GeekBrains

Архитектура ПО

Методичка к уроку 12

Подходы и принципы тестирования приложений

Оглавление

Введение

Тестирование программного обеспечения — это процесс оценки и проверки того, что программный продукт или приложение делает то, что должно делать. Преимущества тестирования включают в себя предотвращение ошибок, снижение затрат на разработку и повышение производительности.

Модель качества является краеугольным камнем системы оценки качества продукта. Модель качества определяет, какие характеристики качества будут приниматься во внимание при оценке свойств программного продукта.

Качество системы — это степень, в которой система удовлетворяет заявленные и подразумеваемые потребности различных заинтересованных сторон и, таким образом, обеспечивает ценность. Потребности заинтересованных сторон (функциональность, производительность, безопасность, ремонтопригодность и т. д.) — это именно то, что представлено в модели качества, которая классифицирует качество продукта по характеристикам и подхарактеристикам.

Модель качества продукции, определённая в стандарте ISO/IEC 25010, включает восемь характеристик качества, показанных на следующем рисунке:

- **Функциональная пригодность**

Эта характеристика представляет собой степень, в которой продукт или система обеспечивают функции, отвечающие заявленным и подразумеваемым потребностям при использовании в определённых условиях. Эта характеристика состоит из следующих подхарактеристик:

- **Эффективность работы**

Эта характеристика представляет собой производительность относительно количества ресурсов, используемых в указанных условиях. Эта характеристика состоит из следующих подхарактеристик:

Тестирование проводится с учётом конкретных целей, которые сформулированы более или менее чётко и с разной степенью точности. Изложение целей тестирования в точных, количественных терминах способствует измерению и контролю процесса тестирования.

Тестирование может быть направлено на проверку различных свойств. Тестовые случаи могут быть разработаны для проверки правильности реализации

функциональных спецификаций, что в литературе называют тестированием на соответствие, тестированием на правильность или функциональным тестированием. Однако можно протестировать и некоторые другие нефункциональные свойства, включая производительность, надёжность и удобство использования, среди многих других (см. раздел «Модели и характеристики качества» в КА по качеству программного обеспечения).

Другие важные цели тестирования включают, но не ограничиваются измерением надёжности, выявление уязвимостей безопасности, оценка удобства использования и приёмка программного обеспечения, для которых применяются различные подходы. Обратите внимание, что в целом цели тестирования зависят от цели тестирования; разные цели решаются на разных уровнях тестирования.

На этом уроке

- Узнаем, что такое надёжность и доступность ПО.
- Разберём понятие качество ПО.
- Узнаем разницу между верификацией, валидацией и тестированием ПО.
- Разберём концепции и виды тестирования ПО.
- Рассмотрим некоторые основные паттерны проверки, тестирования и испытания ПО.
- Изучим основную терминологию тестирования ПО.

Термины, используемые в лекции

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее представляет собой область человеческой деятельности, которую система поддерживает.

Поведение во времени — Степень соответствия времени отклика и обработки, а также пропускной способности продукта или системы при выполнении своих функций требованиям.

Использование ресурсов — Степень соответствия количества и типов ресурсов, используемых продуктом или системой при выполнении своих функций, требованиям.

Пропускная способность — степень соответствия максимальных пределов параметров продукта или системы требованиям.

Функциональная полнота — Степень, с которой набор функций покрывает все указанные задачи и цели пользователя.

Функциональная корректность — Степень, с которой продукт или система обеспечивает правильные результаты с необходимой степенью точности.

Функциональная уместность — Степень, в которой функции способствуют выполнению определённых задач и целей.

Надёжность и доступность ПО и информационных систем

Понятия надёжности и доступности

Надёжность — степень того, насколько пользователи могут положиться на продукт или сервис для решения своих задач. Распределённый продукт или сервис считается надёжным, если он продолжает функционировать, даже когда один или несколько её программных или аппаратных компонентов выходят из строя.

Доступность — степень готовности продукта или сервиса к эксплуатации по требованию. Это понятие часто сводят к обеспечению необходимого излишка ресурсов с учётом статистически независимых отказов. Понятие доступности в широком смысле можно трактовать как гарантию безотказной работы в течение определённого времени.

Если сервис неспособен безотказно работать по требованию, значит, ему не хватает доступности.

Отказ — неспособность узла, компонента или системы выполнять свои функции.

Независимые и коррелированные отказы

Обычно отказы рассматриваются, как случайно возникающие события.

- **Независимые отказы системы.**

Предположим, что автомобиль с некоторой вероятностью может сломаться во время поездки. Часто также предполагается, что отказы являются статистически независимыми событиями. Это означает, что поломка двигателя в одном автомобиле не увеличивает вероятность поломки другого автомобиля на дороге.

- **Коррелированные отказы систем**

Коррелированный отказ — это событие, при котором это предположение о статистической независимости не выполняется.

Если первый автомобиль вышел из строя из-за дефектного двигателя, вы больше не можете считать, что отказ независим, поскольку существует некоторая вероятность того, что в другие автомобили на дороге тоже залили такое же плохое масло. Это может привести к тому, что многие автомобили выйдут из

строя вместе, в течение короткого периода времени, и по одной и той же основной причине. Эти отказы уже не независимые, а коррелированные.

Коррелированные отказы могут привести к тому, что система хранения данных, которая может иметь высокую степень избыточности или репликации, всё равно выйдет из строя.

Устойчивость — способность продукта или сервиса соответствовать заявленным характеристикам в процессе использования. Это значит, что система не просто готова к эксплуатации: благодаря дополнительным мощностям, предусмотренным в ходе проектирования, она может продолжать работать под нагрузкой, как и ожидают пользователи.

Также устойчивость определяет качество работы сервиса, то есть гарантирует максимально эффективное сохранение функциональности и возможности взаимодействовать с пользователем в неблагоприятных условиях.

Если сервис готов к работе, но при этом отклоняется от заявленных характеристик в процессе использования, значит, **не хватает устойчивости**.

Доступность и устойчивость являются видами надёжности

Отказоустойчивость — способность системы сохранять надёжность (доступность и устойчивость) при отказе отдельных компонентов или сбоях в подсистемах.

Отказоустойчивые системы спроектированы таким образом, чтобы смягчать воздействие неблагоприятных факторов и оставаться надёжными для конечного пользователя. Методы обеспечения отказоустойчивости могут использоваться для улучшения доступности и устойчивости.

Понятие **доступности** в широком смысле можно трактовать как гарантию безотказной работы в течение определённого времени. В свою очередь, устойчивость определяет качество этой работы, то есть гарантирует максимально эффективное сохранение функциональности и возможности взаимодействовать с пользователем в неблагоприятных условиях.

Если сервис **не способен безотказно работать по требованию**, значит, ему **не хватает доступности**. А если он готов к работе, но при этом отклоняется от заявленных характеристик в процессе использования, значит, не хватает устойчивости. Методы проектирования отказоустойчивых систем направлены на устранение этих недостатков и обеспечение бесперебойной работы системы как для бизнеса, так и для отдельных пользователей.

Показатели надёжности технических средств

Основными техническими средствами информационных систем являются:

- ЭВМ (состоящая из системного блока, монитора, клавиатуры и мыши);
- периферийные устройства (принтеры, сканеры);
- сетевая инфраструктура
    - пассивное оборудование: кабельная система;
    - активное оборудование: модемы, коммутаторы, маршрутизаторы;
- система электропитания;
- система охлаждения;
- система вентиляции;
- специальное оборудование;
- и т. п.

Некоторые из указанных устройств в случае выхода из строя не ремонтируются, а заменяются на исправные, т. е. являются **невосстанавливаемыми**.

Другие устройства частично ремонтируются, например, замена неисправной материнской платы в системном блоке ЭВМ приводит к восстановлению ЭВМ. В целом, информационная система относится к **восстанавливаемым** системам.

Рассмотрим с точки зрения надёжности невосстанавливаемые устройства информационных систем.

Пусть T — это наработка (продолжительность работы или время безотказной работы) технического средства до отказа, т. е. интервал времени от начала работы до первого отказа. Тогда к основным показателям надёжности технических устройств ИС можно отнести:

- Вероятность безотказной работы.
- Вероятность отказа.
- Частота отказов.
- Среднее время наработки до первого отказа.

Указанные показатели надёжности позволяют достаточно полно оценить надёжность невосстанавливаемых технических средств, а также надёжность восстанавливаемых технических средств до первого отказа.

Наиболее полно надёжность ТС характеризуется частотой отказов. Это объясняется тем, что частота отказов является плотностью распределения, а

поэтому содержит в себе всю информацию о случайном явлении — времени безотказной работы.

Средняя наработка до первого отказа — является достаточно наглядной характеристикой надёжности, однако применение этого показателя ограничено в тех случаях, когда интенсивность отказов непостоянна или время работы отдельных частей ТС разное.

Интенсивность отказов — наиболее удобная характеристика надёжности, так как она позволяет вычислить остальные показатели. Наиболее практичным показателем надёжности ТС является вероятность безотказной работы, поскольку может быть сравнительно просто оценена в процессе эксплуатации.

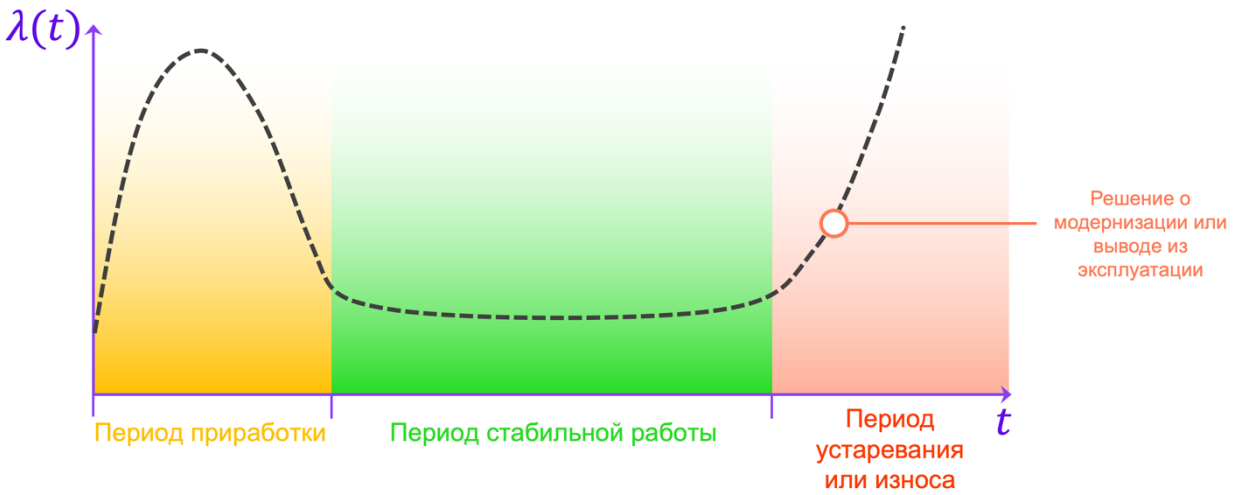
Вероятность безотказной работы технического средства — это вероятность того, что при заданных условиях эксплуатации в течение интервала времени (0,t) не возникнет ни одного отказа, т. е. средство будет работоспособно:

$$P(t) = P(T \geq t)$$

Вероятность отказа технического средства — это вероятность того, что при заданных условиях эксплуатации в течение интервала времени (0,t) произойдёт хотя бы один отказ, т. е. вероятность того, что время безотказной работы T меньше произвольного t:

$$Q(t) = P(T < t) = 1 - P(t)$$

Для многих важных случаев интенсивность отказов имеет вид, приведённый на рисунке:



Функция интенсивности отказов.

Защитное программирование

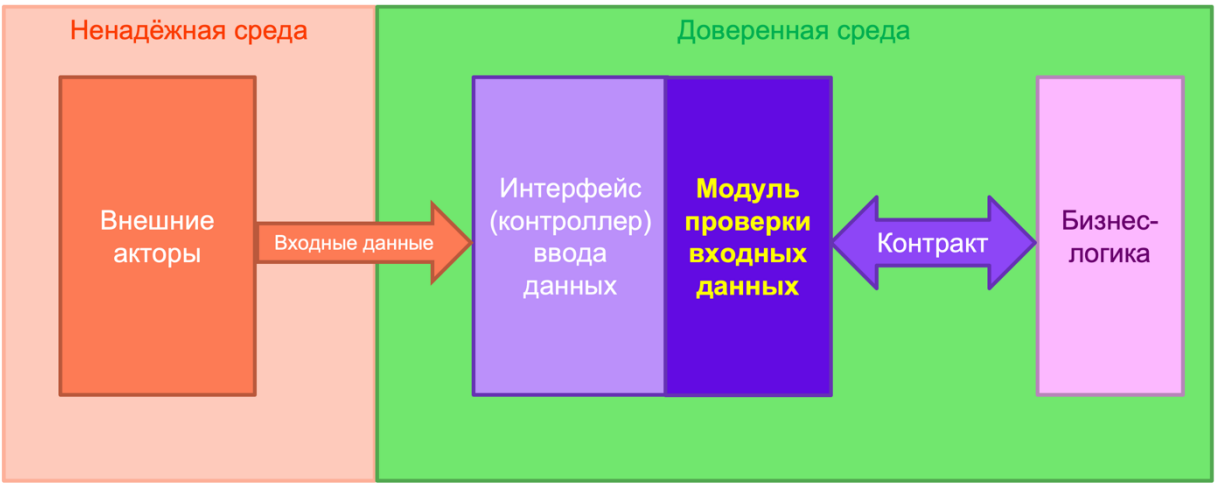
Вопросы обеспечения безопасности и работоспособности системы являются неотъемлемой частью этапа её проектирования. Требования к безопасности конкретных продуктов и систем ИТ устанавливаются исходя из имеющихся и прогнозируемых угроз безопасности, проводимой политики безопасности, а также с учётом условий их применения. Внедрение решений для обеспечения безопасности после того, как система уже разработана, является сложной и дорогостоящей процедурой. Поэтому следует с самого начала учитывать требования к безопасности на протяжении всего жизненного цикла системы.

Defensive programming (Оборонительное, защитное, безопасное программирование) — принцип разработки ПО, при котором разработчики пытаются учесть все возможные ошибки и сбои, максимально изолировать их и при возможности восстановить работоспособность программы в случае неполадок. Это должно делать программное обеспечение более стабильным и менее уязвимым. Например, аппаратной реализацией этого принципа является сторожевой таймер, вычисление контрольной суммы — для выявления ошибок при пакетной передаче данных.

Secure coding (Безопасное программирование) — методика написания программ, устойчивых к атакам со стороны вредоносных программ и злоумышленников. Безопасное программирование помогает защитить данные пользователя от кражи или порчи. Кроме того, небезопасная программа может предоставить злоумышленнику доступ к управлению сервером или компьютером пользователя; последствия могут быть различны: от отказа в обслуживании одному пользователю до компрометации секретной информации, потери обслуживания или повреждения систем тысяч пользователей.

Проверка входных данных

Проектирование по контракту предназначено для формализации взаимоотношения двух программных элементов внутри доверенной среды и не предназначено для взаимодействия программного элемента с внешним миром. Главный принцип защищённого кода «все входные данные зловредны, пока не доказано противное» остаётся в силе, и о нём не следует забывать при разработке любых приложений, независимо от того, построены они по принципам проектирования по контракту или без него.



Доверенная и ненадёжная среда

В защитном программировании существует негласное правило, согласно которому «лишняя проверка никогда не повредит», и действительно, если рассмотреть простой пример (например, функцию извлечения квадратного корня double Sqrt(double x)) может показаться, что никакого вреда от дополнительной проверки нет и быть не может, но так ли это на самом деле?

Существует три основных способа обработки входных мусорных данных, перечисленных далее.

Проверяйте все данные из внешних источников

Получив данные из файла, от пользователя, из сети или любого другого внешнего интерфейса, удостоверьтесь, что все значения попадают в допустимый интервал. Проверьте, что числовые данные имеют разрешённые значения, а строки достаточно коротки, чтобы их можно было обработать. Если строка должна содержать определённый набор значений (скажем, идентификатор финансовой транзакции или что-то подобное), проконтролируйте, что это значение допустимо в данном случае, если же нет – отклоните его. Если вы работаете над приложением, требующим соблюдения безопасности, будьте особенно осмотрительны с данными, которые могут атаковать вашу систему: попыткам переполнения буфера, внедрённым SQL-командам, внедрённому HTML- или XML-коду, переполнения целых чисел, данным передаваемым системным вызовам и т. п.

Проверяйте значение всех входных параметров метода

Проверка значений входных параметров метода практически тоже самое, что и проверка данных из внешнего источника, за исключением того, что все данные

поступают из другого метода, а не из внешнего интерфейса. В разделе 8.5 вы узнаете, как определить, какие методы должны проверять свои входные данные.

Решите, как обрабатывать неправильные входные данные

Что делать, если вы обнаружили неверный параметр?

Защитное программирование — это встраивание отладочных средств в программу.

Защита программы от неправильных входных данных

В защитном программировании главная идея в том, что если методу передаются некорректные данные, то его работа не нарушится, даже если эти данные испорчены по вине другой программы. Обобщая, можно сказать, что в программах всегда будут проблемы, программы будут модифицироваться и разумный программист будет учитывать это при разработке кода.

Виды проверки качества ПО

Тестирование

Тестирование программного обеспечения — это проверка того, что программа ведёт себя ожидаемо на конечном множестве тестовых сценариев, которые были выбраны подходящим образом из обычно большого (иногда бесконечного) количества вариантов. Такая проверка проводится динамическим способом.

Динамическая проверка означает, что тестирование всегда проводится как выполнение программы на выбранных входных данных.

Это значит, одного входного значения не всегда достаточно для проведения теста, так как сложная недетерминированная система может реагировать на один и тот же входной сигнал по-разному, в зависимости от состояния системы.

Статические методы отличаются от динамического тестирования и дополняют его. Статические методы в большей степени относятся к понятию Качество программного обеспечения. Важно отметить, что терминология не является единой в различных сообществах, и некоторые используют термин «тестирование» также применительно к статическим методам.

Верификация и Валидация

Верификация — это процесс проверки того, что программное обеспечение достигает своей цели без каких-либо ошибок. Это процесс, позволяющий убедиться в том, что разработанный продукт правильный или нет. Он проверяет, соответствует ли разработанный продукт требованиям, которые мы предъявляем.

Верификация в тестировании программного обеспечения — это процесс проверки документов, дизайна, кода и программы с целью проверить, было ли программное обеспечение создано в соответствии с требованиями или нет. Основной целью процесса верификации является обеспечение качества программного приложения, дизайна, архитектуры и т. д. Это относительно объективный процесс.

Верификация поможет определить, является ли программное обеспечение высококачественным, но она не гарантирует, что система будет полезной. Верификация связана с тем, хорошо ли спроектирована система и не содержит ли она ошибок.

Методы верификации: статическое тестирование

- Проходное тестирование (проверка наличия).
- Проверка документации, кода, сборок по формальным признакам.
- Обзор разработанных функций.

Верификация означает, правильно ли мы создаём продукт?

Валидация — это процесс проверки соответствия программного продукта требованиям или, другими словами, соответствия продукта требованиям высокого уровня. Это процесс проверки валидности продукта, т. е. проверка того, что мы разрабатываем правильный продукт. Это проверка фактического и ожидаемого продукта.

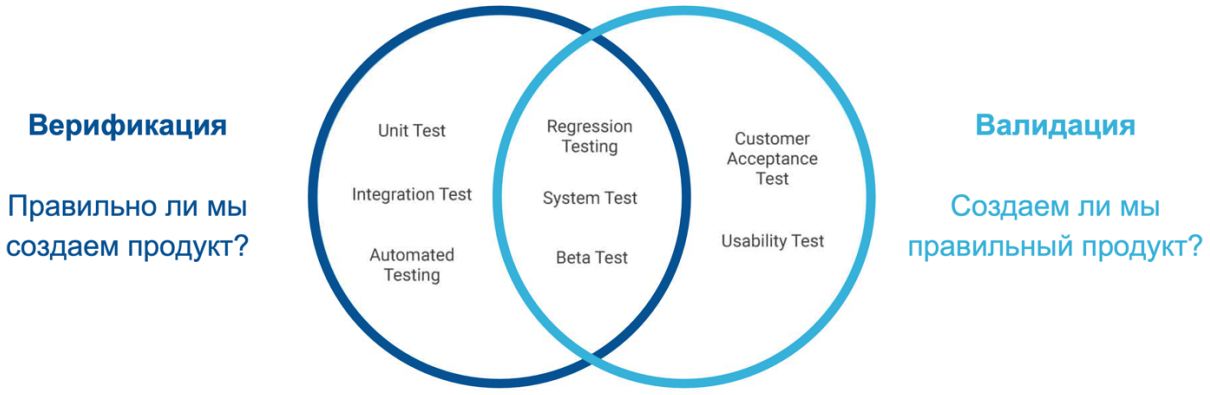
Валидация — это динамическое тестирование.

Валидация означает, создаём ли мы правильный продукт?

Разница между верификацией и валидацией заключается в следующем:

Верификация	Валидация
Она включает проверку документов, дизайна, кодов и программ.	Она включает в себя тестирование и проверку фактического продукта.
Верификация - это статическое тестирование.	Валидация - это динамическое тестирование.
Оно не включает выполнение кода.	Она включает выполнение кода.
Методы, используемые при проверке, - это обзоры, проходы, инспекции и камеральные проверки.	Методы, используемые при проверке, - это тестирование "черного ящика", тестирование "белого ящика" и нефункциональное тестирование.
Проверяется, соответствует ли программное обеспечение спецификациям или нет.	Проверяется, соответствует ли программное обеспечение требованиям и ожиданиям заказчика или нет.
Оно может найти ошибки на ранней стадии разработки.	Она может найти только те ошибки, которые не были обнаружены в процессе верификации.
Целью верификации является архитектура и спецификация приложения и программного обеспечения.	Цель валидации - реальный продукт.
Верификацию выполняет команда обеспечения качества.	Валидация выполняется на программном коде с помощью команды тестирования.
Верификация проводится перед валидацией.	Валидация проводится после верификации.
Она состоит из проверки документов/файлов и выполняется человеком.	Она состоит из выполнения программы и выполняется компьютером.

Пересечение ответственности верификации и валидации.



Пример верификации и валидации

Рассмотрим следующую спецификацию для верификационного тестирования и валидационного тестирования

- Кликабельная кнопка синего цвета с именем «Отправить»

Верификация будет проверять проектную документацию и исправлять орфографические ошибки.

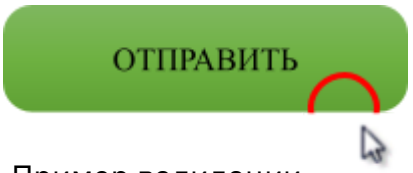
Пример верификации

В противном случае команда разработчиков создаст кнопку другого типа.

Реализация в коде:

- Кликабельная кнопка синего цвета с именем «Отправить», нажимается

Как только код готов, проводится валидация.



Пример валидации

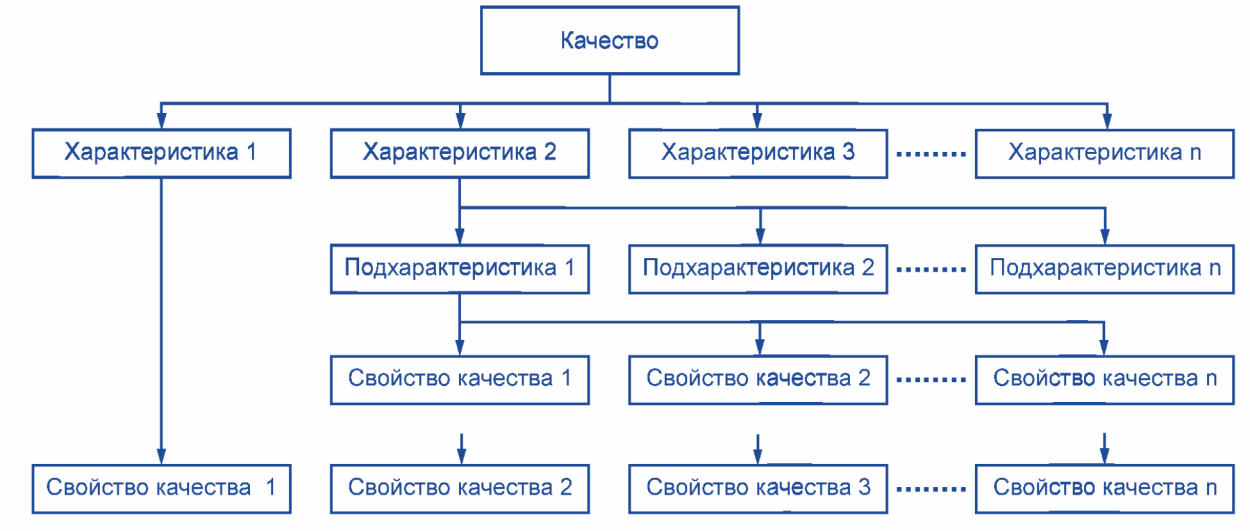
Тест на валидацию обнаружил — что *кнопка не нажимается*

Благодаря валидационному тестированию команда разработчиков сделает кнопку отправки кликабельной.

Понятие качества ПО

Согласно: ГОСТ Р ИСО/МЭК 25010-2015. **Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.**

Качество системы — это степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон, которая позволяет, таким образом, оценить достоинства.



Структура модели качества

Модель качества продукта

Модель качества продукта сводит свойства качества системы/программного продукта к восьми характеристикам, которыми являются:

- Функциональная пригодность.
- Уровень производительности.
- Совместимость.
- Удобство пользования.
- Надёжность.
- Защищённость.
- Сопровождаемость.
- Переносимость (мобильность).

Каждая характеристика, в свою очередь, состоит из ряда соответствующих подхарактеристик.

Модель качества продукта

Модель качества продукта можно применять как для программного продукта, так и для компьютерной системы, в состав которой входит программное обеспечение, поскольку большинство подхарактеристик применимо и к программному обеспечению, и к системам.

Обеспечение качества (англ. quality assurance, QA) — это процесс или результат формирования требуемых свойств и характеристик продукции по мере её создания, а также — поддержание этих характеристик при хранении, транспортировании и эксплуатации продукции.

Обеспечение качества, в сравнении с тестированием, является более широким понятием. QA помогает оценить правильность протекания технологических процессов на всех этапах разработки ПО для обеспечения его высокого качества.

Кроме тестирования, QA также включает в себя контроль качества (**QC — quality control**), который отвечает за соблюдение предъявляемых к системе требований.

Если представить все три термина в виде иерархии, то **тестирование окажется частью QC, а QC — частью QA**.

Таким образом, тестирование заключается в большей степени в проверке работоспособности программного продукта и поиске дефектов, в то время как для QA важно также обеспечить соблюдение стандартов и предотвратить появление ошибок и багов в ПО.

Quality Assurance	Quality Control	Тестирование
Комплекс мероприятий, который охватывает все технологические аспекты на всех этапах разработки, выпуска и введения в эксплуатацию программных систем для обеспечения необходимого уровня качества программного продукта	Процесс контроля соответствия разрабатываемой системы предъявляемым к ней требованиям	Процесс, отвечающий непосредственно за составление и прохождение тест-кейсов, нахождение и локализацию дефектов и т.д.
Фокус в большей степени на процессы и средства, чем на непосредственно исполнение тестирования системы	Фокус на исполнение тестирования путем выполнения программы с целью определения дефектов с использованием утвержденных процессов и средств	Фокус на исполнение тестирования как такового
Процессно-ориентированный подход	Продуктно-ориентированный подход	Продуктно-ориентированный подход
Превентивные меры	Корректирующий процесс	Превентивный процесс
Подмножество процессов Software Test Life Cycle – цикла тестирования ПО	Подмножество процессов QA	Подмножество процессов QC

Виды тестирования программного обеспечения

Цель тестирования

Существует множество различных типов тестирования программного обеспечения, каждый из которых имеет определённые цели и стратегии.

Даже простое приложение может быть подвергнуто большому количеству и разнообразию тестов. План управления тестированием помогает определить приоритеты, какие виды тестирования обеспечивают наибольшую ценность — с учётом имеющегося времени и ресурсов. Эффективность тестирования оптимизируется путём проведения наименьшего количества тестов для выявления наибольшего количества дефектов.

Концепции тестирования

Тестирование программного обеспечения обычно проводится на разных уровнях в течение всего процесса разработки и сопровождения. Уровни можно различать по объекту тестирования, который называется целью, или по назначению, которое называется задачей тестирования.

The Marick Test Matrix «Матрица тестирования Брайана Марика»

Брайан Марик (Brian Marick) предоставил систему распределения тестов по категориям рисунок.

Группировка тестов по двум основаниям

- *К чему относится тест — к бизнесу или технологиям.* Бизнес-тест описывается в терминологии специалиста проблемной области, тогда как для описания технологического теста используется терминология разработчиков и реализации.
- *Какова цель теста* — помочь с написанием кода или дать оценку приложению.

Тестовый квадрант определяет четыре категории тестов:

- Q1 — помощь в программировании с ориентацией на технологии — модульные и интеграционные тесты, могут быть автоматизированными;
- Q2 — помощь в программировании с ориентацией на бизнес-компонентные и сквозные тесты. Эти тесты либо ручные, либо автоматизированные;

- Q3 — оценка приложения с точки зрения бизнеса — проверка удобства использования и исследовательское тестирование, уровень приемлемости системы или пользователя, бизнес-ориентация и ориентация на сценарии в реальном времени. Пользовательские приёмочные тесты принадлежат этому квадранту. Эти тесты являются ручными.
- Q4 — оценка приложения с точки зрения технологий — нефункциональное приёмочное тестирование, такое как проверка производительности. Уровень приемлемости системы или эксплуатации, технологический анализ и тестирование производительности, нагрузки, стресса, ремонтопригодности, масштабируемости. Для этих испытаний могут использоваться специальные инструменты, а также тестирование автоматизации.

Помимо тестового квадранта, существуют и другие способы организации тестов.

Пирамида тестов

Пирамида тестов (рисунок) помогает определить, сколько тестов каждого типа следует написать.

На одном конце спектра располагаются модульные тесты для отдельных классов. Они надёжны, просты в написании и быстро выполняются. На противоположном конце находятся сквозные тесты для всего приложения. Они медленные, их сложно писать, а из-за сложности они часто оказываются ненадёжными. Поскольку наш бюджет на разработку и тестирование ограничен, мы хотим сосредоточиться на написании тестов с небольшим охватом, не ставя при этом под угрозу эффективность тестового набора.

Пирамида тестов описывает относительные пропорции типов тестов, которые нужно написать. Продвигаясь вверх, надо писать всё меньше и меньше тестов.

Суть этого подхода заключается в том, что с продвижением вверх по пирамиде мы должны писать все меньше и меньше тестов. Модульных тестов должно быть много, а сквозных — мало.

Модульные тесты (Modul Test)

Это тесты, с помощью которых тестируется, как правило, отдельный вызов функции или метода. Под эту категорию подпадают тесты, созданные в рамках концепции разработки под контролем тестирования (Test-Driven Design (TDD)), а также разновидности тестов, созданных с помощью такой технологии, как тестирование на основе свойств (Property-Based Testing). Сам сервис здесь не запускается, мы также ограничены в использовании внешних файлов или сетевых подключений. По сути, тестов такого рода требуется очень много. Если они правильно сделаны, то выполняются очень и очень быстро и можно рассчитывать на выполнение многих тысяч таких тестов меньше чем за минуту на современном оборудовании.

Это тесты, помогающие разработчикам, и поэтому должны иметь технологическую, а не бизнес-направленность. В ходе их проведения мы надеемся отловить основную часть своих ошибок. Итак, в нашем примере, когда мы занимаемся клиентским сервисом, блочные тесты будут охватывать небольшую изолированную часть кода.

Основной целью этих тестов является получение очень быстрых ответных результатов, говорящих о качестве функционирования кода. Тесты могут играть весьма важную роль в поддержке разбиения кода на части, позволяя проводить реструктуризацию кода по мере выполнения работы. При этом мы будем знать, что имеющие весьма ограниченную область действия тесты тут же нас остановят, если будет допущена ошибка.

Модульное тестирование проверяет функционирование в изоляции элементов программного обеспечения, которые поддаются тестированию по отдельности. В зависимости от контекста, это могут быть отдельные подпрограммы или более крупный компонент, состоящий из высокосвязанных единиц. Как правило, модульное тестирование проводится с доступом к тестируемому коду и при поддержке инструментов отладки.

Обычно, но не всегда, модульное тестирование проводят программисты, написавшие код.

Интеграционные тесты

Здесь отдельные программные модули объединяются и тестируются в группе.

Интеграционное тестирование — это процесс проверки взаимодействия между модулями (компонентами) программного обеспечения. Классические стратегии интеграционного тестирования, такие как «сверху вниз» и «снизу вверх», часто используются для иерархически структурированного программного обеспечения.

Современные систематические стратегии интеграции, как правило, определяются архитектурой, что предполагает постепенную интеграцию программных компонентов или подсистем на основе идентифицированной архитектуры.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надёжности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования чёрного ящика. Тестирование чёрного ящика или поведенческое тестирование — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве (коде) тестируемого объекта. Иначе говоря, тестированием чёрного ящика занимаются тестировщики, не имеющие доступ к исходному коду приложения.

Компонентные тесты (тесты сервиса)

Тесты сервиса разрабатываются для того, чтобы в обход пользовательского интерфейса выполнять непосредственное тестирование сервиса. В монолитном приложении могут тестироваться коллекции классов, предоставляющие сервис пользовательскому интерфейсу. В системе, содержащей несколько сервисов, тест сервиса используется для тестирования возможностей отдельного сервиса.

Причина, по которой требуется протестировать отдельно взятый сервис, состоит в улучшении изолированности теста с целью более быстрого обнаружения и устранения проблем. Для достижения изолированности нужно заглушить все внешние сотрудничающие компоненты, чтобы в область действия теста попадал только сам сервис.

Некоторые из этих тестов могут выполняться так же быстро, как и небольшие тесты, но если задумать тестирование с участием реальной базы данных или с переходом по сети к заглушённым нижестоящим сотрудничающим компонентам, время проведения теста может увеличиться. Кроме того, эти тесты имеют более широкую область действия, чем простой блочный тест, поэтому, если тест не будет пройден, причину окажется найти труднее, чем при проведении блочного теста. Тем не менее в их область действия попадает гораздо меньше активных компонентов, чем при проведении широкомасштабного тестирования, поэтому проходят проще.

Сквозные тесты (End-To-End test - E2E)

Сквозные тесты проводятся в отношении всей системы. Зачастую управляют ими через графический интерфейс пользователя, имеющийся у браузера, но с возможностью без каких-либо затруднений имитировать другие виды взаимодействия с пользователем, например, выкладывание файла.

Этими тестами охвачен большой объём кода, предназначенного для работы в производственном режиме. Следовательно, при прохождении этих тестов возникает удовлетворённость: с большой долей уверенности можно сказать, что протестированный код будет работать в производственном режиме. Но этой увеличенной области действия присущи некоторые недостатки, и как мы вскоре увидим, в контексте микросервисов они могут иметь весьма запутанный характер.

Приёмочные тесты (User acceptance test — UAT) или Системное тестирование

Системное тестирование направлено на проверку поведения всей системы. Эффективное модульное и интеграционное тестирование позволит выявить многие дефекты программного обеспечения. Системное тестирование обычно считается подходящим для оценки нефункциональных требований к системе, таких как безопасность, скорость, точность и надёжность (см. функциональные и нефункциональные требования в разделе «Требования к программному обеспечению» и «Требования к качеству программного обеспечения» в разделе «Требования к качеству программного обеспечения»). На этом уровне также

обычно оцениваются внешние интерфейсы с другими приложениями, утилитами, аппаратными устройствами или операционной средой.

Проверка, при которой система проверяется на готовность к приёмке. Цель этого тестирования — оценить соответствие системы бизнес-требованиям и определить, приемлема ли она для поставки.

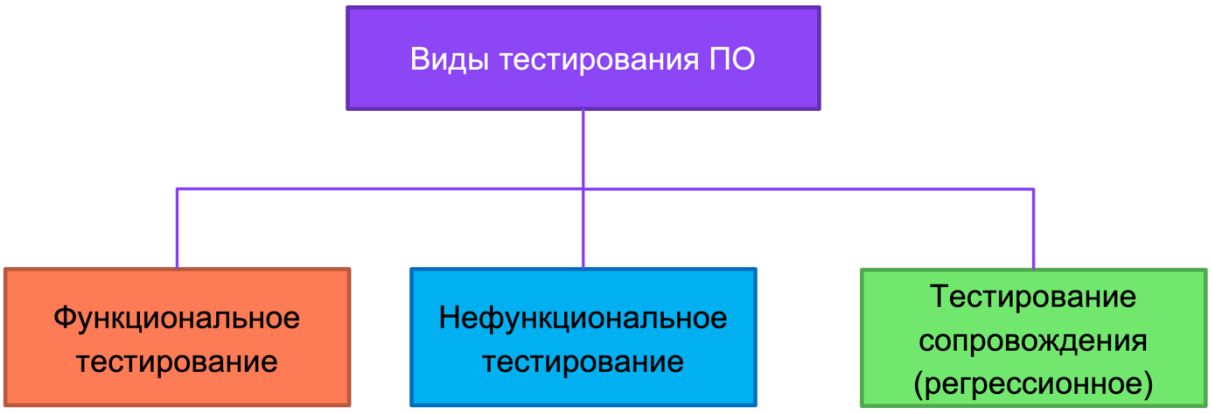
Это тип тестирования, выполняемый конечным пользователем или клиентом для проверки/принятия системы программного обеспечения перед перемещением приложения в производственную среду. UAT выполняется на заключительном этапе тестирования после выполнения других типов тестирования и испытаний.

Основной целью UAT является проверка сквозного бизнес-потока. Он не фокусируется на косметических ошибках, орфографических ошибках или тестировании системы. Приёмочное тестирование пользователя выполняется в отдельной среде тестирования с настройкой данных, аналогичных производственным. Это своего рода тестирование чёрного ящика, в котором будут участвовать два или более конечных пользователя. Полная форма UAT — это приёмочные испытания.

Структура тестирования

Виды тестирования программного обеспечения

Типы тестирования программного обеспечения:



Обычно тестирование подразделяется на три категории.

1. Функциональное тестирование.
2. Нефункциональное тестирование или тестирование производительности.
3. Сопровождение (регрессия и сопровождение).

Функциональное тестирование

- Модульное тестирование.
- Интеграционное тестирование.
- Дымовое (smoke).
- UAT (приёмочное тестирование).
- Локализация.
- Глобализация.
- Интероперабельность.

Нефункциональное тестирование

- Производительность.
- Выносливость.
- Нагрузка.
- Объём.
- Масштабируемость.
- Удобство использования.

Обслуживание

- Регрессия.
- Техническое обслуживание.

Это далеко не полный список, так как существует более 150 видов тестирования и они продолжают пополняться. Также обратите внимание, что не все виды тестирования применимы ко всем проектам, а зависят от характера и масштаба проекта.

Функциональное тестирование

Функциональное тестирование (Functional Testing) — это процесс обеспечения качества (QA) и тип тестирования чёрного ящика, который основывает свои тестовые случаи на спецификациях тестируемого программного компонента. Функции тестируются путём подачи входных данных и изучения выходных, внутренняя структура программы редко рассматривается.

Функциональное тестирование заключается в проверке того, что Программа (ИС) выполняет все функции, которые были прописаны в ТЗ.

Функциональное тестирование проводится для оценки соответствия системы или компонента заданным функциональным требованиям.

Функциональное тестирование обычно описывает, что делает система.

Поскольку функциональное тестирование является разновидностью тестирования чёрного ящика, функциональность программного обеспечения может быть проверена без знания внутренней работы программного обеспечения. Это означает, что тестировщикам не нужно знать языки программирования или то, как было реализовано программное обеспечение. Это, в свою очередь, может привести к снижению предвзятости разработчика (или предвзятости подтверждения) при тестировании, поскольку тестировщик не участвовал в разработке программного обеспечения.

Функциональное тестирование не означает, что вы тестируете функцию (метод) вашего модуля или класса.

Функциональное тестирование тестирует фрагмент функциональности всей системы.

Функциональное тестирование проверяет программу, сверяя её с проектным документом или спецификацией.

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех **уровнях тестирования** (компонентном, интеграционном, системном, приёмочном).

Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (**use cases**).

Функциональное тестирование включает в себя тестирование функциональных аспектов программного приложения. При выполнении функционального тестирования необходимо проверить каждую функциональность. Вы должны увидеть, получаете ли вы желаемые результаты или нет.

Функциональные тесты выполняются как вручную, так и с помощью инструментов автоматизации. Для этого вида тестирования ручное тестирование не представляет сложности, но при необходимости следует использовать инструменты.

Преимущества функционального тестирования:

- Имитирует фактическое использование системы.

Недостатки функционального тестирования:

- Возможность упущения логических ошибок в программном обеспечении.
- Вероятность избыточного тестирования.

Нефункциональное тестирование

Нефункциональное тестирование — это тестирование нефункциональных аспектов приложения, таких как производительность, надёжность, удобство использования, безопасность и так далее.

Нефункциональное тестирование (НФ). Определяет характеристики ПО, которые измеряются в каких-то конкретных величинах. В первую очередь на таких тестах изучают производительность системы — проводят нагрузочное и стрессовое тестирование, исследуют стабильность и работу с большими базами данных.

А после этого проверяют настройки, отказоустойчивость и восстановление системы, ищут способы увеличить её производительность. Тестирование производительности помогает узнать, как меняются стабильность и быстродействие системы под разной нагрузкой, а также проверить её масштабируемость, надёжность и уточнить, сколько ресурсов она будет использовать.

Нефункциональные тесты выполняются после функциональных тестов.

С помощью нефункционального тестирования можно в значительной степени улучшить качество программного обеспечения. Функциональные тесты также улучшают качество, но с помощью нефункциональных тестов есть возможность сделать программное обеспечение ещё лучше.

Нефункциональные тесты обычно не проводятся вручную. На самом деле, сложно выполнить этот вид тестов вручную. Поэтому такие тесты обычно выполняются с помощью инструментов.

Существует несколько типов нефункционального тестирования, например:

- Тестирование производительности.
- Тестирование безопасности.
- Нагрузочное тестирование.
- Тестирование отказоустойчивости.
- Тестирование совместимости.
- Тестирование удобства использования.
- Тестирование масштабируемости.

- Тестирование объёма.
- Стресс-тестирование.
- Тестирование ремонтопригодности.
- Тестирование на соответствие.
- Тестирование эффективности.
- Тестирование надёжности.
- Тестирование выносливости.
- Тестирование аварийного восстановления.
- Тестирование локализации.
- Тестирование интернационализации.

Некоторые паттерны тестирования

Рассмотрим иерархию и некоторые паттерны тестирования

Юнит-тестирование

Модульное тестирование — это вид тестирования программного обеспечения, при котором тестируются отдельные блоки или компоненты программного обеспечения. Цель заключается в проверке того, что каждая единица программного кода работает так, как ожидается.

Модульное тестирование проводится во время разработки (фазы кодирования) приложения разработчиками. Тесты блоков изолируют участок кода и проверяют его корректность. Единицей может быть отдельная функция, метод, процедура, модуль или объект.

Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по отдельности (модули программ, объекты, классы, функции и т. д.)

Модульное тестирование — это первый уровень тестирования, проводимый перед интеграционным тестированием. Юнит-тестирование — это техника тестирования WhiteBox, которая обычно выполняется разработчиком. Хотя в практическом мире из-за нехватки времени или нежелания разработчиков тестировать, инженеры QA также выполняют модульное тестирование.

Юнит-тестирование важно, потому что разработчики программного обеспечения иногда пытаются сэкономить время, проводя минимальное модульное

тестирование, и это миф, потому что неправильное модульное тестирование приводит к большим затратам на исправление дефектов во время системного тестирования, интеграционного тестирования и даже бета-тестирования после создания приложения. Если правильное модульное тестирование выполняется на ранних стадиях разработки, то это экономит время и деньги в итоге.

Вот основные причины для проведения модульного тестирования в программной инженерии:

Юнит-тесты помогают исправить ошибки на ранних стадиях разработки и сэкономить затраты.

- Это помогает разработчикам понять тестируемую кодовую базу и позволяет им быстро вносить изменения.
- Хорошие модульные тесты служат в качестве проектной документации.
- Юнит-тесты помогают в повторном использовании кода. Перенесите и код, и тесты в новый проект. Дорабатывайте код до тех пор, пока тесты не будут работать снова.

Как выполнить модульное тестирование

Для выполнения модульных тестов разработчики пишут участок кода для тестирования определённой функции в программном приложении. Разработчики также могут изолировать эту функцию для более тщательного тестирования, что позволяет выявить ненужные зависимости между тестируемой функцией и другими модулями и устранить эти зависимости. Обычно используют фреймворк UnitTest для разработки автоматизированных тестовых примеров для модульного тестирования.

Юнит-тестирование бывает двух типов:

- Ручное.
- Автоматизированное.

Ручное

Юнит-тестирование обычно автоматизировано, но может выполняться и вручную. Автоматизация предпочтительнее. При ручном подходе к модульному тестированию может использоваться пошаговый инструктивный документ.

Автоматизированное

Разработчик пишет участок кода в приложении только для того, чтобы протестировать функцию. Позже он комментирует и, наконец, удаляет тестовый код, когда приложение развёрнуто.

Разработчик также может изолировать функцию для более тщательного тестирования. Это более тщательная практика модульного тестирования, которая предполагает копирование и вставку кода в собственную среду тестирования, а не в естественную среду. Изолирование кода помогает выявить ненужные зависимости между тестируемым кодом и другими модулями или пространствами данных в продукте. Затем эти зависимости могут быть устранены.

Для разработки автоматизированных тестовых случаев разработчик может использовать фреймворк UnitTest Framework. Используя фреймворк автоматизации, разработчик вводит в тест критерии для проверки правильности кода. Во время выполнения тестовых примеров фреймворк регистрирует неудачные тестовые случаи. Многие фреймворки также автоматически отмечают неудачные тестовые случаи и сообщают о них в виде сводки. В зависимости от серьёзности неудачи фреймворк может приостановить последующее тестирование.

Рабочий процесс модульного тестирования выглядит следующим образом:

1. Создание тестовых примеров.
2. Обзор/переработка.
3. Базовая линия.
4. Тестирование.
5. Обзор/переработка.
6. Базовая линия.
7. Выполнение тестовых примеров.

Пример модульного теста: Макетные объекты

Юнит-тестирование основывается на создании объектов-макетов для тестирования участков кода, которые ещё не являются частью полного приложения. Макетные объекты заполняют недостающие части программы.

Например, у вас может быть функция, которой нужны переменные или объекты, которые ещё не созданы. В модульном тестировании они будут учтены в виде объектов-макетов, созданных исключительно для целей модульного тестирования данного участка кода.

Разработка, управляемая тестами (TDD) и модульное тестирование

Модульное тестирование в TDD предполагает широкое использование фреймворков тестирования. Фреймворк для модульного тестирования используется для создания автоматизированных модульных тестов. Фреймворки модульного тестирования не являются уникальными для TDD, но они необходимы для него. Ниже мы рассмотрим некоторые из тех преимуществ, которые TDD привносит в мир модульного тестирования:

1. Тесты пишутся до кода.
2. В значительной степени полагаются на фреймворки тестирования.
3. Тестируются все классы в приложениях.
4. Обеспечивается быстрая и простая интеграция.

Преимущество модульного тестирования

- Разработчики, желающие узнать, какую функциональность предоставляет модуль и как её использовать, могут посмотреть на тесты модуля, чтобы получить базовое понимание API модуля.
- Юнит-тестирование позволяет программисту впоследствии рефакторить код и убедиться, что модуль по-прежнему работает правильно (т. е. регрессионное тестирование). Процедура заключается в написании тестовых примеров для всех функций и методов, чтобы в случае, когда изменение вызывает ошибку, её можно было быстро выявить и исправить.
- Благодаря модульной природе модульного тестирования мы можем тестировать части проекта, не дожидаясь завершения других.
- Недостатки модульного тестирования.
- Нельзя ожидать, что модульное тестирование выявит все ошибки в программе. Невозможно оценить все пути выполнения даже в самых тривиальных программах.
- По своей природе модульное тестирование сосредоточено на единице кода. Следовательно, оно не может выявить ошибки интеграции или ошибки широкого системного уровня.
- Рекомендуется использовать модульное тестирование в сочетании с другими видами тестирования.

Smoke, ручное тестирование

Дымовое тестирование — это процесс тестирования программного обеспечения, который определяет, является ли развёрнутая сборка программного обеспечения стабильной или нет.

Дымовое тестирование является подтверждением для команды QA, чтобы приступить к дальнейшему тестированию программного обеспечения.

Оно состоит из минимального набора тестов, выполняемых на каждой сборке для проверки функциональности программного обеспечения. Дымовое тестирование также известно как «верификационное тестирование сборки» или «доверительное тестирование».

Проще говоря, дымовые тесты означают проверку работоспособности важных функций и отсутствие проблем в тестируемой сборке. Это мини и быстрое регрессионное тестирование основных функциональных возможностей. Простой тест, который показывает, что продукт готов к тестированию. Помогает определить, есть ли в сборке недостатки, которые делают дальнейшее тестирование пустой тратой времени и ресурсов.

Дымовое тестирование проводится каждый раз, когда разрабатываются новые функциональные возможности программного обеспечения и интегрируются с существующей сборкой, которая развёрнута в среде QA/staging. Оно гарантирует, что все критические функциональные возможности работают правильно или нет.

Дымовое (Smoke) тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.

При этом методе тестирования команда разработчиков развёртывает сборку в QA. Берётся подмножество тестовых случаев, а затем тестировщики запускают тестовые случаи на сборке. Команда QA тестирует приложение на соответствие критическим функциональным возможностям. Эти серии тестовых примеров предназначены для выявления ошибок, которые есть в сборке. Если эти тесты пройдены, команда QA продолжает функциональное тестирование.

Любая неудача указывает на необходимость вернуть систему команде разработчиков. Всякий раз, когда в сборке происходят изменения, мы проводим Smoke Testing для обеспечения стабильности.

Пример: В окно входа добавляется новая кнопка регистрации, и сборка развёртывается с новым кодом.

Мы проводим дымовое тестирование новой сборки.

Дымовое тестирование позволяет подготовить сборку к дальнейшему формальному тестированию. Основная цель дымового тестирования - выявить основные проблемы на ранней стадии. Дымовые тесты предназначены для демонстрации стабильности системы и соответствия требованиям. Сборка включает в себя все файлы данных, библиотеки, многократно используемые модули, разработанные компоненты, необходимые для реализации одной или нескольких функций продукта.

Как проводить дымовое тестирование?

Дымовое тестирование обычно проводится вручную, хотя существует возможность его автоматизации. Это может варьироваться в зависимости от организации.

Ручное дымовое тестирование

Как правило, дымовое тестирование проводится вручную. В разных организациях оно проводится по-разному. Дымовое тестирование проводится для того, чтобы убедиться, что навигация по критическим путям соответствует ожиданиям и не мешает функциональности.

Как только сборка передаётся в QA, берутся высокоприоритетные функциональные тесты, которые тестируются для выявления критических дефектов в системе. Если тест проходит, мы продолжаем функциональное тестирование. Если тест не прошёл, сборка отклоняется и отправляется обратно в команду разработчиков для исправления. QA снова начинает дымовое тестирование с новой версией сборки.

Пример Smoke теста:

- ТЕСТОВЫЕ СЦЕНАРИИ, ОПИСАНИЕ

Тестирование функции входа в веб-приложение, чтобы убедиться, что зарегистрированный пользователь может войти в систему с именем пользователя и паролем

- ШАГИ ТЕСТИРОВАНИЯ

1. Запуск приложения.
2. Перейдите на страницу входа в систему.

3. Введите действительное имя пользователя.
4. Введите правильный пароль.
5. Нажмите на кнопку входа в систему.

- ОЖИДАЕМЫЙ РЕЗУЛЬТАТ

Вход в систему

- ФАКТИЧЕСКИЙ РЕЗУЛЬТАТ

Вход должен быть успешным, как и ожидалось

- СТАТУС

Pass — тест пройден

Преимущества дымовых испытаний

- Простота проведения тестирования.
- Дефекты будут выявлены на ранних стадиях.
- Улучшает качество системы.
- Снижает риск.
- Прогресс более доступен.
- Экономия усилий и времени на тестирование.
- Легко обнаружить ошибки и исправить их.
- Выполняется быстро.
- Минимизирует интеграционные риски.

Дымовое тестирование проводится на новой сборке и интегрируется со старыми сборками для поддержания корректности системы. Это относится к новым разработкам, крупным и мелким релизам системы.

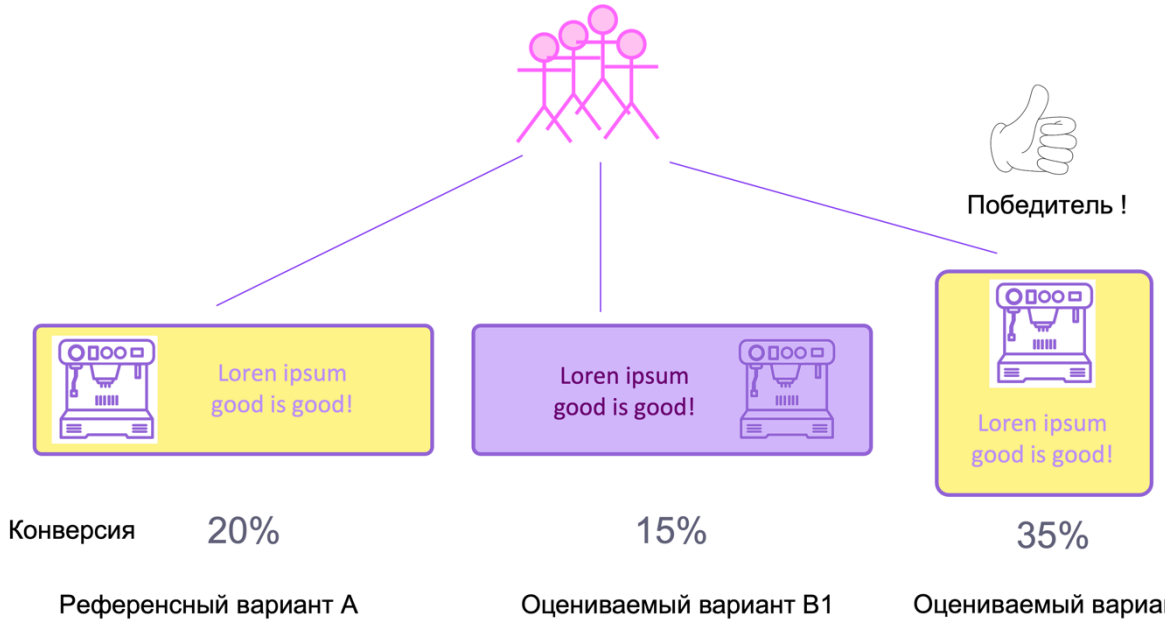
Дымовое тестирование должно проводиться на каждой сборке, которая передаётся на тестирование, так как оно помогает обнаружить дефекты на ранних стадиях.

Перед проведением дымового тестирования команда QA должна проверить правильность версий сборки.

Дымовое тестирование — это заключительный шаг перед тем, как сборка программного обеспечения перейдёт в стадию системного тестирования.

A/B тестирование

A/B-тестирование, также известное как сплит-тестирование, представляет собой рандомизированный процесс экспериментов, в ходе которого две или более версии переменной (веб-страницы, элемента страницы и т. д.) одновременно демонстрируются различным сегментам посетителей сайта, чтобы определить, какая из версий оказывает максимальное воздействие и улучшает бизнес-показатели.



По сути, A/B-тестирование устраняет все догадки при оптимизации веб-сайта и позволяет оптимизаторам принимать решения, основанные на данных. В A/B-тестировании A обозначает «контроль» или исходную переменную тестирования. В то время как B означает «вариацию» или новую версию исходной переменной тестирования.

Версия, которая изменяет ваши бизнес-показатели в положительную сторону, называется «победителем». Внедрение изменений этой победившей вариации на вашей тестируемой странице (страницах) / элементе (элементах) поможет оптимизировать ваш сайт и повысить рентабельность бизнеса.

Метрики конверсии уникальны для каждого веб-сайта. Например, в случае электронной коммерции это может быть продажа товаров. В то же время для B2B это может быть генерация квалифицированных лидов.

A/B-тестирование является одним из компонентов общего процесса оптимизации коэффициента конверсии (CRO), с помощью которого вы можете собрать как

качественные, так и количественные данные о пользователях. В дальнейшем вы можете использовать собранные данные для понимания поведения пользователей, уровня вовлечённости, болевых точек и даже удовлетворённости функциями сайта, включая новые функции, изменённые разделы страниц и т. д. Если вы не проводите A/B-тестирование своего сайта, вы наверняка теряете много потенциального дохода от бизнеса.

Что делает это тестирование замечательным, так это то, что компании получают прямую обратную связь от своих реальных пользователей, представляя им существующие варианты продукта/функций в сравнении с вариантами, таким образом, они могут быстро протестировать новые идеи. В случае если A/B-тестирование покажет, что изменённая версия/подход не эффективны, по крайней мере, компании могут извлечь из этого урок и решить, нужно ли им улучшить его или следует поискать другие идеи.

Преимущества A/B-тестирования

- Позволяет быстро узнать, что работает, а что нет.
- Вы получаете обратную связь непосредственно от фактических/реальных покупателей продукта.
- Поскольку пользователи не знают, что их тестируют, результаты будут непредвзятыми.

Недостатки A/B тестирования

- Представление разного контента/цены/функций разным клиентам, особенно в одной геолокации, может быть потенциально опасным и привести к неприятию изменений (мы обсудим, как это можно решить позже).
- Требует значительного количества ресурсов со стороны продукта, инженеров и специалистов по анализу данных.
- При неправильном проведении может привести к неверным выводам.

End-to-End тестирование

Сквозное тестирование (E2E) — это метод, при котором тестируется весь программный продукт от начала до конца, чтобы убедиться, что поток приложения ведёт себя так, как ожидается. Оно определяет системные зависимости продукта и гарантирует, что все интегрированные части работают вместе так, как ожидается. С точки зрения конечного пользователя путём

имитации реального пользовательского сценария и проверки тестируемой системы и её компонентов на интеграцию и целостность данных.

Сквозное тестирование — также проверяет интеграцию ПО с внешними интерфейсами.

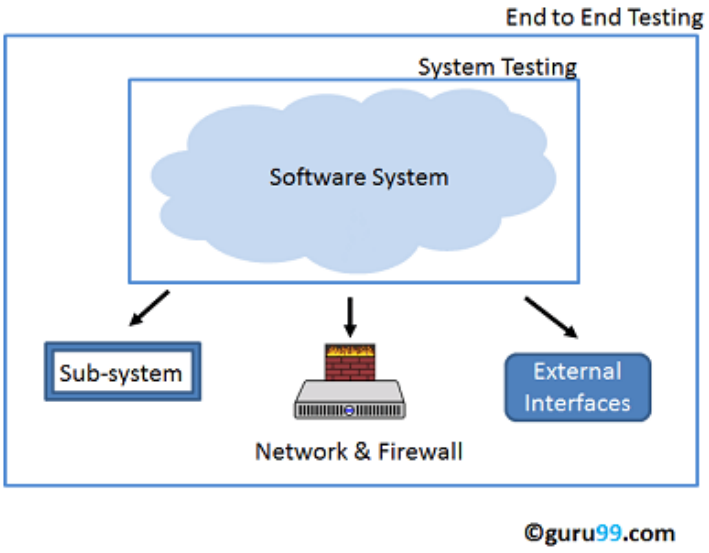
Целью сквозного тестирования является проверка всего программного обеспечения на предмет зависимостей, целостности данных и связи с другими системами, интерфейсами и базами данных для реализации полного сценария производства.

В настоящее время программные системы сложны и взаимосвязаны с многочисленными подсистемами. Если какая-либо из подсистем выйдет из строя, может произойти сбой всей программной системы. Это серьёзный риск, которого можно избежать с помощью сквозного тестирования.

Преимущества сквозного тестирования

- Расширение тестового покрытия.
- Обеспечить корректность приложения.
- Сократить время выхода на рынок.
- Снизить затраты.
- Обнаружить ошибки.

E2E обычно проводится после функционального и системного тестирования. При этом используются реальные производственные данные и тестовая среда для имитации условий реального времени. Тестирование E2E также называют цепным тестированием.



Методы сквозного тестирования

1. **Горизонтальное E2E тестирование**

Часто используемый метод, происходящий горизонтально в контексте нескольких приложений и легко реализуемый в одном приложении ERP (Enterprise Resource Planning).

Это наиболее предпочтительный для тестировщиков метод сквозного тестирования. При горизонтальном тестировании мы проверяем каждый рабочий процесс, тестируя каждое приложение от начала до конца, чтобы проверить, правильно ли функционирует рабочий процесс.

Мы проводим горизонтальное E2E-тестирование, с точки зрения пользователя.

- Оно проверяет, может ли пользователь переходить в другую часть приложения и выполнять действия, как ожидается.
- Оно помогает нам обнаружить ошибки, которые мы могли пропустить во время модульного тестирования.
- Мы можем проводить его на каждом этапе рабочего процесса, это поможет нам сопоставить требования с разработанным приложением.
- Оно обеспечивает корректную работу взаимосвязанного процесса путём тестирования от начала до конца.
- При таком подходе мы можем получить большее покрытие функциональности приложения.
- Обычно мы проводим такое тестирование, когда среда стабильна, в конце цикла выпуска.

Поскольку при этом проверяется рабочий процесс в нескольких приложениях и подсистемах, тестовая среда должна быть настроена заранее.

Преимущества горизонтальных тестов

- Это позволяет протестировать систему, с точки зрения пользователя.
- Это предотвращает попадание ошибок в живую среду.
- Убеждается, что мы покрываем требования к бизнес-логике.

Пример

Клиент использующий веб-приложение для электронной коммерции, включающей в себя счёта, состояние товарных запасов и детали доставки.

Как можно применить сквозное тестирование к этому сценарию.

Шаг 1. Войти в приложение.
Шаг 2. Просмотр различных товаров.
Шаг 3. Выберите товар и добавьте его в корзину.
Шаг 4. Подтвердите и оформите заказ.
Шаг 5. Выполнить оплату.
Шаг 6. Проверьте детали заказа.
Шаг 7. Выйти из приложения.

2. **Вертикальное тестирование E2E**

Этот метод подразумевает послойное тестирование, то есть тесты проводятся в последовательном, иерархическом порядке. Для обеспечения качества каждый компонент системы или продукта тестируется от начала до конца. Вертикальное тестирование часто используется для тестирования критических компонентов сложной вычислительной системы, которая обычно не включает пользователей или интерфейсы.

Обычно используется для критически важных модулей, состоящих из сложных систем. Оно проверяет систему по слоям, здесь мы проводим тесты в последовательном, иерархическом порядке. Это помогает нам протестировать программное обеспечение от начала до конца для всестороннего тестирования.

Вертикальное тестирование фокусируется на слоях, оно тестирует каждый слой архитектуры приложения.

Мы будем работать с подсистемами, которые независимы друг от друга на гранулированном уровне.

Здесь мы начинаем с модульного тестирования, затем продолжаем тестирование как на уровне пользовательского интерфейса, так и на уровне API.

Мы проводим его в иерархическом или последовательном порядке.

Обычно это делается, когда в системе нет пользовательского интерфейса или пользовательский интерфейс находится на высоком уровне сложности.

Предпосылки для вертикального тестирования

Поскольку мы фокусируемся на архитектурном уровне приложения, нам потребуется поддержка со стороны стратегий тестирования или разработки, таких как разработка с учётом поведения (BDD), разработка с учётом тестирования (TDD) или непрерывное тестирование (CI/CD).

Преимущества вертикальных тестов

- Обеспечивает высокое покрытие кода.
- Создаются более целенаправленные тесты.
- Это приводит к более быстрому выполнению тестов.
- Полезно для безопасного тестирования критически важного программного обеспечения.

Пример

В вертикальном тестировании E2E мы тестируем интерфейс или компонент прикладной программы (API).

Протестировать компонент на гранулированном уровне с помощью:

- Модульных тестов.
- Интеграционных тестов (проверить, как компонент ведёт себя при взаимодействии с другими компонентами).
- Автотесты UX/UI (проверить поведение компонента при взаимодействии пользователей через пользовательский интерфейс).

Регрессионное тестирование

Регрессионное тестирование — это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб-сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает, как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты.

Регрессионное тестирование — это полный или частичный выбор уже проведённых тестовых сценариев, которые выполняются повторно, чтобы убедиться, что новые изменения кода не оказывают негативного влияния на существующие функции системы.

Оно гарантирует, что старый код всё ещё работает после внесения последних изменений в код.

Это последняя форма автоматизированной проверки приложений на наличие уязвимостей перед UAT.

По сравнению с ними инструменты статического и динамического анализа сложнее в установке, настройке и поддержке.

Набор для регрессионного тестирования похож на пакеты для функционального тестирования или для тестирования производительности, но проверке подвергаются уже обнаруженные уязвимости, чтобы гарантировать, что они заново не попадут в кодовую базу в результате отката или перезаписи.

Для такого тестирования не требуется специальная среда. Главное условие — способность воспроизвести уязвимость.

Регрессионное тестирование необходимо всякий раз, когда изменяется код, и нужно определить, повлияет ли изменённый код на другие части программного приложения. Более того, регрессионное тестирование необходимо, когда в программное приложение добавляется новая функция. Регрессионное тестирование также может проводиться при устранении функциональных или эксплуатационных дефектов/проблем.

Иногда регрессионные тесты уязвимости настолько просты, что их можно написать ещё до обнаружения этой уязвимости. Это может пригодиться для кода, на который могут сильно повлиять незначительные изменения. В итоге такое тестирование даёт нам простой и эффективный способ предотвращения повторного появления в кодовой базе уже исправленных уязвимостей.

Сами тесты должны по возможности запускаться при всех коммитах или при обновлении хуков (и если тест не пройден, коммит или обновление хуков отклоняются). В более сложных системах управления версиями можно настроить целое расписание регулярных запусков таких тестов.

Как проводить регрессионное тестирование

Чтобы провести регрессионное тестирование, необходимо сначала отладить код, для выявления ошибки.

После выявления ошибок вносятся необходимые изменения для их устранения, затем проводится регрессионное тестирование путём выбора соответствующих тестовых случаев из набора тестов, которые охватывают как изменённые, так и затронутые части кода.

Сопровождение программного обеспечения — это деятельность, которая включает в себя усовершенствование, исправление ошибок, оптимизацию и удаление существующих функций. Эти изменения могут привести к

некорректной работе системы. Поэтому регрессионное тестирование становится необходимым.

Перетестировать всё

Один из методов регрессионного тестирования, при котором все тесты в существующем тестовом ведре или наборе должны быть выполнены заново. Это очень дорого, так как требует огромных затрат времени и ресурсов.

Отбор регрессионных тестов

Отбор регрессионных тестов — это метод, при котором некоторые отобранные тестовые случаи из набора тестов выполняются для проверки того, влияет ли изменённый код на работу программного приложения или нет. Тестовые случаи делятся на две части: повторно используемые тестовые случаи, которые могут быть использованы в последующих циклах регрессии, и устаревшие тестовые случаи, которые не могут быть использованы в последующих циклах.

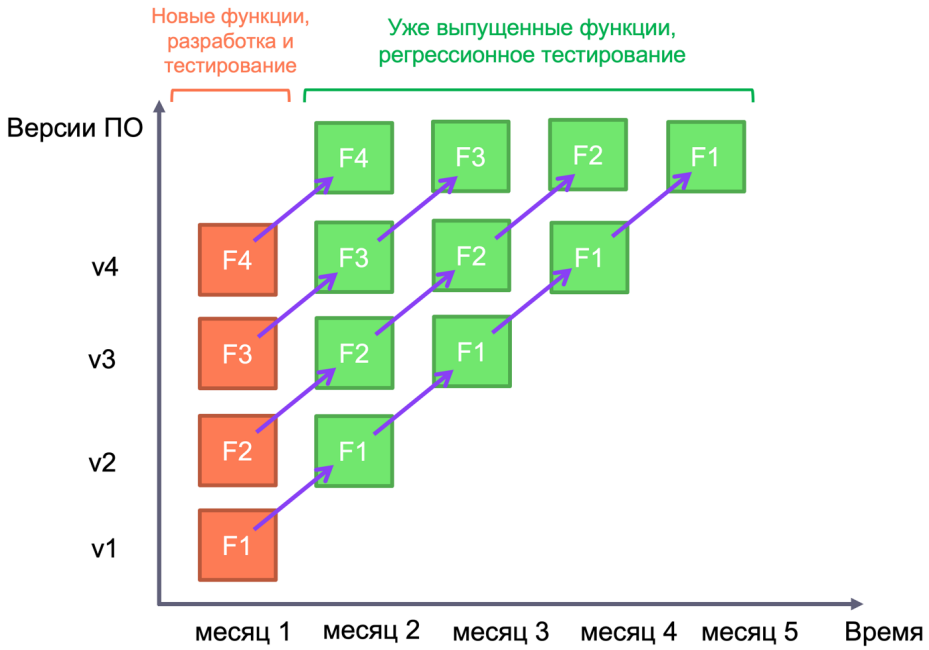
Определение приоритетов тестовых случаев

Расставьте приоритеты тестовых случаев в зависимости от влияния на бизнес, критических и часто используемых функциональных возможностей. Выбор тестовых примеров на основе приоритетов значительно сократит набор регрессионных тестов.

Выбор тестовых примеров для регрессионного тестирования

Из отраслевых данных следует, что значительное количество дефектов, о которых сообщают клиенты, связано с исправлением ошибок в последнюю минуту, что приводит к побочным эффектам, поэтому выбор тестовых примеров для регрессионного тестирования — это целое искусство и не так просто.

Эффективное регрессионное тестирование может быть выполнено путём тщательного отбора тестовых сценариев.



Объём регрессионного тестирования

UAT (User Acceptance test)

Приёмочное тестирование пользователя (UAT) — это вид тестирования, выполняемый конечным пользователем или клиентом для проверки/принятия программной системы перед переносом программного приложения в производственную среду. UAT проводится на заключительном этапе тестирования после функционального, интеграционного и системного тестирования.

UAT также известно как бета-тестирование или тестирование конечного пользователя, определяется как тестирование программного обеспечения пользователем или клиентом, чтобы определить, может ли оно быть принято или нет.

Основной целью такого тестирования является проверка соответствия программного обеспечения бизнес-требованиям. Эта проверка проводится конечными пользователями, которые знакомы с бизнес-требованиями.

Пользователь в контексте программного продукта — это либо потребитель программного обеспечения, либо человек, который попросил создать его для себя (клиент).

Мы знаем, что такое тестирование, а приёмка означает одобрение или согласие.

Основной целью UAT является проверка сквозного бизнес-потока. Оно не фокусируется на косметических ошибках, орфографических ошибках или тестировании системы. Приёмочное тестирование проводится в отдельной тестовой среде с данными, аналогичными производственным. Это своего рода тестирование чёрного ящика, в котором участвуют два или более конечных пользователей.

UAT проводится для:

- Клиента.
- Конечных пользователей.

Формализованный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

Определения удовлетворяет ли система приёмочным критериям.

Вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

UAT, альфа и бета-тестирование — это разные виды приёмочного тестирования.

Поскольку приёмочное тестирование является последним тестированием, которое проводится перед запусом программного обеспечения в эксплуатацию, очевидно, что это последний шанс для заказчика протестировать программное обеспечение и определить, соответствует ли оно поставленной цели.

Необходимость в приёмочном тестировании возникает после того, как программное обеспечение прошло модульное, интеграционное и системное тестирование, поскольку разработчики могли создать программное обеспечение на основе документа с требованиями по собственному разумению, а дальнейшие необходимые изменения в ходе разработки могут быть не доведены до их сведения, поэтому для проверки того, будет ли конечный продукт принят клиентом/конечным пользователем, необходимо приёмочное тестирование.

UAT-тестировщик должен обладать хорошими знаниями о бизнесе. Он должен быть независимым и думать как неизвестный пользователь системы. Тестировщик должен быть аналитиком и объединять все виды данных, чтобы сделать UAT успешным

Тестировщик, бизнес-аналитик или эксперт предметной области, который понимает бизнес-требования или потоки, может подготовить тест и данные, которые реалистичны для бизнеса.

Пример приёмочного теста

Функция:
Построить отчёт о доле продаж 10 товарных категорий за 1 год в виде сегментной диаграммы.

Тест:
Зайти на страницу отчётов для товаров, выбрать 10 товарных категорий, выбрать временной интервал, создать отчёт.

Положительный (принятый) результат:
Построенный отчёт содержит выбранные товарные категории в виде сегментной диаграммы с указанием долей в % каждой категории.

Не является приёмочным (UAT) тестом:
Особенности интерфейса страницы отчётов, цвет сегментов диаграммы, название категорий товаров.

Эти проверки проводятся на этапе верификации спецификации или валидации конкретного параметра, без участия заказчика или пользователя.

Тестирование на отказ и восстановление

Тестирование на отказ и восстановление (Failover and Recovery Testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоёв, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети).

Целью этого **вида тестирования** является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоёв, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу «24x7». Если Вы создаёте продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования вам просто не обойтись. Так как каждая минута простоя или потеря данных в случае отказа оборудования, может стоить вам денег, потери клиентов и репутации на рынке.

Методика подобного **тестирования** заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В

процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя.

Для наглядности рассмотрим некоторые варианты подобного тестирования и общие методы их проведения. Объектом тестирования в большинстве случаев являются весьма вероятные эксплуатационные проблемы, такие как:

- Отказ электричества на компьютере-сервере.
- Отказ электричества на компьютере-клиенте.
- Незавершённые циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации).
- Объявление или внесение в массивы данных невозможных или ошибочных элементов.
- Отказ носителей данных.

Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

- Симулировать внезапный отказ электричества на компьютере (обесточить компьютер).
- Симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство).
- Симулировать отказ носителей (обесточить внешний носитель данных).
- Симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы систем восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определённое требуемое состояние данных продукта:

- Потеря или порча данных в допустимых пределах.
- Отчёт или система отчётов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Стоит заметить, что тестирование на отказ и восстановление — это весьма продукт-специфичное тестирование.

Разработка тестовых сценариев должна производиться с учётом всех особенностей тестируемой системы. Принимая во внимание довольно жёсткие

методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта.

Пентесты

Тестирование на проникновение (также называемое пентестингом или пен-тестом) — это тестирование кибербезопасности, в ходе которого эксперт по тестированию безопасности, называемый пентестером, выявляет и проверяет реальные уязвимости, имитируя действия квалифицированного участника угрозы, решившего получить привилегированный доступ к ИТ-системе или приложению.

Тестирование на проникновение позволяет командам ИТ-безопасности продемонстрировать и улучшить безопасность сетей, приложений, облака, хостов и физических объектов.

Тестирование на проникновение имитирует действия квалифицированного угрожающего субъекта, решившего получить привилегированный доступ.

Пентестер использует опыт, творческий подход и инструменты пентестирования для получения доступа к ИТ-системам, чтобы продемонстрировать, как угрожающий субъект может получить доступ к ИТ-ресурсам или нарушить конфиденциальные данные. Пентестеров также называют оценщиками уязвимостей, хакерами в белой шляпе и этичными хакерами по найму.

Служба тестирования на проникновение, также называемая компанией по пентестингу, выявляет уязвимости в ИТ-системах, которые представляют реальный риск для систем клиента.

Компании по пентесту используют автоматизированные инструменты оценки уязвимостей на этапе обнаружения, предшествующем ручному тестированию на проникновение. Средства сканирования помогают специалистам по тестированию на проникновение выбрать наиболее перспективные цели для использования в серии поэтапных ручных шагов по получению привилегированного доступа.

Важно, чтобы деятельность по тестированию на проникновение не нарушала среду. Иногда пентестеры работают с живыми производственными системами, а иногда — со средами «песочницы», в зависимости от целей тестирования, доступности среды «песочницы» и потенциального воздействия на производственную систему.

Тестирование проводится с использованием привилегированных учётных данных или без них, в зависимости от целей тестирования. Исторически тестирование на проникновение проводилось с точки зрения непривилегированного или анонимного пользователя. Сегодня для глубокого погружения в приложение может потребоваться привилегированный доступ для входа в систему, фактический программный код для визуального просмотра и контроль над операционной системой, на которой размещено приложение.

Типы пентестов

Пентестирование подразделяется на три основных типа тестов.

Они называются:

- Белый ящик.
- Чёрный ящик.
- Серый ящик.

Эти три метода рассматривают различные потенциальные сценарии, в которых может оказаться преступник-хакер, в зависимости от того, как много он знает о компьютерной сети компании.

Тесты на проникновение серый ящик дают испытателю некоторые знания о системе, которую он пытается взломать.

Пен-тесты чёрного ящика дают тестеру нулевые знания о системе

Пен-тесты белого ящика предоставляют тестеру все подробности о системе или сети.

Тестирование на проникновение — это проактивный способ обеспечения безопасности ИТ-сети. Три типа пентестов охватывают различные позиции, в которых может оказаться хакер, и обеспечивают надёжный обзор потенциальных рисков, с которыми может столкнуться организация.

Виды тестирования на проникновение

Целями тестирования на проникновение являются сети, программные приложения, облако и поведение сотрудников. Каждый тип тестирования на проникновение направлен на разные цели:

1. **Тестирование на проникновение в сеть**

Тестирование на проникновение в сеть, также называемое тестированием сетевой безопасности, фокусируется на внутренних и внешних сетях, беспроводных конечных точках и беспроводных сетях, фишинге электронной почты и других видах социальной инженерии. Существует пять типов тестирования на проникновение в сеть:

- Тестирование на проникновение во внешнюю сеть.
- Тестирование на проникновение во внутреннюю сеть.
- Тестирование на проникновение в датацентр.
- Беспроводное тестирование на проникновение.
- Тестирование на проникновение на базе хоста.

2. **Тестирование на проникновение в приложения**

Тестирование на проникновение в приложения, также называемое тестированием безопасности приложений, фокусируется на веб и не веб-приложениях и позволяет найти уязвимости, такие как описанные в OWASP Top Ten и CWE/SANS Top 25 Most Dangerous Software Errors. В отличие от динамического тестирования безопасности приложений (DAST), которое обычно автоматизировано, тестирование на проникновение проводится экспертом-пентестером.

Существует четыре типа тестирования на проникновение в приложения:

- Тестирование на проникновение веб-приложений.
- Тестирование на проникновение мобильных приложений.
- Тестирование на проникновение приложений толстого клиента.
- Тестирование на проникновение виртуальных приложений.

3. **Облачное тестирование на проникновение**

Тестирование на проникновение в облако фокусируется на облачной инфраструктуре. Облачная платформа может создавать уязвимости сети, приложений и конфигурации, которые могут привести к внешнему доступу к учётным данным компании, внутренним системам и конфиденциальным данным.

- Тестирование на проникновение в облачную инфраструктуру AWS, Azure, YandexCloud и другие.

Тестирование производительности

Тестирование производительности — это процесс анализа качества и возможностей продукта. Это метод тестирования, выполняемый для

определения производительности системы с точки зрения скорости, надёжности и стабильности при различных нагрузках. Тестирование производительности также называется **Perf Testing**.

Тестирование производительности — это то тестирование, которое является **нефункциональным**.

Существует множество видов тестирования производительности.

1. Основные показатели (метрики) производительности

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения. Основные из них разобраны ниже.

1) **Потребление ресурсов центрального процессора (CPU, %)**

Метрика, показывающая сколько времени из заданного определённого интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках, для того чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, многопоточности вычислений, и других факторов.

2) **Потребление оперативной памяти (Memory usage, Mb)**

Метрика, показывающая количество памяти, использованной приложением. Использованная память может делиться на три категории:

- Virtual — объём виртуального адресного пространства, которое использует процессор. Этот объём не обязательно подразумевает использование соответствующего дискового пространства или оперативной памяти. Виртуальное пространство и процесс может быть ограничен в возможности загружать необходимые библиотеки.
- Private — объём адресного пространства, занятого процессом и не разделяемого с другими процессами.
- Working Set — набор страниц памяти, недавно использованных процессом. В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. Когда свободной памяти остаётся мало, использованные страницы удаляются.

При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, называемым «сборщиком мусора» (англ. Garbage Collector).

Время затрачиваемое процессором на очистку памяти таким способом может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объёмы памяти, нуждающиеся в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

3) **Потребление сетевых ресурсов**

Эта метрика не связана непосредственно с производительностью приложения, однако её показатели могут указывать на пределы производительности системы в целом.

4) **Работа с дисковой подсистемой (I/O Wait)**

Работа с дисковой подсистемой может значительно влиять на производительность системы, поэтому сбор статистики по работе с диском может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и увеличению времени отклика.

5) **Время выполнения запроса (request response time, ms)**

Время выполнения запроса приложением остаётся одним из главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию/десериализацию, пересылку и обработку запроса. Надо заметить, что не каждое приложение для тестирования производительности может измерить оба этих времени.

6) **Параллелизм / Пропускная способность**

Если конечными пользователями приложения считаются пользователи, выполняющие логин в систему в любой форме, то в этом случае крайне

желательно достижение параллелизма. По определению это максимальное число параллельных работающих пользователей приложения, поддержка которого ожидается от приложения в любой момент времени. Модель поведения пользователя может значительно влиять на способность приложения к параллельной обработке запросов, особенно если он включает в себя периодически вход и выход из системы.

Если концепция приложения не заключается в работе с конкретными конечными пользователями, то преследуемая цель для производительности будет основана на максимальной пропускной способности или числе транзакций в единицу времени. Хорошим примером в данном случае будет являться просмотр веб-страниц, например, на портале Wikipedia.

7) **Время отображения**

Время отображения — одно из самых сложных для приложения для нагрузочного тестирования понятий, так как в общем случае они не используют концепцию работы с тем, что происходит на отдельных узлах системы, ограничиваясь только распознаванием периода времени, в течение которого нет сетевой активности. Чтобы замерить время отображения, в общем случае требуется включать функциональные тестовые сценарии в тесты производительности, но большинство приложений для тестирования производительности не включают в себя такую возможность.

8) **Масштабируемость**

Определяет максимальную пользовательскую нагрузку, которую может выдержать программное приложение.

9) **Стабильность**

Определяет, стабильно ли приложение при изменяющихся нагрузках.

Направления тестирования производительности

Типы тестирования производительности:

Нагрузочное тестирование — проверяет способность приложения работать при ожидаемой пользовательской нагрузке. Цель — выявить узкие места в производительности до того, как программное приложение будет запущено в эксплуатацию.

Стресс-тестирование — включает в себя тестирование приложения под экстремальными нагрузками, чтобы увидеть, как оно справляется с высоким трафиком или обработкой данных. Цель состоит в том, чтобы определить точку разрыва приложения.

Тестирование на выносливость — проводится для того, чтобы убедиться, что программное обеспечение может справиться с ожидаемой нагрузкой в течение длительного периода времени.

Spike-тестирование — тестирование реакции программного обеспечения на внезапные большие скачки нагрузки, создаваемой пользователями.

Объёмное тестирование — при тестировании большим количеством данных. Данные заносятся в базу данных и отслеживается поведение программной системы в целом. Целью является проверка производительности программного приложения при различных объёмах базы данных.

Тестирование масштабируемости — Целью тестирования масштабируемости является определение эффективности программного приложения в «наращивании» для поддержки увеличения пользовательской нагрузки. Это помогает планировать увеличение мощности вашей программной системы.

И другие виды тестирования.

2. Нагрузочное тестирование (Load Test)

Нагрузочное тестирование — это простейшая форма тестирования производительности. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число транзакций за интервал времени. Такой тип тестирования обычно позволяет получить время отклика всех самых важных бизнес-транзакций. В случае наблюдения за базой данных, сервером приложений, сетью и т. д., этот тип тестирования может также идентифицировать некоторые узкие места приложения.

Основными **целями нагрузочного тестирования** являются:

1. Оценка производительности и работоспособности приложения на этапе разработки и передачи в эксплуатацию.

2. Оценка производительности и работоспособности приложения на этапе выпуска новых релизов, патч-сетов.
3. Оптимизация производительности приложения, включая настройки серверов и оптимизацию кода.
4. Подбор соответствующей для данного приложения аппаратной (программной платформы) и конфигурации сервера.

Нагрузка на систему подаётся на протяжении 4–8 часов. В это время собираются метрики производительности: количество запросов в секунду, транзакций в секунду, время отклика от сервера, процент ошибок в ответах, утилизация аппаратных ресурсов и т. д. Собранные метрики проходят проверку на соответствие заданным требованиям. В результате получаем ответ на вопрос: соответствует ли система требованиям производительности?

Также на выходе имеем локализацию узких мест в производительности приложения и дефектов, подробное профилирование всех компонентов системы и утилизацию аппаратных ресурсов под целевой нагрузкой.

Нагрузочное тестирование помогает разработчикам понять поведение системы при определённом значении нагрузки.

3. Стресс-тестирование (Stress Test)

Стресс-тестирование обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надёжности системы во время экстремальных или диспропорциональных нагрузок и отвечает на вопросы о достаточной производительности системы в случае, если текущая нагрузка сильно превысит ожидаемый максимум.

Этот тест проводится первым. Нагрузка постепенно увеличивается до тех пор, пока приложение не перестанет работать корректно. В конце теста фиксируется количество пользователей, которое приложение выдерживало, соответствуя требованиям производительности, и сколько выдержать не смогло. Первое значение и будет пределом производительности вашего приложения. Часто этот вид тестирования проводится, если заказчик предвидит резкое увеличение нагрузки на систему. Например, для e-commerce это могут быть дни распродаж.

Отличие нагрузочного тестирования от стресс тестирования

Задачей нагрузочного тестирования является определение масштабируемости приложения под нагрузкой. При этом происходит:

- Измерение времени выполнения выбранных операций при определённых интенсивностях выполнения этих операций.
- Определение количества пользователей, одновременно работающих с приложением.
- Определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций).
- Исследование производительности на высоких, предельных, стрессовых нагрузках.

Задачей стрессового тестирования является проверка, насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т. е. к возвращению к нормальному состоянию после прекращения воздействия стресса.

Стрессом в этом контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера.

Также одной из задач при стрессовом тестировании может быть оценка деградации производительности.

Заметим, что в рамках одной цели могут использоваться разные виды тестов производительности и нагрузки, например, для первой, второй и третьей цели нужно производить как тестирование производительности, так и тестирование стабильности.

Стресс тестирование (Stress Testing) имеет своей целью проверить, возвращается ли система после запредельной нагрузки (и как скоро) к нормальному режиму.

Также целями стрессового тестирования могут быть проверки поведения системы в случаях когда, один из серверов приложения в пуле перестаёт работать, аварийно изменилась аппаратная конфигурации сервера базы данных и т. д.

При стрессовом тестировании проверяется не производительность системы, а её способность к регенерации после сверх нагрузки.

4. Тестирование на выносливость (Endurance Test)

Endurance Test — это тип тестирования программного обеспечения, при котором система тестируется с нагрузкой, растянутой в течение значительного времени,

для оценки поведения системы при длительном использовании. Цель тестирования состоит в том, чтобы убедиться, что приложение способно выдержать расширенную нагрузку без какого-либо ухудшения времени отклика.

Этот тип тестирования выполняется на последнем этапе цикла выполнения производительности. Испытания на выносливость — это длительный процесс, который иногда длится даже до года. Это может включать применение внешних нагрузок, таких как интернет-трафик или действия пользователя. Это отличает испытание на выносливость от нагрузочного тестирования , которое обычно заканчивается через пару часов или около того.

Выносливость означает способность, поэтому, другими словами, вы можете называть испытание на выносливость испытанием на прочность

Цели испытаний на выносливость:

- Основной целью тестирования на выносливость является проверка на утечки памяти.
- Чтобы узнать, как система работает при устойчивом использовании.
- Чтобы гарантировать, что после длительного периода время отклика системы останется таким же или лучше, чем начало теста.
- Чтобы определить количество пользователей и / или транзакций, эта система будет поддерживать и достигать целей производительности.
- Чтобы управлять будущими нагрузками, нам необходимо понять, сколько дополнительных ресурсов (таких как ёмкость процессора, ёмкость диска, использование памяти или пропускная способность сети) необходимо для поддержки использования в будущем.
- Тестирование на выносливость обычно выполняется либо путём перегрузки системы, либо путём сокращения определённых системных ресурсов и оценки последствий.
- Это выполняется для обеспечения того, чтобы дефекты или утечки памяти не возникали после того, что считается относительно «нормальным» периодом использования.

Что контролировать в тестировании на выносливость

В тестировании на выносливость проверяются следующие вещи.

- **Проверка утечки памяти.** Проверяется, есть ли утечка памяти в приложении, которая может вызвать сбой системы или ОС.

- **Проверьте закрытие соединения между слоями системы.** Если соединение между слоями системы не будет успешно закрыто, это может привести к остановке некоторых или всех модулей системы.
- **Тест соединения с базой данных успешно завершён.** Если соединение с базой данных не было успешно закрыто, это может привести к сбою системы.
- **Проверка времени отклика.** Система проверяется на время отклика системы, поскольку приложение становится менее эффективным в результате длительного использования системы.

Пример испытания на выносливость

В то время как **стресс-тестирование** выводит тестируемую систему к своим пределам, **тестирование на выносливость со временем** приводит приложение к своему пределу.

Например, наиболее сложные проблемы — утечки памяти, использование сервера базы данных и не отвечающая система — возникают, когда программное обеспечение работает в течение длительного периода времени. Если вы пропустите тесты на выносливость, ваши шансы обнаружить такие дефекты до развёртывания весьма низки.

Преимущества испытаний на выносливость:

- Это помогает определить, как рабочая нагрузка может обрабатывать система под нагрузкой.
- Предоставляет точные данные, которые клиент может использовать для проверки или улучшения потребностей своей инфраструктуры.
- Определяет проблемы с производительностью, которые могут возникнуть после того, как система работает на высоком уровне в течение более длительного периода времени.
- Типичные проблемы выявляются в меньших целевых тестах производительности, что означает, что приложение гарантирует, что приложение останется доступным даже при большой нагрузке за очень короткий промежуток времени.
- Тест на выносливость также используется, чтобы проверить, есть ли снижение производительности после длительного периода выполнения.

Недостатки испытаний на выносливость:

- Часто трудно определить, какой стресс стоит применить.

- Тестирование на выносливость может привести к сбоям приложений и / или сети, что может привести к значительным сбоям, если среда тестирования не изолирована.
- Постоянная потеря данных или повреждение могут возникнуть из-за чрезмерной нагрузки на систему.
- Использование ресурсов остаётся очень высоким после снятия стресса.
- Некоторые компоненты приложения не отвечают.
- Необработанные исключения наблюдаются конечным пользователем.

5. Спайк-тестирование (Spike Test)

Ещё одна подгруппа стресс-тестирования оценивает производительность системы при внезапном и значительном увеличении числа имитируемых конечных пользователей.

Спайк-тесты помогают определить, может ли система справиться с резким и значительным увеличением нагрузки в течение короткого периода времени, причём неоднократно. Подобно стресс-тестам, ИТ-команда обычно проводит спайк-тесты перед крупным событием, в ходе которого система, вероятно, будет испытывать более высокие, чем обычно, объёмы трафика.

Тестирование масштабируемости измеряет производительность на основе способности программного обеспечения увеличивать или уменьшать атрибуты измерения производительности. Например, тест на масштабируемость может быть проведён на основе количества пользовательских запросов.

6. Тестирование на больших объёмах данных (Volume Test)

VOLUME TESTING — это тип тестирования программного обеспечения, когда программное обеспечение подвергается огромному объёму данных. Это также упоминается как тестирование наводнением. Объёмное тестирование проводится для анализа производительности системы путём увеличения объёма данных в базе данных.

Этот вид тестирования помогает сделать прогноз относительно работоспособности приложения. Форма подаваемой нагрузки та же, что и при нагрузочном тестировании. Задача теста – узнать, какое влияние окажет увеличение объёма данных на систему. Таким образом, можем найти ответ на вопрос: как изменится производительность приложения спустя X лет, если аудитория приложения вырастет в Y раз?

С помощью объёмного тестирования влияние на время отклика и поведение системы можно изучить при воздействии большого объёма данных.

Целью проведения объёмного тестирования является:

- Проверить производительность системы с увеличением объёмов данных в базе данных.
- Определить проблему, которая может возникнуть с большим объёмом данных.
- Выяснить точку, в которой стабильность системы ухудшается.

Задачей объёмного тестирования является получение оценки производительности при увеличении объёмов данных в базе данных приложения, при этом происходит:

- Измерение времени выполнения выбранных операций при определённых интенсивностях выполнения этих операций.
- Может производиться определение количества пользователей, одновременно работающих с приложением.
- Например, тестирование поведения музыкального сайта, когда миллионы пользователей скачивают песню.

Преимущества объёмного тестирования:

- Выявив проблемы с нагрузкой, можно сэкономить много денег, которые ,в противном случае, будут потрачены на обслуживание приложений.
- Помогает быстрее начать настройку масштабирования.
- Раннее выявление узких мест.
- Гарантирует, что система теперь может использоваться в реальных условиях.

Проблемы в объёмном тестировании:

- Необходима динамическая генерация ключей доступа для больших данных.
- Сохранить целостность реляционных данных.

7. Тестирование отказоустойчивости (Stability Test)

Тестирование стабильности проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти, чтобы выявить потенциальные утечки.

Кроме того, такое тестирование выявляет деградацию производительности, выражающуюся в снижении скорости обработки информации и/или увеличении времени ответа приложения после продолжительной работы по сравнению с началом теста.

Продолжительность нагрузки может варьироваться в зависимости от целей и возможностей проекта, доходя до семи дней и более. В результате получаем представление о том, как изменится производительность системы в течение длительного периода времени под нагрузкой, например, в течение недели. Снизится ли уровень производительности? Способно ли приложение выдерживать стабильную нагрузку без критических сбоёв?

8. Тестирование масштабируемости (Scalability Test)

Профиль нагрузки тот же, что и при нагрузочном тестировании. Что получаем в результате? Ответы на следующие вопросы:

- Увеличится ли производительность приложения, если добавить дополнительные аппаратные ресурсы?
- Увеличится ли производительность пропорционально количеству добавленных аппаратных средств?

По сравнению с тестированием производительности тестирование масштабируемости означает анализ того, как система реагирует на изменения в количестве одновременных пользователей. Ожидается, что системы будут масштабироваться вверх или вниз и корректировать объём ресурсов, используемых для обеспечения того, чтобы пользователи использовали последовательную и стабильную работу, несмотря на число одновременных пользователей.

Тестирование масштабируемости также может быть сделано на оборудовании, сетевых ресурсах и базах данных, чтобы увидеть, как они реагируют на различное количество одновременных запросов. В отличие от тестирования нагрузки, где анализируется, как ваша система реагирует на различные уровни нагрузки, тестирование масштабируемости анализирует, насколько хорошо масштабируется ваша система в ответ на различные уровни нагрузки. Последнее особенно важно в контейнерных средах.

9. Тестирование пропускной способности (Bandwidth Test)

Похоже на стресс-тестирование в том, что оно тестирует нагрузку на трафик, основанную на количестве пользователей, но отличается количеством.

Тестирование пропускной способности рассматривает, может ли программное приложение или среда обрабатывать объём трафика, для которого оно было специально разработано.

10. Конфигурационное тестирование (Config Pref Test)

Конфигурационное тестирование — ещё один из видов традиционного тестирования производительности. В этом случае вместо того, чтобы тестировать производительность системы с точки зрения подаваемой нагрузки, тестируется эффект влияния на производительность изменений в конфигурации. Хорошим примером такого тестирования могут быть эксперименты с различными методами балансировки нагрузки.

Конфигурационное тестирование также может быть совмещено с нагрузочным, стресс или тестированием стабильности.

11. Тестирование производительности в облаке (Cloud Perf Test)

Тестирование производительности можно проводить и в облаке. Преимущество облачного тестирования производительности заключается в возможности тестирования приложений в более широком масштабе, при этом сохраняя преимущества стоимости облака. Сначала организации думали, что перемещение тестирования производительности в облако облегчит процесс тестирования производительности и сделает его более масштабируемым. Процесс мышления заключался в том, что организация может разгрузить процесс в облако, и это решит все их проблемы. Однако, когда организации начали делать это, они начали обнаруживать, что всё ещё существуют проблемы при проведении тестирования производительности в облаке, так как у организации не будет глубоких, «белых» знаний на стороне поставщика облака.

Одной из проблем при переносе приложения из локальной среды в облако является самоуспокоенность. Разработчики и ИТ-персонал могут считать, что приложение будет работать точно так же, как и в облаке. Они сведут к минимуму тестирование и QA и приступят к быстрому развёртыванию. Поскольку приложение тестируется на оборудовании другого поставщика, тестирование может быть не таким точным, как если бы оно было размещено на территории компании.

Затем необходимо скоординировать работу команд разработчиков и операторов, чтобы проверить наличие пробелов в системе безопасности, провести

нагрузочное тестирование, оценить масштабируемость, учесть опыт пользователей и составить карту серверов, портов и путей.

Межприкладная коммуникация может быть одной из самых больших проблем при переносе приложения в облако. Облачные среды, как правило, имеют больше ограничений по безопасности внутренних коммуникаций, чем локальные среды. Перед переносом приложения в облако организация должна составить полную карту серверов, портов и путей связи, которые использует приложение. Также может помочь проведение мониторинга производительности.

12. Испытание «на впитывание» (Soak Test)

Soak Testing — это тип тестирования программного обеспечения, при котором система тестируется под огромной нагрузкой в течение непрерывного периода доступности, чтобы проверить поведение системы в условиях производственного использования.

Soak Testing проверяет, что система может выдержать огромный объём нагрузки в течение длительного периода времени. Оно также проверяет, что произойдёт за пределами проектных ожиданий системы.

Испытание «на впитывание» включает в себя тестирование системы с типичной производственной нагрузкой в течение непрерывного периода доступности, чтобы подтвердить поведение системы в условиях производственного использования.

Может потребоваться экстраполяция результатов, если невозможно провести такое расширенное тестирование. Например, если система должна обрабатывать 10 000 транзакций в течение 100 часов, может быть возможно завершить обработку тех же 10 000 транзакций за более короткое время (скажем, за 50 часов) как репрезентативное (и консервативная оценка) для фактического производственного использования. Хороший тест на впитываемость также включает возможность моделирования пиковых нагрузок, а не только средних нагрузок. Если манипулирование нагрузкой в течение определённых периодов времени невозможно, альтернативно (и консервативно) разрешите системе работать при пиковой производственной нагрузке в течение всего теста.

Например, при тестировании программного обеспечения система может вести себя именно так, как ожидается, при тестировании в течение одного часа. Однако, когда она тестируется в течение трёх часов, такие проблемы, как утечка памяти, приводят к отказу или неожиданному поведению системы.

Испытания на впитывание используются в основном для проверки реакции тестируемого объекта в условиях возможной симулированной среды в течение заданной продолжительности и при заданном пороге. Наблюдения, сделанные во время испытаний на впитывание, используются для улучшения характеристик объекта при дальнейших испытаниях.

Цель тестирования на впитывание:

- Проверить поведение системы под высокой нагрузкой в течение длительного времени.
- Предсказать отказ, вызванный большой нагрузкой.
- Проверить производительность системы.
- Сделать систему надёжной и стабильной.

Сбои или проблемы, обнаруженные в ходе тестирования Soak Testing, следующие:

- Утечки памяти:

Soak-тестирование обнаруживает серьёзные утечки памяти, которые могут вызвать крах приложения или привести к краху операционной системы.

- Отказ соединений уровней:

Soak-тестирование обнаруживает отказ тесных соединений между слоями системы, что может прервать работу модулей системы.

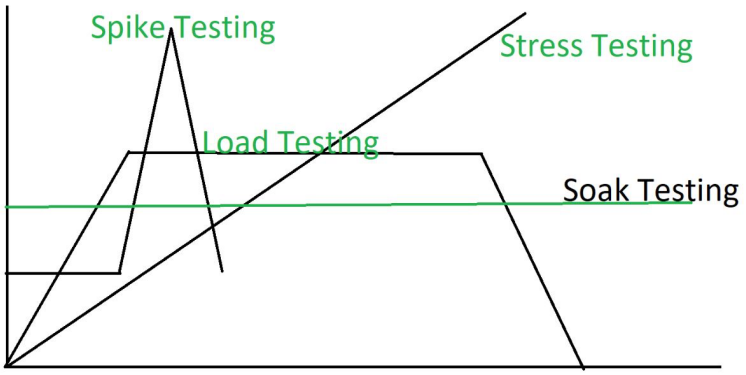
- Отказ соединений с базой данных:

Тестирование на впитывание обнаруживает отказ тесных соединений базы данных при определённых условиях, что может привести к сбою всей системы.

- Деградация времени отклика:

Тестирование на впитывание обнаруживает ухудшение времени отклика системы, так как система становится менее эффективной и требует больше времени для отклика.

Графическое представление Soak Testing:



Пример распределения нагрузки при некоторых видах тестирования производительности

Преимущества Soak Testing:

- Испытание на впитывание улучшает производительность системы.
- Тестирование на впитывание повышает устойчивость системы.
- Тестирование на впитывание заставляет систему работать под большой нагрузкой.
- Оно улучшает поведение системы под большой нагрузкой в течение длительного времени.

Пример

Когда банк объявляет о своём закрытии, ожидается, что его система будет обрабатывать большое количество транзакций в дни закрытия банка. Это событие редкое и неожиданное, но ваша система всё равно должна справиться с этой маловероятной ситуацией.

Система должна выдержать период отсутствия дежурного персонала. Обычно это происходит, когда персонал не работает в выходные дни, поэтому тест на впитывание должен доказать способность системы функционировать в течение периода, превышающего выходные дни.

Другие виды тестирования

Установочные испытания

Часто после завершения системных и приёмочных испытаний программное обеспечение проверяется при установке в целевой среде. Тестирование

установки можно рассматривать как тестирование системы, проводимое в рабочей среде конфигураций оборудования и других эксплуатационных ограничений. Также могут быть проверены процедуры установки.

Тестирование безопасности

Тестирование безопасности направлено на проверку того, что программное обеспечение защищено от внешних атак. В частности, тестирование безопасности проверяет конфиденциальность, целостность и доступность системы и её данных. Обычно тестирование безопасности включает в себя проверку против неправильного использования и злоупотребления программным обеспечением или системой (негативное тестирование).

Тестирование конфигурации

В случаях, когда программное обеспечение создаётся для обслуживания различных пользователей, конфигурационное тестирование проверяет программное обеспечение при различных заданных конфигурациях.

Тестирование интерфейса. Человеко-компьютерное взаимодействие (эргономика — юзабилити)

Дефекты интерфейса часто встречаются в сложных системах. Тестирование интерфейса направлено на проверку правильности взаимодействия компонентов для обеспечения корректного обмена данными и управляющей информацией. Обычно тестовые примеры генерируются на основе спецификации интерфейса. Особой целью тестирования интерфейсов является имитация использования API приложениями конечных пользователей. Это включает в себя генерацию параметров вызовов API, установку условий внешней среды и определение внутренних данных, которые влияют на API.

Основная задача тестирования юзабилити и взаимодействия человека и компьютера — оценить, насколько легко конечным пользователям изучать и использовать программное обеспечение. В общем, это может включать в себя тестирование функций программного обеспечения, которые поддерживают задачи пользователя, документации, которая помогает пользователям, и способности системы восстанавливаться после ошибок пользователя.

Тестирование доступности

Проверка того, доступно ли ваше программное обеспечение для людей с ограниченными возможностями или нет, называется тестированием доступности.

Для этого типа тестов необходимо проверить, смогут ли люди с ограниченными возможностями, например, дальтоники, слепые и глухие, использовать ваше приложение.

Альфа-тестирование

Альфа-тестирование — это вид тестирования для поиска всех ошибок и проблем во всём программном обеспечении. Этот вид тестирования проводится на последнем этапе разработки приложения и выполняется на месте разработчиков, перед запуском продукта или перед передачей его клиенту, чтобы гарантировать, что пользователь/клиент получит безошибочное программное приложение.

Альфа-тестирование проводится перед бета-тестированием, что означает, что после проведения альфа-тестирования необходимо провести бета-тестирование.

Альфа-тестирование не проводится в реальных условиях. Скорее, этот вид тестирования проводится путём создания виртуальной среды, напоминающей реальную.

Бета-тестирование

Как было сказано ранее, бета-тестирование проводится после альфа-тестирования. Бета-тестирование проводится перед запуском продукта. Оно проводится в реальной пользовательской среде ограниченным числом реальных клиентов или пользователей, чтобы убедиться, что в программном обеспечении нет ошибок и оно функционирует без сбоёв. После сбора отзывов и конструктивной критики от этих пользователей вносятся некоторые изменения, чтобы сделать программное обеспечение лучше.

Поэтому, когда программное обеспечение находится в стадии бета-тестирования, оно называется бета-версией. После завершения тестирования программное обеспечение выпускается для публики.

Тестирование на совместимость

Тестирование совместимости включает в себя проверку совместимости программного обеспечения с различными операционными системами, веб-браузерами, сетевыми средами, оборудованием и так далее. Оно проверяет, нормально ли работает разработанное программное приложение с различными конфигурациями.

Приведём несколько примеров: если программное обеспечение является приложением для Windows, необходимо проверить, совместимо ли оно с

различными версиями операционной системы Windows. Если это веб-приложение, то проверяется, легко ли оно доступно с различными версиями широко используемых веб-браузеров. А если это приложение для Android, следует проверить, хорошо ли оно работает со всеми широко используемыми версиями операционной системы Android.

Тестирование на обратную совместимость

Тестирование на обратную совместимость проводится для проверки совместимости новой или обновлённой версии приложения с предыдущими версиями сред (таких как операционные системы и веб-браузеры), на которых работает это программное обеспечение. Иногда приложение обновляется специально для того, чтобы соответствовать стандартам и стилю новой, более современной среды. В этом случае необходима поддержка обратной совместимости.

Это тестерование гарантирует, что все использующие более старые версии конкретной среды, смогут пользоваться вашим программным обеспечением.

Тестирование на совместимость с браузером

Как видно из названия, тестирование совместимости с браузером проверяет веб-приложение на совместимость с браузером. Точнее, проверяется, может ли веб-приложение быть легко доступно из всех версий основных веб-браузеров.

Это специфическая форма тестирования совместимости, в то время как тестирование совместимости проверяет общую совместимость.

Тестирование надёжности

Тестирование надёжности — это вид тестирования программного обеспечения, который проверяет, является ли программное обеспечение надёжным или нет. Другими словами, проверяется, работает ли программное обеспечение без ошибок и можно ли на него положиться.

Например, если важная информация пользователя, хранящаяся в базе данных программного обеспечения, внезапно удаляется через несколько месяцев из-за ошибки в коде, можно сказать, что программное обеспечение ненадёжно.

Техники тестирования

Одна из целей тестирования — выявить как можно больше отказов. Для этого было разработано множество методик. Эти методы пытаются «сломать» программу, будучи максимально системными в определении входных данных, которые приведут к репрезентативному поведению программы; например, рассматривая подклассы входной области, сценарии, состояния и потоки данных.

Приведённая здесь классификация методов тестирования основана на том, как генерируются тесты: на основе интуиции и опыта инженера-программиста, спецификаций, структуры кода, реальных или воображаемых неисправностей, которые необходимо обнаружить, прогнозируемого использования, моделей или характера приложения. В одной категории рассматривается комбинированное использование двух или более методов.

На основе интуиции и опыта инженера-программиста — Ad Hoc

Возможно, наиболее широко практикуемой техникой является тестирование ad hoc: тесты составляются на основе навыков, интуиции и опыта инженера-программиста при работе с аналогичными программами. Специальное тестирование может быть полезно для выявления тестовых случаев, которые нелегко генерируются более формализованными методами.

Ad-hoc тестирование — это вид тестирования, который проводится в разовом порядке, без использования каких-либо тестовых случаев, планов, документации или систем. В отличие от всех других видов тестирования, этот вид тестирования не проводится систематически.

Хотя поиск ошибок может быть затруднён без использования тест-кейсов, существуют технические проблемы, которые легко обнаружить с помощью специального тестирования, но трудно найти с помощью других подходов к тестированию, использующих тест-кейсы.

Этот неформальный тип тестирования программного обеспечения может быть выполнен любым человеком, вовлечённым в проект.

Исследовательское тестирование (Exploratory test)

Исследовательское тестирование определяется как одновременное обучение, разработка тестов и их выполнение; то есть тесты не определяются заранее в установленном плане тестирования, а динамически разрабатываются, выполняются и изменяются.

Эффективность исследовательского тестирования зависит от знаний инженера-программиста, которые могут быть получены из различных источников: наблюдаемое поведение продукта во время тестирования, знакомство с приложением, платформой, процессом отказа, типом возможных неисправностей и отказов, риском, связанным с конкретным продуктом и так далее.

Простейшее определение исследовательского тестирования — это разработка и выполнения тестов в одно и то же время. Что является противоположностью сценарного подхода (с его предопределёнными процедурами тестирования, неважно ручными или автоматизированными). Исследовательские тесты, в отличие от сценарных тестов, не определены заранее и не выполняются в точном соответствии с планом.

Разница между ad hoc и exploratory testing в том, что теоретически, ad hoc может провести кто угодно, а для проведения exploratory необходимо мастерство и владение определёнными техниками.

Профилирование

Профилирование — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш-промахов и т. д. Инструмент, используемый для анализа работы, называют профилировщиком или профайлером (profiler). Обычно выполняется совместно с оптимизацией программы.

Цели профилирования — оценка эффективности работы программы или вычислительной системы. Определение критических участков программы (hotspots) или компонентов вычислительной системы.

Характеристики профилирования могут быть аппаратными (время) или вызванные программным обеспечением (функциональный запрос). Инструментальные средства анализа программы чрезвычайно важны для того,

чтобы понять поведение программы. Проектировщики ПО нуждаются в таких инструментальных средствах, чтобы оценить, как хорошо выполнена работа. Программисты нуждаются в инструментальных средствах, чтобы проанализировать их программы и идентифицировать критические участки программы.

Часто используется, чтобы определить, как долго выполняются определённые части программы, как часто они выполняются, или генерировать граф вызовов (Call Graph). Обычно эта информация используется, чтобы идентифицировать те участки программы, которые работают больше всего. Эти трудоёмкие участки могут быть оптимизированы, чтобы выполняться быстрее.

Также выделяют анализ покрытия (Code Coverage) — процесс выявления неиспользуемых участков кода при помощи, например, многократного запуска программы.

Результаты профилирования:

Характеристики работы программы

- Путь исполнения (покрытие кода, дерево вызовов подпрограмм, количество вызовов подпрограмм, ...).
- Время работы участка программы (функции, оператора языка, машинной команды, ...).
- Количество событий, произошедших в системе (исполненных команд, промахов кэша, неправильно предсказанных переходов, ...).
- Распределение времени и событий по коду программы.
- Потребление ресурсов системы (процессор, память, диск, сеть, ...).
- Анализ характеристик.
- Обнаружение критических участков программы (hotspots, critical path).
- Оценка достигнутой производительности (хорошо/плохо), причины плохой производительности, варианты улучшения.

Способы профилирования:

- Инструментирование кода — внесение в программу дополнительного кода для сбора нужной информации (оказывает сильное влияние на характеристики программы).
- Обработка событий – код профилировщика вызывается при срабатывании определённых событий.
- Отслеживание высокоуровневых событий (call/ret/new/delete/exception/...).

- Статистическое профилирование (sampling) – остановка по системным/аппаратным прерываниям и накопление статистики на точках останова (малое влияние на характеристики программы, но результаты профилирования приблизительны).
- Эмуляция выполнения программы (повторяемость результата, но значительно большее время работы программы и не учитывается «реальная» ситуация).

Профилировщик — это инструмент, который наблюдает за выполнением другого приложения. Профилировщик среды CLR — это библиотека DLL, содержащая функции, которые получают сообщения из среды CLR и отправляют сообщения в среду CLR с помощью API профилирования. Библиотека DLL профилировщика загружается средой CLR во время выполнения.

Традиционные средства профилирования основное внимание уделяют измерению выполнения приложения. То есть они измеряют время, затраченное на каждую функцию, или использование памяти приложением за период времени. API профилирования предназначен для более широкого класса диагностических средств, таких как служебные программы с покрытием кода и расширенные средства отладки. Сфера их применения — вся диагностика в природе. API профилирования не только измеряет, но также наблюдает за выполнением приложения. По этой причине API профилирования никогда не должен использоваться самим приложением, и выполнение приложения не должно ни зависеть от профилировщика, ни подвергаться его влиянию.

Для профилирования приложения среды CLR требуется дополнительная поддержка по сравнению с профилированием стандартно скомпилированного машинного кода. Это объясняется тем, что в среде CLR вводятся такие понятия, как домены приложений, сборка мусора, обработка управляемых исключений, JIT-компиляция кода (преобразование кода MSIL в машинный код) и другие аналогичные возможности. Механизмы традиционного профилирования не могут обнаруживать эти возможности или предоставлять полезные сведения о них. API профилирования эффективно предоставляет эти отсутствующие сведения с минимальным влиянием на производительность среды CLR и профилируемого приложения.

JIT-компиляция во время выполнения обеспечивает прекрасные возможности для профилирования. API профилирования позволяет профилировщику вносить изменения потока кода MSIL в памяти для подпрограммы перед её JIT-компиляцией. Таким образом, профилировщик может динамически

добавлять код инструментирования в определённые подпрограммы, требующие более глубокого анализа. Хотя такой подход возможен в обычных сценариях, его гораздо проще реализовать для среды CLR с помощью API профилирования.

API профилирования

Как правило, API профилирования используется для написания профилировщика кода, который является программой, отслеживающей выполнение управляемого приложения.

Статический анализ кода.

Статический анализ кода — это процесс анализа, обнаружения ошибок и дефектов в исходном коде программного обеспечения, без реального выполнения исследуемого кода.

Статический анализ можно рассматривать как автоматизированный процесс проверки (ревью) кода (code-review).

Рецензирование кода — один из самых проверенных и надёжных методов обнаружения дефектов. Он заключается в совместном внимательном чтении исходного кода и выдаче рекомендаций по его улучшению. Этот процесс позволяет выявить ошибки или фрагменты кода, которые могут стать ошибками в будущем. Также считается, что автор кода не должен давать пояснений о том, как работают те или иные части программы. Алгоритм выполнения программы должен быть понятен из текста программы и комментариев. Если это не так, то код нуждается в доработке.

Рецензирование кода обычно работает хорошо, потому что программистам гораздо легче заметить ошибки в чужом коде, чем в своём.

Единственным существенным недостатком метода совместного просмотра кода является чрезвычайно высокая цена - трудозатраты: необходимо регулярно просматривать свежий код или повторно просматривать код после внесения рекомендованных изменений.

Существуют и другие способы использования инструментов статического анализа кода. Например, статический анализ можно использовать как метод контроля и обучения новых сотрудников, которые ещё недостаточно знакомы с правилами программирования компании.

Статический анализ кода относится к технике приблизительного определения поведения программы во время выполнения. Другими словами, это процесс предсказания результатов работы программы без её фактического выполнения.

Различные виды статического анализа кода

Одним из наиболее распространённых видов статического анализа кода является SAST или статическое тестирование безопасности приложений. Это также считается лучшей практикой для тестирования безопасности приложений, но может применяться и в других областях.

Для выявления всех классов ошибок может потребоваться использование нескольких стандартов кодирования (MISRA, AUTOSAR, CERT, CWE, PEP и т. д.).

Типы статического анализа кода и ошибки, которые они призваны выявить.

1) **Обнаружение ошибок в коде.**
2) **Производительность.** Эти тесты выявляют ошибки, которые снижают общую производительность. Они также могут быть использованы для того, чтобы убедиться, что разработчики не отстают от современных лучших практик.
3) **Вычисление метрик.** Метрика программного обеспечения — это мера, которая позволяет получить числовое значение какого-либо свойства программного обеспечения или его спецификаций. Существует множество различных метрик, которые могут быть вычислены с помощью определённых инструментов.
4) **Безопасность.** Критический тест, анализ исходного кода, связанный с безопасностью, обнаруживает такие риски безопасности, как слабая криптография, проблемы с конфигурацией и специфические для фреймворка ошибки введения команд.
5) **Надёжность.** Эти тесты помогают предотвратить проблемы с функциональностью. Ни один разработчик не хочет иметь дело с аварийным сообщением о неотвечающем сервисе в 4 часа утра. Этот тип статического анализа кода особенно полезен для поиска утечек памяти или проблем с потоками.
6) **Рекомендации по форматированию кода. Стиль (code-style).** Некоторые статические анализаторы позволяют проверить, соответствует ли исходный код стандарту форматирования кода, принятому в вашей компании. Под этим подразумевается контроль количества отступов в различных конструкциях, использование пробелов/табов и так далее. Этот

стиль статического анализа поощряет команды к принятию единого стиля кодирования для простоты использования, понимания и исправления ошибок. При этом экономя время.

Выполнение статического анализа требует выполнения набора шагов:

- Необходимо иметь исходный код, чтобы проверить его качество.
- Затем воспользоваться инструментами статического анализа и запустить статический анализатор кода.
- Просмотреть отмеченные участки, которые не соответствуют предписанному набору правил. Это могут быть ложные срабатывания или даже ожидаемые отклонения.
- Разработчики сначала устраняют критические ошибки, а затем занимаются менее значительными проблемами.
- Перейти к этапу тестирования.

Самое лучшее в статическом анализе — это то, что он не требует выполнения кода. Достаточно запустить анализ, чтобы выявить проблемы, которые необходимо устранить без излишнего риска.

Фаззинг (Fuzzing)

Фаззинг — техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая, заключающая в передаче приложению на вход неправильных, неожиданных или случайных данных. Предметом интереса являются падения и зависания, нарушения внутренней логики и проверок в коде приложения, утечки памяти, вызванные такими данными на входе.

Фаззинг является разновидностью выборочного тестирования (англ. random testing), часто используемого для проверки проблем безопасности в программном обеспечении и компьютерных системах.

Целью Fuzz-тестирования является вставка данных с помощью автоматизированных или полуавтоматических методов и тестирование системы на различные исключения, такие как крах системы, отказ встроенного кода и т. д.

Fuzz-тестирование было первоначально разработано Бартоном Миллером в Университете Висконсина в 1989 году.

Fuzz-тестирование даёт более эффективный результат при использовании с тестированием чёрного ящика, бета-тестированием и другими методами отладки.

Fuzz-тестирование используется для проверки уязвимости программного обеспечения. Это очень экономически эффективный метод тестирования.

Простейшей формой техники фаззинга является передача случайных входных данных в программное обеспечение либо в виде протокольных пакетов, либо в виде события. Эта техника передачи случайного ввода является очень мощной для поиска ошибок во многих приложениях и сервисах.

Как проводить фазз-тестирование

Шаги для фазз-тестирования включают в себя следующие основные этапы тестирования:

Шаг 1. Определите целевую систему.
Шаг 2. Определить входы.
Шаг 3. Генерирование нечётких данных.
Шаг 4. Выполнение теста с использованием нечётких данных.
Шаг 5. Мониторинг поведения системы.
Шаг 6. Регистрация дефектов.

Тестирование не может продолжаться до тех пор, пока спецификация не станет зрелой.

Типы ошибок, обнаруживаемых с помощью фаззинга:

- Сбои и утечки памяти. Эта методика широко используется для больших приложений, где ошибки влияют на сохранность памяти, что является серьёзной уязвимостью.
- Недействительный ввод данных. Фаззеры используются для генерации недействительного ввода, который применяется для тестирования процедур обработки ошибок, что важно для программного обеспечения, которое не контролирует свой ввод. Простой фаззинг может быть известен как способ автоматизации негативного тестирования.
- Ошибки корректности. Фаззинг также может использоваться для обнаружения некоторых типов ошибок «корректности». Например, повреждённая база данных, плохие результаты поиска и т. д.

Преимущества Fuzz-тестирования

- Fuzz-тестирование улучшает тестирование безопасности программного обеспечения.
- Ошибки, найденные при Fuzz-тестировании, иногда бывают серьёзными и чаще всего используются хакерами, включая сбои, утечку памяти, необработанное исключение и т. д.
- Если какие-либо из ошибок не были замечены тестировщиками из-за нехватки времени и ресурсов, эти ошибки также обнаруживаются при Fuzz-тестировании.

Недостатки Fuzz-тестирования

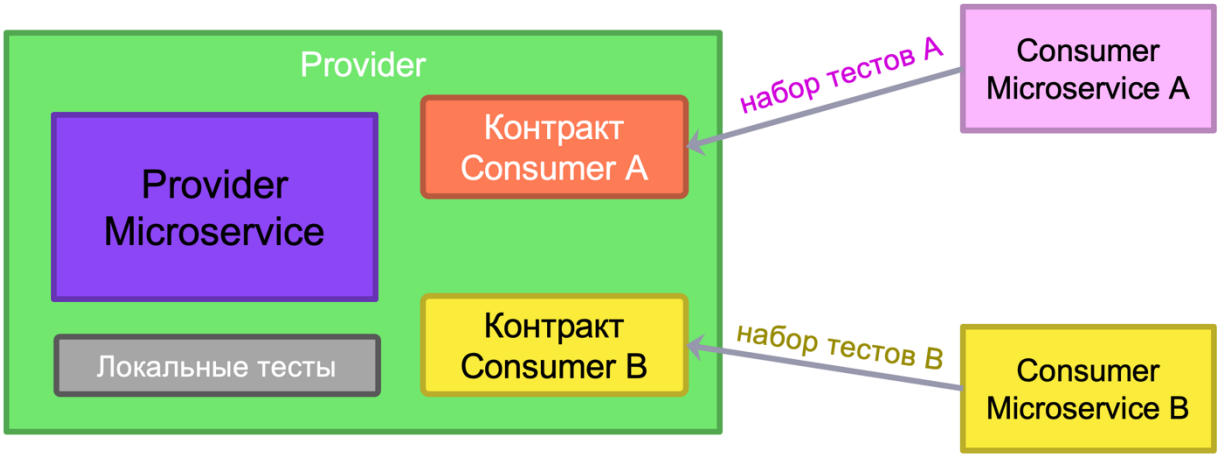
- Fuzz-тестирование само по себе не может дать полную картину общей угрозы безопасности или ошибок.
- Fuzz-тестирование менее эффективно для борьбы с угрозами безопасности, которые не вызывают сбоёв в работе программы, например, некоторые вирусы, черви, трояны и т. д.
- Fuzz-тестирование может обнаружить только простые ошибки или угрозы.
- Для его эффективного проведения потребуется значительное время.

Тестирование контрактов, ориентированных на потребителя

Это один из стилей тестирования, который рекомендуют использовать в крупномасштабных проектах, где несколько команд работают над различными сервисами.

Суть паттерна в том, что набор автоматизированных тестов для каждого сервиса (Provider Microservice) пишется разработчиками других сервисов (Consumer Microservice), вызывающих проверяемый сервис. Каждый такой набор тестов является контрактом, проверяющим, соответствует ли сервис провайдера ожиданиям потребителя. Сами тесты включают в себя запрос и ожидаемый ответ.

Паттерн Consumer-Driven Contract Testing увеличивает автономность команд и позволяет своевременно обнаруживать изменения в сервисах, написанных другими командами. Но его применение может потребовать дополнительной работы по интеграции тестов, так как команды могут пользоваться различными инструментами тестирования.



Consumer-Driven Contract Testing.

CDC включает три ключевых элемента:

- Контракт. Описывается с помощью некоторого DSL, зависит от реализации. Он содержит в себе описание API в виде сценариев взаимодействия: если пришёл определённый запрос, то клиент должен получить определённый ответ.
- Тесты клиентов. Причём в них используется заглушка, которая автоматически формируется из контракта.
- Тесты для API. Они также генерируются из контракта.

Таким образом, контракт — исполняемый. И основная особенность подхода заключается в том, что требования к поведению API идут upstream, от клиента к серверу.

Контракт фокусируется на том поведении, которое действительно важно потребителю. Делает явными его допущения относительно API.

Главная задача CDC — сблизить понимание поведения API его разработчиками и разработчиками его клиентов.

Этот подход хорошо сочетается с BDD (Behavior Driven Development), на встречах трёх амиго (Владелец продукта, Разработчик и Тестировщик) можно набрасывать заготовки для контракта. В конечном счёте этот контракт также служит:

- Улучшению коммуникаций.
- Разделению общего понимания проблемной области.
- Реализации решения внутри и между командами.

Порядок и терминология тестирования

Термины и сущности тестирования ПО

Тест план (Test Plan) — это документ, описывающий весь объём работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Тест дизайн — это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.

Роли, ответственные за тест дизайн:

- Тест аналитик — определяет «ЧТО тестировать?»
- Тест дизайнер — определяет «КАК тестировать?»

Этапы тестирования:

- Анализ предметной области и задачи, работа с требованиями.
- Разработка стратегии тестирования и планирование процедур контроля качества.
- Выбор инструментов тестирования.
- Создание тестовой документации и разработка тестов.
- Тестирование, наблюдение, фиксация результата.
- Обратная связь в команду проектирования и разработки.
- Доработка и стабилизация кода.
- Деплой в продуктовую среду.
- Эксплуатация и мониторинг ошибок.

Traceability matrix — Матрица соответствия требований — это двумерная таблица, содержащая соответсвие функциональных требований (functional requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки.

Матрица соответсвия требований используется QA-инженерами для валидации покрытия продукта тестами. МСТ является неотъемлемой частью тест-плана.

Тестовое покрытие — величина, выражающая процентное отношение функций, проверяемой тестами, к полной функциональности системы. Оценка тестового покрытия проводится при подготовке тестового плана, чтобы тестирование смогло обеспечить требуемый уровень тестового покрытия.

Тестовый случай (Test Case) — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Пример:

Action Expected Result Test Result(passed/failed/blocked)

Open page «login» Login page is opened Passed

Каждый тест кейс должен иметь 3 части:

PreConditions Список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.

Test Case Description Список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод об удовлетворении реализации, поставленным требованиям.

PostConditions Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state)

Виды Тестовых Случаев:

Тест кейсы разделяются по ожидаемому результату на позитивные и негативные:

- Позитивный тест кейс использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
- Негативный тест кейс оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

Чек-лист (check list) — это документ, описывающий что должно быть протестировано. При этом чек-лист может быть абсолютно разного уровня

детализации. На сколько детальным будет чек-лист зависит от требований к отчётности, уровня знания продукта сотрудниками и сложности продукта.

Как правило, чек-лист содержит только действия (шаги), без ожидаемого результата. Чек-лист менее формализован, чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Также чек-лист ассоциируются с гибкими подходами в тестировании.

Дефект (bug) — это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований.

Failure — сбой (причём не обязательно аппаратный) в работе компонента, всей программы или системы. То есть, существуют такие дефекты, которые приводят к сбоям (A defect caused the failure) и которые не приводят. UI-дефекты, например. Но аппаратный сбой, никак не связанный с software, тоже является failure.

Error — ошибка пользователя, то есть он пытается использовать программу иным способом.

Отчёт об ошибке (Bug Report) — это документ, описывающий ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Градация серьёзности дефекта (Severity)

Блокирующая (Blocker)

Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или её ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.

Критическая (Critical)

Критическая ошибка, неправильно работающая ключевая бизнес-логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы

необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.

Значительная (Major)

Значительная ошибка, часть основной бизнес-логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.

Незначительная (Minor)

Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

Обычная/Тривиальная (Trivial)

Тривиальная ошибка, не касающаяся бизнес-логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Дополнительные материалы

1. Brian Marick «The Craft Of Software Testing: Subsystems Testing Including Object Based And Object Oriented Testing». 1-st edition (December 8, 1994), — 553 pages.
2. Janet Gregory Lisa Crispin «MORE AGILE TESTING Learning Journeys for the Whole Team». 1-st edition (October 6, 2014),— 544 pages.
3. Хориков В. «Принципы юнит-тестирования». Издательский Дом ПИТЕР 2021, — 320 с.

Используемые источники

Книги

1. Роберт Мартин «Чистая архитектура. Искусство разработки программного обеспечения». Издательский Дом ПИТЕР, 2022, — 352 с.

2. Мартин, Р. Чистый код: создание, анализ и рефакторинг. Москва: Юпитер, 2018.— 464 с.
3. Фаулер М. Архитектура корпоративных приложений: Пер. с англ. — М: Издательский дом «Вильямс», 2006.— 544 с.
4. С. Макконнелл «Совершенный код» (Steve McConnell Code Complete). Издательство «Русская редакция», 2010. — 896 с.

Электронные ресурсы:

1. Диаграмма компонент https://www.lucidchart.com/pages/uml-component-diagram
2. Контрактное программирование https://ru.wikipedia.org/wiki/Контрактное_программирование, https://habr.com/ru/post/38612/
3. Проверка данных https://leprosus.medium.com/4-шага-проверки-входных-данных-приложением-9d7a6606a9fc
4. Volume testing https://artoftesting.com/volume-testing
5. Soak testing https://www.geeksforgeeks.org/soak-testing/
6. Functional testing https://www.a1qa.com/blog/the-a-to-z-guide-to-functional-testing/
7. Fuzzing https://medium.com/swlh/fuzzing-web-applications-e786ca4c4bb6
8. Brain Marick testing http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2