

**По запуску кода на исполнение**

**Статическое тестирование** (static testing) — тестирование без запуска кода на исполнение. В рамках этого подхода тестированию могут подвергаться:

- Документы (требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т.д.).

- Графические прототипы (например, эскизы пользовательского интерфейса).

- Код приложения (что часто выполняется самими программистами в рамках аудита кода (code review118), являющегося специфической вариацией взаимного просмотра{51} в применении к исходному коду). Код приложения также можно проверять с использованием техник тестирования на основе структур кода{97} .

- Параметры (настройки) среды исполнения приложения.

- Подготовленные тестовые данные.

**Динамическое тестирование** (dynamic testing) — тестирование с запуском кода на исполнение. Запускаться на исполнение может как код всего приложения целиком (системное тестирование), так и код нескольких взаимосвязанных частей (интеграционное тестирование), отдельных частей (модульное или компонентное тестирование{77} ) и даже отдельные участки кода. Основная идея этого вида тестирования состоит в том, что проверяется реальное поведение (части) приложения.

**по доступу к коду и архитектуре приложения**

**Метод белого ящика** white box testing , open box testing, clear box testing, glass box testing) — у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного. Выделяют даже сопутствующую тестированию по методу белого ящика глобальную технику — тестирование на основе дизайна (design-based testing121). Для более глубокого изучения сути метода белого ящика рекомендуется ознакомиться с техниками исследования потока управления{96} или потока данных{96} , использования диаграмм состояний{97} . Некоторые авторы склонны жёстко связывать этот метод со статическим тестированием, но ничто не мешает тестировщику запустить код на выполнение и при этом периодически обращаться к самому коду (а модульное тестирование{77} и вовсе предполагает запуск кода на исполнение и при этом работу именно с кодом, а не с «приложением целиком»).

**Метод чёрного ящика** (black box testing122 , closed box testing, specificationbased testing) — у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования. При этом абсолютное большинство перечисленных на рисунках 2.3.b и 2.3.c видов тестирования работают по методу чёрного ящика, идею которого в альтернативном определении можно сформулировать так: тестировщик оказывает на приложение воздействия (и проверяет реакцию) тем же способом, каким при реальной эксплуатации приложения на него воздействовали бы пользователи или другие приложения. В рамках тестирования по методу чёрного ящика основной информацией для создания тест-кейсов выступает документация (особенно — требования (requirements-based testing) и общий здравый смысл (для случаев, когда поведение приложения в некоторой ситуации не регламентировано явно; иногда это называют «тестированием на основе неявных требований», но канонического определения у этого подхода нет).

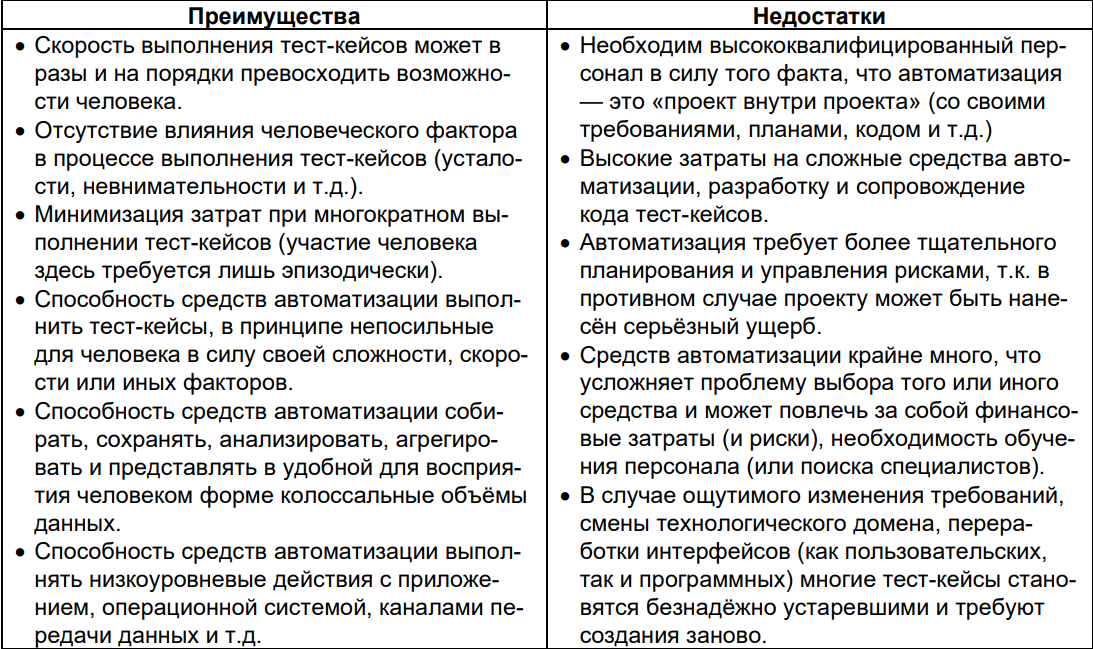
**Метод серого ящика** (gray box testing124) — комбинация методов белого ящика и чёрного ящика, состоящая в том, что к части кода и архитектуры у тестировщика доступ есть, а к части — нет. На рисунках 2.3.b и 2.3.c этот метод обозначен особым пунктиром и серым цветом потому, что его явное упоминание — крайне редкий случай: обычно говорят о методах белого или чёрного ящика в применении к тем или иным частям приложения, при этом понимая, что «приложение целиком» тестируется по методу серого ящика.

