атрибут подробно.

Идентификатор (identifier) представляет собой уникальное значение, позволяющее однозначно отличить один отчёт о дефекте от другого и используемое во всевозможных ссылках. В общем случае идентификатор отчёта о дефекте может представлять собой просто уникальный номер, но (если позволяет инструментальное средство управления отчётами) может быть и куда сложнее: включать префиксы, суффиксы и иные осмысленные компоненты, позволяющие быстро определить суть дефекта и часть приложения (или требований), к которой он относится.



19	Бесконечный цикл обработки входного файла с атрибутом «только для чтения»	Если у входного файла выставлен атрибут «только для чтения», после обработки приложению не удаётся переместить его в каталог-приёмник: создаётся копия файла, но оригинал не удаляется, и приложение снова и снова обрабатывает этот файл и безуспешно пытается переместить его в каталог-приёмник. **Ожидаемый результат:** после обработки файл перемещён из каталога-источника в каталог-приёмник. **Фактический результат:** обработанный файл копируется в каталог-приёмник, но его оригинал остаётся в каталоге-источнике. **Требование:** ДС-2.1	1. Поместить в каталог-источник файл допустимого типа и размера. 2. Установить данному файлу атрибут «только для чтения». 3. Запустить приложение. Дефект: обработанный файл появляется в каталоге-приёмнике, но не удаляется из каталога-источника, файл в каталоге-приёмнике непрерывно обновляется (видно по значению времени последнего изменения)	Всегда	Средняя	Обычная	Некорректная операция	Нет	Если заказчик не планирует использовать установку атрибута «только для чтения» файлам в каталоге-источнике для достижения неких своих целей, можно просто снимать этот атрибут и спокойно перемещать файл	–

Рисунок 5.е – Общий вид отчёта о дефекте

Краткое описание (summary) должно в предельно лаконичной форме давать исчерпывающий ответ на вопросы «Что произошло?» «Где это произошло»? «При каких условиях это произошло?». Например, «Отсутствует логотип на странице приветствия, если пользователь является администратором»:

- Что произошло? Отсутствует логотип.
- Где это произошло? На странице приветствия.
- При каких условиях это произошло? Если пользователь является администратором.

Одной из самых больших проблем для начинающих тестировщиков является именно заполнение поля «краткое описание», которое одновременно должно:

- Содержать предельно краткую, но в то же время доступную для понимания сути проблемы информацию о дефекте.
- Отвечать на только что упомянутые вопросы («что, где и при каких условиях случилось») или как минимум на те несколько вопросов, которые применимы к конкретной ситуации.
- Быть достаточно коротким, чтобы полностью помещаться на экране (в тех системах управления отчётами о дефектах, где конец этого поля обрезается или приводит к появлению скроллинга).
- При необходимости содержать информацию об окружении, под которым был обнаружен дефект.
- Являться законченным предложением на русском или английском (или ином) языке, построенным по соответствующим правилам грамматики.

Для создания хороших кратких описаний дефектов рекомендуется пользоваться следующим алгоритмом:

1. Полноценно понять суть проблемы. До тех пор пока у тестировщика нет чёткого понимания того, «что сломалось», писать отчёт о дефекте едва ли стоит.
2. Сформулировать подробное описание (description) дефекта – сначала без оглядки на длину получившегося текста.
3. Убрать из получившегося подробного описания всё лишнее, уточнить важные детали.
4. Выделить в подробном описании слова (словосочетания, фрагменты фраз), отвечающие на вопросы, «что, где и при каких условиях случилось».
5. Оформить результат, получившийся в пункте 4, в виде законченного грамматически правильного предложения.
6. Если предложение получилось слишком длинным, переформулировать его, сократив длину (за счёт подбора синонимов, использования общепринятых аббревиатур и сокращений). К слову, в английском языке предложение почти всегда будет короче русского аналога.

Рассмотрим несколько примеров применения этого алгоритма.

Ситуация 1. Тестированию подвергается некое веб-приложение, поле описания товара должно допускать ввод максимально 250 символов; в процессе тестирования оказалось, что этого ограничения нет.

1. Суть проблемы: исследование показало, что ни на клиентской, ни на серверной части нет никаких механизмов, проверяющих и/или ограничивающих длину введённых в поле «О товаре» данных.
2. Исходный вариант подробного описания: в клиентской и серверной части приложения отсутствуют проверка и ограничение длины данных, вводимых в поле «О товаре» на странице http://testapplication/admin/goods/edit/.

3. Конечный вариант подробного описания:
   - фактический результат: в описании товара (поле «О товаре», http://testapplication/admin/goods/edit/) отсутствуют проверка и ограничение длины вводимого текста (максимум 250 символов);
   - ожидаемый результат: в случае попытки ввода более 251 символа выводится сообщение об ошибке.
4. Определение «что, где и при каких условиях случилось»:
   - что: отсутствуют проверка и ограничение длины вводимого текста;
   - где: описание товара, поле «О товаре», http://testapplication/admin/goods/edit/;
   - при каких условиях: в данном случае дефект присутствует всегда, вне зависимости от каких бы то ни было особых условий.
5. Первичная формулировка: отсутствуют проверка и ограничение максимальной длины текста, вводимого в поле «О товаре» описания товара.
6. Сокращение (итоговое краткое описание): нет ограничения максимальной длины поля «О товаре». Английский вариант: no check for «О товаре» max length.

Ситуация 2. Попытка открыть в приложении пустой файл приводит к краху клиентской части приложения и потере несохранённых пользовательских данных на сервере:

1. Суть проблемы: клиентская часть приложения бездумно читает заголовок файла, не проверяя ни размер, ни корректность формата, – вовсе ничего; возникает некая внутренняя ошибка, и клиентская часть приложения некорректно прекращает работу, не закрыв сессию с сервером; сервер закрывает сессию по time out (повторный запуск клиентской части запускает новую сессию, так что старая сессия и все данные в ней в любом случае утеряны).
2. Исходный вариант подробного описания: некорректный анализ открываемого клиентом файла приводит к краху клиента и необратимой утере текущей сессии с сервером.
3. Конечный вариант подробного описания:
   - фактический результат: отсутствие проверки корректности открываемого клиентской частью приложения файла (в том числе пустого) приводит к краху клиентской части и необратимой потере текущей сессии с сервером (см. BR852345);
   - ожидаемый результат: производится анализ структуры открываемого файла; в случае обнаружения проблем отображается сообщение о невозможности открытия файла.
4. Определение «что, где и при каких условиях случилось»:
   - что: крах клиентской части приложения;
   - где: конкретное место в приложении определить едва ли возможно;
   - при каких условиях: при открытии пустого или повреждённого файла.
5. Первичная формулировка: отсутствие проверки корректности открываемого файла приводит к краху клиентской части приложения и потере пользовательских данных.
6. Сокращение (итоговое краткое описание): крах клиента и потеря данных при открытии повреждённых файлов. Английский вариант: client crash and data loss on damaged/empty files opening.

Ситуация 3. Крайне редко по совершенно непонятным причинам на сайте нарушается отображение всего русского текста (как статических надписей, так и информации из базы данных, генерируемой динамически и т. д. – всё становится нечитаемым набором символов):

1. Суть проблемы: фреймворк, на котором построен сайт, подгружает специфические шрифты с удалённого сервера; если соединение обрывается, нужные шрифты не подгружаются и используются шрифты по умолчанию, в которых нет русских символов.
2. Исходный вариант подробного описания: ошибка загрузки шрифтов с удалённого сервера приводит к использованию локальных несовместимых с требуемой кодировкой шрифтов.
3. Конечный вариант подробного описания:
   - фактический результат: периодическая невозможность загрузить шрифты с удалённого сервера приводит к использованию локальных шрифтов, несовместимых с требуемой кодировкой;
   - ожидаемый результат: необходимые шрифты подгружаются всегда (или используется локальный источник необходимых шрифтов).
4. Определение «что, где и при каких условиях случилось»:
   - что: используются несовместимые с требуемой кодировкой шрифты;
   - где: не определяется по всему сайту;
   - при каких условиях: в случае ошибки соединения с сервером, с которого подгружаются шрифты.
5. Первичная формулировка: периодические сбои внешнего источника шрифтов приводят к сбою отображения русского текста.
6. Сокращение (итоговое краткое описание): неверное отображение русского текста при недоступности внешних шрифтов. Английский вариант: wrong presentation of Russian text in case of external fonts inaccessibility.

Для закрепления материала ещё раз представим эти три ситуации в виде таблицы 5.а.

Таблица 5.а – Проблемные ситуации и формулировки кратких описаний дефектов

Ситуация	**Русский вариант краткого описания**	**Английский вариант краткого описания**
Тестированию подвергается некоторое веб-приложение, поле описания каждого товара должно допускать ввод максимум 250 символов; в процессе тестирования оказалось, что этого ограничения нет	Нет ограничения максимальной длины поля «О товаре»	No check for «О товаре» max length
Попытка открыть в приложении пустой файл приводит к краху клиентской части приложения и потере несохранённых пользовательских данных на сервере	Крах клиента и потеря данных при открытии повреждённых файлов	Client crash and data loss on damaged/empty files opening
Крайне редко по совершенно непонятным причинам на сайте нарушается отображение всего русского текста (как статических надписей, так и данных из базы данных, генерируемых динамически и т. д. – всё «становится вопросиками»)	Неверный показ русского текста при недоступности внешних шрифтов	Wrong presentation of Russian text in case of external fonts inaccessibility

Возвращаемся к рассмотрению полей отчёта о дефекте.

Подробное описание (description) представляет в развёрнутом виде необхо-

димую информацию о дефекте, а также (обязательно!) описание фактического результата, ожидаемого результата и ссылку на требование (если это возможно).

Рассмотрим пример подробного описания:

1. Если в систему входит администратор, на странице приветствия отсутствует логотип.

2. Фактический результат: логотип отсутствует в левом верхнем углу страницы.

3. Ожидаемый результат: логотип отображается в левом верхнем углу страницы.

4. Требование: R245.3.23b.

В отличие от краткого описания, которое, как правило, является одним предложением, здесь можно и нужно давать подробную информацию. Если одна и та же проблема (вызванная одним источником) проявляется в нескольких местах приложения, можно в подробном описании перечислить эти места.

Шаги по воспроизведению (steps to reproduce, STR) описывают действия, которые необходимо выполнить для воспроизведения дефекта. Это поле похоже на шаги тест-кейса, за исключением одного важного отличия: здесь действия прописываются максимально подробно, с указанием конкретных вводимых значений и самых мелких деталей, т. к. отсутствие этой информации в сложных случаях может привести к невозможности воспроизведения дефекта.

Рассмотрим пример шагов воспроизведения:

1. Открыть http://testapplication/admin/login/.
2. Авторизоваться с именем «defaultadmin» и паролем «dapassword».

Дефект: в левом верхнем углу страницы отсутствует логотип (вместо него отображается пустое пространство с надписью «logo»).

Воспроизводимость (reproducibility) показывает, при каждом ли прохождении по шагам воспроизведения дефекта удаётся вызвать его проявление. Это поле принимает всего два значения: всегда (always) или иногда (sometimes).

Можно сказать, что воспроизводимость «иногда» означает, что тестировщик не нашёл настоящую причину возникновения дефекта. Это приводит к серьёзным дополнительным сложностям в работе с дефектом:

- Тестировщику нужно потратить много времени на то, чтобы удостовериться в наличии дефекта (т. к. однократный сбой в работе приложения мог быть вызван огромным количеством посторонних причин).
- Разработчику тоже нужно потратить время, чтобы добиться проявления дефекта и убедиться в его наличии. После внесения исправлений в приложение разработчик фактически должен полагаться только на свой профессионализм, т. к. даже многократное прохождение по шагам воспроизведения в таком случае не гарантирует, что дефект был исправлен (возможно, через ещё 10–20 повторений он бы проявился).
- Тестировщику, верифицирующему исправление дефекта, и вовсе остаётся верить разработчику на слово по той же самой причине: даже если он попытается воспроизвести дефект сто раз и потом прекратит попытки, может так случиться, что на сто первый раз дефект всё же воспроизвёлся бы.

Как легко догадаться, такая ситуация является крайне неприятной, а потому рекомендуется один раз потратить время на тщательную диагностику проблемы, найти её причину и перевести дефект в разряд воспроизводимых всегда.

Важность (severity) показывает степень ущерба, который наносится проекту существованием дефекта.

В общем случае выделяют следующие градации важности:

- Критическая (critical) – существование дефекта приводит к масштабным последствиям катастрофического характера, например, потеря данных, раскры-

тие конфиденциальной информации, нарушение ключевой функциональности приложения и т. д.

- Высокая (major) – существование дефекта приносит ощутимые неудобства многим пользователям в рамках их типичной деятельности, например, недоступность вставки из буфера обмена, неработоспособность общепринятых клавиатурных комбинаций, необходимость перезапуска приложения при выполнении типичных сценариев работы.
- Средняя (medium) – существование дефекта слабо влияет на типичные сценарии работы пользователей, и/или существует обходной путь достижения цели, например, диалоговое окно не закрывается автоматически после нажатия кнопок «OK»/«Cancel», при распечатке нескольких документов подряд не сохраняется значение поля «Двусторонняя печать», перепутаны направления сортировок по некоему полю таблицы.
- Низкая (minor) – существование дефекта редко обнаруживается незначительным процентом пользователей и (почти) не влияет на их работу, например, опечатка в глубоко вложенном пункте меню настроек, некоторое окно сразу при отображении расположено неудобно (нужно перетянуть его в удобное место), неточно отображается время до завершения операции копирования файлов.

Срочность (priority) показывает, как быстро дефект должен быть устранён.

В общем случае выделяют следующие градации срочности:

- Наивысшая (ASAP, as soon as possible) срочность указывает на необходимость устранить дефект настолько быстро, насколько это возможно. В зависимости от контекста «настолько быстро, насколько возможно» может варьироваться от «в ближайшем билде» до единиц минут.
- Высокая (high) срочность означает, что дефект следует исправить вне очереди, т. к. его существование или уже объективно мешает работе, или начнёт создавать такие помехи в самом ближайшем будущем.
- Обычная (normal) срочность означает, что дефект следует исправить в порядке общей очерёдности. Такое значение срочности получает большинство дефектов.
- Низкая (low) срочность означает, что в обозримом будущем исправление данного дефекта не окажет существенного влияния на повышение качества продукта.

Несколько дополнительных рассуждений о важности и срочности стоит рассмотреть отдельно.

Один из самых часто задаваемых вопросов относится к тому, какая между ними связь. Никакой. Для лучшего понимания этого факта можно сравнить важность и срочность с координатами X и Y точки на плоскости. Хоть «на бытовом уровне» и кажется, что дефект с высокой важностью следует исправить в первую очередь, в реальности ситуация может выглядеть совсем иначе.

Чтобы подробно продемонстрировать эту мысль, вернёмся к перечню градаций: заметили ли вы, что для разных степеней важности примеры приведены, а для разных степеней срочности – нет? И это не случайно.

Зная суть проекта и суть дефекта, его важность определить достаточно легко, т. к. мы можем проследить влияние дефекта на критерии качества, степень выполнения требований той или иной важности и т. д. Но срочность исправления дефекта можно определить только в конкретной ситуации.

Поясним на жизненном примере: насколько для жизни человека важна вода? Очень важна, без воды человек умирает. Значит, важность воды для человека мож-

но оценить как критическую. Но можем ли мы ответить на вопрос «Как быстро человеку нужно выпить воды?», не зная, о какой ситуации идёт речь? Если рассматриваемый человек умирает от жажды в пустыне, срочность будет наивысшей. Если он просто сидит в офисе и думает, не попить ли чая, срочность будет обычной или даже низкой.

Вернёмся к примерам из разработки программного обеспечения и покажем четыре случая сочетания важности и срочности в таблице 5.b.

Таблица 5.b – Примеры сочетания важности и срочности дефектов

Срочность	**Важность**	
	критическая	**низкая**
Наивысшая	Проблемы с безопасностью во введённом в эксплуатацию банковском ПО	На корпоративном сайте повредилась картинка с фирменным логотипом
Низкая	В самом начале разработки проекта обнаружена ситуация, при которой могут быть повреждены или вовсе утеряны пользовательские данные	В руководстве пользователя обнаружено несколько опечаток, не влияющих на смысл текста

Симптом (symptom) – позволяет классифицировать дефекты по их типичному проявлению. Не существует никакого общепринятого списка симптомов. Более того, далеко не в каждом инструментальном средстве управления отчётами о дефектах есть такое поле, а там, где оно есть, его можно настроить. В качестве примера рассмотрим следующие значения симптомов дефекта:

- Косметический дефект (cosmetic flaw) – визуально заметный недостаток интерфейса, не влияющий на функциональность приложения (например, надпись на кнопке выполнена шрифтом не той гарнитуры).
- Повреждение/потеря данных (data corruption/loss) – в результате возникновения дефекта искажаются, уничтожаются (или не сохраняются) некоторые данные (например, при копировании файлов копии оказываются повреждёнными).
- Проблема в документации (documentation issue) – дефект относится не к приложению, а к документации (например, отсутствует раздел руководства по эксплуатации).
- Некорректная операция (incorrect operation) – некоторая операция выполняется некорректно (например, калькулятор показывает ответ 17 при умножении 2 на 3).
- Проблема инсталляции (installation problem) – дефект проявляется на стадии установки и/или конфигурирования приложения.
- Ошибка локализации (localization issue) – что-то в приложении не переведено или переведено неверно на выбранный язык интерфейса.
- Нереализованная функциональность (missing feature) – некая функция приложения не выполняется или не может быть вызвана (например, в списке форматов для экспорта документа отсутствует несколько пунктов, которые там должны быть).
- Проблема масштабируемости (scalability) – при увеличении количества доступных приложению ресурсов не происходит ожидаемого прироста производительности приложения).
- Низкая производительность (slow performance) – выполнение неких операций занимает недопустимо большое время.

- Крах системы (system crash) – приложение прекращает работу или теряет способность выполнять свои ключевые функции (также может сопровождаться крахом операционной системы, веб-сервера и т. д.).
- Неожиданное поведение (unexpected behavior) – в процессе выполнения некоторой типичной операции приложение ведёт себя необычным (отличным от общепринятого) образом (например, после добавления в список новой записи активной становится не новая запись, а первая в списке).
- Недружественное поведение (unfriendly behavior) – поведение приложения создаёт пользователю неудобства в работе (например, на разных диалоговых окнах в разном порядке расположены кнопки «OK» и «Cancel»).
- Расхождение с требованиями (variance from specs) – этот симптом указывает, что дефект сложно соотнести с другими симптомами, но тем не менее приложение ведёт себя не так, как описано в требованиях.
- Предложение по улучшению (enhancement) – во многих инструментальных средствах управления отчётами о дефектах для этого случая есть отдельный вид отчёта, т. к. предложение по улучшению формально нельзя считать дефектом: приложение ведёт себя согласно требованиям, но у тестировщика есть обоснованное мнение о том, как ту или иную функциональность можно улучшить.

Часто встречается вопрос о том, может ли у одного дефекта быть сразу несколько симптомов. Да, может. Например, крах системы очень часто ведёт к потере или повреждению данных. Но в большинстве инструментальных средств управления отчётами о дефектах значение поля «Симптом» выбирается из списка, и потому нет возможности указать два и более симптома одного дефекта. В такой ситуации рекомендуется выбирать либо симптом, который лучше всего описывает суть ситуации, либо «наиболее опасный» симптом (например, недружественное поведение, состоящее в том, что приложение не запрашивает подтверждения перезаписи существующего файла, приводит к потере данных; здесь «потеря данных» куда уместнее, чем «недружественное поведение»).

Возможность обойти (workaround) – показывает, существует ли альтернативная последовательность действий, выполнение которой позволило бы пользователю достичь поставленной цели (например, клавиатурная комбинация Ctrl+P не работает, но распечатать документ можно, выбрав соответствующие пункты в меню). В некоторых инструментальных средствах управления отчётами о дефектах это поле может просто принимать значения «Да» и «Нет», в некоторых при выборе «Да» появляется возможность описать обходной путь. Традиционно считается, что дефектам без возможности обхода стоит повысить срочность исправления.

Комментарий (comments, additional info) – может содержать любые полезные для понимания и исправления дефекта данные. Иными словами, здесь можно писать всё то, что нельзя писать на других полях.

Приложения (attachments) – представляет собой не столько поле, сколько список прикреплённых к отчёту о дефекте приложений (копий экрана, вызывающих сбой файлов и т. д.)

Общие рекомендации по формированию приложений таковы:

- Если вы сомневаетесь, делать или не делать приложение, лучше сделайте.
- Обязательно прилагайте так называемые «проблемные артефакты» (например, файлы, которые приложение обрабатывает некорректно).
- Если вы прилагаете копию экрана:
    - чаще всего вам будет нужна копия активного окна (Alt+Print Screen), а не всего экрана (Print Screen);

- обрежьте всё лишее (используйте Snipping Tool или Paint в Windows, Pinta или XPaint в Linux);
- отметьте на копии экрана проблемные места (обведите, нарисуйте стрелку, добавьте надпись – сделайте всё необходимое, чтобы с первого взгляда проблема была заметна и понятна);
- в некоторых случаях стоит сделать одно большое изображение из нескольких копий экрана (разместив их последовательно), чтобы показать процесс воспроизведения дефекта. Альтернативой этого решения является создание нескольких копий экрана, названных так, чтобы имена образовывали последовательность, например: br_9_sc_01.png, br_9_sc_02.png, br_9_sc_03.png;
- сохраните копию экрана в формате JPG (если важна экономия объёма данных) или PNG (если важна точная передача картинки без искажений).

- Если вы прилагаете видеоролик с записью происходящего на экране, обязательно оставляйте только тот фрагмент, который относится к описываемому дефекту (это будет буквально несколько секунд или минут из возможных многих часов записи). Старайтесь подобрать настройки кодеков так, чтобы получить минимальный размер ролика при сохранении достаточного качества изображения.
- Поэкспериментируйте с различными инструментами создания копий экрана и записи видеороликов с происходящим на экране. Выберите наиболее удобное для вас программное обеспечение и приучите себя постоянно его использовать.

5.4 Инструментальные средства управления отчётами о дефектах

Так называемые «инструментальные средства управления отчётами о дефектах» в обычной разговорной речи называют «баг-трекинговыми системами», «баг-трекерами» и т. д. Но мы здесь будем придерживаться более строгой традиционной терминологии.

Инструментальных средств управления отчётами о дефектах (bug tracking system, defect management tool[301]) очень много[302], к тому же многие компании разрабатывают свои внутренние средства решения этой задачи. Зачастую такие инструментальные средства являются частями инструментальных средств управления тестированием.

Как и в случае с инструментальными средствами управления тестированием, здесь не имеет смысла заучивать, как работать с отчётами о дефектах в том или ином средстве. Мы лишь рассмотрим общий набор функций, как правило, реализуемых такими средствами:

- Создание отчётов о дефектах, управление их жизненным циклом с учётом контроля версий, прав доступа и разрешённых переходов из состояния в состояние.

[301] **Defect management tool, Incident management tool.** A tool that facilitates the recording and status tracking of defects and changes. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of defects and provide reporting facilities. See also incident management tool. ISTQB Glossary.

[302] Comparison of issue-tracking systems. Wikipedia. URL: http://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems.

- Сбор, анализ и предоставление статистики в удобной для восприятия человеком форме.
- Рассылка уведомлений, напоминаний и иных артефактов соответствующим сотрудникам.
- Организация взаимосвязей между отчётами о дефектах, тест-кейсами, требованиями и анализ таких связей с возможностью формирования рекомендаций.
- Подготовка информации для включения в отчёт о результатах тестирования.
- Интеграция с системами управления проектами.

Иными словами, хорошее инструментальное средство управления жизненным циклом отчётов о дефектах не только избавляет человека от необходимости внимательно выполнять большое количество рутинных операций, но и предоставляет дополнительные возможности, облегчающие работу тестировщика и делающие её более эффективной.

Для общего развития и лучшего закрепления темы об оформлении отчётов о дефектах мы сейчас рассмотрим несколько картинок с формами из разных инструментальных средств.

Здесь вполне сознательно не приведено никакого сравнения или подробного описания – подобных обзоров достаточно в Интернете, и они стремительно устаревают по мере выхода новых версий обозреваемых продуктов.

Но интерес представляют отдельные особенности интерфейса, на которые мы обратим внимание в каждом из примеров. Важно! Если вас интересует подробное описание каждого поля, связанных с ним процессов и т. д., обратитесь к официальной документации. Здесь будут лишь самые краткие пояснения.

Система отслеживания ошибок **Jira**[303] содержит следующие поля (рисунок 5.f):

1. Project (проект) позволяет указать, к какому проекту относится дефект.
2. Issue type (тип записи/артефакта) позволяет указать, что именно представляет собой создаваемый артефакт. JIRA позволяет создавать не только отчёты о дефектах, но и множество других артефактов[304], типы которых можно настраивать[305]. По умолчанию представлены:
   - Improvement (предложение по улучшению) – было описано подробно в разделе, посвящённом полям отчёта о дефекте (см. на с. 163, с. 164 описание поля «симптом», значение «предложение по улучшению»);
   - New feature (новая особенность) – описание новой функциональности, нового свойства, новой особенности продукта;
   - Task (задание) – некоторое задание для выполнения тем или иным участником проектной команды;
   - Custom issue (произвольный артефакт) – как правило, это значение при настройке JIRA удаляют, заменяя своими вариантами, или переименовывают в Issue.
3. Summary (краткое описание) позволяет указать краткое описание дефекта.
4. Priority (срочность) позволяет указать срочность исправления дефекта. По умолчанию JIRA предлагает следующие варианты:
   - Highest (самая высокая срочность);
   - High (высокая срочность);
   - Medium (обычная срочность);

[303] JIRA – Issue & Project Tracking Software. URL: https://www.atlassian.com/software/jira/.
[304] What is an Issue. URL: https://confluence.atlassian.com/display/JIRA/What+is+an+Issue.
[305] Defining Issue Type Field Values. URL: https://confluence.atlassian.com/display/JIRA/Defining+Issue+Type+Field+Values.

- Low (низкая срочность);
- Lowest (самая низкая срочность).

Обратите внимание! По умолчанию поля severity (важность) нет. Но его можно добавить.

5. Components (компоненты) содержит перечень компонентов приложения, затронутых дефектом (хотя иногда здесь перечисляют симптомы дефектов).
6. Affected versions (затронутые версии) содержит перечень версий продукта, в которых проявляется дефект.
7. Environment (окружение) содержит описание аппаратной и программной конфигурации, в которой проявляется дефект.
8. Description (подробное описание) позволяет указать подробное описание дефекта.
9. Original estimate (начальная оценка времени исправления) позволяет указать начальную оценку того, сколько времени займёт устранение дефекта.
10. Remaining estimate (расчётное остаточное время исправления) показывает, сколько времени осталось от начальной оценки.
11. Story points (оценочные единицы) позволяет указать сложность дефекта (или иного артефакта) в специальных оценочных единицах, принятых в гибких методологиях управления проектами.
12. Labels (метки) содержит метки (теги, ключевые слова), по которым можно группировать и классифицировать дефекты и иные артефакты.
13. Epic/Theme (история/область) содержит перечень высокоуровневых меток, описывающих относящиеся к дефекту крупные области требований, крупные модули приложения, крупные части предметной области, объёмные пользовательские истории и т. д.
14. External issue id (идентификатор внешнего артефакта) позволяет связать отчёт о дефекте или иной артефакт с внешним документом.
15. Epic link (ссылка на историю/область) содержит ссылку на историю/область (см. пункт 13), наиболее близко относящуюся к дефекту.
16. Has a story/s (истории) содержит ссылки и/или описание пользовательских историй, связанных с дефектом (как правило, здесь приводятся ссылки на внешние документы).
17. Tester (тестировщик) содержит имя автора описания дефекта.
18. Additional information (дополнительная информация) содержит полезную дополнительную информацию о дефекте.
19. Sprint (спринт) содержит номер спринта (итерации 2–4 недель разработки проекта в терминологии гибких методологий управления проектами), во время которого был обнаружен дефект.

Многие дополнительные поля и возможности становятся доступными при других операциях с дефектами (просмотром или редактированием созданного дефекта, просмотре отчётов и т. д.)

Рисунок 5.f – Создание отчёта о дефекте в JIRA

Система отслеживания ошибок **Bugzilla**[306] содержит следующие поля (рисунок 5.g):

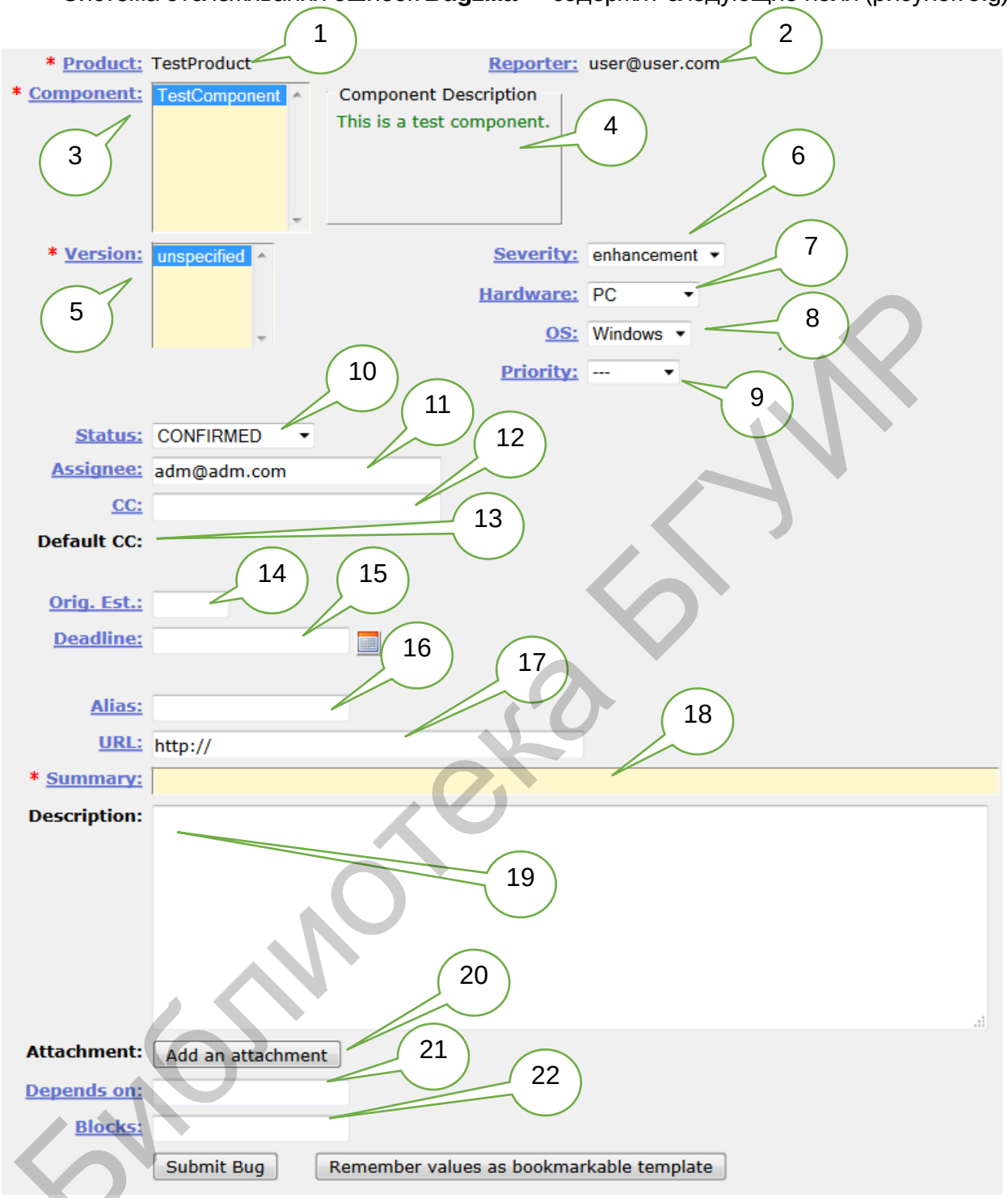


Рисунок 5.g – Создание отчёта о дефекте в Bugzilla

1. Product (продукт) позволяет указать, к какому продукту (проекту) относится дефект.
2. Reporter (автор отчёта) содержит e-mail автора описания дефекта.
3. Component (компонент) содержит указание компонента приложения, к которому относится описываемый дефект.

[306] Bugzilla. URL: https://www.bugzilla.org.

4. Component description (описание компонента) содержит описание компонента приложения, к которому относится описываемый дефект. Эта информация загружается автоматически при выборе компонента.
5. Version (версия) содержит указание версии продукта, в которой был обнаружен дефект.
6. Severity (важность) содержит указание важности дефекта. По умолчанию предложены такие варианты:
   - Blocker (блокирующий дефект) – дефект не позволяет решить с помощью приложения некоторую задачу;
   - Critical (критическая важность);
   - Major (высокая важность);
   - Normal (обычная важность);
   - Minor (низкая важность);
   - Trivial (самая низкая важность);
   - Enhancement (предложение по улучшению) – было описано подробно в разделе, посвящённом полям отчёта о дефекте (см. на с. 163, с. 164 описание поля «симптом», значение «предложение по улучшению»).
7. Hardware (аппаратное обеспечение) позволяет выбрать профиль аппаратного окружения, в котором проявляется дефект.
8. OS (операционная система) позволяет указать операционную систему, под которой проявляется дефект.
9. Priority (срочность) позволяет указать срочность исправления дефекта. По умолчанию Bugzilla предлагает следующие варианты:
   - Highest (самая высокая срочность);
   - High (высокая срочность);
   - Normal (обычная срочность);
   - Low (низкая срочность);
   - Lowest (самая низкая срочность).
10. Status (статус) позволяет установить статус отчёта о дефекте. По умолчанию Bugzilla предлагает следующие варианты:
    - Unconfirmed (не подтверждено) – дефект пока не изучен, и нет гарантии того, что он действительно корректно описан;
    - Confirmed (подтверждено) – дефект изучен, корректность описания подтверждена;
    - In progress (в работе) – ведётся работа по изучению и устранению дефекта.

В официальной документации рекомендуется сразу же после установки Bugzilla сконфигурировать набор статусов и правила жизненного цикла отчёта о дефектах в соответствии с принятыми в вашей компании правилами.

11. Assignee (ответственный) указывает e-mail участника проектной команды, ответственного за изучение и исправление дефекта.
12. CC (уведомлять) содержит список e-mail адресов участников проектной команды, которые будут получать уведомления о происходящем с данным дефектом.
13. Default CC (уведомлять по умолчанию) содержит e-mail адрес(а) участников проектной команды, которые по умолчанию будут получать уведомления о происходящем с любыми дефектами (чаще всего здесь указываются e-mail адреса рассылок).
14. Original estimation (начальная оценка) позволяет указать начальную оценку того, сколько времени займёт устранение дефекта.

15. Deadline (крайний срок) позволяет указать дату, к которой дефект обязательно нужно исправить.
16. Alias (псевдоним) позволяет указать короткое запоминающееся название дефекта (возможно, в виде некоей аббревиатуры) для удобства упоминания дефекта в разнообразных документах.
17. URL (URL) позволяет указать URL, по которому проявляется дефект (особенно актуально для веб-приложений).
18. Summary (краткое описание) позволяет указать краткое описание дефекта.
19. Description (подробное описание) позволяет указать подробное описание дефекта.
20. Attachment (вложение) позволяет добавить к отчёту о дефекте вложения в виде прикреплённых файлов.
21. Depends on (зависит от) позволяет указать перечень дефектов, которые должны быть устранены до начала работы с данным дефектом.
22. Blocks (блокирует) позволяет указать перечень дефектов, к работе с которыми можно будет приступить только после устранения данного дефекта.

