Тестирование ПО. Уровень 1

Классы эквивалентности и граничные условия. Классификация видов тестирования

Техники тест-дизайна для тест-кейсов. Определение и поиск классов эквивалентности. Границы классов эквивалентности. Классификация тестирования.

[Что такое тест-дизайн](#_4oa2692rzw31)

[Классы эквивалентных значений](#_z5ixuebccuo3)

[Пример 1](#_fkcgh2k9hzle)

[Пример 2](#_hazggf518vcg)

[Техника анализа граничных значений](#_88gwp58n7kbz)

[Попарное тестирование](#_an9bh3r7lc22)

[Классификация видов и направлений тестирования](#_afhjoc3460c)

[По запуску кода на исполнение](#_3znysh7)

[По доступу к коду и архитектуре приложения](#_2et92p0)

[По степени автоматизации](#_1iw0n530hm7p)

[По уровню детализации приложения](#_3dy6vkm)

[По степени важности тестируемых функций](#_1t3h5sf)

[По принципам работы с приложением](#_4d34og8)

[По целям и задачам тестирования](#_2s8eyo1)

[Логи и работа с ними](#_17dp8vu)

[Пример 1](#_m2xn6u7jk2mo)

[Пример 2](#_a2wgqpyt3i3v)

[Пример 3](#_pdgbgu626dn6)

[Глоссарий](#_ybu97jj96tj5)

[Практическое задание:](#_1ov219w383k0)

[Требования к выполненной работе](#_r0fswodhddza)

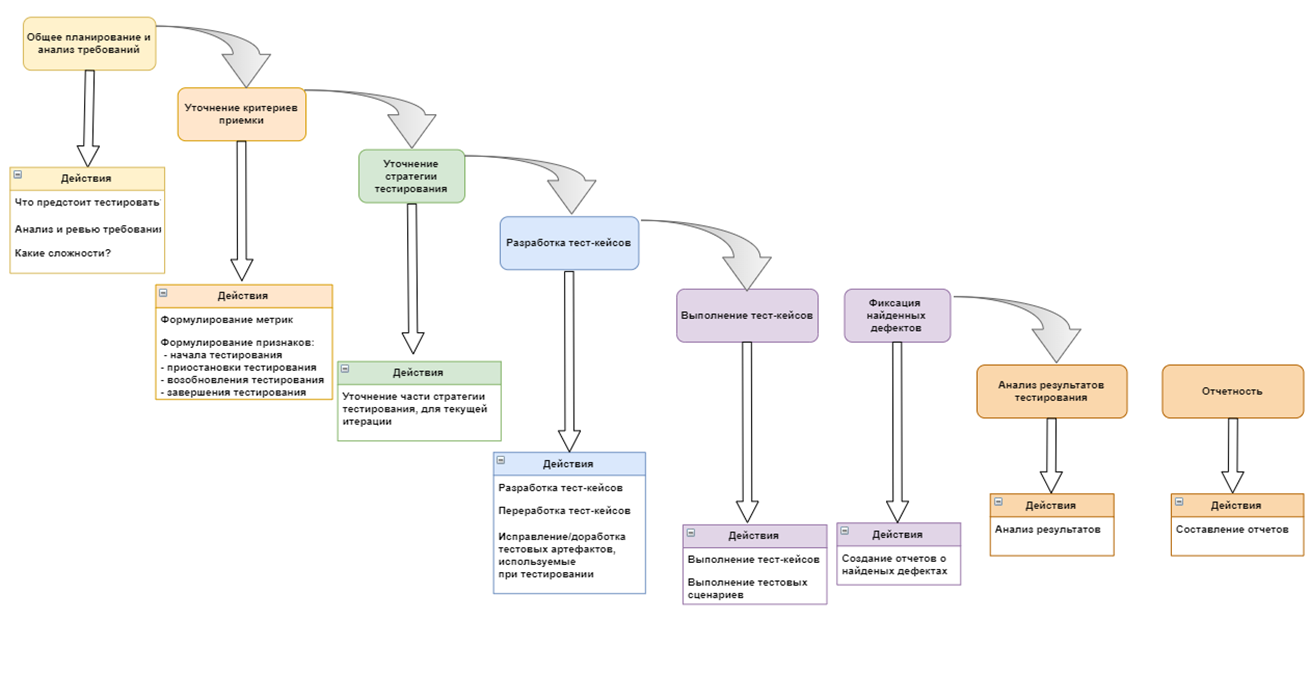
[Дополнительные материалы](#_1xrzud18jpi3)

[Используемые источники](#_4i7ojhp)

# Что такое тест-дизайн

Тест-дизайн — один из этапов процесса тестирования, на котором проектируются и создаются тест-кейсы с определёнными ранее целями тестирования.

Мы уже рассматривали жизненный цикл тестирования. Ещё раз вспомним, какие этапы в жизненном цикле тестирования, и на каком этапе представлен тест-дизайн.



Как вы видите, разработка тест-кейсов находится на четвёртом этапе. Поэтому перед тем, как начать создавать тест-кейсы, мы должны точно понимать, **что** мы будем тестировать и **какие** техники тест-дизайна уместны в данном случае при разработке тест-кейсов.

Мы рассмотрим только три техники тест-дизайна, более детальное их рассмотрение будет во втором уровне курса «Тестирование ПО».

Итак, **тест-дизайн или разработка тестов** — деятельность по получению или выделению тест-кейсов из определённых тестовых условий.

Разработка тест-кейсов базируется на техниках тестирования (методом черного ящика или белого ящика). В данном случае, под тестированием методом черного ящика имеется в виду метод тестирования без доступа к коду. Под тестированием методом белого ящика имеется в виду тестирование с доступом к программному коду.

