Тестирование программного обеспечения — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.

Test-Driven Development (TDD) — разработка через тестирование; подход к разработке и тестированию, при котором сначала создаются тесты, которым должен удовлетворять код, затем его реализация. TDD имеет итеративный процесс. Сперва пишется тест на новый, ещё не реализованный функционал, а затем пишется минимальное количество кода (ничего лишнего) для его реализации. При успешном прохождении теста, можно задуматься о качестве кода и сделать его рефакторинг.

Behavior-Driven Development (BDD) — разработка, основанная на поведении; расширение подхода TDD, где особое внимание уделяется поведению системы в терминах бизнеса. Такие тесты обычно иллюстрируют и тестируют сценарии, интересные заказчику системы. Поэтому для BDD-тестов используются фреймворки (Chai, Mocha, Jest) с синтаксисом, понятным не только программисту, но и представителю заказчика. Обычно этот синтаксис похож на английский язык. BDD-тесты могут быть написаны и поняты не только программистом, но и техническими менеджерами или тестировщиками, что позволяет убрать языковой барьер между всеми ними.

Data-Driven Testing (DDT) — тестирование, управляемое данными; подход к архитектуре автоматизированных тестов, при котором тестовые данные хранятся отдельно от тестов (в файле или базе данных).

Алгоритм *DDT:*

- Часть тестовых данных извлекается из хранилища.
- Выполняется скрипт, в котором вызывается обычный тест с извлечёнными тестовыми данными.
- Сравниваются полученные (actual) результаты с ожидаемыми (expected).
- Алгоритм повторяется со следующим набором входных данных.

Keyword-Driven Testing (KDT) — тестирование, управляемое ключевыми словами; подход, использующий ключевые слова, описывающие набор действий, необходимых для выполнения определённого шага тестового сценария. Для использования подхода нужно определить набор ключевых слов и сопоставить им действия (функции).

Алгоритм *KDT:*

- Считываем ключевые слова вместе с их параметрами из таблицы.
- Последовательно вызываем связанные с ключевыми словами функции.

Принципы тестирования.

Принцип №1 - тестирование показывает наличие дефектов. Тестирование может показать, что в продукте существуют дефекты, но не сигнализирует о том, что дефектов не существует вообще. Т.е. в процессе тестирования мы снижаем вероятность того, что в продукте остались дефекты. Но, даже если мы их вообще не обнаружили, то мы не можем говорить о том, что их нет. Из моей практики, баги в продукте есть всегда.

Принцип №2 - исчерпывающее тестирование невозможно. Протестировать абсолютно всё в продукте невозможно. И даже если вам кажется обратное, то вы потратите огромное количество времени на это, а как известно - время деньги. А как тогда быть уверенным в том, что продукт работает корректно? Всегда есть риски и приоритеты. Для того, чтобы минимизировать риски, есть специальные техники тест-дизайна, которые помогают тестерам проектировать свои тесты так, чтобы с минимальными усилиями покрыть как можно больше тестовых случаев и функциональностей.

Принцип №3 - это раннее тестирование. Т.е. тестовые активности должны начинаться как можно раньше и всегда преследовать определенные цели. В данном случае экономия средств заказчика.

Принцип №4 - скопление дефектов. Он гласит так: в небольшом количестве модулей сокрыто большое количество дефектов. И если мы вспомним с вами правило Парето, то оно применимо и к данному принципу: 80% дефектов находится в 20% функций. Т.е. мы должны понимать, что наибольшее количество дефектов сконцентрировано не во всём приложение, а в какой-то определенной из его функциональностей, поэтому тестеры должны распределять свои усилия пропорционально. Например, если он понимает, что в модуле логина пользователя в систему всегда происходят какие-то физические дефекты, то он должен больше время сконцентрировать на нахождение и устранение этих дефектов именно в этой функциональности.

Принцип №5 - парадокс пестицида. Прогоняя одни и те же тесты вновь и вновь, вы столкнетесь с тем, что они находят всё меньше новых ошибок, поскольку системы эволюционируют. Многие из ранее найденных дефектов исправляют и старые тесты больше не срабатывают. Чтобы преодолеть этот парадокс, необходимо периодически вносить изменения в используемые наборы тестов, корректировать их так, чтобы они отвечали новому состоянию системы и позволяли находить как можно большее количество дефектов. Для этого также нужно постоянно изучать новые методы тестирования и внедрять их свою работу. Для преодоления данного препятствия можно давать прогонять тесты другим участникам команды, либо же производить ротацию кадров, чтобы разные тестировщики в разное время тестировали одну и ту же функциональность. Кстати, вопрос по минимизации данного риска часто задают на собеседовании, так что обязательно запомните ответ на него. Почему же этот принцип называется парадокс пестицида? Всё очень просто. Проведите аналогию с применением какого-то химиката против насекомых, либо же каких-то сорняков. Если их постоянно травить одним и тем же инсектицидом или пестицидом, то у них возникает привыкание, они адаптируются и меньшее количество живности вымирает, либо уничтожается под действием того или иного токсиканта. Также работает принцип пестицида для тестирования.

Принцип №6 - тестирование зависит от контекста. Выбор методологии, техники или типа тестирования будет напрямую зависеть от природы самой программы. Например, программное обеспечение для медицины требует более тщательной проверки, чем компьютерная игра. Или же сайт с большей посещаемостью должен пройти через серьёзное тестирование производительности, чтобы показать возможность работы в условиях высокой нагрузки. Поэтому тестировщик всегда должен ответственно подходить к выбору той среды, в которой он будет тестировать, к выбору той документации, которую будет тестировать. Например, если продукт сложный, то лучше выбрать тест-кейсы, а не чек-листы.

Принцип №7 - это заблуждение об отсутствии ошибок. Каждому тестировщику не стоит полагать, что если тестирование не обнаружило дефект, то программа готова к релизу. Нахождение и исправление дефектов будут не важны, если система окажется неудобна в использовании и не будет удовлетворять нуждам пользователя.

Testing - это самый нижний и первый уровень, который представляет собой проверку создаваемого программного продукта на соответствие требованиям к этому продукту. По факту - это рутинная работа. Т.е. тебе выдали какой-то перечень функциональности, ты их проверил, нашёл какие-то дефекты, описал их, предоставил разработчику, разработчик их пофиксил, предоставил тебе исправленные баги на проверку. В принципе, на этом работа заканчивается. Именно эта рутинная работа помогает выявлять дефекты на уже созданном программном обеспечении, но не более того. Это не значит, что тестирование - это очень просто. По сути, тестирование - это база и минимум, без которого выпускать продукт нельзя. Основная задача тестирования - это выявить и зафиксировать дефекты. Этой работой на проекте занимаются начинающие тестировщики, т.е. junior.

Quality Control (контроль качества). Он включает в себя тестирование, но не ограничивается только им. Quality Control обеспечивает не только проверку продукта на соответствие требованиям, но и соответствие заранее согласованному уровню качества продукта и готовность к выпуску продукта в Production. Основная задача контроля качества - это предоставить объективную картину того, что происходит с качеством продукта на разных этапах разработки. По сути, этой работой занимается middle, т.е. специалист уже среднего звена с некоторым опытом работы.

Quality Assurance (обеспечение качества). Это третий уровень, куда доходят далеко не все специалисты. Обычно это лиды, либо менеджеры и этот уровень включает в себя мероприятия на всех этапах разработки и, по-хорошему, даже использование продукта. Для обеспечения согласованного уровня качества продукта - это уже проактивная работа. Что это значит? Основная задача обеспечения качества - это выстроить систему, которая будет превентивно работать на качество продукта, т.е. предупреждать какие-то дефекты, либо же наладить процесс так, чтобы эти дефекты были найдены как можно раньше, еще до того, как продукт попадёт к разработчику. Огромное поле для фантазии. Наша основная задача - это минимизировать количество найденных дефектов в зависимости от специфики проекта. Сюда может включаться тестирование документации, ревью кода на соответствие стандартам внедрения каких-то методик по работе с качеством, коммуникационные активности и методики, и так далее.

С первого взгляда перечисленное выше может показаться сложным и непонятным. А давайте разберём всё это на примере: конструирование машины. С помощью тестирования и контроля качества мы можем определить: работают ли все детали и сама машина в целом так, как мы ожидаем, из правильных ли материалов она сделана, с применением нужных методик и инструментов или нет. Т.е. подразумевается, что объект уже существует и готов к проверке задачи. Quality Assurance же является обеспечение соответствия всех этапов конструирования нашей машины, определенный стандарт качества, начиная с планирования и создания чертежей, и заканчивая сборкой уже готового автомобиля. Т.е. качеству объекта уделяется внимание еще до того, как сам объект был создан.

Таким образом, мы можем построить модель иерархии процессов обеспечения качества: Тестирование – часть QC. QC – часть QA.

Верификация (Verification) — это статическая практика проверки документов, дизайна, архитектуры, кода, т.д.

- Верификация — это процесс включающий в себя проверку Plans, Requirement Specifications, Design Specifications, Code, Test Cases, Chek-Lists, etc.
- Верификация всегда проходит без запуска кода.
- Верификация использует методы — reviews, walkthroughs, inspections, etc.
- Верификация отвечает на вопрос “Делаем ли мы продукт правильно?”
- Верификация поможет определить, является ли программное обеспечение высокого качества, но оно не гарантирует, что система полезна. Проверка связана с тем, что система хорошо спроектирована и безошибочна.
- Верификация происходит до Validation.

Валидация (Validation) – это процесс оценки конечного продукта, необходимо проверить, соответствует ли программное обеспечение ожиданиям и требованиям клиента. Это динамический механизм проверки и тестирования фактического продукта.

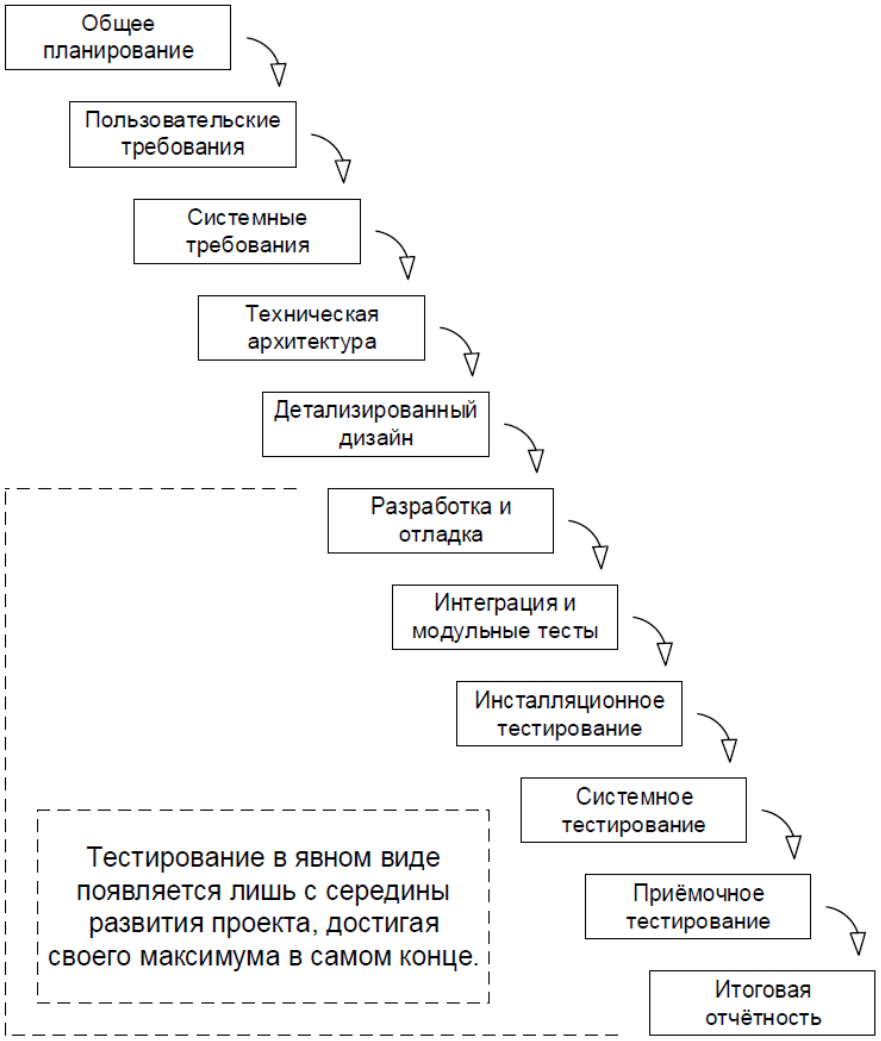
- Валидация всегда включает в себя запуск кода программы.
- Валидация использует методы, такие как тестирование Black Box, тестирование White Box и нефункциональное тестирование.
- Валидация отвечает на вопрос “Делаем ли мы правильный продукт?”
- Валидация проверяет, соответствует ли программное обеспечение требованиям и ожиданиям клиента.
- Валидация может найти ошибки, которые процесс Verification не может поймать.
- Валидация происходит после Verification.

На практике, отличия верификации и валидации имеют большое значение: заказчика интересует в большей степени валидация (удовлетворение собственных требований); исполнителя, в свою очередь, волнует не только соблюдение всех норм качества (верификация) при реализации продукта, а и соответствие всех особенностей продукта желаниям заказчика.

Что же касается примеров, возвращаясь к нашей машине, которую мы описывали ранее, здесь верификация покажет нам, выполнен ли автомобиль из соответствующих спецификации материалов, подогнаны ли детали по высоте, установлен ли заявленный двигатель, что был прописан в нашей спеке. Валидация покажет, сможет ли автомобилем пользоваться невысокий человек, вообще поедет ли автомобиль, удобно ли выполнены сиденья, поместится ли в багажнике огромный чемодан, т.е. насколько наш продукт будет отвечать нуждам нашего пользователя. Ещё один пример: программное обеспечение. У вас есть форма для логина в систему, в которой есть поля. Согласно спеке размер полей строго установлен. Вы проверяете так ли это, примерно размер поля 16 пикселей и, если это так, то всё ок. Это процесс верификации. Но вот если вы оставите поле с логином пустым и нажмёте кнопку войти в систему, то покажут ошибку, т.е. работает валидация, так как это поле должно быть обязательно заполнено. Простой вывод: при верификации проверяется НАЛИЧИЕ чего-нибудь, при валидации - РАБОТОСПОСОБНОСТЬ чего-нибудь.

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.

Водопадная модель (waterfall model) 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 2.1.а). Очень упрощённо можно сказать, что в рамках этой модели в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.



V-образная модель (V-model) является логическим развитием водопадной. Можно заметить (рисунок 2.1.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.

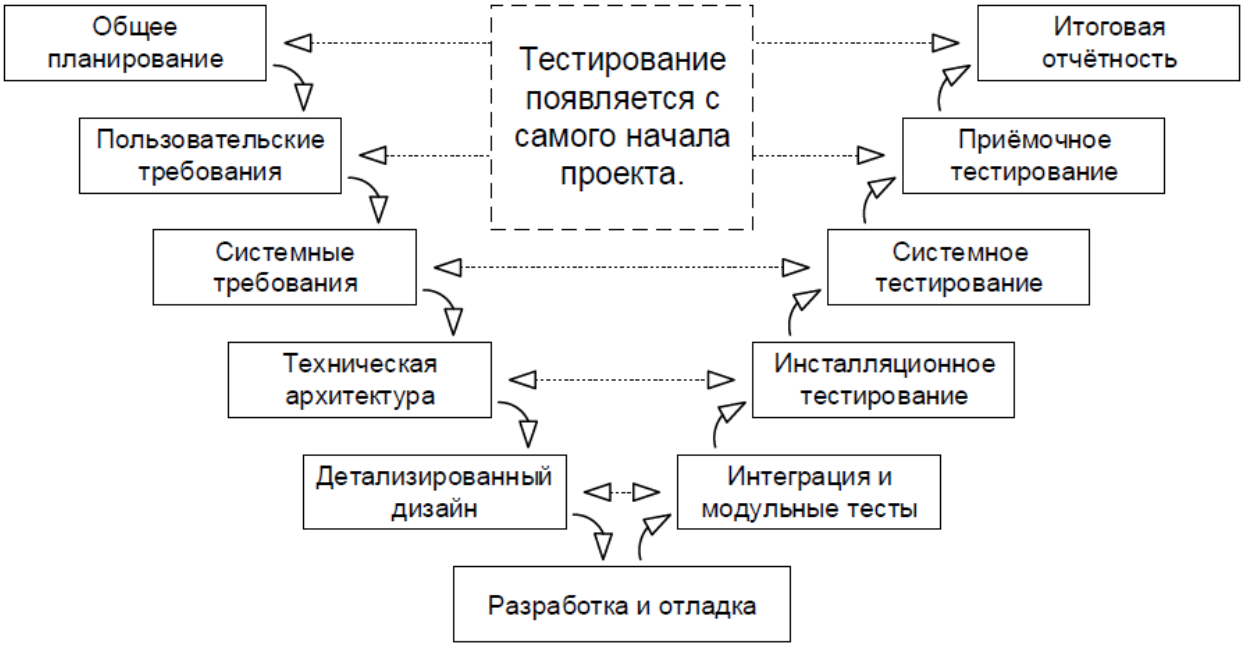


Рисунок 2.1.b — V-образная модель разработки ПО

Итерационная инкрементальная модель (iterative model, incremental model) 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 разбиение проекта на относительно небольшие промежутки (итерации), каждый из которых в общем случае может включать в себя все классические стадии, присущие водопадной и v-образной моделям (рисунок 2.1.c). Итогом итерации является приращение (инкремент) функциональности продукта, выраженное в промежуточном билде (build).

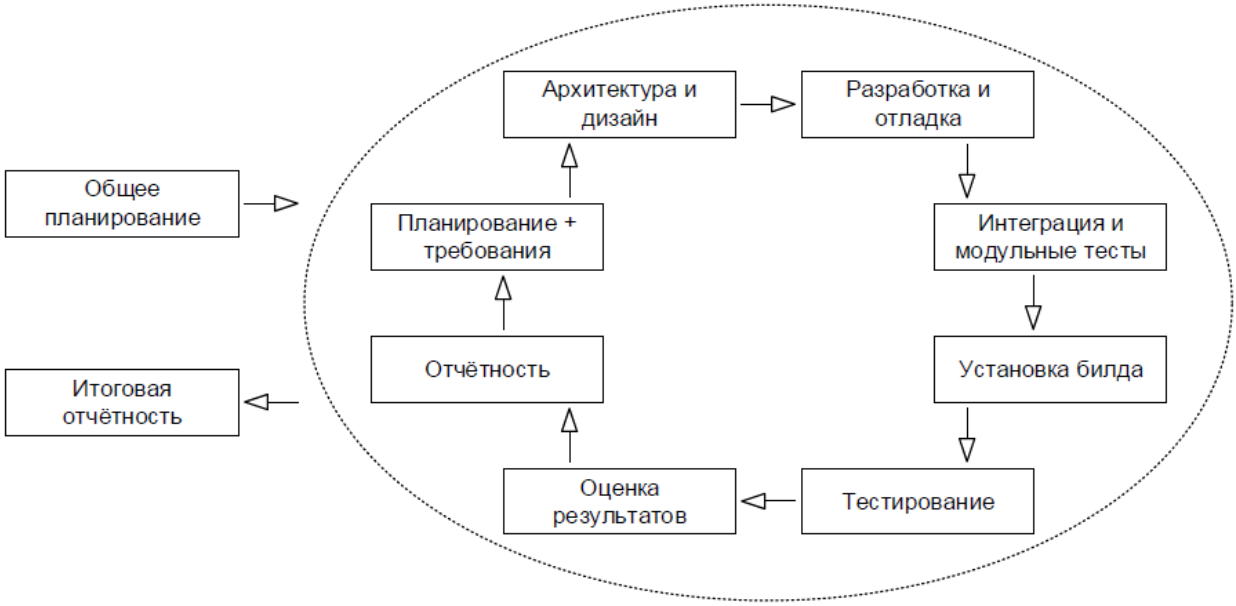
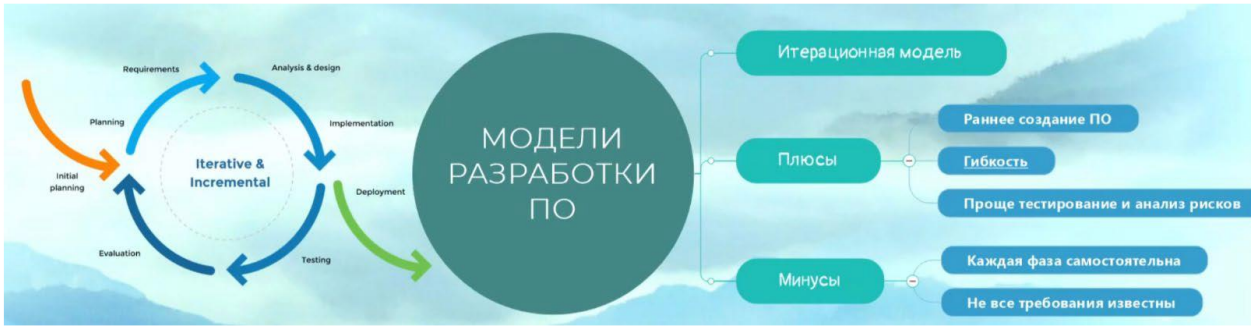


Рисунок 2.1.c — Итерационная инкрементальная модель разработки ПО



Спиральная модель (spiral model) 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 2.1.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.

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).

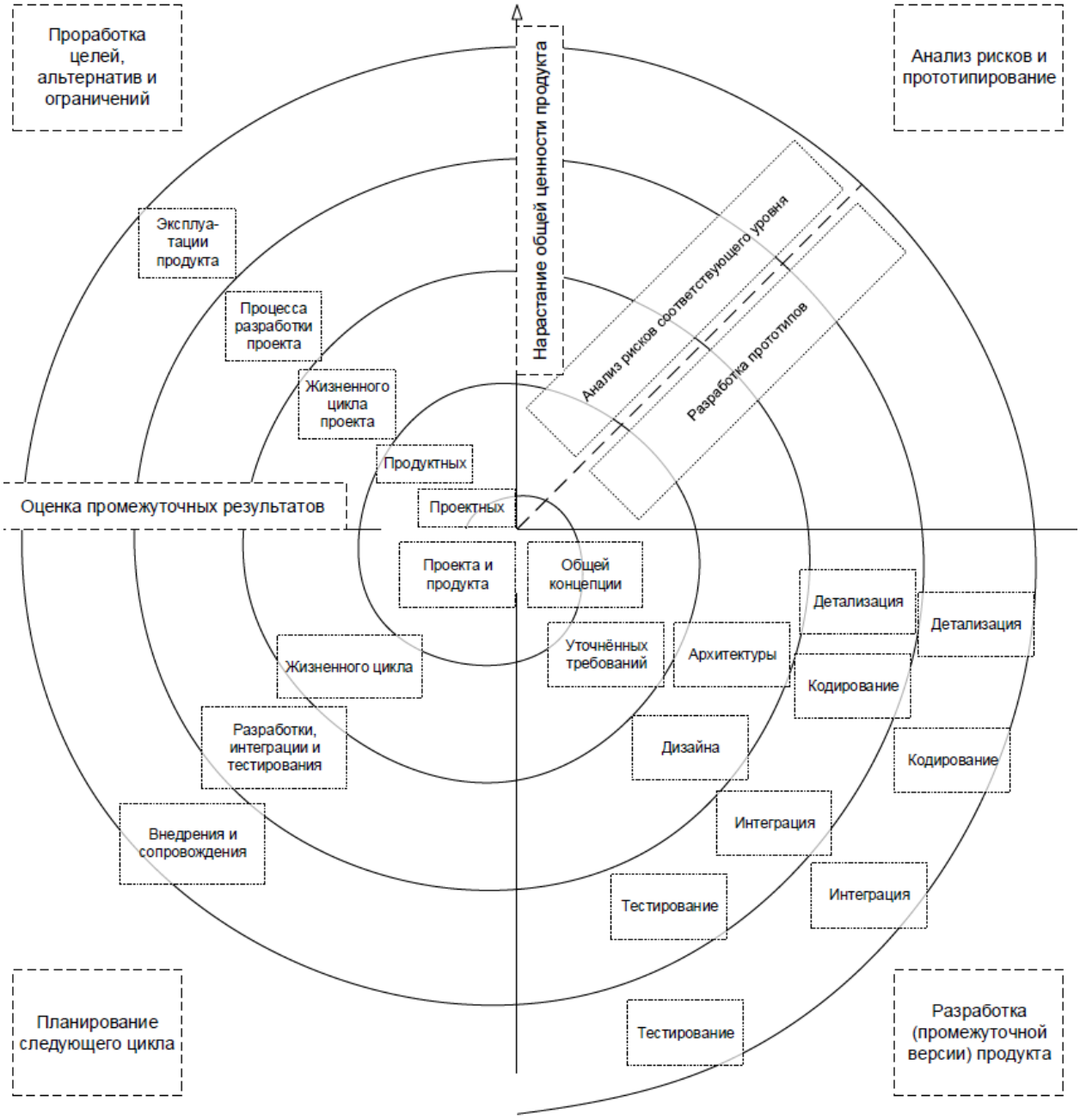


Рисунок 2.1.d — Спиральная модель разработки ПО

Гибкая модель (agile model) представляет собой совокупность различных подходов к разработке ПО и базируется на т.н. «agile-манифесте»:

• 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.

Основополагающие принципы Agile-манифеста:

• Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.

• Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.

• Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.

• На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

• Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.

• Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.

• Работающий продукт — основной показатель прогресса.

• Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.

• Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.

• Простота — искусство минимизации лишней работы — крайне необходима.

• Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.

• Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

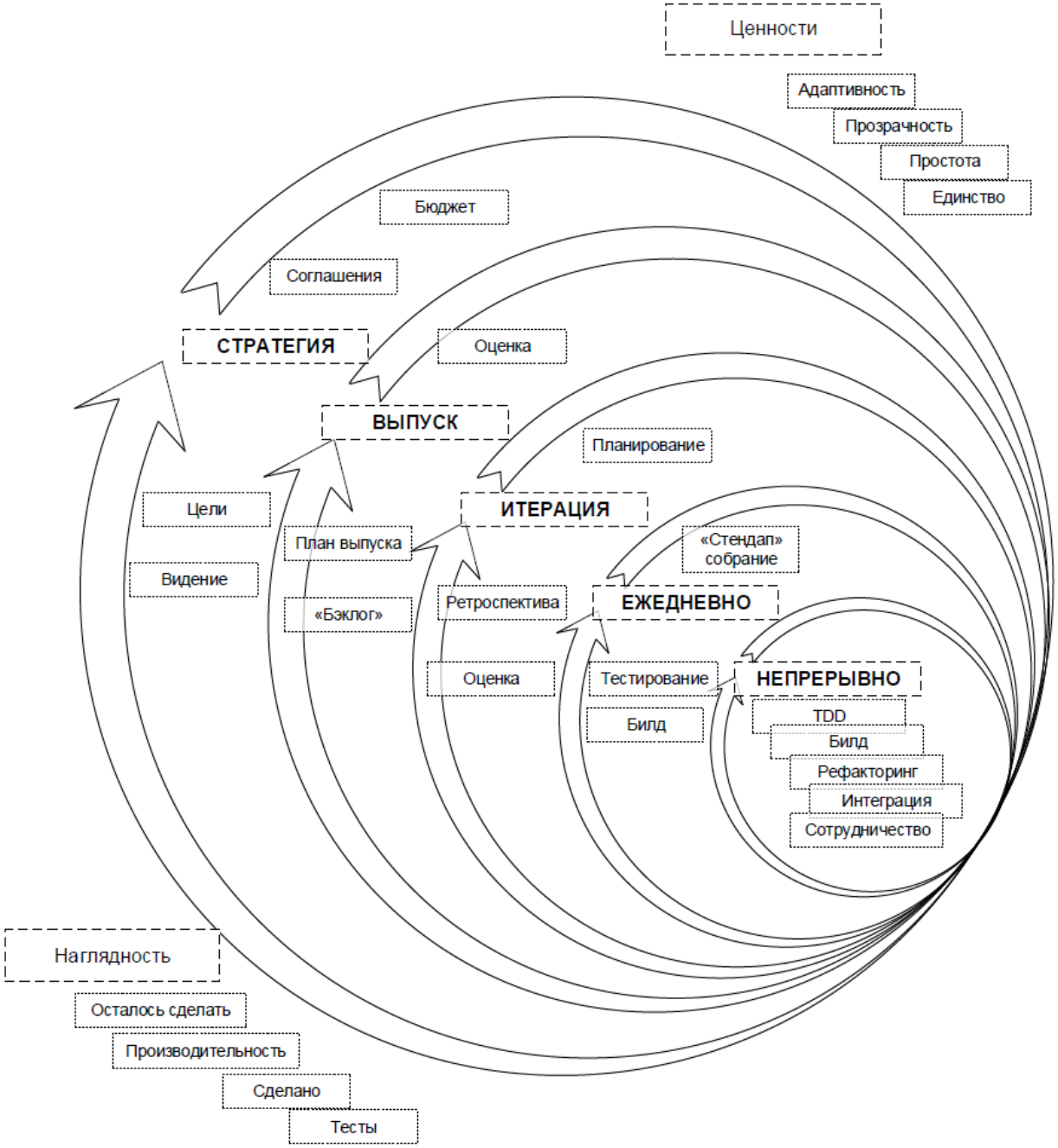


Рисунок 2.1.е — Суть гибкой модели разработки ПО

Таблица 2.1.а — Сравнение 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. • Много 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**.