Система отслеживания ошибок **Mantis**[307] содержит следующие поля (рисунок 5.h):

1. Category (категория) содержит указание проекта или компонента приложения, к которому относится описываемый дефект.
2. Reproducibility (воспроизводимость) дефекта. Mantis предлагает нетипично большое количество вариантов:
   - Always (всегда);
   - Sometimes (иногда);
   - Random (случайным образом) – вариация на тему «иногда», когда не удалось установить никакой закономерности проявления дефекта;
   - Have not tried (не проверено) – это не столько воспроизводимость, сколько статус, но Mantis относит это значение к данному полю;
   - Unable to reproduce (не удалось воспроизвести) – это не столько воспроизводимость, сколько резолюция к отклонению отчёта о дефекте, но в Mantis тоже отнесено к данному полю;
   - N/A (non-applicable, неприменимо) – используется для дефектов, к которым неприменимо понятие воспроизводимости (например, проблемы с документацией).
3. Severity (важность) содержит указание важности дефекта. По умолчанию предложены такие варианты:
   - Block (блокирующий дефект) – дефект не позволяет решить с помощью приложения некоторую задачу;
   - Crash (критическая важность) – как правило, относится к дефектам, вызывающим неработоспособность приложения;
   - Major (высокая важность);
   - Minor (низкая важность);
   - Tweak (доработка) – как правило, косметический дефект;
   - Text (текст) – как правило, дефект относится к тексту (опечатки и т. д.);
   - Trivial (самая низкая важность);
   - Feature (особенность) – отчёт представляет собой не описание дефекта, а запрос на добавление/изменение функциональности или свойств приложения.

[307] Mantis Bug Tracker. URL: https://www.mantisbt.org.

4. Priority (срочность) позволяет указать срочность исправления дефекта. По умолчанию Mantis предлагает следующие варианты:
   - Immediate (незамедлительно);
   - Urgent (самая высокая срочность);
   - High (высокая срочность);
   - Normal (обычная срочность);
   - Low (низкая срочность);
   - None (нет) – срочность не указана или не может быть определена.
5. Select profile (выбрать профиль) позволяет выбрать из заранее подготовленного списка профиль аппаратно-программной конфигурации, под которой проявляется дефект. Если такого списка нет или он не содержит необходимых вариантов, можно вручную заполнить поля 6–8 (см. рисунок 5.h).
6. Platform (платформа) позволяет указать аппаратную платформу, под которой проявляется дефект.
7. OS (операционная система) позволяет указать операционную систему, под которой проявляется дефект.
8. OS Version (версия операционной системы) позволяет указать версию операционной системы, под которой проявляется дефект.
9. Product version (версия продукта) позволяет указать версию приложения, в которой был обнаружен дефект.
10. Summary (краткое описание) позволяет указать краткое описание дефекта.
11. Description (подробное описание) позволяет указать подробное описание дефекта.
12. Steps to reproduce (шаги по воспроизведению) позволяет указать шаги по воспроизведению дефекта.
13. Additional information (дополнительная информация) позволяет указать любую дополнительную информацию, которая может пригодиться при анализе и устранении дефекта.
14. Upload file (загрузить файл) позволяет загрузить копии экрана и тому подобные файлы, которые могут быть полезны при анализе и устранении дефекта.
15. View status (статус просмотра) позволяет управлять правами доступа к отчёту о дефекте и предлагает по умолчанию два варианта:
    - Public (публичный);
    - Private (закрытый).
16. Report stay (остаться в режиме добавления отчётов) – отметка этого поля позволяет после сохранения данного отчёта сразу же начать писать следующий.

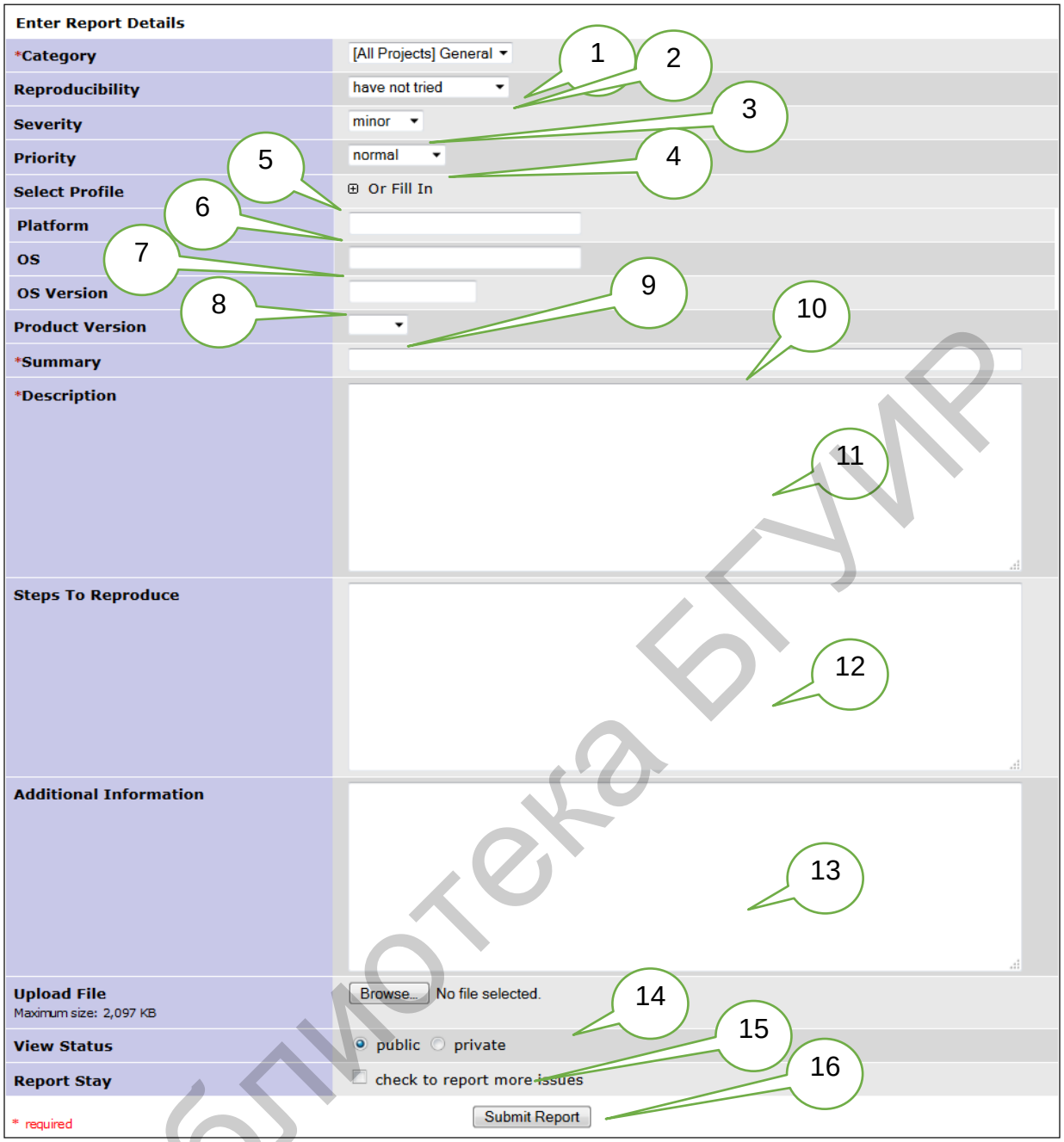


Рисунок 5.h – Создание отчёта о дефекте в Mantis

Задание 5.b: Изучите ещё 3–5 инструментальных средств управления жизненным циклом отчётов о дефектах, почитайте документацию по ним, посоздавайте в них несколько отчётов о дефектах.

5.5 Свойства качественных отчётов о дефектах

Отчёт о дефекте может оказаться некачественным (а следовательно, вероятность исправления дефекта понизится), если в нём нарушено одно из следующих свойств.

Тщательное заполнение всех полей точной и корректной информацией. Нарушение этого свойства происходит по множеству причин: недостаточно опыта у тестировщика, невнимательность, лень и т. д. Самыми яркими проявлениями такой проблемы можно считать следующие:

- Часть важных для понимания проблемы полей не заполнена. В результате отчёт превращается в набор обрывочных сведений, использовать которые для исправления дефекта невозможно.
- Предоставленной информации недостаточно для понимания сути проблемы. Например, из такого плохого подробного описания вообще не ясно, о чём идёт речь: «Приложение иногда неверно конвертирует некоторые файлы».
- Предоставленная информация является некорректной (например, указаны неверные сообщения приложения, неверные технические термины и т. д.) Чаще всего такое происходит по невнимательности (последствия ошибочного copy-paste и отсутствия финальной вычитки отчёта перед публикацией).
- «Дефект» (именно так, в кавычках) найден в функциональности, которая ещё не была объявлена как готовая к тестированию. То есть тестировщик констатирует, что неверно работает то, что и не должно было (пока!) верно работать.
- В отчёте присутствует жаргонная лексика: как в прямом смысле – нелитературные высказывания, так и некие технические жаргонизмы, понятные ограниченному кругу людей. Например, «Фигово подцепились чартники» – имелось в виду «Не все таблицы кодировок загружены успешно».
- Отчёт вместо описания проблемы с приложением критикует работу кого-то из участников проектной команды. Например, «Ну каким дураком надо быть, чтобы сделать подобное?!»
- В отчёте упущена некоторая незначительная на первый взгляд, но по факту критичная для воспроизведения дефекта проблема. Чаще всего это проявляется в виде пропуска какого-то шага по воспроизведению, отсутствию или недостаточной подробности описания окружения, чрезмерно обобщённом указании вводимых значений и т. п.
- Отчёту выставлены неверные (как правило, заниженные) важность или срочность. Чтобы избежать этой проблемы, стоит тщательно исследовать дефект, определять его наиболее опасные последствия и аргументированно отстаивать свою точку зрения, если коллеги считают иначе.
- К отчёту не приложены необходимые копии экрана (особенно важные для косметических дефектов) или иные файлы. Характерно такой ошибке то, что отчёт описывает неверную работу приложения с некоторым файлом, но сам файл не приложен.
- Отчёт написан безграмотно с точки зрения применяемых языковых конструкций, грамматики. Иногда на это можно закрыть глаза, но порой это становится реальной проблемой, например: «Not keyboard in parameters accepting values» (это реальная цитата; и сам автор так и не смог пояснить, что же подразумевается).

Правильный технический язык. Это свойство в равной мере относится и к требованиям, и к тест-кейсам, и к отчётам о дефектах – к любой документации, а поэтому не будем повторяться, см. описанное ранее в подразделе 4.5 «Свойства качественных тест-кейсов».

Сравните два подробных описания дефекта (таблица 5.c).

Таблица 5.c – Сравнение описания дефектов

Плохое подробное описание	Хорошее подробное описание
Когда мы как будто бы хотим убрать папку, где что-то есть внутри, оно не спрашивает, хотим ли мы	Не производится запрос подтверждения при удалении непустого подкаталога в каталоге SOURCE_DIR. **Act:** производится удаление непустого подкаталога (со всем его содержимым) в каталоге SOURCE_DIR без запроса подтверждения
–	**Exp:** в случае если в каталоге SOURCE_DIR приложение обнаруживает непустой подкаталог, оно прекращает работу с выводом сообщения «Non-empty subfolder [имя подкаталога] in SOURCE_DIR folder detected. Remove it manually or restart application with --force_file_operations key to remove automatically.» **Req:** UR.56.BF.4.c

Специфичность описания шагов. Говоря о тест-кейсах, мы подчёркивали, что в их шагах стоит придерживаться золотой середины между специфичностью и общностью. В отчётах о дефектах предпочтение, как правило, отдаётся специфичности по очень простой причине: нехватка какой-то мелкой детали может привести к невозможности воспроизведения дефекта. Поэтому, если у вас есть хоть малейшее сомнение в том, важна ли какая-то деталь, считайте, что она важна.

Сравните два набора шагов по воспроизведению дефекта (таблица 5.d).

Таблица 5.d – Воспроизведение описания дефектов

Недостаточно специфичные шаги	Достаточно специфичные шаги
1. Отправить на конвертацию файл допустимого формата и размера, в котором русский текст представлен в разных кодировках. Дефект: конвертация кодировок производится неверно	1. Отправить на конвертацию файл в формате HTML размером от 100 Кбайт до 1 Мбайт, в котором русский текст представлен в кодировках UTF8 (10 строк по 100 символов) и WIN-1251 (20 строк по 100 символов). Дефект: текст, который был представлен в UTF8, повреждён (представлен нечитаемым набором символов)

В первом случае воспроизвести дефект практически нереально, т. к. он заключается в особенностях работы внешних библиотек по определению кодировок текста в документе, в то время как во втором случае данных достаточно если и не для понимания сути происходящего (дефект на самом деле очень «хитрый»), то хотя бы для гарантированного воспроизведения и признания факта его наличия.

Ещё раз обратим внимание на главное: в отличие от тест-кейса отчёт о дефекте может обладать повышенной специфичностью, и это будет меньшей проблемой, чем невозможность воспроизведения дефекта из-за излишне обобщённого описания проблемы.

Отсутствие лишних действий и/или их длинных описаний. Чаще всего это свойство подразумевает, что не нужно в шагах по воспроизведению дефекта долго и по пунктам расписывать то, что можно заменить одной фразой (таблица 5.e).

Таблица 5.e – Лишние действия в описании дефектов

Плохо	Хорошо
1. Указать в качестве первого параметра приложения путь к папке с исходными файлами. 2. Указать в качестве второго параметра приложения путь к папке с конечными файлами. 3. Указать в качестве третьего параметра приложения путь к файлу журнала. 4. Запустить приложение. Дефект: приложение использует первый параметр командной строки и как путь к папке с исходными файлами, и как путь к папке с конечными файлами	1. Запустить приложение со всеми тремя корректными параметрами (особенно обратить внимание, чтобы SOURCE_DIR и DESTINATION_DIR не совпадали и не были вложены друг в друга в любом сочетании). Дефект: приложение прекращает работу с сообщением «SOURCE_DIR and DESTINATION_DIR may not be the same!» (судя по всему, первый параметр командной строки используется для инициализации имён обоих каталогов)

Вторая по частоте ошибка – начало каждого отчёта о дефекте с запуска приложения и подробного описания по приведению его в то или иное состояние. Вполне допустимой практикой является написание в отчёте о дефекте приготовлений (по аналогии с тест-кейсами) или описание нужного состояния приложения в одном (первом) шаге.

Сравните ошибки в начальных шагах описания дефектов (таблица 5.f).

Таблица 5.f – Ошибки в начальных шагах описания дефектов

Плохо	Хорошо
1. Запустить приложение со всеми верными параметрами. 2. Подождать более 30 мин. 3. Передать на конвертацию файл допустимого формата и размера. Дефект: приложение не обрабатывает файл	Предусловие: приложение запущено и проработало более 30 мин. 1. Передать на конвертацию файл допустимого формата и размера. Дефект: приложение не обрабатывает файл

Отсутствие дубликатов. Когда в проектной команде работает большое количество тестировщиков, может возникнуть ситуация, при которой один и тот же дефект будет описан несколько раз разными людьми. А иногда бывает так, что даже один и тот же тестировщик уже забыл, что когда-то давно уже обнаруживал некоторую проблему, и теперь описывает её заново. Избежать подобной ситуации позволяет следующий набор рекомендаций:

- Если вы не уверены, что дефект не был описан ранее, произведите поиск с помощью вашего инструментального средства управления жизненным циклом отчётов о дефектах.
- Пишите максимально информативные краткие описания (т. к. поиск в первую очередь проводят по ним). Если на вашем проекте накопится множество дефектов с краткими описаниями в стиле «кнопка не работает», вы потратите

очень много времени, раз за разом перебирая десятки таких отчётов в поисках нужной информации.

- Используйте по максимуму возможности вашего инструментального средства: указывайте в отчёте о дефекте компоненты приложения, ссылки на требования, расставляйте теги и т. д. – всё это позволит легко и быстро сузить в будущем круг поиска.
- Указывайте в подробном описании дефекта текст сообщений от приложения, если это возможно. По такому тексту можно найти даже тот отчёт, в котором остальная информация приведена в слишком общем виде.
- Старайтесь по возможности участвовать в так называемых «собраниях по прояснению» (clarification meetings[308]), т. к. пусть вы и не запомните каждый дефект или каждую пользовательскую историю дословно, но в нужный момент может возникнуть ощущение, что «что-то такое я уже слышал», которое заставит вас произвести поиск и подскажет, что именно искать.
- Если вы обнаружили какую-то дополнительную информацию, внесите её в уже существующий отчёт о дефекте (или попросите сделать это его автора), но не создавайте отдельный отчёт.

Очевидность и понятность. Описывайте дефект так, чтобы у читающего ваш отчёт не возникло ни малейшего сомнения в том, что это действительно дефект. Лучше всего это свойство достигается за счёт тщательного объяснения фактического и ожидаемого результата, а также указания ссылки на требование в поле «Подробное описание».

Сравните очевидность и понятность в описании дефектов (таблица 5.g).

Таблица 5.g – Очевидность и понятность в описании дефектов

Плохое подробное описание	**Хорошее подробное описание**
Приложение не сообщает об обнаруженных подкаталогах в каталоге SOURCE_DIR	Приложение не уведомляет пользователя об обнаруженных в каталоге SOURCE_DIR подкаталогах, что приводит к необоснованным ожиданиям пользователями обработки файлов в таких подкаталогах. **Act:** приложение начинает (продолжает) работу, если в каталоге SOURCE_DIR находятся подкаталоги. **Exp:** в случае если в каталоге SOURCE_DIR приложение при запуске или в процессе работы обнаруживает пустой подкаталог, оно автоматически его удаляет (**логично ли это?**), если же обнаружен непустой подкаталог, приложение прекращает работу с выводом сообщения «Non-empty subfolder [имя подкаталога] in SOURCE_DIR folder detected. Remove it manually or restart application with --force_file_operations key to remove automatically». **Req:** UR.56.BF.4.c

[308] **Clarification meeting**. A discussion that helps the customers achieve «advance clarity» – consensus on the desired behavior of each story – by asking questions and getting examples. Crispin L., Gregory J. Agile Testing.

После прочтения подробного описания первого варианта очень хочется спросить: «И что? А оно разве должно уведомлять?» Второй же вариант подробного описания даёт чёткое пояснение, что такое поведение является ошибочным согласно текущему варианту требований. Более того, во втором варианте отмечен вопрос (а в идеале нужно сделать соответствующую отметку и в самом требовании), призывающий пересмотреть алгоритм корректного поведения приложения в подобной ситуации; т. е. мы не только качественно описываем текущую проблему, но и поднимаем вопрос о дальнейшем улучшении приложения.

Прослеживаемость. Из содержащейся в качественном отчёте о дефекте информации должно быть понятно, какую часть приложения, какие функции и какие требования затрагивает дефект. Лучше всего это свойство достигается правильным использованием возможностей инструментального средства управления отчётами о дефектах: указывайте в отчёте о дефекте компоненты приложения, ссылки на требования, тест-кейсы, смежные отчёты о дефектах (похожих дефектах; зависимых и зависящих от данного дефектах), расставляйте теги и т. д.

Некоторые инструментальные средства даже позволяют строить визуальные схемы на основе таких данных, что позволяет управление прослеживаемостью даже на очень больших проектах превратить из непосильной для человека задачи в тривиальную работу.

Отдельные отчёты для каждого нового дефекта. Существует два незыблемых правила:

- В каждом отчёте описывается **ровно один** дефект (если один и тот же дефект проявляется в нескольких местах, эти проявления перечисляются в подробном описании).
- При обнаружении нового дефекта создаётся новый отчёт. **Нельзя** для описания **нового** дефекта править **старые** отчёты, переводя их в состояние «открыт заново».

Нарушение первого правила приводит к объективной путанице, которую проще всего проиллюстрировать так: представьте, что в одном отчёте описано два дефекта, один из которых был исправлен, а второй – нет. В какое состояние переводить отчёт? Неизвестно.

Нарушение второго правила вообще порождает хаос: мало того, что теряется информация о ранее возникавших дефектах, так к тому же возникают проблемы со всевозможными метриками и банальным здравым смыслом. Именно во избежание этой проблемы на многих проектах правом перевода отчёта из состояния «закрыт» в состояние «открыт заново» обладает ограниченный круг участников команды.

Соответствие принятым шаблонам оформления и традициям. Как и в случае с тест-кейсами, с шаблонами оформления отчётов о дефектах проблем не возникает: они определены имеющимся образцом или экранной формой инструментального средства управления жизненным циклом отчётов о дефектах. Но что касается традиций, которые могут различаться даже в разных командах в рамках одной компании, то единственный совет: почитайте уже готовые отчёты о дефектах, перед тем как писать свои. Это может сэкономить вам много времени и сил.

5.6 Контрольные вопросы и задания

- Дайте определение понятия «дефект».
- Что такое отчёт о дефекте?
- Перечислите и опишите этапы жизненного цикла дефекта.
- Назовите атрибуты отчёта о дефекте.
- Что должно быть приведено в подробном описании дефекта?

- Что такое воспроизводимость дефекта?
- Что делать, если дефект воспроизводится не всегда?
- Что такое важность дефекта?
- Какие существуют градации дефектов по важности?
- Приведите примеры дефектов на каждую градацию по важности.
- Что такое срочность дефекта?
- Какая существует связь между важностью и срочностью дефекта?
- Перечислите основные симптомы дефектов.
- Приведите примеры инструментальных средств управления отчётами о дефектах.
- Какие рекомендации по написанию хороших отчётов о дефектах вы знаете?
- Каковы преимущества хорошего отчёта о дефекте?
- Приведите несколько примеров отчётов о дефектах не из ИТ-сферы и из ИТ-сферы.
- Какова стандартная периодичность выпуска отчёта о результатах тестирования? Чем она может определяться?
- Зачем и кому нужен отчёт о результатах тестирования?
- Что такое система отслеживания ошибок и для чего она нужна?
- Каковы цели написания отчёта о результатах тестирования?

6 ОЦЕНКА ТРУДОЗАТРАТ, ПЛАНИРОВАНИЕ И ОТЧЁТНОСТЬ

6.1 Планирование и отчётность

В подразделе 4.7 «Логика создания эффективных проверок» мы на примере «Конвертера файлов» рассматривали, как при минимальных трудозатратах получить максимальный эффект от тестирования. Это было достаточно просто, т. к. наше приложение очень мало по своим масштабам. Но давайте представим, что тестировать приходится реальный проект, где требования в «страничном эквиваленте» занимают сотни и даже тысячи страниц. Также давайте вспомним подраздел 3.2 «Подробная классификация тестирования» с её несколькими десятками видов тестирования (и это без учёта того факта, что их можно достаточно гибко комбинировать, получая новые варианты) и подумаем, как применить все эти знания (и открываемые ими возможности) в крупном проекте.

Даже если допустить, что мы идеально знаем все технические аспекты предстоящей работы, без ответа остаются такие вопросы, как:

- Когда и с чего начать?
- Всё ли необходимое для выполнения работы у нас есть? Если нет, где взять недостающее?
- В какой последовательности выполнять разные виды работ?
- Как распределить ответственность между участниками команды?
- Как организовать отчётность перед заинтересованными лицами?
- Как объективно определять прогресс и достигнутые успехи?
- Как заранее увидеть возможные проблемы, чтобы успеть их предотвратить?
- Как организовать нашу работу так, чтобы при минимуме затрат получить максимальный результат?

Эти и многие подобные вопросы уже лежат вне технической области – они относятся к управлению проектом. Эта задача сама по себе огромна, поэтому мы рассмотрим лишь малую её часть, с которой многим тестировщикам приходится иметь дело, – планирование и отчётность.

Вспомним жизненный цикл тестирования (см. подраздел 1.2 «Жизненный цикл тестирования»): каждая итерация начинается с планирования и заканчивается отчётностью, которая становится основой для планирования следующей итерации, и так далее (рисунок 6.а). Таким образом, планирование и отчётность находятся в тесной взаимосвязи, и проблемы с одним из этих видов деятельности неизбежно приводят к проблемам с другим видом, а в конечном итоге и к проблемам с проектом в целом.

Если выразить эту мысль чётче, получается:

- Без качественного планирования неясно, кому и что нужно делать.
- Когда не ясно, кому и что нужно делать, работа выполняется плохо.
- Когда работа выполнена плохо и не ясны точные причины, невозможно сделать правильные выводы о том, как исправить ситуацию.
- Без правильных выводов невозможно создать качественный отчёт о результатах работы.
- Без качественного отчёта о результатах работы невозможно создать качественный план дальнейшей работы.
- Всё. Порочный круг замкнулся. Проект «умирает».

Рисунок 6.а – Взаимосвязь (взаимозависимость) планирования и отчётности

Казалось бы, так и в чём же сложность? Давайте будем хорошо планировать и писать качественные отчёты – и всё будет хорошо. Проблема в том, что соответствующие навыки развиты в достаточной мере у крайне небольшого процента людей. Если вы не верите, вспомните, как учили материал в ночь перед экзаменом, как опаздывали на важные встречи и… повторяли это раз за разом, так и не сделав выводов. (Если в вашей жизни такого не было, можно надеяться, что вам повезло оказаться в том небольшом проценте людей, у которых соответствующие навыки развиты хорошо.)

Корень проблемы состоит в том, что планированию и отчётности в школах и университетах учат достаточно поверхностно, при этом (увы) на практике часто выхолащивая эти понятия до пустой формальности (планов, на которые никто не смотрит, и отчётов, которые никто не читает; опять же, кому-то повезло увидеть противоположную ситуацию, но явно немногим).

Итак, к сути. Сначала рассмотрим классические определения.

!!! **Планирование** (planning[309]) – непрерывный процесс принятия управленческих решений и методической организации усилий по их реализации с целью обеспечения качества некоторого процесса на протяжении длительного периода времени.

К высокоуровневым задачам планирования относятся:

- снижение неопределённости;
- повышение эффективности;
- улучшение понимания целей;
- создание основы для управления процессами.

[309] Planning is a continuous process of making entrepreneurial decisions with an eye to the future, and methodically organizing the effort needed to carry out these decisions. There are four basic reasons for project planning: to eliminate or reduce uncertainty; to improve efficiency of the operation; to obtain a better understanding of the objectives; to provide a basis for monitoring and controlling work. Kerzner H. Project Management: A Systems Approach to Planning, Scheduling, and Controlling.

Отчётность (reporting[310]) – сбор и распространение информации о результатах работы (включая текущий статус, оценку прогресса и прогноз развития ситуации).

К высокоуровневым задачам отчётности относятся:

- сбор, агрегация и предоставление в удобной для восприятия форме объективной информации о результатах работы;
- формирование оценки текущего статуса и прогресса (в сравнении с планом);
- обозначение существующих и возможных проблем (если такие есть);
- формирование прогноза развития ситуации и фиксация рекомендаций по устранению проблем и повышению эффективности работы.

Как и было упомянуто ранее, планирование и отчётность относятся к области управления проектом, которая выходит за рамки данной книги.

Если вас интересуют детали, рекомендуется ознакомиться с двумя фундаментальными источниками информации:

- Kerzner H. Project Management: A Systems Approach to Planning, Scheduling, and Controlling;
- PMBOK (Project Management Body of Knowledge).

Мы же переходим к более конкретным вещам, с которыми приходится работать (пусть и на уровне использования, а не создания) даже начинающему тестировщику.

6.2 Тест-план и отчёт о результатах тестирования

Тест-план (test plan[311]) – документ, описывающий и регламентирующий перечень работ по тестированию, а также соответствующие техники и подходы, стратегию, области ответственности, ресурсы, расписание и ключевые даты.

К низкоуровневым задачам планирования в тестировании относятся:

- оценка объёма и сложности работ;
- определение необходимых ресурсов и источников их получения;
- определение расписания, сроков и ключевых точек;
- оценка рисков и подготовка превентивных контрмер;
- распределение обязанностей и ответственности;
- согласование работ по тестированию с деятельностью участников проектной команды, занимающихся другими задачами.

Как и любой другой документ, тест-план может быть качественным или обладать недостатками. Качественный тест-план обладает большинством свойств качественных требований, а также расширяет их набор следующими пунктами:

---

[310] Reporting – collecting and distributing performance information (including status reporting, progress measurement, and forecasting). PMBOK. – 3rd ed.

[311] **Test plan.** A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process. ISTQB Glossary.

- Реалистичность (запланированный подход реально выполним).
- Гибкость (качественный тест-план не только является модифицируемым с точки зрения работы с документом, но и построен таким образом, чтобы при возникновении непредвиденных обстоятельств допускать быстрое изменение любой из своих частей без нарушения взаимосвязи с другими частями).
- Согласованность с общим проектным планом и иными отдельными планами (например, планом разработки).

Тест-план создаётся в начале проекта и дорабатывается по мере необходимости на протяжении всего времени жизни проекта при участии наиболее квалифицированных представителей проектной команды, задействованных в обеспечении качества. Ответственным за создание тест-плана, как правило, является ведущий тестировщик («тест-лид»).

В общем случае тест-план включает следующие разделы (примеры их наполнения будут показаны далее, поэтому здесь – только перечисление):

- **Цель** (purpose). Предельно краткое описание цели разработки приложения (частично это напоминает бизнес-требования, но здесь информация подаётся в ещё более сжатом виде и в контексте того, на что следует обращать первостепенное внимание при организации тестирования и повышения качества).
- **Области, подвергаемые тестированию** (features to be tested). Перечень функций и/или нефункциональных особенностей приложения, которые будут подвергнуты тестированию. В некоторых случаях здесь также приводится приоритет соответствующей области.
- **Области, не подвергаемые тестированию** (features not to be tested). Перечень функций и/или нефункциональных особенностей приложения, которые не будут подвергнуты тестированию. Причины исключения той или иной области из списка тестируемых могут быть самыми различными – от предельно низкой их важности для заказчика до нехватки времени или иных ресурсов. Этот перечень составляется, чтобы у проектной команды и иных заинтересованных лиц было чёткое единое понимание, что тестирование таких-то особенностей приложения не запланировано. Такой подход позволяет исключить появление ложных ожиданий и неприятных сюрпризов.
- **Тестовая стратегия** (test strategy[312]) **и подходы** (test approach[313]). Описание процесса тестирования с точки зрения применяемых методов, подходов, видов тестирования, технологий, инструментальных средств и т. д.
- **Критерии** (criteria). Этот раздел включает следующие подразделы:
    - **приёмочные критерии, критерии качества** (acceptance criteria[314]) – любые объективные показатели качества, которым разрабатываемый продукт должен соответствовать с точки зрения заказчика или пользователя, чтобы считаться готовым к эксплуатации;

[312] **Test strategy.** A high-level description of the test levels to be performed and the testing within those levels (group of test activities that are organized and managed together, e.g. component test, integration test, system test and acceptance test) for an organization or program (one or more projects). ISTQB Glossary.

[313] **Test approach.** The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed. ISTQB Glossary.

[314] **Acceptance criteria.** The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity. ISTQB Glossary.

- **критерии начала тестирования** (entry criteria[315]) – перечень условий, при выполнении которых команда приступает к тестированию. Наличие этого критерия страхует команду от бессмысленной траты усилий в условиях, когда тестирование не принесёт ожидаемой пользы;
- **критерии приостановки тестирования** (suspension criteria[316]) – перечень условий, при выполнении которых тестирование приостанавливается. Наличие этого критерия также страхует команду от бессмысленной траты усилий в условиях, когда тестирование не принесёт ожидаемой пользы;
- **критерии возобновления тестирования** (resumption criteria[317]) – перечень условий, при выполнении которых тестирование возобновляется (как правило, после приостановки);
- **критерии завершения тестирования** (exit criteria[318]) – перечень условий, при выполнении которых тестирование завершается. Наличие этого критерия страхует команду как от преждевременного прекращения тестирования, так и от продолжения тестирования в условиях, когда оно уже перестаёт приносить ощутимый эффект.

- **Ресурсы** (resources). В данном разделе тест-плана перечисляются все необходимые для успешной реализации стратегии тестирования ресурсы, которые в общем случае можно разделить на следующие:
  - программные ресурсы (какое ПО необходимо команде тестировщиков, сколько копий и с какими лицензиями (если речь идёт о коммерческом ПО));
  - аппаратные ресурсы (какое аппаратное обеспечение, в каком количестве и к какому моменту необходимо команде тестировщиков);
  - человеческие ресурсы (сколько специалистов какого уровня и со знаниями в каких областях должно подключиться к команде тестировщиков в тот или иной момент времени);
  - временные ресурсы (сколько по времени займёт выполнение тех или иных работ);
  - финансовые ресурсы (в какую сумму обойдётся использование имеющихся или получение недостающих ресурсов, перечисленных в предыдущих пунктах этого списка); во многих компаниях финансовые ресурсы могут быть представлены отдельным документом, т. к. являются конфиденциальной информацией.
- **Расписание** (test schedule[319]). Фактически это календарь, в котором указано, что и к какому моменту должно быть сделано. Особое внимание уделяется так называемым «ключевым точкам» (milestones), к моменту наступления которых

[315] **Entry criteria.** The set of generic and specific conditions for permitting a process to go forward with a defined task, e.g. test phase. The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria. ISTQB Glossary.

[316] **Suspension criteria.** The criteria used to (temporarily) stop all or a portion of the testing activities on the test items. ISTQB Glossary.

[317] **Resumption criteria.** The criteria used to restart all or a portion of the testing activities that were suspended previously. ISTQB Glossary.

[318] **Exit criteria.** The set of generic and specific conditions, agreed upon with the stakeholders for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing. ISTQB Glossary.

[319] **Test schedule.** A list of activities, tasks or events of the test process, identifying their intended start and finish dates and/or times, and interdependencies. ISTQB Glossary.

должен быть получен какой-то значимый ощутимый результат.

- **Роли и ответственность** (roles and responsibility). Перечень необходимых ролей (например, «ведущий тестировщик», «эксперт по оптимизации производительности») и область ответственности специалистов, выполняющих эти роли.
- **Оценка рисков** (risk evaluation). Перечень рисков, которые с высокой вероятностью могут возникнуть в процессе работы над проектом. По каждому риску даётся оценка представляемой им угрозы и приводятся варианты выхода из ситуации.
- **Документация** (documentation). Перечень используемой тестовой документации с указанием, кто и когда должен её готовить и кому передавать.
- **Метрики** (metrics[320]). Числовые характеристики показателей качества, способы их оценки, формулы и т. д. На этот раздел, как правило, формируется множество ссылок из других разделов тест-плана.

Метрики в тестировании являются настолько важными, что о них мы поговорим отдельно.

Метрика (metric[320]) – числовая характеристика показателя качества. Может включать описание способов оценки и анализа результата.

Сначала поясним важность метрик на тривиальном примере. Представьте, что заказчик интересуется текущим положением дел и просит вас кратко охарактеризовать ситуацию с тестированием на проекте. Общие слова «всё хорошо», «всё плохо», «нормально» и т. п. его, конечно, не устроят, т. к. они предельно субъективны и могут быть крайне далеки от реальности. И совсем иначе выглядит ответ наподобие такого: «Реализовано 79 % требований (в том числе 94 % важных), за последние три спринта тестовое покрытие выросло с 63 до 71 %, а общий показатель прохождения тест-кейсов вырос с 85 до 89 %. Иными словами, мы полностью укладываемся в план по всем ключевым показателям, а по разработке даже идём с небольшим опережением».

Чтобы оперировать всеми этими числами (а они нужны не только для отчётности, но и для организации работы проектной команды), их нужно как-то вычислить. Именно это и позволяют сделать метрики. Затем вычисленные значения можно использовать:

- для принятия решений о начале, приостановке, возобновлении или прекращении тестирования (см. раздел «Критерии» тест-плана, с. 183);
- определения степени соответствия продукта заявленным критериям качества;
- определения степени отклонения фактического развития проекта от плана;
- выявления «узких мест», потенциальных рисков и иных проблем;
- оценки результативности принятых управленческих решений;
- подготовки объективной информативной отчётности и т. д.

Метрики могут быть как прямыми (не требуют вычислений), так и расчётными (вычисляются по формуле). Типичные примеры прямых метрик – количество разработанных тест-кейсов, количество найденных дефектов и т. д. В расчётных метриках могут использоваться как совершенно тривиальные, так и довольно сложные формулы (таблица 6.а).

[320] **Metric.** A measurement scale and the method used for measurement. ISTQB Glossary.