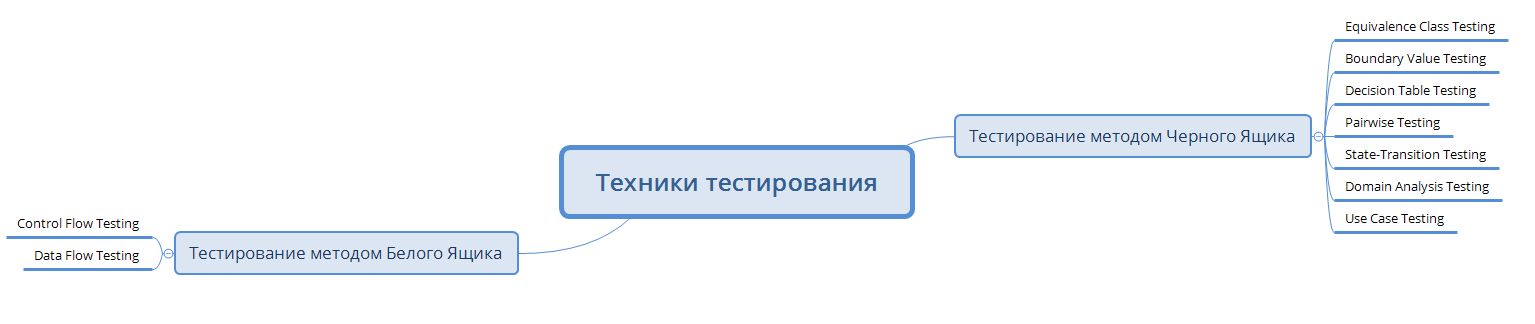
Согласно Ли Копланду (Lee Copeland — A Practitioner's Guide to Software Test Design) существуют следующие техники тестирования методом черного ящика:

* Тестирование классов эквивалентности (Equivalence Class Testing).
* Тестирование граничных значений (Boundary Value Testing).
* Тестирование с использованием таблиц принятия решений (Decision Table Testing).
* Попарное тестирование (Pairwise Testing).
* Тестирование состояний и переходов (State-Transition Testing).
* Доменное анализ (Domain Analysis Testing).
* Тестирование на основании Use Case или вариантов использования (Use Case Testing).

Тестирование методом **«черного ящика»** — у тестировщика нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.

И существуют техники тестирования методом белого ящика:

* Тестирование потока управления (Control Flow Testing).
* Тестирование потока данных (Data Flow Testing).



Существуют различные версии разбиения на техники тест-дизайна, исходя из техник тестирования.

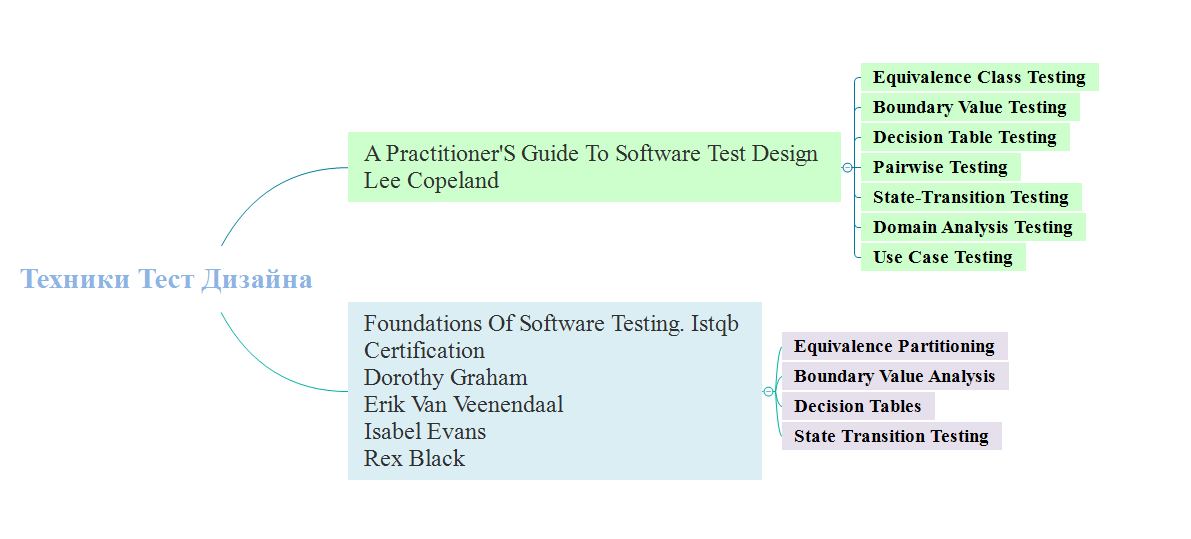
Например, по одной из версий набор техник тест-дизайна такой:

* Эквивалентное разделение;
* Граничные значения;
* Предугадывание ошибок;
* Причина-следствие;
* Таблица принятия решений;
* Попарное тестирование.

По другой версии, это:

* Эквивалентное разделение;
* Анализ граничных значений;
* Тестирование переходов между состояниями;
* Тестирование ситуации гонок;
* Прогнозирование ошибок.

В книгах «Практическое руководство по тест-дизайну программного обеспечения» и FOUNDATIONS OF SOFTWARE TESTING рассматриваются такие техники, как Equivalence Class Testing, Boundary Value Testing, которые будут рассмотрены в этом уроке. Также будет рассмотрена техника Pairwise Testing. Вышеупомянутые книги рекомендуются к прочтению во время курса.



## Классы эквивалентных значений

Согласно ISTQB, **разделение на эквивалентные классы** (Equivalence partitioning) — техника черного ящика, в которой спроектированные/разработанные тест-кейсы для использования эквивалентных классов (в оригинале раздел) используют только один репрезентативный член из каждого класса.

**Класс эквивалентности** (equivalence class) — набор данных, обрабатываемых одинаковым образом и приводящих к одинаковому результату.

Везде, где вы используете технику эквивалентных классов, вы используете или будете использовать и вторую, тесно связанную с эквивалентными классами, технику анализа граничных значений.

Согласно ISTQB, **анализ граничных значений** — техника черного ящика, в которой разрабатываемые тест-кейсы основываются на граничных значениях.