Scrum — легкий фреймворк, который помогает людям, командам и организациям создавать ценность с помощью адаптивных решений комплексных проблем. **Scrum** — минимально необходимый набор мероприятий, артефактов, ролей, на которых строится процесс Scrum-разработки, позволяющий за фиксированные небольшие промежутки времени, называемые спринтами (sprints), предоставлять конечному пользователю работающий продукт с новыми бизнес-возможностями, для которых определён наибольший приоритет. В конце спринта Scrum-команда встречается на обзорном совещании результатов спринта (Sprint Review) с заказчиком, и представляет ему инкремент бизнес-продукта (версия продукта с законченным набором функциональности, который уже можно отдавать заказчику и пользователю для использования), который она успела сделать за спринт. Цель Sprint Review — получение обратной связи от заказчика, чтобы понять, на чём нужно делать акцент в дальнейшем, и какой должен быть следующий инкремент бизнес-продукта. Строго фиксированная небольшая длительность спринта (от 1 до 4 недель) снижает риски, и даёт возможность быстро получить обратную связь от заказчика, чтобы скорректировать видение продукта.

Прозрачность: Появляющиеся процесс и работа должны быть видны и тем, кто выполняет работу, и тем, кто получает результаты. Важные решения в Scrum основаны на оценке состояния трех формальных артефактов. Артефакты с низкой прозрачностью могут привести к решениям, снижающим ценность и повышающим риск.

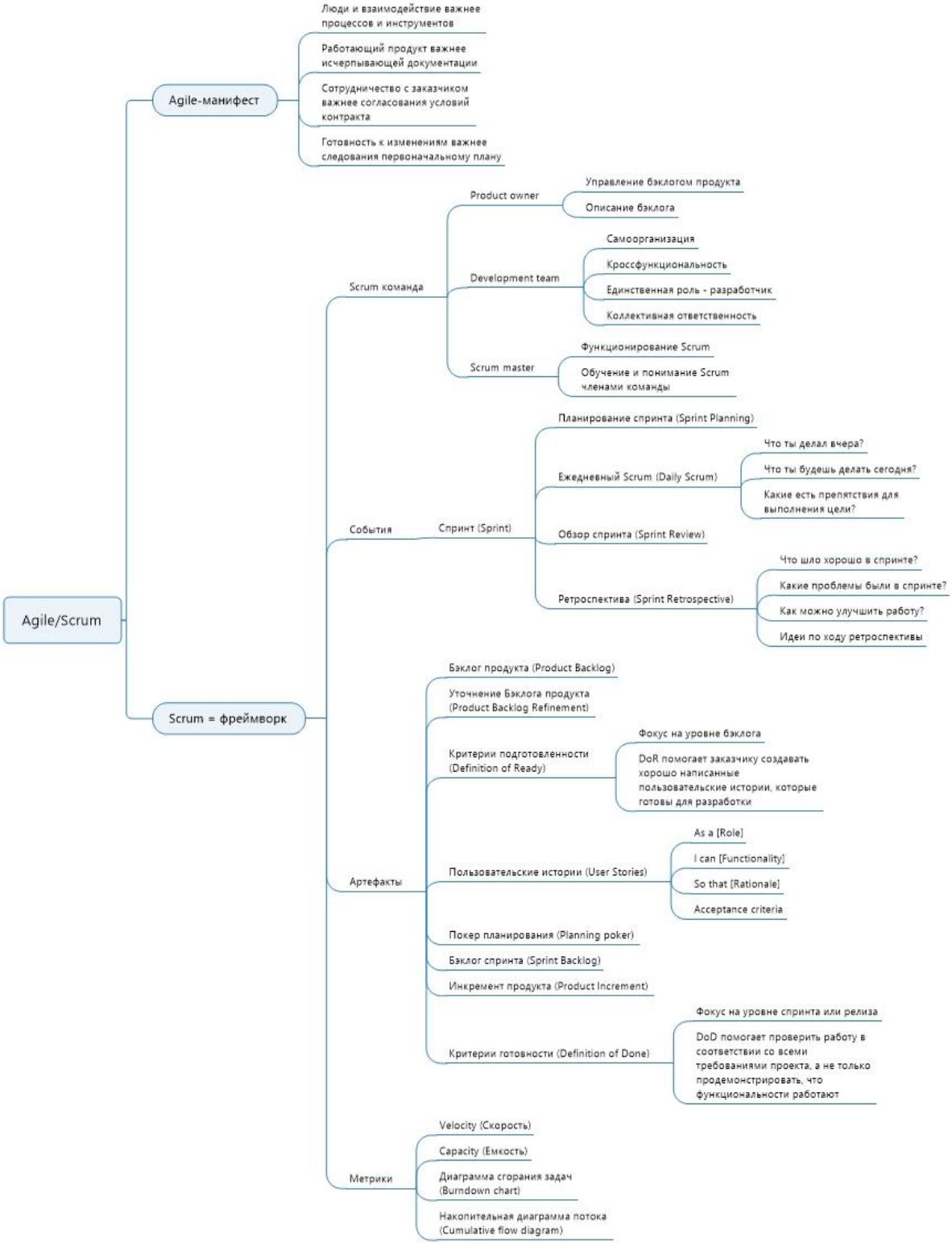
Прозрачность делает возможной инспекцию. Инспекция без прозрачности вводит в заблуждение и является потерями.

Инспекция: Для выявления потенциально нежелательных отклонений и проблем необходимо регулярно и тщательно инспектировать артефакты Scrum и прогресс в достижении согласованных целей.

Инспекция делает возможной адаптацию. Инспекция без адаптации считается бессмысленной. События Scrum спроектированы так, чтобы провоцировать изменения.

Адаптация: Если какие-либо аспекты процесса выходят за допустимые пределы, или если конечный продукт является неприемлемым, применяемый процесс или производимые материалы должны быть скорректированы. Корректировку необходимо произвести как можно скорее, чтобы минимизировать дальнейшее отклонение. Адаптация становится более сложной, когда участвующие в ней люди не обладают полномочиями или не самоуправляемы. Ожидается, что Scrum Team адаптируется в тот момент, когда узнает что-то новое при инспекции.

Sprint Goal — единственная цель на Sprint. Несмотря на то, что Developers привержены Sprint Goal, она обеспечивает гибкость с точки зрения выбора конкретной работы, необходимой для ее достижения. Sprint Goal также обеспечивает связность и сфокусированность, побуждая Scrum Team работать совместно, а не над отдельными инициативами. Sprint Goal создается во время Sprint Planning, а затем добавляется в Sprint Backlog. Developers помнят о Sprint Goal в ходе работы над задачами Sprint. Если работа не соответствует ожиданиям, они взаимодействуют с Product Owner, чтобы пересмотреть содержание Sprint Backlog в рамках Sprint, не изменяя Sprint Goal.


Agile/Scrum
Agile-манифест
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
Scrum = фреймворк
Scrum команда
Product owner
Управление бэклогом продукта
Описание бэклога
Development team
Самоорганизация
Кроссфункциональность
Единственная роль - разработчик
Коллективная ответственность
Scrum master
Функционирование Scrum
Обучение и понимание Scrum членами команды
События
Спринт (Sprint)
Планирование спринта (Sprint Planning)
Ежедневный Scrum (Daily Scrum)
Что ты делал вчера?
Что ты будешь делать сегодня?
Какие есть препятствия для выполнения цели?
Обзор спринта (Sprint Review)
Ретроспектива (Sprint Retrospective)
Что шло хорошо в спринте?
Какие проблемы были в спринте?
Как можно улучшить работу?
Идеи по ходу ретроспективы
Артефакты
Бэклог продукта (Product Backlog)
Уточнение Бэклога продукта (Product Backlog Refinement)
Критерии подготовленности (Definition of Ready)
Фокус на уровне бэклога
DoR помогает заказчику создавать хорошо написанные пользовательские истории, которые готовы для разработки
Пользовательские истории (User Stories)
As a [Role]
I can [Functionality]
So that [Rationale]
Acceptance criteria
Покер планирования (Planning poker)
Бэклог спринта (Sprint Backlog)
Инкремент продукта (Product Increment)
Критерии готовности (Definition of Done)
Фокус на уровне спринта или релиза
DoD помогает проверить работу в соответствии со всеми требованиями проекта, а не только продемонстрировать, что функциональности работают
Метрики
Velocity (Скорость)
Capacity (Емкость)
Диаграмма сгорания задач (Burndown chart)
Накопительная диаграмма потока (Cumulative flow diagram)

Kanban (от яп. 看板 «рекламный щит, вывеска») — метод управления разработкой, реализующий принцип «точно в срок» и способствующий равномерному распределению нагрузки между работниками. Kanban предполагает обсуждение производительности в режиме реального времени и полную прозрачность рабочих процессов. Этапы работы визуально представлены на Kanban-доске, что позволяет членам команды видеть состояние каждой задачи в любой момент времени.

Kanban - стратегия оптимизации потока ценности внутри процесса, который использует вытягивающую систему с ограничением незавершенной работы и визуализацией.

Центральное место в определении Kanban занимает концепция потока (flow). Поток - это движение ценности через систему разработки продукта. Kanban оптимизирует поток за счет повышения общей эффективности, результативности и предсказуемости процесса.

Четыре основных метрики потока:

● Незавершенная работа (WIP): количество рабочих элементов, работа над которыми начата, но ещё не завершена. Команда может использовать метрику WIP, чтобы обеспечить прозрачность своего прогресса в сокращении WIP и улучшении своего потока.

● Время цикла: время, прошедшее между началом и завершением работы над рабочим элементом.

● Возраст рабочего элемента: время между моментом, когда рабочий элемент попал в систему, и текущим временем. Это относится только к тем элементам, которые еще не завершены.

● Пропускную способность: количество рабочих элементов, завершенных за единицу времени.

Работа kanban-команд строится вокруг kanban-доски, которая используется для визуализации и оптимизации рабочего процесса. Хотя некоторые команды предпочитают реальные доски, виртуальные доски давно стали обязательной функцией любого инструмента agile-разработки ПО: с ними проще отследить процессы, организовать совместную работу и доступ из разных мест.

Доски нужны, чтобы визуализировать работу команды, стандартизировать процесс, а также найти и устранить блокеры и зависимости. И не важно, в какой форме они представлены — в физической или в цифровой. На стандартной Kanban-доске процесс состоит из трех шагов: «Запланировано», «В работе» и «Сделано». Однако доску можно настроить в соответствии с процессом, принятым в той или иной команде, в зависимости от ее размеров, структуры и целей.

Методология Kanban основана на полной прозрачности работы и обмене информацией по ресурсам в режиме реального времени. Таким образом, Kanban-доска должна стать единственным достоверным источником информации о работе команды.

У команд, использующих Kanban, каждая рабочая задача представлена в виде отдельной карточки на доске. Зачем отображать работу в виде карточки на Kanban-доске? Благодаря такому наглядному представлению членам команды будет проще и удобнее отслеживать жизненный цикл рабочих задач. На Kanban-карточках отображается важная информация о конкретной рабочей задаче, доступная всей команде: имя ответственного за выполнение задачи, краткое описание выполненной работы, оценка необходимого времени и т. д. На виртуальных Kanban-досках в карточки также часто добавляют снимки экрана и другие важные для исполнителя технические детали. Когда все члены команды видят состояние каждой рабочей задачи в любой момент времени, а также всю связанную с ней информацию, повышается концентрация, обеспечивается полная прозрачность, быстрее выявляются блокеры и зависимости.

Kanban-команда концентрируется только на текущей работе. По завершении рабочей задачи команда забирает следующую задачу с верха бэклога. Владелец продукта может менять приоритет задач в бэклоге, не мешая работе команды, поскольку изменения происходят за пределами текущих рабочих задач. Если владелец продукта следит, чтобы наверху бэклога были самые важные рабочие задачи, команда разработчиков будет гарантированно поставлять максимально ценный продукт бизнесу. Таким образом, необходимости в спринтах фиксированной длительности, используемых в методике Scrum, просто нет.

Продолжительность цикла — ключевой показатель для Kanban-команд. Под продолжительностью цикла понимается время прохождения рабочей задачей жизненного цикла, от начала работы над задачей до ее поставки. Оптимизировав продолжительность цикла, в будущем команда сможет с уверенностью предсказывать срок поставки задач.

Если теми или иными навыками обладает несколько человек, продолжительность цикла сокращается, если же только один — в процессе появляется узкое место. Именно поэтому команды стремятся делиться знаниями и внедряют такие практики, как проверка кода и наставничество. Благодаря обмену знаниями члены команды могут выполнять разнообразные задачи, что еще больше оптимизирует продолжительность цикла. Это также означает, что в случае скопления работы вся команда сможет взяться за нее и восстановить нормальное течение процесса. К примеру, тестирование не всегда выполняют только инженеры по тестированию. Разработчики тоже могут поучаствовать. В методологии Kanban все члены команды отвечают за то, чтобы рабочие процессы протекали без сучка, без задоринки.

Многозадачность убивает эффективность. Чем больше незавершенных задач, тем чаще приходится переключаться между ними, а это сказывается на сроках их завершения. Поэтому ключевой принцип Kanban состоит в ограничении объема незавершенной работы (WIP). Лимиты незавершенной работы позволяют быстро находить в работе команды узкие и проблемные места, вызванные нехваткой внимания, людей или навыков.

К примеру, типичная команда разработчиков ПО может использовать четыре состояния процесса разработки: «Запланировано», «В работе», «Проверка кода» и «Сделано». Для состояния проверки кода можно установить лимит WIP, равный 2. Число может показаться маленьким, но на все есть свои причины: разработчики предпочитают писать собственный код, а не проверять чужой. Низкий лимит стимулирует команду уделять особое внимание задачам в состоянии проверки, а также проверять чужую работу, прежде чем создавать свои задачи на проверку кода. В конечном итоге это сокращает общее время цикла.

Сравнение Scrum и Kanban

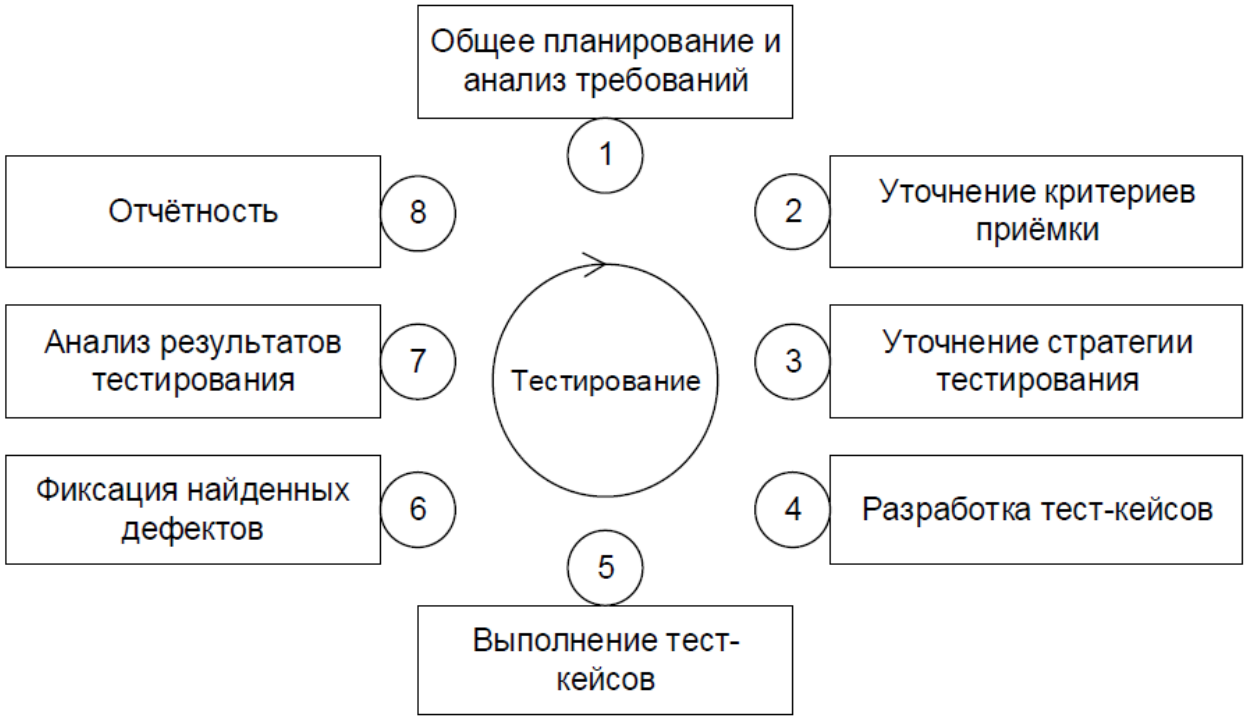
Некоторые концепции Kanban и Scrum похожи, однако в этих методиках используются совершенно разные подходы. Их необходимо четко разграничивать.

	Scrum	Kanban
График	Регулярные спринты фиксированной продолжительности (например, 2 недели)	Непрерывный процесс
Подходы к релизу	В конце каждого спринта после одобрения владельцем продукта	Поставка выполняется непрерывно или на усмотрение команды
Роли	Владелец продукта, Scrum-мастер, команда разработчиков	Ролей нет, в некоторых командах работают тренеры по agile
Ключевые показатели	Скорость команды	Продолжительность цикла
Отношение к изменениям	В ходе спринта команды стремятся избегать изменений в прогнозах спринта: изменения приведут к неверным выводам относительно оценки задач	Изменение может произойти в любой момент

Некоторые команды объединяют идеалы Scrum и Kanban в Scrumban. Из Scrum берут спринты фиксированной длительности и роли, а из Kanban — концентрацию на продолжительности цикла и ограничение количества одновременно выполняемых задач.

Жизненный цикл тестирования

Следуя общей логике итеративности, превалирующей во всех современных моделях разработки ПО, жизненный цикл тестирования также выражается замкнутой последовательностью действий (рисунок 2.1.g).



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), приостановки (suspension criteria) и возобновления (resumption criteria) тестирования, завершения или прекращения тестирования (exit criteria).

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), 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.

Требование (requirement) — описание того, какие функции и с соблюдением каких условий должно выполнять приложение в процессе решения полезной для пользователя задачи.

Продуктная документация (product documentation, development documentation) используется проектной командой во время разработки и поддержки продукта. Она включает:

• План проекта (project management plan) и в том числе тестовый план (test plan).

• Требования к программному продукту (product requirements document, PRD) и функциональные спецификации (functional specifications doc-ument, FSD; software requirements specification, SRS).

• Архитектуру и дизайн (architecture and design).

• Тест-кейсы и наборы тест-кейсов (test cases, test suites).

• Технические спецификации (technical specifications), такие как схемы баз данных, описания алгоритмов, интерфейсов и т.д.

Проектная документация (project documentation) 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 accompanying documentation), такую как встроенная помощь, руководство по установке и использованию, лицензионные соглашения и т.д.

• Маркетинговую документацию (market requirements document, MRD), 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).

Рисунок 2.2.c — Соотношение понятий «продуктная документация» и «проектная документация»

Источники и пути выявления требований

Требования начинают свою жизнь на стороне заказчика. Их сбор (gathering) и выявление (elicitation) осуществляются с помощью следующих основных техник (рисунок 2.2.d).

Рисунок 2.2.d — Основные техники сбора и выявления требований

Уровни и типы требований

Форма представления, степень детализации и перечень полезных свойств требований зависят от уровней и типов требований, которые схематично представлены на рисунке 2.2.е.

Рисунок 2.2.е — Уровни и типы требований

Бизнес-требования (business requirements) выражают цель, ради которой разрабатывается продукт (зачем вообще он нужен, какая от него ожидается польза, как заказчик с его помощью будет получать прибыль). Результатом выявления требований на этом уровне является общее видение (vision and scope) — 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.

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

Пользовательские требования (user requirements) 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), пользовательских историй (user stories), пользовательских сценариев (user scenarios). (Также см. создание пользовательских сценариев в процессе выполнения тестирования.)

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) 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) расширяют собой нефункциональные требования и на уровне пользовательских требований могут быть представлены в виде описания ключевых для проекта показателей качества (свойств продукта, не связанных с функциональностью, но являющихся важными для достижения целей создания продукта — производительность, масштабируемость, восстанавливаемость). Атрибутов качества очень много, но для любого проекта реально важными является лишь некоторое их подмножество.

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) 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. Стоит помнить, что к поведению системы относится не только то, что система должна делать, но и то, что она не должна делать (например: «приложение не должно выгружать из оперативной памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции»).

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) описывают свойства системы (удобство использования, безопасность, надёжность, расширяемость и т.д.), которыми она должна обладать при реализации своего поведения. Здесь приводится более техническое и детальное описание атрибутов качества. 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 часов.

• Ни при каких условиях общий объём используемой приложением памяти не может превышать 2 ГБ.

• Размер шрифта для любой надписи на экране должен поддерживать настройку в диапазоне от 5 до 15 пунктов.

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

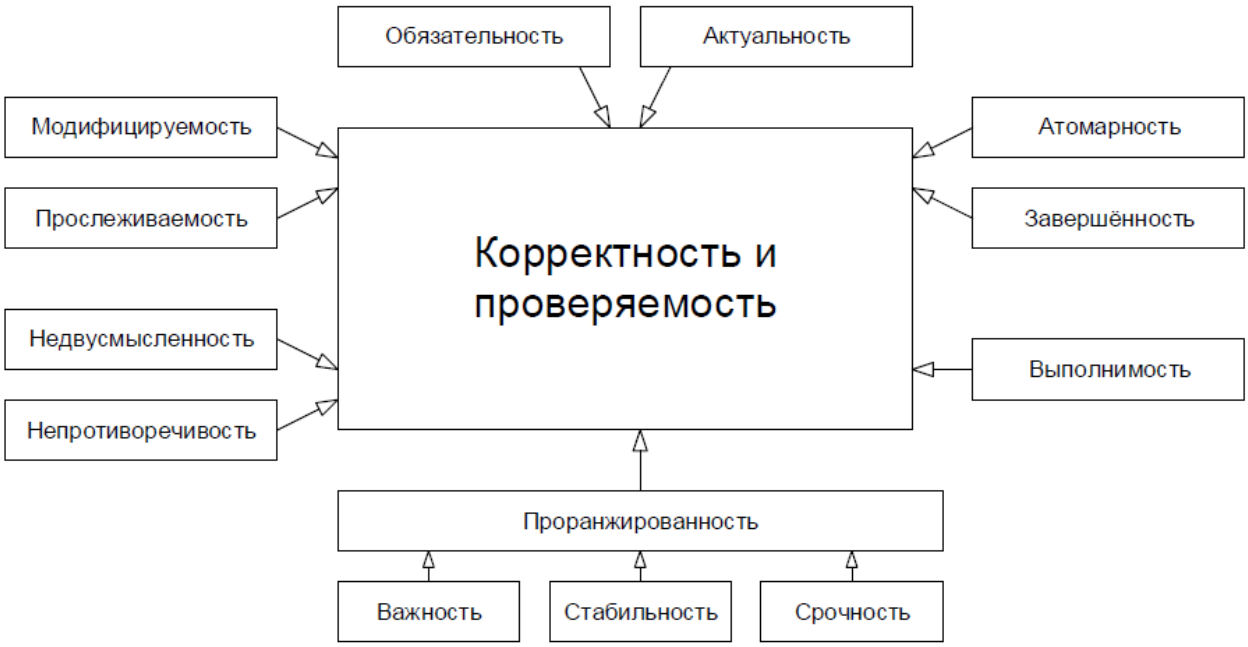
Требования к интерфейсам (external interfaces requirements) описывают особенности взаимодействия разрабатываемой системы с другими системами и операционной средой.

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

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

Свойства качественных требований

В процессе тестирования требований проверяется их соответствие определённому набору свойств (рисунок 2.2.f).



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

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

Непротиворечивость, последовательность (consistency). Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.

Недвусмысленность (unambiguousness, clearness). Требование должно быть описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок, должно допускать только однозначное объективное понимание и быть атомарным в плане невозможности различной трактовки сочетания отдельных фраз.

Выполнимость (feasibility). Требование должно быть технологически выполнимым и реализуемым в рамках бюджета и сроков разработки проекта.

Обязательность, нужность (obligatoriness) 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 исключены (или переработаны) должны быть требования, утратившие актуальность.

Прослеживаемость (traceability). Прослеживаемость бывает вертикальной (vertical traceability) и горизонтальной (horizontal traceability). Вертикальная позволяет соотносить между собой требования на различных уровнях требований, 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) и/или матрицы прослеживаемости (traceability matrix).

Модифицируемость (modifiability). 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.

Проранжированность по важности, стабильности, срочности (ranked for importance, stability, priority). Важность характеризует зависимость успеха проекта от успеха реализации требования. Стабильность характеризует вероятность того, что в обозримом будущем в требование не будет внесено никаких изменений. Срочность определяет распределение во времени усилий проектной команды по реализации того или иного требования.

Корректность (correctness) и *проверяемость* (verifiability). Фактически эти свойства вытекают из соблюдения всех вышеперечисленных (или можно сказать, что они не выполняются, 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.

Техники тестирования требований

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

• *Взаимный просмотр* (peer review). 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) может выражаться как в показе автором своей работы коллегам с целью создания 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, дешёвый и часто 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) выполняется группой специалистов. В идеальной ситуации каждый специалист должен представлять свою область знаний. Тестируемый продукт не может считаться достаточно качественным, пока хотя бы у одного просматривающего остаются замечания. Для запоминания: аналог технического просмотра — это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т.д.

- Формальная инспекция (inspection) представляет собой структурированный, систематизированный и документируемый подход к анализу документации. Для его выполнения привлекается большое количество специалистов, само выполнение занимает достаточно много времени, и потому 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 спросить представителей заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы полученный ответ позволил улучшить требования.

• *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 в голову не приходит — это тревожный знак.

Рекомендуется для начала убедиться, что вы понимаете требование (в том числе прочесть соседние требования, задать вопросы 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 помогает — 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, очень удобно использовать рисунки, схемы, диаграммы, интеллект-карты 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.