Таблица 6.а – Примеры расчётных метрик

Простая расчётная метрика	**Сложная расчётная метрика**
$T^{SP} = \frac{T^{Success}}{T^{Total}} \cdot 100\ \%,$ где T^{SP} – процентный показатель успешного прохождения тест-кейсов; $T^{Success}$ – количество успешно выполненных тест-кейсов; T^{Total} – общее количество выполненных тест-кейсов. Минимальные границы значений: • начальная фаза проекта: 10 %; • основная фаза проекта: 40 %; • финальная фаза проекта: 85 %	$T^{SC} = \sum_{Level}^{MaxLevel} \frac{(T_{Level} \cdot I)^{R_{Level}}}{B_{Level}},$ где T^{SC} – интегральная метрика прохождения тест-кейсов во взаимосвязи с требованиями и дефектами; T_{Level} – степень важности тест-кейса; I – количество выполнений тест-кейса; R_{Level} – степень важности требования, проверяемого тест-кейсом; B_{Level} – количество дефектов, обнаруженных тест-кейсом. Способ анализа: • идеальным состоянием является непрерывный рост значения T^{SC}; • в случае отрицательной динамики уменьшение значения T^{SC} на 15 % и более за последние три спринта может трактоваться как недопустимое и являться достаточным поводом для приостановки тестирования

В тестировании существует большое количество общепринятых метрик, многие из которых могут быть собраны автоматически с использованием инструментальных средств управления проектами. Например[321]:

- процентное отношение (не) выполненных тест-кейсов ко всем имеющимся;
- процентный показатель успешного прохождения тест-кейсов (см. «Простая расчётная метрика» в таблице 6.а);
- процентный показатель заблокированных тест-кейсов;
- плотность распределения дефектов;
- эффективность устранения дефектов;
- распределение дефектов по важности и срочности и т. д.

Как правило, при формировании отчётности нас будет интересовать не только текущее значение метрики, но и её динамика во времени, которую очень удобно изображать графически (что тоже могут выполнять автоматически многие средства управления проектами).

Некоторые метрики могут вычисляться на основе данных о расписании. *Например* – Метрика «сдвига расписания» вычисляется так:

$$\text{SheduleSlippage} = \frac{\text{DaysToDeadline}}{\text{NeededDays}} - 1\ ,$$

где $SheduleSlippage$ – значение сдвига расписания;
$DaysToDeadline$ – количество дней до запланированного завершения работы;
$NeededDays$ – количество дней, необходимое для завершения работы.

Значение $SheduleSlippage$ не должно становиться отрицательным.

[321] Important Software Test Metrics and Measurements – Explained with Examples and Graphs. URL: http://www.softwaretestinghelp.com/software-test-metrics-and-measurements/.

Таким образом, мы видим, что метрики являются мощнейшим средством сбора и анализа информации. И вместе с тем здесь кроется опасность, когда инструментальное средство собирает большое количество данных, вычисляет множество чисел и строит десятки графиков, но… никто не понимает, как их трактовать. Ни при каких условиях нельзя допускать ситуацию «метрика ради метрики». Обратите внимание, что к обеим метрикам в таблице 6.а и к только что рассмотренной метрике $SheduleSlippage$ прилагается краткое руководство по их трактовке. И чем сложнее и уникальнее метрика, тем более подробное руководство необходимо для её эффективного применения.

И, наконец, стоит упомянуть про так называемые «метрики покрытия», т. к. они очень часто упоминаются в различной литературе.

Покрытие (coverage[322]) – процентное выражение степени, в которой исследуемый элемент (coverage item[323]) затронут соответствующим набором тест-кейсов.

Самыми простыми представителями метрик покрытия считаются:

- Метрика покрытия требований (требование считается «покрытым», если на него ссылается хотя бы один тест-кейс):

$$R^{SimpleCoverage} = \frac{R^{Covered}}{R^{Total}} \cdot 100\ \%,$$

где $R^{SimpleCoverage}$ – метрика покрытия требований;
$R^{Covered}$ – количество требований, покрытых хотя бы одним тест-кейсом;
R^{Total} – общее количество требований.

- метрику плотности покрытия требований (учитывается, сколько тест-кейсов ссылается на несколько требований):

$$R^{DensityCoverage} = \frac{\sum T_i}{T^{Total} \cdot R^{Total}} \cdot 100\ \%,$$

где $R^{DensityCoverage}$ – плотность покрытия требований;
T_i – количество тест-кейсов, покрывающих *i*-е требование;
T^{Total} – общее количество тест-кейсов;
R^{Total} – общее количество требований.

- Метрика покрытия классов эквивалентности (анализируется, сколько классов эквивалентности затронуто тест-кейсами):

$$E^{Coverage} = \frac{E^{Covered}}{E^{Total}} \cdot 100\ \%,$$

где $E^{Coverage}$ – метрика покрытия классов эквивалентности;
$E^{Covered}$ – количество классов эквивалентности, покрытых хотя бы одним тест-кейсом;
E^{Total} – общее количество классов эквивалентности.

[322] **Coverage, Test coverage.** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite. ISTQB Glossary.
[323] **Coverage item.** An entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements. ISTQB Glossary.

- Метрика покрытия граничных условий (анализируется, сколько значений из группы граничных условий затронуто тест-кейсами):

$$B^{Coverage} = \frac{B^{Covered}}{B^{Total}} \cdot 100\ \%,$$

где $B^{Coverage}$ – метрика покрытия граничных условий;
$B^{Covered}$ – количество граничных условий, покрытых хотя бы одним тест-кейсом;
B^{Total} – общее количество граничных условий.

- Метрики покрытия кода модульными тест-кейсами. Таких метрик очень много, но вся их суть сводится к выявлению некоторой характеристики кода (количество строк, ветвей, путей, условий и т. д.) и определению, какой процент представителей этой характеристики покрыт тест-кейсами.

Метрик покрытия настолько много, что даже в ISTQB-глоссарии дано определение полутора десяткам таковых. Вы можете найти эти определения, выполнив в файле ISTQB-глоссария поиск по слову «coverage».

На этом мы завершаем теоретическое рассмотрение планирования и переходим к примеру – учебному тест-плану для нашего приложения «Конвертер файлов» (см. подраздел 2.7 «Пример анализа и тестирования требований»). Напомним, что приложение является предельно простым, поэтому и тест-план будет очень маленьким (однако, обратите внимание, сколь значительную его часть будет занимать раздел метрик).

ПРИМЕР ТЕСТ-ПЛАНА

Для того чтобы заполнить некоторые части тест-плана, нам придётся сделать допущения о составе проектной команды и времени, отведённом на разработку проекта. Поскольку данный тест-план находится внутри текста книги, у него нет таких типичных частей, как заглавная страница, содержание и т. п.

Цель

Корректное автоматическое преобразование содержимого документов к единой кодировке с производительностью, значительно превышающей производительность человека при выполнении аналогичной задачи.

Области, подвергаемые тестированию

- ПТ-1.*: дымовой тест.
- ПТ-2.*: дымовой тест, тест критического пути.
- ПТ-3.*: тест критического пути.
- БП-1.*: дымовой тест, тест критического пути.
- АК-1.*: дымовой тест, тест критического пути.
- О-4: дымовой тест.
- О-5: дымовой тест.
- ДС-*: дымовой тест, тест критического пути.

Области, не подвергаемые тестированию

- СХ-1: приложение разрабатывается как консольное.
- СХ-2, О-1, О-2: приложение разрабатывается на PHP указанной версии.
- АК-1: заявленная характеристика находится вблизи нижней границы производительности операций, характерных для разрабатываемого приложения.
- О-3: не требует реализации.
- О-6: не требует реализации.

Тестовая стратегия и подходы

Общий подход.

Специфика работы приложения состоит в однократном конфигурировании опытным специалистом и дальнейшем использовании конечными пользователями, для которых доступна только одна операция – размещение файла в каталоге-приёмнике. Поэтому вопросы удобства использования, безопасности и т. п. не исследуются в процессе тестирования.

Уровни функционального тестирования:

- дымовой тест: автоматизированный с использованием командных файлов ОС Windows и Linux;
- тест критического пути: выполняется вручную;
- расширенный тест не производится, т. к. для данного приложения вероятность обнаружения дефектов на этом уровне пренебрежимо мала.

В силу кроссфункциональности проектной команды значительного вклада в повышение качества можно ожидать от аудита кода, совмещённого с ручным тестированием по методу белого ящика. Автоматизация тестирования кода не будет применяться в силу крайне ограниченного времени.

Критерии

- Приёмочные критерии: успешное прохождение 100 % тест-кейсов уровня дымового тестирования и 90 % тест-кейсов уровня критического пути (см. метрику «Успешное прохождение тест-кейсов») при условии устранения 100 % дефектов критической и высокой важности (см. метрику «Общее устранение дефектов»). Итоговое покрытие требований тест-кейсами (см. метрику «Покрытие требований тест-кейсами») должно составлять не менее 80 %.
- Критерии начала тестирования: выход билда.
- Критерии приостановки тестирования: переход к тесту критического пути допустим только при успешном прохождении 100 % тест-кейсов дымового теста (см. метрику «Успешное прохождение тест-кейсов»); тестирование может быть приостановлено в случае, если при выполнении не менее 25 % запланированных тест-кейсов более 50 % из них завершились обнаружением дефекта (см. метрику «Стоп-фактор»).
- Критерии возобновления тестирования: исправление более 50 % обнаруженных на предыдущей итерации дефектов (см. метрику «Текущее устранение дефектов»).
- Критерии завершения тестирования: выполнение более 80 % запланированных на итерацию тест-кейсов (см. метрику «Выполнение тест-кейсов»).

Ресурсы

- Программные ресурсы: четыре виртуальных машины (две с ОС Windows 7 Ent x64, две с ОС Linux Ubuntu 14 LTS x64), две копии PHP Storm 8.
- Аппаратные ресурсы: две стандартных рабочих станции (8 Гбайт RAM, i7 3 ГГц).
- Человеческие ресурсы:
    - старший разработчик с опытом тестирования (стопроцентная занятость на всём протяжении проекта). Роли на проекте: лидер команды, старший разработчик;
    - тестировщик со знанием PHP (стопроцентная занятость на всём протяжении проекта). Роль на проекте: тестировщик.
- Временные ресурсы: одна рабочая неделя (40 ч).
- Финансовые ресурсы: согласно утверждённому бюджету. Дополнительные финансовые ресурсы не требуются.

Расписание

- 25.05 – формирование требований.
- 26.05 – разработка тест-кейсов и скриптов для автоматизированного тестирования.
- 27–28.05 – основная фаза тестирования (выполнение тест-кейсов, написание отчётов о дефектах).
- 29.05 – завершение тестирования и подведение итогов.

Роли и ответственность

- Старший разработчик: участие в формировании требований, участие в аудите кода.
- Тестировщик: формирование тестовой документации, реализация тестирования, участие в аудите кода.

Оценка рисков

- Персонал (вероятность низкая): в случае нетрудоспособности какого-либо из участников команды можно обратиться к представителям проекта «Каталогизатор» для предоставления временной замены (договорённость с менеджером «Каталогизатора» Джоном Смитом достигнута).
- Время (вероятность высокая): заказчиком обозначен крайний срок сдачи 01.06, поэтому время является критическим ресурсом. Рекомендуется приложить максимум усилий для того, чтобы фактически завершить проект 28.05 с тем, чтобы один день (29.05) остался в запасе.
- Иные риски: иных специфических рисков не выявлено.

Документация

- Требования. Ответственный – тестировщик, дата готовности 25.05.
- Тест-кейсы и отчёты о дефектах. Ответственный – тестировщик, период создания 26–28.05.
- Отчёт о результатах тестирования. Ответственный – тестировщик, дата готовности 29.05.

Метрики

- Успешное прохождение тест-кейсов:

$$T^{SP} = \frac{T^{Success}}{T^{Total}} \cdot 100\ \%,$$

где T^{SP}– процентный показатель успешного прохождения тест-кейсов;

$T^{Success}$ – количество успешно выполненных тест-кейсов;

T^{Total} – общее количество выполненных тест-кейсов.

Минимальные границы значений:

- начальная фаза проекта: 10 %;
- основная фаза проекта: 40 %;
- финальная фаза проекта: 80 %.

- Общее устранение дефектов (таблица 6.b):

$$D_{Level}^{FTP} = \frac{D_{Level}^{Closed}}{D_{Level}^{Found}} \cdot 100\ \%,$$

где D_{Level}^{FTP} – процентный показатель устранения дефектов уровня важности за время Level существования проекта;

D_{Level}^{Closed} – количество устранённых за время существования проекта дефектов уровня важности $Level$;

D_{Level}^{Found} – количество обнаруженных за время существования проекта дефектов уровня важности Level.

Таблица 6.b – Минимальные границы значений процентного показателя устранения дефектов за время существования проекта

Фаза проекта	Важность дефекта			
	низкая	средняя	высокая	критическая
Начальная	10 %	40 %	50 %	80 %
Основная	15 %	50 %	75 %	90 %
Финальная	20 %	60 %	100 %	100 %

- Текущее устранение дефектов (таблица 6.c):

$$D_{Level}^{FTP} = \frac{D_{Level}^{Closed}}{D_{Level}^{Found}} \cdot 100\ \%,$$

где D_{Level}^{FTP} – процентный показатель устранения в текущем билде дефектов уровня важности $Level$, обнаруженных в предыдущем билде;

D_{Level}^{Closed} – количество устранённых в текущем билде дефектов уровня важности $Level$;

D_{Level}^{Found} – количество обнаруженных в предыдущем билде дефектов уровня важности $Level$.

Таблица 6.c – Минимальные границы значений процентного показателя устранения дефектов в текущем билде

Фаза проекта	Важность дефекта			
	низкая	средняя	высокая	критическая
Начальная	60 %	60 %	60 %	60 %
Основная	65 %	70 %	85 %	90 %
Финальная	70 %	80 %	95 %	100 %

- Стоп-фактор:

$$S = \begin{cases} Yes, T^E \geq 25\ \% \ \&\& \ T^{SP} < 50\ \% \\ No, T^E < 25\ \% \ \| \ T^{SP} \geq 50\ \% \end{cases},$$

где S – решение о приостановке тестирования;

T^E – текущее значение метрики T^E;

T^{SP} – текущее значение метрики T^{SP}.

- Выполнение тест-кейсов:

$$T^E = \frac{T^{Executed}}{T^{Planned}} \cdot 100\ \%,$$

где T^E – процентный показатель выполнения тест-кейсов;

$T^{Executed}$ – количество выполненных тест-кейсов;

$T^{Planned}$ – количество тест-кейсов, запланированных к выполнению.

Уровни (границы):

- минимальный уровень: 80 %;
- желаемый уровень: 95–100 %.

- Покрытие требований тест-кейсами:

$$R^C = \frac{R^{Covered}}{R^{Total}} \cdot 100\ \%,$$

где R^C – процентный показатель покрытия требования тест-кейсами;
$R^{Covered}$ – количество покрытых тест-кейсами требований;
R^{Total} – общее количество требований.

Минимальные границы значений:

- начальная фаза проекта: 40 %;
- основная фаза проекта: 60 %;
- финальная фаза проекта: 80 % (рекомендуется 90 % и более).

Задание 6.а: Найдите в сети Интернет более развёрнутые примеры тест-планов. Они периодически появляются, но и столь же быстро удаляются, т. к. настоящие (не учебные) тест-планы, как правило, являются конфиденциальной информацией.

На этом мы завершаем обсуждение планирования и переходим к отчётности, которая завершает цикл тестирования.

Отчёт о результатах тестирования (test progress report[324], test summary report[325]) – документ, обобщающий результаты работ по тестированию и содержащий информацию, достаточную для соотнесения текущей ситуации с тест-планом и принятия необходимых управленческих решений.

К низкоуровневым задачам отчётности в тестировании относятся:

- оценка объёма и качества выполненных работ;
- сравнение текущего прогресса с тест-планом (в том числе с помощью анализа значений метрик);
- описание имеющихся сложностей и формирование рекомендаций по их устранению;
- предоставление лицам, заинтересованным в проекте, полной и объективной информации о текущем состоянии качества проекта, выраженной в конкретных фактах и числах.

Как и любой другой документ, отчёт о результатах тестирования может быть качественным или обладать недостатками. Качественный отчёт о результатах тестирования обладает многими свойствами качественных требований, а также расширяет их набор следующими пунктами:

- Информативность (в идеале после прочтения отчёта не должно оставаться никаких открытых вопросов о том, что происходит с проектом в контексте качества).
- Точность и объективность (ни при каких условиях в отчёте не допускается искажение фактов, а личные мнения должны быть подкреплены твёрдыми обоснованиями).

[324] **Test progress report.** A document summarizing testing activities and results, produced at regular intervals, to report progress of testing activities against a baseline (such as the original test plan) and to communicate risks and alternatives requiring a decision to management. ISTQB Glossary.

[325] **Test summary report.** A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria. ISTQB Glossary.

Отчёт о результатах тестирования создаётся по заранее оговорённому расписанию (зависящему от модели управления проектом) при участии большинства представителей проектной команды, задействованных в обеспечении качества. Большое количество фактических данных для отчёта может быть легко извлечено в удобной форме из системы управления проектом. Ответственным за создание отчёта, как правило, является ведущий тестировщик («тест-лид»). При необходимости отчёт может обсуждаться на небольших собраниях.

Отчёт о результатах тестирования в первую очередь нужен следующим лицам:

- менеджеру проекта – как источник информации о текущей ситуации и основа для принятия управленческих решений;
- руководителю команды разработчиков («дев-лиду») – как дополнительный объективный взгляд на происходящее на проекте;
- руководителю команды тестировщиков («тест-лиду») – как способ структурировать собственные мысли и собрать необходимый материал для обращения к менеджеру проекта по насущным вопросам, если в этом есть необходимость;
- заказчику – как наиболее объективный источник информации о том, что происходит на проекте, за который он платит свои деньги.

В общем случае отчёт о результатах тестирования включает следующие разделы (примеры их наполнения будут показаны далее, поэтому здесь – только перечисление).

Важно! Если по поводу вида тест-плана в сообществе тестировщиков есть более-менее устоявшееся мнение, то формы отчётов о результатах тестирования исчисляются десятками (особенно, если отчёт привязан к некоторому отдельному виду тестирования). Здесь приведён наиболее универсальный вариант, который может быть адаптирован под конкретные нужды.

- **Краткое описание** (summary). В предельно краткой форме отражает основные достижения, проблемы, выводы и рекомендации. В идеальном случае прочтения краткого описания может быть достаточно для формирования полноценного представления о происходящем, что избавит от необходимости читать весь отчёт (это важно, т. к. отчёт о результатах тестирования может попасть в руки очень занятым людям).

Важно! Различайте краткое описание отчёта о результатах тестирования и краткое описание отчёта о дефекте (см. подраздел 5.3 «Атрибуты (поля) отчёта о дефекте»). При одинаковом названии они создаются по разным принципам и содержат разную информацию.

- **Команда тестировщиков** (test team). Список участников проектной команды, задействованных в обеспечении качества, с указанием их должностей и ролей в подотчётный период.
- **Описание процесса тестирования** (testing process description). Последовательное описание того, какие работы были выполнены за подотчётный период.
- **Расписание** (timetable). Детализированное расписание работы команды тестировщиков и/или личные расписания участников команды.
- **Статистика по новым дефектам** (new defects statistics). Таблица, в которой представлены данные по обнаруженным за подотчётный период дефектам (с классификацией по стадии жизненного цикла и важности).

- **Список новых дефектов** (new defects list). Список обнаруженных за подотчётный период дефектов с их краткими описаниями и важностью.
- **Статистика по всем дефектам** (overall defects statistics). Таблица, в которой представлены данные по обнаруженным за всё время существования проекта дефектам (с классификацией по стадии жизненного цикла и важности). Как правило, в этот же раздел добавляется график, отражающий такие статистические данные.
- **Рекомендации** (recommendations). Обоснованные выводы и рекомендации по принятию тех или иных управленческих решений (изменению тест-плана, запросу или освобождению ресурсов и т. д.) Здесь этой информации можно отвести больше места, чем в кратком описании (summary), сделав акцент именно на том, что и почему рекомендуется сделать в имеющейся ситуации.
- **Приложения** (appendixes). Фактические данные (как правило, значения метрик и графическое представление их изменения во времени).

Логика построения отчёта о результатах тестирования

Для того чтобы отчёт о результатах тестирования был действительно полезным, при его создании следует постоянно помнить об универсальной логике отчётности (рисунок 6.b), особенно актуальной для таких разделов отчёта о результатах тестирования, как краткое описание (summary) и рекомендации (recommendations):

- Выводы строятся на основе целей (которые были отражены в плане).
- Выводы дополняются рекомендациями.
- Как выводы, так и рекомендации строго обосновываются.
- Обоснование опирается на объективные факты.

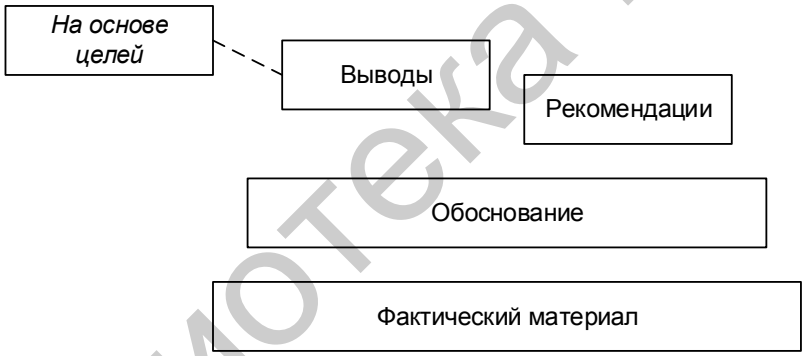


Рисунок 6.b – Универсальная логика отчётности

Выводы должны быть:

- **Краткими.** Сравните примеры выводов в отчёте о результатах тестирования, представленные в таблице 6.d.

Таблица 6.d – Примеры выводов в отчете о результатах тестирования с точки зрения краткости

Плохо	Хорошо
1.17. Как показал глубокий анализ протоколов о выполнении тестирования, можно сделать достаточно уверенные выводы о том, что основная часть функций, отмеченных заказчиком как наиболее важные, функционирует в рамках допустимых отклонений от согласованных на последнем обсуждении с заказчиком метрик качества	1.11. Базовая функциональность полностью работоспособна (см. 2.1–2.2). 1.23. Существуют некритические проблемы с детализацией сообщений в файле журнала (см. 2.3–2.4). 1.28. Тестирование приложения под ОС Linux не удалось провести из-за недоступности сервера SR-85 (см. 2.5)

- **Информативными.** Сравните примеры выводов в отчёте о результатах тестирования, представленные в таблице 6.e.

Таблица 6.e – Примеры выводов в отчете о результатах тестирования с точки зрения информативности

Плохо	Хорошо
1.8. Результаты обработки файлов с множественными кодировками, представленными в сопоставимых пропорциях, оставляют желать лучшего. 1.9. Приложение не запускается при некоторых значениях параметров командной строки. 1.10. Непонятно, что происходит с анализом изменения содержимого входного каталога	1.8. Обнаружены серьёзные проблемы с библиотекой распознавания кодировок (см. BR 834). 1.9. Нарушена функциональность анализа параметров командной строки (см. BR 745, BR 877, BR 878). 1.10. Выявлена нестабильность в работе модуля «Сканер», проводятся дополнительные исследования

- **Полезными** для читающего отчёт. Сравните примеры выводов в отчете о результатах тестирования, предаставленные в таблице 6.f.

Таблица 6.f – Примеры выводов в отчете о результатах тестирования с точки зрения полезности

Плохо	Хорошо
1.18. Некоторые тесты прошли на удивление хорошо. 1.19. В процессе тестирования мы не испытывали сложности с настройкой среды автоматизации. 1.20. По сравнению с результатами, которые были получены вчера, ситуация немного улучшилась. 1.21. С качеством по-прежнему есть некоторые проблемы. 1.22. Часть команды была в отпуске, но мы всё равно справились	Представленного в колонке «Плохо» просто не должно быть в отчёте!

Рекомендации должны быть:

- **Краткими**. Да, мы снова говорим о краткости, т. к. её отсутствием страдает слишком большое количество документов. Сравните примеры рекомендаций в отчете о результатах тестирования, представленные в таблице 6.g.

Таблица 6.g – Примеры рекомендаций в отчете о результатах тестирования с точки зрения краткости

Плохо	Хорошо
2.98. Мы рекомендуем рассмотреть возможные варианты исправления данной ситуации в контексте поиска оптимального решения при условии минимизации усилий разработчиков и максимального повышения соответствия приложения заявленным критериям качества, а именно: исследовать возможность замены некоторых библиотек их более качественными аналогами	2.98. Необходимо изменить способ определения кодировки текста в документе. Возможные решения: • [сложно, надёжно, но очень долго] написать собственное решение; • [требует дополнительного исследования и согласования] заменить проблемную библиотеку «cflk_n_r_coding» аналогом (возможно, коммерческим)

- **Реально выполнимыми**. Сравните примеры рекомендаций в отчете о результатах тестирования, представленные в таблице 6.h.

Таблица 6.h – Примеры рекомендаций в отчете о результатах тестирования с точки зрения выполнимости

Плохо	Хорошо
2.107. Использовать механизм обработки слов, аналогичный используемому в Google. 2.304. Не загружать в оперативную память информацию о файлах во входном каталоге. 2.402. Полностью переписать проект без использования внешних библиотек	2.107. Реализовать алгоритм приведения слов русского языка к именительному падежу (см. описание по ссылке …) 2.304. Увеличить размер доступной скрипту оперативной памяти на 40–50 % (в идеале – до 512 Мбайт). 2.402. Заменить собственными решениями функции анализа содержимого каталога и параметров файлов библиотеки «cflk_n_r_flstm»

- **Дающими как понимание того, что надо сделать, так и некоторое пространство для принятия собственных решений**. Сравните примеры рекомендаций в отчёте о результатах тестировани, представленные в таблице 6.i.

Таблица 6.i – Примеры рекомендаций в отчете о результатах тестирования с альтернативными решениями проблемы

Плохо	Хорошо
2.212. Рекомендуем поискать варианты решения этого вопроса.	2.212. Возможные варианты решения: а) … ; б) [рекомендуем!] … ; в) …
2.245. Использовать только дисковую сортировку.	2.245. Добавить функциональность определения оптимального метода сортировки в зависимости от количества доступной оперативной памяти.
2.278. Исключить возможность передачи некорректных имён файла журнала через параметр командной строки	2.278. Добавить фильтрацию имени файла журнала, получаемого через параметр командной строки, с помощью регулярного выражения

Обоснование выводов и рекомендаций – промежуточное звено между предельно сжатыми результатами анализа и огромным количеством фактических данных. Оно даёт ответы на вопросы наподобие:

- «Почему мы так считаем?»
- «Неужели это так?!»
- «Где взять дополнительные данные?»

Сравните (таблица 6.j).

Таблица 6.j – Примеры обоснований выводов и рекомендаций в отчете о результатах тестирования

Плохо	Хорошо
4.107. Покрытие требований тест-кейсами достаточно	4.107. Покрытие требований тест-кейсами вышло на достаточный уровень (значение R^C составило 63 % при заявленном минимуме 60 % для текущей стадии проекта)
4.304. Необходимо больше усилий направить на регрессионное тестирование	4.304. Необходимо больше усилий направить на регрессионное тестирование, т. к. две предыдущих итерации выявили 21 дефект высокой важности (см. список в 5.43) в функциональности, в которой ранее не обнаруживалось проблем
4.402. От сокращения сроков разработки стоит отказаться	4.402. От сокращения сроков разработки стоит отказаться, т. к. текущее опережение графика на 30 человеко-часов может быть легко поглощено на стадии реализации требований R84* и R89*

Фактический материал содержит самые разнообразные данные, полученные в процессе тестирования. Сюда могут относиться отчёты о дефектах, журналы работы средств автоматизации, созданные различными приложениями наборы файлов и т. д. Как правило, к отчёту о результатах тестирования прилагаются лишь сокращённые агрегированные выборки подобных данных (если это возможно), а также приводятся ссылки на соответствующие документы, разделы системы управления проектом, пути в хранилище данных и т. д.

На этом мы завершаем теоретическое рассмотрение отчётности и переходим к примеру – учебному отчёту о результатах тестирования нашего приложения «Конвертер файлов». Напомним, что приложение является предельно простым, потому и отчёт о результатах тестирования будет очень маленьким.

ПРИМЕР ОТЧЁТА О РЕЗУЛЬТАТАХ ТЕСТИРОВАНИЯ

Для того чтобы заполнить некоторые части отчёта, нам придётся сделать допущения о текущем моменте развития проекта и сложившейся ситуации с качеством. Поскольку данный отчёт находится внутри текста книги, у него нет таких типичных частей, как обложка, содержание и т. п.

Краткое описание. За период 26–28 мая было выпущено четыре билда, на последнем из которых успешно прошло 100 % тест-кейсов дымового тестирования и 76 % тест-кейсов тестирования критического пути. 98 % требований высокой важности реализовано корректно. Метрики качества находятся в зелёной зоне, потому есть все основания рассчитывать на завершение проекта в срок (на текущий момент реальный прогресс в точности соответствует плану). На следующую итерацию (29 мая) запланировано выполнение оставшихся низкоприоритетных тест-кейсов.

Команда тестировщиков представлена в таблице 6.k.

Таблица 6.k – Описание команды тестировщиков в отчете о результатах тестирования

Имя	Должность	Роль
Джо Блэк	Тестировщик	Ответственный за обеспечение качества
Джим Уайт	Старший разработчик	Ответственный за парное тестирование и аудит кода

Описание процесса тестирования. Каждый из четырёх выпущенных за подотчётный период билдов (3–6) был протестирован под ОС Windows 7 Ent x64 и ОС Linux Ubuntu 14 LTS x64 в среде исполнения PHP 5.6.0. Дымовое тестирование (см. http://projects/FC/Testing/SmokeTest) выполнялось с использованием автоматизации на основе командных файлов (см. http:\\PROJECTS\FC\Testing\Aut\Scripts). Тестирование критического пути (см. http://projects/FC/Testing/CriticalPathTest) выполнялось вручную. Регрессионное тестирование показало высокую стабильность функциональности (обнаружен только один дефект с важностью «средняя»), а повторное тестирование показало ощутимый прирост качества (исправлено 83 % обнаруженных ранее дефектов).

Расписание представлено в таблице 6.l.

Таблица 6.l – Пример расписания в отчете о результатах тестирования

Имя	Дата	Деятельность	Продолжительность, ч
Джо Блэк	27.05.2015	Разработка тест-кейсов	2
Джо Блэк	27.05.2015	Парное тестирование	2
Джо Блэк	27.05.2015	Автоматизация дымового тестирования	1
Джо Блэк	27.05.2015	Написание отчётов о дефектах	2
Джим Уайт	27.05.2015	Аудит кода	1
Джим Уайт	27.05.2015	Парное тестирование	2
Джо Блэк	28.05.2015	Разработка тест-кейсов	3
Джо Блэк	28.05.2015	Парное тестирование	1
Джо Блэк	28.05.2015	Написание отчётов о дефектах	2
Джо Блэк	28.05.2015	Написание отчёта о результатах тестирования	1
Джим Уайт	28.05.2015	Аудит кода	1
Джим Уайт	28.05.2015	Парное тестирование	1

Статистика по новым дефектам представлена в таблице 6.m.

Таблица 6.m – Пример статистики по новым дефектам

Статус	Количество	Важность			
		низкая	средняя	высокая	критическая
Найдено	23	2	12	7	2
Исправлено	17	0	9	6	2
Проверено	13	0	5	6	2
Открыто заново	1	0	0	1	0
Отклонено	3	0	2	1	0

Список новых дефектов представлен в таблице 6.n.

Таблица 6.n – Пример списка новых дефектов

Идентификатор	Важность	Описание
BR 21	Высокая	Приложение не различает файлы и символические ссылки на файлы
BR 22	Критическая	Приложение игнорирует файлы .md во входном каталоге
И так далее (описание всех 23 найденных дефектов)		

Статистика по всем дефектам представлена в таблице 6.o.

Таблица 6.o – Пример статистики по всем дефектам

Статус	Количество	Важность			
		низкая	средняя	высокая	критическая
Найдено	34	5	18	8	3
Исправлено	25	3	12	7	3
Проверено	17	0	7	7	3
Открыто заново	1	0	0	1	0
Отклонено	4	0	3	1	0

График статистики по всем дефекта приведен на рисунке 6.c, график изменения значений метрик – на рисунке 6.d.

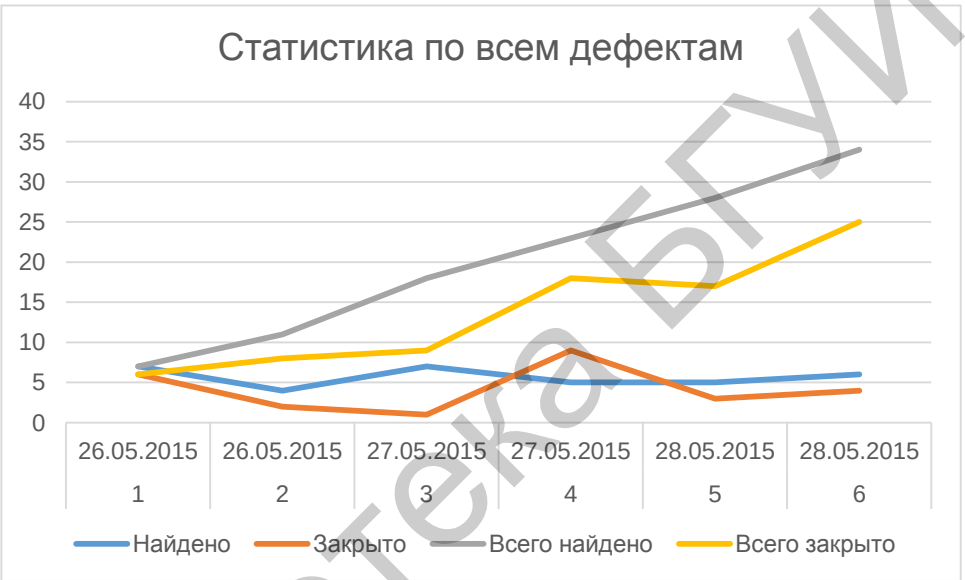


Рисунок 6.c – График статистики по всем дефектам

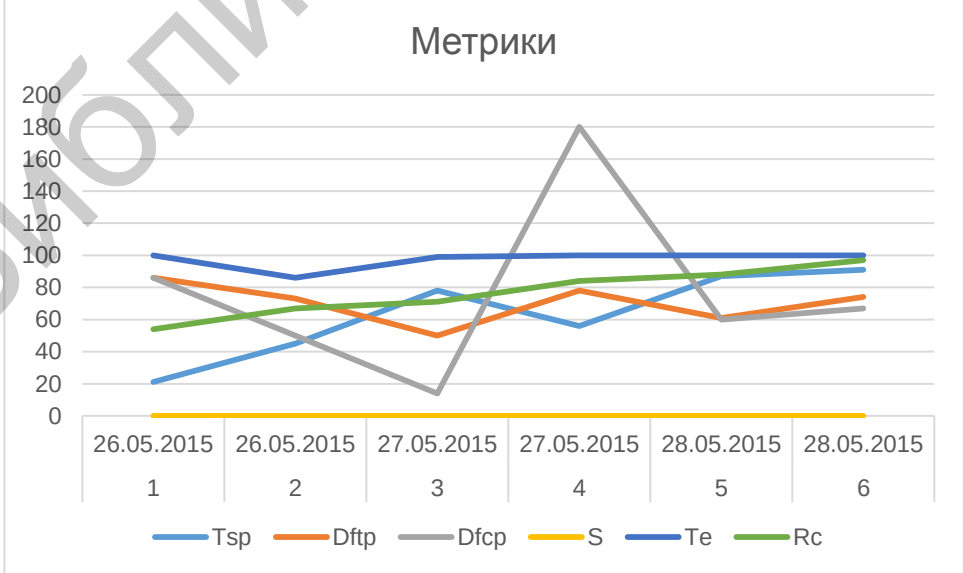


Рисунок 6.d – График изменения значений метрик

Задание 6.b: Найдите в сети Интернет более развёрнутые примеры отчётов о результатах тестирования. Они периодически появляются, но и настолько же быстро удаляются, т. к. настоящие (неучебные) отчёты, как правило, являются конфиденциальной информацией.

6.3 Оценка трудозатрат

В завершение данного раздела мы снова обращаемся к планированию, но уже куда более простому – к оценке трудозатрат.

!!! **Трудозатраты** (man-hours[326]) – количество рабочего времени, необходимого для выполнения работы (выражается в человеко-часах).

Каждый раз, когда вы получаете задание или выдаёте кому-то задание, явно или неявно возникают вопросы наподобие следующих:

- Как много времени понадобится на выполнение работы?
- Когда всё будет готово?
- Можно ли гарантированно выполнить работу к такому-то сроку?
- Каковы наиболее оптимистичный и пессимистичный прогнозы по времени?

Рассмотрим несколько соображений относительно того, как производится оценка трудозатрат.

Любая оценка лучше её отсутствия. Даже если область предстоящей работы для вас совершенно нова, даже если вы ошибётесь в своей оценке на порядок, вы как минимум получите опыт, который сможете использовать в будущем при возникновении подобного рода задач.