**Граничное условие** (border condition, boundary condition) — значение, находящееся на границе классов эквивалентности.

Ключевая мысль\идея в классах эквивалентности и граничных значениях такая, что все значения входных данных внутри одного класса будут приводить к одинаковому результату, а при пересечении граничных значений поведение системы будет меняться.

Основная цель разбиения на классы эквивалентности — уменьшение количества тест-кейсов.

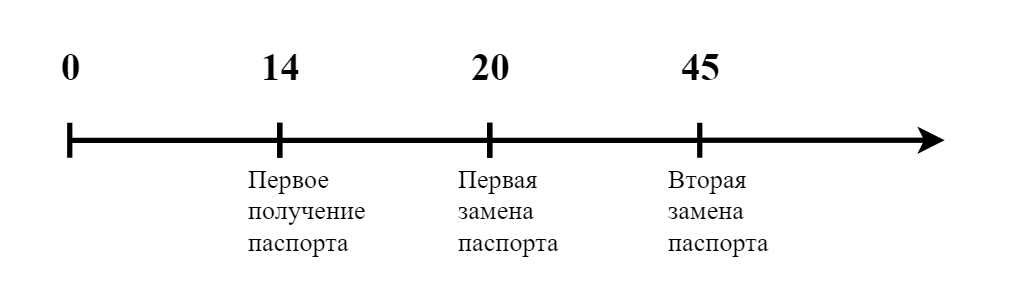
### Пример 1

Рассмотрим пример подачи заявки на получение или замену паспорта по достижении определенного возраста. Мы не рассматриваем случаи потери/кражи и т. д. Рассматриваем с точки зрения именно возраста.

Как мы знаем, в соответствии с законом РФ, первый паспорт можно получить по достижении 14 лет. А замену паспорта можно произвести после 20 и после 45 лет.

На сайте госуслуг можно найти сервис, который помогает оформить соответствующую заявку на получение или обмен паспорта.

Можно визуально выделить классы эквивалентности для возраста заявителей.



Если мы будем выступать в роли тестировщиков сайта госуслуг, у нас будут следующие требования для тестирования услуги получения паспорта (первой выдачи или последующих обменов по сроку):

1. Первый паспорт выдаётся лицу по достижении им 14 лет.
2. Первый обмен паспорта производится по достижении лица возраста 20 лет
3. Второй обмен паспорта производится по достижении возраста 45 лет.

Как тестировщики, мы выделим классы эквивалентности и создадим тестовые профили с разным возрастом для тестирования в дальнейшем.

Первый класс эквивалентности — от рождения до 14 лет, 14 лет не включаем. Гражданин не может получить паспорт, если его возраст попадает в этот диапазон значений.

Второй класс эквивалентности — от 14 лет (включительно) до 20 лет (не включая).

Третий класс эквивалентности — от 20 лет (включительно) до 45 лет (не включая).

Четвертый класс эквивалентности — от 45 лет (включительно) и больше.

Если бы мы пытались протестировать сервис выдачи и обмена паспорта, используя ВСЕ возможные варианты возраста, мы бы начали от 0 лет и заканчивали бы примерно 130 годами. В Википедии указано, что официально никто не прожил больше 130 лет, таких долгожителей пока нет. Итак, порядка 131 тест-кейса, так?

Такое количество тест-кейсов нерационально. Поскольку мы знаем, что система должна вести себя одинаково, если ей будут давать на вход значения из одного класса эквивалентности, то перебирать **все** значения этого класса нет смысла.

Поэтому в идеале мы должны составить всего 4 тест-кейса вместо 131. Уточним: всего 4 тест-кейса, используя технику эквивалентных классов.

В первом тест-кейсе будет возраст от 0 до 14 лет минус один день — ожидаемый результат: отказ в получении паспорта.

Второй тест-кейс будет при достижении 14 лет (включительно) до 20 лет минус один день — ожидаемый результат: согласованная заявка на получение паспорта.

Третий тест-кейс будет при достижении 20 лет (включительно) до 45 лет минус один день — ожидаемый результат: согласованная заявка на обмен паспорта.

Четвертый тест-кейс будет при достижении 45 лет (включительно) и до 130 лет — ожидаемый результат: согласованная заявка на обмен паспорта.

### Пример 2

Рассмотрим еще один пример. При регистрации на [тестовом сайте](http://www.sharelane.com/cgi-bin/register.py) есть поле ввода ZIP code\*, которое должно допускать только 5 цифровых символов.

Если вы вводите 4 цифры для поля ZIP code, то появляется сообщение об ошибке. Тоже самое должно быть. когда вы вводите больше пяти символов. Но так как там дефект, в этом случае такое сообщения не появляется.

Если мы будем пытаться вводить цифры и пытаться разбить на классы эквивалентности цифровые, мы можем выделить три класса.

Первый класс эквивалентности — ZIP code, длина которого меньше 5 цифр: мы получим сообщение об ошибке.

Второй класс эквивалентности — ZIP code, у которого 5 цифр: успешное принятие ZIP code.

Третий класс эквивалентности — ZIP code с длинной больше 5 цифр: у нас должно быть сообщение об ошибке, но мы знаем, что здесь ошибка, и поле принимает ZIP code с длиной больше 5 цифр.

А что будет если мы введем буквы? Или среди цифр будет буква?

Будет сообщение об ошибке – что должны быть цифры, о чем нам и говорит сообщение об ошибке.

Исходя из знания, что допускаются только цифры, мы разделим тестирование этого поля с точки зрения двух функциональностей:

* Проверка на количество символов.
* Проверка на ввод валидных значений.

Какие тест-кейсы мы можем здесь выделить?

Проверка на количество символов:

Позитивный тест-кейс: ввод пяти цифр. Ожидаемый результат — успешное принятие введенного ZIP code.

Негативный: ввод четырёх цифр. Ожидаемый результат — сообщение об ошибке.