**Классификация по степени автоматизации**

**Ручное тестирование** (manual testing126) — тестирование, в котором тесткейсы выполняются человеком вручную без использования средств автоматизации. Несмотря на то что это звучит очень просто, от тестировщика в те или иные моменты времени требуются такие качества, как терпеливость, наблюдательность, креативность, умение ставить нестандартные эксперименты, а также умение видеть и понимать, что происходит «внутри системы», т.е. как внешние воздействия на приложение трансформируются в его внутренние процессы.

**Автоматизированное тестирование** (automated testing, test automation127) — набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования. Тест-кейсы частично или полностью выполняет специальное инструментальное средство, однако разработка тест-кейсов, подготовка данных, оценка результатов выполнения, написания отчётов об обнаруженных дефектах — всё это и многое другое по-прежнему делает человек.

Преимущества и недостатки автоматизированного тестирования



**Классификация по уровню детализации приложения (по уровню тестирования)**

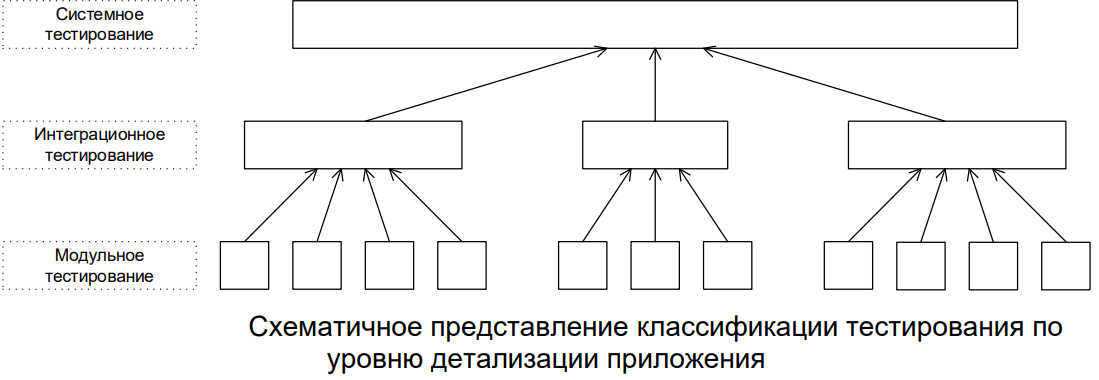
(вспоминай пирамиду тестирования) <https://docs.google.com/document/d/1OTu4jGs_QAvco7qK78WomBFMGS_0utIMBKSRwm1dgjE/edit?usp=drive_link>

**Модульное** (компонентное) тестирование (**unit** **testing**, module testing, component testing) направлено на проверку отдельных небольших частей приложения, которые (как правило) можно исследовать изолированно от других подобных частей. При выполнении данного тестирования могут проверяться отдельные функции или методы классов, сами классы, взаимодействие классов, небольшие библиотеки, отдельные части приложения. Часто данный вид тестирования реализуется с использованием специальных технологий и инструментальных средств автоматизации тестирования{76} , значительно упрощающих и ускоряющих разработку соответствующих тест-кейсов.

**Интеграционное тестирование** (integration testing, component integration testing, pairwise integration testing , system integration testing , incremental testing, interface testing, thread testing) направлено на проверку взаимодействия между несколькими частями приложения (каждая из которых, в свою очередь, проверена отдельно на стадии модульного тестирования). К сожалению, даже если мы работаем с очень качественными отдельными компонентами, «на стыке» их взаимодействия часто возникают проблемы. Именно эти проблемы и выявляет интеграционное тестирование. (См. также техники восходящего, нисходящего и гибридного тестирования в хронологической классификации по иерархии компонентов)

<https://docs.google.com/document/d/1Crs5U4GXWtOMlTr1_Ny_4R8S6aBbSo2P3Zt78HT6g98/edit?usp=drive_link>

**Системное тестирование** (system testing) направлено на проверку всего приложения как единого целого, собранного из частей, проверенных на двух предыдущих стадиях. Здесь не только выявляются дефекты «на стыках» компонентов, но и появляется возможность полноценно взаимодействовать с приложением с точки зрения конечного пользователя, применяя множество других видов тестирования, перечисленных в данной главе.



<https://docs.google.com/document/d/1lv6ShfJrERuTFZqEza7qg39BHIzLZUnu3axrRKFcWD0/edit?usp=drive_link>

**Классификация по (убыванию) степени важности тестируемых функций**

**(по уровню функционального тестирования)**

**Дымовое тестирование** (**smoke** test , intake test, build verification test) направлено на проверку самой главной, самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения (или иного объекта, подвергаемого дымовому тестированию). Дымовое тестирование проводится после выхода нового билда, чтобы определить общий уровень качества приложения и принять решение о (не)целесообразности выполнения тестирования критического пути и расширенного тестирования. Поскольку тест-кейсов на уровне дымового тестирования относительно немного, а сами они достаточно просты, но при этом очень часто повторяются, они являются хорошими кандидатами на автоматизацию. В связи с высокой важностью тест-кейсов на данном уровне пороговое значение метрики их прохождения часто выставляется равным 100 % или близким к 100 %.

Пример: пользователь авторизовался, смог найти и добавить товар в корзину, совершил оплату и оформил доставку - в качестве примера для онлайн-магазина.

Проверка основной функциональности обычно не меняется и проводится на каждой новой промежуточной версии продукта, или билде. Иногда этот процесс называют сертификацией билда.

Если Smoke не проходит, то остальное тестирование останавливается и ждет доработки кода.