Оптимизм губителен. Как правило, люди склонны недооценивать сложность незнакомых задач, что приводит к занижению оценки трудозатрат.

Но даже при достаточно точном определении самих трудозатрат люди без опыта выполнения оценки склонны рассматривать предстоящую работу как некоторую изолированную деятельность, забывая о том, что на протяжении любого рабочего дня «чистую производительность труда» будут снижать такие факторы, как переписка по почте, участие в собраниях и обсуждениях, решение сопутствующих технических вопросов, изучение документации и обдумывание сложных частей задачи, форс-мажорные обстоятельства (неотложные дела, проблемы с техникой и т. д.).

Таким образом, обязательно стоит учитывать, что в действительности вы сможете заниматься поставленной задачей не 100 % рабочего времени, а меньше (насколько меньше – зависит от конкретной ситуации, в среднем принято считать, что на поставленную задачу из каждых восьми рабочих часов вы сможете потратить не более шести). Учитывая этот факт, стоит сделать соответствующие поправки в оценке общего времени, которое понадобится на выполнение работы (а именно оно чаще всего интересует постановщика задачи).

Оценка должна быть аргументирована. Это не значит, что вы всегда должны пускаться в подробные пояснения, но следует быть готовым объяснить, почему вы считаете, что та или иная работа займёт именно столько времени. Во-первых, продумывая эти аргументы, вы получаете дополнительную возможность лучше оценить предстоящую работу и скорректировать оценку. Во-вторых, если ваша оценка не соответствует ожиданиям постановщика задачи, вы сможете отстоять свою точку зрения.

[326] **Man-hour.** A unit for measuring work in industry, equal to the work done by one man in one hour. URL: http://dictionary.reference.com/browse/man-hour.

Простой способ научиться оценивать – оценивать. В специализированной литературе (см. ниже небольшой список) приводится множество технологий, но первична сама привычка выполнять оценку предстоящей работы. В процессе выработки этой привычки вы естественным образом встретитесь с большинством типичных проблем и через некоторое время научитесь делать соответствующие поправки в оценке, даже не задумываясь.

Что оценивать? Что угодно. Сколько времени у вас уйдёт на прочтение новой книги. За сколько времени вы доедете домой новым маршрутом. За сколько времени вы напишете курсовую или дипломную работу. И так далее. Не важно, что именно вы оцениваете, важно, что вы повторяете это раз за разом, учитывая накапливающийся опыт.

Если вас заинтересовал профессиональный подход к формированию оценки трудозатрат, то рекомендуем ознакомиться со следующими источниками:

- Brooks F. «The Mythical Man Month»;
- Marco T. «Controlling Software Projects»;
- Conte S. «Software engineering metrics and models».

Алгоритм обучения формированию оценки:

- Сформируйте оценку. Ранее уже было отмечено, что нет ничего страшного в том, что полученное значение может оказаться очень далёким от действительности. Для начала оно просто должно быть.
- Запишите полученную оценку. Обязательно именно запишите. Это застрахует вас как минимум от двух рисков: забыть полученное значение (особенно, если работа заняла много времени), обмануть себя: «Ну, я как-то примерно так и думал».
- Выполните работу. В отдельных случаях люди склонны подстраиваться под заранее сформированную оценку, ускоряя или замедляя выполнение работы, – это тоже полезный навык, но сейчас такое поведение будет мешать. Однако если вы будете тренироваться на десятках и сотнях различных задач, вы физически не сможете «подстроиться» под каждую из них и начнёте получать реальные результаты.
- Сверьте реальные результаты с ранее сформированной оценкой.
- Учтите ошибки при формировании новых оценок. На этом этапе очень полезно не просто отметить отклонение, а подумать, что привело к его появлению.
- Повторяйте этот алгоритм как можно чаще для самых различных областей жизни. Сейчас цена ваших ошибок крайне мала, а наработанный опыт от этого становится ничуть не менее ценным.

Полезные идеи по формированию оценки трудозатрат:

- Добавляйте небольшой «буфер» (по времени, бюджету или иным критическим ресурсам) на непредвиденные обстоятельства. Чем более дальний прогноз вы строите, тем большим может быть этот «буфер» – от 5–10 % до 30–40 %. Но ни в коем случае не стоит осознанно завышать оценку в разы.
- Выясните свой «коэффициент искажения»: большинство людей в силу особенности своего мышления склонны постоянно или занижать, или завышать оценку. Многократно формируя оценку трудозатрат и сравнивая её впоследствии с действительностью, вы можете заметить определённую закономерность, которую вполне можно выразить числом. Например, может оказаться, что вы склонны занижать оценку в 1,3 раза. Попробуйте в следующий раз внести соответствующую поправку.

- Принимайте во внимание не зависящие от вас обстоятельства. Например, вы точно уверены, что выполните тестирование очередного билда за N человекочасов, вы учли все отвлекающие факторы и т. д. и решили, что точно закончите к конкретной дате. А потом в действительности выпуск билда задерживается на два дня, и ваш прогноз о моменте завершения работы оказывается неправдоподобным.
- Задумывайтесь заранее о необходимых ресурсах. Так, например, необходимую инфраструктуру можно (и нужно!) подготовить (или заказать) заранее, т. к. на подобные вспомогательные задачи может быть потрачено много времени, к тому же основная работа часто не может быть начата, пока не будут завершены все приготовления.
- Ищите способы организовать параллельное выполнение задач. Даже если вы работаете один, всё равно какие-то задачи можно и нужно выполнять параллельно (например, уточнение тест-плана, пока происходит разворачивание виртуальных машин). В случае если работа выполняется несколькими людьми, распараллеливание работы можно считать жизненной необходимостью.
- Периодически сверяйтесь с планом, вносите корректировки в оценку и уведомляйте заинтересованных лиц о внесённых изменениях заблаговременно. Например, вы поняли (как в упомянутом выше примере с задержкой билда), что завершите работу как минимум на два дня позже. Если вы оповестите проектную команду немедленно, у ваших коллег появляется шанс скорректировать свои собственные планы. Если же вы в «час икс» преподнесёте сюрприз о сдвигах срока на два дня, то создадите коллегам объективную проблему.
- Используйте инструментальные средства от электронных календарей до возможностей вашей системы управления проектом – это позволит вам как минимум не держать в памяти кучу мелочей, а как максимум – повысит точность формируемой оценки.

Оценка с использованием структурной декомпозиции.

....	С другими техниками формирования оценки вы можете ознакомиться в следующих литературных источниках: • Rubin K. «Essential Scrum»; • Cohn M. «Agile Estimating and Planning»; • Beck K. «Extreme programming explained: Embrace change»; • PMBOK (Project Management Body of Knowledge); • Краткий перечень основных техник с пояснениями можно посмотреть в статье «Software Estimation Techniques – Common Test Estimation Techniques used in SDLC[327]».

!!!	**Структурная декомпозиция** (work breakdown structure, WBS[328]) – иерархическая декомпозиция объёмных задач на всё более и более малые подзадачи с целью упрощения оценки, планирования и мониторинга выполнения работы.

[327] Software Estimation Techniques – Common Test Estimation Techniques used in SDLC. URL: http://www.softwaretestingclass.com/software-estimation-techniques/.

[328] The WBS is a deliverable-oriented hierarchical decomposition of the work to be executed by the project team, to accomplish the project objectives and create the required deliverables. The WBS organizes and defines the total scope of the project. The WBS subdivides the project work into smaller, more manageable pieces of work, with each descending level of the WBS representing an increasingly detailed definition of the project work. The planned work contained within the lowest-level WBS components, which are called work packages, can be scheduled, cost estimated, monitored, and controlled. PMBOK. – 3rd ed.

В процессе выполнения структурной декомпозиции большие задачи делятся на всё более и более мелкие подзадачи, что позволяет нам:

- описать весь объём работ с точностью, достаточной для чёткого понимания сути задач, формирования достаточно точной оценки трудозатрат и выработки показателей достижения результатов;
- определить весь объём трудозатрат как сумму трудозатрат по отдельным задачам (с учётом необходимых поправок);
- от интуитивного представления перейти к конкретному перечню отдельных действий, что упрощает построение плана, принятие решений о распараллеливании работ и т. д.

Сейчас мы рассмотрим применение структурной декомпозиции в сочетании с упрощённым взглядом на оценку трудозатрат на основе требований и тест-кейсов.

С подробной теорией по данному вопросу можно ознакомиться в следующих статьях:

- Nageswaran S. «Test Effort Estimation Using Use Case Points[329]»;
- Patel N. «Test Case Point Analysis[330]».

Если абстрагироваться от научного подхода и формул, то суть такой оценки сводится к следующим шагам:

- декомпозиции требований до уровня, на котором появляется возможность создания качественных чек-листов;
- декомпозиции задач по тестированию каждого пункта чек-листа до уровня «тестировщицких действий» (создание тест-кейсов, выполнение тест-кейсов, создание отчётов о дефектах и т. д.);
- выполнению оценки с учётом собственной производительности.

Рассмотрим этот подход на примере тестирования требования ДС-2.4 (см. подраздел 2.7 «Пример анализа и тестирования требований»): «При указании неверного значения любого из параметров командной строки приложение должно завершить работу, выдав сообщение об использовании (ДС-3.1), а также сообщив имя неверно указанного параметра, его значение и суть ошибки (см. ДС-3.2)».

Это требование само по себе является низкоуровневым и почти не требует декомпозиции, но чтобы проиллюстрировать суть подхода, проведём разделение требования на составляющие:

- Если все три параметра командной строки указаны верно, сообщение об ошибке не выдаётся.
- Если указано неверно от одного до трёх параметров, то выдаётся сообщение об использовании, имя (или имена) неверно указанного параметра и неверное значение, а также сообщение об ошибке:
  - если неверно указан SOURCE_DIR или DESTINATION_DIR: «Directory not exists or inaccessible»;
  - если DESTINATION_DIR находится в SOURCE_DIR: «Destination dir may not reside within source dir tree»;
  - если неверно указан LOG_FILE_NAME): «Wrong file name or inaccessible path».

[329] Test Effort Estimation Using Use Case Points. S. Nageswaran. URL: http://www.bfpug.com.br/Artigos/UCP/Nageswaran-Test_Effort_Estimation_Using_UCP.pdf.

[330] Test Case Point Analysis. N. Patel. URL: http://www.stickyminds.com/sites/default/files/article/file/2013/XUS373692file1_0.pdf.

Создадим чек-лист и здесь же пропишем **примерное** количество тест-кейсов на каждый пункт из предположения, что мы будем проводить достаточно глубокое тестирование этого требования:

- Все параметры корректные {1 тест-кейс}.
- Несуществующий/некорректный путь для:
  - SOURCE_DIR {3 тест-кейса};
  - DESTINATION_DIR {3 тест-кейса}.
- Недопустимое имя файла LOG_FILE_NAME {3 тест-кейса}.
- Значения SOURCE_DIR и DESTINATION_DIR являются корректными именами существующих каталогов, но DESTINATION_DIR находится внутри SOURCE_DIR {3 тест-кейса}.
- Недопустимые/несуществующие имена объектов ФС указаны в более чем одном параметре {5 тест-кейсов}.
- Значения SOURCE_DIR и DESTINATION_DIR не являются корректными/существующими именами каталогов, и при этом DESTINATION_DIR находится внутри SOURCE_DIR {3 тест-кейса}.

У нас получилось примерно 22 тест-кейса. Также давайте для большей показательности примера предположим, что часть тест-кейсов (например, 5) уже была создана ранее.

Теперь сведём полученные данные в таблицу 6.р, где также отразим количество проходов. Этот показатель появляется из соображения, что некоторые тест-кейсы будут находить дефекты, что потребует повторного выполнения тест-кейса для верификации исправления дефекта; в некоторых случаях дефекты будут открыты заново, что потребует повторной верификации. Это относится лишь к части тест-кейсов, потому количество проходов может быть дробным, чтобы оценка была более точной.

Количество проходов для тестирования новой функциональности в общем случае можно грубо оценивать так:

- простая функциональность: 1–1.5 (не все тесты повторяются);
- функциональность средней сложности: 2;
- сложная функциональность: 3–5.

Таблица 6.р – Оценка количества создаваемых и выполняемых тест-кейсов

	Создание	**Выполнение**
Количество	12	22
Повторения (проходы)	1	1.2
Общее количество	12	26.4
Время на один тест-кейс		
Итоговое время		

Осталось заполнить ячейки со значениями времени, необходимого на разработку и выполнение одного тест-кейса. К сожалению, не существует никаких магических способов выяснения этих параметров – только накопленный опыт о вашей собственной производительности, на которую среди прочего влияют следующие факторы (по каждому из них можно вводить коэффициенты, уточняющие оценку):

- ваш профессионализм и опыт;
- сложность и объёмность тест-кейсов;
- производительность тестируемого приложения и тестового окружения;
- вид тестирования;
- наличие и удобство средств автоматизации;
- стадия разработки проекта.

Тем не менее существует простой способ получения интегральной оценки вашей собственной производительности, при котором влиянием этих факторов можно пренебречь: нужно измерять свою производительность на протяжении длительного периода времени и фиксировать, сколько создать и выполнить тест-кейсов вы можете в час, день, неделю, месяц и т. д. Чем больший промежуток времени будет рассмотрен, тем меньше на результаты измерений будут влиять кратковременные отвлекающие факторы, появление которых сложно предсказывать.

Допустим, что для некоего тестировщика эти значения оказались следующими – за месяц (28 рабочих дней) ему удаётся:

- создать 300 тест-кейсов (примерно 11 тест-кейсов в день, или 1,4 в час);
- выполнить 1000 тест-кейсов (примерно 36 тест-кейсов в день, или 4,5 в час).

Подставим полученные значения в таблицу 6.p и получим таблицу 6.q.

Таблица 6.q – Оценка трудозатрат

	Создание	**Выполнение**
Количество	12	22
Повторения (проходы)	1	1,2
Общее количество	12	26,4
Время на один тест-кейс, ч	0,7	0,2
Итоговое время, ч	8,4	5,2
ИТОГО	**13,6 ч**	

Если бы оценка производительности нашего тестировщика производилась на коротких отрезках времени, полученное значение нельзя было бы использовать напрямую, т. к. в нём не было бы учтено время на написание отчётов о дефектах, участие в различных собраниях, переписку и прочие виды деятельности. Но мы потому и ориентировались на итоги измерений за месяц, что за 28 типичных рабочих дней все эти факторы многократно проявляли себя, и их влияние уже учтено в оценке производительности.

Если бы мы всё же опирались на краткосрочные исследования, можно было бы ввести дополнительный коэффициент или использовать допущение о том, что работе с тест-кейсами за один день удаётся посвятить не 8 ч, а меньше (например, 6).

Итого у нас получилось 13,6 ч, или 1,7 рабочих дня. Учитывая идею о закладке небольшого «буфера», можем считать, что за два полных рабочих дня наш тестировщик точно справится с поставленной задачей.

В заключение этого раздела ещё раз отметим, что для уточнения собственной производительности и улучшения своих навыков оценки трудозатрат стоит формировать оценку, после чего выполнять работу и сравнивать фактический результат с оценкой. И повторять эту последовательность шагов раз за разом.

Задание 6.c: Разработайте на основе итогового чек-листа, представленного в подразделе 2.4, тест-кейсы и оцените свою производительность в процессе выполнения этой задачи.

6.4 Контрольные вопросы и задания

- Что такое тестовый план?
- Каковы основные задачи планирования тестовых испытаний и составления тестового плана?

- Назовите и охарактеризуйте основные разделы тестового плана.
- Что представляет собой тестовая стратегия?
- Какие существуют виды критериев в тестовом плане?
- Что описывают в разделе «Ресурсы» тестового плана?
- Что такое метрика?
- Приведите примеры прямых и расчетных метрик.
- Как выполняется оценка трудозатрат при планировании тестирования?
- Какие полезные советы по формированию оценки трудозатрат вы запомнили?
- Что содержит отчет о результатах тестирования?
- Кому нужен отчет о результатах тестирования?
- Какие рекомендации по составлению качетсвенного отчета о результатах тестирования вы запомнили?

7 ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ РАЗЛИЧНЫХ ТЕХНИК ТЕСТИРОВАНИЯ

7.1 Позитивные и негативные тест-кейсы

Ранее мы уже рассматривали (см. подраздел 4.7 «Логика создания эффективных проверок») алгоритм продумывания идей тест-кейсов, в котором предлагается ответить себе на следующие вопросы относительно тестируемого объекта:

- Что перед вами?
- Кому и зачем тестируемый объект нужен (и насколько это важно)?
- Как тестируемый объект обычно используется?
- Как тестируемый объект может «сломаться», т. е. начать работать неверно?

Сейчас мы применим этот алгоритм, сконцентрировавшись на двух последних вопросах, т. к. именно ответы на них позволяют нам придумать много позитивных и негативных тест-кейсов. Продолжим тестировать наш «Конвертер файлов» (см. подраздел 2.7 «Пример анализа и тестирования требований»), выбрав для исследования первый параметр командной строки – SOURCE_DIR – имя каталога, в котором приложение ищет файлы, подлежащие конвертации.

Что перед нами? Путь к каталогу. Казалось бы, просто. Но стоит вспомнить, что наше приложение должно работать как минимум под управлением Windows и Linux, что приводит к необходимости освежить в памяти принципы работы файловых систем в этих ОС. А ещё может понадобиться работа с сетью.

Кому и зачем тестируемый объект нужен (и насколько это важно)? Конечные пользователи не занимаются конфигурированием приложения, т. е. этот параметр нужен администратору (предположительно, это человек квалифицированный и не допустит абсурда, но из его же квалификации вытекает возможность придумать такие варианты использования, не присущие рядовому пользователю). Важность параметра критическая, т. к. при каких-то проблемах с ним есть риск полной потери работоспособности приложения.

Как тестируемый объект обычно используется? Здесь нам понадобится понимание принципов работы файловых систем.

- Корректное имя существующего каталога:
  - Windows:
    - X:\dir
    - “X:\dir with spaces”
    - .\dir
    - ..\dir
    - \\host\dir
    - Всё вышеперечисленное с “\” в конце пути.
    - X:\
  - Linux:
    - /dir
    - “/dir with spaces”
    - host:/dir
    - smb://host/dir
    - ./dir
    - ../dir
    - Всё вышеперечисленное с “/” в конце пути.
    - /

И всё, т. е. в данном конкретном случае существует единственный вариант верного использования первого параметра – указать корректное имя существующего каталога (пусть вариантов таких корректных имён и много). Фактически мы полу-

чили чек-лист для позитивного тестирования. Все ли варианты допустимых имён мы учли? Может быть, и не все. Но эту проблему мы рассмотрим в следующем разделе, посвящённому классам эквивалентности и граничным условиям (см. подраздел 7.2 «Классы эквивалентности и граничные условия»).

В настоящий момент важно ещё раз подчеркнуть мысль о том, что сначала мы проверяем работу приложения на позитивных тест-кейсах, т. е. в корректных условиях. Если эти проверки не пройдут, в некоторых совершенно допустимых и типичных ситуациях приложение будет неработоспособным, т. е. ущерб качеству будет весьма ощутимым.

Как тестируемый объект может «сломаться», т. е. начать работать неверно? Негативных тест-кейсов (за редчайшим исключением) оказывается намного больше, чем позитивных. Итак, какие проблемы с именем каталога-источника (и самим каталогом-источником) могут помешать нашему приложению?

- Указанный путь не является корректным именем каталога:
  - пустое значение (“”);
  - слишком длинное имя:
    - для Windows: более 256 символов. (Важно! Путь к каталогу длиной в 256 символов допустим, но надо учесть ограничение на полное имя файла, т. к. его превышение может быть достигнуто естественным образом, что приведёт к возникновению сбоя.);
    - для Linux: более 4096 байт;
  - недопустимые символы, например: ? < > \ * | " \0;
  - недопустимые комбинации допустимых символов, например: “....\dir”.
- Каталог не существует:
  - на локальном диске;
  - в сети.
- Каталог существует, но к нему нет прав доступа.
- Доступ к каталогу утерян после запуска приложения:
  - каталог удалён или переименован;
  - изменены права доступа;
  - потеря соединения с удалённым компьютером.
- Использование зарезервированного имени:
  - для Windows: com1-com9, lpt1-lpt9, con, nul, prn;
  - для Linux: “..”.
- Проблемы с кодировками, например: имя указано верно, но не в той кодировке.

Если погружаться в детали поведения отдельных операционных систем и файловых систем, данный список можно значительно расширить. И тем не менее открытыми будут оставаться два вопроса:

- Все ли эти варианты надо проверить?
- Не упустили ли мы что-то важное?

На первый вопрос ответ можно найти, опираясь на рассуждения, описанные в подразделе 4.7 «Логика создания эффективных проверок». Ответ на второй вопрос помогут найти рассуждения, описанные в двух следующих подразделах, т. к. классы эквивалентности, граничные условия и доменное тестирование значительно упрощают решение подобных задач.

Задание 7.а: Как вы думаете, почему в вышеприведённых чек-листах мы не учли требование о том, что SOURCE_DIR не может содержать внутри себя DESTINATION_DIR?

7.2 Классы эквивалентности и граничные условия

В данном разделе мы рассмотрим примеры упомянутых ранее техник тестирования на основе классов эквивалентности и граничных условий (см. подраздел 3.2 «Подробная классификация тестирования»). Если уточнить определения, получается:

!!!	**Класс эквивалентности** (equivalence class[331]) – набор данных, обрабатываемых одинаковым образом и приводящих к одинаковому результату.

!!!	**Граничное условие** (border condition, boundary condition[332]) – значение, находящееся на границе классов эквивалентности.

....	Иногда под классом эквивалентности понимают набор тест-кейсов, полное выполнение которого является избыточным. Это определение не противоречит предыдущему, т. к. показывает ту же ситуацию, но с другой точки зрения.

В качестве пояснения идеи рассмотрим тривиальный пример. Допустим, нам нужно протестировать функцию, которая определяет, корректное или некорректное имя ввёл пользователь при регистрации.

Требования к имени пользователя таковы:

- От трёх до двадцати символов включительно.
- Допускаются цифры, знак подчёркивания, буквы английского алфавита в верхнем и нижнем регистрах.

Если попытаться решить задачу «в лоб», нам для позитивного тестирования придётся перебрать все комбинации допустимых символов длиной [3, 20] (это 18-разрядное 63-ричное число, т. е. 2.4441614509104E+32). А негативных тест-кейсов здесь и вовсе будет бесконечное количество, ведь мы можем проверить строку длиной в 21 символ: 100, 10 000, 1 000 000, 1 000 000 000 и т. д.

Представим графически классы эквивалентности относительно требований к длине (рисунок 7.a).

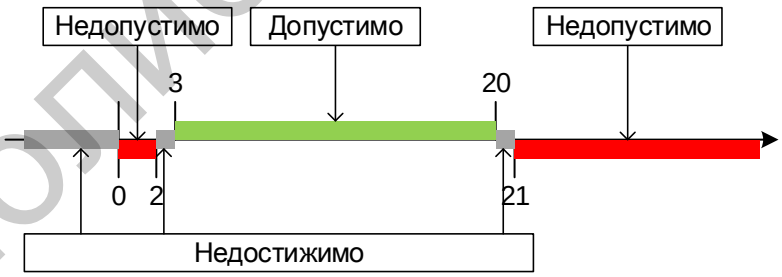


Рисунок 7.a – Классы эквивалентности для значений длины имени пользователя

Поскольку для длины строки невозможны дробные и отрицательные значения, мы видим три недостижимые области, которые можно исключить, и получаем окончательный вариант (рисунок 7.b).

Мы получили три класса эквивалентности:

[331] An **equivalence class** consists of a set of data that is treated the same by a module or that should produce the same result. Copeland L. A practitioner's guide to software test design.

[332] The **boundaries** – the «edges» of each equivalence class. Copeland L. A practitioner's guide to software test design.

- [0, 2] – недопустимая длина;
- [3, 20] – допустимая длина;
- [21, ∞] – недопустимая длина.

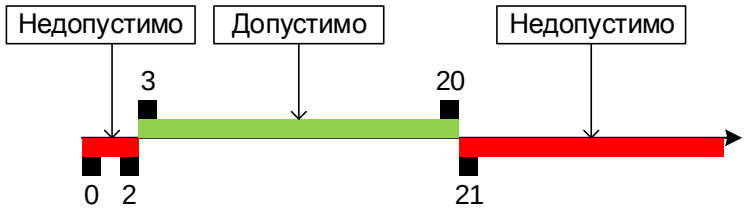


Рисунок 7.b – Итоговое разбиение на классы эквивалентности значений длины имени пользователя

Обратите внимание, что области значений [0, 2] и [21, ∞] относятся к разным классам эквивалентности, т. к. принадлежность длины строки к этим диапазонам проверяется отдельными условиями на уровне кода программы.

Граничные условия уже отмечены на рисунке 7.b – это 2, 3, 20 и 21. Значение нуль тоже на всякий случай стоит включить в этот набор, т. к. в программировании нуль, NULL, нулевой байт и т. п. исторически являются «опасными значениями».

В итоге мы получаем следующий набор входных данных для тест-кейсов (сами символы для составления строк можно выбирать из набора допустимых символов случайным образом, но желательно учесть все типы символов, т. е. буквы в обоих регистрах, цифры, знак подчёркивания) (таблица 7.a).

Таблица 7.a – Значения входных данных для тест-кейсов (реакция на длину имени пользователя)

Критерии	**Позитивные тест-кейсы**		**Негативные тест-кейсы**		
Значение	AAA	123_zzzzzzzzzzzzzzz	AA	Пустая строка	1234_zzzzzzzzzzzzzzz
Пояснение	Строка минимально допустимой длины	Строка максимально допустимой длины	Строка недопустимой длины по нижней границе	Строка недопустимой длины, учтена для надёжности	Строка недопустимой длины по верхней границе

Осталось решить вопрос с недопустимыми символами. К сожалению, так же наглядно, как с длиной, здесь не получится. Даже если подойти строго научно, т. е. выбрать кодировку и по её кодовой таблице определить диапазоны кодов символов (на рисунке 7.c приведён пример такого разделения для ASCII-таблицы), у нас нет никакой гарантии, что символы с кодами из каждого диапазона трактуются единообразно.

Это ярчайший пример случая, в котором тестирование по методу белого ящика сильно облегчило бы нам жизнь. Если бы мы видели, как в коде приложения реализована проверка на допустимые и недопустимые символы, мы могли бы подобрать очень показательные значения входных данных.

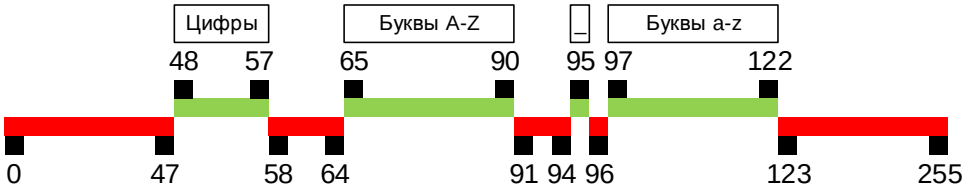


Рисунок 7.c – Неудачный способ поиска классов эквивалентности для наборов допустимых и недопустимых символов (коды символов приведены по ASCII-таблице)

Раз оказалось, что по кодам символов подбирать классы эквивалентности в нашем случае нерационально, посмотрим на ситуацию по-другому (что намного проще). Поделим символы на недопустимые и допустимые, а последние, в свою очередь, – на группы (рисунок 7.d).

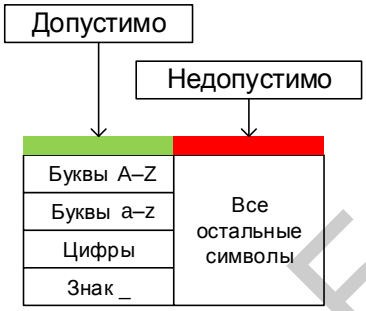


Рисунок 7.d – Классы эквивалентных допустимых и недопустимых символов

Интересующие нас комбинации допустимых символов (с представителями всех групп) мы уже учли при проверке реакции приложения на имена пользователя допустимых и недопустимых длин, потому остаётся учесть только вариант с допустимой длиной строки, но недопустимыми символами (которые можно выбирать случайным образом из соответствующего набора). В таблицу 7.a добавим одну колонку и получим таблицу 7.b.

Таблица 7.b – Значения всех входных данных для тест-кейсов

Категории	**Позитивные тест-кейсы**		**Негативные тест-кейсы**			
Значение	AAA	123_zzzzzzzzzzzzzz	AA	Пустая строка	1234_zzzzzzzzzzzzzz	#$%
Пояснение	Строка минимально допустимой длины	Строка максимально допустимой длины	Строка недопустимой длины по нижней границе	Строка недопустимой длины, учтена для надёжности	Строка недопустимой длины по верхней границе	Строка допустимой длины, недопустимые символы

Конечно, в случае критически важных приложений (например, системы управления ядерным реактором) мы бы проверили с помощью средств автоматизации реакцию приложения на каждый недопустимый символ. Но предположив, что перед нами некое тривиальное приложение, мы можем считать, что одной проверки на недопустимые символы будет достаточно.

Теперь мы возвращаемся к «Конвертеру файлов» (см. подраздел 2.7 «Пример анализа и тестирования требований») и ищем ответ на вопрос о том, не упустили ли мы какие-то важные проверки в подразделе 7.1 «Позитивные и негативные тест-кейсы».

Начнём с того, что выделим группы свойств SOURCE_DIR, от которых зависит работа приложения (такие группы называются «измерениями»):

- Существование каталога (изначальное и во время работы приложения).
- Длина имени.
- Наборы символов в имени.
- Комбинации символов в имени.
- Расположение каталога (локальный или сетевой).
- Права доступа к каталогу (изначальные и во время работы приложения).
- Зарезервированные имена.
- Поведение, зависящее от операционной системы.
- Поведение, зависящее от работы сети.

Задание 7.b: Какие ещё группы свойств вы бы добавили в этот список и какие бы выделили подгруппы у уже имеющихся в списке свойств?

Очевидно, что отмеченные группы свойств оказывают взаимное влияние. Графически его можно отобразить в виде концепт-карты[333] (рисунок 7.e).

Чтобы иметь возможность применить стандартную технику классов эквивалентности и граничных условий, нам нужно по рисунку 7.e дойти от центрального элемента («SOURCE_DIR») до любого конечного, однозначно относящегося к позитивному или негативному тестированию.

Один из таких путей на рисунке 7.e отмечен кружками. Словесно его можно выразить так: SOURCE_DIR → Windows → Локальный каталог → Имя → Свободное → Длина → В кодировке UTF16 → Допустимая или недопустимая.

Максимальная длина пути для Windows в общем случае равна 256 символам: [диск][:\][путь][null] = 1 + 2 + 256 + 1 = 260. Минимальная длина равна 1 символу (точка обозначает «текущий каталог»). Казалось бы, всё очевидно и может быть представлено рисунком 7.f.

[333] Concept map. Wikipedia. URL: http://en.wikipedia.org/wiki/Concept_map.

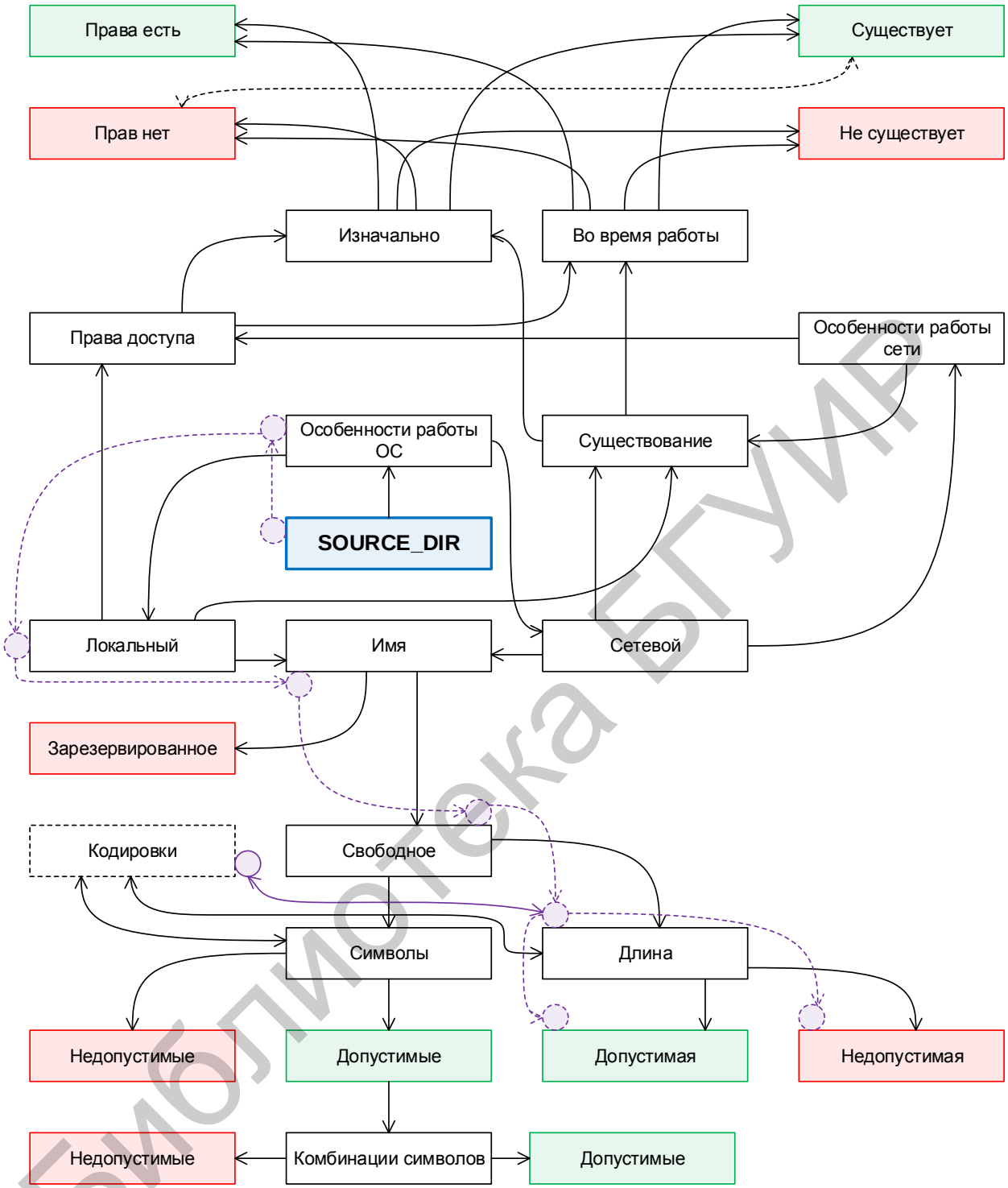


Рисунок 7.e – Концепт-карта взаимовлияния групп свойств каталога

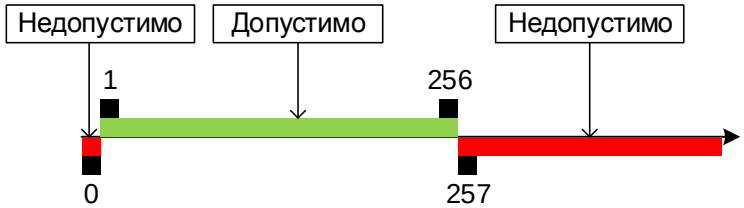


Рисунок 7.f – Классы эквивалентности и граничные условия для длины пути

Но если почитать внимательно спецификацию[334], выясняется, что «физически» путь может быть длиной до 32’767 символов, а ограничение в 260 символов распространяется лишь на так называемое «полное имя». Потому возможна, например, ситуация, когда в каталог с полным именем длиной 200 символов помещается файл с именем длиной 200 символов, и длина полного имени файла получается равной 400 символам (что очевидно больше 260).

Так мы подошли к ситуации, в которой для проведения тестирования нужно либо знать внутреннюю реализацию поведения приложения, либо вносить правки в требования, вводя искусственные ограничения (например, длина имени SOURCE_DIR не может быть более 100 символов, а длина имени любого файла в SOURCE_DIR не может быть более 160 символов, что в сумме может дать максимальную длину в 260 символов).

Ввод искусственных ограничений – плохая идея, поэтому с точки зрения качества мы вполне вправе считать представленное на рисунке 7.f разбиение корректным, а сбои в работе приложения (если таковые будут), вызванные описанной выше ситуацией вида «200 символов + 200 символов», – дефектом.