Негативный: ввод шести цифр. Ожидаемый результат — сообщение об ошибке.

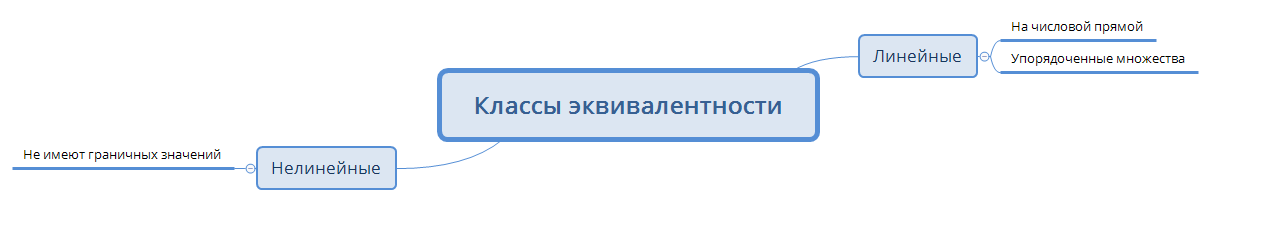
Проверка на ввод валидных значений:

Позитивный тест-кейс: ввод пяти цифр уже описан выше, поэтому дополнительно создавать второй нет необходимости.

Негативный — ввод пяти символов (4 цифры и буква). Ожидаемый результат — сообщение об ошибке.

Суммарно у нас получится 4 тест-кейса для тестирования поля ZIP code. Это минимальный набор проверок.

Чтобы не было путаницы или недопонимания, необходимо уточнить, что классы эквивалентности можно разделить на линейные и нелинейные.



Линейные классы эквивалентности выделяются на числовой прямой или их можно представить в виде упорядоченного множества. Пример — цифровые значения, упорядоченные по какому-то определенному смыслу. Например, возраст человека, размер скидки и т. д.

Нелинейные — множество элементов, которые не имеют граничных значений. Пример — множество специальных символов.

Что можно разбить на классы эквивалентности:

* Числа.
* Символы.
* Длину строки.
* Размер файла.
* Объем памяти.
* Разрешение экрана.
* Версии операционных систем.
* Объем передаваемых данных.

Надо помнить, что цель этой техники — не только сокращение числа тестов, но и сохранение приемлемого тестового покрытия.

**Тестовое покрытие** — одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода.

Краткое резюме по классам эквивалентности: выбирая и выделяя классы эквивалентности, мы получаем следующее:

* Если один тест-кейс эквивалентного класса помогает обнаружить дефект, то все остальные тест-кейсы из этого же эквивалентного класса похожим образом помогут его найти.
* Если один тест-кейс эквивалентного класса не находит дефект, то все остальные тест-кейсы из этого же эквивалентного класса также его не найдут.
* Тестирование с использованием классов эквивалентности может быть использовано на всех уровнях тестирования — юнит-тестировании, интеграционном тестировании, системном тестировании и приёмочном тестировании.

## Техника анализа граничных значений

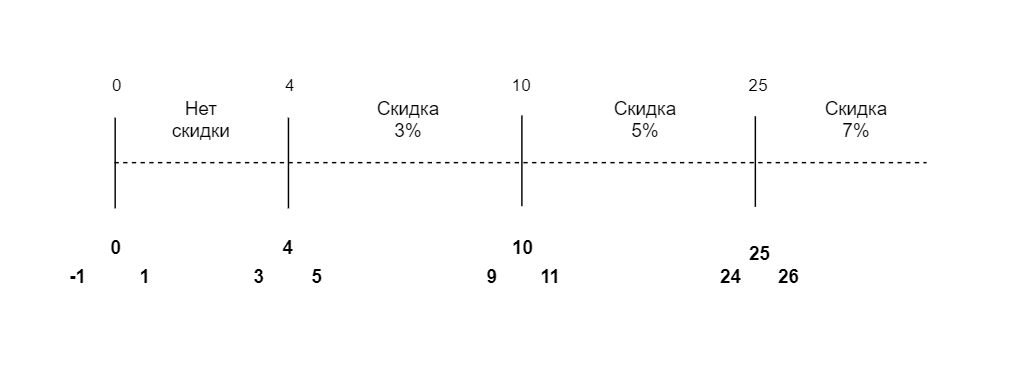
Как мы уже говорили раньше, **анализ граничных значений** — техника черного ящика, в которой разрабатываемые тест-кейсы основываются на граничных значениях.

Ключевой момент в технике анализа граничных значений, когда тестировщик составляет тест-кейсы, это то, что для каждой границы, должно браться три значения – одно значение на самой границе, одно значение непосредственно перед границей и одно значение сразу после границы.

Рассмотрим на следующем примере:

* При покупке более 4 билетов в кино даётся скидка на билеты 3%.
* При покупке более 10 билетов даётся скидка 5%.
* При покупке более 25 билетов даётся скидка 15%.

Снова для более лёгкой визуализации изобразим это на числовой прямой.



Мы должны будем выделить 12 тест-кейсов: по три для каждой из границ — 0, 4, 10 и 25.

Краткое резюме по анализу граничных значений — при их определении мы получаем следующее:

1. Выделение граничных значений — продолжение техники разбиения на классы эквивалентности.
2. Эта техника используется для уменьшения количества тест-кейсов.
3. Эта техника сосредоточена только на анализе граничных значений.
4. При тестировании граничных значений используется значение самой границы, непосредственно до границы и сразу после неё.
5. Техника граничных значений также может быть использована на всех уровнях тестирования — юнит-тестировании, интеграционном тестировании, системном тестировании и приемочном тестировании.

## Попарное тестирование

Хотя техника «Попарное тестирование» не выделяется в книге Dorothy Graham, Dorothy Graham, Isabel Evans, Rex Black — Foundations of Software Testing. ISTQB, она там упоминается и отсылает к книге Lee Copeland — A Practitioner's Guide to Software Test Design (2004), поэтому мы рассмотрим эту технику в этом уроке.