**Тестирование критического пути** (critical path test) направлено на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. Существует большинство пользователей, которые чаще всего используют некое подмножество функций приложения Именно эти функции и нужно проверить, как только мы убедились, что приложение «в принципе работает»Если по каким-то причинам приложение не выполняет эти функции или выполняет их некорректно, очень многие пользователи не смогут достичь множества своих целей.

**Расширенное** тестирование (extended test146) направлено на исследование всей заявленной в требованиях функциональности — даже той, которая низко проранжирована по степени важности. При этом здесь также учитывается, какая функциональность является более важной, а какая — менее важной. Но при наличии достаточного количества времени и иных ресурсов тесткейсы этого уровня могут затронуть даже самые низкоприоритетные требования.

**Классификация по принципам работы с приложением**

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

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

**Деструктивное тестирование (destructive testing)** - одна из форм негативного тестирования с целью нарушить работоспособность приложения и обнаружить точку отказа. *Пример: нагрузка приложения выше его предела, чтобы оно перестало работать*

**Важно**: Любое тестирование необходимо начинать с позитивных проверок. Наша первоочередная задача убедиться, что приложение работает в стандартных условиях и готово для использования. Нельзя объединять позитивные и негативные тесты, так как это затрудняет локализацию дефектов.

К**лассификация по природе приложения**

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

**Тестирование веб-приложений** (web-applications testing) сопряжено с интенсивной деятельностью в области тестирования совместимости{89} (в особенности — кросс-браузерного тестирования{90} ), тестирования производительности{91} , автоматизации тестирования с использованием широкого спектра инструментальных средств.

**Тестирование мобильных приложений** (mobile applications testing) также требует повышенного внимания к тестированию совместимости{89} , оптимизации производительности{91} (в том числе клиентской части с точки зрения снижения энергопотребления), автоматизации тестирования с применением эмуляторов мобильных устройств.

**Тестирование настольных приложений** (desktop applications testing) является самым классическим среди всех перечисленных в данной классификации, и его особенности зависят от предметной области приложения, нюансов архитектуры, ключевых показателей качества и т.д. Эту классификацию можно продолжать очень долго. Например, можно отдельно рассматривать тестирование консольных приложений (console applications testing) и приложений с графическим интерфейсом (GUI-applications testing), серверных приложений (server applications testing) и клиентских приложений (client applications testing) и т.д.

**Классификация по фокусировке на уровне архитектуры приложения**

Данный вид классификации, как и предыдущий, также является искусственным и отражает лишь концентрацию внимания на отдельной части приложения.

**Тестирование уровня представления** (presentation tier testing) сконцентрировано на той части приложения, которая отвечает за взаимодействие с «внешним миром» (как пользователями, так и другими приложениями). Здесь исследуются вопросы удобства использования, скорости отклика интерфейса, совместимости с браузерами, корректности работы интерфейсов.

**Тестирование уровня бизнес-логики** (business logic tier testing) отвечает за проверку основного набора функций приложения и строится на базе ключевых требований к приложению, бизнес-правил и общей проверки функциональности.

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

**Классификация по привлечению конечных пользователей**

Все три перечисленных ниже вида тестирования относятся к операционному тестированию{88} . **Альфа-тестирование** (alpha testing151) выполняется внутри организации-разработчика с возможным частичным привлечением конечных пользователей. Может являться формой внутреннего приёмочного тестирования{87} . В некоторых источниках отмечается, что это тестирование должно проводиться без привлечения команды разработчиков, но другие источники не выдвигают такого требования. Суть этого вида вкратце: продукт уже можно периодически показывать внешним пользователям, но он ещё достаточно «сырой», потому основное тестирование выполняется организацией-разработчиком.

**Бета-тестирование** (beta testing152) выполняется вне организации-разработчика с активным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования{87} . Суть этого вида вкратце: продукт уже можно открыто показывать внешним пользователям, он уже достаточно стабилен, но проблемы всё ещё могут быть, и для их выявления нужна обратная связь от реальных пользователей.

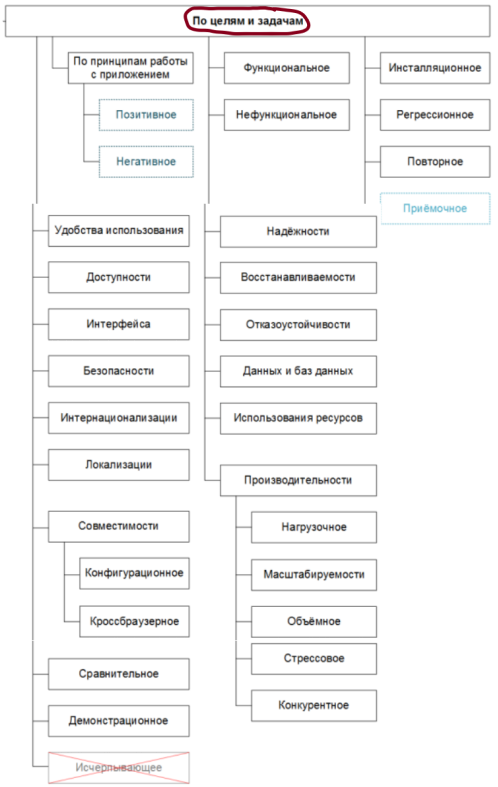
**Гамма-тестирование** (gamma testing153) — финальная стадия тестирования перед выпуском продукта, направленная на исправление незначительных дефектов, обнаруженных в бета-тестировании. Как правило, также выполняется с максимальным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования{87} . Суть этого вида вкратце: продукт уже почти готов, и сейчас обратная связь от реальных пользователей используется для устранения последних недоработок.