Таблица 7.c – Значения всех входных данных для тест-кейсов по проверке выбранного на рисунке 7.e пути

Критерии	**Позитивные тест-кейсы**		**Негативные тест-кейсы**	
Значение	. (точка)	C:*256символов*	Пустая строка	C:*257символов*
Пояснение	Имя с минимально допустимой длиной	Имя с максимально допустимой длиной	Имя с недопустимой длиной, учтено для надёжности	Имя с недопустимой длиной

Итак, с одним путём на рисунке 7.e мы разобрались. Но их там значительно больше, и поэтому в следующем разделе мы рассмотрим, как быть в ситуации, когда приходится учитывать влияние на работу приложения большого количества параметров.

7.3 Доменное тестирование и комбинации параметров

Уточним данное ранее определение (см. подраздел 3.2 «Подробная классификация тестирования»):

Доменное тестирование (domain testing, domain analysis[335]) – техника создания эффективных и результативных тест-кейсов в случае, когда несколько переменных могут или должны быть протестированы одновременно.

В качестве инструментов доменного тестирования активно используются техники определения классов эквивалентности и граничных условий, которые были рассмотрены в соответствующем подразделе 7.2. Поэтому мы сразу перейдём к практическому примеру.

[334] Naming Files, Paths, and Namespaces, MSDN. URL: https://msdn.microsoft.com/en-us/library/aa365247.aspx#maxpath.

[335] **Domain analysis** is a technique that can be used to identify efficient and effective test cases when multiple variables can or should be tested together. Copeland L. A practitioner’s guide to software test design.

На рисунке 7.e кружками отмечен путь, один из вариантов которого мы рассмотрели в предыдущем разделе, но вариантов может быть много:

- Семейство ОС:
  - Windows;
  - Linux.
- Расположение каталога:
  - локальный;
  - сетевой.
- Доступность имени:
  - зарезервированное;
  - свободное.
- Длина:
  - допустимая;
  - недопустимая.

Чтобы не усложнять пример, остановимся на этом наборе. Графически комбинации вариантов можно представить в виде иерархии (рисунок 7.g). Исключив совсем нетипичную для нашего приложения экзотику (всё же мы не разрабатываем сетевую утилиту), вычеркнем из списка случаи зарезервированных сетевых имён (отмечены на рисунке 7.g крестиком).

Легко заметить, что при всей своей наглядности графическое представление не всегда удобно в обработке (к тому же мы пока ограничились только общими идеями, не отметив конкретные классы эквивалентности и интересующие нас значения граничных условий).

Альтернативным подходом является представление комбинаций в виде таблицы, которое можно получать последовательно за несколько шагов.

Сначала учтём комбинации значений первых двух параметров – семейства ОС и расположения каталога. Получается таблица 7.d.

Таблица 7.d – Комбинации значений первых двух параметров

Параметры	**Windows**	**Linux**
Локальный путь	+	+
Сетевой путь	+	+

На пересечении строк и столбцов можно отмечать необходимость выполнения проверки (в нашем случае таковая есть, поэтому там стоит «+») или её отсутствие, приоритет проверки, отдельные значения параметров, ссылки и т. д.

Добавим третий параметр (признак зарезервированного имени) и получим таблицу 7.e.

Таблица 7.e – Комбинации значений трёх параметров

Параметры		**Windows**	**Linux**
Зарезервированное имя	**Локальный путь**	+	+
	Сетевой путь	–	–
Свободное имя	**Локальный путь**	+	+
	Сетевой путь	+	+

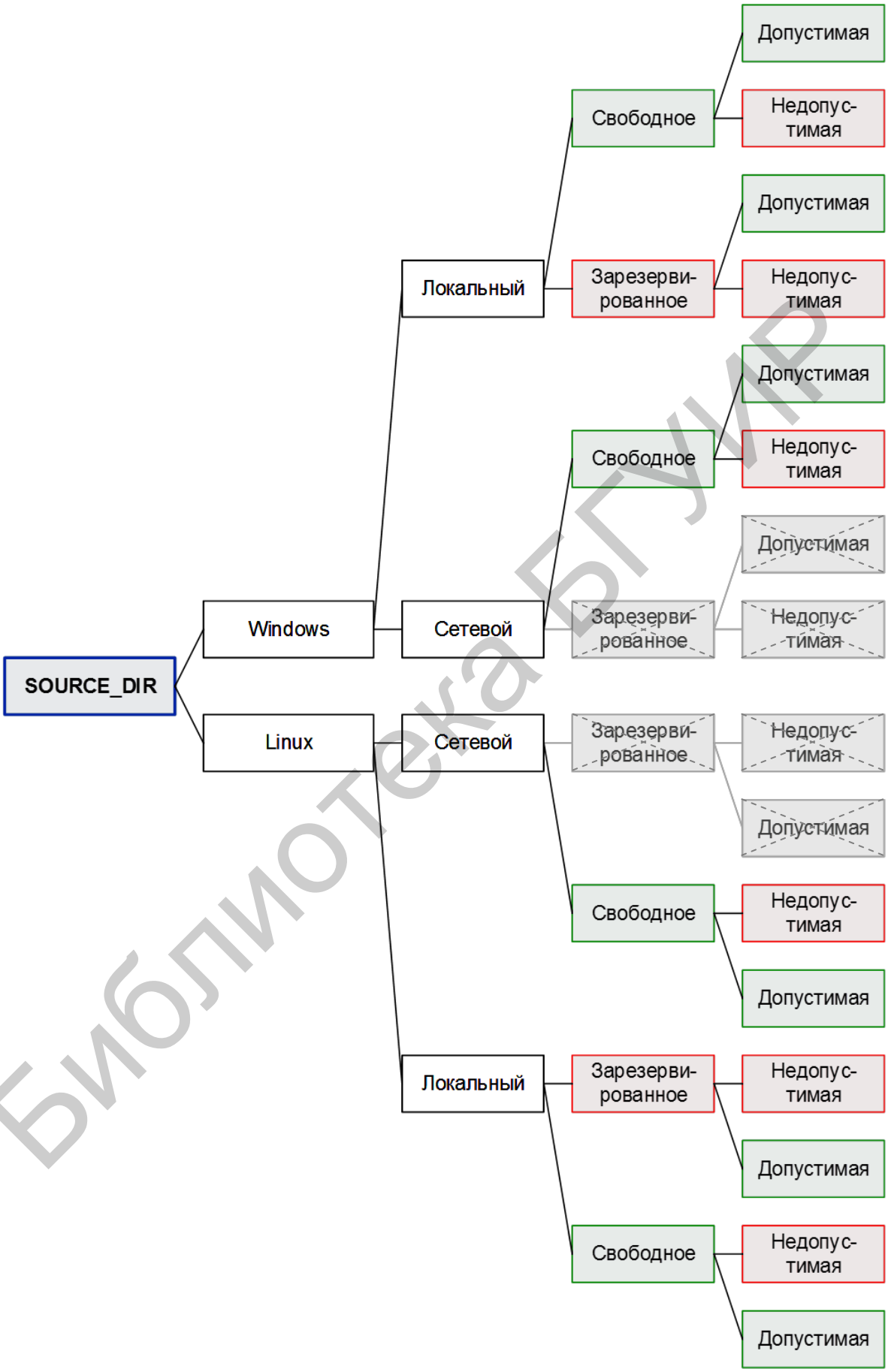


Рисунок 7.g – Графическое представление комбинаций параметров

Добавим четвёртый параметр (признак допустимости длины) и получим таблицу 7.f. Чтобы таблица равномерно увеличивалась по высоте и ширине, удобно добавлять каждый последующий параметр попеременно – то как столбец, то как строку. Третий параметр мы добавили как столбец, четвёртый добавим как строку.

Таблица 7.f – Комбинации значений четырёх параметров

Параметры		Недопустимая длина		Допустимая длина	
		Windows	Linux	Windows	Linux
Зарезервированное имя	Локальный путь	–	–	+	+
	Сетевой путь	–	–	–	–
Свободное имя	Локальный путь	+	+	+	+
	Сетевой путь	+	+	+	+

Такое представление по сравнению с графическим оказывается более компактным и позволяет очень легко увидеть комбинации значений параметров, которые необходимо подвергнуть тестированию. Вместо знаков «+» в ячейки можно поставить ссылки на другие таблицы (хотя иногда все данные совмещают в одной таблице), в которых будут представлены классы эквивалентности и граничные условия для каждого выбранного случая.

Как несложно догадаться, при наличии большого количества параметров, каждый из которых может принимать много значений, таблица наподобие 7.f будет состоять из сотен строк и столбцов. Даже её построение займёт много времени, а выполнение всех соответствующих проверок и вовсе может оказаться невозможным в силу нехватки времени. В следующем разделе мы рассмотрим ещё одну технику тестирования, призванную решить проблему чрезмерно большого количества комбинаций.

7.4 Попарное тестирование и поиск комбинаций

Уточним данное ранее определение (см. подраздел 3.2 «Подробная классификация тестирования»):

Попарное тестирование (pairwise testing[336]) – техника тестирования, в которой вместо проверки всех возможных комбинаций значений всех параметров проверяются только комбинации значений каждой пары параметров.

Выбрать и проверить пары значений – звучит вроде бы просто. Но как выбирать такие пары? Существует несколько тесно взаимосвязанных математических методов создания комбинаций всех пар:

- на основе ортогональных массивов[337, 341];
- на основе латинских квадратов[338];

[336] The answer is not to attempt to test all the combinations for all the values for all the variables but to test **all pairs** of variables. Copeland L. A practitioner's guide to software test design.
[337] Bolton M. Pairwise Testing. URL: http://www.developsense.com/pairwiseTesting.html.

- IPO (in parameter order) метод[339];
- на основе генетических алгоритмов[340];
- на основе рекурсивных алгоритмов[341].

Глубоко в основе этих методов лежит серьёзная математическая теория[341]. В упрощённом виде на примерах суть и преимущества этого подхода показаны в книге Ли Коупленда[342] и статье Майкла Болтона[337], а справедливая критика – в статье Джеймса Баха[343].

Итак, суть проблемы: если мы попытаемся проверить все сочетания всех значений всех параметров для более-менее сложного случая тестирования, мы получим количество тест-кейсов, превышающее все разумные пределы.

Если представить изображённую на рисунке 7.e схему в виде набора параметров и количества их значений, то получается ситуация, представленная таблицей 7.g. Минимальное количество значений получено по принципу «расположение: локально или в сети», «существование: да или нет», «семейство ОС: Windows или Linux» и т. д. Вероятное количество значений оценено исходя из необходимости учитывать несколько классов эквивалентности. Количество значений с учётом полного перебора получено исходя из технических спецификаций операционных систем, файловых систем и т. д. Значение нижней строки получено перемножением значений в соответствующей колонке.

Таблица 7.g – Список параметров, влияющих на работу приложения

Параметр	**Минимальное количество значений**	**Вероятное количество значений**	**Количество значений с учётом полного перебора**
Расположение	2	25	32
Существование	2	2	2
Наличие прав доступа	2	3	155
Семейство ОС	2	4	28
Зарезервированное или свободное имя	2	7	23
Кодировки	2	3	16
Длина	2	4	4096
Комбинации символов	2	4	82
ИТОГО тест-кейсов	**256**	**201’600**	**34’331’384’872’960**

Конечно, мы не будем перебирать все возможные значения (для того нам и нужны классы эквивалентности), но даже 256 тест-кейсов для проверки всего лишь

[338] An Improved Test Generation Algorithm for Pair-Wise Testing. Maity S. [et al.]. URL: http://www.iiserpune.ac.in/~soumen/115-FA-2003.pdf.

[339] Tai K., Lei Y. A Test Generation Strategy for Pairwise Testing. URL: http://www.cs.umd.edu/class/spring2003/cmsc838p/VandV/pairwise.pdf.

[340] Evolutionary Algorithm for Prioritized Pairwise Test Data Generation. Ferrer J. [et al.]. URL: http://neo.lcc.uma.es/staff/javi/files/gecco12.pdf.

[341] Sherwood G. On the Construction of Orthogonal Arrays and Covering Arrays Using Permutation Groups. URL: http://testcover.com/pub/background/cover.htm.

[342] Copeland L. A Practitioner's Guide to Software Test Design.

[343] Bach J. Pairwise Testing: A Best Practice That Isn’t. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3811&rep=rep1&type=pdf.

одного параметра командной строки – это много. И куда вероятнее, что придётся выполнять около 200 000 тест-кейсов. Если делать это вручную и выполнять по одному тесту в пять секунд круглосуточно, понадобится около 11 суток.

Но мы можем применить технику попарного тестирования для генерации оптимального набора тест-кейсов, учитывающего сочетание пар каждого значения каждого параметра. Опишем сами значения. Обратите внимание, что уже на этой стадии мы провели оптимизацию, собрав в один набор информацию о расположении, длине, значении, комбинации символов и признаке зарезервированного имени. Это сделано потому, что сочетания вида «длина 0, зарезервированное имя com1» не имеют смысла. Также мы усилили часть проверок, добавив русскоязычные названия каталогов.

Таблица 7.h – Список параметров и их значений

Параметр	**Значения**
Расположение / длина / значение / комбинация символов / зарезервированное или свободное	1. X:\ 2. X:\dir 3. “X:\пробелы и русский” 4. .\dir 5. ..\dir 6. \\host\dir 7. [256 символов только для Windows] + Пункты 2-6 с “\” в конце пути. 8. / 9. /dir 10. “/пробелы и русский” 11. host:/dir 12. smb://host/dir 13. ./dir 14. ../dir 15. [4096 символов только для Linux] + Пункты 9-14 с “/” в конце пути. 16. Недопустимое имя. 17. [0 символов] 18. [4097 символов только для Linux] 19. [257 символов только для Windows] 20. " 21. // 22. \\ 23. .. 24. com1-com9 25. lpt1-lpt9 26. con 27. nul 28. prn
Существование	1. Да 2. Нет
Наличие прав доступа	1. К каталогу и его содержимому 2. Только к каталогу 3. Ни к каталогу, ни к его содержимому

Продолжение таблицы 7.h

Параметр	Значения
Семейство ОС	1. Windows 32 bit 2. Windows 64 bit 3. Linux 32 bit 4. Linux 64 bit
Кодировки	1. UTF8 2. UTF16 3. OEM

Количество потенциальных тест-кейсов уменьшилось до 2736 (38*2*3*4*3), что уже много меньше 200 тысяч, но всё равно нерационально.

Теперь воспользуемся любым из представленных в огромном количестве инструментальных средств[344] (например, PICT) и сгенерируем набор комбинаций на основе попарного сочетания всех значений всех параметров. Пример первых десяти строк результата представлен в таблице 7.i. Всего получилось 152 комбинации, т. е. в 1326 раз меньше (201’600 / 152) исходной оценки или в 18 раз меньше (2736 / 152) оптимизированного варианта.

Таблица 7.i – Наборы значений, полученные методом попарных комбинаций

Расположение / длина / значение / комбинация символов / зарезервированное или свободное	**Существование**	**Наличие прав доступа**	**Семейство ОС**	**Кодировки**
1 X:\	Да	К каталогу и его содержимому	Windows 64 bit	UTF8
2 smb://host/dir/	Нет	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF16
3 /	Нет	Только к каталогу	Windows 32 bit	OEM
4 [0 символов]	Да	Только к каталогу	Linux 32 bit	UTF8
5 smb://host/dir	Нет	К каталогу и его содержимому	Linux 32 bit	UTF16
6 ../dir	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	OEM
7 [257 символов только для Windows]	Да	Только к каталогу	Windows 64 bit	OEM
8 [4096 символов только для Linux]	Нет	Ни к каталогу, ни к его содержимому	Windows 32 bit	UTF8
9 [256 символов только для Windows]	Нет	Ни к каталогу, ни к его содержимому	Linux 32 bit	OEM
10 /dir/	Да	Только к каталогу	Windows 32 bit	UTF16

Если исследовать набор полученных комбинаций, можно исключить из них те, которые не имеют смысла (например, существование каталога с именем нулевой длины или проверку под Windows характерных только для Linux случаев – см. строки 4 и 8). Завершив такую операцию, мы получаем 124 комбинации. По соображениям экономии места эта таблица не будет приведена, но в приложении Б «Пример данных для попарного тестирования» представлен конечный итог оптимизации (из таб-

[344] Tools A. Pairwise Testing. URL: http://www.pairwise.org/tools.asp.

лицы убраны ещё некоторые комбинации, например, проверка под Linux имён, являющихся зарезервированными для Windows). Получилось 85 тест-кейсов, что даже немного меньше минимальной оценки в 256 тест-кейсов, и при этом мы учли куда больше опасных для приложения сочетаний значений параметров.

Задание 7.c: В представленной в приложении Б «Пример данных для попарного тестирования» колонке «Наличие прав доступа» иногда отсутствуют значения. Как вы думаете, почему? Также в этой таблице всё ещё есть «лишние» тесты, выполнение которых не имеет смысла или представляет собой крайне маловероятный вариант стечения событий. Найдите их.

Итак, на протяжении последних четырёх подразделов мы рассмотрели несколько техник тестирования, позволяющих определить наборы данных и идей для написания эффективных тест-кейсов. Следующий подраздел будет посвящен ситуации, когда времени на столь вдумчивое тестирование нет.

7.5 Исследовательское тестирование

Исследовательское и свободное направления тестирования уже были упомянуты ранее на уровне определения. Для начала ещё раз подчеркнём, что это разные виды тестирования, пусть в каждом из них степень формализации процесса значительно меньше, чем в тестировании на основе тест-кейсов. Сейчас мы будем рассматривать применение именно исследовательского тестирования.

Сэм Канер определяет[345] исследовательское тестирование как стиль, основанный на свободе и ответственности тестировщика в непрерывной оптимизации своей работы за счёт выполняемых параллельно на протяжении всего проекта и взаимодополняющих изучения, планирования, выполнения проверок и оценки их результатов. Если сказать короче, исследовательское тестирование – это одновременное изучение, планирование и тестирование.

Кроме очевидной проблемы с тестированием на основе тест-кейсов, состоящей в высоких затратах времени, существует ещё одна – существующие техники оптимизации направлены на то, чтобы максимально исследовать приложение во всех учтённых ситуациях, которые мы можем контролировать – но невозможно учесть и проконтролировать всё. Эта идея визуально представлена на рисунке 7.h.

Исследовательское же тестирование часто позволяет обнаружить дефекты, вызванные этими неучтёнными факторами. К тому же оно хорошо проявляет себя в следующих ситуациях:

- отсутствие или низкое качество необходимой документации;
- необходимость быстрой оценки качества при нехватке времени;
- подозрение на неэффективность имеющихся тест-кейсов;
- необходимость проверить компоненты, разработанные «третьими сторонами»;
- верификация устранения дефекта (для проверки, что он не проявляется при незначительном отступлении от шагов воспроизведения).

В своей работе Сэм Канер подробно показывает способы проведения исследовательского тестирования с использованием базовых методов, моделей, примеров, частичных изменений сценариев, вмешательства в работу приложения, проверки обработки ошибок, командного тестирования, сравнения продукта с требованиями, дополнительного исследования проблемных областей и т. д.

[345] Kaner C. A Tutorial in Exploratory Testing. URL: http://www.kaner.com/pdfs/QAIExploring.pdf.

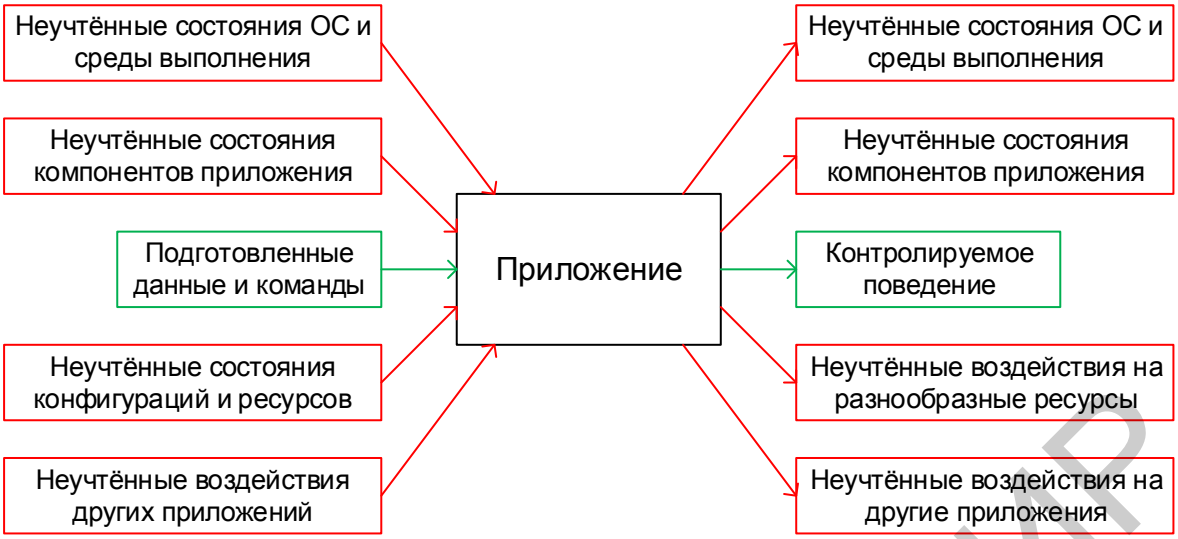


Рисунок 7.h – Факторы, которые могут быть пропущены тестированием на основе тест-кейсов

Вернёмся к нашему «Конвертеру файлов» (см. подраздел 2.7 «Пример анализа и тестирования требований»). Представим следующую ситуацию: разработчики очень уж быстро выпустили первый билд, тест-кейсов (и всех тех наработок, что были рассмотрены ранее в этой книге) у нас пока нет, а проверить билд нужно. Допустим, в уведомлении о выходе билда сказано: «Реализованы и готовы к тестированию требования: СХ-1, СХ-2, СХ-3, ПТ-1.1, ПТ-1.2, ПТ-2.1, ПТ-3.1, ПТ-3.2, БП-1.1, БП-1.2, ДС-1.1, ДС-2.1, ДС-2.2, ДС-2.3, ДС-2.4, ДС-3.1, ДС-3.2 (текст сообщений приведён к информативному виду), ДС-4.1, ДС-4.2, ДС-4.3».

Ранее мы отметили, что исследовательское тестирование – это тесно взаимосвязанные изучение, планирование и тестирование. Применим эту идею.

Изучение

Представим полученную от разработчиков информацию в виде таблицы 7.j и проанализируем соответствующие требования, чтобы понять, что нам нужно будет сделать.

Таблица 7.j – Подготовка к исследовательскому тестированию

Требование	**Что и как будем делать**
СХ-1	Не требует отдельной проверки, т. к. вся работа с приложением будет выполняться в консоли
СХ-2	Не требует отдельной проверки, видно по коду
СХ-3	Провести тестирование под Windows и Linux
ПТ-1.1	Стандартная проверка реакции консольного приложения на различные варианты указания параметров. Учесть, что обязательными являются первые два параметра из трёх (третий принимает значение по умолчанию, если не задан). См. «Идеи», пункт 1
ДС-2.1	
ДС-2.2	
ДС-2.3	
ДС-2.4	
ПТ-1.2	См. «Идеи», пункт 2
ПТ-2.1	Не требует отдельной проверки, покрывается другими тестами

Продолжение таблицы 7.j

Требование	Что и как будем делать
ПТ-3.1	На текущий момент можно только проверить факт ведения лога и формат записей, т. к. основная функциональность ещё не реализована. См. «Идеи», пункт 4
ПТ-3.2	
ДС-4.1	
ДС-4.2	
ДС-4.3	
БП-1.1	См. «Идеи», пункт 3
БП-1.2	
ДС-1.1	Тестирование проводить на PHP 5.5
ДС-3.1	Проверять выводимые сообщения в процессе выполнения пунктов 1–2 (см. «Идеи»)
ДС-3.2	

Планирование

Частично планированием можно считать колонку «Что и как будем делать» таблицы 7.j, но для большей ясности представим эту информацию в виде обобщённого списка, который для простоты назовём «идеи» (да, это – вполне классический чек-лист).

Идеи:

1 Проверить сообщения в ситуациях запуска:
   a) без параметров;
   b) с верно указанными одним, двумя, тремя параметрами;
   c) с неверно указанными первым, вторым, третьим, одним, двумя, тремя параметрами.
2 Остановить приложение по Ctrl+C.
3 Проверить сообщения в ситуациях запуска:
   a) каталог-приёмник и каталог-источник в разных ветках ФС;
   b) каталог-приёмник внутри каталога-источника;
   c) каталог-приёмник, совпадающий с каталогом-источником.
4 Проверить содержимое лога.
5 Посмотреть в код классов, отвечающих за анализ параметров командной строки и ведение лога.

Задание 7.d: Сравните представленный набор идей с ранее рассмотренными подходами (см. подразделы 4.7 «Логика создания эффективных проверок», 7.1 «Позитивные и негативные тест-кейсы», 7.2. «Классы эквивалентности и граничные условия», 7.3 «Доменное тестирование и комбинации параметров», 7.4 «Попарное тестирование и поиск комбинаций»). Какой вариант вам кажется более простым в разработке, а какой в выполнении и почему?

Итак, список идей есть. Фактически, это почти готовый сценарий, если пункт 2 (про остановку приложения) повторять в конце проверок из пунктов 1 и 3.

Тестирование

Можно приступать к тестированию, но стоит отметить, что для его проведения нужно привлекать специалиста, имеющего богатый опыт работы с консольными приложениями, иначе тестирование будет проведено крайне формально и окажется неэффективным.

Что делать с обнаруженными дефектами? Сначала следует фиксировать их в таком же формате, т. е. как список идей: переключение между прохождением

некоторого сценария и написанием отчёта о дефекте сильно отвлекает. Если вы опасаетесь что-то забыть, включите запись происходящего на экране (отличный трюк – записывать весь экран так, чтобы были видны часы, а в списках идей отмечать время, когда вы обнаружили дефект, чтобы потом в записи его было проще найти).

Список «идей дефектов» можно для удобства оформлять в виде таблицы (таблица 7.k).

Таблица 7.k – Список «идей дефектов»

Что делали	Что получили	Что ожидали / Что не так
а) Во всех случаях сообщения приложения вполне корректны с точки зрения происходящего и информативны, но противоречат требованиям (обсудить с заказчиком изменения в требованиях). б) Лог ведётся, формат даты-времени верный, но нужно уточнить, что в требованиях имеется в виду под «имя_операции параметры_операции результат_операции», т. к. для разных операций единый формат не очень удобен – нужно ли приводить всё к одному формату или нет?		
1. php converter.php	Error: Too few command line parameters. USAGE: php converter.php SOURCE_DIR DESTINATION_DIR [LOG_FILE_NAME] Please note that DESTINATION_DIR may NOT be inside SOURCE_DIR	Сообщение совершенно не соответствует требованиям
2. php converter.php zzz:/ c:/	Error: SOURCE_DIR name [zzz:] is not a valid directory	Странно, что от «zzz:/» осталось только «zzz:»
3. php converter.php "c:/non/existing/directory/" c:/	Error: SOURCE_DIR name [c:\non\existing\directory] is not a valid directory	Слеши заменены на бэк-слеши, конечный бэк-слеш удалён: так и надо? Посмотреть в коде, пока не ясно, дефект это или так и задумано
4. php converter.php c:/ d:/	2015.06.12 13:37:56 Started with parameters: SOURCE_DIR=[C:\], DESTINATION_DIR=[D:\], LOG_FILE_NAME=[.\conver = ter.log]	Буквы дисков приведены к верхнему регистру, слеши заменены на бэк-слеши. Почему имя лог-файла относительное?
5. php converter.php c:/ c:/	Error: DESTINATION_DIR [C:\] and SOURCE_DIR [C:\] **mat** NOT be the same dir	Опечатка в сообщении. Явно должно быть must или may
6. php converter.php "c:/каталог с русским именем/" c:/	Error: SOURCE_DIR name [c:\ърЄрыюу ё Ёєёёъшь шьхэхь] is not a valid directory	Дефект: проблема с кодировками

Продолжение таблицы 7.k

Что делали	Что получили	Что ожидали / Что не так
7. php converter.php / c:/Windows/Temp	Error: SOURCE_DIR name [] is not a valid directory	Проверить под Linux: маловероятно, конечно, что кто-то прямо в «/» будет что-то рабочее хранить, но имя «/» урезано до пустой строки, что допустимо для Windows, но не для Linux
8. Примечание: «e:» -- DVD-привод. php converter.php c:/ e:/	file_put_contents(e:f41c7142310c5910e2cfb57993b4d004620aa3b8): failed to open stream: Permission denied in \classes\CLPAnalyser.class.php at line 70 Error: DESTINATION_DIR [e] is not writeable	Дефект: сообщение от PHP не перехвачено
9. php converter.php /var/www /var/www/1	Error: SOURCE_DIR name [var/www] is not a valid directory	Дефект: в Linux обрезается начальный «/» в имени каталога, т. е. можно смело считать, что под Linux приложение неработоспособно (можно задавать только относительные пути, начинающиеся с «.» или «..»)

Выводы по итогам тестирования (которое заняло около получаса):

- Нужно подробно обсудить с заказчиком форматы и содержание сообщений об использовании приложения и об ошибках, а также формат записей лог-файла. Разработчики предложили идеи, выглядящие более адекватно, чем изначально описано в требованиях, но всё равно нужно согласование.
- Под Windows серьёзных дефектов не обнаружено, приложение вполне работоспособно.
- Под Linux есть критическая проблема с исчезновением «/» в начале пути, что не позволяет указывать абсолютные пути к каталогам.
- Если обобщить вышенаписанное, то можно констатировать, что дымовой тест успешно пройден под Windows и провален под Linux.

Цикл «изучение, планирование, тестирование» можно многократно повторить, дополняя и рассматривая список обнаруженных проблем (таблица 7.k) как новую информацию для изучения, ведь каждая такая проблема даёт пищу для размышлений и придумывания дополнительных тест-кейсов.

Задание 7.e: Опишите дефекты, представленные в таблице 7.k в виде полноценных отчётов о дефектах.

В данном разделе в таблице 7.k некоторые пункты представляют собой очевидные дефекты. Но что их вызывает? Почему они возникают, как могут проявляться и на что влиять? Как описать их максимально подробно и правильно в отчётах о дефектах? Ответам на эти вопросы – поиску и исследованию причин возникновения дефектов – посвящен следующий подраздел.

7.6 Поиск причин возникновения дефектов

Ранее мы отмечали, что используем слово «дефект» для обозначения проблемы потому, что описание конечного симптома несёт мало пользы, а выяснение исходной причины может быть достаточно сложным. И всё же наибольший эффект приносит как раз определение и устранение первопричины, что позволяет снизить риск появления новых дефектов, обусловленных той же самой (ненайденной и неустранённой) недоработкой.

!!! **Анализ первопричин** (root cause analysis[346]) – процесс исследования и классификации первопричин возникновения событий, негативно влияющих на безопасность, здоровье, окружающую среду, качество, надёжность и производственный процесс.

Как видно из определения, анализ первопричин не ограничивается разработкой программ, но нас он будет интересовать всё же в ИТ-контексте. Часто ситуация, в которой тестировщик пишет отчёт о дефекте, может быть отражена рисунком 7.i.

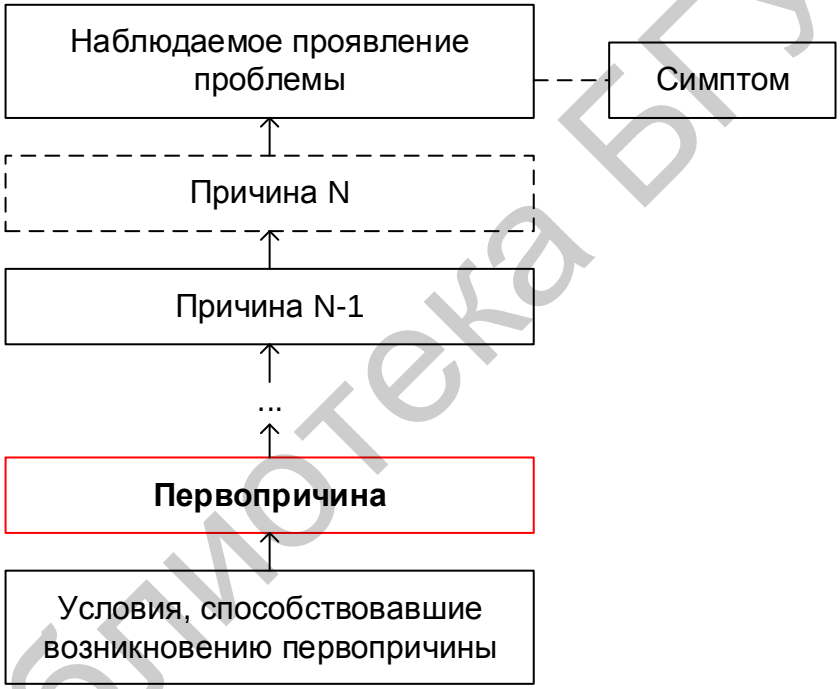


Рисунок 7.i – Проявление и причины дефекта

В самом худшем случае проблема вообще будет пропущена (не замечена), и отчёт о дефекте не будет написан. Чуть лучше выглядит ситуация, когда отчёт описывает исключительно внешние проявления проблемы. Приемлемым может считаться описание лежащих на поверхности причин. Но в идеале нужно стремиться добраться до двух самых нижних уровней – первопричины и условий, способствовавших её возникновению (хоть последнее часто и лежит в области управления проектом, а не тестирования как такового).

[346] **Root cause analysis** (RCA) is a process designed for use in investigating and categorizing the root causes of events with safety, health, environmental, quality, reliability and production impacts. Rooney J., Vanden H. L. Root Cause Analysis for Beginners. URL: https://www.env.nm.gov/aqb/Proposed_Regs/Part_7_Excess_Emissions/NMED_Exhibit_18Root_Cause_Analysis_for_Beginners.pdf.

Коротко вся эта идея выражается тремя простыми вопросами. Нам нужно понять:

- **Что** произошло?
- **Почему** это произошло (найти первопричину)?
- Как **снизить вероятность повторения** такой ситуации?

Сразу же рассмотрим практический пример. В таблице 7.k в строке под номером 9 есть упоминание крайне опасного поведения приложения под Linux – из путей, переданных приложению из командной строки, удаляется начальный символ «/», что для Linux приводит к некорректности любого абсолютного пути.

Пройдём по цепочке, представленной рисунком 7.i, и отразим этот путь в таблице 7.l:

Таблица 7.l – Пример поиска первопричины

Уровень анализа	**Наблюдаемая ситуация**	**Рассуждения и выводы**
Наблюдаемое проявление проблемы	Тестировщик выполнил команду «php converter.php **/var/www** /var/www/1» и получил такой ответ приложения: «Error: SOURCE_DIR name [**var/www**] is not a valid directory.» в ситуации, когда указанный каталог существует и доступен для чтения	Сразу же бросается в глаза, что в сообщении об ошибке имя каталога отличается от заданного – отсутствует начальный «/». Несколько контрольных проверок подтверждают догадку – во всех параметрах командной строки начальный «/» удаляется из полного пути
Причина N	Факт: во всех параметрах командной строки начальный «/» удаляется из полного пути. Проверка с относительными путями («php converter.php . .») и проверка под Windows («php converter.php c:\ d:\») показывает, что в таких ситуациях приложение работает	Дело явно в обработке введённых имён: в некоторых случаях имя обрабатывается корректно, в некоторых – нет. Гипотеза: убираются начальные и концевые «/» (может быть, ещё и «\»)
Причина N1	Проверки «php converter.php \\\\\c:\\\\\ \\\\\d:\\\\\» и «php converter.php /////c:///// /////d://///» показывают, что приложение под Windows запускается, корректно распознав правильные пути: «Started with parameters: SOURCE_DIR=[C:\], DESTINATION_DIR=[D:\]»	Гипотеза подтвердилась: из имён каталогов приложение убирает все «/» и «\», в любом количестве присутствующие в начале или конце имени

Продолжение таблицы 7.I

Уровень анализа	Наблюдаемая ситуация	Рассуждения и выводы

Причина N2	Гипотеза: где-то в коде есть первичный фильтр полученных значений путей, который обрабатывает их до начала проверки каталога на существование. Этот фильтр работает некорректно. Откроем код класса, отвечающего за анализ параметров командной строки. Очень быстро мы обнаруживаем метод, который виновен в происходящем:
	
	`private function getCanonicalName($name)`
	`{`
	`  $name = str_replace('\\', '/', $name);`
	`  $arr = explode('/', $name);`
	`  $name = trim(implode(DIRECTORY_SEPARATOR, $arr), DIRECTORY_SEPARATOR);`
	`  return $name;`
	`}`
		Мы нашли конкретное место в коде приложения, которое является первопричиной обнаруженного дефекта. Информацию об имени файла, номере строки и выдержку самого кода с пояснениями, что в нём неверно, можно приложить в комментарии к отчёту о дефекте. Теперь программисту намного проще устранить проблему

На этом этапе очень часто начинающие тестировщики описывают дефект как «неверно распознаётся имя каталога», «приложение не обнаруживает доступные каталоги» и другими подобными словами. Это плохо как минимум по двум причинам: а) описание дефекта некорректно; б) программисту придётся самому проводить всё исследование.