Более детально вы разберёте эту технику на втором уровне курса «Тестирование ПО».

**Попарное тестирование** — техника дизайна тест-кейсов методом черного ящика, в которой тест-кейсы создаются для выполнения всех возможных комбинаций каждой пары входных параметров.

Рассмотрим, сколько всего возможных вариантов, если для разрабатываемого клиент-серверного приложения будут следующие условия:

* 3 поддерживаемые операционные системы (Windows, Linux, MacOs) для клиента
* 8 браузеров (Chrome, Firefox, Opera, IE, Edge, Yandex Browser, Safari, Konqueror)
* 6 поддерживаемых серверных ОС (CentOS, Windows Server 2008, Windows Server 2010, Windows Server 2012R2, Windows Server 2016, Windows Server 2019)

В итоге мы получим 144 комбинаций, которые нам необходимо было бы проверить. К примеру, что ОС клиента с браузером Chrome будет стабильно работать с серверным приложением на ОС Windows Server 2019. И таких комбинаций еще 143.

Какие неудачные ситуации могут быть в данном случае:

* Не тестировать совсем. Поскольку такое огромное количество невозможно протестировать — нет такого количества серверных ОС.
* Протестировать все комбинации хотя бы раз. Но здесь может быть риск, что поставка продукта будет задержана.
* Выбрать одну или две комбинации, используя интуицию (заранее имея запасное место работы).
* Выбрать тесты, комбинации которых вы уже запускали. То есть существуют настроенные среды для повторного тестирования.
* Выбрать тесты, которые проще всего запустить (наиболее распространенные ОС).
* Сделать выборку комбинаций и выбрать первые несколько.
* Сделать выборку комбинаций и выбрать наугад несколько.

В реальности используют технику попарного тестирования, когда стараются протестировать все возможные комбинации каждой пары значений.

Рассмотрим другой пример. Пример покупки билетов:

У нас есть в приложении есть три параметра – День, Время и количество билетов

|  |  |  |
| --- | --- | --- |
| **День** | **Время вылета** | **Количество билетов** |
| Завтра | 12:00 | 1 |
| Послезавтра | 15:00 | 2 |
|  | 18:00 | 3 |
|  |  | 4 |

Если тестировать все варианты, получим 2\*3\*4 = 24 теста.

Используя технику попарного тестирования, мы можем сократить количество тестов до 12.

То есть для дня мы тестируем комбинацию со временем. День комбинируется с количеством билетов. Время вылета комбинируется с количеством билетов.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **День** | **Время вылета** | **Количество билетов** |
| 1 | Завтра | 12:00 | 1 |
| 2 | Послезавтра | 15:00 | 1 |
| 3 | Послезавтра | 12:00 | 2 |
| 4 | Завтра | 15:00 | 2 |
| 5 | Завтра | 18:00 | 3 |
| 6 | Послезавтра | 12:00 | 3 |
| 7 | Послезавтра | 18:00 | 4 |
| 8 | Завтра | 12:00 | 4 |
| 9 | Завтра | 18:00 | 1 |
| 10 | Послезавтра | 18:00 | 2 |
| 11 | Завтра | 15:00 | 3 |
| 12 | Послезавтра | 15:00 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **День** | **Время вылета** | **Количество билетов** |
| 1 | Завтра | 12:00 | 1 |
| 2 | Послезавтра | 15:00 | 1 |
| 3 | Послезавтра | 12:00 | 2 |
| 4 | Завтра | 15:00 | 2 |
| 5 | Завтра | 18:00 | 3 |
| 6 | Послезавтра | 12:00 | 3 |
| 7 | Послезавтра | 18:00 | 4 |
| 8 | Завтра | 12:00 | 4 |
| 9 | Завтра | 18:00 | 1 |
| 10 | Послезавтра | 18:00 | 2 |
| 11 | Завтра | 15:00 | 3 |
| 12 | Послезавтра | 15:00 | 4 |

Проверив таблицу, мы можем убедиться, что для каждого значения параметра «день» есть соответствующая проверка для значения параметра времени вылета. Точно так же для каждого значения параметра «день» есть проверка значения параметра количества билетов.

# Классификация видов и направлений тестирования

Тестирование можно классифицировать по множеству признаков, во многих книгах практически у каждого автора или блогера, вы можете найти свою интерпретацию разбиения видов тестирования по тем или иным признакам. Возможно, что это связано с тем, что у каждого свой опыт и усвоенная в начале теория ложилась на практику, которая была получена на разных реальных проектах. Поэтому каждый автор смотрит через призму своего опыта. Мы с вами рассмотрим сейчас самую простую классификацию по видам тестирования.



Материал взят из книги Святослава Куликова, которая была рекомендована к прочтению перед началом курса.

## По запуску кода на исполнение

По запуску кода на исполнение тестирование делится на:

* Статическое.
* Динамическое.

Статическое тестирование выполняется без запуска ПО. Тестирование ведётся путём анализа программного кода (code review) или скомпилированного кода. Анализ может производиться как вручную, так и с помощью специальных инструментов. Цель анализа — раннее выявление ошибок и потенциальных проблем в продукте.

Статическое тестирование кода проводят разработчики. В рамках такого тестирования выявляются ошибки, связанные с эффективностью и правильностью программных конструкций.

Статическое тестирование проводится и при проверке требований и дизайна по макетам.

Примеры ошибок, которые можно выявить с помощью автоматического статического тестирования:

* Утечка ресурсов (утечка памяти, не освобождаемые файловые дескрипторы и т.д.).
* Возможность переполнения буфера (buffer overflows).
* Ситуации частичной (неполной) обработки ошибок.

Примеры статического тестирования:

* Тестирование требований, тест-кейсы, описание архитектуры приложения, схемы базы данных и т. д. В целом документация по проекту.
* Параметры настройки среды исполнения приложения.
* Статическое тестирование кода (Code review).

В отличие от статического, динамическое тестирование производится путём запуска продукта и проверки его функционала. Она осуществляется с помощью ручного или автоматического выполнения заранее подготовленного набора тестов. Примеры динамического тестирования:

* Модульное тестирование (unit testing).
* Интеграционное тестирование (integration testing).
* Приемочное тестирование (acceptance testing).

## По доступу к коду и архитектуре приложения

По доступу к коду, как правило, разделяют на два основных вида — тестирование методом белого и методом черного ящика. Но многие авторы выделяют и метод серого ящика.

* Метод белого ящика, white box testing, open box testing, clear box testing, glass box testing.
* Метод черного ящика, black box testing, closed box testing, specification-based testing.
* Метод серого ящика, gray box testing.

Тестирование методом **«черного ящика»** — 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.

В рамках тестирования по методу **«черного ящика»** основная информация для создания тест-кейсов — документация, особенно требования (requirements-based testing) и общий здравый смысл для случаев, когда поведение приложения в некоторой ситуации не регламентировано явно. Часто на проектах этот метод может занимать значительное время у тестировщиков.

Метод **«белого ящика»** (white box testing, open box testing, clear box testing, glass box testing) — у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного. Юнит-тестирование, которое проводят, как правило, разработчики продукта, — пример white box testing.

Метод «**серого ящика**» (gray box testing) — часто можно встретить формулировку, что это комбинация методов белого и черного ящиков, состоящая в том, что к части кода и архитектуры у тестировщика доступ есть, а к части — нет. Но стоит отметить, что в случае, если у вас есть доступ к коду, то всё-таки это метод белого ящика.

Можно интерпретировать определение серого ящика как метод тестирования, при котором нет доступа к коду приложения, но во время тестирования можно получить представление об архитектуре приложения, или есть доступ к базе данных. Этот метод отсутствует в официальной документации ISTQB, но бывает, что этот вопрос задают на собеседованиях.

## По степени автоматизации

* Ручное тестирование.
* Автоматизированное тестирование.

**Ручное тестирование** (manual testing) — тестирование, в котором тест-кейсы выполняются человеком вручную без использования средств автоматизации.

**Автоматизированное тестирование** (automated testing, test automation) — 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 делает человек

**Преимущества:**

* 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.).
* 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.

**Недостатки:**

* Необходим высококвалифицированный персонал, так как автоматизация — «проект внутри проекта» (со своими требованиями, планами, кодом и т. д.).
* Высокие затраты на сложные средства автоматизации, разработку и сопровождение кода тест-кейсов.
* 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 заново.

## По уровню детализации приложения

Классификация будет рассматриваться в соответствии с ISTQB по уровню детализации. У неё четыре уровня:

* Модульное тестирование.
* Интеграционное тестирование.
* Системное тестирование.
* Приёмочное тестирование.

**Модульное (компонентное) тестирование** (unit testing, module testing, component testing) направлено на проверку отдельных небольших частей приложения, которые,как правило, можно исследовать изолированно от других подобных частей. При выполнении тестирования могут проверяться отдельные функции или методы классов, они сами, их взаимодействие, небольшие библиотеки, отдельные части приложения.

**Интеграционное тестирование** (integration testing, component integration testing, pairwise integration testing, system integration testing, incremental testing, interface testing, thread testing) направлено на проверку взаимодействия между несколькими частями приложения, каждая из которых, в свою очередь, проверена отдельно на стадии модульного тестирования).

**Системное тестирование** (system testing) 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.

**Приемочное тестирование** (Acceptance testing, user acceptance testing) — формальное тестирование в отношении потребностей пользователей, требований и бизнес-процессов, проводимое для определения того, удовлетворяет ли система критериям приемлемости и позволяет ли пользователю, клиентам или другому уполномоченному органу определять, принимать или не принимать систему.

## По степени важности тестируемых функций

Перед началом рассмотрения классификации необходимо уточнить, что её нет в ISTQB, но есть в некоторых компаниях. Например, в компании EPAM systems.

Итак, по степени важности тестируемых функций можно разделить на:

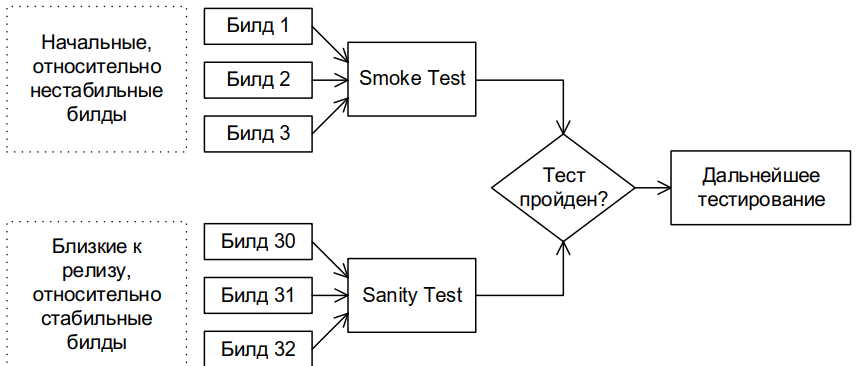
* Дымовое (smoke) тестирование.
* Тестирование критического пути.
* Расширенное тестирование.

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

К примеру, есть ли смысл пытаться тестировать веб-приложение ВКонтакте, если не работает страница авторизации? Ответ очевиден — нет.

Дымовое тестирование производится на новых билдах (сборках приложения), чтобы определить, насколько целесообразно проводить дальнейшее тестирование полученного билда. Так как дымовое тестирование производится каждый раз, как правило, тест-кейсы для этого уровня автоматизируют, и они могут запускаться автоматически при получении новой сборки. Как правило, уровень прохождения (успешность) для тест-кейсов этого уровня должна составлять 100% от всех тест-кейсов этого уровня.

Часто можно услышать мнение, что smoke test — то же самое, что и sanity test. В глоссарии ISTQB нет четкого разделения на sanity test и smoke test. Некоторые авторы утверждают, что разница в том, что дымовое тестирование проводится на начальных (относительно нестабильных) билдах. А санитарное тестирование проводится уже на относительно стабильных билдах.