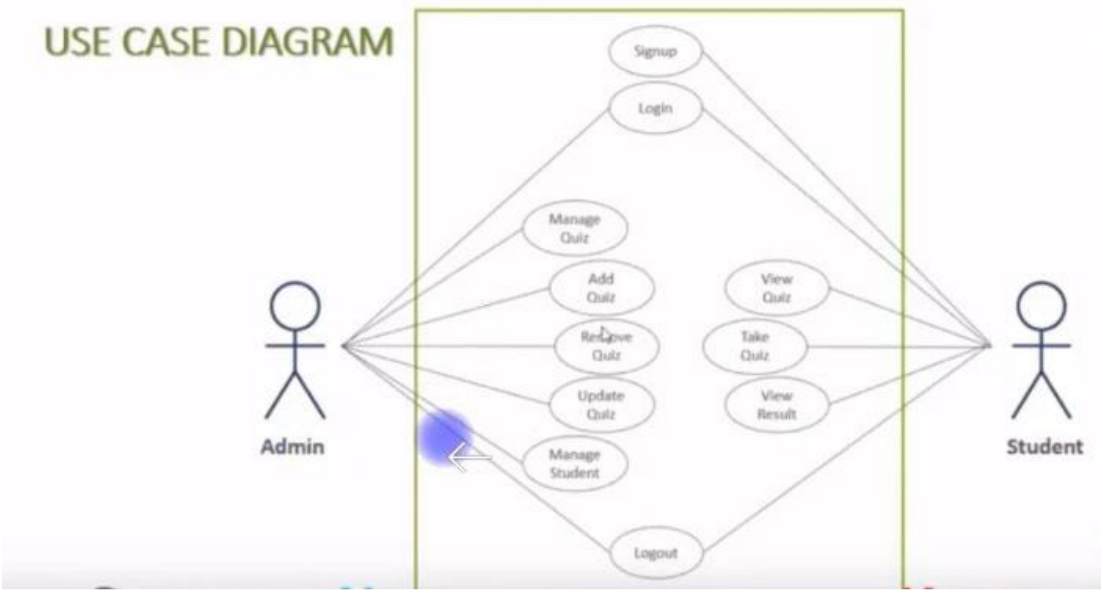
• *Прототипирование.* Можно сказать, что прототипирование часто является следствием создания графического представления и анализа поведения системы. С использованием специальных инструментов можно очень быстро сделать наброски пользовательских интерфейсов, оценить применимость тех или иных решений и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

Типичные ошибки при анализе и тестировании требований:

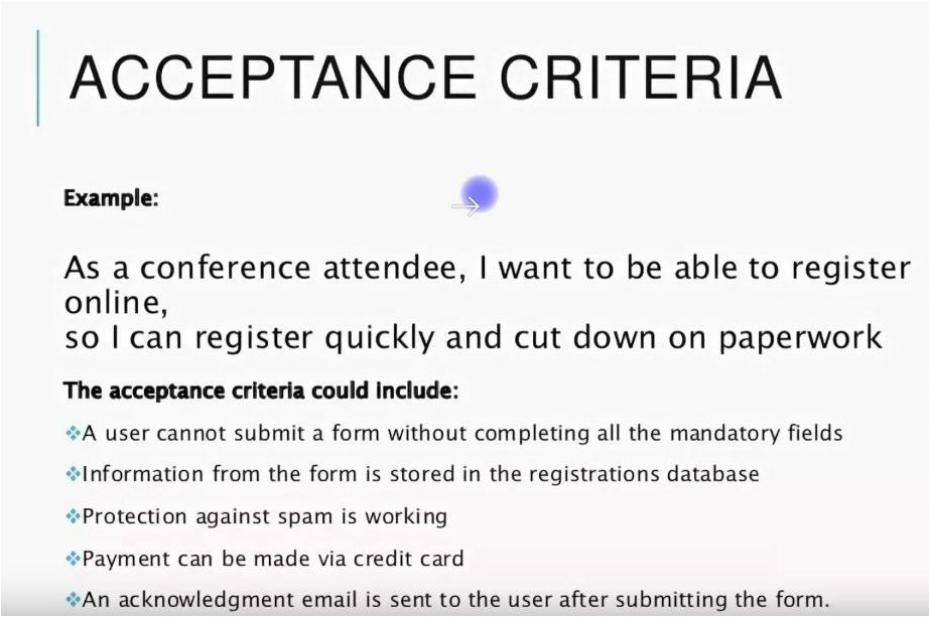
- Изменение формата файла и документа
- Отметка того факта, что с требованием всё в порядке
- Описание одной и той же проблемы в нескольких местах
- Написание вопросов и комментариев без указания места требования, к которым они относятся.
- Задавание плохо сформулированных вопросов.
- Написание очень длинных комментариев и/или вопросов.
- Критика текста или даже его автора
- Категоричные заявления без обоснования
- Указание проблемы с требованиями без пояснения её сути
- Плохое оформление вопросов и комментариев
- Описание проблемы не в том месте, к которому она относится
- Скрытое редактирование требований.
- Анализ, не соответствующий уровню требований

Способы представления требований.

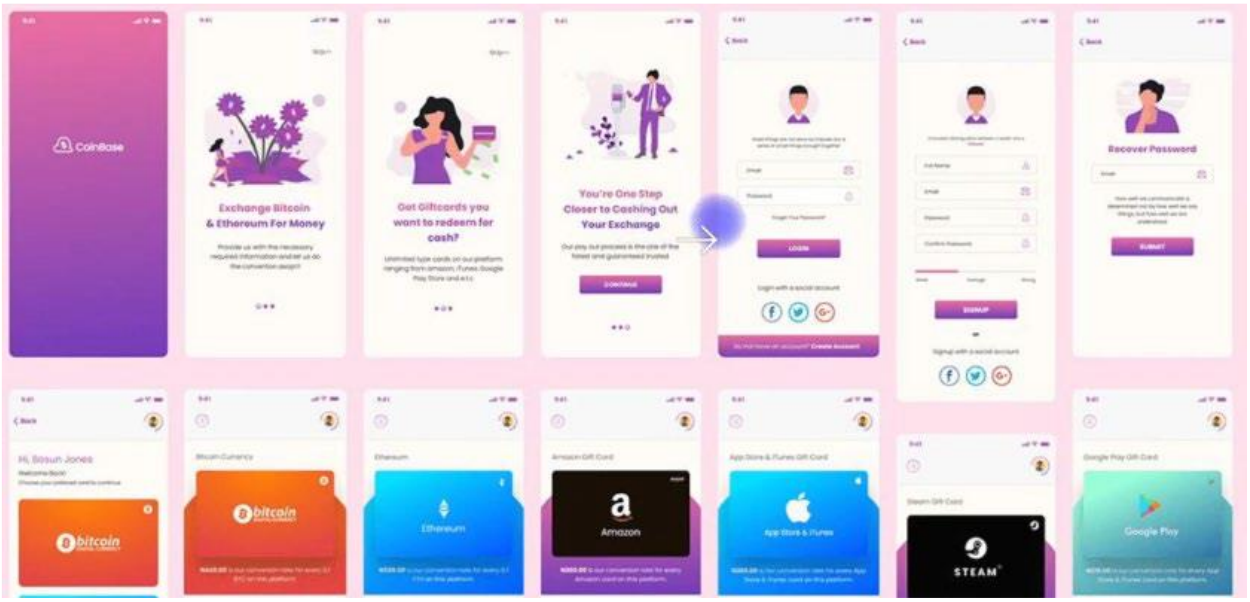
Use-cases. У нас есть так называемые актеры, которые выполняют наши действия. Т.е. в данном случае у нас представлена система для написания каких-то квизов, и у нас есть администратор и студент. Дальше прописаны наши функциональности, которые мы реализовываем. Всё, что можно произвести внутри этого блока создания наших квизов. Администратор может логиниться в систему, управлять нашими квизами, добавлять их, удалять, апдейтить и т.д. Студент - регистрироваться, логиниться и т.д.



User story. Во-первых, у нас есть описание нашей user story, что мы хотим получить в рамках этой функциональности. Как она пишется? Сначала мы пишем следующую фразу: “как участник конференции, как пользователь, я хочу иметь возможность регистрироваться онлайн для того, чтобы регистрироваться быстро и снизить количество бумажной работы”. Здесь есть три логических блока. Т.е., как пользователь, я хочу получить такую функциональность, которая позволяла бы мне сделать следующее. Дальше у нас прописан acceptance criteria - это критерии приемки. То, что у нас должно быть реализовано в нашем требовании в нашей user story для того, чтобы мы в дальнейшем смогли сказать, что ваша функциональность реализована. Что делает тестировщик? Он получает эту юзер стори и должен придумать тест-кейсы для каждого statement, который прописан в acceptance criteria для того, чтобы убедиться, что да, действительно, функциональность реализована правильно и все работает корректно. Т.е. мы берем каждый этот statement и проверяем его. Этим занимаются аналитики и продукт оунеры, т.е. написанием самих требований, поэтому наша задача проревьюить, протестировать сами требования на предмет того, что они соответствуют как свойствам хорошего требования, так и в принципе здравого смысла.



UI-mockup. Это шаблон для мобильного приложения, т.е. мы здесь подробно расписываем как выглядит каждая кнопка, какие поля есть в нем, что мы должны вводить, какие навигационные сообщения у нас выскакивают, т.е. этим занимается больше UI-дизайнер. Чаще всего шаблоны уже появляются после того, когда написаны требования, но также, к примеру, если заказчик проявит, скажем так, свою инициативу, и он хочет, чтобы действительно всё выглядело так, то он может предоставить уже разработанные шаблоны. Это может выглядеть просто в виде некой схемы, к примеру, там просто прописаны все блоки которые, у нас должны быть в нашем приложении и для каждого блока прописано, что должно быть на каждой из этих страниц, т.е. и такой вариант написания требований может быть. Иногда вообще это может быть просто клочок бумаги, на котором прописано, что я хочу получить то-то, то-то и то-то, просто как-то высокоуровнево, без каких-то уточнений на эту тему.



Спецификации. Это такие документы, в которых тоже прописаны наши требования. Их написанием уже занимаются непосредственно бизнес-аналитики. Спецификациями проверяем точно так же. Мы находим требования, мы проверяем эти требования внутри спецификации согласно свойствам хорошего требования и выносим вердикт для того, чтобы в дальнейшем смогли это поправить и наше требование соответствовало всем необходимым свойствам.

Классификация тестирования

Рисунок 2.3.а — Упрощённая классификация тестирования

- **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.

Таблица 2.3.а — Преимущества и недостатки методов белого, чёрного и серого ящиков

	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 автоматизируется. • Стимулирует разработчиков к написанию качественного кода. • Многие техники этого метода являются проверенными, хорошо себя зарекомендовавшими 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[143].
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[143]. • 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 ящика.	

- **По степени автоматизации:**

- *Ручное тестирование* — тест-кейсы выполняет человек.

- *Автоматизированное тестирование* — тест-кейсы частично или полностью выполняет специальное инструментальное средство.

Таблица 2.3.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.). • 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.	• 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. • 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 (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.

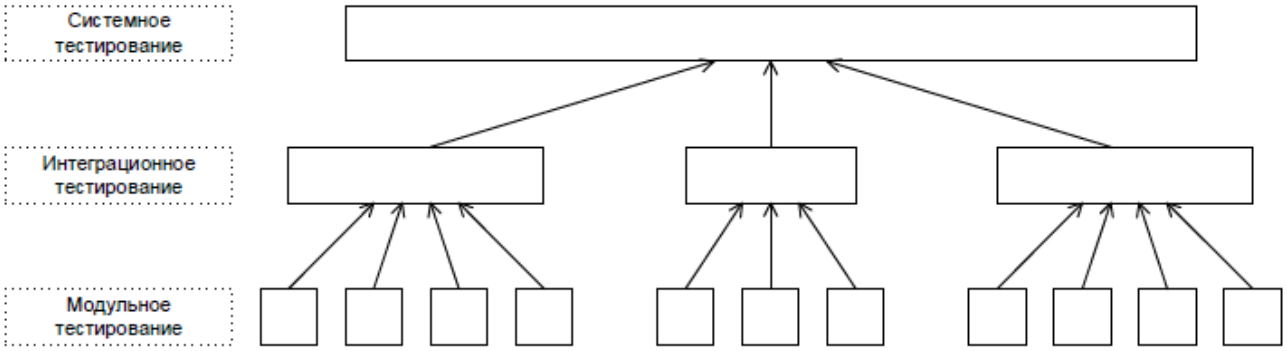


Рисунок 2.3.d — Схематичное представление классификации тестирования по уровню детализации приложения

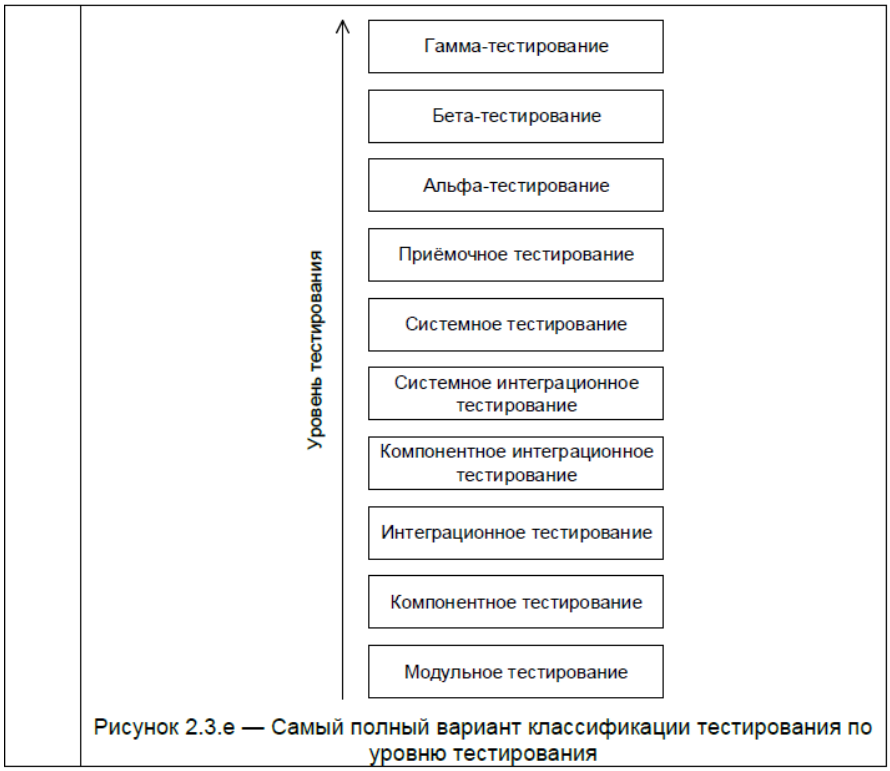


Рисунок 2.3.e — Самый полный вариант классификации тестирования по уровню тестирования

• **По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):**

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

- *Тестирование критического пути* — проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности.

- *Расширенное тестирование* — проверка всей (остальной) функциональности, заявленной в требованиях.

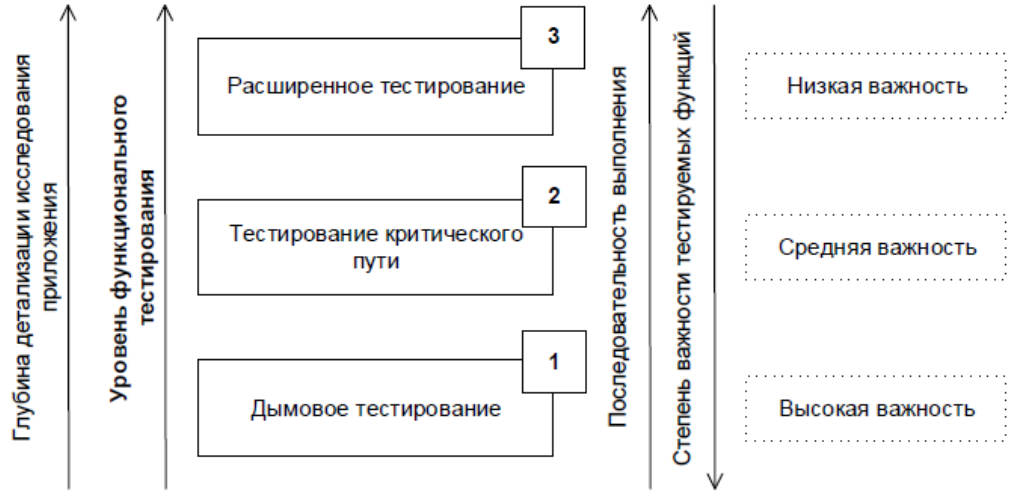


Рисунок 2.3.h — Классификация тестирования по (убыванию) степени важности тестируемых функций (по уровню функционального тестирования)

• **По принципам работы с приложением:**

- *Позитивное тестирование* — все действия с приложением выполняются строго по инструкции без никаких 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 ноль).

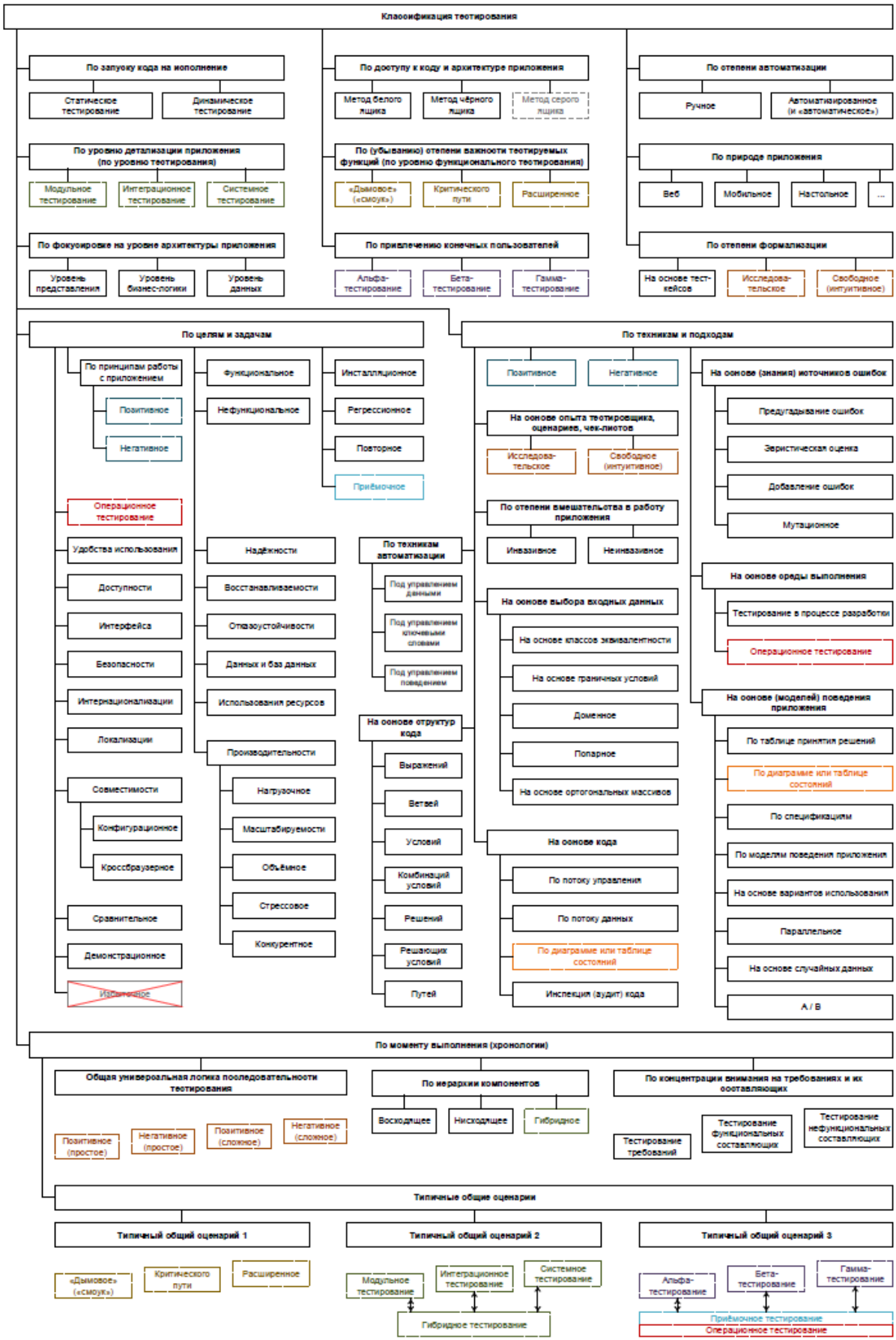


Рисунок 2.3.b — Подробная классификация тестирования (русскоязычный вариант)

Функциональное тестирование (functional testing) — вид тестирования, направленный на проверку корректности работы функциональности приложения (корректность реализации функциональных требований).

Нефункциональное тестирование (non-functional testing) — вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований).

Регрессионное тестирование (regression testing) — тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования.

Приёмочное тестирование (acceptance testing) — 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).

Тестирование надёжности (reliability testing) — тестирование способности приложения выполнять свои функции в заданных условиях на протяжении заданного времени или заданного количества операций.

Тестирование восстанавливаемости (recoverability testing) — 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) — 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) — исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке. В рамках тестирования производительности выделяют следующие подвиды:

- *Нагрузочное тестирование* (load testing, capacity testing) — исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).
- *Тестирование масштабируемости* (scalability testing) — исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов.
- *Объёмное тестирование* (volume testing) — исследование производительности приложения при обработке различных (как правило, больших) объёмов данных.
- *Стрессовое тестирование* (stress testing) — 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) и представляет собой крайнюю форму негативного тестирования.
- *Конкурентное тестирование* (concurrency testing) — 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.).

Классификация на основе выбора входных данных:

- *Тестирование на основе классов эквивалентности* (equivalence partitioning) — 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 из таких наборов небольшого подмножества тест-кейсов, с наибольшей вероятностью обнаруживающих проблему.
- *Тестирование на основе граничных условий* (boundary value analysis) — 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, 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) — 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) 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).

- *Тестирование на основе ортогональных массивов* (orthogonal array testing) — инструментальная техника попарного и N-комбинационного тестирования, основанная на использовании т.н. «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).

Чек-лист (checklist) — 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:

• в котором последовательность пунктов не имеет значения (например, список значений некоего поля);

• в котором последовательность пунктов важна (например, шаги в краткой инструкции);

• структурированный (многоуровневый) список, который позволяет отразить иерархию идей.

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

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

Последовательность и структурированность. Со структурированностью 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 неких небольших групп идей, переход между которыми является понятным и очевидным (например, сначала можно прописать идеи простых позитивных тест-кейсов, потом идеи простых негативных тест-кейсов, потом постепенно повышать сложность тест-кейсов, но не стоит писать эти идеи вперемешку).

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

Тест-кейс (test case) — набор входных данных, условий выполнения и ожидаемых результатов, разработанный с целью проверки того или иного свойства или поведения программного средства.

Высокоуровневый тест-кейс (high level test case) — 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) — 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 снизив ценность тест-кейса.

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 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: написание чек-листов и тест-кейсов — хорошая техника тестирования требований).

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

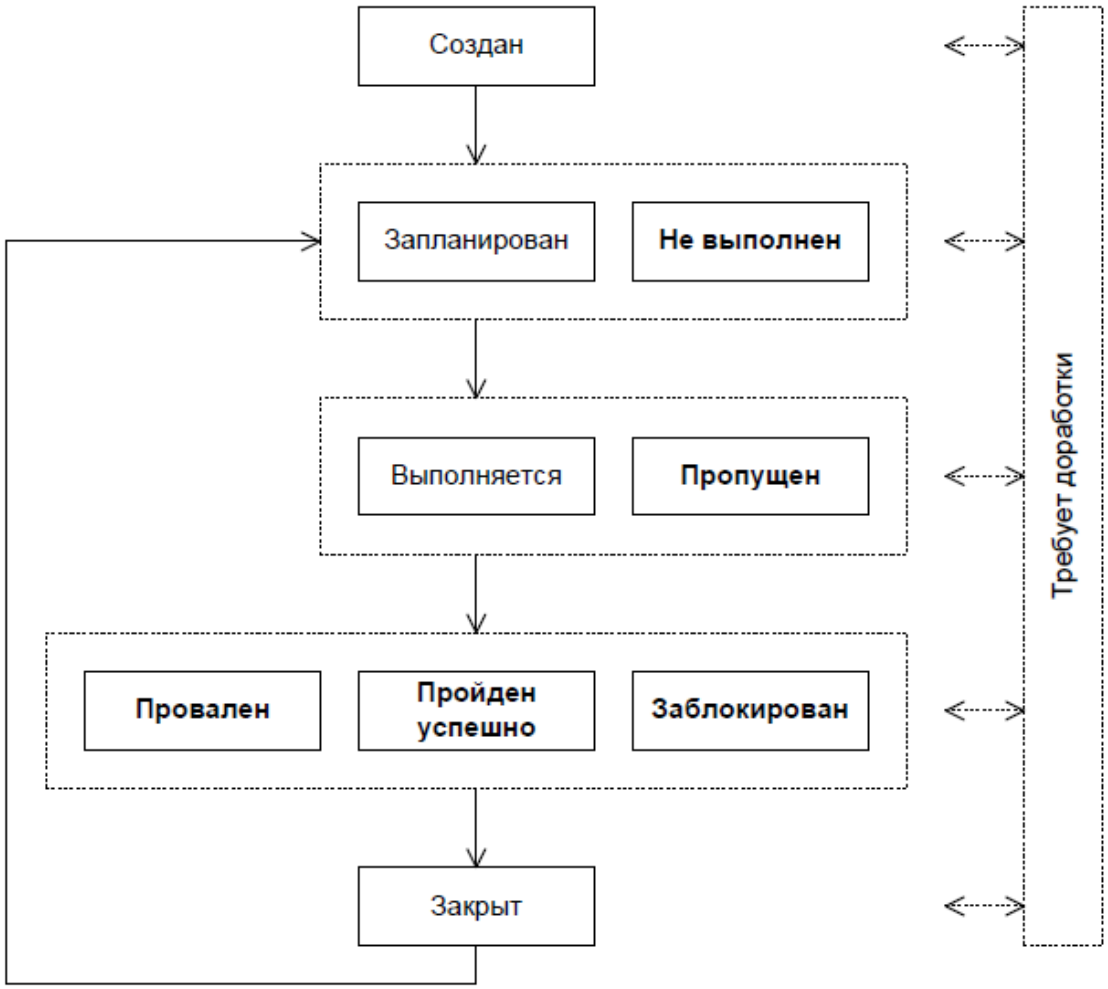


Рисунок 2.4.a — Жизненный цикл (набор состояний) тест-кейса

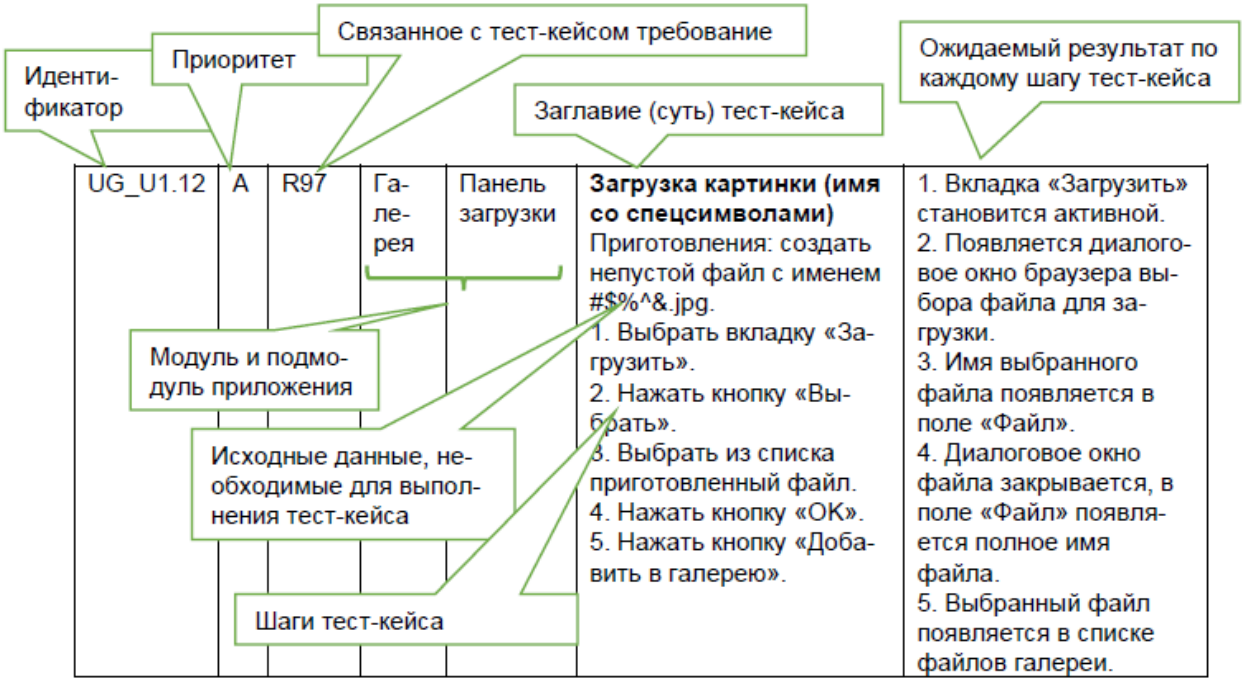


Рисунок 2.4.b — Общий вид тест-кейса

Свойства качественных тест-кейсов:

1. Правильный технический язык, точность и единообразие формулировок:

• используйте безличную форму глаголов (например, «открыть» вместо «от-кройте»);

• пишите лаконично, но понятно;

• 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);

• везде называйте одни и те же вещи одинаково (например, нельзя в одном тест-кейсе некий режим работы приложения назвать «графическое представление», а в другом тот же режим — «визуальное отображение», т.к. многие люди могут подумать, что речь идёт о разных вещах);

• следуйте принятому на проекте стандарту оформления и написания тест-кейсов (иногда такие стандарты могут быть весьма жёсткими: вплоть до регламентации того, названия каких элементов должны быть приведены в двойных кавычках, а каких — в одинарных).