В принципе, на этой стадии уже можно писать отчёт о дефекте с кратким описанием в стиле «Удаление краевых «/» и «\» из параметров запуска повреждает абсолютные пути в Linux ФС». Но что нам мешает пойти ещё дальше?

Задание 7.f: Представьте, что программист исправил проблему сменой удаления краевых «/» и «\» на концевые (т. е. теперь они удаляются только в конце имени, но не в начале). Хорошее ли это решение?

Обобщённый алгоритм поиска первопричин можно сформулировать следующим образом (рисунок 7.j):

- Определить проявление проблемы:
  - «Что именно происходит?»
  - «Почему это плохо?»
- Собрать необходимую информацию:
  - «Происходит ли то же самое в других ситуациях?»
  - «Всегда ли оно происходит одинаковым образом?»
  - «От чего зависит возникновение или исчезновение проблемы?»
- Выдвинуть гипотезу о причине проблемы:
  - «Что может являться причиной?»
  - «Какие действия или условия могут приводить к проявлению проблемы?»
  - «Какие другие проблемы могут быть причинами данной проблемы?»

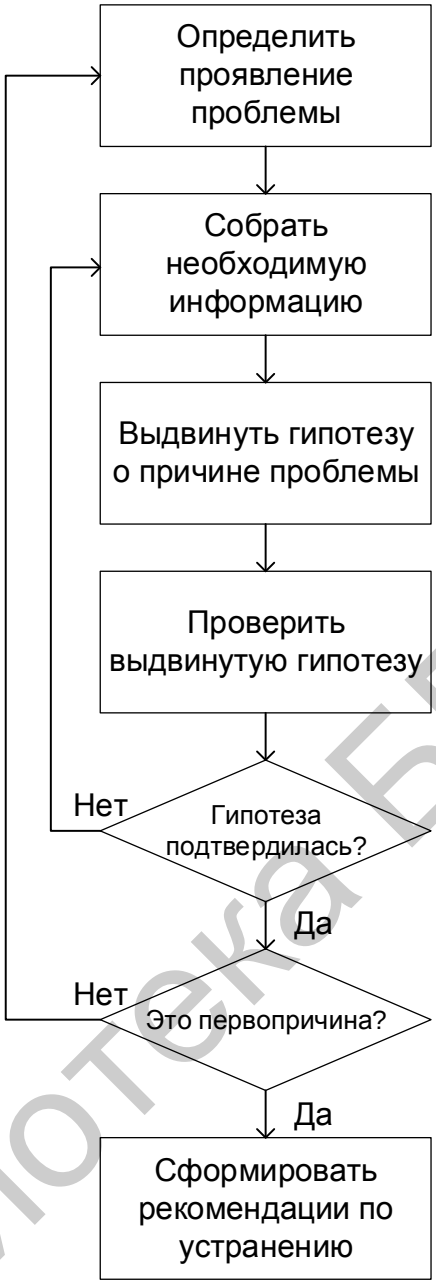


Рисунок 7.j – Алгоритм поиска первопричины дефекта

- Проверить гипотезу:
  - провести дополнительное исследование;
  - если гипотеза не подтвердилась, проработать другие.
- Убедиться, что обнаружена именно первопричина, а не очередная причина в длинной цепи событий:
  - если обнаружена первопричина, то сформировать рекомендации по её устранению;
  - если обнаружена промежуточная причина, повторить алгоритм для неё.

....	Здесь мы рассмотрели очень узкое применение поиска первопричин. Но представленный алгоритм универсален: он работает и в разных предметных областях, и в управлении проектами, и в работе программистов (как часть процесса отладки).

На этом мы завершаем основную часть данного учебного пособия, посвящённую «тестированию в принципе». Далее будет рассмотрена автоматизация тестирования как совокупность техник, повышающих эффективность работы тестировщика по многим показателям.

7.7 Контрольные вопросы и задания

- В чем заключается алгоритм продумывания идей тест-кейсов на основе позитивных и негативных проверок?
- Что такое класс эквивалентности?
- Что такое граничное условие?
- Опишите назначение и сущность техники классов эквивалентности.
- Опишите назначение и сущность техники анализа граничных условий.
- Что такое доменное тестирование?
- Как следует комбинировать параметры при доменном тестировании?
- Что такое попарное тестирование?
- Как следует комбинировать параметры при попарном тестировании?
- Опишите стратегию исследовательского тестирования.
- Опишите стратегию поиска причин возникновения дефекта.
- Что вызывает дефект?
- Почему возникают дефекты?
- Как могут проявляться дефекты и на что влиять?
- Как описать дефекты максимально подробно и правильно в отчётах о дефектах?

8 АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

8.1 Выгоды и риски автоматизации

В разделе, посвящённом подробной классификации тестирования, мы кратко рассматривали, что собой представляет автоматизированное тестирование – это набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования. В таблице 3.b был приведён краткий список преимуществ и недостатков автоматизации, который сейчас мы рассмотрим подробно.

- Скорость выполнения тест-кейсов может в разы и на порядки превосходить возможности человека. Если представить, что человеку придётся вручную сверять несколько файлов размером в несколько десятков мегабайт каждый, оценка времени ручного выполнения баснословная: месяцы или даже годы. При этом 36 проверок, реализуемых в рамках дымового тестирования командными скриптами, выполняются менее чем за пять секунд и требуют от тестировщика только одного действия – запустить скрипт.
- Отсутствует влияние человеческого фактора в процессе выполнения тест-кейсов (усталости, невнимательности и т. д.) Продолжим пример из предыдущего пункта: какова вероятность, что человек ошибётся, сравнивая (посимвольно!) даже два обычных текста размером в сто страниц каждый? А если таких текстов десять или двадцать? И проверки нужно повторять раз за разом? Можно смело утверждать, что человек однозначно ошибётся. Автоматика не ошибётся.
- Средства автоматизации способны выполнить тест-кейсы, в принципе непосильные для человека в силу своей сложности, скорости или иных факторов. И снова наш пример со сравнением больших текстов является актуальным: мы не можем позволить себе потратить годы, раз за разом выполняя крайне сложную рутинную операцию, в которой мы к тому же будем гарантированно допускать ошибки. Другим хорошим примером непосильных для человека тест-кейсов является исследование производительности, в рамках которого необходимо с высокой скоростью выполнять определённые действия, а также фиксировать значения широкого набора параметров. Сможет ли человек, например, сто раз в секунду измерять и записывать объём оперативной памяти, занимаемой приложением? Нет. Автоматика сможет.
- Средства автоматизации способны собирать, сохранять, анализировать, агрегировать и представлять в удобной для восприятия человеком форме колоссальные объёмы данных. В нашем примере с дымовым тестированием «Конвертера файлов» объём данных, полученный в результате тестирования, невелик – его вполне можно обработать вручную. Но если обратиться к реальным проектным ситуациям, журналы работы систем автоматизированного тестирования могут занимать десятки гигабайт по каждой итерации. Логично, что человек не в состоянии вручную проанализировать такие объёмы данных, но правильно настроенная среда автоматизации сделает это сама, предоставив на выход аккуратные отчёты на две-три страницы, удобные графики и таблицы, а также возможность погружаться в детали, переходя от агрегированных данных к подробностям, если в этом возникнет необходимость.
- Средства автоматизации способны выполнять низкоуровневые действия с приложением, операционной системой, каналами передачи данных и т. д. В одном из предыдущих пунктов мы упоминали такую задачу, как «сто раз в секунду измерить и записать объём оперативной памяти, занимаемой приложе-

нием». Подобная задача сбора информации об используемых приложением ресурсах является классическим примером. Однако средства автоматизации могут не только собирать подобную информацию, но и воздействовать на среду исполнения приложения или само приложение, эмулируя типичные события (например, нехватку оперативной памяти или процессорного времени) и фиксируя реакцию приложения. Даже если у тестировщика будет достаточно квалификации, чтобы самостоятельно выполнить подобные операции, ему всё равно понадобится то или иное инструментальное средство – так почему не решить эту задачу сразу на уровне автоматизации тестирования?

Итак, с использованием автоматизации мы получаем возможность увеличить тестовое покрытие за счёт:

- выполнения тест-кейсов, о которых раньше не стоило и думать;
- многократного повторения тест-кейсов с разными входными данными;
- высвобождения времени на создание новых тест-кейсов.

Но всё ли так хорошо с автоматизацией? Увы, нет. Очень наглядно одну из серьёзных проблем можно представить рисунком 8.а.

В первую очередь следует осознать, что автоматизация не происходит сама по себе, не существует волшебной кнопки, нажатием которой решаются все проблемы. Более того, с автоматизацией тестирования связана серия серьёзных недостатков и рисков:

- Необходимость наличия высококвалифицированного персонала в силу того факта, что автоматизация – это «проект внутри проекта» (со своими требованиями, планами, кодом и т. д.). Даже если забыть на мгновение про «проект внутри проекта», техническая квалификация сотрудников, занимающихся автоматизацией, как правило, должна быть ощутимо выше, чем у их коллег, занимающихся ручным тестированием.

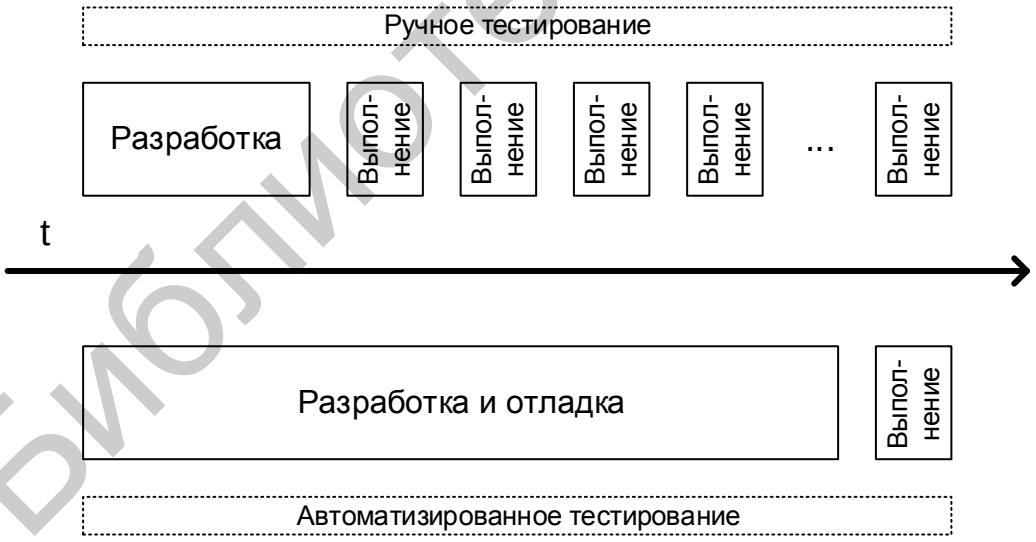


Рисунок 8.а – Соотношение времени разработки и выполнения тест-кейсов в ручном и автоматизированном тестировании

- Разработка и сопровождение как самих автоматизированных тест-кейсов, так и всей необходимой инфраструктуры занимает очень много времени. Ситуация усугубляется тем, что в некоторых случаях (при серьёзных изменениях в проекте или в случае ошибок в стратегии) всю соответствующую работу приходится выполнять заново с нуля: в случае ощутимого изменения требований,

смены технологического домена, переработки интерфейсов (как пользовательских, так и программных) многие тест-кейсы становятся безнадёжно устаревшими и требуют создания заново.

- Автоматизация требует более тщательного планирования и управления рисками, т. к. в противном случае проекту может быть нанесён серьёзный ущерб (см. предыдущий пункт про переделку всех наработок с нуля).
- Коммерческие средства автоматизации стоят ощутимо дорого, а имеющиеся бесплатные аналоги не всегда позволяют эффективно решать поставленные задачи. И здесь мы снова вынуждены вернуться к вопросу ошибок в планировании: если изначально набор технологий и средств автоматизации был выбран неверно, придётся не только переделывать всю работу, но и покупать новые средства автоматизации.
- Средств автоматизации крайне много, что усложняет проблему выбора того или иного средства, затрудняет планирование и определение стратегии тестирования, может повлечь за собой дополнительные временные и финансовые затраты, а также необходимость обучения персонала или найма соответствующих специалистов.

Итак, автоматизация тестирования требует ощутимых инвестиций и сильно повышает проектные риски, а потому существуют специальные подходы[347, 348, 349, 350] по оценке применимости и эффективности автоматизированного тестирования. Если выразить всю их суть очень кратко, то в первую очередь следует учесть:

- Затраты времени на ручное выполнение тест-кейсов и на выполнение этих же тест-кейсов, но уже автоматизированных. Чем ощутимее разница, тем более выгодной представляется автоматизация.
- Количество повторений выполнения одних и тех же тест-кейсов. Чем оно больше, тем больше времени мы сможем сэкономить за счёт автоматизации.
- Затраты времени на отладку, обновление и поддержку автоматизированных тест-кейсов. Этот параметр сложнее всего оценить, и именно он представляет наибольшую угрозу успеху автоматизации, потому здесь для проведения оценки следует привлекать наиболее опытных специалистов.
- Наличие в команде соответствующих специалистов и их рабочую загрузку. Автоматизацией занимаются самые квалифицированные сотрудники, которые в это время не могут решать иные задачи.

В качестве небольшого примера беглой оценки эффективности автоматизации можно привести следующую формулу[351]:

$$A_{effect} = \frac{N \cdot T_{manual}^{overall}}{N \cdot T_{automated}^{run\ and\ analyse} + T_{automated}^{development\ and\ support}},$$

где A_{effect} – коэффициент выгоды от использования автоматизации;

[347] Garrett T. Implementing Automated Software Testing – Continuously Track Progress and Adjust Accordingly. URL: http://www.methodsandtools.com/archive/archive.php?id=94.
[348] The Return of Investment (ROI) of Test Automation. Münch S. [et al.]. URL: https://www.ispe.org/pe-ja/roi-of-test-automation.pdf.
[349] Kelly M. The ROI of Test Automation. URL: http://www.sqetraining.com/sites/default/files/articles/XDD8502filelistfilename1_0.pdf.
[350] Hoffman D. Cost Benefits Analysis of Test Automation. URL: http://www.softwarequality methods.com/papers/star99%20model%20paper.pdf.
[351] Zhyrytskyy V. Introduction to automation.

N – планируемое количество билдов приложения;

$T_{manual}^{overall}$ – расчётное время разработки, выполнения и анализа результатов ручного тестирования;

$T_{automated}^{run\ and\ analyse}$ – расчётное время выполнения и анализа результатов автоматизированного тестирования;

$T_{automated}^{development\ and\ support}$ – расчётное время разработки и сопровождения автоматизированного тестирования.

Чтобы нагляднее представить, как эта формула может помочь, изобразим график коэффициента выгоды автоматизации в зависимости от количества билдов (рисунок 8.b). Допустим, что в некотором проекте значения параметров таковы:

$T_{manual}^{overall}$ = 30 ч на каждый билд;

$T_{automated}^{run\ and\ analyse}$ = 5 ч на каждый билд;

$T_{automated}^{development\ and\ support}$ = 300 ч на весь проект.

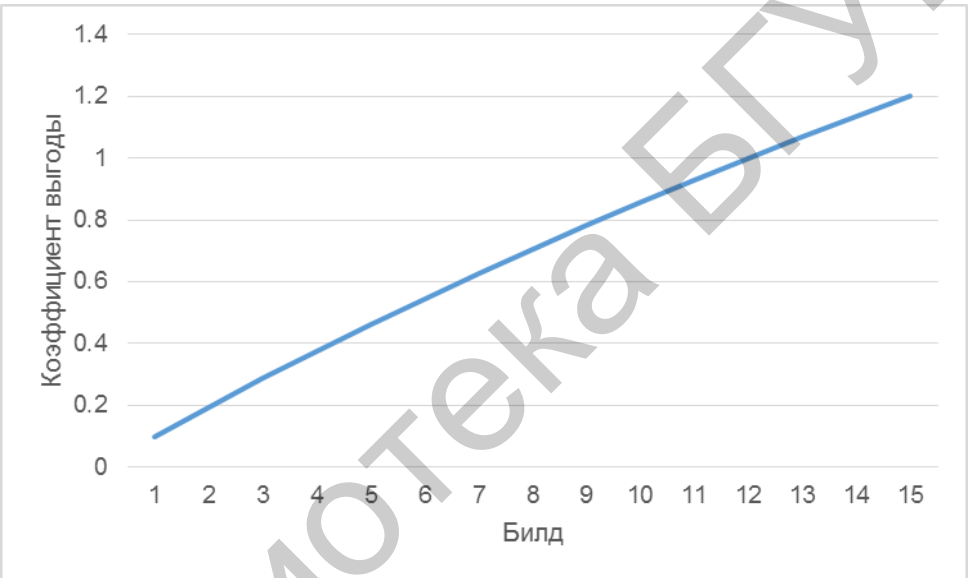


Рисунок 8.b – Коэффициент выгоды автоматизации в зависимости от количества билдов

Как видно на рисунке 8.b, лишь к 12-му билду автоматизация окупит вложения и с 13-го билда начнёт приносить пользу. И тем не менее существуют области, в которых автоматизация даёт ощутимый эффект почти сразу. Их рассмотрению посвящен следующий подраздел.

Для того чтобы выяснить области применения автоматизации, сначала мы ещё раз посмотрим на список задач, решить которые помогает автоматизация:

- Выполнение тест-кейсов, непосильных человеку.
- Решение рутинных задач.
- Ускорение выполнения тестирования.
- Высвобождение человеческих ресурсов для интеллектуальной работы.
- Увеличение тестового покрытия.
- Улучшение кода за счёт увеличения тестового покрытия и применения специальных техник автоматизации.

Эти задачи чаще всего встречаются и проще всего решаются в следующих случаях (таблица 8.a).

Таблица 8.а – Случаи наибольшей применимости автоматизации

Случай/задача	Какую проблему решает автоматизация
Регрессионное тестирование	Необходимость выполнять вручную тесты, количество которых неуклонно растёт с каждым билдом, но вся суть которых сводится к проверке того факта, что ранее работавшая функциональность продолжает работать корректно
Инсталляционное тестирование и настройка тестового окружения	Множество часто повторяющихся рутинных операций по проверке работы инсталлятора, размещения файлов в файловой системе, содержимого конфигурационных файлов, реестра и т. д. Подготовка приложения в заданной среде и с заданными настройками для проведения основного тестирования
Конфигурационное тестирование и тестирование совместимости	Выполнение одних и тех же тест-кейсов на большом множестве входных данных, под разными платформами и в разных условиях. Классический пример: есть файл настроек, в нём сто параметров, каждый может принимать сто значений: существует 100^{100} вариантов конфигурационного файла – все их нужно проверить
Использование комбинаторных техник тестирования (в т. ч. доменного тестирования)	Генерация комбинаций значений и многократное выполнение тест-кейсов с использованием этих сгенерированных комбинаций в качестве входных данных
Модульное тестирование	Проверка корректности работы атомарных участков кода и элементарных взаимодействий таких участков кода – практически невыполнимая для человека задача при условии, что нужно выполнить тысячи таких проверок и нигде не ошибиться
Интеграционное тестирование	Глубокая проверка взаимодействия компонентов в ситуации, когда человеку почти нечего наблюдать, т. к. все представляющие интерес и подвергаемые тестированию процессы проходят на уровнях более глубоких, чем пользовательский интерфейс
Тестирование безопасности	Необходимость проверки прав доступа, паролей по умолчанию, открытых портов, уязвимостей текущих версий ПО и т. д., т. е. быстрое выполнение очень большого количества проверок, в процессе которого нельзя что-то пропустить, забыть или «не так понять»
Тестирование производительности	Создание нагрузки с интенсивностью и точностью, недоступной человеку. Сбор с высокой скоростью большого набора параметров работы приложения. Анализ большого объёма данных из журналов работы системы автоматизации
Дымовой тест для крупных систем	Выполнение при получении каждого билда большого количества достаточно простых для автоматизации тест-кейсов

Продолжение таблицы 8.a

Случай/задача	Какую проблему решает автоматизация
Приложения (или их части) без графического интерфейса	Проверка консольных приложений на больших наборах значений параметров командной строки (и их комбинаций). Проверка приложений и их компонентов, вообще непредназначенных для взаимодействия с человеком (веб-сервисы, серверы, библиотеки и т. д.)
Длительные, рутинные, утомительные для человека и/или требующие повышенного внимания операции	Проверки, требующие сравнения больших объёмов данных, высокой точности вычислений, обработки большого количества размещённых по всему дереву каталогов файлов, ощутимо большого времени выполнения и т. д. Особенно, когда такие проверки повторяются очень часто
Проверка «внутренней функциональности» веб-приложений (ссылок, доступности страниц и т. д.)	Автоматизация предельно рутинных действий (например, проверить все 30’000+ ссылки на предмет того, что все они ведут на реально существующие страницы). Автоматизация здесь упрощается в силу стандартности задачи – существует много готовых решений
Стандартная, однотипная для многих проектов функциональность	Даже высокая сложность при первичной автоматизации в таком случае окупится за счёт простоты многократного использования полученных решений в разных проектах
«Технические задачи»	Проверки корректности протоколирования, работы с базами данных, корректности поиска, файловых операций, корректности форматов и содержимого генерируемых документов и т. д.

С другой стороны, существуют случаи, в которых автоматизация скорее всего приведёт только к ухудшению ситуации. Вкратце – это все те области, где требуется человеческое мышление, а также некоторый перечень технологических областей. Чуть более подробно список выглядит так (таблица 8.b):

Таблица 8.b – Случаи наименьшей применимости автоматизации

Случай/задача	В чём проблема автоматизации
Планирование	Компьютер пока не научился думать
Разработка тест-кейсов	
Написание отчётов о дефектах	
Анализ результатов тестирования и отчётность	
Функциональность, которую нужно (достаточно) проверить всего несколько раз	Затраты на автоматизацию не окупятся
Тест-кейсы, которые нужно выполнить всего несколько раз (если человек может их выполнить)	
Низкий уровень абстракции в имеющихся инструментах автоматизации	Придётся писать очень много кода, что не только сложно и долго, но и приводит к появлению множества ошибок в самих тест-кейсах

Продолжение таблицы 8.b

Случай/задача	В чём проблема автоматизации
Слабые возможности средства автоматизации по протоколированию процесса тестирования и сбору технических данных о приложении и окружении	Есть риск получить данные в виде «что-то где-то сломалось», что не помогает в диагностике проблемы
Низкая стабильность требований	Придётся очень многое переделывать, что в случае автоматизации обходится дороже, чем в случае ручного тестирования
Сложные комбинации большого количества технологий	Высокая сложность автоматизации, низкая надёжность тест-кейсов, высокая сложность оценки трудозатрат и прогнозирования рисков
Проблемы с планированием и ручным тестированием	Автоматизация хаоса приводит к появлению автоматизированного хаоса, но при этом ещё и требует трудозатрат. Сначала стоит решить имеющиеся проблемы, а потом включаться в автоматизацию
Нехватка времени и угроза срыва сроков	Автоматизация не приносит мгновенных результатов. Поначалу она лишь потребляет ресурсы команды (в том числе время). Также есть универсальный афоризм: «Лучше руками протестировать хоть что-то, чем автоматизированно протестировать ничего»
Области тестирования, требующие оценки ситуации человеком (тестирование удобства использования, тестирование доступности и т. д.)	В принципе, можно разработать некие алгоритмы, оценивающие ситуацию так, как её мог бы оценить человек. Но на практике живой человек может сделать это быстрее, проще, надёжнее и дешевле

Вывод: стоит помнить, что эффект от автоматизации наступает не сразу и не всегда. Как и любой дорогостоящий инструмент, автоматизация при верном применении может дать ощутимую выгоду, но при неверном принесёт лишь весьма ощутимые затраты.

8.2 Особенности автоматизированного тестирования

Во множестве источников, посвящённых основам автоматизации тестирования, можно встретить схемы наподобие представленной на рисунке 8.c, т. е. автоматизация тестирования представляет собой сочетание программирования и тестирования в разных масштабах (в зависимости от проекта и конкретных задач).

Отсюда следует простой вывод, что специалист по автоматизации тестирования должен сочетать в себе навыки и умения как программиста, так и тестировщика. Но этим перечень не заканчивается: умение администрировать операционные системы, сети, различные серверы, умение работать с базами данных, понимание мобильных платформ и т. д. – всё это может пригодиться.

Но даже если остановиться только на навыках программирования и тестирования, в автоматизации тоже есть свои особенности – набор технологий. В классическом ручном тестировании развитие происходит постепенно и эволюционно – проходят годы и даже десятилетия между появлением новых подходов, завоёвывающих популярность. В программировании прогресс идёт чуть быстрее, но и там специалистов выручает согласованность и схожесть технологий.

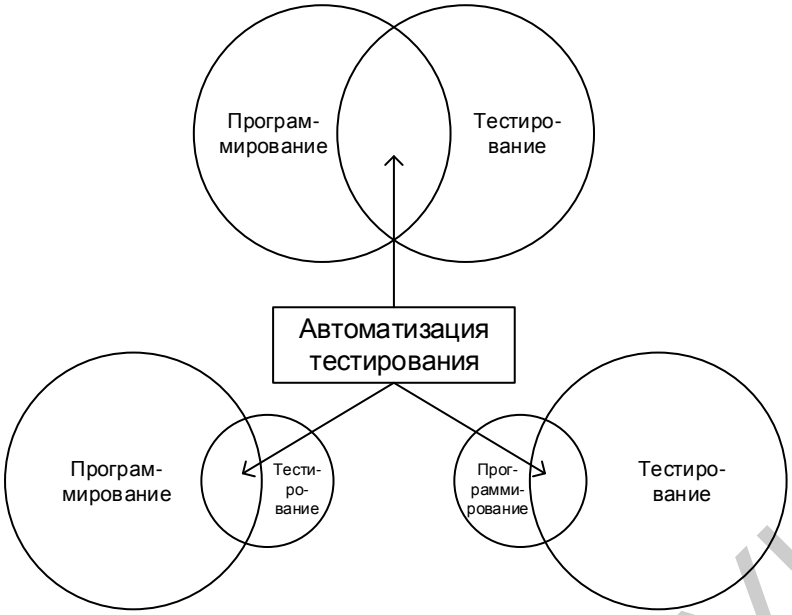


Рисунок 8.с – Сочетание программирования и тестирования в автоматизации тестирования

В автоматизации тестирования ситуация выглядит иначе: десятки и сотни технологий и подходов (как заимствованных из смежных дисциплин, так и уникальных) появляются и исчезают очень стремительно. Количество инструментальных средств автоматизации тестирования уже исчисляется тысячами и продолжает неуклонно расти.

Поэтому для программирования и тестирования крайне важны такие навыки, как высокая обучаемость и способность в предельно сжатые сроки самостоятельно найти, изучить, понять и начать применять на практике совершенно новую информацию из, возможно, ранее абсолютно незнакомой области. Звучит немного пугающе, но одно можно гарантировать: скучно не будет точно.

О нескольких наиболее распространённых технологиях мы поговорим в подразделе 8.2 «Особенности автоматизированного тестирования».

Особенности тест-кейсов в автоматизации

Часто (а в некоторых проектах и «как правило») автоматизации подвергаются тест-кейсы, изначально написанные на естественном языке (и, в принципе, пригодные для выполнения вручную), т. е. обычные классические тест-кейсы, которые мы уже рассматривали подробно в соответствующем разделе.

И всё же есть несколько важных моментов, которые стоит учитывать при разработке (или доработке) тест-кейсов, предназначенных для дальнейшей автоматизации.

Главная проблема состоит в том, что компьютер – это не человек, и соответствующие тест-кейсы не могут оперировать «интуитивно понятными описаниями», а специалисты по автоматизации совершенно справедливо не хотят тратить время на то, чтобы дополнить такие тест-кейсы необходимыми для выполнения автоматизации техническими подробностями, – у них хватает собственных задач.

Отсюда следует список рекомендаций по подготовке тест-кейсов к автоматизации и непосредственно самой автоматизации:

- Ожидаемый результат в автоматизированных тест-кейсах должен быть описан предельно чётко с указанием конкретных признаков его корректности (таблица 8.с). Сравните.

Таблица 8.c – Пример описания ожидаемого результата в тест-кейсах для автоматизации

Плохо	Хорошо
… 7. Загружается стандартная страница поиска	… 7. Загружается страница поиска: title = «Search page», присутствует форма с полями «input type=”text”», «input type=“submit” value=“Go!”», присутствует логотип «logo.jpg» и отсутствуют иные графические элементы («img»)

- Поскольку тест-кейс может быть автоматизирован с использованием различных инструментальных средств, следует описывать его, избегая специфических для того или иного инструментального средства решений (таблица 8.d). Сравните.

Таблица 8.d – Пример описания инструментальных средств в шагах тест-кейса для автоматизации

Плохо	Хорошо
1. Щёлкнуть на ссылке «Search». 2. Использовать clickAndWait для синхронизации тайминга	1. Щёлкнуть на ссылке «Search». 2. Дождаться завершения загрузки страницы

- В продолжение предыдущего пункта: тест-кейс может быть автоматизирован для выполнения под разными аппаратными и программными платформами, поэтому не стоит изначально прописывать в него что-то, характерное лишь для одной платформы (таблица 8.e). Сравните.

Таблица 8.e – Пример описания шагов в тест-кейсах для автоматизации

Плохо	Хорошо
… 8. Отправить приложению сообщение WM_CLICK в любое из видимых окон	… 8. Передать фокус ввода любому из несвёрнутых окон приложения (если таких нет – развернуть любое из окон). 9. Проэмулировать событие «щёлкнуть кнопкой мыши» для активного окна

- Одной из неожиданно проявляющихся проблем до сих пор является синхронизация средства автоматизации и тестируемого приложения по времени: в случаях, когда для человека ситуация является понятной, средство автоматизации тестирования может среагировать неверно, «не дождавшись» определённого состояния тестируемого приложения (таблица 8.f). Это приводит к завершению неудачей тест-кейсов на корректно работающем приложении. Сравните.

Таблица 8.f – Пример синхронизации средства автоматизации и тестируемого приложения по времени в тест-кейсах для автоматизации

Плохо	Хорошо
1. Щёлкнуть на ссылку «Expand data». 2. Выбрать из появившегося списка значение «Unknown»	1. Щёлкнуть на ссылку «Expand data». 2. Дождаться завершения загрузки данных в список «Extended data» (select id=”extended_data”): список перейдёт в состояние enabled. 3. Выбрать в списке «Extended data» значение «Unknown»

- Не стоит побуждать специалиста по автоматизации к вписыванию в код тест-кейса константных значений (так называемый «хардкодинг»). Если вы можете понятно описать словами значение и/или смысл некоторой переменной, то сделайте это (таблица 8.g). Сравните.

Таблица 8.g – Пример использования константных значений в тест-кейсах для автоматизации

Плохо	Хорошо
1. Открыть http://application/	1. Открыть главную страницу приложения

- По возможности следует использовать наиболее универсальные способы взаимодействия с тестируемым приложением. В случае если изменится набор технологий, с помощью которых реализовано приложение, это значительно сократит время поддержки тест-кейсов (таблица 8.h). Сравните.

Таблица 8.h – Пример использования универсальных способов взаимодействия с тестируемым приложением в тест-кейсах для автоматизации

Плохо	Хорошо
… 8. Передать в поле «Search» набор событий WM_KEY_DOWN, {знак}, WM_KEY_UP, в результате чего в поле должен быть введён поисковый запрос	… 8. Проэмулировать ввод значения поля «Search» с клавиатуры (не годится вставка значения из буфера или прямое присваивание значения!)

- Автоматизированные тест-кейсы должны быть независимыми. Из любого правила бывают исключения, но в абсолютном большинстве случаев следует предполагать, что мы не знаем, какие тест-кейсы будут выполнены до и после нашего тест-кейса (таблица 8.i). Сравните.

Таблица 8.i – Пример реализации принципа независимости тест-кейсов для автоматизации

Плохо	Хорошо
1. Из файла, созданного предыдущим тестом…	1. Перевести чек-бокс «Use stream buffer file» в состояние checked. 2. Активировать процесс передачи данных (щёлкнуть на кнопку «Start»). 3. Из файла буферизации прочитать…

- Стоит помнить, что автоматизированный тест-кейс – это программа, и стоит учитывать хорошие практики программирования хотя бы на уровне отсутствия так называемых «магических значений», «хардкодинга» и т. п. (таблица 8.j). Сравните.

Таблица 8.j – Пример тест-кейсов для автоматизации, отражающий различные подходы к написанию кода

Плохо	Хорошо

if ($date_value == '2015.06.18')
{
…
}
«Хардкодинг»
«Магическое значение»
if ($status = 42)
{
…
}
Ошибка в выражении (= вместо ==)
	if ($date_value == date('Y.m.d'))
	{
	…
	}
	Осмысленная константа
	Актуальные данные
	if (POWER_USER == $status)
	{
	…
	}
	Ошибка исправлена, к тому же константа в сравнении находится слева от переменной

- Стоит внимательно изучать документацию по используемому средству автоматизации, чтобы избежать ситуации, когда из-за неверно выбранной команды тест-кейс становится ложно положительным, т. е. успешно проходит в ситуации, когда приложение работает неверно.

Так называемые ложно положительные тест-кейсы – едва ли не самое опасное, что бывает в автоматизации тестирования: они вселяют в проектную команду ложную уверенность в том, что приложение работает корректно, т. е. фактически прячут дефекты, вместо того, чтобы обнаруживать их.

Поскольку для многих начинающих тестировщиков первым учебным средством автоматизации тестирования является Selenium IDE[352], приведём пример с его использованием. Допустим, в некотором шаге тест-кейса нужно было проверить, что чек-бокс с id = cb выбран (checked). По какой-то причине тестировщик выбрал неверную команду, и сейчас на этом шаге проверятся, что чек-бокс позволяет изменять своё состояние (enabled, editable), а не что он выбран (таблица 8.k).

Таблица 8.k – Пример тест-кейсов для автоматизации, отражающий различие между проверкой и действием

Плохо (неверная команда)	Хорошо (верная команда)
… … … verifyEditable id = cb … … … .	… … … verifyChecked id = cb … … …

[352] Selenium IDE. URL: http://docs.seleniumhq.org/projects/ide/.

- И напоследок рассмотрим ошибку, которую по какой-то мистической причине совершает добрая половина начинающих автоматизаторов – это замена проверки действием, и наоборот. Например, вместо проверки значения поля они изменяют значение. Или вместо изменения состояния чек-бокса проверяют его состояние. Здесь не будет примеров на «плохо/хорошо», т. к. хорошего варианта здесь нет – такого просто не должно быть: это грубейшая ошибка.

Кратко подытожив рассмотренное, отметим, что тест-кейс, предназначенный для автоматизации, будет куда более похож на миниатюрное техническое задание по разработке небольшой программы, чем на описание корректного поведения тестируемого приложения, понятное человеку.

И ещё одна особенность автоматизированных тест-кейсов, которая заслуживает отдельного рассмотрения, – это источники данных и способы их генерации. Для выполняемых вручную тест-кейсов эта проблема не столь актуальна, т. к. при выполнении их 3–5–10 раз мы также вручную вполне можем подготовить нужное количество вариантов входных данных. Но если мы планируем выполнить тест-кейс 50–100–500 раз с разными входными данными, вручную столько данных мы не подготовим. Источниками данных в такой ситуации могут стать:

- Случайные величины: случайные числа, случайные символы, случайные элементы из некоторого набора и т. д.
- Генерация (случайных) данных по алгоритму: случайные числа в заданном диапазоне, строки случайной длины из заданного диапазона из случайных символов из определённого набора (например, строка длиной 10–100 символов, состоящая только из букв), файлы с увеличивающимся по некоторому правилу размером (например, 10 Кбайт, 100 Кбайт, 1000 Кбайт и т. д.).
- Получение данных из внешних источников: извлечение данных из базы данных, обращение к некоторому веб-сервису и т. д.
- Собранные рабочие данные, т. е. данные, созданные реальными пользователями в процессе их реальной работы (например, если бы мы захотели разработать собственный текстовый редактор, тысячи имеющихся у нас и наших коллег doc(x)-файлов были бы такими рабочими данными, на которых мы бы проводили тестирование).
- Ручная генерация – да, она актуальная и для автоматизированных тест-кейсов. Например, вручную создать по десять (да даже и по 50–100) корректных и некорректных e-mail-адресов получится куда быстрее, чем писать алгоритм их генерации.