Также встречается и такая интерпретация разницы между дымовым и санитарным тестированием, что дымовое направлено на тестирование определенного количества функций, то есть по горизонтали протестировать, а санитарное — на более детальное/углубленное тестирование функциональности (тестирование вглубь).

**Тестирование критического пути** (critical path test) направлено на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. Если по каким-то причинам приложение не выполняет эти функции или выполняет их некорректно, очень многие пользователи не смогут достичь 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 70–80–90 % — в зависимости от сути проекта).

**Расширенное тестирование** (extended test) 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 важной. Но при достаточном количестве времени и иных ресурсов тест-кейсы этого уровня могут затронуть даже самые низкоприоритетные требования.

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.

Также надо понимать, что набор тестов для дымового тестирования, тестирования критического пути или расширенного пути могут дополняться или изменяться. По мере того, как функционал приложения дополняется, набор тестов тоже должен дополняться.

## По принципам работы с приложением

* Позитивное тестирование.
* Негативное тестирование.

**Позитивное тестирование** (positive testing) направлено на исследование приложения в ситуации, когда все действия выполняются строго по инструкции без каких бы то ни было ошибок, отклонений, ввода неверных данных и т. д. Если позитивные тест-кейсы завершаются ошибками, это тревожный признак — приложение работает неверно даже в идеальных условиях (и можно предположить, что в неидеальных условиях оно работает ещё хуже).

**Негативное тестирование** (negative testing, invalid testing) 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 ноль). 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).

## По целям и задачам тестирования

Здесь необходимо отдельно рассмотреть те виды тестирования, которые не вошли в предыдущие.

* Функциональные виды тестирования.
* Нефункциональные виды тестирования.
* Регрессионное тестирование.
* Повторное тестирование.

**Функциональное тестирование** (functional testing) — вид тестирования, направленный на проверку корректности работы функциональности приложения (корректность реализации функциональных требований). Часто функциональное тестирование ассоциируют с тестированием по методу чёрного ящика, однако и по методу белого ящика вполне можно проверять корректность реализации функциональности.

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

**Регрессионное тестирование** (regression testing) — тестирование, направленное на проверку того, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования. Фредерик Брукс в книге «Мифический человеко-месяц» писал: «Фундаментальная проблема при сопровождении программ состоит в том, что исправление одной ошибки с большой вероятностью (20–50%) влечёт появление новой». Поэтому регрессионное тестирование — неотъемлемый инструмент обеспечения качества и активно используется практически в любом проекте. Также тест-кейсы данного вида тестирования — кандидаты для автоматизации.

**Повторное тестирование** (re-testing, confirmation testing) — выполнение тест-кейсов, которые ранее обнаружили дефекты, чтобы подтвердить их устранение. Фактически этот вид тестирования сводится к действиям на финальной стадии жизненного цикла отчёта о дефекте, направленным на то, чтобы перевести дефект в состояние «проверен» и «закрыт».

Отметим, что разница между функциональным и нефункциональным тестированием заключается в том, что функциональное тестирование проверяет, **что** делает приложение (или делает ли оно то, что должно), а нефункциональное тестирование проверяет, **как** приложение это делает.

# Логи и работа с ними

Работа с логами необходима, чтобы лучше понять, что происходит с приложением, когда мы явно видим ошибку, которая отображается на пользовательском интерфейсе, или когда мы только подозреваем, что что-то идёт не так, и нам необходимо больше информации о поведении приложения или системы в целом.

Как правило, работают с логами непосредственно самого приложения и с серверными, то есть логами серверов, на которых установлено (развернуто) разрабатываемое приложение.

**Логирование** — запись данных о работе программы. Место, куда эти данные записываются, называется «лог».

Наиболее распространённые расширения для файлов с логами — формат LOG.

Расширение LOG — это файл журнала, который, как правило, хранится в формате обычного текстового файла и используются многими программами, в том числе операционной системой, для хранения информации о системных процессах и их работе. Журнал обычно содержит записи, когда, в какое время и какие процессы программ были запрошены. Как правило, данная информация полезна для пользователей и программистов.

Рассмотри один из примеров лог-файла веб-сайта.

Лог-файл веб-сайта (log file, log-файл, лог-файл, лог) — текстовый файл, в котором регистрируются все запросы к сайту, а также все ошибки, связанные с этими запросами.

Как происходит запись событий в лог-файл сайта:

* интернет-пользователь набирает в браузере адрес какого-либо сайта (веб-страницы) и нажимает Enter;
* браузер пользователя передаёт на сервер (на котором находится сайт) запрос на выдачу веб-страницы (в лог-файле это оформляется записью "GET <адрес\_ресурса>");
* на сервер передаётся следующая информация:
  + IP-адрес посетителя;
  + дата и время запроса;
  + используемый пользователем браузер;
  + используемая пользователем операционная система;
  + запрашиваемый URL;
  + адрес страницы, с которой зашёл посетитель;
  + …;
* сервер выдает посетителю запрашиваемый документ/страницу/картинку/файл (если он существует);
* сервер записывает информацию о произошедшей транзакции в журнал событий (лог-файл).

## Пример 1

80.133.67.3 - - [21/Jul/2009:01:01:34 +0400] "GET http://<адрес\_ресурса>/news.xml" 304 0 "http://<адрес\_ресурса>" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; InfoPath.2; .NET CLR 3.5.30729; OfficeLiveConnector.1.4; OfficeLivePatch.1.3; .NET CLR 3.0.30729)" 127.0.0.1

Расшифровка:

21 июля 2009 г. в 01:01:34 (по местному времени хостинг-провайдера) с ip-адреса 80.133.67.3 к RSS-ленте news.xml сайта http://<адрес\_ресурса> обращался встроенный RSS-ридер браузера Internet Explorer 8 (при этом, операционная система — Windows Vista).