2. Баланс между специфичностью и общностью.

3. 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 они совершенно не обязаны это делать).

4. «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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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 нарушений этого свойства можно долго с удивлением смотреть, например, на какое требование ссылается тест-кейс, и пытаться понять, как же они друг с другом связаны.

10. Возможность повторного использования. Это свойство редко выполняется для низкоуровневых тест-кейсов, но при создании высокоуровневых тест-кейсов можно добиться таких формулировок, при которых:

• тест-кейс будет пригодным к использованию с различными настройками тестируемого приложения и в различных тестовых окружениях;

• тест-кейс практически без изменений можно будет использовать для тестирования аналогичной функциональности в других проектах или других областях приложения.

11. Повторяемость. Тест-кейс должен быть сформулирован таким образом, чтобы при многократном повторении он показывал одинаковые результаты. Это свойство можно разделить на два подпункта:

• во-первых, даже общие формулировки, допускающие разные варианты выполнения тест-кейса, должны очерчивать соответствующие явные границы (например: «ввести какое-нибудь число» — плохо, «ввести целое число в диапазоне от -273 до +500 включительно» — хорошо);

• действия (шаги) тест-кейса по возможности не должны приводить к необратимым (или сложно обратимым) последствиям (например: удалению данных, нарушению конфигурации окружения и т.д.) — не стоит включать в тест-кейс такие «разрушительные действия», если они не продиктованы явным образом целью тест-кейса; если же цель тест-кейса обязывает нас к выполнению таких действий, в самом тест-кейсе должно быть описание действий по восстановлению исходного состояния приложения (данных, окружения).

12. 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 иного совета, кроме как «почитайте уже готовые тест-кейсы перед тем как писать свои».

Набор тест-кейсов (test case suite, 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.

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.

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 «точечные» воздействия на приложение.

Таблица 2.4.c — Подробная классификация наборов тест-кейсов

		По изолированности тест-кейсов друг от друга	
		Изолированные	**Обобщённые**
По образованию тест-кейсами строгой последовательности	**Свободные**	Изолированные свободные	Обобщённые свободные
	Последовательные	Изолированные последовательные	Обобщённые последовательные

• Набор изолированных свободных тест-кейсов (рисунок 2.4.g): 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 2.4.h): 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 2.4.i): 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 2.4.j): действия из раздела «приготовления» нужно выполнить один раз (а потом просто выполнять тест-кейсы), а сами тест-кейсы нужно выполнять в строго определённом порядке.

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 некоего тест-кейса (приложение не пришло в ожидаемое состояние) остальные тест-кейсы по-прежнему можно выполнять.

Рисунок 2.4.g — Набор изолированных свободных тест-кейсов

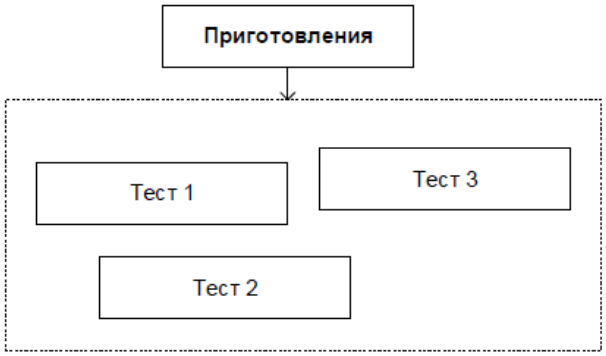


Рисунок 2.4.h — Набор обобщённых свободных тест-кейсов

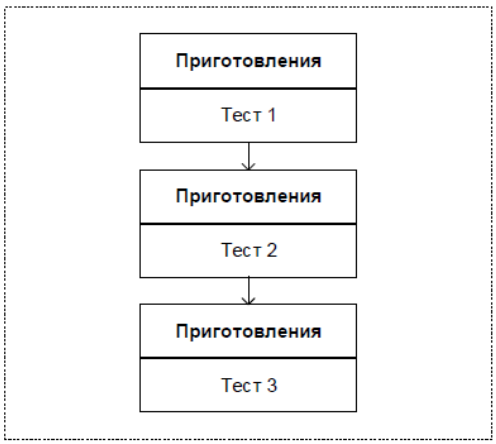


Рисунок 2.4.i — Набор изолированных последовательных тест-кейсов

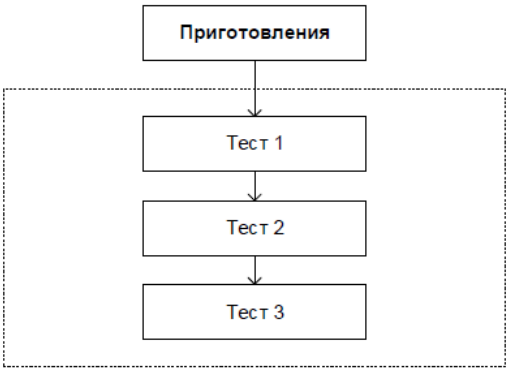


Рисунок 2.4.j — Набор обобщённых последовательных тест-кейсов

Логика создания эффективных проверок

Приступая к продумыванию чек-листа, 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 уйдёте дальше бездумных формальных проверок.

• Кому и зачем оно нужно (и насколько это важно)? Ответ на этот вопрос позволит вам быстро придумать несколько характерных сценариев использования 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 (их тоже удобно оформить в виде чек-листа).

Дефект — отклонение (deviation) фактического результата (actual result) от ожиданий наблюдателя (expected result), 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, в архитектуре и дизайне, в настройках тестируемого приложения или тестового окружения — где угодно.

Отчёт о дефекте (defect report) — документ, описывающий и приоритизирующий 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;

• приоритизировать проблему — определить степень опасности проблемы для проекта и желаемые сроки 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.

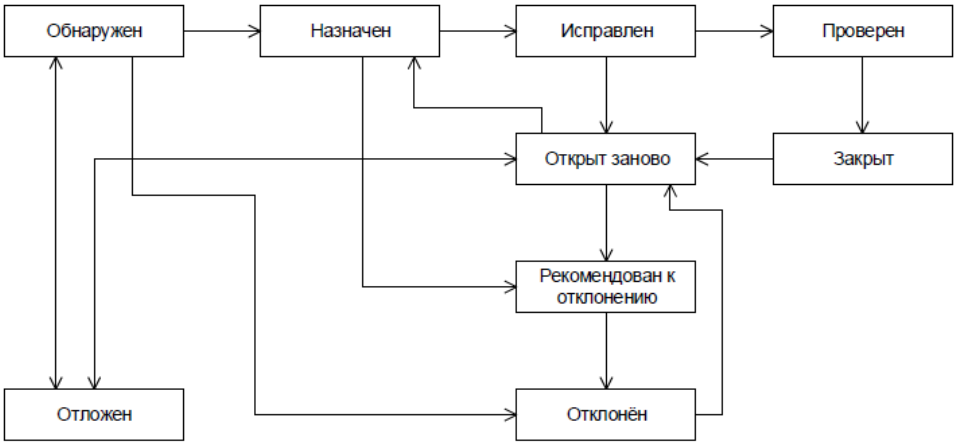


Рисунок 2.5.c — Жизненный цикл отчёта о дефекте с наиболее типичными переходами между состояниями

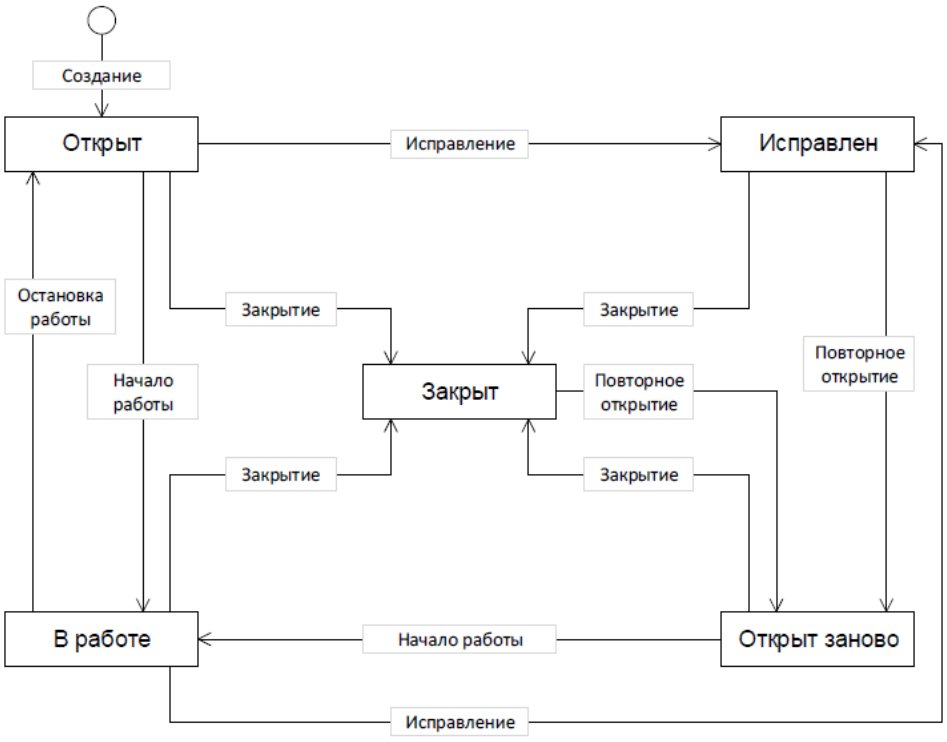


Рисунок 2.5.d — Жизненный цикл отчёта о дефекте в JIRA

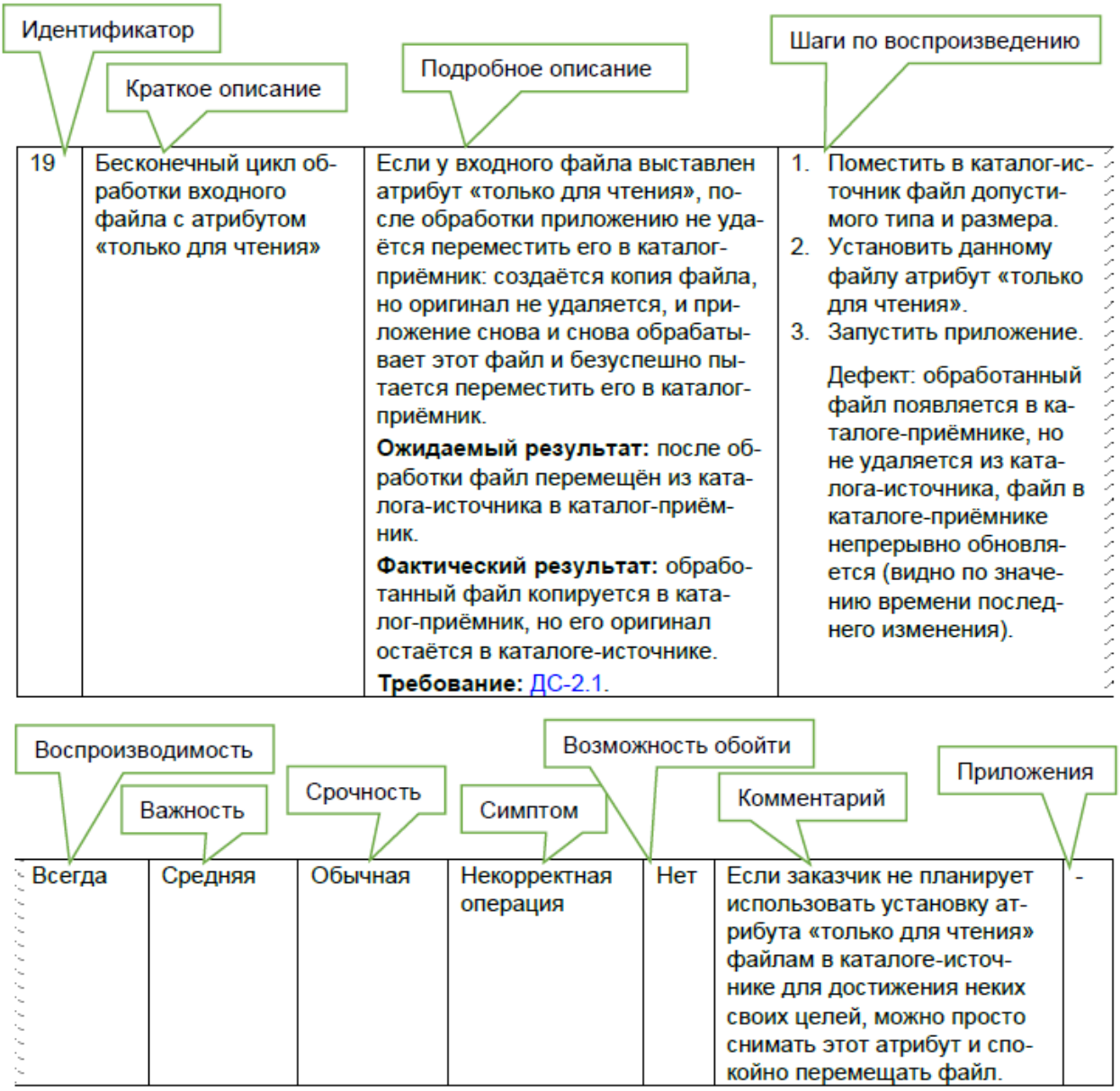


Рисунок 2.5.е — Общий вид отчёта о дефекте

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.

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 достаточную 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 1–2 вопроса, 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;

• по возможности не дублировать краткие описания других дефектов (и даже не быть похожими на них), чтобы дефекты было сложно перепутать или посчитать дубликатами друг друга;

• быть законченным предложением русского или английского (или иного) языка, 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, кристально чистого понимания того, «что сломалось», писать отчёт о дефекте вообще 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 получившееся 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.

Подробное описание (description) представляет в развёрнутом виде необходимую информацию о дефекте, а также (обязательно!) описание фактического результата, ожидаемого результата и ссылку на требование (если это возможно).

Пример подробного описания:

Если в систему входит администратор, на странице приветствия отсутствует логотип.
Фактический результат: логотип отсутствует в левом верхнем углу страницы.
Ожидаемый результат: логотип отображается в левом верхнем углу страницы.
Требование: R245.3.23b.

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.

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.

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, раскрытие 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, некое 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.

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 (см. инсталляционное тестирование).

• Ошибка локализации (localization issue) — что-то в приложении не переведено или переведено неверно на выбранный язык интерфейса (см. локализационное тестирование).

• 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 (см. тестирование производительности и тестирование масштабируемости).

• Низкая производительность (low performance) — выполнение неких операций занимает недопустимо большое время (см. тестирование производительности).

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, сюда можно писать всё то, что нельзя писать в остальные поля.

Вложения (attachments) — представляет собой не столько поле, сколько список прикреплённых к отчёту о дефекте приложений (копий экрана, вызывающих сбой файлов и т.д.).

Окружение (Environment). Всегда необходимо указывать окружение, на котором был обнаружен данный дефект. Операционная система, разрядность, service pack, браузер, в котором был обнаружен данный дефект, его версия. Т.е. мы подробно здесь расписываем в каком именно окружении был обнаружен наш дефект. Также в окружении может быть прописано, к примеру, если у нас на проекте есть несколько окружений, например, есть окружение для разработчиков DEV, есть окружение стабильной версии STAGE, и есть окружение продакшна PROD, когда уже в нем работает наш конечный пользователь. Давайте с вами пометим эти три вида окружения. Я расскажу вам тоже немножко более подробнее о них. Как я уже сказал, на каждом из этих окружений работает определенный круг лиц.

- DEV окружение - обычно работают разработчики, т.е. они туда заливают какие-то фичи, возможно также подключаются тестировщики к тестированию каких-то отдельных фич на DEV окружении, и также могут заводить дефекты. Если мы обнаруживаем такие дефекты на этом окружении, мы прописываем в этом поле DEV.
- STAGE окружение - на него попадает уже какая-то стабильная версия нашего приложения, на нем чаще всего работают тестировщики и именно на stage доводят это приложение до определенного предрелизного состояния.
- PROD окружение - если на stage не остаётся дефектов, то такое приложение переводится на стадию прод, где у нас работают наши конечные пользователи.

Свойства качественных отчётов о дефектах:

Тщательное заполнение всех полей точной и корректной информацией. Нарушение этого свойства происходит по множеству причин: недостаточный опыт тестировщика, невнимательность, лень и т.д.

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

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. Потому, если у вас есть хоть малейшее сомнение в том, важна ли какая-то деталь, считайте, что она важна.

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. 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.

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 некую проблему, и теперь описывает её заново.

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».

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.

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».

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.

Логика создания эффективных отчётов о дефектах

При создании отчёта о дефекте рекомендуется следовать следующему алгоритму:

0. Обнаружить дефект ☺.
1. Понять суть проблемы.
2. Воспроизвести дефект.
3. Проверить наличие описания найденного вами дефекта в системе управления дефектами.
4. Сформулировать суть проблемы в виде «что сделали, что получили, что ожидали получить».
5. Заполнить поля отчёта, начиная с подробного описания.
6. После заполнения всех полей внимательно перечитать отчёт, исправив неточности и добавив подробности.
7. Ещё раз перечитать отчёт, т.к. в пункте 6 вы точно что-то упустили.

Типичные ошибки при написании отчётов о дефектах

Ошибки оформления и формулировок:

- Плохие краткие описания (summary)
- Идентичные краткое и подробное описания
- Отсутствие в подробном описании явного указания фактического результата, ожидаемого результата и ссылки на требование
- Игнорирование кавычек, приводящее к искажению смысла
- Общие проблемы с формулировками фраз на русском и английском языках
- Лишние пункты в шагах воспроизведения
- Копии экрана в виде «копий всего экрана целиком»
- Копии экрана, на которых не отмечена проблема
- Откладывание написания отчёта «на потом»
- Пунктуационные, орфографические, синтаксические и им подобные ошибки.

Логические ошибки:

- Выдуманные дефекты.
- Отнесение расширенных возможностей приложения к дефектам
- Неверно указанные симптомы
- Чрезмерно заниженные (или завышенные) важность и срочность
- Концентрация на мелочах в ущерб главному
- Техническая безграмотность
- Указание в шагах воспроизведения информации, неважной для воспроизведения ошибки
- Отсутствие в шагах воспроизведения информации, важной для воспроизведения дефекта
- Игнорирование т.н. «последовательных дефектов».

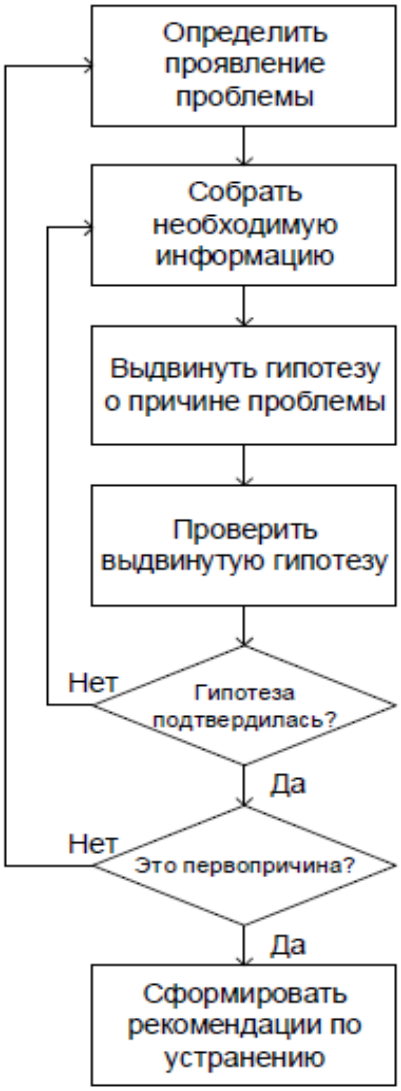


Рисунок 2.7.j — Алгоритм поиска первопричины дефекта

Тест-план (test plan) — 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:

• 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:

• *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) и подходы (test approach). 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) — любые объективные показатели качества, которым разрабатываемый продукт должен соответствовать с точки зрения заказчика или пользователя, чтобы считаться готовым к эксплуатации.

- Критерии начала тестирования (entry criteria) — 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) — 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) — перечень условий, при выполнении которых тестирование возобновляется (как правило, после приостановки).

- Критерии завершения тестирования (exit criteria) — 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:

1. программные ресурсы (какое ПО необходимо команде тестировщиков, сколько копий и с какими лицензиями (если речь идёт о коммерческом ПО));

2. аппаратные ресурсы (какое аппаратное обеспечение, в каком количестве и к какому моменту необходимо команде тестировщиков);

3. человеческие ресурсы (сколько специалистов какого уровня и со знаниями в каких областях должно подключиться к команде тестировщиков в тот или иной момент времени);

4. временные ресурсы (сколько по времени займёт выполнение тех или иных работ);

5. 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). Фактически это календарь, в котором указано, что и к какому моменту должно быть сделано. Особое внимание уделяется т.н. «ключевым точкам» (milestones), к моменту наступления которых должен быть получен некий 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). Числовые характеристики показателей качества, способы их оценки, формулы и т.д. На этот раздел, как правило, формируется множество ссылок из других разделов тест-плана.

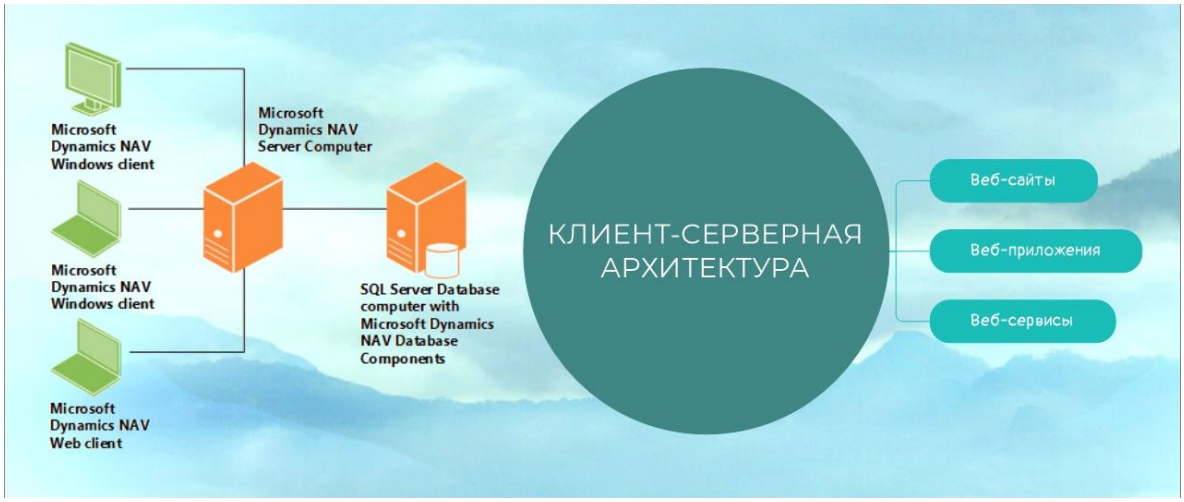
Метрика (metric) — числовая характеристика показателя качества. Может включать описание способов оценки и анализа результата.

В тестировании существует большое количество общепринятых метрик, многие из которых могут быть собраны автоматически с использованием инструментальных средств управления проектами. Например:

• процентное отношение (не) выполненных тест-кейсов ко всем имеющимся;
• процентный показатель успешного прохождения тест-кейсов;
• процентный показатель заблокированных тест-кейсов;
• плотность распределения дефектов;
• эффективность устранения дефектов;
• распределение дефектов по важности и срочности;

Покрытие (coverage) — процентное выражение степени, в которой исследуемый элемент (coverage item) затронут соответствующим набором тест-кейсов.

Клиент-серверная архитектура. Веб-сайт, веб-приложение и веб-сервис



«Клиент — сервер» (англ. client–server) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных) или в виде сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, её размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики её оборудования и программного обеспечения, её также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Когда мы говорим только о том, что у нас есть клиент и сервер, которые взаимодействуют между собой, клиент отправляет запрос (request) на сервер, сервер его обрабатывает и отправляет клиенту обратно ответ (response), т.е. ответ от сервера.

Преимущества

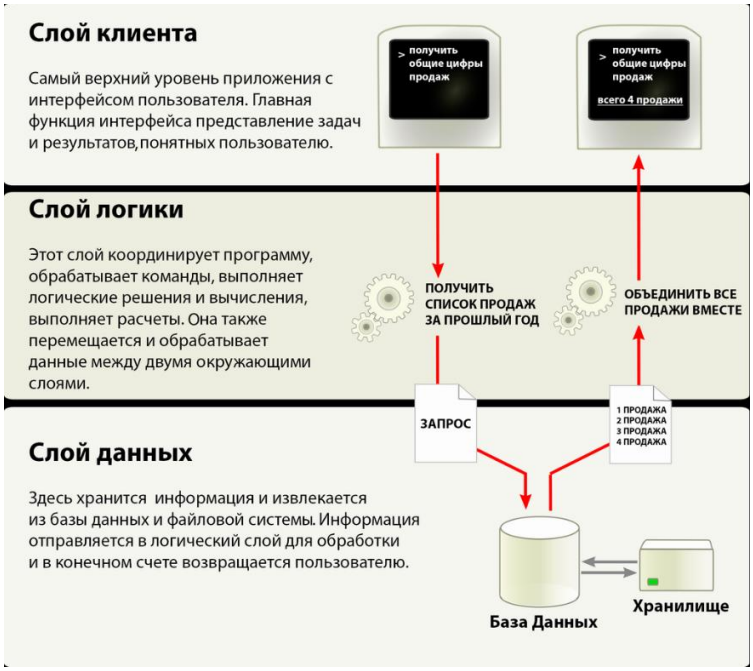
- Отсутствие дублирования кода программы-сервера программами-клиентами.
- Так как все вычисления выполняются на сервере, то требования к компьютерам, на которых установлен клиент, снижаются.
- Все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще организовать контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа. Обычно этим занимается отдельно выделенный человек - системный администратор.

Недостатки

- Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Неработоспособным сервером следует считать сервер, производительности которого не хватает на обслуживание всех клиентов, а также сервер, находящийся на ремонте, профилактике и т. п.
- Поддержка работы данной системы требует отдельного специалиста — системного администратора.
- Высокая стоимость оборудования.

Также, говоря о клиентах и серверах, стоит поговорить о так называемой трехуровневой архитектуре. Т.е., если у нас есть только клиент и сервер, то такая архитектура двухуровневая, но когда у нас появляется база данных, то это трехуровневая.

Трёхуро́вневая архитекту́ра (англ. three-tier) — архитектурная модель программного комплекса, предполагающая наличие в нём трёх компонентов: клиента, сервера приложений (к которому подключено клиентское приложение) и сервера баз данных (с которым работает сервер приложений).



Например, если мы говорим о какой-то социальной сети, когда клиент отправляет информацию на сервер с запросом того, что, например, ищет, и эта информация содержится в базе данных, то серверу необходимо обратиться к ней, чтобы получить данную информацию, т.е. у нас добавляется дополнительно база данных и её сервер, и тогда у нас есть вот этих три звена, т.е. информация с базы данных уходит обратно на сервер и через сервер попадает обратно на клиент.

Самым распространённым примером клиента является браузер. Все мы с вами знаем, что это такое. Например, это google, chrome, либо сафари, либо mozilla, либо опера. Т.е. мы вводим туда какие-то наши запросы, по сути, это то, что мы вводим в адресную строку, и этот запрос уходит на сервер.

Компоненты

Клиент (слой клиента) — это интерфейсный (обычно графический) компонент комплекса, предоставляемый конечному пользователю. Этот уровень не должен иметь прямых связей с базой данных (по требованиям безопасности и масштабируемости), быть нагруженным основной бизнес-логикой (по требованиям масштабируемости) и хранить состояние приложения (по требованиям надёжности). На этот уровень обычно выносится только простейшая бизнес-логика: интерфейс авторизации, алгоритмы шифрования, проверка вводимых значений на допустимость и соответствие формату, несложные операции с данными (сортировка, группировка, подсчёт значений), уже загруженными на терминал.

Тонкий клиент (англ. thin client) в компьютерных технологиях — компьютер или программа-клиент в сетях с клиент-серверной или терминальной архитектурой, который переносит все или большую часть задач по обработке информации на сервер. Примером тонкого клиента может служить компьютер с браузером, использующийся для работы с веб-приложениями.

Толстый клиент (англ. fat client; rich client; heavy client; thick client; также, Rich-клиент) — в архитектуре клиент — сервер — это приложение, обеспечивающее (в противовес тонкому клиенту) расширенную функциональность независимо от центрального сервера. Часто сервер в этом случае является лишь хранилищем данных, а вся работа по обработке и представлению этих данных переносится на машину клиента. Например, если взять такое сложное программное обеспечение, как 1с бухгалтерия, т.е. в нем содержится вся основная бизнес-логика, на сервер у нас передаются только те данные, которые необходимо сохранить в базе данных, либо же когда нам будет необходимо их получить обратно, отправить какой-то запрос и получить из базы данных ответ. А всё остальное основное находится именно вот на этом толстом клиенте в 1с.

Достоинства

- Толстый клиент обладает широкой функциональностью в отличие от тонкого.
- Режим многопользовательской работы.

- Предоставляет возможность работы даже при обрывах связи с сервером.
- Высокое быстродействие (зависит от аппаратных средств клиента)

Недостатки

- Большой размер дистрибутива.
- Многое в работе клиента зависит от того, для какой платформы он разрабатывался.
- При работе с ним возникают проблемы с удаленным доступом к данным.
- Довольно сложный процесс установки и настройки.
- Сложность обновления и связанная с ней неактуальность данных.
- Наличие бизнес-логики

Сервер приложений (средний слой, связующий слой) располагается на втором уровне, на нём сосредоточена бóльшая часть бизнес-логики. Вне его остаются только фрагменты, экспортируемые на клиента (терминалы), а также элементы логики, погруженные в базу данных (хранимые процедуры и триггеры). Реализация данного компонента обеспечивается связующим программным обеспечением.

Сервер баз данных (слой данных) обеспечивает хранение данных и выносится на отдельный уровень, реализуется, как правило, средствами систем управления базами данных, подключение к этому компоненту обеспечивается только с уровня сервера приложений.

Веб-сайты обычно носят какой-то информационный характер, т.е. они состоят из неких веб-страниц, объединённых друг с другом в единый ресурс, имеют какую-то простую архитектуру на основе html кода. Такие вот веб-сайты, по сути, служат в качестве платформы для предоставления контента для посетителей, они содержат какие-то текстовые файлы, изображения, возможно музыку. Сайты не предоставляют возможности взаимодействия с нашей программой, т.е. пользователи не имеют доступа к размещению своей информации, кроме как заполнение формы для получения, к примеру, подписки. Наиболее яркими примерами типичных сайтов могут быть новостные, кулинарные сайты, прогнозы погоды.

Веб-приложения. В отличие от веб-сайта, веб-приложения - это такие интерактивные компьютерные приложения, которые специально разрабатываются для сети интернет и позволяет пользователям вводить, получать и манипулировать данными с помощью взаимодействия. Такие программы имеют очень тесную связь с нашим сервером и отправляют на него очень много запросов. Такие приложения могут быть встроены в ПО страницы, либо же сами веб-страницы могут являться приложениями. К ним можно отнести, к примеру, фейсбук, gmail, youtube, ebay, twitter, различные соцсети. Тот же вконтакте, если взять приложение, используют имя пользователя и пароль для аутентификации и позволяют своим посетителям обмениваться, например, мгновенными сообщениями. Если мы говорим о соцсетях, либо же каких-то блогах - создавать контент на основе пользовательских предпочтений, обеспечивать к нему неограниченный доступ, также там могут быть встроены какие-то мини-программы для развлечений. И еще одно отличие от веб-сайтов то, что многие интернет приложения могут не иметь реального информативного содержания. Что же это значит? Т.е. они используются для выполнения каких-то дополнительных задач, т.е. это могут быть какие-то интернет переводчики, мессенджеры, конвертеры файлов, конвертеры валют, все, что угодно.

Веб-сервисы. Если мы с вами вспомним один из первых уроков, посвященных уровням тестирования, то один из них был интеграционный, и вот именно логика интеграционного уровня привязана к веб-сервису. Это прикладной программный интерфейс (API), который работает на сервере и предоставляет клиенту данные через http протокол, через стандартизированную систему обмена сообщениями. Веб-сервисы, в свою очередь, подразделяются на SOAP и REST.

Просто запомните, что веб-сайты - это некие простые веб-страницы, которые представляют собой информационную какую-то нагрузку, а веб-приложения уже позволяют пользователю взаимодействовать с этими разными веб-страницами, вводить свои какие-то данные, взаимодействовать с контентом, а веб-сервис - это наш какой-то прикладной интерфейс программы, который позволяет различным веб-приложениям взаимодействовать между собой.

веб-сайт - frontend и только он

веб-приложение - frontend + backend

веб-сервис - API

Сетевая модель OSI (The Open Systems Interconnection model) — сетевая модель стека сетевых протоколов OSI/ISO. Посредством данной модели различные сетевые устройства могут взаимодействовать друг с другом. Модель определяет различные уровни взаимодействия систем. Каждый уровень выполняет определённые функции при таком взаимодействии.

Уровни модели OSI [править | править код]

Модель					
Уровень (layer)		Тип данных (PDU[14])	Функции	Примеры	Оборудование
Host layers	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, POP3, WebSocket	Хосты (клиенты сети)
	6. Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC	
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP	
	4. Транспортный (transport)	Сегменты (segment) / Датаграммы (datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP, SCTP, PORTS	
Media[15] layers	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IPv4, IPv6, IPsec, AppleTalk, ICMP	Маршрутизатор
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.	Коммутатор, точка доступа
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, кабель («витая пара», коаксиальный, оптоволоконный), радиоканал	Концентратор, Повторитель (сетевое оборудование)

Уровни модели OSI:

Прикладной уровень (уровень приложений; англ. application layer) — верхний уровень модели, обеспечивающий взаимодействие пользовательских приложений с сетью:

- позволяет приложениям использовать сетевые службы: удалённый доступ к файлам и базам данных, пересылка электронной почты;
- отвечает за передачу служебной информации;
- предоставляет приложениям информацию об ошибках;
- формирует запросы к уровню представления.

Уровень представления (англ. presentation layer) обеспечивает преобразование протоколов и кодирование/декодирование данных. Запросы приложений, полученные с прикладного уровня, на уровне представления преобразуются в формат для передачи по сети, а полученные из сети данные преобразуются в формат приложений. На этом уровне может осуществляться сжатие/распаковка или шифрование/дешифрование, а также перенаправление запросов другому сетевому ресурсу, если они не могут быть обработаны локально.

Сеансовый уровень (англ. session layer) модели обеспечивает поддержание сеанса связи, позволяя приложениям взаимодействовать между собой длительное время. Уровень управляет созданием/завершением сеанса, обменом информацией, синхронизацией задач, определением права на передачу данных и поддержанием сеанса в периоды неактивности приложений.

Транспортный уровень (англ. transport layer) модели предназначен для обеспечения надёжной передачи данных от отправителя к получателю.

Сетевой уровень (англ. network layer) модели предназначен для определения пути передачи данных. Отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок и «заторов» в сети.

Канальный уровень (англ. data link layer) предназначен для обеспечения взаимодействия сетей на физическом уровне и контроля ошибок, которые могут возникнуть. Полученные с физического уровня данные, представленные в битах, он упаковывает в кадры, проверяет их на целостность и, если нужно, исправляет ошибки (либо формирует повторный запрос повреждённого кадра) и отправляет на сетевой уровень. Канальный уровень может взаимодействовать с одним или несколькими физическими уровнями, контролируя и управляя этим взаимодействием.

Физический уровень (англ. physical layer) — нижний уровень модели, который определяет метод передачи данных, представленных в двоичном виде, от одного устройства (компьютера) к другому.

TCP/IP — сетевая модель передачи данных, представленных в цифровом виде. Модель описывает способ передачи данных от источника информации к получателю. В модели предполагается прохождение информации через четыре уровня, каждый из которых описывается правилом (протоколом передачи). Название TCP/IP происходит из двух важнейших протоколов семейства — Transmission Control Protocol (TCP) и Internet Protocol (IP).

Стек протоколов TCP/IP включает в себя четыре уровня:

- *Прикладной уровень* (Application Layer)

  На прикладном уровне работает большинство сетевых приложений. Эти программы имеют свои собственные протоколы обмена информацией, например, интернет браузер для протокола HTTP, ftp-клиент для протокола FTP (передача файлов), почтовая программа для протокола SMTP (электронная почта), SSH (безопасное соединение с удалённой машиной), DNS (преобразование символьных имён в IP-адреса) и многие другие.
- *Транспортный уровень* (Transport Layer)

  Протоколы транспортного уровня могут решать проблему негарантированной доставки сообщений («дошло ли сообщение до адресата?»), а также гарантировать правильную последовательность прихода данных.

  TCP - «гарантированный» транспортный механизм с предварительным установлением соединения, предоставляющий приложению надёжный поток данных, дающий уверенность в безошибочности получаемых данных, перезапрашивающий данные в случае потери и устраняющий дублирование данных. TCP позволяет регулировать нагрузку на сеть, а также уменьшать время ожидания данных при передаче на большие расстояния. Более того, TCP гарантирует, что полученные данные были отправлены точно в такой же последовательности. В этом его главное отличие от UDP. В приложениях, требующих гарантированной передачи данных, используется протокол TCP.

  UDP - протокол передачи датаграмм без установления соединения. Также его называют протоколом «ненадёжной» передачи, в смысле невозможности удостовериться в доставке сообщения адресату, а также возможного перемешивания пакетов. UDP обычно используется в таких приложениях, как потоковое видео и компьютерные игры, где допускается потеря пакетов, а повторный запрос затруднён или не оправдан, либо в приложениях вида запрос-ответ (например, запросы к DNS), где создание соединения занимает больше ресурсов, чем повторная отправка.
- *Межсетевой уровень* (Internet Layer)

  Межсетевой уровень изначально разработан для передачи данных из одной сети в другую. На этом уровне работают маршрутизаторы, которые перенаправляют пакеты в нужную сеть путём расчёта адреса сети по маске сети.
- *Канальный уровень* (Network Access Layer).

  Канальный уровень описывает способ кодирования данных для передачи пакета данных на физическом уровне (то есть специальные последовательности бит, определяющих начало и конец пакета данных, а также обеспечивающие помехоустойчивость). Ethernet, например, в полях заголовка пакета содержит указание того, какой машине или машинам в сети предназначен этот пакет.

  Кроме того, канальный уровень описывает среду передачи данных (будь то коаксиальный кабель, витая пара, оптическое волокно или радиоканал), физические характеристики такой среды и принцип передачи данных (разделение каналов, модуляцию, амплитуду сигналов, частоту сигналов, способ синхронизации передачи, время ожидания ответа и максимальное расстояние).

Протоколы этих уровней полностью реализуют функциональные возможности модели OSI. На стеке протоколов TCP/IP построено всё взаимодействие пользователей в IP-сетях. Стек является независимым от физической среды передачи данных, благодаря чему, в частности, обеспечивается полностью прозрачное взаимодействие между проводными и беспроводными сетями.

Распределение протоколов по уровням модели TCP/IP

Прикладной (Application layer)	напр., HTTP, RTSP, FTP, DNS
Транспортный (Transport Layer)	напр., TCP, UDP, SCTP, DCCP *(RIP, протоколы маршрутизации, подобные OSPF, что работают поверх IP, являются частью сетевого уровня)*
Сетевой (Межсетевой) (Internet Layer)	Для TCP/IP это IP *(вспомогательные протоколы, вроде ICMP и IGMP, работают поверх IP, но тоже относятся к сетевому уровню; протокол ARP является самостоятельным вспомогательным протоколом, работающим поверх канального уровня)*
Уровень сетевого доступа (Канальный) (Network Access Layer)	Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS, физическая среда и принципы кодирования информации, T1, E1

HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

Протокол передачи данных — набор определённых правил или соглашений интерфейса логического уровня, который определяет обмен данными между различными устройствами. Эти правила задают единообразный способ передачи сообщений и обработки ошибок.

Помимо http протокола, есть еще такой протокол, как **HTTPS**. Отличие этих двух протоколов в том, что наша информация попадает на сервер в каком-то зашифрованном виде, поэтому чаще всего сейчас используют именно https протокол, более надёжная информация, она зашифрована, какие-то злоумышленники не могут ее получить так просто, поэтому обязательно обращайте внимание на это.

Основой HTTP является технология «клиент-сервер». HTTP в настоящее время повсеместно используется во Всемирной паутине для получения информации с веб-сайтов.

Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (в частности, для этого используется HTTP-заголовок).

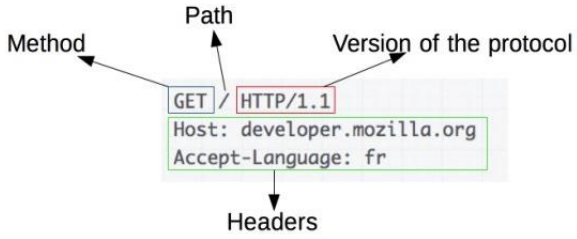
Большинство протоколов предусматривает установление TCP-сессии, в ходе которой один раз происходит авторизация, и дальнейшие действия выполняются в контексте этой авторизации. HTTP же устанавливает отдельную TCP-сессию на каждый запрос; в более поздних версиях HTTP было разрешено делать несколько запросов в ходе одной TCP-сессии, но браузеры обычно запрашивают только страницу и включённые в неё объекты (картинки, каскадные стили и т. п.), а затем сразу разрывают TCP-сессию. Для поддержки авторизованного (неанонимного) доступа в HTTP используются cookies; причём такой способ авторизации позволяет сохранить сессию даже после перезагрузки клиента и сервера.

Структура HTTP-сообщения

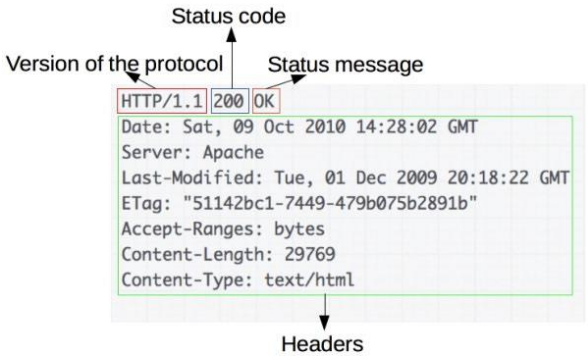
Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке:

1. *Стартовая строка* (англ. Starting line) — определяет тип сообщения;
2. *Заголовки* (англ. Headers) — характеризуют тело сообщения, параметры передачи и прочие сведения;
3. *Тело сообщения* (англ. Message Body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

Тело сообщения может отсутствовать, но стартовая строка и заголовок являются обязательными элементами.



Из чего состоит request: у нас указывается метод, далее идет версия нашего протокола, хост машина, т.е. там, где вообще находится наше приложение, т.е. это наш сервер. Также в request у нас содержатся headers (заголовки), как я уже сказал, это некая служебная информация, которая характеризует нашу основную часть. Она как может быть, так может и не быть, они могут отличаться в каждом отдельном request. Также в первой строке может содержаться URL, т.е. то, к какому ресурсу мы обращаемся.



Если мы говорим с вами про http response, то здесь у нас также содержится информация о версии протокола, также содержится status code, который говорит об успешности того, как отвечает нам сервер. Каждый status code имеет цифровое обозначение и текстовую информацию. Дальше у нас содержится дата, когда вообще был отправлен ответ от сервера, информация о сервере и также какие-то хедеры, которые характеризуют нашу основную нагрузку. Также здесь обычно ставится пустая строка и, если наш ответ содержит в себе какую-то полезную нагрузку, то дальше идет информация именно с этой полезной нагрузкой. Например, документ в формате html.

Метод HTTP (англ. HTTP Method) — последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Обратите внимание, что название метода чувствительно к регистру.

Методы HTTP:

OPTIONS

Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса.

GET

Используется для запроса содержимого указанного ресурса. С помощью метода `GET` можно также начать какой-либо процесс. В этом случае в тело ответного сообщения следует включить информацию о ходе выполнения процесса.

Согласно стандарту HTTP, запросы типа `GET` считаются идемпотентными (Идемпоте́нтность — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом.)

HEAD

Аналогичен методу `GET`, за исключением того, что в ответе сервера отсутствует тело. Запрос `HEAD` обычно применяется для извлечения метаданных, проверки наличия ресурса (валидация URL) и чтобы узнать, не изменился ли он с момента последнего обращения.

Заголовки ответа могут кэшироваться. При несовпадении метаданных ресурса с соответствующей информацией в кэше — копия ресурса помечается как устаревшая.

POST

Применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные (в примере с блогами — текст комментария) включаются в тело запроса. Аналогично с помощью метода `POST` обычно загружаются файлы на сервер.

В отличие от метода `GET`, метод `POST` не считается идемпотентным, то есть многократное повторение одних и тех же запросов `POST` может возвращать разные результаты (например, после каждой отправки комментария будет появляться очередная копия этого комментария).

PUT

Применяется для загрузки содержимого запроса на указанный в запросе URI.

Фундаментальное различие методов `POST` и `PUT` заключается в понимании предназначений URI ресурсов. Метод `POST` предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого. Используя `PUT`, клиент предполагает, что загружаемое содержимое соответствует находящемуся по данному URI ресурсу.

PATCH

Аналогично PUT, но применяется только к фрагменту ресурса.

DELETE

Удаляет указанный ресурс.

Что вам нужно знать еще о методах? Методы отвечают за так называемые CRUD (Create, Read, Update, Delete) операции. Вы будете очень часто встречать в тестировании именно эту аббревиатуру. Всегда, когда мы что-либо тестируем, мы должны рассматривать объект тестирования с точки зрения этих четырех

действий. К примеру, у вас есть какой-то рядовой пользователь, у которого есть права только на чтение, есть модератор, который может изменять какие-то данные в вашей системе, есть еще какой-нибудь super user, который обладает способностью создавать что-то в системе, опять же, читать это, апдейтить, и, например, есть администратор, который может выполнять все эти четыре действия одновременно.

Коды состояния

Информационные.

В этот класс выделены коды, информирующие о процессе передачи. Сами сообщения от сервера содержат только стартовую строку ответа и, если требуется, несколько специфичных для ответа полей заголовка.

100 Continue — сервер удовлетворён начальными сведениями о запросе, клиент может продолжать пересылать заголовки.

102 Processing — запрос принят, но на его обработку понадобится длительное время. Используется сервером, чтобы клиент не разорвал соединение из-за превышения времени ожидания.

103 Early Hints — используется для раннего возврата части заголовков, когда заголовки полного ответа не могут быть быстро сформированы.

Успех

Сообщения данного класса информируют о случаях успешного принятия и обработки запроса клиента. В зависимости от статуса сервер может ещё передать заголовки и тело сообщения.

200 OK — успешный запрос. Если клиентом были запрошены какие-либо данные, то они находятся в заголовке и/или теле сообщения.

201 Created — в результате успешного выполнения запроса был создан новый ресурс. При обработке запроса новый ресурс должен быть создан до отправки ответа клиенту, иначе следует использовать ответ с кодом `202`.

202 Accepted — запрос был принят на обработку, но она не завершена. Клиенту не обязательно дожидаться окончательной передачи сообщения, так как может быть начат очень долгий процесс.

203 Non-Authoritative Information — аналогично ответу `200`, но в этом случае передаваемая информация была взята не из первичного источника (резервной копии, другого сервера и т. д.) и поэтому может быть неактуальной.

204 No Content — сервер успешно обработал запрос, но в ответе были переданы только заголовки без тела сообщения.

Перенаправление

Коды этого класса сообщают клиенту, что для успешного выполнения операции необходимо сделать другой запрос, как правило, по другому URI. Адрес, по которому клиенту следует произвести запрос, сервер указывает в заголовке `Location`.

300 Multiple Choices — по указанному URI существует несколько вариантов предоставления ресурса по типу MIME, по языку или по другим характеристикам. Сервер передаёт с сообщением список альтернатив, давая возможность сделать выбор клиенту автоматически или пользователю.

307 Temporary Redirect — запрашиваемый ресурс на короткое время доступен по другому URI, указанный в поле `Location` заголовка. Метод запроса (GET/POST) менять не разрешается.

308 Permanent Redirect — запрашиваемый ресурс был окончательно перенесен на новый URI, указанный в поле `Location` заголовка. Метод запроса (GET/POST) менять не разрешается.

303 See Other — документ по запрошенному URI нужно запросить по адресу в поле `Location` заголовка с использованием метода `GET` несмотря даже на то, что первый запрашивался иным методом. Например, на веб-странице есть поле ввода текста для быстрого перехода и поиска. После ввода данных браузер делает запрос методом `POST`, включая в тело сообщения введённый текст. Если обнаружен документ с введённым названием, то сервер отвечает кодом `303`, указав в заголовке `Location` его постоянный адрес. Тогда браузер гарантировано его запросит методом `GET` для получения содержимого. В противном случае сервер просто вернёт клиенту страницу с результатами поиска.

305 Use Proxy — запрос к запрашиваемому ресурсу должен осуществляться через прокси-сервер, URI которого указан в поле `Location` заголовка.

Ошибка клиента

Класс кодов `4xx` предназначен для указания ошибок со стороны клиента.

400 Bad Request — сервер обнаружил в запросе клиента синтаксическую ошибку.

401 Unauthorized — для доступа к запрашиваемому ресурсу требуется аутентификация.

403 Forbidden — сервер понял запрос, но он отказывается его выполнять из-за ограничений в доступе для клиента к указанному ресурсу. Иными словами, клиент не уполномочен совершать операции с запрошенным ресурсом.

404 Not Found — самая распространённая ошибка при пользовании Интернетом, основная причина — ошибка в написании адреса Web-страницы. Сервер понял запрос, но не нашёл соответствующего ресурса по указанному URL. Если серверу известно, что по этому адресу был документ, то ему желательно использовать код 410. Ответ `404` может использоваться вместо `403`, если требуется тщательно скрыть от посторонних глаз определённые ресурсы.

410 Gone — такой ответ сервер посылает, если ресурс раньше был по указанному URL, но был удалён и теперь недоступен. Серверу в этом случае неизвестно и местоположение альтернативного документа (например копии).

405 Method Not Allowed — указанный клиентом метод нельзя применить к текущему ресурсу. В ответе сервер должен указать доступные методы в заголовке `Allow`, разделив их запятой. Эту ошибку сервер должен возвращать, если метод ему известен, но он не применим именно к указанному в запросе ресурсу, если же указанный метод не применим на всём сервере, то клиенту нужно вернуть код `501`.

Ошибка сервера

Коды `5xx` выделены под случаи необработанных исключений при выполнении операций на стороне сервера.

500 Internal Server Error — любая внутренняя ошибка сервера, которая не входит в рамки остальных ошибок класса.

501 Not Implemented — сервер не поддерживает возможностей, необходимых для обработки запроса. Типичный ответ для случаев, когда сервер не понимает указанный в запросе метод. Если же метод серверу известен, но он не применим к данному ресурсу, то нужно вернуть ответ `405`.

502 Bad Gateway — сервер, выступая в роли шлюза или прокси-сервера, получил недействительное ответное сообщение от вышестоящего сервера.

503 Service Unavailable — сервер временно не имеет возможности обрабатывать запросы по техническим причинам.

504 Gateway Timeout — сервер в роли шлюза или прокси-сервера не дождался ответа от вышестоящего сервера для завершения текущего запроса.

URI (Uniform Resource Identifier) — унифицированный (единообразный) идентификатор ресурса. URI — символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Прежде всего, речь идёт о ресурсах сети Интернет и Всемирной паутины.

URL (Uniform Resource Locator) - это уникальный адрес сайта в сети, который определяет его местонахождение в сети интернет.

URN (Uniform Resource Name) - это неизменяемая последовательность символов, определяющая только имя некоторого ресурса. Это имя определяет только название самого ресурса, но не говорит как к нему подключаться.

URL — это URI, который, помимо идентификации ресурса, предоставляет ещё и информацию о местонахождении этого ресурса. A URN — это URI, который только идентифицирует ресурс в определённом пространстве имён (и, соответственно, в определённом контексте), но не указывает его местонахождение. Например, URN urn:ISBN:0-395-36341-1 — это URI, который указывает на ресурс (книгу) 0-395-36341-1 в пространстве имён ISBN, но, в отличие от URL, URN не указывает на местонахождение этого ресурса: в нём не сказано, в каком магазине её можно купить или на каком сайте скачать.

URL состоит из различных частей, некоторые из которых являются обязательными, а некоторые - факультативными. Рассмотрим наиболее важные части на примере:

http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument

http:// это протокол. Он отображает, какой протокол браузер должен использовать.

www.example.com это доменное имя. В качестве альтернативы может быть использован и IP-адрес.

:80 это порт. Он отображает технический параметр, используемый для доступа к ресурсам. Порт - это факультативная составная часть URL.

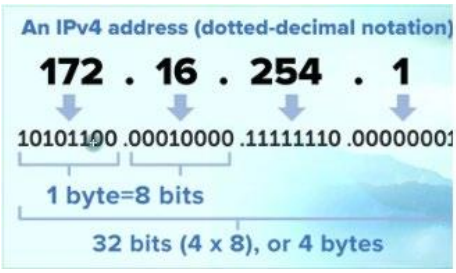
/path/to/myfile.html это адрес ресурса.

?key1=value1&key2=value2 это дополнительные параметры, которые браузер сообщает серверу. Эти параметры - список пар ключ/значение, которые разделены символом &. Сервер может использовать эти параметры для исполнения дополнительных команд, перед тем как отдать ресурс.

#SomewhereInTheDocument это якорь на другую часть того же самого ресурса. Якорь представляет собой вид "закладки" внутри ресурса, которая переадресовывает браузер на "заложенную" часть ресурса. Важно отметить, что часть URL после #, никогда не посылается на сервер вместе с запросом.

IP-адрес (Internet Protocol) — уникальный числовой идентификатор устройства в компьютерной сети, работающей по протоколу IP.

В сети Интернет требуется глобальная уникальность адреса; в случае работы в локальной сети требуется уникальность адреса в пределах сети. В версии протокола IPv4 IP-адрес имеет длину 4 байта, а в версии протокола IPv6 — 16 байт.



Белые и серые IP-адреса

Каждое устройство в сети имеет свой уникальный IP-адрес. Он нужен для того, чтобы устройства сети понимали куда необходимо направить запрос и ответ. Это также как и наши дома и квартиры имеют свой точный адрес (индекс, город, улица, № дома, № квартиры).

В рамках вашей локальной сети (квартиры, офиса или здания) есть свой диапазон уникальных адресов. Я думаю многие замечали, что ip-адрес компьютера, например, начинается с цифр 192.168.X.X. Так вот это локальный адрес вашего устройства.

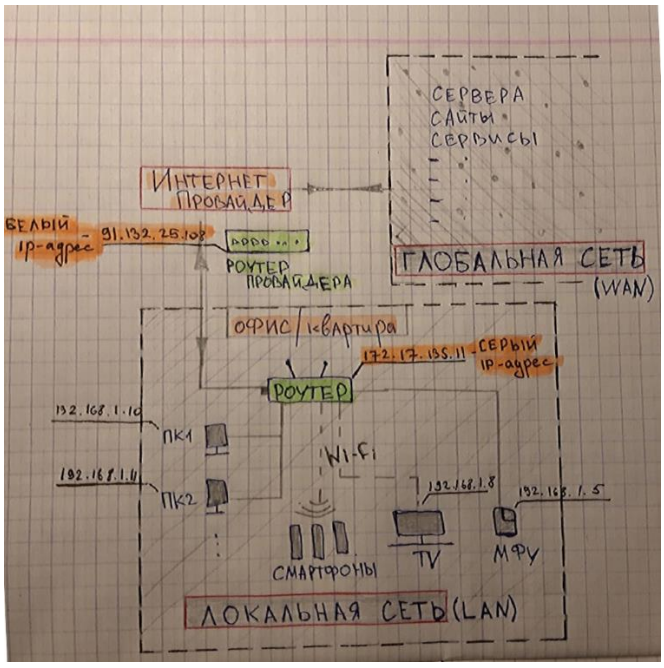
Существуют разрешенные диапазоны локальных сетей:

с IP-адреса	до IP-адреса	Количество возможных подключенных уст-в
10.0.0.0	10.255.255.255	16777216
172.16.0.0	172.31.255.255	65536
192.168.0.0 192.168.1.0	192.168.0.255 192.168.1.255	256 256

Для выхода в глобальные сети, ваш локальный ip-адрес подменяется роутером на глобальный, который вам выдал провайдер. Глобальные ip-адреса не попадают под диапазоны из таблички выше.

Так вот локальные ip-адреса — это серые ip-адреса, а глобальные — это белые.

Для большего понимания рассмотрите схему ниже. На ней я подписал каждое устройство своим ip-адресом.



На схеме видно, что провайдер выпускает нас в глобальные сети (в интернет) с белого ip-адреса 91.132.25.108

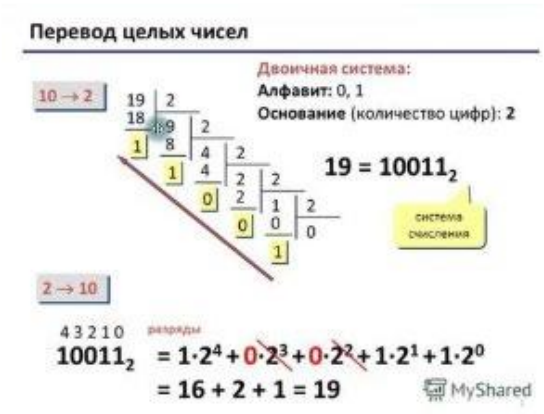
Для нашего роутера провайдер выдал серый ip-адрес 172.17.135.11
И в нашей локальной сети все устройства соответственно тоже имеют серые ip-адреса 192.168.X.X

Но из всего этого стоит помнить один очень важный фактор!

В настоящее время обострилась проблема нехватки белых ip-адресов, так как число сетевых устройств давно превысило количество доступных ip. И по этой причине интернет-провайдеры выдают пользователям серые ip-адреса (в рамках локальной сети провайдера, например в пределах нескольких многоквартирных домов) и выпускают в глобальную сеть под одним общим белым ip-адресом.

Чтобы узнать серый ip-адрес выдает вам провайдер или белый, можно зайти к себе на роутер и посмотреть там, какой ip-адрес получает ваш роутер от провайдера. Для подключения белого ip-адреса провайдеры обычно предоставляют доп. услугу с абон. платой.

Иногда на собеседованиях могут спросить - как переводить ip-адреса из десятичной системы счисления в двоичную и наоборот.



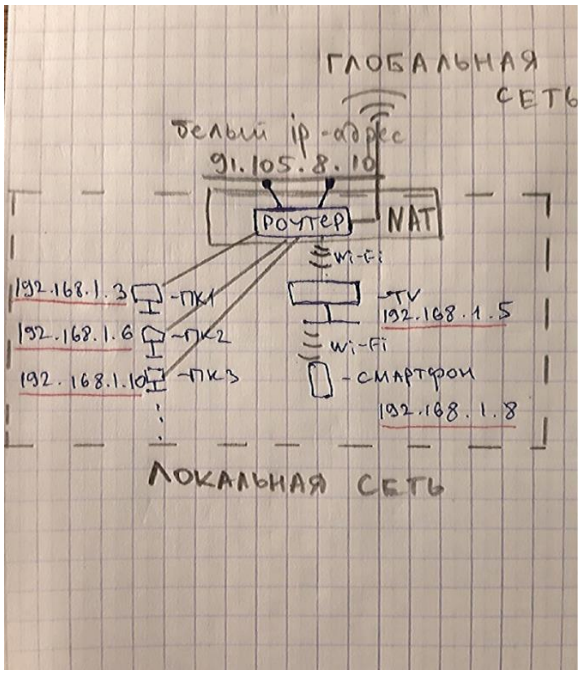
NAT

Изначально всем выдавались белые ip-адреса, и вскоре, чтобы избежать проблему дефицита белых ip-адресов, как раз и был придуман NAT (Network Address Translation) — механизм преобразования ip-адресов.

NAT работает на всех роутерах и позволяет нам из локальной сети выходить в глобальную.

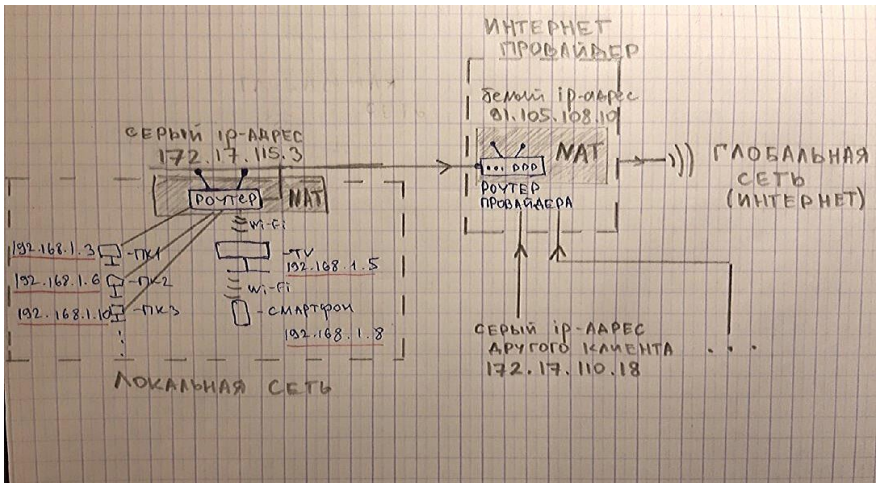
Для лучшего понимания разберем два примера:

1. Первый случай: у вас куплен белый ip-адрес 91.105.8.10 и в локальной сети подключено несколько устройств.



Каждое локальное устройство имеет свой серый ip-адрес. Но выход в интернет возможен только с белого ip-адреса. Следовательно когда, например, ПК1 с ip-адресом 192.168.1.3 решил зайти в поисковик Яндекса, то роутер, выпуская запрос ПК1 в глобальную сеть, подключает механизм NAT, который преобразует ip-адрес ПК1 в белый глобальный ip-адрес 91.105.8.10. Также и в обратную сторону, когда роутер получит от сервера Яндекса ответ, он с помощью механизма NAT направит этот ответ на ip-адрес 192.168.1.3, по которому подключен ПК1.

2. Второй случай: у вас также в локальной сети подключено несколько устройств, но вы не покупали белый ip-адрес у интернет-провайдера.

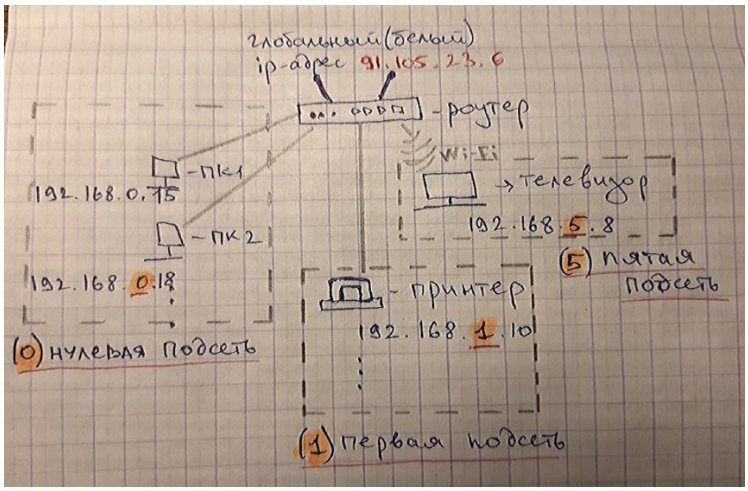


В этом случае локальный адрес ПК1(192.168.1.3) сначала преобразуется NAT'ом вашего роутера и превращается в серый ip-адрес 172.17.115.3, который вам выдал интернет-провайдер, а далее ваш серый ip-адрес преобразуется NAT'ом роутера провайдера в белый ip-адрес 91.105.108.10, и только после этого осуществляется выход в интернет (глобальную сеть). То есть, в этом случае получается, что ваши устройства находятся за двойным NAT'ом.

DHCP (Dynamic Host Configuration Protocol — протокол динамической настройки узла) — прикладной протокол, позволяющий сетевым устройствам автоматически получать IP-адрес и другие параметры, необходимые для работы в сети.

Каждый роутер оснащен DHCP-сервером. IP-адреса, полученные автоматически являются динамическими ip-адресами. Почему динамические? Потому что, при каждом новом подключении или перезагрузки роутера, DHCP-сервер тоже перезагружается и может выдать устройствам разные ip-адреса. То есть, например, сейчас у вашего компьютера ip-адрес 192.168.1.10, после перезагрузки роутера ip-адрес компьютера может стать 192.168.1.35. Чтобы ip-адрес не менялся, его можно задать статически. Это можно сделать, как на компьютере в настройках сети, так и на самом роутере.

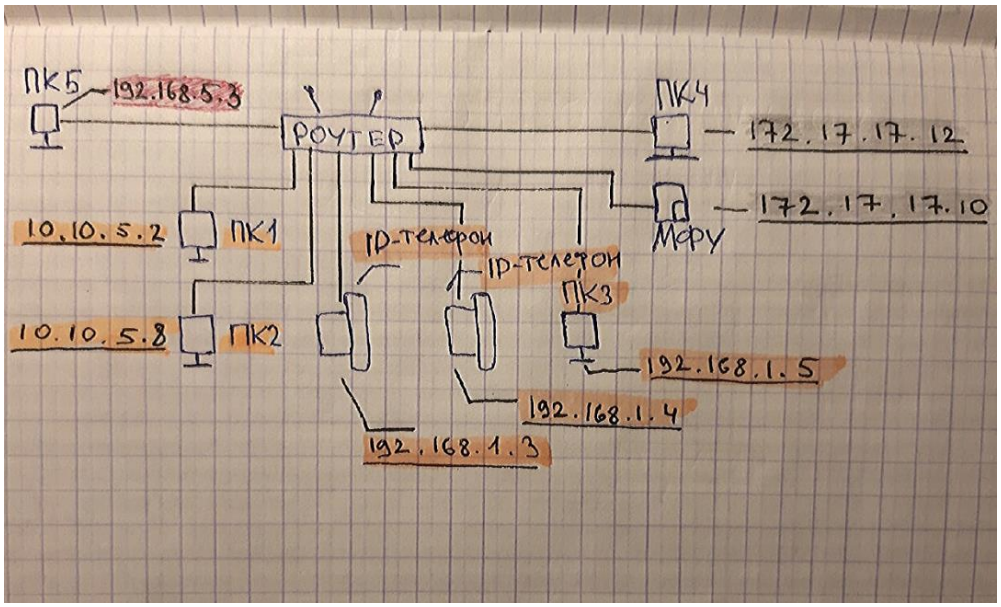
DHCP-сервер на роутере вообще можно отключить и задавать ip-адреса вручную. Можно настроить несколько DHCP-серверов на одном роутере. Тогда локальная сеть разделится на подсети. Например, компьютеры подключим к нулевой подсети в диапазон 192.168.0.2-192.168.0.255, принтеры к первой подсети в диапазон 192.168.1.2-192.168.1.255, а Wi-Fi будем раздавать на пятую подсеть с диапазоном 192.168.5.2-192.168.5.255



Обычно, разграничение по подсетям производить нет необходимости. Это делают, когда в компании большое количество устройств, подключаемых к сети и при настройке сетевой безопасности. Но такая схема в компаниях встречается довольно часто.

Внимание!

Если вам необходимо с ПК зайти на web-интерфейс, например, принтера или ip-телефона и при этом ваш ПК находится в другой подсети, то подключиться не получится. Для понимания разберем пример:



Допустим вы работаете за ПК1 с локальным ip-адресом 10.10.5.2 и хотите зайти на web-интерфейс ip-телефона с локальным ip-адресом 192.168.1.3, то подключиться не получится. Так как устройства находятся в разных подсетях. К ip-телефона, находящиеся в подсети 192.168.1.X, можно подключиться только с ПК3 (192.168.1.5). Также и к МФУ (172.17.17.10) вы сможете подключиться только с ПК4 (172.17.17.12).

Поэтому, когда подключаетесь удаленно к пользователю на ПК, чтобы зайти на web-интерфейс ip-телефона, то обязательно сначала сверяйте их локальные ip-адреса, чтобы убедиться, что оба устройства подключены к одной подсети.

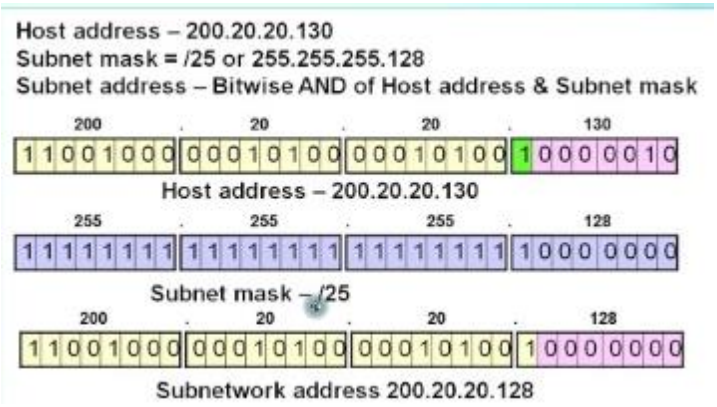
Маска подсети — битовая маска для определения по IP-адресу адреса подсети и адреса узла (хоста, компьютера, устройства) этой подсети.

Когда ip-адрес присваивается интерфейсу, например, сетевому адаптеру компьютера или маршрутизатора, то кроме самого адреса данного устройства (ip-адреса), ему назначают еще и маску подсети.

Компьютерная маска подсети нужна для определения границ самой подсети, чтобы каждый мог определить, кто находится с ним в одной сети подсети, а кто за ее пределами. Маска подсети, по сути, это тоже 32 бита, но, в отличие от ip-адреса, нули и единицы в ней не могут чередоваться. Всегда сначала идут несколько единиц, потом несколько нулей.

Например, если мы с вами посмотрим на маску подсети (subnet mask), то, как увидите, у нас сначала прописаны только единицы, а потом 0. Т.е. такого, как в ip-адресе у нас быть не может, когда есть чередование. Я думаю, что после рассмотрения ip-адреса вам стало понятно, что такая запись в двоичной системе исчисления не очень удобна, поэтому используют упрощенную версию. Прописывают просто количество единиц, которые у нас есть в маске подсети, т.е. здесь у нас их 25, поэтому после ip-адреса могут поставить “/25”. Думаю, что те, кто когда-либо настраивал браузер, либо же подключение к сети в системе windows, то вы обращали внимание на ip-адреса и на то, как там прописывается маска подсети. Т.е. иногда можно встретить такую запись через “/” и прописано количество вот этих самых единиц, которые у нас содержатся в маске подсети. Но также вы могли встретить и другую запись, когда у нас не прописывается через “/” эта маска подсети, а вообще прописана граница нашей подсети вот в таком вот виде. Здесь уже учтена и маска подсети, и сам ip-адрес.

Как вообще происходит определение границ подсети? Берется наш ip-адрес, берется маска подсети и каждая цифра, которая содержится в этих адресах, перемножается друг на друга. В итоге у нас получается вот такая вот запись, которую в дальнейшем также переводят в десятичную систему счисления для удобства записи.



MAC-адрес (Media Access Control — надзор за доступом к среде) — уникальный идентификатор, присваиваемый каждой единице активного оборудования или некоторым их интерфейсам в компьютерных сетях Ethernet.

При проектировании стандарта Ethernet было предусмотрено, что каждая сетевая карта (равно как и встроенный сетевой интерфейс) должна иметь уникальный шестибайтный номер (MAC-адрес), «прошитый» в ней при изготовлении. Этот номер используется для идентификации отправителя и получателя фрейма; и предполагается, что при появлении в сети нового компьютера (или другого устройства, способного работать в сети) сетевому администратору не придётся настраивать этому компьютеру MAC-адрес вручную.

Уникальность MAC-адресов достигается тем, что каждый производитель получает в координирующем комитете IEEE Registration Authority диапазон из 16 777 216 (2^{24}) адресов и, по мере исчерпания выделенных адресов, может запросить новый диапазон. Поэтому по трём старшим байтам MAC-адреса можно определить производителя. Существуют таблицы, позволяющие определить производителя по MAC-адресу

MAC-адрес - это физический адрес нашего устройства, он прописывается при производстве сетевой карты. Т.е. если у вас на собеседованиях спросят сколько вообще mac-адресов есть системе, то вы всегда должны сказать, что столько же, сколько сетевых карт у нас установлено в оборудовании, которое к этой системе принадлежит. Т.е. если у нас есть сетевая карта в нашем принтере, то она тоже будет с mac-адресом и учитываться в этом общем количестве. Т.е. не всегда верно говорить так, что если у вас, например, есть сетевая карта в компьютере, которая отвечает за наше подключение через вай-фай, то она одна. Всегда считаем все устройства, которые подключены к компьютеру и у которых есть mac-адрес, т.е. они могут выходить в сеть.

DNS (Domain Name System) — компьютерная распределённая система для получения информации о доменах. Для преобразования доменного имени в IP-адрес и наоборот служит система DNS.

Смысл её в том, что каждому цифровому IP-адресу присваивается понятное буквенное имя, либо же его еще называют домен. Т.е. когда вы вводите в браузере доменное имя, вот эти сервера DNS преобразуют его в IP-адрес. Т.е. у каждого нашего домена, у каждого нашего URL, есть соответствующий IP-адрес сервера. С помощью DNS эта информация в буквенном значении преобразуется в числовое, и уже севера могут понимать что от них хотят и какую информацию хотят получить. Поэтому мы получаем уже информацию от серверов в этом буквенном виде с помощью преобразования в DNS.

DNS важна для работы Интернета, так как для соединения с узлом необходима информация о его IP-адресе, а для людей проще запоминать буквенные (обычно осмысленные) адреса, чем последовательность цифр IP-адреса.

Что же такое **кэш браузера** и для чего он вообще необходим? Когда вы серфите на просторах интернета, загружаете, просматриваете веб-страницы, ваш браузер автоматически сохраняет определенные данные на жесткий диск вашего компьютера. Это могут быть различные элементы дизайна сайта - картинка, изображение, видеофайлы, музыка и так далее. Это делается для того, чтобы при следующем обращении к странице она загрузилась быстрее благодаря тому, что часть информации загружается уже не с сервера, а непосредственно с вашего жесткого диска. Это и есть наш кэш, т.е. это данные, которые загружаются с нашего компьютера для того, чтобы упростить нам работу на тех сайтах, которые мы раньше посещали.

Cookies — небольшой фрагмент данных, отправленный сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя, на практике обычно используется для:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сеанса доступа пользователя;
- сведения статистики о пользователях.

Cookie используются веб-серверами для идентификации пользователей и хранения данных о них. К примеру, если вход на сайт осуществляется при помощи cookie, то, после ввода пользователем своих данных на странице входа, cookie позволяют серверу запомнить, что пользователь уже идентифицирован и ему разрешён доступ к соответствующим услугам и операциям.

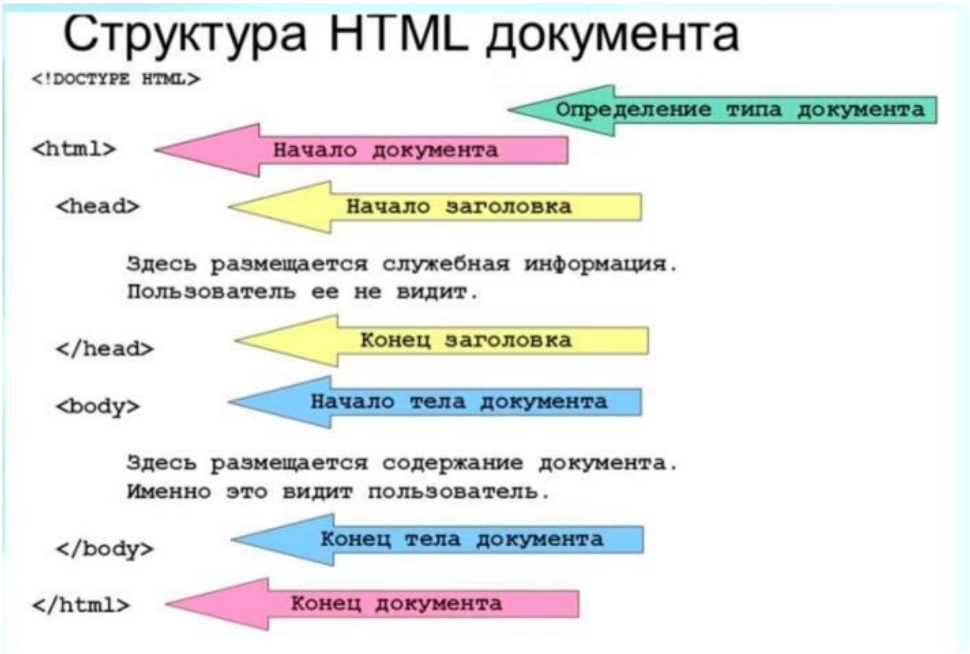
Многие сайты также используют cookie для сохранения настроек пользователя. Эти настройки могут использоваться для персонализации, которая включает в себя выбор оформления и функциональности. Например, Википедия позволяет авторизованным пользователям выбрать дизайн сайта. Поисковая система Google позволяет пользователям (в том числе и не зарегистрированным в ней) выбрать количество результатов поиска, отображаемых на одной странице.

Cookie также используются для отслеживания действий пользователей на сайте. Как правило, это делается с целью сбора статистики, а рекламные компании на основе такой статистики формируют анонимные профили пользователей для более точного нацеливания рекламы.

Большинство современных браузеров позволяет пользователям выбрать — принимать cookie или нет, но их отключение делает невозможной работу с некоторыми сайтами. Кроме того, по законам некоторых стран (например, согласно постановлению Евросоюза от 2016 года) сайты должны в обязательном порядке запрашивать согласие пользователя перед установкой cookie.

HTML (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартизированный язык разметки документов для просмотра веб-страниц в браузере. Веб-браузеры получают HTML документ от сервера по протоколам HTTP/HTTPS или открывают с локального диска.

Текстовые документы, содержащие разметку на языке HTML (такие документы традиционно имеют расширение .html или .htm), обрабатываются специальными приложениями (веб-браузер), которые отображают документ в его форматированном виде.



С помощью HTML разные конструкции, изображения и другие объекты могут быть встроены в отображаемую страницу. HTML предоставляет средства для создания заголовков, абзацев, списков, ссылок, цитат и других элементов.

Единица информации в HTML - это тэг, который обрамлен в скобки, <tag>.

<a> - это тэг гиперссылки, т.е. мы здесь можем прописывать гиперссылки. - говорит о том, что наш текст должен быть написан жирным шрифтом. И у таких тегов должна быть закрывающая часть со слешем, бла бла бла. Между нашими двумя тэгами содержится какая-то информация.

Но также есть тэги, которые не нужно закрывать. Например, к таким можно отнести тэг <input/>, тэг <img/>, т.е., как вы видите, необходимо просто поставить перед второй скобкой закрывающий слэш.

В чем же отличие таких вот тэгов с закрывающим элементом и тэгов без него? В том, что если между этими тэгами заключается какая-то информация, например, если мы говорим о тэгах , которые говорят о том, что шрифт наш жирный, то здесь может быть какой-то текст бла бла бла, и этот текст будет отображаться на странице жирным шрифтом.

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>HTML Document</title>
   </head>
   <body>
      <p>
         <b>
            Этот текст будет полужирным, <i>а этот — ещё и курсивным</i>.
         </b>
      </p>
   </body>
</html>
```

даст такой результат:

Этот текст будет полужирным, *а этот — ещё и курсивным*.

<input/>, например, говорит о том, что это поле ввода, т.е. говорит о том, что у нас на странице отображается поле для ввода. Если мы говорим про <img/>, то тогда необходимо добавить какой-то атрибут со ссылкой на нашу картинку.

Пример

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Тег INPUT</title>
 </head>
 <body>

 <form name="test" method="post" action="input1.php">
  <p><b>Ваше имя:</b><br>
   <input type="text" size="40">
  </p>
  <p><b>Каким браузером в основном пользуетесь:</b><Br>
   <input type="radio" name="browser" value="ie"> Internet Explorer<Br>
   <input type="radio" name="browser" value="opera"> Opera<Br>
   <input type="radio" name="browser" value="firefox"> Firefox<Br>
  </p>
  <p>Комментарий<Br>
   <textarea name="comment" cols="40" rows="3"></textarea></p>
  <p><input type="submit" value="Отправить">
   <input type="reset" value="Очистить"></p>
 </form>

 </body>
</html>
```

Ваше имя:

Каким браузером в основном пользуетесь:

- Internet Explorer
- Opera
- Firefox

Комментарий

Отправить Очистить

Атрибут используется для того, чтобы уточнить информацию. Например, расположение нашей картинки. Если мы говорим про атрибуты тэга гиперссылки <a>, т.е. такой атрибут Яндекс. На странице, которую мы верстаем отобразится вот этот Яндекс в виде гиперссылки, т.е. он будет синим шрифтом подчеркнут, и при нажатии на эту гиперссылку, которая у нас ya.ru, мы перейдем на этот сайт. Есть такой атрибут как src, это уже ссылка на нашу картинку, т.е. информация о расположении данной картинки либо же в нашем локальном сервере, либо же в сети интернет. <img src=”c/img/1.png”>, тогда на наш сайт ставится картинка, которая находится по вот поэтому расположению.

Кроме элементов, в HTML-документах есть и *сущности* (англ. entities) — «специальные символы». Сущности начинаются с символа амперсанда и имеют вид `&имя;` или `&#NNNN;`, где NNNN — код символа в Юникоде в десятичной системе счисления.

Например, `&copy;` — знак авторского права (©). Как правило, сущности используются для представления символов, отсутствующих в кодировке документа, или же для представления «специальных» символов: `&` — амперсанда (&), `<` — символа «меньше» (<) и `>` — символа «больше» (>), которые некорректно записывать «обычным» образом, из-за их особого значения в HTML.

CSS (Cascading Style Sheets «каскадные таблицы стилей») — формальный язык описания внешнего вида документа (веб-страницы).

CSS используется создателями веб-страниц для задания цветов, шрифтов, стилей, расположения отдельных блоков и других аспектов представления внешнего вида этих веб-страниц. Кроме того, CSS позволяет представлять один и тот же документ в различных стилях или методах вывода, таких как экранное представление, печатное представление, чтение голосом (специальным голосовым браузером или программой чтения с экрана).

До появления CSS оформление веб-страниц осуществлялось исключительно средствами HTML, непосредственно внутри содержимого документа. Однако с появлением CSS стало возможным принципиальное разделение содержания и представления документа.

CSS-вёрстка

Преимущества:

- Несколько дизайнов страницы для разных устройств просмотра. Например, на экране дизайн будет рассчитан на большую ширину, во время печати меню не будет выводиться, а на КПК и сотовом телефоне меню будет следовать за содержимым.
- Уменьшение времени загрузки страниц сайта за счёт переноса правил представления данных в отдельный CSS-файл. В этом случае браузер загружает только структуру документа и данные, хранимые на странице, а представление этих данных загружается браузером только один раз и может быть закэшировано.
- Простота последующего изменения дизайна. Не нужно править каждую страницу, а достаточно лишь изменить CSS-файл.

Недостатки:

- Различное отображение вёрстки в различных браузерах (особенно устаревших), которые по-разному интерпретируют одни и те же данные CSS.

Способы подключения CSS к документу

Правила CSS могут располагаться как в самом веб-документе, внешний вид которого они описывают, так и во внешних файлах, имеющих расширение `.css`. Формат CSS — это текстовый файл, в котором содержится перечень правил CSS и комментариев к ним.

Стили CSS могут быть подключены или внедрены в описываемый ими веб-документ четырьмя способами:

- когда описание стилей находится в отдельном файле, оно может быть подключено к документу посредством элемента `<link>`, включённого в элемент `<head>`:

```
<!DOCTYPE html>
<html>
   <head>
      .....
     <link rel="stylesheet" type="text/css" href="style.css">
   </head>
   <body>
      .....
   </body>
</html>
```

- когда файл стилей размещается отдельно от родительского документа, он может быть подключён к документу инструкцией @import в элементе `<style>`:

```
<!DOCTYPE html>
<html>
   <head>
      .....
      <style media="all">
         @import url(style.css);
      </style>
   </head>
</html>
```

- когда стили описаны внутри документа, они могут быть включены в элемент <style>, который, включается в элемент <head>:

```
<!DOCTYPE html>
<html>
   <head>
      .....
      <style>
         body {
            color: red;
         }
      </style>
   </head>
   <body>
      .....
   </body>
</html>
```

- когда стили описаны в теле документа, они могут располагаться в атрибутах отдельного элемента:

```
<!DOCTYPE>
<html>
   <head>
      .....
   </head>
   <body>
      <p style="font-size: 20px; color: green; font-family: arial,
helvetica, sans-serif">
         .....
      </p>
   </body>
</html>
```

В первых двух случаях к документу применены внешние стили, в остальных — внутренние стили.

Правила построения CSS

В первых трёх случаях подключения стилей CSS к документу каждое правило CSS из файла имеет две основные части — селектор и блок объявлений. Селектор, расположенный в левой части правила до знака «{», определяет, на какие части документа (возможно, специально обозначенные) распространяется правило. Блок объявлений располагается в правой части правила. Он помещается в фигурные скобки, и, в свою очередь, состоит из одного или более объявлений, разделённых знаком «;». Каждое объявление представляет собой сочетание свойства CSS и значения, разделённых знаком «:». Селекторы могут группироваться в одной строке через запятую. В таком случае свойство применяется к каждому из них.

```
селектор, селектор {
  свойство: значение;
  свойство: значение;
  свойство: значение;
}
```

В четвёртом случае подключения CSS к документу правило CSS, являющееся значением атрибута style элемента, к которому он применяется, представляет собой перечень объявлений («свойство CSS : значение»), разделённых знаком «;».

API (Application Programming Interface - программный интерфейс приложения) - описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой. Обычно входит в описание какого-либо интернет-протокола, фреймворка или стандарта вызовов функций операционной системы. Часто реализуется отдельной программной библиотекой или сервисом операционной системы. Используется программистами при написании всевозможных приложений.

API как средство интеграции приложений

Если программу (модуль, библиотеку) рассматривать как чёрный ящик, то API — это множество «ручек», которые доступны пользователю данного ящика и которые он может вертеть и дёргать.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

Практически все операционные системы (UNIX, Windows, OS X, Linux и т. д.) имеют API, с помощью которого программисты могут создавать приложения для этой операционной системы. Главный API операционных систем — это множество системных вызовов.

В индустрии программного обеспечения общие стандартные API для стандартной функциональности играют важную роль, так как они гарантируют, что все программы, использующие общий API, будут работать одинаково хорошо или, по крайней мере, типичным привычным образом.

Web API

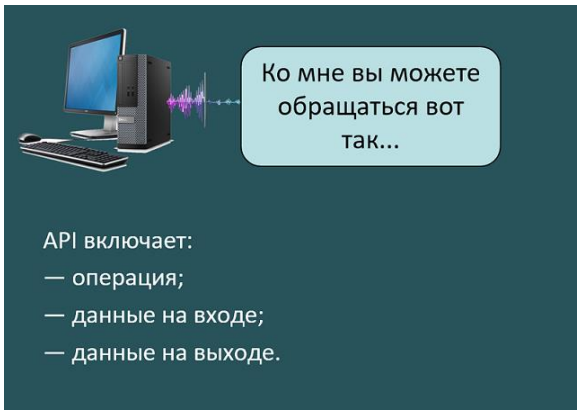
Используется в веб-разработке — содержит, как правило, определённый набор HTTP-запросов, а также определение структуры HTTP-ответов, для выражения которых используют XML− или JSON−формат.

API — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».

Когда вы покупаете машину, вы составляете договор, в котором прописываете все важные для вас пункты. Точно также и между программами должны составляться договоры. Они указывают, как к той или иной программе можно обращаться.

Соответственно, API отвечает на вопрос “Как ко мне, к моей системе можно обратиться?”, и включает в себя:

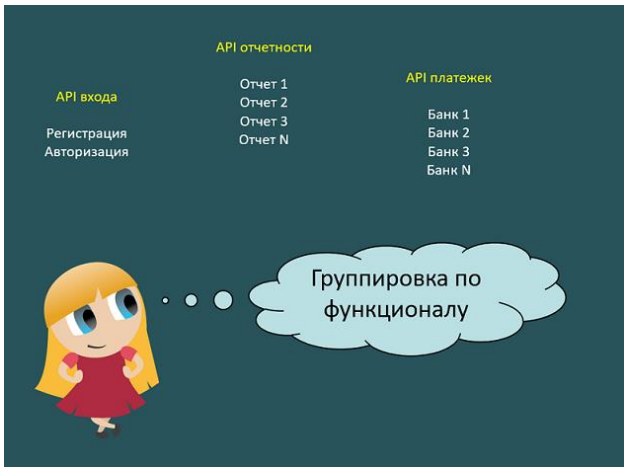
- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).



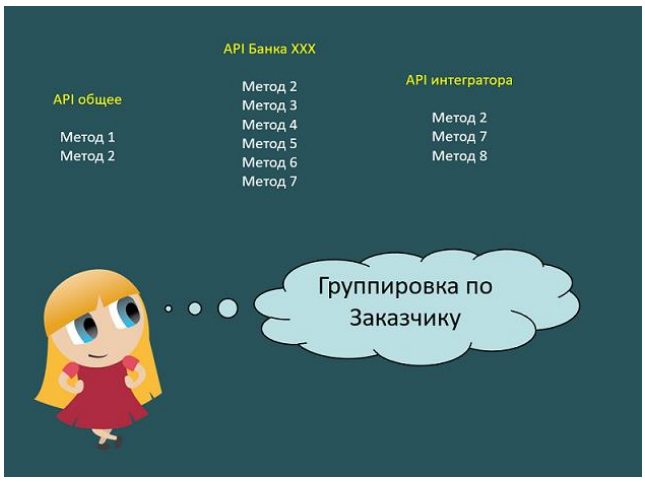
API — это набор функций. Это может быть одна функция, а может быть много.

Как составляется набор функций? Да без разницы как. Как разработчик захочет, так и сгруппирует. Например, можно группировать API по функционалу, то есть:

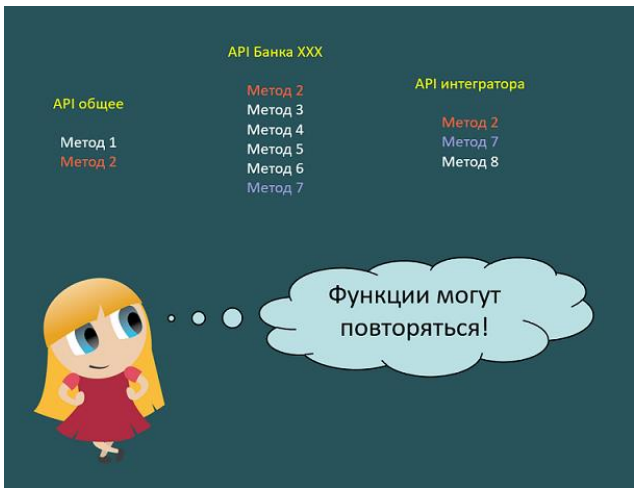
- отдельно API для входа в систему, где будет регистрация и авторизация;
- отдельно API для отчетности — отчет 1, отчет 2, отчет 3… отчет N. Для разных отчетов у нас разные формулы = разные функции. И все мы их собираем в один набор, api для отчетности.
- отдельно API платежек — для работы с каждым банком своя функция.
- ...



Можно не группировать вообще, а делать одно общее API. Можно сделать одно общее API, а остальные «под заказ». Если у вас коробочный продукт, то в него обычно входит набор стандартных функций. А любые хотелки заказчиков выносятся отдельно.



И конечно, функции можно переиспользовать. То есть одну и ту же функцию можно включать в разные наборы, в разные API. Никто этого не запрещает.



Как вызывается API?

- Вызов API напрямую:

1. Система вызывает функции внутри себя

Разные части программы как-то общаются между собой. Они делают это на программном уровне, то есть на уровне API. Это самый «простой» в использовании способ, потому что автор API, которое вызывается — разработчик. В этом случае в качестве документации будут комментарии в коде. А они, увы, тоже бывают неактуальны. Или разработчики разные, или один, но уже забыл, как делал исходное api и как оно должно работать…

2. Система вызывает метод другой системы

А вот это типичный кейс, которые тестируют тестировщики в интеграторах. Одна система дергает через api какой-то метод другой системы. Она может попытаться получить данные из другой системы или, наоборот, отправить данные в эту систему.

Допустим, я решила подключить подсказки из Дадаты к своему интернет-магазинчику, чтобы пользователь легко ввел адрес доставки. Я подключаю подсказки по API. И теперь, когда пользователь начинает вводить адрес на моем сайте, он видит подсказки из Дадаты. Как это получается:

- Он вводит букву на моем сайте
- Мой сайт отправляет запрос в подсказки Дадаты по API
- Дадата возвращает ответ
- Мой сайт его обрабатывает и отображает результат пользователю

И так на каждый введенный символ. Пользователь не видит этого взаимодействия, но оно есть.

3. Человек вызывает метод

Причины разные:

1. Для ускорения работы
2. Для локализации бага (проблема где? На сервере или клиенте?)
3. Для проверки логики без докруток фронта

Если система предоставляет API, обычно проще дернуть его, чем делать то же самое через графический интерфейс. Тем более что вызов API можно сохранить в инструменте. Один раз сохранил — на любой базе применяешь, пусть даже она по 10 раз в день чистится.

4. Автотесты дергают методы

- Косвенный вызов API

Когда пользователь работает с GUI, на самом деле он тоже работает с API. Просто не знает об этом, ему это просто не нужно. То есть когда пользователь открывает систему и пытается загрузить отчет, ему не важно, как работает система, какой там magic внутри. У него есть кнопочка «загрузить отчет», на которую он и нажимает. Но на самом деле под GIU находится API. И когда пользователь нажимает на кнопочку, кнопочка вызывает функцию построения отчета. А функция построения отчета уже может вызывать 10 разных других функций, если ей это необходимо. И вот уже пользователь видит перед собой готовый отчет. Он вызвал сложное API, даже не подозревая об этом!

SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде. SOAP является одним из стандартов, на которых базируются технологии веб-служб.

Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур. Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур.

SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTP, HTTPS и др., однако его взаимодействие с каждым из этих протоколов имеет свои особенности, которые должны быть определены отдельно. Чаще всего SOAP используется поверх HTTP.

Использование SOAP для передачи сообщений увеличивает их объём и снижает скорость обработки. В системах, где скорость важна, чаще используется пересылка XML-документов через HTTP напрямую, где параметры запроса передаются как обычные HTTP-параметры.

XSD (XML Schema Definition) — язык описания структуры XML-документа. XSD был задуман для определения правил, которым должен подчиняться документ. XSD описывает статическую структуру сложных типов данных. Он описывает типы данных, их поля, любые ограничения на эти поля (например, максимальная длина) и т. д.

WSDL (Web Services Description Language) — язык описания веб-сервисов и доступа к ним, основанный на языке XML. WSDL описывает ваш веб-сервис и его операции - как называется веб-сервис, какие методы он предлагает, какие параметры и возвращаемые значения имеют эти методы? Это описание поведения веб-сервиса - его функциональности.

Структура

Каждый документ WSDL можно разбить на следующие логические части:

1. Определение типов данных (types) — определение вида отправляемых и получаемых сервисом XML-сообщений
2. Элементы данных (message) — сообщения, используемые web-сервисом
3. Абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями
4. Связывание сервисов (binding) — способ, которым сообщение будет доставлено

Важно понимать, если присутствует описание типа WSDL, то это SOAP
Пример WSDL:

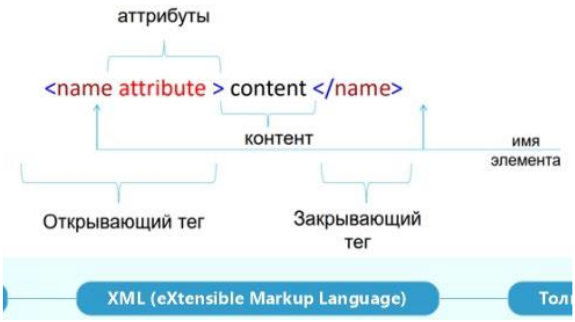
```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
```

Обязательно необходимо запомнить правила, которые используются при написании XML документа. XML похож на html, но у него, как видите, другая расшифровка, т.е. это расширенный язык разметки. Если HTML у нас используется именно для разметки страницы, для верстки, то XML уже хранит в себе некоторую информацию, с помощью которой веб-сервисы могут общаться между собой.

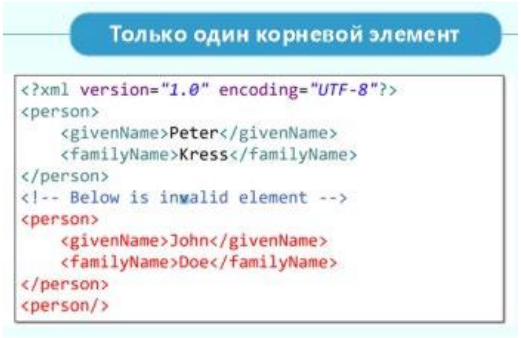
Структурными единицами XML являются элементы. Вот здесь есть схема таких элементов:



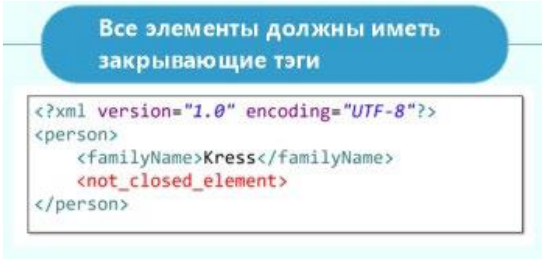
У нас есть открывающий тэг, закрывающий тэг, контент, который хранится между этими двумя тэгами и атрибуты. Всё очень похоже на HTML, однако, что вам необходимо знать при написании, либо же тестировании XML, на что необходимо обращать внимание?

Во-первых, у XML есть только один корневой элемент. Т.е. смотрите, у нас здесь есть корневой элемент <person>, это правильное написание, есть открывающий тэг <person>, закрывающий тэг </person> и больше у нас никаких корневых элементов нет.

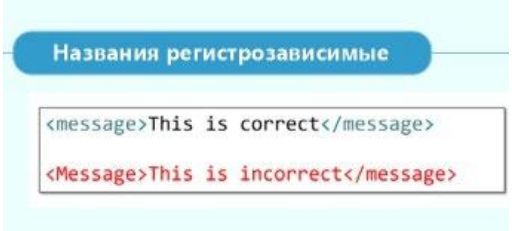
Дальше у нас прописан неправильный формат, когда у нас, как вы видите, есть корневой элемент <person> и он у нас почему-то дублируется. Т.е. такого не должно быть.



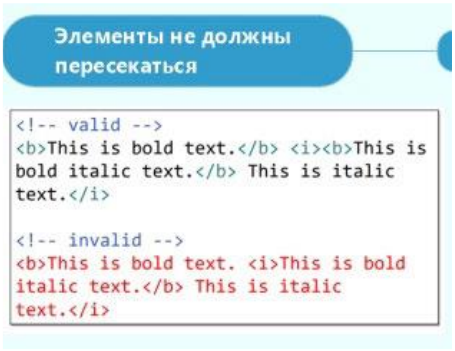
Следующий аспект - все элементы должны иметь закрывающие тэги. Как вы видите, у нас здесь есть два тэга, <person> и <familyname>, они есть как открывающие, так и закрывающие, однако есть один элемент, который не имеет закрывающего тэга (помечен красным), т.е. так неправильно. Мы не должны оставлять просто один открывающий тэг.



Название наших тэгов регистрозависимые. Тэги чувствительны к прописным буквам, либо же к строчным. Всегда должны использоваться одни и те же символы.



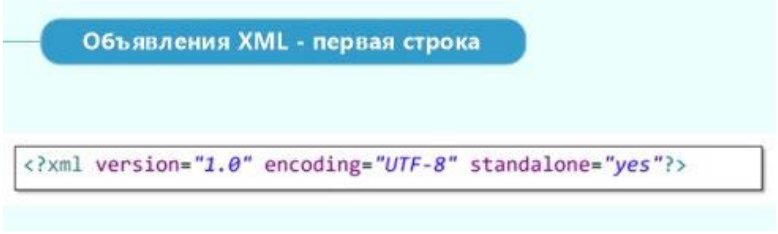
Элементы не должны пересекаться. Если мы внутри тэга ставим другой тэг, то и закрыть его должны внутри первого тэга, а не после него.



Все значения атрибутов должны быть в кавычках.

Знаки < > & нельзя использовать в текстовых блоках. Мы должны записывать их в текстовом формате. < = «<», > = «>», & = «&», ' = «‘», " = «“». Иначе система может их неправильно обработать.

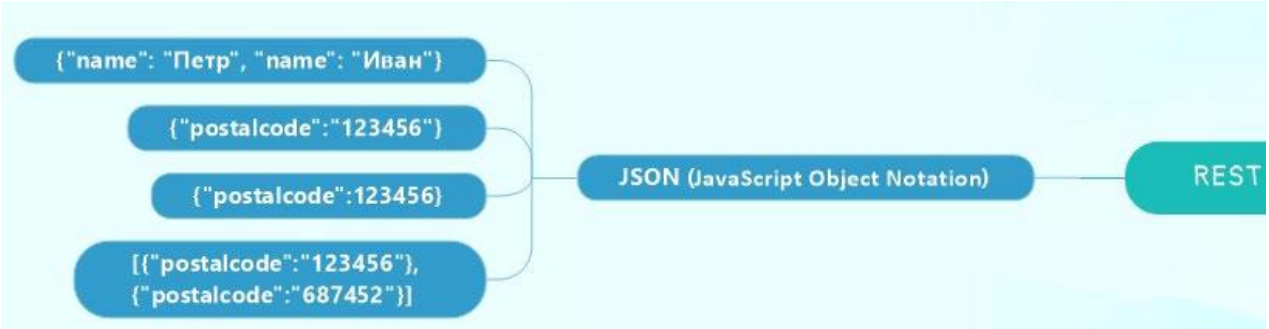
Объявления XML - это всегда первая строка. Это отдельная строка, в которой содержится информация о номере версии нашего xml, указания на кодировку наших символов и параметр standalone, который указывает запрещены ли ссылки на внешние документы.



REST (Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Другими словами, REST - это набор правил о том, как программисту организовать написание кода серверного приложения так, чтобы все системы легко обменивались данными.

В чем разница архитектурного стиля REST и протокола SOAP? К архитектурному стилю не применяются какие-то жёсткие правила, здесь не нужны никакие WSDL, хотя есть тут такой формат документов, как WADL, но не обязательно. Т.е. если его нету, то как бы все окей, и поэтому REST предпочитают сейчас использовать все чаще, так как этот архитектурный стиль позволяет записывать информацию в более удобном формате, который занимает меньше места и повышает производительность нашей системы. В отличие от SOAP, в REST у нас используется уже JSON.

JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. И давайте поговорим об основных элементах, которые у нас содержатся в JSON.



Во-первых, документы в формате JSON состоят из объектов. Объекты - это неупорядоченное множество пар “ключ - значение”. Объект обычно заключается в фигурные скобки и внутри него содержатся, как я уже сказал, пары “ключ - значение”. Ключ “name”, значение “Петр”. Между собой эти пары разделяются запятой, а ключ и значение между собой разделяются “:”.

Ключ - это всегда строка, т.е. тип данных для него - это строка, а само значение, как здесь у нас, например, “Петр”, может быть как строкой в двойных кавычках, так и числом, каким-то логическим значением булевым (true или falls), массивом или значением “Null”.

Если мы говорим о таком типе данных, как строка, то это упорядоченное множество из нуля или более символов юникода, заплетённые в двойные кавычки. Если мы говорим о простых числах, то их не нужно заключать кавычки. Как раз на примере индекса можно заметить разницу, если у нас используется в значении число, то кавычки не записываем, и тогда наша система воспринимает эти данные как числовые. Как только мы с вами используем кавычки для записи нашего числа, то автоматически этот тип данных превращается в строку и воспринимается системой как текст. Аналогично с булевскими значениями true, falls.

И последний тип данных - это массивы. Это множество наших объектов. Как мы с вами помним, объекты - это некоторое количество пар “ключ и значение”. Как только у нас используется несколько таких вот объектов, то тогда мы заключаем их в массивы. Массив заключается в квадратные скобки, а значения внутри этого массива отделяются запятыми.

Следующий пример показывает JSON-представление данных об объекте, описывающем человека. В данных присутствуют строковые поля имени и фамилии, информация об адресе и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address": {
       "streetAddress": "Московское ш., 101, кв.101",
       "city": "Ленинград",
       "postalCode": 101101
   },
   "phoneNumbers": [
       "812 123-1234",
       "916 123-4567"
   ]
}
```

На языке XML подобная структура выглядела бы примерно так:

```
<person>
  <firstName>Иван</firstName>
  <lastName>Иванов</lastName>
  <address>
    <streetAddress>Московское ш., 101, кв.101</streetAddress>
    <city>Ленинград</city>
    <postalCode>101101</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

или так:

```
<person firstName="Иван" lastName="Иванов">
  <address streetAddress="Московское ш., 101, кв.101"
city="Ленинград" postalCode="101101" />
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

Очень часто на собеседованиях спрашивают в чем разница между REST и SOAP веб-сервисами. Давайте перечислим эти различия:

- REST - поддерживает различные форматы. Самый распространенный JSON, но также можно использовать XML и текстовые форматы. Но SOAP - только XML.
- REST работает только по протоколам http и https. SOAP - с различными протоколами.
- SOAP на основе чтения не может быть помещен в кэш. Rest можно кэшировать.
- REST - это архитектурный стиль без строгих правил. SOAP - протокол, который сильно ограничен правилами.

Что необходимо выбирать на проекте, REST или SOAP? Простота против стандарта. REST - у вас будет скорость, расширяемость, поддержка многих форматов. SOAP - у вас будет больше возможностей по безопасности, а также больше возможности провести тестирование качественно, так как сразу в SOAP у вас будут WSDL, где будут прописаны образцы ваших запросов. Т.е. не нужно будет лишний раз обращаться к разработчику, просить его, чтобы он предоставил вам информацию, какие-то примеры запросов. Сразу можно брать и тестировать. SOAP более ресурсоемкий, занимает больше места, он медленнее. Разработка на SOAP дольше.

База данных - это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе.

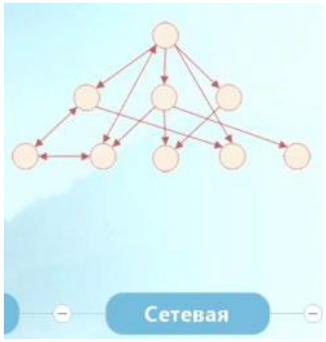
Классификация по модели данных:

1. Иерархическая модель данных — это модель данных, где используется представление базы данных в виде древовидной (иерархической) структуры, состоящей из объектов (данных) различных уровней. Между объектами существуют связи, каждый объект может включать в себя несколько объектов более низкого уровня. Такие объекты находятся в отношении предка (объект более близкий к корню) к потомку (объект более низкого уровня), при этом возможна ситуация, когда объект-предок имеет несколько потомков, тогда как у объекта-потомка обязателен только один предок. Объекты, имеющие общего предка, называются близнецами.

2. Сетевая модель данных — логическая модель данных, являющаяся расширением иерархического подхода. Разница между иерархической моделью данных и сетевой состоит в том, что в иерархических структурах запись-потомок должна иметь в точности одного предка, а в сетевой структуре данных у потомка может иметься любое число предков.

Достоинством сетевой модели данных является возможность эффективной реализации по показателям затрат памяти и оперативности.

Недостатком сетевой модели данных являются высокая сложность и жесткость схемы БД, построенной на её основе. Поскольку логика процедуры выборки данных зависит от физической организации этих данных, то эта модель не является полностью независимой от приложения. Другими словами, если необходимо изменить структуру данных, то нужно изменить и приложение.



3. Реляционные базы данных – это базы данных, которые используются для хранения и предоставления доступа к взаимосвязанным элементам информации. Реляционные базы данных основаны на реляционной модели — интуитивно понятном, табличном способе представления данных.

Реляционная БД используется всеми сейчас. Когда у нас есть некоторое множество таких вот двумерных таблиц, в которых содержатся наши данные, и эти таблицы также могут между собой быть связаны. Есть так называемый первичный ключ (primary key), который является однозначным идентификатором данной строки, и он обязательный. Потому что он уникальный, а если, например, мы возьмем значение “имя” столбца, то мы увидим, что здесь могут быть одинаковые имена, и, в таком случае, наша база данных будет отрабатывать, порой, некорректно, когда мы будем запрашивать какое-то одно имя и у него будет много одинаковых значений, и она будет выдавать нам очень много значений по одному этому запросу, поэтому всегда есть такие идентификаторы, которые уникальны и определяют строки в нашей базе данных. И есть, так называемые, вторичные ключи (foreign key) - служат для того, например, если у нас есть две таблицы, в которых есть связанные данные, то как нам эти таблицы между собой можно связать? С помощью вторичного ключа, т.е. мы вводим какие-то одинаковые идентификаторы для этих двух таблиц, и с помощью этого индикатора мы связываем наши таблицы. Вторичный ключ не обязателен, он появляется только тогда, когда у нас есть несколько таблиц и нам необходимо найти какие-то закономерности, связи между ними. Строки в таких таблицах называются записями, а столбцы называются полями.

Primary Key (PK)				Foreign Key (FK)
ID	**Name**	**Status**	**City**	**Number**
1	Ivan	Single	Oslo	11
2	Petr	Married	New York	13
3	Artem	Single	Minsk	14
4	Maria	Married	Moscow	15
5	Anna	Married	London	16

Foreign Key (FK)		
Number	**Gender**	**Age**
11	Male	25
13	Male	40
14	Male	21
15	Female	28
16	Female	55

Кроме того, в состав реляционной модели данных включают теорию нормализации.

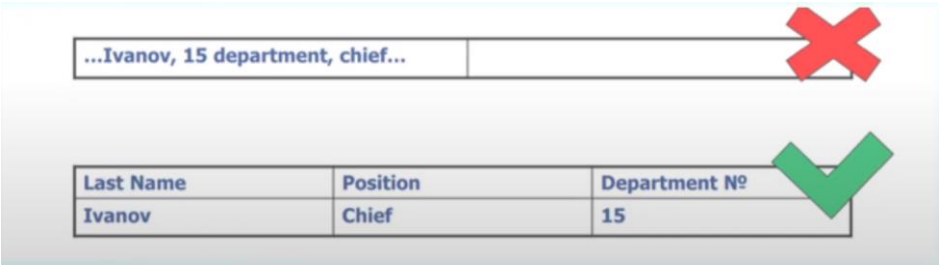
Нормализация - это процесс преобразования отношений (таблиц) базы данных к виду, отвечающему нормальным формам.

Нормальная форма — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации.

Первая нормальная форма (1NF)

Переменная отношения находится в первой нормальной форме (1NF) тогда и только тогда, когда в любом допустимом значении отношения каждый его кортеж (строка) содержит только одно значение для каждого из атрибутов.



Исходная ненормализованная (то есть не являющаяся правильным представлением некоторого отношения) таблица:

Сотрудник	Номер телефона
Иванов И. И.	283-56-82 390-57-34
Петров П. П.	708-62-34

Таблица, приведённая к 1NF, являющаяся правильным представлением некоторого отношения:

Сотрудник	Номер телефона
Иванов И. И.	283-56-82
Иванов И. И.	390-57-34
Петров П. П.	708-62-34

Вторая нормальная форма (2NF)

Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут (поле таблицы) неприводимо (функционально полно) зависит от её потенциального ключа.

Если потенциальный ключ является простым, то есть состоит из единственного атрибута, то любая функциональная зависимость от него является неприводимой (полной). Если потенциальный ключ является составным, то, согласно определению второй нормальной формы, в отношении не должно быть неключевых атрибутов, зависящих от части составного потенциального ключа.

Вторая нормальная форма по определению запрещает наличие неключевых атрибутов, которые вообще не зависят от потенциального ключа. Таким образом, вторая нормальная форма в том числе запрещает создавать отношения как несвязанные (хаотические, случайные) наборы атрибутов.

Пример приведения отношения ко второй нормальной форме:

Пусть в следующем отношении первичный ключ образует пара атрибутов {Филиал компании, Должность}:

R

Филиал компании	**Должность**	**Зарплата**	**Наличие компьютера**
Филиал в Томске	Уборщик	20000	Нет
Филиал в Москве	Программист	40000	Есть
Филиал в Томске	Программист	25000	Есть

Допустим, что зарплата зависит от филиала и должности, а наличие компьютера зависит только от должности. Существует функциональная зависимость Должность → Наличие компьютера, в которой левая часть является лишь частью первичного ключа, что нарушает условие второй нормальной формы.

Для приведения к 2NF исходное отношение следует декомпозировать на два отношения:

R1

Филиал компании	**Должность**	**Зарплата**
Филиал в Томске	Уборщик	20000
Филиал в Томске	Программист	25000
Филиал в Москве	Программист	40000

R2

Должность	**Наличие компьютера**
Уборщик	Нет
Программист	Есть

Третья нормальная форма (3NF)

Переменная отношения находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме, и отсутствуют транзитивные функциональные зависимости неключевых атрибутов от ключевых.

Транзитивная функциональная зависимость выражается следующим образом: A → B и B → C. То есть атрибут C транзитивно зависит от атрибута A, если атрибут C зависит от атрибута B, а атрибут B зависит от атрибута A.

Рассмотрим в качестве примера переменную отношения R1:

R1

Сотрудник	**Отдел**	**Телефон**
Гришин	Бухгалтерия	11-22-33
Васильев	Бухгалтерия	11-22-33
Петров	Снабжение	44-55-66

Каждый сотрудник относится исключительно к одному отделу; каждый отдел имеет единственный телефон. Атрибут Сотрудник является первичным ключом. Личных телефонов у сотрудников нет, и телефон сотрудника зависит исключительно от отдела.

В примере существуют следующие функциональные зависимости: Сотрудник → Отдел, Отдел → Телефон, Сотрудник → Телефон.

Переменная отношения R1 находится во второй нормальной форме, поскольку каждый атрибут имеет неприводимую функциональную зависимость от потенциального ключа Сотрудник.

Зависимость Сотрудник → Телефон является транзитивной (Сотрудник → Отдел, Отдел → Телефон), следовательно, отношение не находится в третьей нормальной форме.

В результате разделения R1 получаются две переменные отношения, находящиеся в 3NF:

R2

Отдел	**Телефон**
Бухгалтерия	11-22-33
Снабжение	44-55-66

R3

Сотрудник	**Отдел**
Гришин	Бухгалтерия
Васильев	Бухгалтерия
Петров	Снабжение

Денормализация (denormalization) — намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации, обычно проводимое с целью ускорения операций чтения из базы за счет добавления избыточных данных.

Устранение аномалий данных в соответствии с теорией реляционных баз данных требует, чтобы любая база данных была нормализована, то есть соответствовала требованиям нормальных форм. Соответствие требованиям нормализации минимизирует избыточность данных в базе данных и обеспечивает отсутствие многих видов логических ошибок обновления и выборки данных.

Однако при запросах большого количества данных операция соединения нормализованных отношений выполняется неприемлемо долго. Вследствие этого в ситуациях, когда производительность таких запросов невозможно повысить иными средствами, может проводиться денормализация — композиция нескольких отношений (таблиц) в одну, которая, как правило, находится во второй, но не в третьей нормальной форме. За счёт такого перепроектирования операция соединения при выборке становится ненужной и запросы выборки, которые ранее требовали соединения, работают быстрее.

Следует помнить, что денормализация всегда выполняется за счёт повышения риска нарушения целостности данных при операциях модификации. Поэтому денормализацию следует проводить в крайнем случае, если другие меры повышения производительности невозможны. Идеально, если денормализованная БД используется только на чтение. Кроме того, следует учесть, что ускорение одних запросов на денормализованной БД может сопровождаться замедлением других запросов, которые ранее выполнялись отдельно на нормализованных отношениях.

Отношения в БД:

Отношения один к одному - это означает, что каждая запись в одной таблице соответствует только одной записи в другой таблице.

В одной таблице.

Связь один-к-одному легко моделируется в одной таблице. Записи таблицы содержат данные, которые находятся в связи один-к-одному с первичным ключом или записью.

В отдельных таблицах.

В редких случаях связь один-к-одному моделируется используя две таблицы. Такой вариант иногда необходим, чтобы преодолеть ограничения РСУБД или с целью увеличения производительности. Или порой вы можете решить, что вы хотите разделить две сущности в разные таблицы в то время, как они все еще имеют связь один-к-одному.

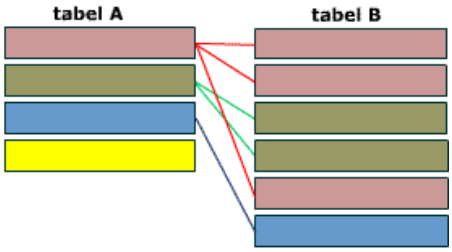
Пример связи один-к-одному:

- Люди и их паспорта. Каждый человек в стране имеет только один действующий паспорт и каждый паспорт принадлежит только одному человеку.

Отношения один-ко-многим - означает, что для каждой записи в одной таблице соответствует одна или несколько записей в другой таблице.

Отношения многие к одному - являются точно такими же как, “один ко многим”, просто такой тип отношений зависит от вашей точки зрения.

Когда одна запись в таблице А может быть связана с 0, 1 или множеством записей в таблице В, вы имеете дело со связью один-ко-многим. В реляционной модели данных связь один-ко-многим использует две таблицы.



Схематическое представление связи один-ко-многим. Запись в таблице А имеет 0, 1 или множество ассоциированных ей записей в таблице В.

Как опознать связь один-ко-многим?

Если у вас есть две сущности спросите себя:

1) Сколько объектов из В могут относится к объекту А?

2) Сколько объектов из А могут относиться к объекту из В?

Если на первый вопрос ответ – множество, а на второй – один (или возможно, что ни одного), то вы имеете дело со связью один-ко-многим.

Некоторые примеры связи один-ко-многим:

- Машина и ее части. Каждая часть машины единовременно принадлежит только одной машине, но машина может иметь множество частей.
- Кинотеатры и экраны. В одном кинотеатре может быть множество экранов, но каждый экран принадлежит только одному кинотеатру.
- Дома и улицы. На улице может быть несколько домов, но каждый дом принадлежит только одной улице.

Отношения многие ко многим - когда возникают отношения между двумя таблицами в тех случаях, когда каждой записи в одной таблице соответствует 0, 1, 2 или более записи в другой таблице, и наоборот.

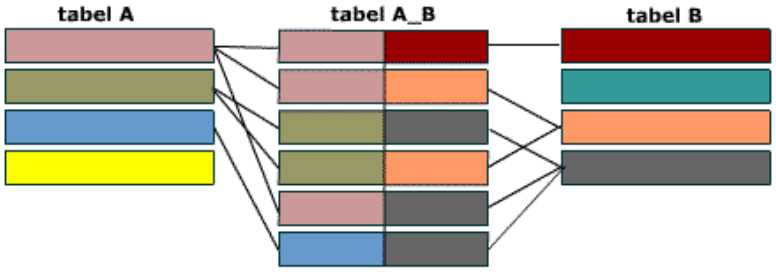
Связь многие-ко-многим – это связь, при которой множественным записям из одной таблицы (А) могут соответствовать множественные записи из другой (В).

Примером такой связи может служить школа, где учителя обучают учащихся. В большинстве школ каждый учитель обучает многих учащихся, а каждый учащийся может обучаться несколькими учителями.

Связь между поставщиком пива и пивом, которое они поставляют – это тоже связь многие-ко-многим. Поставщик, во многих случаях, предоставляет более одного вида пива, а каждый вид пива может быть предоставлен множеством поставщиков.

Создание связи многие-ко-многим.

Связь многие-ко-многим создается с помощью трех таблиц. Две таблицы – “источника” и одна соединительная таблица. Первичный ключ соединительной таблицы A_B – составной. Она состоит из двух полей, двух внешних ключей, которые ссылаются на первичные ключи таблиц А и В.



Все первичные ключи должны быть уникальными. Это подразумевает и то, что комбинация полей А и В должна быть уникальной в таблице A_B.

Пример проект базы данных ниже демонстрирует вам таблицы, которые могли бы существовать в связи многие-ко-многим между бельгийскими брендами пива и их поставщиками в Нидерландах. Обратите внимание, что все комбинации beer_id и distributor_id уникальны в соединительной таблице.

Table beer

beer_id	name
157	Gentse Tripel
158	Uilenspiegel
146	Duvel
123	Leffe
222	Brugse Tripel
160	Sint Bernardus Pater
163	Jupiler

Table beer_distributor

beer_id	distributor_id
157	**AC001**
157	**AB899**
157	**AC009**
158	**AC009**
163	**AC009**
160	**AB999**
163	**AB999**
222	**AB999**
123	**AB999**
146	**AB999**
158	**AB999**
157	**AB999**

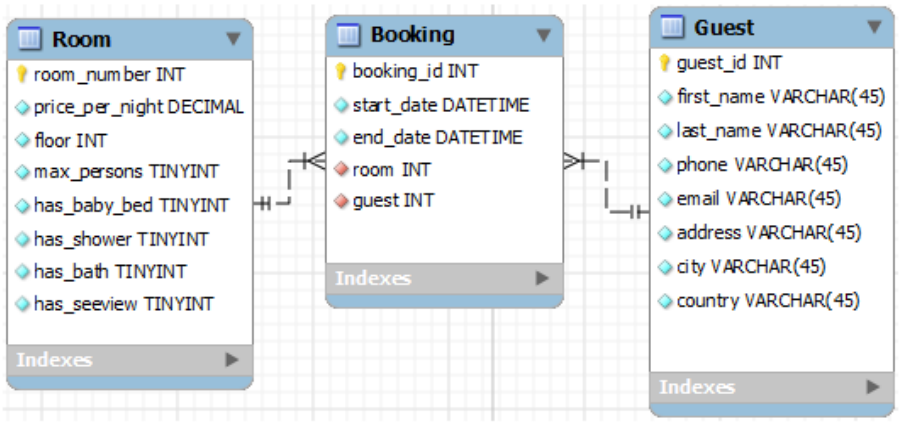
Table distributor

distributor_id	name
AB999	De vrolijke drinker
AC001	Horeca Import NL
AC002	van Gent bierimport
AB899	Jansen Horeca
AC008	De bierleverancier
AC009	Petersen Drankenhandel

Обратите внимание, что в таблицах выше поля первичных ключей окрашены в синий цвет и имеют подчеркивание. И снова обратите внимание, что соединительная таблица beer_distributor имеет первичный ключ, составленный из двух внешних ключей. Соединительная таблица всегда имеет составной первичный ключ.

Другой пример связи многие-ко-многим: заказ билетов в отеле.

В качестве последнего примера позвольте мне показать, как могла бы быть смоделирована таблица заказов номеров гостиницы посетителями.



Соединительная таблица связи многие-ко-многим имеет дополнительные поля.

В этом примере вы видите, что между таблицами гостей и комнат существует связь многие-ко-многим. Одна комната может быть заказана многими гостями с течением времени и с течением времени гость может заказывать многие комнаты в отеле. Соединительная таблица в данном случае является не классической соединительной таблицей, которая состоит только из двух внешних ключей. Она является отдельной сущностью, которая имеет связи с двумя другими сущностями.

Вы часто будете сталкиваться с такими ситуациями, когда совокупность двух сущностей будет являться новой сущностью.

СУБД (Систе́ма управле́ния ба́зами да́нных) (Database Management System, DBMS) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

СУБД — комплекс программ, позволяющих создать базу данных (БД) и манипулировать данными (вставлять, обновлять, удалять и выбирать). Система обеспечивает безопасность, надёжность хранения и целостность данных, а также предоставляет средства для администрирования БД

Основные функции СУБД

- управление данными во внешней памяти (на дисках);
- управление данными в оперативной памяти с использованием дискового кэша;
- журнализация изменений, резервное копирование и восстановление базы данных после сбоев;
- поддержка языков БД (язык определения данных, язык манипулирования данными).

По способу доступа к БД СУБД бывают:

1. Файл-серверные

В файл-серверных СУБД файлы данных располагаются централизованно на файл-сервере. СУБД располагается на каждом клиентском компьютере (рабочей станции). Доступ СУБД к данным осуществляется через локальную сеть. Синхронизация чтений и обновлений осуществляется посредством файловых блокировок.

Преимуществом этой архитектуры является низкая нагрузка на процессор файлового сервера.

Недостатки: потенциально высокая загрузка локальной сети; затруднённость или невозможность централизованного управления; затруднённость или невозможность обеспечения таких важных характеристик, как высокая надёжность, высокая доступность и высокая безопасность. Применяются чаще всего в локальных приложениях, которые используют функции управления БД; в системах с низкой интенсивностью обработки данных и низкими пиковыми нагрузками на БД.

На данный момент файл-серверная технология считается устаревшей, а её использование в крупных информационных системах — недостатком.

Примеры: Microsoft Access, Paradox, dBase, FoxPro, Visual FoxPro.

2. Клиент-серверные

Клиент-серверная СУБД располагается на сервере вместе с БД и осуществляет доступ к БД непосредственно, в монопольном режиме. Все клиентские запросы на обработку данных обрабатываются клиент-серверной СУБД централизованно.

Недостаток клиент-серверных СУБД состоит в повышенных требованиях к серверу.

Достоинства: потенциально более низкая загрузка локальной сети; удобство централизованного управления; удобство обеспечения таких важных характеристик, как высокая надёжность, высокая доступность и высокая безопасность.

Примеры: Oracle Database, Firebird, Interbase, IBM DB2, Informix, MS SQL Server, Sybase Adaptive Server Enterprise, PostgreSQL, MySQL, Caché, ЛИНТЕР.

SQL - Structured Query Language (язык структурированных запросов).

3. Встраиваемые

Встраиваемая СУБД — СУБД, которая может поставляться как составная часть некоторого программного продукта, не требуя процедуры самостоятельной установки. Встраиваемая СУБД предназначена для локального хранения данных своего приложения и не рассчитана на коллективное использование в сети.

Физически встраиваемая СУБД чаще всего реализована в виде подключаемой библиотеки. Доступ к данным со стороны приложения может происходить через SQL либо через специальные программные интерфейсы.

Примеры: OpenEdge, SQLite, BerkeleyDB, Firebird Embedded, Microsoft SQL Server Compact, ЛИНТЕР.

Тестирование мобильных приложений.

Что самое первое мы должны сделать перед тем как начинать тестировать мобильные приложения? Конечно же выбрать на какой платформе мы хотим осуществлять нашу разработку, какие девайсы мы хотим использовать, какой форм-фактор, т.е. нам необходимо собрать нужную статистику. Обычно этим занимаются не тестировщики, но также вас могут привлекать к этим вопросам, потому что именно вам необходимо будет проводить это самое тестирование.

Поэтому первый аспект, который мы затронем - это сбор статистики по мобилке.

developer.android.com/about/dashboards - статистика Android
developer.apple.com/support/app-store - статистика iOS
mixpanel.com/trends/#report/android_vs_ios - Android vs iOS
gs.statcounter.com/vendor-market-share/mobile/worldwide - статистика по использованию устройств (1)
appbrain.com/stats/top-android-phones-tablets-by-country?country=gl - статистика по использованию устройств (2)
www.browserstack.com/test-on-the-right-mobile-devices - рекомендации по формированию парка девайсов для тестирования мобильных приложений
https://developer.android.com/studio - Android Studio - эмулятор Android
https://developer.apple.com/xcode - Xcode – эмулятор Apple

Типы мобильных приложений:

Web-приложения - это более раскрученные с точки зрения функциональности веб-сайты, по сути, это вебсайт, который адаптирован для наших мобильных приложений, для экранов мобильных приложений. Самый простой пример - Вконтакте. Т.е. если мы просто перейдем на него, то нам откроется обычный веб-сайт в приложении для нашей с вами работы на десктопе. Но также у этого веб-сайта есть мобильная версия, и именно она уже адаптирована для мобильных устройств. Т.е. это вот и есть мобильное приложение.

Давайте с вами теперь поговорим о том, какие есть плюсы и минусы именно у мобильных веб-приложений.

Начнем с плюсов. Веб-приложение представляет собой одну версию для всех платформ, т.е. нам не нужно разрабатывать отдельное приложение для ios, для android. Есть одна версия для всех платформ. Это вебсайт, который адаптирован для работы с мобильным устройством. Также из плюсов можно вынести то, что обновление этого ресурса осуществляется на сервере, т.е. нам не нужно привлекать юзеров к этой работе, всё происходит в автоматическом режиме. Веб-приложения достаточно просты в разработке, потому что здесь используются веб-технологии.

Из минусов можно отнести то, что для веб-приложения всегда необходимо интернет-соединение для работы. Нет возможности использовать функции мобильного устройства. Т.е. у мобильника есть камера, есть push-уведомления, есть доступ к gps, всё это веб-приложение не может использовать. Данное приложение нельзя загрузить из магазина, их там нет, потому что это, в принципе, веб-сайт.

Веб-приложения - maps.google.com, vk.com и т.д.

Нативные приложения - это те приложения, которые мы непосредственно скачиваем с нашего магазина. Они разрабатываются непосредственно на тех инструментах, тулах, на тех языках, которые нужны и используются именно для платформы, они специфические. Приложения для ios, к примеру, разработаны на языке swift. Приложения для android разрабатываются на java, либо kotlin. Это специфические языки именно свойственные для данных платформ.

Плюсы: нативные приложения работают без интернет-соединения. Имеют доступ к возможностям мобильного устройства, т.е. можно отсылать push-уведомления, можно использовать ресурсы камеры, можно использовать gps и т.д., если пользователь это разрешает делать. Эти приложения достаточно высокоскоростные и более интересные, чем другие виды приложений, потому что могут использовать функции устройства.

Минусы: разработка отдельной версии приложения осуществляется для каждой платформы. У каждой платформы, для каждого магазина, есть свои требования к разработке того или иного приложения, к его дизайну. Мы не можем выпускать одинаковые приложения, написанные на одном и том же языке для двух платформ. Нам требуется больше денег и времени для разработки. Больше денег, к примеру, может пойти на покупку различных тестовых девайсов на ios устройства или на android. Этих устройств очень много, их нужно покупать в различных вариациях. Обновление и загрузка этих приложений происходит с привлечением юзера. Т.е. мы не можем в автоматическом режиме проводить эти обновления, когда нам захочется. User обязательно должен это скачать. Это также вызывает некоторые риски, потому что если у вас есть какой-то критический баг для работы вашего приложения, а юзер долго не скачивал это обновление, то юзер может воспринять это, как некачественный продукт, оставить негативный отзыв.

Нативные приложения - Instagram, Shazam и т.д.

Гибридные приложения. Включают в себя как веб-элементы, так и элементы нативных приложений.

Плюсы: кроссплатформенные. Потому что чаще всего они пишутся с использованием различных языков вэба, веб-технологий, т.е. там нет такой привязки именно к тому, что мы должны написать только на java или kotlin, swift. Нам не нужно разрабатывать отдельные версии для различных маркетов. Гибридное приложение имеет доступ к функциям телефона, т.е. мы можем их подключать, у них есть доступ к внутренним данным телефона. Они дешевле в разработке, чем нативные, потому что не нужно разрабатывать сто пятьсот версий для различных версий платформ и самих платформ, и не нужно такое количество большое устройств, которые необходимы нам для тестирования.

Минусы: низкая скорость работы, так как у нас здесь используются веб-технологии, мы не можем некоторые моменты оптимизировать. Они не практичны в виду долгого обновления фреймворка. Т.е. если мы, например, используем наше приложение и написали его на каком-то фреймворке, то на его обновление может уйти, например, от 3 до 6 месяцев, это очень долго.

Гибридные приложения - агрегаторы новостей, различные приложения, связанные с фотографированием, Heartcamera для ios, Tripcase.

Гайдлайн. Это требование к нашему дизайну. Если мы говорим про android, у нас есть непосредственно гайды, где содержится информация по юзер интерфейсу и навигации. Когда мы выбираем необходимый нам элемент - открывается информация о том, как это должно выглядеть, как должно работать, и мы должны обязательно соответствовать всему тому, что здесь написано. Если мы этому не будем соответствовать - ваше приложение просто declined, и скажут, что вы не соответствуете нашим требованиям. Если вы для себя решили выбрать путь тестирования мобильных приложений - вы должны выучить эти правила и основные принципы, которые в них используются. Часто на собеседованиях спрашивают именно эти гайдлайны и насколько вы их знаете, спрашивают по элементам, какие могут быть отличия в ios устройствах, какие отличия в android устройствах. То же самое про гайдлайны и для ios.

developer.android.com/guide/topics/ui – Гайдлайн Android
developer.apple.com/design/human-interface-guidelines/ - Гайдлайн Apple

Сегодня расскажу вам о тех проверках, которые нужно проводить на наших мобильных приложениях непосредственно для того, чтобы убедиться, что они отрабатывают свои функциональности.

Существует несколько типов основных проверок, которые нужно осуществлять на нашем мобильном телефоне:

Проверки на прерывание. Т.е. когда у нас работа нашего приложения прерывается чем-то извне. Входящие/исходящие вызовы. К примеру, нам кто-то звонит и мы сидим в каком-то приложении, в игрушке, либо в нашем браузере, и мы должны обязательно проверить, что происходит с приложением после того, когда заканчивается вызов. Например, оно может крашиться, вылетать, либо же зависать. Т.е. мы должны убедиться в том, что все отрабатывается корректно. То же касается и входящих вызовов, т.е., к примеру, если мы свернули наше приложение в трей и мы кому-то звоним, а потом хотим обратно вернуться в наше приложение, то мы должны посмотреть, всё ли там хорошо, действительно ли осталась информация о том, что мы делали, в оперативной памяти и эта информация прогрузилась и показалась нам в том же неизменном виде.

Также есть такие типы прерываний, как всплывающее окно уведомлений (push-уведомления). Т.е., к примеру, вы во что-то играете и вам приходит сообщение из какой-то социальной сети, появляется push-уведомление, мы можем это проверить, посмотреть, как реагирует на это основное приложение, в котором мы сейчас работаем.

Следующий вид - прерывание при разрядке, подзарядке. Т.е., к примеру, если наш телефон находится на минимальном пороге по уровню зарядки - как ведет себя приложение, опять же, может появляться какое-

то уведомление о том, что вам нужно поставить телефон на зарядку, или что телефон переведен в экономный режим. Т.е. смотрим на то, что наше приложение работает корректно. Подзарядка значит то, что, к примеру, мы берем наш телефон и ставим его на зарядную станцию, смотрим, всё ли работает корректно в нашем приложении, т.е. не вылетает ли оно, не сворачивается в трей, не начинает тормозить.

Сворачивание/разворачивание приложений. Если мы наше приложение сворачиваем в трей и обратно его разворачиваем, смотрим, что там действительно сохранилась информация, если нам позволяет это сделать количество оперативной памяти на нашем телефоне, так как если у нас этой памяти мало и нашим приложением, к примеру, будет какая-нибудь тяжелая игрушка, то, конечно же, у нас есть вероятность того, что просто именно из-за оперативной памяти не будет осуществляется загрузка данных.

Тестирование установки. Проверяем, что наше приложение устанавливается корректно, что мы его можем удалить с какого-то маркета, либо же можем его удалить непосредственно с телефона, посредством, например, долгого зажатия, нажав на крестик, и все действительно удаляется, либо же из табеля приложения внутри настроек. Нужно посмотреть, как проходит переустановка, опять же, если мы удалили приложение когда-то и решили его обратно переустановить. Возможно, у нас сохранилась какая-то папка с данными от этого приложения, и это может повлиять на то, что приложение работает некорректно после переустановки. Ну и конечно обновление, когда мы осуществляем какой-то апдейт из нашего маркета, посмотреть, что переход на новую версию никак не влияет на корректную работу нашего приложения.

Проверка интернет-соединения. Мы должны проверить, как работает наше приложение на различных типах соединения, будь то вай-фай, 2g, 3g, 4g, 5g и т.д. Проверить качество соединения, т.е. когда у нас плохое соединение, когда хорошее, когда отличное, когда у нас, например, вообще отсутствует интернет, когда происходит потеря связи, например, блокируют интернет в стране, или мы находимся на природе и там нет интернета. Смотрим, как работает наше приложение, и изменяем типы соединений, к примеру, мы сидим на 3g, меняем на 4g и др., смотрим, всё ли будет работать хорошо, не будет ли прерываться соединение.

Все эти функции проверок реализованы на эмуляторах, т.е. вам не обязательно это все делать на реальном девайсе, хотя рекомендовано делать тесты на реальном девайсе, но если такой возможности нет, вы можете симулировать.

Работа с функциями телефона. У нас есть какие-то встроенные функции телефона, например, gps, фото, видео и др. Если мы тестируем gps, то смотрим, что, к примеру, с помощью api мы можем подключаться к api телефона и использовать a-gps, смотрим, действительно ли наша геолокация отрабатывает корректно. Камера - проверить фронтальную камеру, тыльную, ширик, телекамеру и т.д.

Мы должны проверить размер экрана, разрешение, ориентацию на различных форм-факторах. Например, если заявлено, что наш продукт может работать на планшете, на телефоне, где разрешение fullHD, или fullHD+, или HD, всё должны проверить, корректно ли всё отображается на всех устройствах, прорисовываются ли все наши страницы в нашем приложении. Обязательно смотрим на ориентацию, т.е. если у нас заявлено, что может работать в горизонтальном и вертикальном режиме, действительно ли это происходит, когда мы переворачиваем телефон, для этого в телефоне есть акселерометр, для определения положения в пространстве.

Работа с жестами. Проверяем тап на различные части нашего экрана телефона, проверяем сворачивание, разворачивание. Всё вот это мы должны проверить, все наши жесты, свайпы, увеличения, уменьшения нашей картинки.

Тестирование производительности. Проверяем скорость нашего приложения, как оно быстро работает, отзывчивость, например, если мы тапаем на него, то оно должно быстренько переходить на следующую страницу, либо же если мы свайпаем, тоже должно после него наступать какое-то действие, как это быстро происходит.

Загрузка оперативной памяти - обязательно смотрим, чтобы, к примеру, наше приложение было хорошо оптимизировано для той или иной платформы и не нагружало очень сильно телефон.

Зависимость от заряда батареи - т.е. это взаимосвязи с прерываниями, также смотрим не тормозит ли наше приложение, когда уровень батареи падает на определенное количество процентов, можем проверить чуть ли не каждые 10 процентов. Всё зависит от вашей фантазии и от того, что от вас хочет получить конечный пользователь и сам бизнес.

Запуск с внутренней памяти/flash накопителя. Это больше актуально для android приложений. Мы можем установить приложение как на внутреннюю память нашего телефона, так и установить на флешку и посмотреть, как работает наше приложение. Обязательно проверить установку на флэш-накопитель, удаление, обновления, переустановку и т.д.

Количество проверок зависит только от того, насколько тщательно вам нужно это все провести и сколько у вас есть времени на это тестирование. Это основной перечень проверок мобильных приложений. Помимо этого, существуют и наши стандартные все проверки, все наши типы тестирования, которые мы также можем применять для наших мобильных приложений.