**Классификация по степени формализации**

**Исследовательское тестирование** (exploratory testing155) — частично формализованный подход, в рамках которого тестировщик выполняет работу с приложением по выбранному сценарию{146} , который, в свою очередь, дорабатывается в процессе выполнения с целью более полного исследования приложения. Ключевым фактором успеха при выполнении исследовательского тестирования является именно работа по сценарию, а не выполнение разрозненных бездумных операций. Существует даже специальный сценарный подход, называемый сессионным тестированием (session-based testing156). В качестве альтернативы сценариям при выборе действий с приложением иногда могут использоваться чек-листы, и тогда этот вид тестирования называют тестированием на основе чек-листов (checklist-based testing157).

**Свободное (интуитивное) тестирование** (ad hoc testing159) — полностью неформализованный подход, в котором не предполагается использования ни тест-кейсов, ни чек-листов, ни сценариев — тестировщик полностью опирается на свой профессионализм и интуицию (experience-based testing160) для спонтанного выполнения с приложением действий, которые, как он считает, могут обнаружить ошибку. Этот вид тестирования используется редко и исключительно как дополнение к полностью или частично формализованному тестированию в случаях, когда для исследования некоторого аспекта поведения приложения (пока?) нет тест-кейсов.

**Классификация по целям и задачам**



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

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

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

*Функциональное тестирование* отвечает на вопрос *"Что делает система?".*

**Нефункциональное** (non-functional testing) — вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований{41} ), таких как удобство использования, совместимость, производительность, безопасность и т.д.

**Инсталляционное** (installation testing, installability testing) — тестирование, направленное на выявление дефектов, влияющих на протекание стадии инсталляции (установки) приложения. В общем случае такое тестирование проверяет множество сценариев и аспектов работы инсталлятора в таких ситуациях, как:

- новая среда исполнения, в которой приложение ранее не было инсталлировано;

- обновление существующей версии («апгрейд»);

- изменение текущей версии на более старую («даунгрейд»);

- повторная установка приложения с целью устранения возникших проблем («переинсталляция»);

- повторный запуск инсталляции после ошибки, приведшей к невозможности продолжения инсталляции;

- удаление приложения;

- установка нового приложения из семейства приложений;

- автоматическая инсталляция без участия пользователя

или более простое обьяснение

1 Установка ПО

2 Удаление ПО

3 Обновление ПО

4 Откат на предыдущую версию

5 Повторный запуск установки после возникновения ошибки или исправления уже возникших проблем

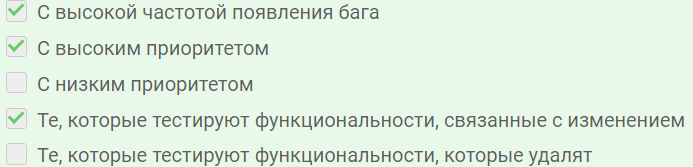
6 Автоматическая установка

7 Установка отдельного компонента из общего пакета программ

**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. Фредерик Брукс в своей книге «Мифический человеко-месяц» писал: «Фундаментальная проблема при сопровождении программ состоит в том, что исправление одной ошибки с большой вероятностью (20–50 %) влечёт появление новой» возможно это также потому что в наше время над проектом работают несколько человек и зачастую никто из них не знает код полностью. Потому регрессионное тестирование является неотъемлемым инструментом обеспечения качества и активно используется практически в любом проекте.

Регрессионное тестирование проводится после любого изменения (новая функциональность, исправление дефекта, удаление функциональности), чтобы убедиться в том, что приложение работает стабильно. Оно не включает в себя ретест (описан следующим), а проводится уже после него.

на практике регресс не проводится каждый раз, когда в системе появляется изменение, Например, две недели мы разрабатывали новые функциональности и тестировали только их. В конце этого периода (итерации) у нас запланирован релиз. И именно перед ним мы проведем одну общую регрессию для всего приложения.

Какие тесты следует добавить в регрессионный набор ? 

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

или

Подтверждающее (re-testing, ретест, повторное тестирование)- тип тестирования, связанного с изменениями, которое выполняется после исправления дефекта для подтверждения того, что отказ, вызванный этим дефектом, не воспроизводится.

**Приёмочное** тестирование (acceptance testing170) — формализованное тестирование, направленное на проверку приложения с точки зрения конечного пользователя/заказчика и вынесения решения о том, принимает ли заказчик работу у исполнителя (проектной команды). Можно выделить следующие подвиды приёмочного тестирования (хотя упоминают их крайне редко, ограничиваясь в основном общим термином «приёмочное тестирование»):

Производственное приёмочное тестирование (factory acceptance testing171) — выполняемое проектной командой исследование полноты и качества реализации приложения с точки зрения его готовности к передаче заказчику. Этот вид тестирования часто рассматривается как синоним альфа-тестирования{84} . o Операционное приёмочное тестирование (operational acceptance testing172 , production acceptance testing) — операционное тестирование{88} , выполняемое с точки зрения выполнения инсталляции, потребления приложением ресурсов, совместимости с программной и аппаратной платформой и т.д.

Итоговое приёмочное тестирование (site acceptance testing173) — тестирование конечными пользователями (представителями заказчика) приложения в реальных условиях эксплуатации с целью вынесения решения о том, требует ли приложение доработок или может быть принято в эксплуатацию в текущем виде.