Код 304 означает, что записей о новых веб-страницах сайта http://<адрес\_ресурса> в RSS-ленте не появилось, поэтому браузер не загружал файл news.xml (о чём красноречиво свидетельствует 0).

## Пример 2

Рассмотрим пример логов с веб-сервера WAMP, который можно установить на локальную машину.

Это логи запросов GET и POST:

**127.0.0.1 - - [01/Oct/2019:20:17:20** +0200] "POST /testlink/lib/ajax/gettprojectnodes.php?root\_node=98&tcprefix=2019- HTTP/1.1" 200 4

**127.0.0.1 - - [01/Oct/2019:20:17:20 +0200]** "GET /testlink/third\_party/ext-js/images/default/tree/folder.gif HTTP/1.1" 200 952

**127.0.0.1 - - [01/Oct/2019:20:17:27 +0200]** "GET /testlink/lib/general/mainPage.php HTTP/1.1" 200 11689

**127.0.0.1 - - [03/Oct/2019:21:40:06 +0200]** "GET /testlink/login.php?note=logout&viewer= HTTP/1.1" 200 2165

**127.0.0.1 - - [03/Oct/2019:21:40:38 +0200]** "POST /testlink/login.php?viewer= HTTP/1.1" 200 157

## Пример 3

Логи работы базы данных:

**2019-06-07T20:11:44.700475Z** 0 [Warning] InnoDB: Resizing redo log from 2\*3072 to 2\*16384 pages, LSN=2529307

**2019-06-07T20:11:44.899041Z** 0 [Warning] InnoDB: Starting to delete and rewrite log files.

100 200

100 200

**2019-06-07T20:11:56.837638Z** 0 [Warning] InnoDB: New log files created, LSN=2529307

**2019-06-16T08:03:37.229892Z** 0 [ERROR] Can't start server: Bind on TCP/IP port: No such file or directory

**2019-06-16T08:03:37.231699Z** 0 [ERROR] Do you already have another mysqld server running on port: 3306 ?

Обычно, когда в логах пишутся ошибки, то они записываются со Stack Trace.

В Stack Trace записывается или указывается, в каком месте и что ломается. По последовательности вызова методов разработчики видят, что за чем происходило, и какие функции или методы приводили в итоге к ошибке.

StudentException: Error finding students

at StudentManager.findStudents(StudentManager.java:13)

at StudentProgram.main(StudentProgram.java:9)

Caused by: DAOException: Error querying students from database

at StudentDAO.list(StudentDAO.java:11)

at StudentManager.findStudents(StudentManager.java:11)

... 1 more

Caused by: java.sql.SQLException: Syntax Error

at DatabaseUtils.executeQuery(DatabaseUtils.java:5)

at StudentDAO.list(StudentDAO.java:8)

... 2 more

Короткое объяснение выше приведенного Stack Trace:

1. Ошибка при поиске студента.
2. Вызвана ошибкой запроса к базе данных.
3. Ошибка запроса произошла из-за ошибки в синтаксисе самого запроса к базе данных.

Какими программами можно открывать для просмотра логов:

* NotePAD++ с AnalysePlugin.
* LogExpert.
* Baretail.

Они бесплатные для скачивания, и могут вам пригодиться на первое время, даже дома.

# Глоссарий

**Тест-дизайн (Test Design)** — деятельность по получению или выделению тест-кейсов, или из определённых тестовых условий.

**Разделение на эквивалентные классы** (Equivalence partitioning) — техника черного ящика, в которой спроектированные/разработанные тест-кейсы для использования эквивалентных классов (в оригинале раздел) используют только один репрезентативный член из каждого класса.

**Класс эквивалентности** (equivalence class) — набор данных, обрабатываемых одинаковым образом и приводящих к одинаковому результату.

**Анализ граничных значений** — техника черного ящика, в которой разрабатываемые тест-кейсы основываются на граничных значениях.

**Граничное условие** (border condition, boundary condition) — значение, находящееся на границе классов эквивалентности.

**Тестовое покрытие** — одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода.

**Попарное тестирование** — техника дизайна тест-кейсов методом черного ящика, в которой тест-кейсы создаются для выполнения всех возможных комбинаций каждой пары входных параметров.

# Практическое задание

**Используя приложенную таблицу, выполнить следующие задания:**

1. Прочитать из книги Святослава Куликова страницы 234-238.
2. Ответить на вопросы в таблице.
3. Выделить классы эквивалентности, исходя из приведенных требований.
4. Ответить на вопросы в приведённой форме.
5. \* Решить Challenge #3 с сайта http://testingchallenges.thetestingmap.org/.
6. \* Установить на локальной машине Testlink.

## Требования к выполненной работе

1. Домашнее задание должно быть выполнено в приложенной форме.
2. Расширение файла должно быть формата Excel (XLS, XLSX).
3. Название файла не менять, и вместо ФИО укажите свои фамилию и имя.

# Дополнительные материалы

1. Lee Copeland. A Practitioner's Guide to Software Test Design (2004).
2. [PAIRWISE TESTING](https://training.qatestlab.com/blog/technical-articles/pairwise-testing/).
3. Dorothy Graham, Dorothy Graham, Isabel Evans, Rex Black. Foundations of Software Testing. ISTQB.

# Используемые источники

Для подготовки методического пособия мы использовали эти ресурсы:

1. Software Testing. Base Course (Svyatoslav Kulikov).
2. [ISTQB Glossary](https://glossary.istqb.org/search).
3. Lee Copeland. A Practitioner's Guide to Software Test Design (2004).
4. Dorothy Graham, Dorothy Graham, Isabel Evans, Rex Black. Foundations of Software Testing. ISTQB.
5. [Тестирование областей определения или нечто большее, чем анализ граничных значений](https://habr.com/ru/company/infopulse/blog/270909/).
6. [Что такое лог-файлы сайта и зачем их нужно анализировать?](http://netler.ru/pc/log-file.htm)