Применение некоторых из этих идей по генерации данных мы рассмотрим подробнее.

Для того чтобы познакомиться с технологиями автоматизации, мы рассмотрим несколько высокоуровневых технологий автоматизации тестирования, каждая из которых, в свою очередь, базируется на своём наборе технических решений (инструментальных средствах, языках программирования, способах взаимодействия с тестируемым приложением и т. д.)

Начнём с краткого обзора эволюции высокоуровневых технологий, при этом следует подчеркнуть, что «старые» решения по-прежнему используются (или как компоненты «новых», или самостоятельно в отдельных случаях) (таблица 8.I).

Таблица 8.I – Эволюция высокоуровневых технологий автоматизации тестирования

Подход	Суть	Преимущества	Недостатки
1. Частные решения	Для решения каждой отдельной задачи пишется отдельная программа	Быстро, просто	Нет системности, много времени уходит на поддержку. Почти невозможно повторное использование
2.Тестирование под управлением данными (DDT)	Из тест-кейса вовне выносятся входные данные и ожидаемые результаты	Один и тот же тест-кейс можно повторять многократно с разными данными	Логика тест-кейса по-прежнему строго определяется внутри, а потому для её изменения тест-кейс надо переписывать
3.Тестирование под управлением ключевыми словами (KDT)	Из тест-кейса вовне выносится описание его поведения	Концентрация на высокоуровневых действиях. Данные и особенности поведения хранятся вовне и могут быть изменены без изменения кода тест-кейса	Сложность выполнения низкоуровневых операций
4.Использование фреймворков	Конструктор, позволяющий использовать остальные подходы	Мощность и гибкость	Относительная сложность (особенно – в создании фреймворка)
5.Запись и воспроизведение (Record & Playback)	Средство автоматизации записывает действия тестировщика и может воспроизвести их, управляя тестируемым приложением	Простота, высокая скорость создания тест-кейсов	Крайне низкое качество, линейность, неподдерживаемость тест-кейсов. Требуется серьёзная доработка полученного кода
6.Тестирование под управлением поведением (BDT)	Развитие идей тестирования под управлением данными и ключевыми словами. Отличие – в концентрации на бизнес-сценариях без выполнения мелких проверок	Высокое удобство проверки высокоуровневых пользовательских сценариев	Такие тест-кейсы пропускают большое количество функциональных и нефункциональных дефектов, а потому должны быть дополнены классическими более низкоуровневыми тест-кейсами

На текущем этапе развития тестирования представленные в таблице 8.l технологии иерархически можно изобразить следующей схемой (рисунок 8.d):

Рисунок 8.d – Иерархия технологий автоматизации тестирования

Сейчас мы рассмотрим эти технологии подробнее и с примерами, но сначала стоит упомянуть один основополагающий подход, который находит применение практически в любой технологии автоматизации, – функциональную декомпозицию.

!!! **Функциональная декомпозиция** (functional decomposition[353]) – процесс определения функции через её разделение на несколько низкоуровневых подфункций.

Функциональная декомпозиция активно используется как в программировании, так и в автоматизации тестирования с целью упрощения решения поставленных задач и получения возможности повторного использования фрагментов кода для решения различных высокоуровневых задач. Рассмотрим пример (рисунок 8.e) (легко заметить, что часть низкоуровневых действий (поиск и заполнение полей, поиск и нажатие кнопок) универсальна и может быть использована при решении других задач, например, при регистрации, оформлении заказа и т. д.).

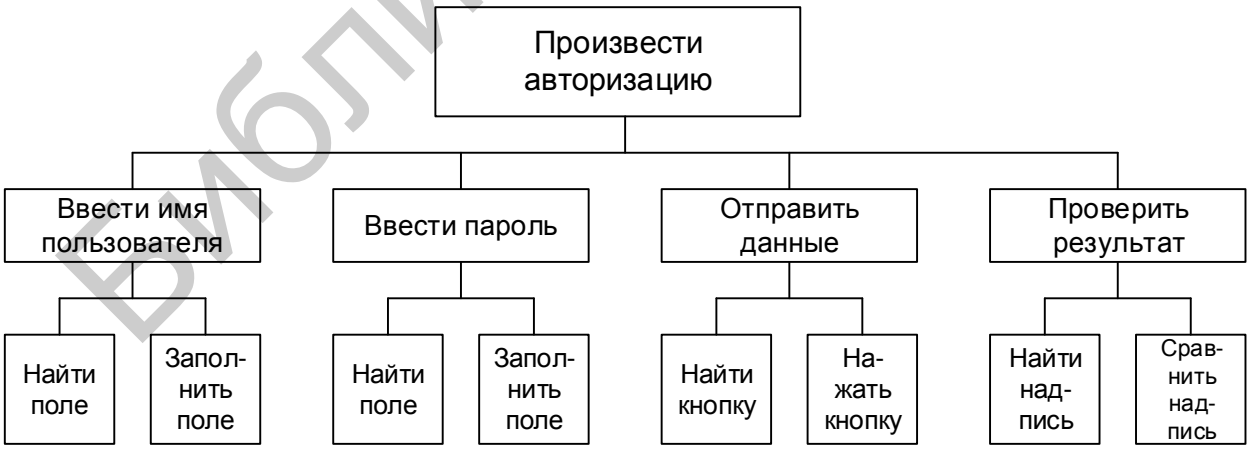


Рисунок 8.e – Пример функциональной декомпозиции в программировании и тестировании

[353] **Functional decomposition.** The process of defining lower-level functions and sequencing relationships. System Engineering Fundamentals, Defense Acquisition University Press.

Применение функциональной декомпозиции позволяет не только упростить процесс решения поставленных задач, но и избавиться от необходимости самостоятельной реализации действий на самом низком уровне, т. к. они, как правило, уже решены авторами соответствующих библиотек или фреймворков.

Возвращаемся к технологиям автоматизации тестирования.

Частные решения

Иногда перед тестировщиком возникает уникальная (в том плане, что такой больше не будет) задача, для решения которой нет необходимости использовать мощные инструментальные средства, а достаточно написать небольшую программу на любом из высокоуровневых языков программирования (Java, C#, PHP и т. д.) или даже воспользоваться возможностями командных файлов операционной системы или подобными тривиальными решениями.

Ярчайшим примером такой задачи и её решения является автоматизация дымового тестирования нашего «Конвертера файлов» (код командных файлов для Windows и Linux приведён в соответствующем приложении). Также сюда можно отнести задачи вида:

- Подготовить базу данных, наполнив её тестовыми данными (например, добавить в систему миллион пользователей со случайными именами).
- Подготовить файловую систему (например, создать структуру каталогов и набор файлов для выполнения тест-кейсов).
- Перезапустить набор серверов и/или привести их в требуемое состояние.

Удобство частных решений состоит в том, что их можно реализовать быстро, просто, «вот прямо сейчас». Но у частных решений есть и огромный недостаток – это «кустарные решения», которыми может воспользоваться всего пара человек. И при появлении новой задачи, даже очень похожей на ранее решённую, скорее всего, придётся всё автоматизировать заново.

Более подробно преимущества и недостатки частных решений в автоматизации тестирования показаны в таблице 8.m.

Таблица 8.m – Преимущества и недостатки частных решений в автоматизации тестирования

Преимущества	Недостатки
• Быстрота и простота реализации. • Возможность использования любых доступных инструментов, которыми тестировщик умеет пользоваться. • Эффект от использования наступает незамедлительно. • Возможность нахождения очень эффективных решений в случае, когда основные инструменты, используемые на проекте для автоматизации тестирования, оказываются малопригодными для выполнения данной отдельной задачи. • Возможность быстрого создания и оценки прототипов перед применением более тяжеловесных решений	• Отсутствие универсальности и, как следствие, невозможность или крайняя сложность повторного использования (адаптации для решения других задач). • Разрозненность и несогласованность решений между собой (разные подходы, технологии, инструменты, принципы решения). • Крайне высокая сложность развития, поддержки и сопровождения таких решений (чаще всего, кроме самого автора никто вообще не понимает, что и зачем было сделано, и как оно работает). • Является признаком «кустарного производства», не приветствуется в промышленной разработке программ

Тестирование под управлением данными (DDT)

Обратите внимание, как много раз в командных файлах (приложение А) для Windows и Linux, автоматизирующих выполнение дымового тестирования, повторяются строки, выполняющие одно и то же действие над набором файлов (и нам ещё повезло, что файлов немного). Ведь куда логичнее было бы каким-то образом подготовить список файлов и просто передать его на обработку. Это и будет тестированием под управлением данными. В качестве примера приведём небольшой скрипт на PHP, который читает CSV-файл с тестовыми данными (именами сравниваемых файлов) и выполняет сравнение файлов:

```
function compare_list_of_files($file_with_csv_data)
{
     $data = file($file_with_csv_data);
     foreach ($data as $line)
     {
          $parsed = str_csv($line);
          if (md5_file($parsed[0]) === md5_file($parsed[1])) {
               file_put_contents('smoke_test.log',
               "OK! File '".$parsed[0]."' was processed correct-
          ly!\n");
          } else {
               file_put_contents('smoke_test.log',
               "ERROR! File '".$parsed[0]."' was NOT
               processed correctly!\n");
          }
     }
}
```

Пример CSV-файла (первые две строки):

```
Test_ETALON/«Мелкий» эталон WIN1251.txt,OUT/«Мелкий» файл в
WIN1251.txt
Test_ETALON/«Средний» эталон CP866.txt,OUT/«Средний» файл
CP866.txt
```

Теперь нам достаточно подготовить CSV-файл с любым количеством имён сравниваемых файлов, а код тест-кейса не увеличится ни на байт.

К другим типичным примерам использования тестирования под управлением данными относятся:

- Проверка авторизации и прав доступа на большом наборе имён пользователей и паролей.
- Многократное заполнение полей форм разными данными и проверка реакции приложения.
- Выполнение тест-кейса на основе данных, полученных с помощью комбинаторных техник (пример таких данных представлен в соответствующем приложении).

Данные для рассматриваемого подхода к организации тест-кейсов могут поступать из файлов, баз данных и иных внешних источников или даже генерироваться в процессе выполнения тест-кейса (см. описание источников данных для автоматизированного тестирования).

Преимущества и недостатки тестирования под управлением данными показаны в таблице 8.n.

Таблица 8.n – Преимущества и недостатки тестирования под управлением данными

Преимущества	Недостатки
• Устранение избыточности кода тест-кейсов. • Удобное хранение и понятный человеку формат данных. • Возможность поручения генерации данных сотруднику, не имеющему навыков программирования. • Возможность использования одного и того же набора данных для выполнения разных тест-кейсов. • Возможность повторного использования набора данных для решения новых задач. • Возможность использования одного и того же набора данных в одном и том же тест-кейсе, но реализованном под разными платформами	• При изменении логики поведения тест-кейса его код всё равно придётся переписывать. • При неудачно выбранном формате представления данных резко снижается их понятность для неподготовленного специалиста. • Необходимость использования технологий генерации данных. • Высокая сложность кода тест-кейса в случае сложных неоднородных данных. • Риск неверной работы тест-кейсов в случае, когда несколько тест-кейсов работают с одним и тем же набором данных, и он был изменён таким образом, на который не были рассчитаны некоторые тест-кейсы.Слабые возможности по сбору данных в случае обнаружения дефектов. • Качество тест-кейса зависит от профессионализма сотрудника, реализующего код тест-кейса

Тестирование под управлением ключевыми словами

Логическим развитием идеи о вынесении вовне тест-кейса данных является вынесение вовне тест-кейса команд (описания действий). Разовьём только что показанный пример, добавив в CSV-файл ключевые слова, являющиеся описанием выполняемой проверки:

- moved – файл отсутствует в каталоге-источнике и присутствует в каталоге-приёмнике;
- intact – файл присутствует в каталоге-источнике и отсутствует в каталоге-приёмнике;
- equals – содержимое файлов идентично.

```
function execute_test($scenario)
{
     $data = file($scenario);
     foreach ($data as $line)
     {
          $parsed = str_csv($line);
          switch ($parsed[0])
          {
               // Проверка перемещения файла
               case 'moved':
                    if
(is_file($parsed[1]))&&(!is_file($parsed[2])) {
                         file_put_contents('smoke_test.log',
                         "OK!   '".$parsed[0]."'   file   was   pro-
cessed!\n");
                    } else {
```

```
                    file_put_contents('smoke_test.log',
                    "ERROR! '".$parsed[0]."' file was
                    NOT processed!\n");
               }
          break;

          // Проверка отсутствия перемещения файла
          case 'intact':
               if
(!is_file($parsed[1]))||(is_file($parsed[2])) {
                    file_put_contents('smoke_test.log',
                    "OK!   '".$parsed[0]."'   file   was   pro-
               cessed!\n");
               } else {
                    file_put_contents('smoke_test.log',
                    "ERROR! '".$parsed[0]."' file was
                    NOT processed!\n");
               }
          break;

          // Сравнение файлов
          case 'equals':
               if          (md5_file($parsed[1])          ===
md5_file($parsed[2])){
                    file_put_contents('smoke_test.log',
                    "OK! File '".$parsed[0]."' Was
                    processed correctly!\n");
               } else {
                    file_put_contents('smoke_test.log',
                    "ERROR! File '".$parsed[0]."' Was
                    NOT processed correctly!\n");
               }
          break;
     }
  }
}
```

Пример CSV-файла (первые пять строк):

```
moved,IN/«Мелкий» эталон WIN1251.txt,OUT/«Мелкий» файл в
WIN1251.txt
moved,IN/«Средний» эталон CP866.txt,OUT/«Средний» файл CP866.txt
intact,IN/Картинка.jpg,OUT/Картинка.jpg
equals,Test_ETALON/«Мелкий» эталон WIN1251.txt,OUT/«Мелкий» файл в
WIN1251.txt
equals,Test_ETALON/«Средний» эталон CP866.txt,OUT/«Средний» файл
CP866.txt
```

Ярчайшим примером инструментального средства автоматизации тестирования, идеально следующего идеологии тестирования под управлением ключевыми словами, является Selenium IDE, в котором каждая операция тест-кейса описывается в следующем виде:

Действие (ключевое слово)	Необязательный параметр 1	Необязательный параметр 2

Тестирование под управлением ключевыми словами стало тем переломным моментом, начиная с которого стало возможным привлечение к автоматизации тестирования нетехнических специалистов. Согласитесь, что нет необходимости в знаниях программирования и тому подобных технологий, чтобы наполнять подобные только что показанному CSV-файлы или (что очень часто практикуется) XLSX-файлы.

Вторым естественным преимуществом тестирования под управлением ключевыми словами (хотя она вполне характерна и для тестирования под управлением данными) стала возможность использования различных инструментов одними и теми же наборами команд и данных. Так, например, ничто не мешает нам взять показанные CSV-файлы и написать новую логику их обработки не на PHP, а на C#, Java, Python или даже с использованием специализированных средств автоматизации тестирования.

Преимущества и недостатки тестирования под управлением ключевыми словами показаны в таблице 8.о.

Таблица 8.о – Преимущества и недостатки тестирования под управлением ключевыми словами

Преимущества	**Недостатки**
• Максимальное устранение избыточности кода тест-кейсов. • Возможность построения мини-фреймворков, решающих широкой спектр задач. • Повышение уровня абстракции тест-кейсов и возможность их адаптации для работы с разными техническими решениями. • Удобное хранение и понятный человеку формат данных и команд тест-кейса. • Возможность поручения описания логики тест-кейса сотруднику, не имеющему навыков программирования. • Возможность повторного использования для решения новых задач. • Расширяемость (возможность добавления нового поведения тест-кейса на основе уже реализованного поведения)	• Высокая сложность (и, возможно, длительность) разработки. • Высокая вероятность наличия ошибок в коде тест-кейса. • Высокая сложность (или невозможность) выполнения низкоуровневых операций, если код тест-кейса не поддерживает соответствующие команды. • Эффект от использования данного подхода наступает далеко не сразу (сначала идёт длительный период разработки и отладки самих тест-кейсов и вспомогательной функциональности). • Для реализации данного подхода требуется наличие высококвалифицированного персонала. • Необходимо обучить персонал языку ключевых слов, используемых в тест-кейсах

Использование фреймворков

Фреймворки автоматизации тестирования представляют собой не что иное, как успешно развившиеся и ставшие популярными решения, объединяющие в себе лучшие стороны тестирования под управлением данными, ключевыми словами и возможности реализации частных решений на высоком уровне абстракции.

Фреймворков автоматизации тестирования очень много, они очень разные, но их объединяет несколько общих черт:

- высокая абстракция кода (нет необходимости описывать каждое элементарное действие) с сохранением возможности выполнения низкоуровневых действий;
- универсальность и переносимость используемых подходов;
- достаточно высокое качество реализации (для популярных фреймворков).

Как правило, каждый фреймворк специализируется на своём виде тестирования, уровне тестирования, наборе технологий. Существуют фреймворки для модульного тестирования (например, семейство xUnit), тестирования веб-приложений (например, семейство Selenium), тестирования мобильных приложений, тестирования производительности (JMeter, HP LoadRunner).

Существуют бесплатные и платные фреймворки, оформленные в виде библиотек на некотором языке программирования или в виде привычных приложений с графическим интерфейсом, узко- и широкоспециализированные и т. д.

К сожалению, подробное описание даже одного фреймворка может по объёму быть сопоставимо со всем текстом данного учебного пособия. Но если вам интересно, начните хотя бы с изучения Selenium WebDriver[354].

Преимущества и недостатки фреймворков автоматизации тестирования показаны в таблице 8.р.

Таблица 8.р – Преимущества и недостатки фреймворков автоматизации тестирования

Преимущества	Недостатки
• Широкое распространение. • Универсальность в рамках своего набора технологий. • Хорошая документация и большое сообщество специалистов, с которыми можно проконсультироваться. • Высокий уровень абстракции. • Наличие большого набора готовых решений и описаний соответствующих лучших практик применения того или иного фреймворка для решения тех или иных задач	• Требуется время на изучение фреймворка. • В случае написания собственного фреймворка фактически получается новый проект по разработке ПО. • Высокая сложность перехода на другой фреймворк. • В случае прекращения поддержки фреймворка тест-кейсы рано или поздно придётся переписывать с использованием нового фреймворка. • Высокий риск выбора неподходящего фреймворка

Запись и воспроизведение (Record & Playback)

Технология записи и воспроизведения (Record & Playback) стала актуальной с появлением достаточно сложных средств автоматизации, обеспечивающих глубокое взаимодействие с тестируемым приложением и операционной системой. Использование этой технологии, как правило, сводится к следующим основным шагам:

1. Тестировщик вручную выполняет тест-кейс, а средство автоматизации записывает все его действия.
2. Результаты записи представляются в виде кода на высокоуровневом языке программирования (в некоторых средствах – специально разработанном).
3. Тестировщик редактирует полученный код.

[354] Selenium WebDriver Documentation. URL: http://docs.seleniumhq.org/docs/03_webdriver.jsp.

4. Готовый код автоматизированного тест-кейса выполняется для проведения тестирования в автоматизированном режиме.

Возможно, вам приходилось записывать макросы в офисных приложениях. Это тоже технология записи и воспроизведения, только применённая для автоматизации решения офисных задач.

Сама технология при достаточно высокой сложности внутренней реализации очень проста в использовании и по самой своей сути, поэтому часто применяется для обучения начинающих специалистов по автоматизации тестирования. Её основные преимущества и недостатки показаны в таблице 8.q.

Таблица 8.q – Преимущества и недостатки технологии записи и воспроизведения

Преимущества	Недостатки
• Предельная простота освоения (достаточно буквально нескольких минут, чтобы начать использовать данную технологию). • Быстрое создание «скелета тест-кейса» за счёт записи ключевых действий с тестируемым приложением. • Автоматический сбор технических данных о тестируемом приложении (идентификаторов и локаторов элементов, надписей, имён и т. д.) • Автоматизация рутинных действий (заполнения полей, нажатий на ссылки, кнопки и т. д.) • В отдельных случаях допускает использование тестировщиками без навыков программирования	• Линейность тест-кейсов: в записи не будет циклов, условий, вызовов функций и прочих характерных для программирования и автоматизации явлений. • Запись лишних действий (как просто ошибочных случайных действий тестировщика с тестируемым приложением, так и (во многих случаях) переключений на другие приложения и работы с ними). • Так называемый «хардкодинг», т. е. запись внутрь кода тест-кейса конкретных значений, что потребует ручной доработки для перевода тест-кейса на технологию тестирования под управлением данными. • Неудобные имена переменных, неудобное оформление кода тест-кейса, отсутствие комментариев и прочие недостатки, усложняющие поддержку и сопровождение тест-кейса в будущем. • Низкая надёжность самих тест-кейсов в силу отсутствия обработки исключений, проверки условий и т. д.

Таким образом, технология записи и воспроизведения не является универсальным средством на все случаи жизни и не может заменить ручную работу по написанию кода автоматизированных тест-кейсов, но в отдельных ситуациях может сильно ускорить решение простых рутинных задач.

Тестирование под управлением поведением

Рассмотренные выше технологии автоматизации максимально сфокусированы на технических аспектах поведения приложения и обладают общим недостатком – с их помощью сложно проверять высокоуровневые пользовательские сценарии (а именно в них и заинтересованы заказчики и конечные пользователи). Этот недостаток призвано исправить тестирование под управлением поведения, в котором акцент делается не на отдельных технических деталях, а на общей работоспособности приложения при решении типичных пользовательских задач.

Такой подход не только упрощает выполнение целого класса проверок, но и облегчает взаимодействие между разработчиками, тестировщиками, бизнес-аналитиками и заказчиком, т. к. в основе подхода лежит очень простая формула «given-when-then»:

- Given («имея, предполагая, при условии») описывает начальную ситуацию, в которой находится пользователь в контексте работы с приложением.
- When («когда») описывает набор действий пользователя в данной ситуации.
- Then («тогда») описывает ожидаемое поведение приложения.

Рассмотрим на примере нашего «Конвертера файлов»:

- **При условии**, что приложение запущено.
- **Когда** я помещаю во входной каталог файл поддерживаемого размера и формата.
- **Тогда** файл перемещается в выходной каталог, а текст внутри файла перекодируется в UTF-8.

Такой принцип описания проверок позволяет даже участникам проекта, не имеющим глубокой технической подготовки, принимать участие в разработке и анализе тест-кейсов, а для специалистов по автоматизации упрощается создание кода автоматизированных тест-кейсов, т.к. такая форма является стандартной, единой и при этом предоставляет достаточно информации для написания высокоуровневых тест-кейсов. Существуют специальные технические решения (например, Behat, JBehave, NBehave, Cucumber), упрощающие реализацию тестирования под управлением поведением.

Преимущества и недостатки тестирования под управлением поведением показаны в таблице 8.r.

Таблица 8.r – Преимущества и недостатки тестирования под управлением поведением

Преимущества	Недостатки
• Фокусировка на потребностях конечных пользователей. • Упрощение сотрудничества между различными специалистами	• Высокоуровневые поведенческие тест-кейсы пропускают много деталей, а потому могут не обнаружить часть проблем в приложении или не предоставить необходимой для понимания обнаруженной проблемы информации
• Простота и скорость создания и анализа тест-кейсов (что, в свою очередь, повышает полезный эффект автоматизации и снижает накладные расходы)	• В некоторых случаях информации, предоставленной в поведенческом тест-кейсе, недостаточно для его непосредственной автоматизации

К классическим технологиям автоматизации тестирования также можно отнести разработку под управлением тестированием (Test-driven Development, TDD) с её принципом «красный, зелёный, улучшенный» (Red-Green-Refactor), разработку под управлением поведением (Behavior-driven Development), модульное тестирование (Unit Testing) и т. д. Но эти технологии уже находятся на границе тестирования и разработки приложений, поэтому выходят за рамки данного учебного пособия.

8.3 Автоматизация вне прямых задач тестирования

На протяжении данного раздела мы рассматривали, как автоматизация может помочь в создании и выполнении тест-кейсов. Но все те же принципы можно перенести и на остальную работу тестировщика, в которой также бывают длительные и утомительные задачи, рутинные задачи или задачи, требующие предельного внимания, но не связанные с интеллектуальной работой. Всё перечисленное также можно автоматизировать.

Да, это требует технических знаний и первоначальных затрат сил и времени на реализацию, но в перспективе такой подход может экономить до нескольких часов в день. К самым типичным решениям из данной области можно отнести:

- Использование командных файлов для выполнения последовательностей операций – от копирования нескольких файлов из разных каталогов до развёртывания тестового окружения. Даже в рамках многократно рассмотренных примеров по тестированию «Конвертера файлов» запуск приложения через командный файл, в котором указаны все необходимые параметры, избавляет от необходимости вводить их каждый раз вручную.
- Генерация и оформление данных с использованием возможностей офисных приложений, баз данных, небольших программ на высокоуровневых языках программирования. В этом случае тестировщику не придется вручную нумеровать три сотни строк в таблице.
- Подготовка и оформление технических разделов для отчётов. Можно тратить часы на скрупулёзное вычитывание журналов работы некоего средства автоматизации, а можно один раз написать скрипт, который будет за мгновение готовить документ с аккуратными таблицами и графиками, и останется только запускать этот скрипт и прикреплять результаты его работы к отчёту.
- Управление своим рабочим местом: создание и проверка резервных копий, установка обновлений, очистка дисков от устаревших данных и т. д. и т. п. Компьютер всё это может (и должен!) делать сам, без участия человека.
- Сортировка и обработка почты. Даже раскладывание входящей корреспонденции по подпапкам гарантированно занимает у вас несколько минут в день. Если предположить, что настройка специальных правил в вашем почтовом клиенте сэкономит вам полчаса в неделю, за год экономия составит примерно сутки.
- Виртуализация как способ избавления от необходимости каждый раз устанавливать и настраивать необходимый набор программ. Если у вас есть несколько заранее подготовленных виртуальных машин, их запуск займёт секунды. А в случае необходимости устранения сбоев разворачивание виртуальной машины из резервной копии заменяет весь процесс установки и настройки с нуля операционной системы и всего необходимого программного обеспечения.

Иными словами, автоматизация объективно облегчает жизнь любого ИТ-специалиста, а также расширяет его кругозор, технические навыки и способствует профессиональному росту.

8.4 Контрольные вопросы и задания

- Какие области тестирования можно автоматизировать?
- Дайте определение автоматизации тестирования.
- Какие тесты являются наилучшими и какие наихудшими кандидатами на автоматизацию?

- Почему в тестировании не всегда следует автоматизировать даже то, что можно автоматизировать?
- Каковы условия успешной автоматизации?
- Каковы преимущества автоматизации?
- Каковы недостатки автоматизации?
- Какие лучшие идеи автоматизации вы запомнили?
- Что такое технология Record&Playback?
- Что такое тестирование Data-Driven и Keyword-Driven?
- Какие наиболее распространённые средства автоматизации вы знаете?
- Сравните соотношение времени разработки и выполнения тест-кейсов в ручном и автоматизированном тестированиях.
- Перечислите необходимые знания и навыки для автоматизации тестирования.
- Какие особенности написания тест-кейсов следует учитывать при автоматизации?
- Какие существуют технологии автоматизации тестирования?
- Дайте характеристику технологии тестирования под управлением данными.
- Дайте характеристику технологии тестирования под управлением ключевыми словами.
- Перечислите преимущества и недостатки фреймворков автоматизации тестирования.
- Дайте характеристику технологии записи и воспроизведения.
- Дайте характеристику технологии тестирования под управлением поведением.
- Приведите формулу оценки эффективности автоматизации.

ПРИЛОЖЕНИЕ А

(справочное)

Командные файлы для Windows и Linux, автоматизирующие выполнение дымового тестирования

CMD-скрипт для Windows

```
rem Переключение кодовой таблицы консоли
rem (чтобы корректно обрабатывались спецсимволы в командах):
chcp 65001

rem Удаление файла журнала от прошлого запуска:
del smoke_test.log /Q

rem Очистка входного каталога приложения:
del IN\*.* /Q

rem Запуск приложения:
start php converter.php IN OUT converter.log

rem Размещение тестовых файлов во входном каталоге приложения:
copy Test_IN\*.* IN > nul

rem Тайм-аут в 10 секунд, чтобы приложение успело обработать файлы:
timeout 10

rem Остановка приложения:
taskkill /IM php.exe

rem ============================================================
rem Проверка появления в выходном каталоге файлов,
rem которые должны быть обработаны,
rem и непоявления файлов, которые не должны быть обработаны:
echo Processing test: >> smoke_test.log

IF EXIST "OUT\«Мелкий» файл в WIN1251.txt" (
 echo OK! '«Мелкий» файл в WIN1251.txt' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Мелкий» файл в WIN1251.txt' file was NOT pro-
cessed! >> smoke_test.log
)

IF EXIST "OUT\«Средний» файл в CP866.txt" (
 echo OK! '«Средний» файл в CP866.txt' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в CP866.txt' file was NOT processed!
>> smoke_test.log
)
```

```
IF EXIST "OUT\«Крупный» файл в KOI8R.txt" (
 echo OK! '«Крупный» файл в KOI8R.txt' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в KOI8R.txt' file was NOT processed!
>> smoke_test.log
)

IF EXIST "OUT\«Крупный» файл в win-1251.html" (
 echo OK! '«Крупный» файл в win-1251.html' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в win-1251.html' file was NOT pro-
cessed! >> smoke_test.log
)

IF EXIST "OUT\«Мелкий» файл в cp-866.html" (
 echo OK! '«Мелкий» файл в cp-866.html' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Мелкий» файл в cp-866.html' file was NOT pro-
cessed! >> smoke_test.log
)

IF EXIST "OUT\«Средний» файл в koi8-r.html" (
 echo OK! '«Средний» файл в koi8-r.html' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в koi8-r.html' file was NOT pro-
cessed! >> smoke_test.log
)

IF EXIST "OUT\«Средний» файл в WIN_1251.md" (
 echo OK! '«Средний» файл в WIN_1251.md' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в WIN_1251.md' file was NOT pro-
cessed! >> smoke_test.log
)

IF EXIST "OUT\«Крупный» файл в CP_866.md" (
 echo OK! '«Крупный» файл в CP_866.md' file was processed! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в CP_866.md' file was NOT processed!
>> smoke_test.log
)

IF EXIST "OUT\«Мелкий» файл в KOI8_R.md" (
 echo OK! '«Мелкий» файл в KOI8_R.md' file was processed! >>
smoke_test.log
) ELSE (
```

```
  echo ERROR! '«Мелкий» файл в KOI8_R.md' file was NOT processed!
>> smoke_test.log
)

IF EXIST "OUT\Слишком большой файл.txt" (
 echo ERROR! 'Too big' file was processed! >> smoke_test.log
) ELSE (
  echo OK! 'Too big' file was NOT processed! >> smoke_test.log
)

IF EXIST "OUT\Картинка.jpg" (
 echo ERROR! Picture file was processed! >> smoke_test.log
) ELSE (
  echo OK! Picture file was NOT processed! >> smoke_test.log
)

IF EXIST "OUT\Картинка в виде TXT.txt" (
 echo  OK!  Picture  file  with  TXT  extension  was  processed!  >>
smoke_test.log
) ELSE (
  echo  ERROR!  Picture  file  with  TXT  extension  was  NOT  processed!
>> smoke_test.log
)

IF EXIST "OUT\Пустой файл.md" (
 echo OK! Empty was processed! >> smoke_test.log
) ELSE (
  echo ERROR! Empty file was NOT processed! >> smoke_test.log
)
rem ============================================================
rem ============================================================
rem Проверка удаления из входного каталога файлов,
rem которые должны быть обработаны,
rem и неудаления файлов, которые не должны быть обработаны:
echo. >> smoke_test.log
echo Moving test: >> smoke_test.log

IF NOT EXIST "IN\«Мелкий» файл в WIN1251.txt" (
 echo  OK!  '«Мелкий»  файл  в  WIN1251.txt'  file  was  moved!  >>
smoke_test.log
) ELSE (
  echo ERROR! '«Мелкий» файл в WIN1251.txt' file was NOT moved! >>
smoke_test.log
)

IF NOT EXIST "IN\«Средний» файл в CP866.txt" (
 echo  OK!  '«Средний»  файл  в  CP866.txt'  file  was  moved!  >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в CP866.txt' file was NOT moved! >>
smoke_test.log
)
```

```
IF NOT EXIST "IN\«Крупный» файл в KOI8R.txt" (
 echo OK! '«Крупный» файл в KOI8R.txt' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в KOI8R.txt' file was NOT moved! >>
smoke_test.log
)

IF NOT EXIST "IN\«Крупный» файл в win-1251.html" (
 echo OK! '«Крупный» файл в win-1251.html' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в win-1251.html' file was NOT moved!
>> smoke_test.log
)

IF NOT EXIST "IN\«Мелкий» файл в cp-866.html" (
 echo OK! '«Мелкий» файл в cp-866.html' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Мелкий» файл в cp-866.html' file was NOT moved! >>
smoke_test.log
)

IF NOT EXIST "IN\«Средний» файл в koi8-r.html" (
 echo OK! '«Средний» файл в koi8-r.html' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в koi8-r.html' file was NOT moved!
>> smoke_test.log
)
IF NOT EXIST "IN\«Средний» файл в WIN_1251.md" (
 echo OK! '«Средний» файл в WIN_1251.md' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Средний» файл в WIN_1251.md' file was NOT moved!
>> smoke_test.log
)

IF NOT EXIST "IN\«Крупный» файл в CP_866.md" (
 echo OK! '«Крупный» файл в CP_866.md' file was moved! >>
smoke_test.log
) ELSE (
  echo ERROR! '«Крупный» файл в CP_866.md' file was NOT moved! >>
smoke_test.log
)

IF NOT EXIST "IN\«Мелкий» файл в KOI8_R.md" (
 echo OK! '«Мелкий» файл в KOI8_R.md' file was moved! >>
smoke_test.log
) ELSE (
```

```
  echo ERROR! '«Мелкий» файл в KOI8_R.md' file was NOT moved! >>
smoke_test.log
)
IF NOT EXIST "IN\Слишком большой файл.txt" (
 echo ERROR! 'Too big' file was moved! >> smoke_test.log
) ELSE (
  echo OK! 'Too big' file was NOT moved! >> smoke_test.log
)

IF NOT EXIST "IN\Картинка.jpg" (
 echo ERROR! Picture file was moved! >> smoke_test.log
) ELSE (
  echo OK! Picture file was NOT moved! >> smoke_test.log
)

IF NOT EXIST "IN\Картинка в виде TXT.txt" (
 echo  OK!  Picture  file  with  TXT  extension  was  moved!  >>
smoke_test.log
) ELSE (
  echo ERROR! Picture file with TXT extension was NOT moved! >>
smoke_test.log
)
rem ============================================================
cls
rem ============================================================
rem Проверка конвертации файлов путём сравнения
rem результатов работы приложения с эталонными файлами:
echo. >> smoke_test.log
echo Comparing test: >> smoke_test.log

:st1
fc "Test_ETALON\«Мелкий» эталон WIN1251.txt" "OUT\«Мелкий» файл в
WIN1251.txt" /B > nul
IF ERRORLEVEL 1 GOTO st1_fail
echo OK! File '«Мелкий» файл в WIN1251.txt' was processed correct-
ly! >> smoke_test.log
GOTO st2
:st1_fail
echo ERROR! File '«Мелкий» файл в WIN1251.txt' was NOT processed
correctly! >> smoke_test.log

:st2
fc "Test_ETALON\«Средний» эталон CP866.txt" "OUT\«Средний» файл в
CP866.txt" /B  > nul
IF ERRORLEVEL 1 GOTO st2_fail
echo OK! File '«Средний» файл в CP866.txt' was processed correct-
ly! >> smoke_test.log
GOTO st3
:st2_fail
echo ERROR! File '«Средний» файл в CP866.txt' was NOT processed
correctly! >> smoke_test.log
```

```
:st3
fc "Test_ETALON\«Крупный» эталон KOI8R.txt" "OUT\«Крупный» файл в
KOI8R.txt" /B  > nul
IF ERRORLEVEL 1 GOTO st3_fail
echo OK! File '«Крупный» файл в KOI8R.txt' was processed correct-
ly! >> smoke_test.log
GOTO st4
:st3_fail
echo ERROR! File '«Крупный» файл в KOI8R.txt' was NOT processed
correctly! >> smoke_test.log

:st4
fc "Test_ETALON\«Крупный» эталон в win-1251.html" "OUT\«Крупный»
файл в win-1251.html" /B  > nul
IF ERRORLEVEL 1 GOTO st4_fail
echo OK! File '«Крупный» файл в win-1251.html' was processed cor-
rectly! >> smoke_test.log
GOTO st5
:st4_fail
echo ERROR! File '«Крупный» файл в win-1251.html' was NOT pro-
cessed correctly! >> smoke_test.log

:st5
fc "Test_ETALON\«Мелкий» эталон в cp-866.html" "OUT\«Мелкий» файл
в cp-866.html" /B  > nul
IF ERRORLEVEL 1 GOTO st5_fail
echo OK! File '«Мелкий» файл в cp-866.html' was processed correct-
ly! >> smoke_test.log
GOTO st6
:st5_fail
echo ERROR! File '«Мелкий» файл в cp-866.html' was NOT processed
correctly! >> smoke_test.log

:st6
fc "Test_ETALON\«Средний» эталон в koi8-r.html" "OUT\«Средний»
файл в koi8-r.html" /B  > nul
IF ERRORLEVEL 1 GOTO st6_fail
echo OK! File '«Средний» файл в koi8-r.html' was processed cor-
rectly! >> smoke_test.log
GOTO st7
:st6_fail
echo ERROR! File '«Средний» файл в koi8-r.html' was NOT processed
correctly! >> smoke_test.log

:st7
fc "Test_ETALON\«Средний» эталон в WIN_1251.md" "OUT\«Средний»
файл в WIN_1251.md" /B  > nul
IF ERRORLEVEL 1 GOTO st7_fail
echo OK! File '«Средний» файл в WIN_1251.md' was processed cor-
rectly! >> smoke_test.log
GOTO st8
:st7_fail
```

```
echo ERROR! File '«Средний» файл в WIN_1251.md' was NOT processed correctly! >> smoke_test.log

:st8
fc "Test_ETALON\«Крупный» эталон в CP_866.md" "OUT\«Крупный» файл в CP_866.md" /B  > nul
IF ERRORLEVEL 1 GOTO st8_fail
echo OK! File '«Крупный» файл в CP_866.md' was processed correctly! >> smoke_test.log
GOTO st9
:st8_fail
echo ERROR! File '«Крупный» файл в CP_866.md' was NOT processed correctly! >> smoke_test.log

:st9
fc "Test_ETALON\«Мелкий» эталон в KOI8_R.md" "OUT\«Мелкий» файл в KOI8_R.md" /B  > nul
IF ERRORLEVEL 1 GOTO st9_fail
echo OK! File '«Мелкий» файл в KOI8_R.md' was processed correctly! >> smoke_test.log
GOTO st10
:st9_fail
echo ERROR! File '«Мелкий» файл в KOI8_R.md' was NOT processed correctly! >> smoke_test.log

:st10
fc "Test_ETALON\Пустой файл.md" "OUT\Пустой файл.md" /B  > nul
IF ERRORLEVEL 1 GOTO st10_fail
echo OK! File 'Пустой файл.md' was processed correctly! >> smoke_test.log
GOTO end
:st10_fail
echo ERROR! File 'Пустой файл.md' was NOT processed correctly! >> smoke_test.log

:end
echo WARNING! File 'Картинка в виде TXT.txt' has NO etalon decision, and it's OK for this file to be corrupted. >> smoke_test.log
rem ============================================================
```