Операционное тестирование (operational testing174) — тестирование, проводимое в реальной или приближенной к реальной операционной среде (operational environment175), включающей операционную систему, системы управления базами данных, серверы приложений, веб-серверы, аппаратное обеспечение и т.д.

Тестирование удобства использования (usability testing) — тестирование, направленное на исследование того, насколько конечному пользователю понятно, как работать с продуктом а также на то, насколько ему нравится использовать продукт (attractiveness180). И это не оговорка — очень часто успех продукта зависит именно от эмоций, которые он вызывает у пользователей. Для эффективного проведения этого вида тестирования требуется реализовать достаточно серьёзные исследования с привлечением конечных пользователей, проведением маркетинговых исследований и т.д

.

**Тестирование удобства использования (usability testing)** — тестирование, направленное на исследование того, насколько конечному пользователю понятно, как работать с продуктом (understandability, learnability, operability), а также на то, насколько ему нравится использовать продукт (attractiveness).

Что нужно тестировать:

-Общая доступность

-Скорость, производительность

-Удобство навигации и интерфейс

-Плавность

Также этот тип тестирования относится к нефункциональному тестированию.

**Тестирование доступности** (accessibility testing) — тестирование, направленное на исследование пригодности продукта к использованию людьми с ограниченными возможностями (слабым зрением и т.д.).

Что можно тестировать:

Использование вспомогательных технологий в ПО (распознавание речи, экранная клавиатура и лупа, скринридеры)

Возможность использовать приложение одной рукой

Настройки специальной цветопередачи

Наличие понятных инструкций и руководства пользователя

Доступность сайта можно тестировать специальными инструментами, например, WAVE, TAV, Accessibility Valet, Accessibility Developer Tools.

Также этот тип тестирования относится к нефункциональному тестированию.

**Тестирование интерфейса** (interface testing182) — тестирование, направленное на проверку интерфейсов приложения или его компонентов. По определению ISTQB-глоссария этот вид тестирования относится к интеграционному тестированию{77} , и это вполне справедливо для таких его вариаций как тестирование интерфейса прикладного программирования (API testing183) и интерфейса командной строки (CLI testing184), хотя последнее может выступать и как разновидность тестирования пользовательского интерфейса, если через командную строку с приложением взаимодействует пользователь, а не другое приложение. Однако многие источники предлагают включить в состав тестирования интерфейса и тестирование непосредственно интерфейса пользователя (GUI testing185).

**Тестирование безопасности** (security testing186) — тестирование, направленное на проверку способности приложения противостоять злонамеренным попыткам получения доступа к данным или функциям, права на доступ к которым у злоумышленника нет.

**Тестирование интернационализации** (internationalization testing, i18n testing, globalization testing, localizability testing) — тестирование, направленное на проверку готовности продукта к работе с использованием различных языков и с учётом различных национальных и культурных особенностей.

Этот вид тестирования не подразумевает проверки качества соответствующей адаптации (этим занимается тестирование локализации, см. следующий пункт), оно сфокусировано именно на проверке возможности такой адаптации (например: что будет, если открыть файл с иероглифом в имени; как будет работать интерфейс, если всё перевести на японский; может ли приложение искать данные в тексте на корейском и т.д.).

**Тестирование локализации** (localization testing190 , l10n) — тестирование, направленное на проверку корректности и качества адаптации продукта к использованию на том или ином языке с учётом национальных и культурных особенностей. Это тестирование следует за тестированием интернационализации (см. предыдущий пункт) и проверяет корректность перевода и адаптации продукта, а не готовность продукта к таким действиям.

**Тестирование совместимости** (compatibility testing, interoperability testing) — тестирование, направленное на проверку способности приложения работать в указанном окружении.

Здесь, например, может проверяться:

-Совместимость с аппаратной платформой, операционной системой и сетевой инфраструктурой (конфигурационное тестирование, configuration testing192).

- Совместимость с браузерами и их версиями (кросс-браузерное тестирование, cross-browser testing193). (Cм. также тестирование веб-приложений{83} ).

- Совместимость с мобильными устройствами (mobile testing194). (См. также тестирование мобильных приложений).

**Тестирование данных** (data quality testing) **и баз данных** (database integrity testing199) — два близких по смыслу вида тестирования, направленных на исследование таких характеристик данных, как полнота, непротиворечивость, целостность, структурированность и т.д. В контексте баз данных исследованию может подвергаться адекватность модели предметной области, способность модели обеспечивать целостность и консистентность данных, корректность работы триггеров, хранимых процедур и т.д.

**Тестирование использования ресурсов** (resource utilization testing200 , efficiency testing201 , storage testing202) — совокупность видов тестирования, проверяющих эффективность использования приложением доступных ему ресурсов и зависимость результатов работы приложения от количества доступных ему ресурсов. Часто эти виды тестирования прямо или косвенно примыкают к техникам тестирования производительности.

**Сравнительное тестирование** (comparison testing203) — тестирование, направленное на сравнительный анализ преимуществ и недостатков разрабатываемого продукта по отношению к его основным конкурентам.

**Демонстрационное тестирование** (qualification testing204) — формальный процесс демонстрации заказчику продукта с целью подтверждения, что продукт соответствует всем заявленным требованиям. В отличие от приёмочного тестирования{87} этот процесс более строгий и всеобъемлющий, но может проводиться и на промежуточных стадиях разработки продукта.

**Исчерпывающее тестирование** (exhaustive testing205) — тестирование приложения со всеми возможными комбинациями всех возможных входных данных во всех возможных условиях выполнения. Для сколь бы то ни было сложной системы нереализуемо, но может применяться для проверки отдельных крайне простых компонентов.

**Тестирование надёжности** (reliability testing206) — тестирование способности приложения выполнять свои функции в заданных условиях на протяжении заданного времени или заданного количества операций.

**Тестирование восстанавливаемости** (recoverability testing207) — 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 testing208) — 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 testing210 , capacity testing211) — исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).

Тестирование масштабируемости (scalability testing212) — исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов.

Объёмное тестирование (volume testing213) — исследование производительности приложения при обработке различных (как правило, больших) объёмов данных.

Стрессовое тестирование (stress testing214) — 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 testing215) и представляет собой крайнюю форму негативного тестирования{82} .

Конкурентное тестирование (concurrency testing216) — исследование поведения приложения в ситуации, когда ему приходится обрабатывать большое количество одновременно поступающих запросов, что вызывает конкуренцию между запросами за ресурсы (базу данных, память, канал передачи данных, дисковую подсистему и т.д.). Иногда под конкурентным тестированием понимают также исследование работы многопоточных приложений и корректность синхронизации действий, производимых в разных потоках

**Классификация по техникам и подходам**

**Позитивное тестирование** (positive testing147) направлено на исследование приложения в ситуации, когда все действия выполняются строго по инструкции без каких бы то ни было ошибок, отклонений, ввода неверных данных и т.д. Если позитивные тест-кейсы завершаются ошибками, это тревожный признак — приложение работает неверно даже в идеальных условиях (и можно предположить, что в неидеальных условиях оно работает ещё хуже). Для ускорения тестирования несколько позитивных тест-кейсов можно объединять (например, перед отправкой заполнить все поля формы верными значениями) — иногда это может усложнить диагностику ошибки, но существенная экономия времени компенсирует этот риск.

**Негативное тестирование** (negative testing148 , invalid testing149) — направлено на исследование работы приложения в ситуациях, когда с ним выполняются (некорректные) операции и/или используются данные, потенциально приводящие к ошибкам (классика жанра — деление на ноль). Поскольку в реальной жизни таких ситуаций значительно больше (пользователи допускают ошибки, злоумышленники осознанно «ломают» приложение, в среде работы приложения возникают проблемы и т.д.), негативных тест-кейсов оказывается значительно больше, чем позитивных (иногда — в разы или даже на порядки). В отличие от позитивных негативные тест-кейсы не стоит объединять, т.к. подобное решение может привести к неверной трактовке поведения приложения и пропуску (необнаружению) дефектов.

**Тестирование на основе опыта тестировщика, сценариев, чек-листов**

(некоторые авторы называют это) **Классификация по степени формализации**

Тестирование на основе тест-кейсов (scripted testing, test case based testing)— формализованный подход, в котором тестирование производится на основе заранее подготовленных тест-кейсов, наборов тест-кейсов и иной документации.

Пример: у вас есть четкие требования, вы написали тестовую документацию с конкретными шагами и проверками и приступаете к тестированию.

Исследовательское тестирование (exploratory testing) — тестировщик выполняет работу с приложением по выбранному сценарию, который, в свою очередь, дорабатывается в процессе выполнения с целью более полного исследования приложения. Ключевым фактором успеха при выполнении исследовательского тестирования является именно работа по сценарию, а не выполнение разрозненных бездумных операций. Существует даже специальный сценарный подход, называемый сессионным тестированием (session-based testing). Дополнительную информацию об исследовательском тестировании можно получить из статьи Джеймса Баха «Что такое исследовательское тестирование?

Пример: у вас есть требования или общее понимание работы продукта, вы набросали первичные проверки, которые будете дорабатывать по ходу тестирования.

Свободное (интуитивное) тестирование (ad hoc testing159) — полностью неформализованный подход, в котором не предполагается использования ни тест-кейсов, ни чек-листов, ни сценариев — тестировщик полностью опирается на свой профессионализм и интуицию (experience-based testing160) для спонтанного выполнения с приложением действий, которые, как он считает, могут обнаружить ошибку. Этот вид тестирования используется редко и исключительно как дополнение к полностью или частично формализованному тестированию в случаях, когда для исследования некоторого аспекта поведения приложения (пока?) нет тест-кейсов.

Пример: у вас могут быть требования или полностью отсутствовать, вы не создаете набросок тестирования или официальную документацию, а тестируете приложениe основываясь только на своем опыте.

**Классификация по степени вмешательства в работу приложения:**

Инвазивное тестирование (intrusive testing218) — тестирование, выполнение которого может повлиять на функционирование приложения в силу работы инструментов тестирования (например, будут искажены показатели производительности) или в силу вмешательства (level of intrusion219) в сам код приложения (например, для анализа работы приложения было добавлено дополнительное протоколирование, включён вывод отладочной информации и т.д.). Некоторые источники рассматривают220 инвазивное тестирование как форму негативного{82} или даже стрессового{92} тестирования.

Неинвазивное тестирование (nonintrusive testing221) — тестирование, выполнение которого незаметно для приложения и не влияет на процесс его обычной работы.

**Классификация по техникам автоматизации:**

Тестирование под управлением данными (data-driven testing222) — способ разработки автоматизированных тест-кейсов, в котором входные данные и ожидаемые результаты выносятся за пределы тесткейса и хранятся вне его — в файле, базе данных и т.д.

Тестирование под управлением ключевыми словами (keyworddriven testing) — способ разработки автоматизированных тест-кейсов, в котором за пределы тест-кейса выносится не только набор входных данных и ожидаемых результатов, но и логика поведения тесткейса, которая описывается ключевыми словами (командами).

Тестирование под управлением поведением (behavior-driven testing224) — способ разработки автоматизированных тест-кейсов, в котором основное внимание уделяется корректности работы бизнес-сценариев, а не отдельным деталям функционирования приложения.

**Классификация на основе (знания) источников ошибок:**

Тестирование предугадыванием ошибок (error guessing) — техника тестирования, в которой тесты разрабатываются на основе опыта тестировщика и его знаний о том, какие дефекты типичны для тех или иных компонентов или областей функциональности приложения. Может комбинироваться с техникой т.н. «ошибка ориентированного» тестирования (failure-directed testing), в котором новые тесты строятся на основе информации о ранее обнаруженных в приложении проблемах.

Эвристическая оценка (heuristic evaluation) — техника тестирования удобства использования{88} , направленная на поиск проблем в интерфейсе пользователя, представляющих собой отклонение от общепринятых норм.

Мутационное тестирование (mutation testing) — техника тестирования, в которой сравнивается поведение нескольких версий одного и того же компонента, причём часть таких версий может быть специально разработана с добавлением ошибок (что позволяет оценить эффективность тест-кейсов — качественные тесты обнаружат эти специально добавленные ошибки). Может комбинироваться со следующим в этом списке видом тестирования (тестированием добавлением ошибок).

Тестирование добавлением ошибок (error seeding) — техника тестирования, в которой в приложение специально добавляются заранее известные, специально продуманные ошибки с целью мониторинга их обнаружения и устранения и, таким образом, формирования более точной оценки показателей процесса тестирования. Может комбинироваться с предыдущим в этом списке видом тестирования (мутационным тестированием).

**Классификация 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.

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

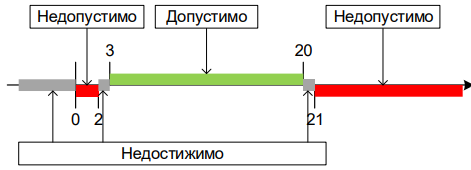
В качестве пояснения идеи рассмотрим тривиальный пример. Допустим, нам нужно протестировать функцию, которая определяет, корректное или некорректное имя ввёл пользователь при регистрации. Требования к имени пользователя таковы:

• От трёх до двадцати символов включительно.

• Допускаются цифры, знак подчёркивания, буквы английского алфавита в верхнем и нижнем регистрах.

Если попытаться решить задачу «в лоб», нам для позитивного тестирования придётся перебрать все комбинации допустимых символов длиной [3, 20] (это 18- разрядное 63-ричное число, т.е. 2.4441614509104E+32). А негативных тест-кейсов здесь и вовсе будет бесконечное количество, ведь мы можем проверить строку длиной в 21 символ, 100, 10000, миллион, миллиард и т.д.

Представим графически классы эквивалентности относительно требований к длине

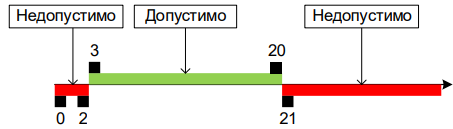


Поскольку для длины строки невозможны дробные и отрицательные значения, мы видим три недостижимых области, которые можно исключить (отмечены серым), и получаем окончательный вариант. Мы получили три класса эквивалентности:

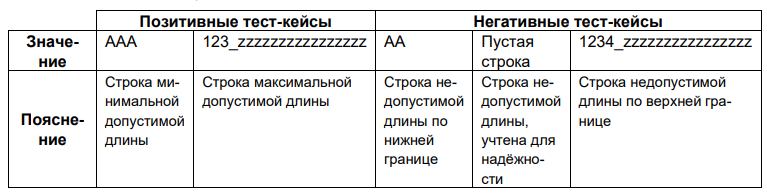
• [0, 2] — недопустимая длина;

• [3, 20] — допустимая длина;

• [21, ∞] — недопустимая длина.



Обратите внимание, что области значений [0, 2] и [21, ∞] относятся к разным классам эквивалентности, т.к. принадлежность длины строки к этим диапазонам проверяется отдельными условиями на уровне кода программы. Граничные условия уже отмечены на рисунке — это 2, 3, 20 и 21. Значение 0 тоже стоит включить в этот набор на всякий случай, т.к. в программировании ноль, NULL, нулевой байт и т.п. исторически являются «опасными значениями». В итоге мы получаем следующий набор входных данных для тест-кейсов (сами символы для составления строк можно выбирать из набора допустимых символов случайным образом, но желательно учесть все типы символов, т.е. буквы в обоих регистрах, цифры, знак подчёркивания).

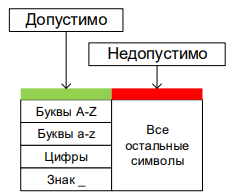


Значения входных данных для тест-кейсов (реакция на длину имени пользователя

Осталось решить вопрос с недопустимыми символами. К сожалению, столь же наглядно, как с длиной, здесь не получится. Даже если подойти строго научно, т.е. выбрать кодировку и по её кодовой таблице определить диапазоны кодов символов, у нас нет никакой гарантии, что символы с кодами из каждого диапазона трактуются единообразно.

Здесь мы видим ярчайший пример случая, в котором тестирование по методу белого ящика сильно облегчило бы нам жизнь. Если бы мы видели, как в коде приложения реализована проверка на допустимые и недопустимые символы, мы могли бы подобрать очень показательные значения входных данных.

Раз оказалось, что по кодам символов подбирать классы эквивалентности в нашем случае нерационально, посмотрим на ситуацию по-другому (и намного проще). Поделим символы на недопустимые и допустимые, а последние, в свою очередь, — на группы



Интересующие нас комбинации допустимых символов (с представителями всех групп) мы уже учли при проверке реакции приложения на имена пользователя допустимых и недопустимых длин, потому остаётся учесть только вариант с допустимой длиной строки, но недопустимыми символами (которые можно выбирать случайным образом из соответствующего набора). Добавим ещё одну колонку к нашей таблице



Конечно, в случае критически важных приложений (например, системы управления ядерным реактором) мы бы проверили с помощью средств автоматизации реакцию приложения на каждый недопустимый символ. Но предположив, что перед нами некое тривиальное приложение, мы можем считать, что одной проверки на недопустимые символы будет достаточно.

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.

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

Доменное тестирование (domain analysis232 , 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).

ИЛИ ДРУГИМИ СЛОВАМИ:

Доменное тестирование (domain testing, domain analysis353) — техника создания эффективных и результативных тест-кейсов в случае, когда несколько переменных могут или должны быть протестированы одновременно.

Данная техника также описывает подходы к выбору минимального множества показательных тест-кейсов из всего набора возможных тест-кейсов.

Попарное тестирование (pairwise testing233) — 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 testing234) 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 testing236) — инструментальная техника попарного и Nкомбинационного 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)

**Классификация на основе среды выполнения:**

Тестирование в процессе разработки (development testing239) — тестирование, выполняемое непосредственно в процессе разработки приложения и/или в среде выполнения, отличной от среды реального использования приложения. Как правило, выполняется самими разработчиками.

Операционное тестирование (рассмотрено ранее ).

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

Тестирование по потоку управления (control flow testing240) — семейство техник тестирования, в которых тест-кейсы разрабатываются с целью активации и проверки выполнения различных последовательностей событий, которые определяются посредством анализа исходного кода приложения. Дополнительное подробное пояснение см. дальше в этом разделе (см. тестирование на основе структур кода{97} ).

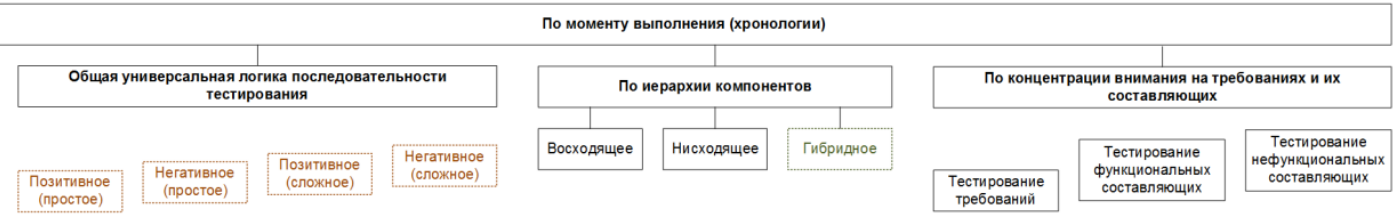
Тестирование по потоку данных (data-flow testing241) — семейство техник тестирования, основанных на выборе отдельных путей из потока управления с целью исследования событий, связанных с изменением состояния переменных. Дополнительное подробное пояснение см. дальше в этом разделе (в части, где тестирование по потоку данных пояснено с точки зрения стандарта ISO/IEC/IEEE 29119-4{108} ).

Тестирование по диаграмме или таблице состояний (state transition testing242) — техника тестирования, в которой тест-кейсы разрабатываются для проверки переходов приложения из одного состояния в другое. Состояния могут быть описаны диаграммой состояний (state diagram243) или таблицей состояний (state table244).

Инспекция (аудит) кода (code review, code inspection247) — семейство техник повышения качества кода за счёт того, что в процессе создания или совершенствования кода участвуют несколько человек. Степень формализации аудита кода может варьироваться от достаточно беглого просмотра до тщательной формальной инспекции. В отличие от техник статического анализа кода (по потоку управления и потоку данных) аудит кода также улучшает такие его характеристики, как понятность, поддерживаемость, соответствие соглашениям об оформлении и т.д. Аудит кода выполняется в основном самими программистами.

ДАЛЬШЕ НЕ СТАЛ РАЗБИРАТЬ, УЖ СЛИШКОМ ЗАНУДНО.





**или**

Все виды тестирования можно группировать **по разным критериям:**

****

-По целям: **функциональное, нефункциональное** (производительность, UI, usability, совместимость, локализация, …)



-Способ проведения / степень автоматизации: **ручное**, **автоматизированное**.

-Позитивность сценария: **позитивное**, **неготивное**.

-Доступность к коду: **белый, чёрный, серый ящик.**

-уровень (обьект): **модульное**(unit), **интеграционное**, **системное**, **приёмочное** (UAT). Модульное осуществляют сами разраб. интеграционное это как отдельно написанные модули работают в связке, системное это как вся система, когда она собрана, работает целиком. Приёмочное это уже когда сам заказчик тестирует и принимает работу, или нет ))).

-Исполнитель (субьект): **альфа**, **бета**. Альфа - это тестирование на пользователях, но не большого, ограниченного круга, возможно с работниками компании которые не имеют отношения к программированию. Бета тестирование это всё тоже самое но с большим количеством людей, чем больше людей тем больше вероятность нахождения бага.

-Формальность: **По тестам** / **Explority** (исследовательское), **Ad-hoc** (свободное).

-Характер изменений: **smoke**, **re-test**, **regression**.