Bash-скрипт для Linux

```
#!/bin/bash

# Удаление файла журнала от прошлого запуска:
rm -f smoke_test.log

# Очистка входного каталога приложения:
rm -r -f IN/*

# Запуск приложения:
php converter.php IN OUT converter.log &
```

```
# Размещение тестовых файлов во входном каталоге приложения:
cp Test_IN/* IN/

# Тайм-аут в 10 секунд, чтобы приложение успело обработать файлы:
sleep 10
# Остановка приложения:
killall php
# ================================================================
# Проверка появления в выходном каталоге файлов, которые должны
быть обработаны,
# и непоявления файлов, которые не должны быть обработаны:
echo "Processing test:" >> smoke_test.log

if [ -f "OUT/«Мелкий» файл в WIN1251.txt" ]
then
  echo "OK! '«Мелкий» файл в WIN1251.txt' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в WIN1251.txt' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Средний» файл в CP866.txt" ]
then
  echo "OK! '«Средний» файл в CP866.txt' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в CP866.txt' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Крупный» файл в KOI8R.txt" ]
then
  echo "OK! '«Крупный» файл в KOI8R.txt' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Крупный» файл в KOI8R.txt' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Крупный» файл в win-1251.html" ]
then
  echo "OK! '«Крупный» файл в win-1251.html' file was processed!"
>> smoke_test.log
else
  echo "ERROR! '«Крупный» файл в win-1251.html' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Мелкий» файл в cp-866.html" ]
then
  echo "OK! '«Мелкий» файл в cp-866.html' file was processed!" >>
```

```
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в cp-866.html' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Средний» файл в koi8-r.html" ]
then
  echo "OK! '«Средний» файл в koi8-r.html' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в koi8-r.html' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Средний» файл в WIN_1251.md" ]
then
  echo "OK! '«Средний» файл в WIN_1251.md' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в WIN_1251.md' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Крупный» файл в CP_866.md" ]
then
  echo "OK! '«Крупный» файл в CP_866.md' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Крупный» файл в CP_866.md' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/«Мелкий» файл в KOI8_R.md" ]
then
  echo "OK! '«Мелкий» файл в KOI8_R.md' file was processed!" >>
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в KOI8_R.md' file was NOT pro-
cessed!" >> smoke_test.log
fi

if [ -f "OUT/Слишком большой файл.txt" ]
then
  echo "ERROR! 'Too big' file was processed!" >> smoke_test.log
else
  echo "OK! 'Too big' file was NOT processed!" >> smoke_test.log
fi

if [ -f "OUT/Картинка.jpg" ]
then
  echo "ERROR! Picture file was processed!" >> smoke_test.log
```

```
else
  echo "OK! Picture file was NOT processed!" >> smoke_test.log
fi

if [ -f "OUT/Картинка в виде TXT.txt" ]
then
  echo "OK! Picture file with TXT extension was processed!" >>
smoke_test.log
else
  echo "ERROR! Picture file with TXT extension was NOT processed!"
>> smoke_test.log
fi

if [ -f "OUT/Пустой файл.md" ]
then
  echo "OK! Empty file was processed!" >> smoke_test.log
else
  echo "ERROR! Empty file was NOT processed!" >> smoke_test.log
fi
# ================================================================
# ================================================================
# Проверка удаления из входного каталога файлов, которые должны
быть обработаны,
# и неудаления файлов, которые не должны быть обработаны:
echo "" >> smoke_test.log
echo "Moving test:" >> smoke_test.log

if [ ! -f "IN/«Мелкий» файл в WIN1251.txt" ]
then
  echo "OK! '«Мелкий» файл в WIN1251.txt' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в WIN1251.txt' file was NOT moved!"
>> smoke_test.log
fi

if [ ! -f "IN/«Средний» файл в CP866.txt" ]
then
  echo "OK! '«Средний» файл в CP866.txt' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в CP866.txt' file was NOT moved!"
>> smoke_test.log
fi
if [ ! -f "IN/«Крупный» файл в KOI8R.txt" ]
then
  echo "OK! '«Крупный» файл в KOI8R.txt' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Крупный» файл в KOI8R.txt' file was NOT moved!"
>> smoke_test.log
fi
```

```
if [ ! -f "IN/«Крупный» файл в win-1251.html" ]
then
  echo "OK! '«Крупный» файл в win-1251.html' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Крупный» файл в win-1251.html' file was NOT
moved!" >> smoke_test.log
fi

if [ ! -f "IN/«Мелкий» файл в cp-866.html" ]
then
  echo "OK! '«Мелкий» файл в cp-866.html' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в cp-866.html' file was NOT moved!"
>> smoke_test.log
fi

if [ ! -f "IN/«Средний» файл в koi8-r.html" ]
then
  echo "OK! '«Средний» файл в koi8-r.html' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в koi8-r.html' file was NOT moved!"
>> smoke_test.log
fi

if [ ! -f "IN/«Средний» файл в WIN_1251.md" ]
then
  echo "OK! '«Средний» файл в WIN_1251.md' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Средний» файл в WIN_1251.md' file was NOT moved!"
>> smoke_test.log
fi

if [ ! -f "IN/«Крупный» файл в CP_866.md" ]
then
  echo "OK! '«Крупный» файл в CP_866.md' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Крупный» файл в CP_866.md' file was NOT moved!"
>> smoke_test.log
fi

if [ ! -f "IN/«Мелкий» файл в KOI8_R.md" ]
then
  echo "OK! '«Мелкий» файл в KOI8_R.md' file was moved!" >>
smoke_test.log
else
  echo "ERROR! '«Мелкий» файл в KOI8_R.md' file was NOT moved!" >>
```

```
smoke_test.log
fi

if [ ! -f "IN/Слишком большой файл.txt" ]
then
  echo "ERROR! 'Too big' file was moved!" >> smoke_test.log
else
  echo "OK! 'Too big' file was NOT moved!" >> smoke_test.log
fi

if [ ! -f "IN/Картинка.jpg" ]
then
  echo "ERROR! Picture file was moved!" >> smoke_test.log
else
  echo "OK! Picture file was NOT moved!" >> smoke_test.log
fi

if [ ! -f "IN/Картинка в виде TXT.txt" ]
then
  echo "OK! Picture file with TXT extension was moved!" >>
smoke_test.log
else
  echo "ERROR! Picture file with TXT extension was NOT moved!" >>
smoke_test.log
fi

if [ ! -f C"IN/Пустой файл.md" ]
then
  echo "OK! Empty file was moved!" >> smoke_test.log
else
  echo "ERROR! Empty file was NOT moved!" >> smoke_test.log
fi

# ================================================================
clear
# ================================================================
# Проверка конвертации файлов путём сравнения результатов
# работы приложения с эталонными файлами:
echo "" >> smoke_test.log
echo "Comparing test:" >> smoke_test.log

if cmp -s "Test_ETALON/«Мелкий» эталон WIN1251.txt" "OUT/«Мелкий»
файл в WIN1251.txt"
then
  echo "OK! File '«Мелкий» файл в WIN1251.txt' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Мелкий» файл в WIN1251.txt' was NOT pro-
cessed correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Средний» эталон CP866.txt" "OUT/«Средний»
```

```
файл CP866.txt"
then
  echo "OK! File '«Средний» файл CP866.txt' was processed correct-
ly!" >> smoke_test.log
else
  echo "ERROR! File '«Средний» файл CP866.txt' was NOT processed
correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Крупный» эталон KOI8R.txt" "OUT/«Крупный»
файл KOI8R.txt"
then
  echo "OK! File '«Крупный» файл KOI8R.txt' was processed correct-
ly!" >> smoke_test.log
else
  echo "ERROR! File '«Крупный» файл KOI8R.txt' was NOT processed
correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Крупный» файл в win-1251.html"
"OUT/«Крупный» файл в win-1251.html"
then
  echo "OK! File '«Крупный» файл в win-1251.html' was processed
correctly!" >> smoke_test.log
else
  echo "ERROR! File '«Крупный» файл в win-1251.html' was NOT pro-
cessed correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Мелкий» эталон в cp-866.html"
"OUT/«Мелкий» файл в cp-866.html"
then
  echo "OK! File '«Мелкий» файл в cp-866.html' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Мелкий» файл в cp-866.html' was NOT pro-
cessed correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Средний» эталон в koi8-r.html"
"OUT/«Средний» файл в koi8-r.html"
then
  echo "OK! File '«Средний» файл в koi8-r.html' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Средний» файл в koi8-r.html' was NOT pro-
cessed correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Средний» эталон в WIN_1251.md"
"OUT/«Средний» файл в WIN_1251.md"
then
```

```
  echo "OK! File '«Средний» файл в WIN_1251.md' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Средний» файл в WIN_1251.md' was NOT pro-
cessed correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Крупный» эталон в CP_866.md"
"OUT/«Крупный» файл в CP_866.md"
then
  echo "OK! File '«Крупный» файл в CP_866.md' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Крупный» файл в CP_866.md' was NOT processed
correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/«Мелкий» эталон в KOI8_R.md" "OUT/«Мелкий»
файл в KOI8_R.md"
then
  echo "OK! File '«Мелкий» файл в KOI8_R.md' was processed cor-
rectly!" >> smoke_test.log
else
  echo "ERROR! File '«Мелкий» файл в KOI8_R.md' was NOT processed
correctly!" >> smoke_test.log
fi

if cmp -s "Test_ETALON/Пустой файл.md" "OUT/Пустой файл.md"
then
  echo "OK! File 'Пустой файл.md' was processed correctly!" >>
smoke_test.log
else
  echo "ERROR! File 'Пустой файл.md' was NOT processed correctly!"
>> smoke_test.log
fi

echo "WARNING! File 'Картинка в виде TXT.txt' has NO etalon deci-
sion, and it's OK for this file to be corrupted." >>
smoke_test.log
# ============================================================
```

Пример результатов выполнения (на одном из первых билдов, содержащих множество дефектов)

```
Processing test:
OK! '«Мелкий» файл в WIN1251.txt' file was processed!
OK! '«Средний» файл в CP866.txt' file was processed!
OK! '«Крупный» файл в KOI8R.txt' file was processed!
OK! '«Крупный» файл в win-1251.html' file was processed!
OK! '«Мелкий» файл в cp-866.html' file was processed!
OK! '«Средний» файл в koi8-r.html' file was processed!
OK! '«Средний» файл в WIN_1251.md' file was processed!
OK! '«Крупный» файл в CP_866.md' file was processed!
```

```
OK! '«Мелкий» файл в KOI8_R.md' file was processed!
OK! 'Too big' file was NOT processed!
OK! Picture file was NOT processed!
OK! Picture file with TXT extension was processed!

Moving test:
ERROR! '«Мелкий» файл в WIN1251.txt' file was NOT moved!
ERROR! '«Средний» файл в CP866.txt' file was NOT moved!
ERROR! '«Крупный» файл в KOI8R.txt' file was NOT moved!
ERROR! '«Крупный» файл в win-1251.html' file was NOT moved!
ERROR! '«Мелкий» файл в cp-866.html' file was NOT moved!
ERROR! '«Средний» файл в koi8-r.html' file was NOT moved!
ERROR! '«Средний» файл в WIN_1251.md' file was NOT moved!
ERROR! '«Крупный» файл в CP_866.md' file was NOT moved!
ERROR! '«Мелкий» файл в KOI8_R.md' file was NOT moved!
OK! 'Too big' file was NOT moved!
OK! Picture file was NOT moved!
ERROR! Picture file with TXT extension was NOT moved!

Comparing test:
ERROR! File '«Мелкий» файл в WIN1251.txt' was NOT processed cor-
rectly!
ERROR! File '«Средний» файл в CP866.txt' was NOT processed cor-
rectly!
ERROR! File '«Крупный» файл в KOI8R.txt' was NOT processed cor-
rectly!
ERROR! File '«Крупный» файл в win-1251.html' was NOT processed
correctly!
ERROR! File '«Мелкий» файл в cp-866.html' was NOT processed cor-
rectly!
ERROR! File '«Средний» файл в koi8-r.html' was NOT processed cor-
rectly!
ERROR! File '«Средний» файл в WIN_1251.md' was NOT processed cor-
rectly!
ERROR! File '«Крупный» файл в CP_866.md' was NOT processed cor-
rectly!
ERROR! File '«Мелкий» файл в KOI8_R.md' was NOT processed correct-
ly!
OK! File 'Пустой файл.md' was processed correctly!
WARNING! File 'Картинка в виде TXT.txt' has NO etalon decision,
and it's OK for this file to be corrupted.
```

ПРИЛОЖЕНИЕ Б

(справочное)

Пример данных для попарного тестирования

Расположение/ длина/значение/комбинация символов/ зарезервированное или свободное	Существование	Наличие прав доступа	Семейство ОС	Кодировки
1. X:\	Да	К каталогу и его содержимому	Windows 64 bit	UTF8
2. smb://host/dir	Нет		Linux 32 bit	UTF16
3. …/dir	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	OEM
4. [257 символов только для Windows]	Да	Только к каталогу	Windows 64 bit	OEM
5. smb://host/dir/	Да	К каталогу и его содержимому	Linux 64 bit	UTF8
6. nul	Да	Ни к каталогу, ни к его содержимому	Windows 64 bit	OEM
7. \\	Нет		Linux 64 bit	UTF16
8. /dir	Да	Ни к каталогу, ни к его содержимому	Linux 32 bit	OEM
9. ./dir/	Нет		Linux 32 bit	OEM
10. ./dir	Нет	К каталогу и его содержимому	Linux 64 bit	UTF8
11. smb://host/dir	Да	Только к каталогу	Linux 64 bit	UTF8
12. \\host\dir\	Да	К каталогу и его содержимому	Linux 32 bit	UTF8
13. host:/dir	Нет		Windows 32 bit	UTF8
14. .\dir\	Нет		Windows 64 bit	UTF8
15. [0 символов]	Нет		Windows 32 bit	UTF16
16. [4097 символов только для Linux]	Нет		Linux 32 bit	UTF16
17. ...\dir\	Нет		Windows 32 bit	UTF16
18. "/пробелы и русский/"	Да	К каталогу и его содержимому	Windows 32 bit	OEM
19. smb://host/dir/	Да	Только к каталогу	Linux 32 bit	OEM
20. nul	Да		Windows 32 bit	UTF8
21. "/пробелы и русский"	Нет		Linux 32 bit	OEM
22. host:/dir/	Да	Только к каталогу	Windows 64 bit	UTF8
23. …/dir	Нет		Windows 64 bit	UTF16
24. ./dir/	Нет		Linux 64 bit	UTF16
25. [257 символов только для Windows]	Нет		Windows 32 bit	UTF16

26. "/пробелы и русский/"	Нет		Linux 64 bit	UTF8
27. ...	Нет		Windows 32 bit	UTF8
28. host:/dir/	Нет		Linux 64 bit	OEM
29. X:\dir\	Да	К каталогу и его содержимому	Windows 64 bit	UTF8
30. \\	Да	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF8
31. //	Нет	Только к каталогу	Windows 64 bit	UTF8
32. ...\dir\	Нет	Ни к каталогу, ни к его содержимому	Windows 64 bit	OEM
33. X:\dir	Нет	Только к каталогу	Windows 64 bit	OEM
34. "X:\пробелы и русский\"	Да	Только к каталогу	Windows 64 bit	UTF16
35. \\host\dir\	Нет	Только к каталогу	Windows 32 bit	UTF16
36. [256 символов только для Windows]	Да	К каталогу и его содержимому	Windows 32 bit	UTF8
37. [4096 символов только для Linux]	Нет	Только к каталогу	Linux 64 bit	UTF16
38. /dir/	Да	К каталогу и его содержимому	Linux 64 bit	UTF8
39. [256 символов только для Windows]	Да	К каталогу и его содержимому	Windows 64 bit	OEM
40. .\dir	Нет	К каталогу и его содержимому	Windows 32 bit	UTF16
41. //	Да	Ни к каталогу, ни к его содержимому	Windows 32 bit	OEM
42. prn	Да	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF16
43. …\dir	Нет	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF16
44. \\host\dir\	Нет	Только к каталогу	Windows 64 bit	UTF16
45. …/dir/	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	UTF8
46. ...	Да	Только к каталогу	Linux 32 bit	OEM
47. ...\dir	Да	Только к каталогу	Windows 32 bit	UTF8
48. /dir	Да	Только к каталогу	Linux 64 bit	UTF8
49. "	Нет	Только к каталогу	Windows 32 bit	UTF8
50. …/dir/	Нет	К каталогу и его содержимому	Linux 32 bit	UTF16
51. .\dir	Да	Только к каталогу	Windows 64 bit	OEM
52. host:/dir/	Нет	Ни к каталогу, ни к его содержимому	Linux 32 bit	UTF16
53. "/пробелы и русский"	Нет	К каталогу и его содержимому	Linux 64 bit	UTF16
54. com1-com9	Да	Ни к каталогу, ни к его содержимому	Windows 64 bit	UTF16
55. lpt1-lpt9	Да	Только к каталогу	Windows 32 bit	UTF8
56. [0 символов]	Нет	Только к каталогу	Linux 64 bit	UTF16

57. \\host\dir	Да	Ни к каталогу, ни к его содержимому	Windows 32 bit	UTF16
58. "X:\пробелы и русский"	Да	Только к каталогу	Windows 64 bit	UTF16
59. \\host\dir	Нет	Только к каталогу	Linux 64 bit	UTF8
60. lpt1-lpt9	Да	Только к каталогу	Windows 64 bit	UTF8
61. "X:\пробелы и русский"	Нет	К каталогу и его содержимому	Windows 32 bit	OEM
62. host:/dir	Да	К каталогу и его содержимому	Linux 32 bit	OEM
63. X:\	Да	Только к каталогу	Windows 32 bit	OEM
64. \\	Нет	Только к каталогу	Windows 32 bit	OEM
65. [4096 символов только для Linux]	Да	К каталогу и его содержимому	Linux 32 bit	UTF8
66. \\host\dir	Нет	К каталогу и его содержимому	Windows 64 bit	OEM
67. "	Нет	Ни к каталогу, ни к его содержимому	Linux 32 bit	OEM
68. con	Нет	К каталогу и его содержимому	Windows 32 bit	UTF16
69. …/dir	Нет	Только к каталогу	Linux 32 bit	UTF16
70. X:\dir	Да	К каталогу и его содержимому	Windows 32 bit	OEM
71. ./dir	Да	К каталогу и его содержимому	Linux 32 bit	UTF16
72. //	Да	К каталогу и его содержимому	Linux 32 bit	UTF16
73. host:/dir	Нет	Ни к каталогу, ни к его содержимому	Linux 64 bit	UTF8
74. /	Нет	К каталогу и его содержимому	Linux 64 bit	UTF8
75. "X:\пробелы и русский\"	Да	Ни к каталогу, ни к его содержимому	Windows 32 bit	OEM
76. .\dir\	Да	Ни к каталогу, ни к его содержимому	Windows 32 bit	OEM
77. //	Нет	Только к каталогу	Linux 64 bit	OEM
78. X:\dir\	Да	Только к каталогу	Windows 32 bit	UTF8
79. "	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	UTF16
80. /	Да	К каталогу и его содержимому	Linux 32 bit	UTF16
81. ...	Да	К каталогу и его содержимому	Windows 64 bit	UTF16
82. com1-com9	Да	Ни к каталогу, ни к его содержимому	Windows 32 bit	OEM
83. ...	Да	Ни к каталогу, ни к его содержимому	Linux 64 bit	OEM
84. /dir/	Да	К каталогу и его содержимому	Linux 32 bit	UTF16
85. [4097 символов только для Linux]	Нет	К каталогу и его содержимому	Linux 64 bit	UTF16

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ammann, P. Introduction to Software Testing. Chapter 4. Input Space Partition Testing / P. Ammann, O. Jeff. – Saarbrücken : LAP Lambert Academic Publishing, 2012. – 56 p.
2. Barker, T. T. Documentation for software and IS development. Encyclopedia of Information Systems / T. T. Barker. – Amsterdam : Elsevier Press, 2002. – P. 679–680.
3. Barker T. T. Writing Software Documentation : A Task-Oriented Approach (Part of the Allyn Bacon Series in Technical Communication) / T. T. Barker. – 2nd ed. – Harlow : Longman Publishing Group, 2002. – 496 p.
4. Documenting Software Architectures / P. Clements [at al.]. – Boston : Addison-Wesley Professional, 2003. – 512 p.
5. Copeland, L. A practitioner's guide to software test design / L. Copeland. – Norwood : Artech House, 2004. – 300 p.
6. Crispin, L. Agile Testing / L. Crispin, J. Gregory. Boston : Addison-Wesley, 2009. – 533 p.
7. Hass, A. M. Guide to Advanced Software Testing / A. M. Hass. – 2nd ed. – Norwood : Artech House Publishers, 2014. – 482 p.
8. Jyoti, J. M. Software Testing and Quality Assurance / J. M. Jyoti, S. T. Bhavana. – Hoboken : John Wiley & Sons, Inc., 2005. – 441 p.
9. Kerzner, H. Project Management : A Systems Approach to Planning, Scheduling, and Controlling / H. Kerzner. – Hoboken : Wiley, 2013. – 1296 p.
10. Nageshwar, R. P. Software Testing Concepts And Tools / R. P. Nageshwar. – M. : Dreamtech, 2006. – 492 p.
11. Ауэр, К. Экстремальное программирование: постановка процесса. С первых шагов и до победного конца. Сер. Библиотека программиста / К. Ауэр, Р. Миллер. – СПб. : Питер, 2004. – 368 с.
12. Бейзер, Б. Тестирование черного ящика. Технологии функционального тестирования ПО. Сер. Библиотека программиста / Б. Бейзер. – СПб. : Питер, 2004. – 320 с.
13. Бек, К. Экстремальное программирование: разработка через тестирование. Сер. Библиотека программиста / К. Бек. – СПб. : Питер, 2003. – 224 с.
14. Блэк, Р. Ключевые процессы тестирования. Планирование, подготовка, проведение, совершенствование / Р. Блэк. – М. : Лори, 2011. – 544 с.
15. Браун, К. Быстрое тестирование / К. Браун, Р. Калбертсон, Г. Кобб. – М. : Вильямс, 2002. – 384 с.
16. Вигерс, К. Разработка требований к программному обеспечению (Software Requirements, Third Ed.) / К. Вигерс, Д. Битти. – 3-е изд. – М. : Русская редакция, 2014. – 737с.
17. Дастин, Э. Автоматизированное тестирование программного обеспечения / Э. Дастин, Д. Рэшка. – М. : Лори, 2005. – 592 с.
18. Канер, С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений / С. Канер, Д. Фолк, Е. К. Нгуен. – Ярославль : ДиаСофт, 2001. – 538 с.
19. Котляров, В. П. Основы тестирования программного обеспечения / В. П. Котляров. – М. : «Интернет-университет информационных технологий ИНТУИТ.ру, 2006. – 285 с.

20. Майерс, Г. Искусство тестирования программ / Г. Майерс, Т. Баджетт, К. Сандлер. – 3-е изд. – М. : Вильямс, 2012. – 272 с.
21. Макгрегор, Д. Тестирование объектно-ориентированного программного обеспечения : практ. пособие / Д. Макгрегор, Д. Сайкс. – Ярославль : Диасофт, 2002. – 432 с.
22. Макконнелл, С. Совершенный код. Русская редакция. Сер. Мастер-класс / С. Макконнелл. – 2-е изд. – СПб. : Питер, 2005. – 896 с.
23. Роббинс, Д. Отладка приложений. Сер. Мастер / Д. Роббинс. – СПб. : BHV, 2001. – 512 с.
24. Савин, Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах / Р. Савин. – М. : Дело, 2007. – 312 с.
25. Стотлемайер, Д. Тестирование Web-приложений / Д. Стотлемайер. – М. : Кудиц-образ. 2003. – 240 с.
26. Тамре, Л. Введение в тестирование программного обеспечения / Л. Тамре. – М. : Вильямс, 2003. – 359 с.
27. Торрес, Дж. Практическое руководство по проектированию и разработке пользовательского интерфейса / Дж. Торрес. – М. : Вильямс, 2003. – 400 с.
28. Beginner’s Guide to Mobile Application Testing [Электронный ресурс]. – 2016. – Режим доступа : http://www.softwaretestinghelp.com/beginners-guide-to-mobile-application-testing/.
29. Project Lifecycle Models: How They Differ and When to Use Them [Электронный ресурс] . – 2002. – Режим доступа : http://www.business-esolutions. com/islm.htm.
30. Smoke testing and sanity testing – Quick and simple differences [Электронный ресурс]. – 2016. – Режим доступа : http://www.softwaretestinghelp.com/smoke-testing-and-sanity-testing-difference/.
31. Westfall, L. Software Requirements Engineering: What, Why, Who, When, and How / L. Westfall [Электронный ресурс]. – 2015. – Режим доступа : http://www.westfallteam.com/Papers/The_Why_What_Who_When_and_How_Of_Software_Requirements.pdf.
32. Блок-схема выбора оптимальной методологии разработки ПО [Электронный ресурс]. – 2015. – Режим доступа : http://megamozg.ru/post/23022/.
33. Blain, T. Writing Good Requirements – The Big Ten Rules / T. Blain [Электронный ресурс]. – 2006. – Режим доступа : http://tynerblain.com/blog/2006/05/25/writing-good-requirements-the-big-ten-rules/.
34. Boehm, B. A Spiral Model of Software Development and Enhancement / B. Boehm [Электронный ресурс]. – 1988. – Режим доступа : http://csse.usc.edu/csse/TECHRPTS/1988/usccse88-500/usccse88-500.pdf.
35. Boehm, B. Spiral Development: Experience, Principles, and Refinements / B. Boehm [Электронный ресурс]. – 1988. – Режим доступа : http://www.sei.cmu.edu/reports/00sr008.pdf.
36. Brandenburg, L. Requirements Gathering vs. Elicitation / L. Brandenburg [Электронный ресурс]. – 2006. – Режим доступа : http://www.bridging-the-gap.com/requirements-gathering-vs-elicitation/.

37. Cohen, J. Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.) / J. Cohen [Электронный ресурс]. – 2013. – Режим доступа : http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf.
38. Cross-Browser Testing – Overview [Электронный ресурс]. – 2016. – Режим доступа : http://support.smartbear.com/viewarticle/55299/.
39. Firesmith, D. Using V Models for Testing / D. Firesmith [Электронный ресурс]. – 2013. – Режим доступа : http://blog.sei.cmu.edu/post.cfm/using-v-models-testing-315.
40. Gelperin, D. The Growth of Software Testing / D. Gelperin, B. Hetzel [Электронный ресурс]. – 1988. – Режим доступа : http://www.clearspecs.com/downloads/ClearSpecs16V01_GrowthOfSoftwareTest.pdf.
41. Ghahrai, A. Spiral Model / A. Ghahrai [Электронный ресурс]. – 2002. – Режим доступа : http://www.testingexcellence.com/spiral-model/.
42. Hanks, B. Requirements in the Real World / B. Hanks [Электронный ресурс]. – 2002. – Режим доступа : https://classes.soe.ucsc.edu/cmps109/Winter02/notes/requirementsLecture.pdf.
43. Kaner, C. A Tutorial in Exploratory Testing / C. Kaner [Электронный ресурс]. – 2015. – Режим доступа : http://www.kaner.com/pdfs/QAIExploring.pdf.
44. Kaner, C. An Introduction to Scenario Testing / C. Kaner [Электронный ресурс]. – 2003. – Режим доступа : http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf.
45. Kuijt, R. Extended Use Case Test Design Pattern / R. Kuijt [Электронный ресурс]. – 2013. – Режим доступа : http://www.softwaretestingclass.com/positive-and-negative-testing-in-software-testing/.
46. Meerts, J. The History of Software Testing / J. Meerts [Электронный ресурс]. – Режим доступа : http://www.testingreferences.com/testinghistory.php.
47. Rangaiah, J. Behaviour driven testing an introduction / J. Rangaiah [Электронный ресурс]. – 2015. – Режим доступа : http://www.womentesters.com/behaviour-driven-testing-an-introduction/.
48. Rouse, M. Gray box testing (gray box) definition / M. Rouse [Электронный ресурс]. – 2013. – Режим доступа: http://searchsoftwarequality.techtarget.com/definition/gray-box.
49. SmartBear TestComplete user manual [Электронный ресурс]. – 2016. – Режим доступа : http://support.smartbear.com/viewarticle/55004/.
50. What are Alpha, Beta and Gamma Testing? [Электронный ресурс]. – 2012. – Режим доступа : http://www.360logica.com/blog/2012/06/what-are-alpha-beta-and-gamma-testing.htm.
51. Whittaker, J. The 7th Plague and Beyond / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/09/7th-plague-and-beyond.html.
52. Whittaker, J. The Plague of Amnesia / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/07/plague-of-amnesia.html.
53. Whittaker, J. The Plague of Blindness / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/07/plague-of-blindness.html.
54. Whittaker, J. The Plague of Boredom / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/07/plague-of-boredom.html.
55. Whittaker, J. The Plague of Entropy / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/09/plague-of-entropy.html.

56. Whittaker, J. The Plague of Homelessness / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/07/plague-of-homelessness.html.
57. Whittaker, J. The plague of repetitiveness / J. Whittaker [Электронный ресурс]. – 2009. – Режим доступа : http://googletesting.blogspot.com/2009/06/by-james.html.
58. Kelly, M. The ROI of Test Automation / M. Kelly [Электронный ресурс]. – 1999. – Режим доступа : http://www.sqetraining.com/sites/default/files/articles/XDD8502filelistfilename1_0.pdf.
59. The Return of Investment (ROI) of Test Automation / S. Münch [et al.] [Электронный ресурс]. – 2012. – Режим доступа : https://www.ispe.org/pe-ja/roi-of-test-automation.pdf.
60. Rooney, J. Root Cause Analysis for Beginners / J. Rooney, L. V. Heuvel [Электронный ресурс]. – 2012. – Режим доступа : https://www.env.nm.gov/aqb/Proposed_Regs/Part_7_Excess_Emissions/NMED_Exhibit_18-Root_Cause_Analysis_for_Beginners.pdf.
61. Garrett, T. Implementing Automated Software Testing – Continuously Track Progress and Adjust Accordingly / T. Garrett [Электронный ресурс]. – 2009. – Режим доступа : http://www.methodsandtools.com/archive/archive.php?id=94.
62. Patel, N. Test Case Point Analysis / N. Patel [Электронный ресурс]. – 2001. – Режим доступа : http://www.stickyminds.com/sites/default/files/article/file/2013/XUS373692file1_0.pdf.
63. Sherwood, G. On the Construction of Orthogonal Arrays and Covering Arrays Using Permutation Groups / G. Sherwood [Электронный ресурс]. – 2002. – Режим доступа : http://testcover.com/pub/background/cover.htm.
64. Bolton, M. Pairwise Testing / M. Bolton [Электронный ресурс]. – 2002. – Режим доступа : http://www.developsense.com/pairwiseTesting.html.
65. Kuo-Chung, T. A Test Generation Strategy for Pairwise Testing / T. Kuo-Chung, L. Yu [Электронный ресурс]. – 2002. – Режим доступа : http://www.cs.umd.edu/class/spring2003/cmsc838p/VandV/pairwise.pdf.
66. An Improved Test Generation Algorithm for Pair-Wise Testing / S. Maity [et al.] [Электронный ресурс]. – 2003. – Режим доступа : http://www.iiserpune.ac.in/~soumen/115-FA-2003.pdf.
67. Nageswaran, S. Test Effort Estimation Using Use Case Points / S. Nageswaran [Электронный ресурс]. – 2001. – Режим доступа : http://www.bfpug.com.br/Artigos/UCP/Nageswaran-Test_Effort_Estimation_Using_UCP.pdf.
68. Software Estimation Techniques – Common Test Estimation Techniques used in SDLC [Электронный ресурс]. – 2013. – Режим доступа : http://www.softwaretestingclass.com/software-estimation-techniques/.
69. Important Software Test Metrics and Measurements – Explained with Examples and Graphs [Электронный ресурс]. – 2016. – Режим доступа : http://www.softwaretestinghelp.com/software-test-metrics-and-measurements/.

Св. план 2018, поз.73

Учебное издание

Куликов Святослав Святославович
Данилова Галина Владимировна
Смолякова Ольга Георгиевна
Меженная Марина Михайловна

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

УЧЕБНОЕ ПОСОБИЕ

Редактор *Е. В. Иванюшина*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 13.05.2019. Формат 60x84 1/8. Бумага офсетная. Гарнитура «Arial».
Отпечатано на ризографе. Усл. печ. л. 32,55. Уч.-изд. л. 19,5. Тираж 250 экз. Заказ 382.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск