**Manual part 1**

**HR-часть**

**Вопросы с реальных собеседований с этапа HR**

Часть из них зададут в любом случае. Список, разумеется, не полный:

* Расскажи о себе (всё что хочешь, что нам нужно знать о тебе)
* Есть ли релевантный опыт?
* Какие курсы проходил и вообще, что изучал?
* Что не устраивало на прошлом месте работы (если было), особенно если решил сменить сферу?
* Почему выбрал именно тестирование?
* Почему выбрал именно нашу компанию?
* Как часто бываешь на собеседованиях?
* Уровень английского?
* (Если требуется и уровень хороший) расскажите на английском: как доехали до собеседования/о себе (только не как в обществе анонимных алкоголиков) /почему считаешь, что можешь стать тестировщиком/ как прошёл вчерашний день/о своих хобби/ и т.п.
* Как в целом смотришь на мир, как решаешь возникающие проблемы?
* 3 твоих сильных и 3 слабых стороны?
* Как отдыхаешь? Как проводишь свободное время?
* Какие хобби?
* Что последнее прочитал техническое? Не техническое?
* Если бы мог вернуться в начало осознанной жизни, выбрал бы иной карьерный путь?
* 3 примера, что тебе положительного дал предыдущий опыт работы (если есть)
* 3 плюса и 3 минуса в сфере тестирования лично для тебя
* Как видишь развитие в этой сфере, кем видишь себя через год, три?
* Какая-то одна вещь или ситуация, которой ты гордишься
* Представим, что остальных кандидатов много и они опытнее (обычно так и есть), может у тебя есть какие-то преимущества перед ними? Почему ты думаешь, что лучше других кандидатов?
* Зарплатные ожидания сейчас, после испытательного срока, через год?
* Есть ли какие-то факторы, с которыми ты согласишься на меньшие деньги?
* С чем точно не готов мириться в отношении компании или руководителя?
* Ожидания от работы?
* Отношение к переработкам?
* Представь, что ты работаешь уже полгода. Опиши свой рабочий день.
* Что если при выполнении задачи понимаешь, что не укладываешься в сроки?

**Общее о тестировании**

**Что означает тестирование ПО?**

**Тестирование** (testing) программного обеспечения (ПО) - с технической точки зрения заключается в выполнении приложения на некотором множестве исходных данных и сверке получаемых результатов с заранее известными (эталонными) с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам. Как одна из основных фаз процесса разработки программного продукта (Дизайн приложения - Разработка кода - Тестирование ), тестирование характеризуется достаточно большим вкладом в суммарную трудоемкость разработки продукта. Широко известна оценка распределения трудоемкости между фазами создания программного продукта: 40%-20%-40%, из чего следует, что наибольший эффект в снижении трудоемкости может быть получен прежде всего на фазах Design и Testing. В более широком смысле, тестирование ПО - это техника контроля качества программного продукта, включающая в себя проектирование тестов, выполнение тестирования и анализ полученных результатов.

**Почему требуется тестирование ПО?**

* Процесс тестирования гарантирует, что ПО будет работать в соответствии с ожиданиями клиентов и на имеющемся у них оборудовании.
* Это уменьшает циклы кодирования, выявляя проблемы на начальном этапе разработки. Обнаружение проблем на более ранних этапах SDLC обеспечивает правильное использование ресурсов и предотвращает повышение стоимости.
* Команда тестирования привносит взгляд клиента в процесс и находит варианты использования, о которых разработчик может не подумать.
* Любой сбой, дефект или ошибка, обнаруженные клиентом в готовом продукте, нарушают доверие к компании.

**Что означает обеспечение качества (Quality Assurance — QA) при тестировании ПО?**

Это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО и информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

**Что означает контроль качества (Quality Control — QC) при тестировании ПО?**

Это подмножество QA, совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии и соответствии ожидаемым результатам.

**Что означает качество ПО? (Software Quality)**

Качеству очень сложно дать неформальное определение, оно заключается в соответствии требованиям (conformance to requirements) и пригодности к использованию (fitness for use). Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения и продаж. Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта. То есть, основная последовательность действий при выборе и оценке критериев качества программного продукта включает:

* Определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта.
* Определение критериев, формирующих представление о качестве для каждого из участников.
* Приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника.
* Определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев.
* Определение способов и механизмов достижения каждого критерия.
* Определение стратегии тестирования исходя из набора критериев, попадающих под ответственность группы тестирования, выбранных приоритетов и целей.

Углубиться можно тут: <https://analytics.infozone.pro/software-quality/>  
<https://www.intuit.ru/studies/courses/48/48/lecture/1440?page=1>

**Объясните отличия в QA, QC и тестировании**

**Что означает Verification при тестировании ПО?**

Верификация (Verification) - это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

**Что означает Validation в тестировании ПО?**

Валидация (Validation) - это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе

**Разница между Design Verification и Design Validation?**

* Design Verification используется, когда фактический результат проекта должен быть таким же, как и ожидаемый результат и удовлетворять спецификациям продукта.
* Design Validation используется для определения того, что окончательный дизайн соответствует ожиданиям пользователя.
* Design Verification спрашивает: Правильно ли вы спроектировали продукт?
* Design Validation спрашивает: Правильный ли продукт вы разработали?
* Design Verification включает unit и primary integration level testing.
* Design Validation включает в себя secondary или higher-level integration и system level testing.
* Определенные аспекты Design Validation могут быть выполнены во время Design Verification, но Design Verification не заменяет Design Validation.
* Design Validation следует после успешного Design Verification.
* Design Verification может проводиться на отдельном модуле или на готовой системе при любых условиях. Design Validation должен проводиться при определенных условиях согласно требованиям пользователя.
* Design Verification может использовать статические методы. Он включает в себя проверку системы, анализ и формальную проверку (тестирование).
* Design Validation состоит из окончательного отчета (результатов выполнения теста), который проверяется, утверждается и подписывается. Эти документы хранятся для использования в будущем.

**Принципы тестирования?**

* Тестирование демонстрирует наличие дефектов
* Исчерпывающее тестирование недостижимо
* Раннее тестирование
* Скопление/кластеризация дефектов
* Парадокс пестицида
* Тестирование зависит от контекста
* Заблуждение об отсутствии ошибок
* \*Garbage in, garbage out (GIGO)

**Принцип 1**. Тестирование показывает наличие дефектов.

Тестирование может показать, что дефекты присутствуют, но не может доказать, что дефектов нет. Сколько бы успешных тестов вы не провели, вы не можете утверждать, что нет таких тестов, которые не нашли бы ошибку. Но если мы нашли хотя бы один дефект, мы уже можем утверждать, что в данном ПО присутствуют дефекты.

**Принцип 2.** Исчерпывающее тестирование невозможно.

Вместо попыток «протестировать всё» нам нужен некий подход к тестированию (стратегия), который обеспечит правильный объем тестирования для данного проекта, данных заказчиков (и других заинтересованных лиц) и данного продукта. При определении, какой объем тестирования достаточен, необходимо учитывать уровень риска, включая технические риски и риски, связанные с бизнесом, и такие ограничения проекта как время и бюджет. Оценка и управление рисками – одна из наиболее важных активностей в любом проекте.

**Принцип 3**. Раннее тестирование.

Тестовые активности должны начинаться как можно раньше в цикле разработки и быть сфокусированы на определенных целях. Этот принцип связан с понятием «цена дефекта» (cost of defect). Цена дефекта существенно растет на протяжении жизненного цикла разработки ПО. Чем раньше обнаружен дефект, тем быстрее, проще и дешевле его исправить. Дефект, найденный в требованиях, обходится дешевле всего. Еще одно важное преимущество раннего тестирования – экономия времени. Тестовые активности могут начинаться еще до того, как написана первая строчка кода. По мере того, как готовятся требования и спецификации, тестировщики могут приступать к разработке и ревью Test case. И когда появится первая тестовая версия, можно будет сразу приступать к выполнению тестов.

**Принцип 4.** Скопление дефектов.

Небольшое количество модулей содержит большинство дефектов, обнаруженных на этапе предрелизного тестирования, или же демонстрируют наибольшее количество отказов на этапе эксплуатации. Многие тестировщики наблюдали такой эффект – дефекты «кучкуются». Это может происходить потому, что определенная область кода особенно сложна и запутана, или потому, что внесение изменений производит «эффект домино». Это знание часто используется для оценки рисков при планировании тестов – тестировщики фокусируются на известных «проблемных зонах». Также полезно проводить анализ первопричин (root cause analysis), чтобы предотвратить повторное появление дефектов, обнаружить причины возникновения скоплений дефектов и спрогнозировать потенциальные скопления дефектов в будущем.

**Принцип 5**. Парадокс пестицида.

Если повторять те же тесты снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты. Повторное применение тех же тестов и тех же методик приводит к тому, что в продукте остаются именно те дефекты, против которых эти тесты и эти методики неэффективны. Чтобы преодолеть «парадокс пестицидов», необходимо регулярно пересматривать существующие тест-кейсы и создавать новые, разнообразные тесты, которые будут выполняться на различных частях системы.

**Принцип 6**. Тестирование зависит от контекста.

Тестирование выполняется по-разному, в зависимости от контекста. Например, тестирование систем, критических с точки зрения безопасности, проводится иначе, чем тестирование сайта интернет-магазина. Этот принцип тесно связан с понятием риска. Что такое риск? Риск – это потенциальная проблема. У риска есть вероятность (likelihood) – она всегда выше 0 и ниже 100% – и есть влияние (impact) – те негативные последствия, которых мы опасаемся. Анализируя риски, мы всегда взвешиваем эти два аспекта: вероятность и влияние. То же можно сказать и о мире ПО: разные системы связаны с различными уровнями риска, влияние того или иного дефекта также сильно варьируется. Одни проблемы довольно тривиальны, другие могут дорого обойтись и привести к большим потерям денег, времени, деловой репутации, а в некоторых случаях даже привести к травмам и смерти. Уровень риска влияет на выбор методологий, техник и типов тестирования.

**Принцип 7.** Заблуждение об отсутствии ошибок..

Нахождение и исправление дефектов бесполезно, если построенная система неудобна для использования и не соответствует нуждам и ожиданиям пользователей. Заказчики ПО – люди и организации, которые покупают и используют его, чтобы выполнять свои повседневные задачи – на самом деле совершенно не интересуются дефектами и их количеством, кроме тех случаев, когда они непосредственно сталкиваются с нестабильностью продукта. Им также неинтересно, насколько ПО соответствует формальным требованиям, которые были задокументированы. Пользователи ПО более заинтересованы в том, чтобы оно помогало им эффективно выполнять задачи. ПО должно отвечать их потребностям, и именно с этой точки зрения они его оценивают. Даже если вы выполнили все тесты и ошибок не обнаружили, это еще не гарантия того, что ПО будет соответствовать нуждам и ожиданиям пользователей. Иначе говоря, верификация != валидация.

\* **Принцип 8.** GIGO.

В компьютерной науке «garbage in – garbage out» (GIGO) — это концепция, в которой ошибочные или бессмысленные входные данные создают бессмысленный вывод или «мусор», т.е. при неверных входящих данных будут получены неверные результаты, даже если сам по себе алгоритм правилен. В тестировании такие случаи иногда создают намеренно, но я добавил этот принцип в общий список для того, чтобы подчеркнуть важность подготовки качественных тестовых данных, положительные они или отрицательные.

**Критерии выбора тестов?**

Требования к идеальному критерию тестирования:

* Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
* Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
* Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы.
* Критерий должен быть легко проверяемым, например вычисляемым на тестах.

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций. Поэтому мы стремимся к идеальному общему критерию через реальные частные. Классы критериев:

* Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика").
* Функциональные критерии формулируются в описании требований к программному изделию ( критерии так называемого "черного ящика" ).
* Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
* Мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

Структурные критерии используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing). Структурные критерии базируются на основных элементах УГП (Управляющий граф программы), операторах, ветвях и путях.

* Условие критерия тестирования команд (критерий С0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
* Условие критерия тестирования ветвей (критерий С1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.
* Условие критерия тестирования путей (критерий С2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или числом классов выходных путей).

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.  
Функциональный критерий - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика". Проблема функционального тестирования - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (Software requirement specification, Functional specification и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей. Ниже приведены частные виды функциональных критериев:

* Тестирование пунктов спецификации - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза. Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестировании должно быть проверено в соответствии с критерием не менее чем одним тестом.
* Тестирование классов входных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза. При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы - источники входных данных), мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия.
* Тестирование правил - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики. Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации фазы тестирования).
* Тестирование классов выходных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out). При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.
* Тестирование функций - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза. Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту). Критерий тестирования функций объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.
* Комбинированные критерии для программ и спецификаций - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза. При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

**Стохастическое тестирование** применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную мощность.

**Мутационный критерий** (класс IV). Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа - перестановка местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе. Подход базируется на следующих понятиях: Мутации - мелкие ошибки в программе. Мутанты - программы, отличающиеся друг от друга мутациями . Метод мутационного тестирования - в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P1, P2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y). Если на наборе (X,Y) подтверждается правильность программы P и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов (X,Y) соответствует мутационному критерию, а тестируемая программа объявляется правильной. Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов (X,Y) и продолжать тестирование.

Подробнее и с примерами в источнике: <https://www.intuit.ru/studies/courses/48/48/lecture/1428?page=1>

**Что подразумевается под тестовым покрытием? (Test Coverage)**

**Тестовое Покрытие** - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Сложность современного ПО и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более-менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.

Существуют следующие **подходы к оценке** и измерению тестового покрытия:

* Покрытие требований (Requirements Coverage) - оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).
* Покрытие кода (Code Coverage) - оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей ПО.
* Тестовое покрытие на базе анализа потока управления - это одна из техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых Test case для покрытия этих путей.

Различия: Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода — на полноте проверки тестами разработанной части продукта (исходного кода), а анализ потока управления - на прохождении путей в графе или модели выполнения тестируемых функций (Control Flow Graph).

***Ограничения:***

* Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом.
* Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

Альтернативное мнение:

**Покрытие кода** — совершенно бесполезная метрика. Не существует «правильного» показателя. Это вопрос-ловушка. У вас может быть проект со 100% покрытием кода, в котором по-прежнему остаются баги и проблемы. В реальности нужно следить за другими метриками — хорошо известными показателям CTM (Codepipes testing Metrics).

**PDWT** (процент разработчиков, пишущих тесты) — вероятно, самый важный показатель. Нет смысла говорить об антипаттернах тестирования ПО, если у вас вообще нет тестов. Все разработчики в команде должны писать тесты. Любую новую функцию можно объявлять сделанной только если она сопровождается одним или несколькими тестами.

PBCNT (процент багов, приводящих к созданию новых тестов). Каждый баг в продакшне — отличный повод для написания нового теста, проверяющего соответствующее исправление. Любой баг должен появиться в продакшне не более одного раза. Если ваш проект страдает от появления повторных багов даже после их первоначального «исправления», то команда действительно выиграет от использования этой метрики.

**PTVB** (процент тестов, которые проверяют поведение, а не реализацию). Тесно связанные тесты пожирают массу времени при рефакторинге основного кода.

**PTD** (процент детерминированных тестов от общего числа). Тесты должны завершаться ошибкой только в том случае, если что-то не так с бизнес-кодом. Если тесты периодически ломаются без видимой причины — это огромная проблема.

Если после прочтения о метриках вы по-прежнему настаиваете на установке жёсткого показателя для покрытия кода, я дам вам число 20%. Это число должно использоваться как эмпирическое правило, основанное на законе Парето. 20% вашего кода вызывает 80% ваших ошибок.

**Что такое модель зрелости тестирования (TMM — Test Maturity Model)?**

Существует определение Maturity Models, то есть модели зрелости различных процессов в организации, состоящая из 5 уровней. Нас же в рамках этого материала интересует один конкретный подвид моделей MM - модель зрелости тестирования, которая тоже состоит из 5 уровней. TMM в свою очередь основан на модели зрелости возможностей (CMM — Capability Maturity Model). Модель зрелости ПО (CMM или SW-CMM) — это модель для оценки зрелости программных процессов в организации. В ней также перечислены некоторые стандартные практики, которые увеличивают зрелость этих процессов. TMM это подробная модель для улучшения процесса тестирования. Она может быть дополнена любой моделью улучшения процесса или может использоваться как одиночная модель.

Модель ТММ имеет два основных компонента:

* Набор из 5 уровней, которые определяют возможности тестирования (testing capability)
* Модель оценки (An Assessment Model)

Пять уровней TMM помогают организации определить зрелость своего процесса и определить следующие шаги по улучшению, которые необходимы для достижения более высокого уровня зрелости тестирования. Приведенный далее текст является переводом, но есть и русскоязычные на эту тему. Однако материал везде несколько отличается, поэтому представляю несколько точек зрения: <https://devsday.ru/blog/details/5427> <https://www.software-testing.ru/library/around-testing/processes/3092-test-maturity-model> <https://habr.com/ru/company/otus/blog/479368/>

* **Уровень 1. Начальный.** ПО должно успешно работать.
  + На этом уровне области процессов не определены.
  + Цель тестирования — убедиться, что ПО работает нормально.
  + На этом уровне не хватает ресурсов, инструментов и обученного персонала.
  + Нет проверки качества перед поставкой ПО.
* **Уровень 2. Определенный**. Разработка целей и политик тестирования и отладки.
  + Этот уровень отличает тестирование от отладки, и они считаются различными действиями.
  + Этап тестирования наступает после кодирования.
  + Основная цель тестирования — показать, что ПО соответствует спецификации.
  + Основные методы и методики тестирования.
* **Уровень 3: Комплексный**. Интеграция тестирования в жизненный цикл ПО.
  + Тестирование интегрируется в весь жизненный цикл.
  + На основании требований определяются цели испытаний.
  + Структура тестирования существует.
  + Тестирование на уровне профессиональной деятельности.
* **Уровень 4: Управление и измерение**. Создание программы тестовых измерений.
  + Тестирование — это измеренный и количественный процесс.
  + Проверка на всех этапах разработки признается как тестирование.
  + Для повторного использования и регрессионного тестирования есть Test case и они записаны в базу тестов.
  + Дефекты регистрируются и получают уровни серьезности.
* **Уровень 5: Оптимизированный**. Оптимизация процесса тестирования.
  + Тестирование управляется и определено.
  + Эффективность и стоимость тестирования можно отслеживать.
  + Тестирование может постоянно настраиваться и улучшаться.
  + Практика контроля качества и предотвращения дефектов.
  + Практикуется процесс повторного использования (Reuse).
  + Метрики, связанные с тестированием, также имеют средства поддержки.
  + Инструменты обеспечивают поддержку разработки тестовых наборов и сбора дефектов.

**Что такое тестирование со сдвигом влево? (Shift left testing)**

В попытке перенести тестирование на более ранний этап жизненного цикла разработки при одновременном улучшении показателей качества, задачи смещаются влево по схеме жизненного цикла разработки ПО. По возможности, тестирование должно проводиться с самого начала фазы проектирования, чтобы построить соответствующую стратегию тестирования. Проще говоря, это подход к тестированию программного обеспечения и тестированию системы, при котором тестирование выполняется на более раннем этапе жизненного цикла. Ключевые преимущества:

* Сокращение затрат
* Более высокое качество
* Повышение эффективности
* Конкурентные преимущества

**Что такое независимое тестирование? (Independent testing)**

Можете ли вы доверять вердикту судьи, который является частью внутреннего круга людей, которых он должен судить? Чтобы этот процесс был справедливым, лица, принимающие решения, должны быть беспристрастными. Теперь, когда вы активно участвуете в разработке какого-либо продукта или программного обеспечения, тестировать его с нейтральным мышлением это легче сказать, чем сделать. Как разработчик, вы бы хотели отгружать продукт в кратчайшие сроки и считать его безупречным и в конечном итоге упустите из виду некоторые ошибки. Чтобы избежать такой ситуации, иногда следует нанять независимую организацию по тестированию, которая тщательно проверит ваш продукт на наличие сбоев, готовя его к развертыванию.

Тестирование по уровням независимости:

* Программист тестирует свой код
* Тестирование проводится другим программистом в организации
* Внутренняя команда тестирования
* Независимая организация тестирования
* Когда программист проверяет свой код: Вы бы никогда не попросили шеф-повара быть его собственным критиком. И даже если вы это сделаете, вам будет трудно поверить всему, что он говорит. Смысл — создатель никогда не может быть хорошим критиком своей собственной работы. Программист знает свой код от и до. Их цель - создать продукт и отправить его в кратчайшие сроки. Вместо того, чтобы искать ошибки со всех возможных точек зрения, они будут искушены найти способы обойти найденные ошибки и ошибки. Писатель Гленфорд Майерс в своей книге «Искусство тестирования программного обеспечения» перечислил разницу в мышлении разработчика и тестировщика. Он сказал, что разработчик думает как строитель, сосредоточенный на строительстве, в то время как тестировщик ищет недостатки, которые приведут к разрушению здания, если не будут решены.
* Тестирование проводится другим программистом в организации: Компромисс — это найти кого-то в организации. Это может быть какой-то другой программист, который участвует в некоторых других проектах. Это дает определенный уровень независимости. Но проблема возникает из-за того же reporting manager. Менеджер может попросить программиста пропустить некоторые тесты, когда есть ограничения по времени. Это приведет к неполному тестированию продукта. Кроме того, если попросить других разработчиков провести тестирование, это приведет к развертыванию различных ресурсов в одном проекте. Это будет вредно для всей работы организации.
* Внутренняя команда тестирования: Наличие другой внутренней команды — это хорошее решение. Но поскольку они будут в организации, на них будут влиять ограничительные сроки. Кроме того, это будет дорого поддерживать внутреннюю команду. Это приведет к большим бюджетным и ресурсным ограничениям для команды. Команда может иметь доступ к ограниченным инструментам и программному обеспечению, таким образом, не отвечая требованиям всех проектов. Среда тестирования также будет варьироваться в зависимости от количества пользователей и числа выполненных интеграций. Затем тестирование будет проводиться в спешном порядке, что приведет к упущению некоторых ошибок, которые могут появиться после выпуска продукта. Решение, которое позаботится обо всех этих недостатках, - «Независимое тестирование».
* Почему независимое тестирование? Независимые тестирующие организации изучат все аспекты вашей продукции. Они работают с мышлением поиска недостатков и ошибок. Они не будут использовать ярлыки в процессе тестирования. И поскольку они не были частью процесса разработки, они будут проводить тесты на нейтральной основе, чтобы прежние интересы не мешали процессу тестирования. Мысль о поиске максимальных «точек останова» пойдет на пользу вашему продукту. Почти все сторонние тестирующие организации предоставят вам подробные отчеты об ошибках и предложат корректирующие меры.

**В чем разница между превентивным и реактивным подходами в тестировании? (Preventative and Reactive approaches)**

* **Превентивный** (профилактический) подход: он также известен как Verification Process. Этот подход заключается в предотвращении дефектов. При таком подходе тесты разрабатываются на ранних этапах SDLC, то есть до того, как программное обеспечение было создано. Он подпадает под анализ качества (QA).
* **Реактивный** подход: он также известен как Validation Process. Этот подход заключается в выявлении дефектов. При таком подходе тесты предназначены для выполнения после того, как программное обеспечение было произведено. Здесь мы пытаемся найти недостатки. Подпадает под контроль качества (QC).

**Перечислите типичные возможные обязанности инженера по обеспечению качества?**

* Команда QA отвечает за мониторинг всего процесса разработки.
* Они несут ответственность за отслеживание результатов каждого этапа SDLC и корректировку их в соответствии с ожиданиями.
* Они несут ответственность за чтение и понимание необходимых документов.
* Анализируют требования к тестированию, а также разрабатываюти выполняют тесты.
* Разрабатывают Test case и расставляют приоритеты в тестировании.
* Записывают проблемы и инциденты в соответствии с задачами проекта и планами управления инцидентами.
* Работают с командой приложения и/или клиентом для решения любых проблем, возникающих в процессе тестирования.
* Проводят регрессионное тестирование каждый раз, когда в код вносятся изменения для исправления дефектов.
* Должны взаимодействовать с клиентами, чтобы лучше понять требования продукта.
* Принимают участие в прохождении процедур тестирования.

**Что такое аудит качества?**

Аудит качества — это процесс систематической и независимой проверки программного продукта или процесса для оценки соответствия спецификациям, стандартам, соглашениям и другим соответствующим критериям.

**Почему тестирование делится на отдельные этапы?**

* Каждый этап испытаний имеет свое назначение
* Проще управлять поэтапно
* Мы можем запустить разные тесты в разных средах
* Производительность и качество тестирования улучшаются с помощью поэтапного тестирования

**Почему невозможно полностью протестировать ПО?**

* Спецификации ПО могут быть субъективными и приводить к различным интерпретациям.
* ПО может потребоваться слишком много входов, слишком много выходов и слишком много комбинаций путей для тестирования.

**Как вы тестируете продукт, если требования еще не зафиксированы?**

Если спецификация требований недоступна для продукта, тогда план тестирования может быть создан на основе предположений, сделанных относительно продукта. Но мы должны хорошо документировать все предположения в плане тестирования.

**Как вы узнаете, было ли создано достаточно тестов для тестирования продукта?**

Прежде всего, мы проверим, охватывает ли каждое требование хотя бы один Test case. Если да, то можно сказать, что тестовых примеров достаточно для тестирования продукта.

**Как вы понимаете инспекцию?**

Это процесс проверки на уровне группы и улучшения качества документации по продукту. Он фокусируется на следующих двух аспектах:

* Улучшение документа продукта
* Улучшение процесса (как производства документов, так и проверки)

**Какие есть роли/должности в команде?**

Project manager

Это человек, который берет входящие от заказчика требования (несформулированные хотелки), уточняет всё и нормальным языком передаёт разработчикам, следит за рисками, прогрессом и доводит до финала. На нём вся коммуникация с заказчиком, согласования и т. д.

Product Owner

Может выполнять немного разные роли в разных компаниях. Человек, который является хранителем информации о продукте. Его роль быть decision maker, он отвечает со стороны бизнеса за приложение, с него будет спрашивать заказчик. Противоположные роли с ПМ, как адвокат с обвинителем. ПМ со стороны команды, пытается извертеться на плюшки, а PO со стороны заказчика выбивает всё по максимуму для себя.

Knowledge manager

Управление знаниями — это о том, как распоряжаться всеми имеющимися в компании знаниями, чтобы позволять всем сотрудникам максимально быстро находить ответы на вопросы, принимать решения, избегать ошибок, придумывать что-то новое, управлять проектами, подбирать и развивать сотрудников. А значит, нужно выстраивать коммуникации между подразделениями, учить их разговаривать друг с другом. Вопросы менеджера по знаниям ведущим специалистам имеют высший приоритет, поэтому тот всегда держит руку на пульсе. Полученные знания после доставки не теряются, а превращаются в обучающие документы, которые изучают все сотрудники.

**Опишите жизненный цикл продукта по этапам — какие участники на каждом этапе, какие у них роли? Какие артефакты на каждом этапе?**

Хотелось бы уточнить, что описанные этапы не являются эталонными и от компании к компании процессы могут радикально отличаться. В данном случае представлены некие «сферические процессы в вакууме» для какого-то начального понимания. Параллельно описывается SDLC и STLC.

* Анализ. Участники: Product owner, BA(бизнес-аналитик), QA. Артефакты: спецификация требований к ПО (Software Requirement Specification, SRS).

Во время этого этапа BA выясняет у PO все пожелания к продукту и документирует это в процессе обсуждения требований. Необходимо убедиться в том, что все участники правильно поняли поставленные задачи и то, как именно каждое требование будет реализовано на практике. Таким образом, этот этап предполагает сбор требований к разрабатываемому ПО, их систематизацию, документирование, анализ, а также выявление и разрешение противоречий. В компаниях, где налажены процессы обеспечения качества, уже на этом этапе включается в работу QA, т.к. бывают и дефекты требований.

* Проектирование. Участники: Product owner, разработчики, системные архитекторы, QA. Артефакты: дизайн-спецификация (Design Specification Document, DSD), Тест-план, тест-сценарии, тест-кейсы; вариативно: тестовая стратегия.

На стадии проектирования (называемой также стадией дизайна и архитектуры) программисты и системные архитекторы, руководствуясь требованиями, разрабатывают высокоуровневый дизайн системы. Разнообразные технические вопросы, возникающие в процессе проектирования, обсуждаются со всеми заинтересованными сторонами. Определяются технологии, которые будут использоваться в проекте, загрузка команды, ограничения, временные рамки и бюджет. В соответствии с уточненными требованиями выбираются наиболее подходящие проектные решения. Утвержденный дизайн системы определяет перечень разрабатываемых программных компонентов, взаимодействие с третьими сторонами, функциональные характеристики программы, используемые базы данных и многое другое. QA/test analyst проектируют процесс тестирования в тест плане, руководствуясь также (если есть) политикой и стратегией тестирования. Тестировщики начинают писать сценарии и по ним кейсы для тестирования.

* Разработка. Участники: разработчики. Артефакты: -.

Системные администраторы настраивают программное окружение, frontend программисты разрабатывают пользовательский интерфейс программы и логику ее взаимодействия с сервером.

Кроме того, программисты пишут Unit-тесты для проверки правильности работы кода каждого компонента системы, проводят ревью написанного кода, создают билды и разворачивают готовое ПО в программной среде. Этот цикл повторяется до тех пор, пока все требования не будут реализованы.

* Тестирование. Участники: QA. Артефакты: дефект-репорты, сводный отчет о тестировании.

Тестовый администратор (если есть) настраивает тестовые среды/стенды для проведения тестирования. Тестировщики занимаются поиском дефектов в программном обеспечении и сравнивают описанное в требованиях поведение системы с реальным. В фазе тестирования обнаруживаются пропущенные при разработке баги. При обнаружении дефекта, тестировщик составляет отчет об ошибке, который передается разработчикам. Последние его исправляют, после чего тестирование повторяется – но на этот раз для того, чтобы убедиться, что проблема была исправлена, и само исправление не стало причиной появления новых дефектов в продукте. По итогам проведенного процесса тестирования составляется итоговый отчет.

* Внедрение и сопровождение. Участники: команда технической поддержки. Артефакты: замечания, запросы на исправление/улучшение.

Когда программа протестирована и в ней больше не осталось серьезных дефектов, приходит время релиза и передачи ее конечным пользователям. После выпуска новой версии программы в работу включается отдел технической поддержки. Его сотрудники обеспечивают обратную связь с пользователями, их консультирование и поддержку. В случае обнаружения пользователями тех или иных пост-релизных багов, информация о них передается в виде отчетов об ошибках команде разработки, которая, в зависимости от серьезности проблемы, либо немедленно выпускает исправление (т.н. hot-fix), либо откладывает его до следующей версии программы. Кроме того, команда технической поддержки помогает собирать и систематизировать различные метрики – показатели работы программы в реальных условиях и т.п.

**Кто такой SDET?**

SDET (Software Development Engineer in Test) инженер по разработке ПО в тестировании — это ИТ-специалист, который может одинаково эффективно работать в сфере разработки и тестирования, и он / она принимает участие в полном процессе разработки ПО. В отличие от Quality Analyst (QA), которым желательны базовые знания в программировании, но нет необходимости писать код, SDET это делает на постоянной основе, совмещая в себе и тестировщика и разработчика.

|  |  |
| --- | --- |
| **SDET** | **Manual Tester** |
| Знает всю систему от начала до конца | Ограниченные знания о системе |
| SDET участвует в каждом этапе процесса разработки ПО, как Проектирование, разработка и тестирование. | QA участвует только в жизненном цикле тестирования процесса разработки ПО. |
| Высококвалифицированный специалист со знаниями как в разработке, так и в тестировании | Тестировщик ПО участвует только в подготовке и выполнении Test case |
| SDET может участвовать в разработке средств автоматизации тестирования | Не ожидается разработка средств автоматизации тестирования. |
| SDET должны выполнять такие обязанности, как тестирование производительности, автоматическое создание тестовых данных и т. д. | Только задачи по ручному тестированию |
| Знает требования и гайдлайны для продуктов | От QA таких знаний не ожидается. |

**Что такое тестирование как сервис? (TaaS – testing as a Service)**

ТЕСТИРОВАНИЕ КАК СЕРВИС (TaaS) — это модель аутсорсинга, при которой деятельность по тестированию передается третьей стороне. Здесь тестирование проводится сторонними подрядчиками, а не сотрудниками организации. TaaS используется, когда Компании не хватает навыков или ресурсов для внутреннего тестирования или чтобы получить свежий взгляд со стороны. Чаще всего на аутсорс отдают тестирование функциональности, производительности и безопасности.

**Что подразумевается под тестовой средой? (Test Environment/Test Bed)**

Вообще существует несколько сред:

* Среда разработки (Development Env) – в ней разработчики пишут код, проводят отладку, исправляют ошибки, выполняют Unit-тестирование. За эту среду отвечают также разработчики.
* Среда тестирования (Test Env) – в этой среде работают тестировщики. Тут тестируются новые билды. Здесь тестировщики проверяют функционал, проводят регрессионные проверки, воспроизводят ошибки.
* Интеграционная среда (Integration Env) – иногда реализована в рамках среды тестированя, а иногда в рамках превью среды. В этой среде собрана необходимая для end-to-end тестирования схема взаимодействующих друг с другом модулей, систем, продуктов. Собственно, необходима она для интеграционного тестирования. Поддержка среды – также, как и в случае со средой тестирования
* Превью среда (Preview, Preprod Env) – в идеале, это среда идентичная или максимально приближенная к продуктивной: те же данные, то же аппаратно-программное окружение, та же производительность. Она используется, чтобы сделать финальную проверку ПО в условиях максимально приближенным к «боевым». Здесь тестировщики проводят заключительное end-to-end тестирование функционала, бизнес и/или пользователи проводят UAT, а команды поддержки L3 и L2 выполняют DryRun (пробную установку релиза). Как правило за эту среду отвечает группа L3 поддержки.
* Продакшн среда (Prodaction Env) – среда, в которой работают пользователи. С этой средой работает команда L2 поддержки устанавливая поставки ПО или патчи с исправлениями, выполняя настройки, отвечая за работоспособность всех систем. Инциденты и проблемы требующие исправления ПО передаются в работу команде на L3

В общем случае среда тестирования — это настройка программного и аппаратного обеспечения для тестирования.

Испытательный стенд (Test Bed) – более глобальная сущность и включает в себя operating system, configuration management for the products, hardware, network topology и т. д. Настраиваются в соответствии с требованиями тестируемого приложения. В некоторых случаях испытательный стенд может представлять собой комбинацию тестовой среды и тестовых данных, которые он использует.

Настройка правильной среды тестирования гарантирует успех тестирования ПО. Любые недостатки в этом процессе могут привести к дополнительным затратам и времени для клиента. Следующие люди участвуют в настройке тестовой среды: Системные администраторы, Разработчики, Тестеры.

**Что подразумевается под тестовыми данными?**

Тестовые данные — это набор входных значений, необходимых для выполнения Test case. Тестеры определяют данные в соответствии с требованиями. Они могут сделать это вручную или использовать инструменты генерации.

**Основные фазы тестирования?**

* Pre-Alpha: — ПО является прототипом. Пользовательский интерфейс завершен. Но не все функции завершены. На данном этапе ПО не публикуется.
* Alpha: является ранней версией программного продукта. Цель — вовлечь клиента в процесс разработки. Хороший Альфа-тест должен иметь четко определенный план тестирования с комплексными тестовыми примерами. Это дает лучшее представление о надежности программного обеспечения на ранних стадиях. В некоторых случаях тестирование может быть передано на аутсорс.
* Beta: ПО стабильно и выпускается для ограниченной пользовательской базы. Цель состоит в том, чтобы получить отзывы клиентов о продукте и внести соответствующие изменения в ПО.
* Release Candidate (RC): основываясь на отзывах Beta Test, вы вносите изменения в ПО и хотите проверить исправления ошибок. На этом этапе вы не хотите вносить радикальные изменения в функциональность, а просто проверяете наличие ошибок. RC также выпущен для общественности
* Release: Все работает, ПО выпущено для общественности.

**Подробнее про бета-тестирование?**

Бета-тестирование происходит на конечных пользователях. Это нужно для обеспечения обратной связи.

Существуют различные типы бета-тестов в тестировании ПО, и они заключаются в следующем:

* Традиционное бета-тестирование: продукт распространяется на целевой рынок, и соответствующие данные собираются по всем аспектам. Эти данные могут быть использованы для улучшения продукта.
* Публичное бета-тестирование: продукт публикуется во внешнем мире через онлайн-каналы, и данные могут быть получены от любого пользователя. На основе обратной связи могут быть сделаны улучшения продукта.
* Техническое бета-тестирование: продукт передается во внутреннюю группу организации и собирает отзывы / данные от сотрудников организации.
* Целевая бета-версия: продукт выпущен на рынок для сбора отзывов об особенностях программы.
* Бета-версия после выпуска. Продукт выпущен на рынок, и данные собираются для внесения улучшений в будущем выпуске продукта.

**Что означает пилотное тестирование? (Pilot)**

PILOT testing определяется как тип тестирования программного обеспечения, который проверяет компонент системы или всю систему в режиме реального времени. Целью пилотного теста является оценка осуществимости, времени, стоимости, риска и эффективности исследовательского проекта. Это тестирование проводится точно между UAT и Production. В пилотном тестировании выбранная группа конечных пользователей пробует тестируемую систему и предоставляет обратную связь до полного развертывания системы. Другими словами, это означает проведение генеральной репетиции для последующего теста на удобство использования. Пилотное тестирование помогает в раннем обнаружении ошибок в Системе.

Пилотное тестирование связано с установкой системы на площадке заказчика (или в среде, моделируемой пользователем) для тестирования на предмет постоянного и регулярного использования. Выявленные недостатки затем отправляются команде разработчиков в виде отчетов об ошибках, и эти ошибки исправляются в следующей сборке системы. Во время этого процесса иногда приемочное тестирование также включается как часть тестирования на совместимость. Это происходит, когда система разрабатывается для замены старой.

**В чем отличие build от release?**

Билд это номер, даваемый ПО при передаче от разработчиков тестировщикам.

Релиз это номер, даваемый ПО при передаче конечному пользователю.

**Что такое бизнес – логика (domain)?**

Бизнес – логика (domain) это то, что конкретная программа по задумке должна сделать. Например, в складской программе проверка на возможность отправить товар (вдруг его нет в наличии). Это правила, которые должны соблюдаться в данной конкретной программе, определенные бизнес-клиентом. Слои приложения – слой пользовательского интерфейса, слой бизнес логики, слой сохранения данных.

**Ты – единственный тестировщик на проекте. Что делать?**

Начинать знакомство с проектом лучше с интервью. Станьте журналистом. Что из себя представляет структура организации, а именно кто над кем стоит и кто за что отвечает? Где больше всего багов, каким видят тестированиее сейчас и какие ожидания в будущем? Оцените зрелость процессов по CMM и TMM, зрелось проекта (новый/старый-зрелый/старые-зрелые где будут глобальные изменения) и команды, определите методологию разработки. Проведите вводную лекциию команде: что такое QA, как оно может помочь, с какими проблемами обращаться и зачем вообще оно надо. Далее в зависимости от всего этого ищем и смотрим соответствующий вебинар/статью и т.п., в крайнем случае можно поинтересоваться у коллег в тематических сообществах, например, в tg. Вообще создавать отдел тестирования обычно нанимают QA Lead, а если вы джун, то либо тут что-то не то, либо от вас хотят чего-то вполне конкретного и проводить целое расследование не придется - наверняка всё объяснят еще на собеседовании. Больше о построении процессов и роли QA можно узнать, например, отсюда:

<https://www.youtube.com/watch?v=RAC1bjWzIVk>  
<https://www.youtube.com/watch?v=54JtRR4GbPQ>  
<https://www.youtube.com/watch?v=4KT_ouolMQs>  
<https://www.youtube.com/watch?v=UW8sTq8SuFQ&feature=emb_logo>  
<https://www.youtube.com/watch?v=POYyrqpbr94&feature=emb_logo>  
<https://www.youtube.com/watch?v=dHGq4ago09k>  
<https://habr.com/ru/company/miro/blog/505282/>  
<https://habr.com/ru/post/439128/>  
<https://habr.com/ru/company/livetyping/blog/307860/>  
<https://dou.ua/lenta/articles/why-do-you-need-test-strategy/>

**Основные инструменты тестировщика?**

* DevTools
* Сниффер (например, Charles Proxy, Fidler)
* Тестирование API (например, Postman или SoapUI)
* Тестирование производительности (например, JMeter)
* Различные инструменты из категории тестирования безопасности (например, OWASP)
* Прочее: мессенджеры, баг-трекинговые системы, генераторы тестовых данных и т.п.

Что такое **DevTools**?

В каждый современный браузер встроены инструменты разработчика — они позволяют быстро отловить и исправить ошибки в разметке или в коде. С их помощью можно узнать, как построилось DOM-дерево, какие теги и атрибуты есть на странице, почему не подгрузились шрифты и многое другое:

* [Проверка ответа сервера](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#2)
* [Проверка мобильной адаптивности](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#3)
* [Проверка мобильной выдачи](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#4)
* [Региональная поисковая выдача](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#5)
* [Изменение дизайна](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#6)
* [Анализ протокола безопасности](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#7)
* [Анализ скорости загрузки страницы](https://siteclinic.ru/blog/seo-instrumenty/seo-chrome-devtools/#8)

**Виды тестирования**

**Какие существуют основные виды тестирования ПО?**

* Функциональные виды («Что?» — проверяет весь функционал продукта):
  + Функциональное тестирование (Functional testing)
  + Тестирование взаимодействия (Interoperability testing)
* Нефункциональное («Как?»):
  + Производительности (Performance)
    - Тестирование ёмкости/способностей (Capacity testing)
    - Стрессовое (Stress testing)
    - Нагрузочное (Load testing)
    - Объемное тестирование (Volume testing)
    - Выносливости (Soak/Endurance testing)
    - Стабильности/надежности (Stability / Reliability testing)
    - Шиповое (Spike)
    - Отказоустойчивости (Stability testing)
    - Масштабируемости (Scalability test)
  + Отказ и восстановление (Failover and Recovery testing)
  + Удобство пользования (Usability testing)
  + Тестирование установки (Installation testing)
  + Тестирование безопасности (Security and Access Control testing)
  + Конфигурационное (Configuration testing)
* Связанное с изменениями:
  + Регрессионное (Regression testing)
  + Санитарное или проверка согласованности/исправности (Sanity testing)
  + Дымовое (Smoke testing)
  + Тестирование сборки (Build Verification testing)

В разных источниках предлагается разный взгляд:

**Типы тестирования? (White/Black/Grey Box)**

Самым высоким уровнем в иерархии подходов к тестированию будет понятие типа, которое может охватывать сразу несколько смежных техник тестирования. То есть, одному типу тестирования может соответствовать несколько его видов. Отличаются они знанием внутреннего устройства объекта тестирования.

**Что означает тестирование черного ящика?**

Summary: Мы не знаем, как устроена внутри тестируемая система.

Тестирование методом «черного ящика», также известное как тестирование, основанное на спецификации или тестирование поведения – техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.

– тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы.

– тест-дизайн, основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

Почему именно «черный ящик»? Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Целью этой техники является поиск ошибок в таких категориях:

– неправильно реализованные или недостающие функции;

– ошибки интерфейса;

– ошибки в структурах данных или организации доступа к внешним базам данных;

– ошибки поведения или недостаточная производительности системы;

Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, ЧТО программа делает, а не на том, КАК она это делает.

Пример:

Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.

Поскольку это тип тестирования, по определению он может включать другие его виды. Тестирование черного ящика может быть как функциональным, так и нефункциональным. Функциональное тестирование предполагает проверку работы функций системы, а нефункциональное – соответственно, общие характеристики нашей программы.

Техника черного ящика применима на всех уровнях тестирования (от модульного до приемочного), для которых существует спецификация. Например, при осуществлении системного или интеграционного тестирования, требования или функциональная спецификация будут основой для написания тест-кейсов.

Техники тест-дизайна, основанные на использования черного ящика, включают:

– классы эквивалентности;

– анализ граничных значений;

– таблицы решений;

– диаграммы изменения состояния;

– тестирование всех пар.

Преимущества:

– тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;

– тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;

– тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;

– можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

– тестируется только очень ограниченное количество путей выполнения программы;

– без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;

– некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования;

Противоположностью техники черного ящика является тестирование методом белого ящика, речь о котором пойдет ниже.

**Что означает тестирование белого ящика?**

Тестирование методом белого ящика (также: прозрачного, открытого, стеклянного ящика; основанное на коде или структурное тестирование) – метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов.

Техника белого ящика применима на разных уровнях тестирования – от модульного до системного, но главным образом применяется именно для реализации модульного тестирования компонента его автором.

Преимущества:

– тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;

– можно провести более тщательное тестирование, с покрытием большого количества путей выполнения программы.

Недостатки:

– для выполнения тестирования белого ящика необходимо большое количество специальных знаний

Основным методом тестирования Белого ящика является анализ покрытия кода. Анализ покрытия кода устраняет пробелы в наборе тестовых примеров. Он определяет области программы, которые не покрываются набором Test case. После выявления пробелов вы создаете контрольные примеры для проверки непроверенных частей кода, тем самым повышая качество программного продукта.

* Охват операторов: — Этот метод требует, чтобы каждое возможное утверждение в коде было проверено хотя бы один раз в процессе тестирования разработки ПО.
* Покрытие ветвления — этот метод проверяет все возможные пути (если-еще и другие условные циклы) программного приложения.

Помимо вышесказанного, существует множество типов покрытия, таких как покрытие условий, покрытие нескольких условий, покрытие пути, покрытие функций и т. д. Каждый метод имеет свои достоинства и пытается протестировать (охватить) все части программного кода. Используя покрытие Statement и Branch, вы обычно достигаете 80-90% покрытия кода, что является достаточным.

**Что означает тестирование серого ящика? (Grey box)**

Тестирование методом серого ящика – метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции пользователя. Либо нам не доступна внутренняя реализация функций, но мы знаем на уровень ниже, чем пользователи – интерфейсы/эндпоинты и т.п.

Техника серого ящика применима на разных уровнях тестирования – от модульного до системного, но главным образом применяется на интеграционном уровне для проверки взаимодействия разных модулей программы.

Пример тестирования «серого ящика»: при тестировании веб-сайтов на битые ссылки, если тестер сталкивается с какой-либо проблемой с этими ссылками, он может сразу же внести изменения в HTML-код и проверить в реальном времени.

Методы:

* Матричное тестирование: этот метод тестирования включает в себя определение всех переменных, которые существуют в их программах.
* Регрессионное тестирование: чтобы проверить, повлияло ли изменение в предыдущей версии другие аспекты программы в новой версии.
* Тестирование ортогональных массивов или OAT: обеспечивает максимальное покрытие кода с минимальным количеством тестов.
* Pattern testing: это тестирование выполняется на данных истории предыдущих дефектов системы. В отличие от тестирования черного ящика, в тестировании серого ящика копаются в коде и определяют причину сбоя.

**Основные отличия White/grey/black box?**

**Что такое деструктивное/разрушающее/негативное тестирование? (DT — Destructive testing)**

ОТРИЦАТЕЛЬНОЕ ТЕСТИРОВАНИЕ — тип тестирования ПО для поиска точек отказа в программном обеспечении, который проверяет систему на обработку исключительных ситуаций (срабатывание валидаторов на некорректные данные), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора. Неожиданные условия могут быть чем угодно, от неправильного типа данных до хакерской атаки. Целью отрицательного тестирования является предотвращение сбоя приложений из-за некорректных входных данных. Просто проводя положительное тестирование, мы можем только убедиться, что наша система работает в нормальных условиях. Мы должны убедиться, что наша система может справиться с непредвиденными условиями, чтобы обеспечить 100% безошибочную систему.

Типичные примеры: ввести неправильно составленный e-mail и номер телефона, загрузить файл непредусмотренного расширения или размера.

Для деструктивного тестирования существует множество способов его тестирования:

* Метод анализа точек отказа: это пошаговое прохождение системы, проводящее оценку того, что может пойти не так в разных точках. Для этой стратегии может быть использована помощь BA (Business Analyst).
* Экспертная проверка тестировщика: проанализируйте или дайте на ревью ваши Test вашему коллеге-тестировщику, который менее знаком с системой/функцией
* Бизнес-анализ тестовых случаев. Конечные пользователи или эксперты могут подумать о многих допустимых сценариях, которые иногда тестировщики могут не учитывать или упустить, так как все их внимание будет сосредоточено на тестировании требований.
* Проведите предварительное тестирование с использованием контрольных таблиц (run sheets). Исследовательское тестирование с использованием контрольных таблиц поможет определить, что было проверено, повторить тесты и позволит вам контролировать охват тестами.
* Используйте другой источник: вы можете попросить кого-нибудь сломать программный продукт и проанализировать различные сценарии.

**Что такое недеструктивное/неразрушающее/позитивное тестирование? (NDT – Non Destructive testing)**

НЕДЕСТРУКТИВНОЕ ТЕСТИРОВАНИЕ — это тип тестирования программного обеспечения, который включает в себя правильное взаимодействие с программным обеспечением. Другими словами, неразрушающее тестирование (NDT) также можно назвать позитивным тестированием или тестированием «счастливого пути». Это дает ожидаемые результаты и доказывает, что программное обеспечение ведет себя так, как ожидалось. Пример: - Ввод правильных данных в модуль входа в систему и проверка, принимает ли он учетные данные и переходит на следующую страницу

**Что такое пирамида/уровни тестирования? (Testing Levels)**

«Пирамида тестов» — метафора, которая означает группировку тестов программного обеспечения по разным уровням детализации. Она также даёт представление, какое количество тестов должно быть в каждой из этих групп. Ее полезно увидеть мануальным тестировщикам.

… В тесте более высокого уровня вы не тестируете всю условную логику и пограничные случаи, которые уже покрыты юнит-тестами более низкого уровня. Убедитесь, что тест высокого уровня фокусируется только на том, что не покрыто тестами более низкого уровня.

Правило трёх А(AAA) (arrange, act, assert) или триада «дано, когда, тогда» — хорошая мнемоника, чтобы поддерживать хорошую структуру тестов.

**Что подразумевается под компонентным/модульным/юнит тестированием? (Component/Module/Unit testing)**

С этими терминами происходит путаница и даже глоссарий ISTQB не проясняет ситуацию. Обычно эти термины используют как синонимы, а конкретика варьируется от компании к компании. Но если копнуть и попробовать разобраться, получается примерно следующее:

Модульное тестирование (юнит-тестирование). Модульные тесты используются для тестирования какого-либо одного логически выделенного и изолированного элемента системы (отдельные методы класса или простая функция, subprograms, subroutines, классы или процедуры) в коде. Очевидно, что это тестирование методом белого ящика и чаще всего оно проводится самими разработчиками. Целью тестирования модуля является не демонстрация правильного функционирования модуля, а демонстрация наличия ошибки в модуле, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования. Изоляция тестируемого блока достигается с помощью заглушек (stubs), манекенов (dummies) и макетов (mockups). Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", модульное тестирование характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

* На основе анализа потока управления. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе структурных критериев тестирования С0, С1,С2. К ним относятся вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.
* На основе анализа потока данных, когда элементы, которые должны быть покрыты, определяются при помощи потока данных, т. е. информационного графа программы.

Тестирование на основе потока управления. Особенности использования структурных критериев тестирования С0,С1,С2 были рассмотрены ранее. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей - С1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая функция программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (call pair coverage).

Тестирование на основе потока данных. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий потока данных ). Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг информационного графа программы). Недостаток стратегии в том, что она не включает критерий С1, и не гарантирует покрытия решений.

Стратегия требуемых пар также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий СР, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в вершине УГП, а ее использование - на дугах, исходящих из решений, или в вычислительных вершинах.

Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

* Конструирование УГП.
* Выбор тестовых путей.
* Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами. На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей. Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

* Статические методы.
* Динамические методы.
* Методы реализуемых путей.

Статические методы. Самое простое и легко реализуемое решение - построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина управляющего графа программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый путь (входной тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток статических методов заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.

Динамические методы. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей динамических методов является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

Методы реализуемых путей. Данная методика заключается в выделении из множества путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство статических методов состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную, а эта работа достаточно трудоемкая. Динамические методы требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический интерпретатор, решатель неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень - реализуемость путей. Методы реализуемых путей дают самый лучший результат.

Компонентное тестирование — тип тестирования ПО, при котором тестирование выполняется для каждого отдельного компонента отдельно, без интеграции с другими компонентами. Его также называют модульным тестированием (Module testing), если рассматривать его с точки зрения архитектуры. Как правило, любое программное обеспечение в целом состоит из нескольких компонентов. Тестирование на уровне компонентов (Component Level testing) имеет дело с тестированием этих компонентов индивидуально. Это один из самых частых типов тестирования черного ящика, который проводится командой QA. Для каждого из этих компонентов будет определен сценарий тестирования, который затем будет приведен к Test case высокого уровня -> детальным Test case низкого уровня с предварительными условиями.

Исходя из глубины уровней тестирования, компоненное тестирование можно классифицировать как:

* Тестирование компонентов в малом (CTIS — Component testing In Small). Тестирование компонентов может проводиться с или без изоляции остальных компонентов в тестируемом программном обеспечении или приложении. Если это выполняется с изоляцией другого компонента, то это называется CTIS. Пример: веб-сайт, на котором есть 5 разных веб-страниц, тестирование каждой веб-страницы отдельно и с изоляцией других компонентов.
* Тестирование компонентов в целом (CTIL — Component testing In Large). Тестирование компонентов, выполненное без изоляции других компонентов в тестируемом программном обеспечении или приложении, называется CTIL. Давайте рассмотрим пример, чтобы понять это лучше. Предположим, что есть приложение, состоящее из трех компонентов, таких как Компонент A, Компонент B и Компонент C. Разработчик разработал компонент B и хочет его протестировать. Но для того, чтобы полностью протестировать компонент B, некоторые его функции зависят от компонента A, а некоторые — от компонента C. Функциональный поток: A -> B -> C, что означает, что существует зависимость от B как от A, так и от C, заглушка - вызываемая функция, а драйвер - вызывающая функция. Но компонент A и компонент C еще не разработаны. В этом случае, чтобы полностью протестировать компонент B, мы можем заменить компонент A и компонент C заглушкой и драйверами по мере необходимости. Таким образом, в основном, компоненты A & C заменяются заглушками и драйверами, которые действуют как фиктивные объекты до тех пор, пока они фактически не будут разработаны.

|  |  |
| --- | --- |
| Unit testing | Component testing |
| Тестирование отдельных программ, модулей, функций для демонстрации того, что программа выполняется согласно спецификации | Тестирование каждого объекта или частей программного обеспечения отдельно с или без изоляции других объектов |
| Проверка в(на) соответствии с design documents | Проверка в(на) соответствии с test requirements, use case |
| Пишутся и выполняются(обычно) разработчиками | Тестировщиками |
| Выполняется первым | Выполняется после Unit |

Другой источник:

Разница между компонентным и модульным тестированием: По-существу эти уровни тестирования представляют одно и тоже, разница лишь в том, что в компонентном тестировании в качестве параметров функций используют реальные объекты и драйверы, а в модульном тестировании — конкретные значения.

**Что подразумевается под интеграционным тестированием? (Integration testing)**

Интеграционное тестирование предназначено для проверки насколько хорошо два или более модулей ПО взаимодействуют друг с другом, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами). С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки ( Stub ) на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой. Больше: <https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=1>

Уровни интеграционного тестирования:

* Компонентный интеграционный уровень (Component Integration testing): Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
* Системный интеграционный уровень (System Integration testing): Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

* Подход Большого взрыва:
* Инкрементальный подход:
  + Нисходящий подход
  + Подход снизу-вверх
  + Сэндвич-подход

Некоторые утверждают, что всех участников (например, вызываемые классы) тестируемого субъекта следует заменить на имитации (mocks) или заглушки (stubs), чтобы создать идеальную изоляцию, избежать побочных эффектов и сложной настройки теста. Другие утверждают, что на имитации и заглушки следует заменять только участников, которые замедляют тест или проявляют сильные побочные эффекты (например, классы с доступом к БД или сетевыми вызовами). Иногда эти два вида юнит-тестов называют одинокими (solitary) в случае тотального применения имитаций и заглушек или общительными (sociable) в случае реальных коммуникаций с другими участниками.

Информация должна приходить в течение нескольких секунд или нескольких минут с быстрых тестов на ранних этапах конвейера. И наоборот, более длительные тесты — обычно с более широкой областью — размещаются на более поздних этапах, чтобы не тормозить фидбек от быстрых тестов. Как видите, этапы конвейера развёртывания определяются не типами тестов, а их скоростью и областью действия. Поэтому очень разумно может быть разместить некоторые из самых узких и быстрых интеграционных тестов на ту же стадию, что и юнит-тесты — просто потому что они дают более быструю обратную связь

Особенности интеграционного тестирования для объектно-ориентированного программирования: <https://www.intuit.ru/studies/courses/48/48/lecture/1434>

**Разница между Unit testing и Integration testing?**

Именно здесь больше всего споров о названиях. «Область» интеграционных тестов также весьма противоречива, особенно по характеру доступа к приложению (тестирование в чёрном или белом ящике; разрешены mock-объекты или нет). На практике получается так: если тест…

* использует базу данных,
* использует сеть для вызова другого компонента/приложения,
* использует внешнюю систему (например, очередь или почтовый сервер),
* читает/записывает файлы или выполняет другие операции ввода-вывода,
* полагается не на исходный код, а на бинарник приложения,

… то это интеграционный, а не модульный тест

Подведём итог: хотя теоретически можно использовать только интеграционные тесты, на практике

* Юнит-тесты легче поддерживать.
* Юнит-тесты легко воспроизводят пограничные случаи и редкие ситуации.
* Юнит-тесты выполняются гораздо быстрее интеграционных тестов.
* Сбойные юнит-тесты легче исправить, чем интеграционные.

Если у вас есть только интеграционные тесты, то вы впустую тратите и время разработки, и деньги компании. Нужны как модульные, так и интеграционные тесты одновременно. Они не взаимоисключающие.

**Что такое системное интеграционное тестирование? (SIT — System Integration testing)**

Это тип тестирования программного обеспечения, проводимого в интегрированной аппаратной и программной среде для проверки поведения всей системы. Это тестирование, проведенное на полной интегрированной системе для оценки соответствия системы ее установленным требованиям. SIT выполняется для проверки взаимодействия между модулями программной системы. Оно занимается проверкой требований к программному обеспечению высокого и низкого уровня, указанных в Software Requirements Specification/Data and the Software Design Document. Он также проверяет сосуществование программной системы с другими и тестирует интерфейс между модулями программного приложения. В этом типе тестирования модули сначала тестируются индивидуально, а затем объединяются в систему. Например, программные и / или аппаратные компоненты объединяются и тестируются постепенно, пока не будет интегрирована вся система.

**Что подразумевается под инкрементальным подходом? (Incremental Approach)**

При таком подходе тестирование выполняется путем объединения двух или более логически связанных модулей. Затем другие связанные модули поэтапно добавляются и тестируются для правильного функционирования. Процесс продолжается до тех пор, пока все модули не будут соединены и успешно протестированы. Осуществляется двумя разными методами: Снизу-вверх и сверху-вниз.

**Что подразумевается под подходом снизу-вверх? (Bottom-Up Approach)**

В восходящей стратегии каждый модуль на более низких уровнях последовательно тестируется с более высокоуровневыми модулями, пока не будут протестированы все модули. Требуется помощь драйверов для тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения.

Преимущества: Локализация ошибок проще. Не тратится время на ожидание разработки всех модулей, в отличие от подхода Большого взрыва

Недостатки: Критические модули (на верхнем уровне архитектуры ПО), которые контролируют поток приложения, тестируются последними и могут быть подвержены дефектам. Ранний прототип невозможен.

**Что подразумевается под подходом сверху-вниз? (Top-Down Approach)**

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.

Преимущества: Локализация неисправностей проще. Возможность получить ранний прототип. Основные недостатки дизайна могут быть найдены и исправлены в первую очередь. Недостатки: Нужно много заглушек. Модули на более низком уровне тестируются недостаточно.

**Что подразумевается под гибридным/сэндвич-подходом? (Sandwich Approach)**

Представляет собой комбинацию подходов сверху вниз и снизу-вверх. Здесь верхние модули тестируются с нижними модулями, а нижние модули интегрируются с верхними модулями и тестируются. Эта стратегия использует и заглушки и драйверы.

**Что подразумевается под подходом Большого взрыва? (Big Bang Approach)**

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если Test case и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования

**В чем разница между тест-драйвером и тест-заглушкой? (Test Driver and Test Stub)**

Тестовый драйвер — это фрагмент кода, который вызывает тестируемый программный компонент. Это полезно при тестировании по принципу «снизу-вверх». Тестовая заглушка - это фиктивная программа, которая интегрируется с приложением для полной функциональности. Они актуальны для тестирования, в котором используется нисходящий подход. Давайте возьмем пример.

1. Допустим, есть сценарий для проверки интерфейса между модулями A и B. Мы разработали только модуль-A. Затем мы можем проверить модуль-A, если у нас есть реальный модуль-B или фиктивный модуль для него. В этом случае мы называем модуль-B тестовой заглушкой.
2. Теперь модуль B не может отправлять или получать данные напрямую из модуля A. В таком сценарии мы перемещаем данные из одного модуля в другой, используя некоторые внешние функции, называемые Test Driver.

Заглушки и драйверы не реализуют всю логику программного модуля, а только моделируют обмен данными с вызывающим модулем. Заглушка: вызывается тестируемым модулем. Драйвер: вызывает модуль для тестирования.

**Что подразумевается под системным тестированием?**

Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача системного тестирования - в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.

Системное тестирование производится над проектом в целом с помощью метода "черного ящика". Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю.

Категории тестов системного тестирования:

* Полнота решения функциональных задач.
* Стрессовое тестирование - на предельных объемах нагрузки входного потока.
* Корректность использования ресурсов (утечка памяти, возврат ресурсов).
* Оценка производительности.
* Эффективность защиты от искажения данных и некорректных действий.
* Проверка инсталляции и конфигурации на разных платформах.
* Корректность документации

Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи. Системное тестирование относят к черному ящику.

Можно выделить два подхода к системному тестированию:

* на базе требований (requirements based): Для каждого требования пишутся [Test case](http://www.protesting.ru/testing/testcase.html), проверяющие выполнение данного требования.
* на базе случаев использования (use case based): На основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся [Test case](http://www.protesting.ru/testing/testcase.html), которые должны быть протестированы.

**Можем ли мы провести системное тестирование на любом этапе?**

Нет. Системное тестирование следует начинать, только если все модули написаны и работают правильно. Тем не менее, это должно произойти до UAT (пользовательского приемочного тестирования).

**Что такое функциональное тестирование?**

Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

Тестирование в перспективе «требования» использует спецификацию функциональных требований к системе как основу для дизайна Test case. В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии. Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в перспективе «бизнес-процессы» использует знание этих самых бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

Преимущества функционального тестирования:

* имитирует фактическое использование системы;

Недостатки функционального тестирования:

* возможность упущения логических ошибок в программном обеспечении;
* вероятность избыточного тестирования.

**Что такое тестирование совместимости/взаимодействия? (Compatibility/Interoperability testing)**

Тестирование взаимодействия — функциональное тестирование, проверяющее способность приложения/устройства взаимодействовать с одним и более компонентами/системами/устройствами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).

ПО с хорошими характеристиками взаимодействия может быть легко интегрировано с другими системами, не требуя каких-либо серьезных модификаций. В этом случае, количество изменений и время, требуемое на их выполнение, могут быть использованы для измерения возможности взаимодействия. Например, тестирование совместимости проводится между смартфонами и планшетами для проверки передачи данных через Bluetooth.

Существуют разные уровни тестирования совместимости:

* Аппаратное обеспечение: проверяет совместимость программного обеспечения с различными аппаратными конфигурациями.
* Операционные системы: Он проверяет ваше программное обеспечение на совместимость с различными операционными системами, такими как Windows, Unix\*, Mac OS и т. д.
* Программное обеспечение: проверяет ваше разработанное программное обеспечение на совместимость с другим программным обеспечением. Например, приложение MS Word должно быть совместимо с другими программами, такими как MS Outlook, MS Excel, VBA и т. д.
* Сеть: оценка производительности системы в сети с различными параметрами, такими как пропускная способность, скорость работы, емкость.
* Браузер: проверяет совместимость вашего сайта с различными браузерами, такими как Firefox, Google Chrome, Internet Explorer и т. д.
* Устройства: проверяет совместимость вашего программного обеспечения с различными устройствами, такими как устройства USB-порта, принтеры и сканеры, другие мультимедийные устройства и Bluetooth.
* Mobile: проверка совместимости вашего программного обеспечения с мобильными платформами, такими как Android, iOS и т. д.
* Версии программного обеспечения. Он проверяет совместимость вашего программного приложения с различными версиями программного обеспечения. Например, проверка вашего Microsoft Word на совместимость с Windows 7, Windows 7 SP1, Windows 7 SP2, Windows 7 SP3.

Существует два типа проверки версий:

* Тестирование обратной совместимости предназначено для проверки поведения разработанного аппаратного / программного обеспечения с использованием более старых версий аппаратного / программного обеспечения.
* Тестирование прямой совместимости заключается в проверке поведения разработанного аппаратного / программного обеспечения с использованием более новых версий аппаратного / программного обеспечения.

Пример тестирования взаимодействия:

* Подключите (connect) два или более устройств от разных производителей
* Проверьте связь между устройствами
* Проверьте, может ли устройство отправлять / получать пакеты или фреймы друг от друга
* Проверьте, правильно ли обрабатываются данные на уровне сети и объектов
* Проверьте, правильно ли работают реализованные алгоритмы
* Результат в порядке: проверьте следующий результат. Результат не в порядке: используйте инструменты мониторинга, чтобы обнаружить источник ошибки
* Отчет о результатах в тестовом отчете.

**Что такое тестирование на соответствие? (Conformance/Compilance testing)**

CONFORMANCE testing — это тип тестирования программного обеспечения, который удостоверяет, что система программного обеспечения соответствует стандартам и правилам, определенным IEEE, W3C или ETSI. Цель проверки соответствия состоит в том, чтобы определить, в какой степени отдельная реализация конкретного стандарта соответствует индивидуальным требованиям этого стандарта. Включает в себя:

* Производительность
* Функции
* Прочность (Robustness)
* Совместимость (Interoperability)
* Поведение системы

Тестирование соответствия может быть логическим или физическим, и оно включает в себя следующие типы тестирования:

* Тестирование на соответствие (Compliance testing)
* Нагрузочное тестирование ([Load testing](https://www.guru99.com/load-testing-tutorial.html))
* Стресс тестирование ([Stress testing](https://www.guru99.com/stress-testing-tutorial.html))
* Объемное тестирование (Volume testing)

|  |  |
| --- | --- |
| Conformance testing | Compliance testing |
| Conformance является формальным и точным способом тестирования стандартов | Compliance является неформальным и менее точным способом тестирования стандартов |
| Сертификация Conformance применима только к операционной системе, имеющей официальный Certification Authority | Операционная система, которая обеспечивает единый API (Portable Operating System Interface), считается совместимой |
| Conformance testing используется для тестирования системы, которая обеспечивает полную поддержку данных стандартов | Compliance testing используется для тестирования системы, обеспечивающей поддержку некоторых из указанных стандартов |

Тестирование соответствия также называется Type testing, который является формальным способом тестирования.

**Что такое нефункциональное тестирование?**

НЕФУНКЦИОНАЛЬНОЕ тестирование определяется как тип тестирования ПО для проверки нефункциональных аспектов ПО. Оно предназначено для проверки готовности системы по нефункциональным параметрам, которые никогда не учитываются при функциональном тестировании.

* Нефункциональное тестирование должно повысить удобство использования, эффективность, ремонтопригодность и portability продукта.
* Помогает снизить производственный риск и затраты, связанные с нефункциональными аспектами продукта.

Позволяет:

* оптимизировать способ установки, настройки, выполнения, управления и мониторинга продукта.
* Собирать и производить измерения и метрики для внутренних исследований и разработок.
* Улучшать и расширять знания о поведении продукта и используемых технологиях.

Основные нефункциональные типы тестирования:

* Производительности (Performance)
  + Стрессовое (Stress testing)
  + Тестирование ёмкости/способностей (Capacity testing)
  + Нагрузочное (Load testing)
  + Объемное тестирование (Volume testing)
  + Выносливости/стабильности/надежности (Soak/Endurance/Stability/Reliability testing)
  + Шиповое (Spike)
  + Масштабируемости (Scalability Test)
* Тестирование времени отклика (Response Time testing)
* Тестирование на отказоустойчивость (Failover testing)
* Тестирование совместимости (Compatibility testing)
* Тестирование на удобство пользования (Usability testing)
* Тестирование на поддерживаемость/ремонтопригодность (Maintainability testing)
* Тестирование безопасности (Security testing)
* Тестирование аварийного восстановления (Disaster Recovery testing)
* Тестирование на соответствие (Compliance testing)
* Тестирование переносимости (Portability testing)
* Тестирование эффективности (Efficiency testing)
* Базовое тестирование (Baseline testing)
* Тестирование документации (Documentation testing)
* Тестирование восстановления (Recovery testing)
* Интернационализация (Globalization/Internationalization testing)
* Тестирование локализации (Localization testing)

**Основные понятия в тестировании производительности?**

* Время задержки (Latency) — временной интервал между запросом и ответом. Например, у вашего сервиса время задержки составляет 100ms, что означает, что такому сервису потребуется 100 миллисекунд на обработку запроса и генерирование ответа. Как правило, чем ниже время задержки, тем лучше клиентский опыт.
* Время ответа (Response time) — время, необходимое для ответа на запрос
* Пропускная способность (Throughput) - фактическое количество запросов (или пользователей), которое может обработать система за определенное время. В то время как время задержки говорит вам только о времени, метрика пропускной способности информирует об объеме данных, полученных и обработанных в момент времени. Важно не отделять показатели времени задержки от пропускной способности, т.к. высокий показатель времени задержки часто прямо связан с увеличением показателей метрики пропускной способности. Пропускная способность обычно измеряется в rps – (кол-во) запросов в секунду (requests per second).
* Ширина пропускания канала (Bandwidth) - максимальное число запросов (или пользователей), которое может обработать система. В отличие от пропускной способности ширина пропускания канала измеряет максимальный объем, который может обработать приложение.
* Транзакции в секунду. Пользовательские транзакции – это последовательность действий пользователя в интерфейсе. Сравнивая реальное время прохождения транзакции с ожидаемой (или количество транзакций в секунду), вы сможете сделать вывод о том, насколько успешной системой было пройдено нагрузочное тестирование.
* Процент ошибок рассчитывается как отношение невалидных ответов к валидным за промежуток времени.
* Про Average, медианы, перцентили и т.п. углубляться в рамках этой статьи не буду, есть в гугле.

**Тестирование производительности клиентской части и серверной, в чем разница?**

Оценка скорости работы клиентской и серверной частей веб-приложения осуществляется двумя разными видами тестирования: для Frontend применяется тестирование клиентской части, или Client-Side testing, а для Back-end – тестирование серверной части.

Основная цель тестирования клиентской части состоит в измерении времени, необходимого браузеру, для загрузки HTML-страницы. Наиболее важными показателями здесь являются количество загружаемых данных, их объем, а также количество выполненных запросов.

Собрать данную статистику можно как с использованием встроенных инструментов браузера (DevTools), так и с помощью специализированных инструментов и онлайн-сервисов, которые позволяют замерить необходимые показатели с учетом интересующего региона.

Помимо общего веса страницы, инструменты предоставляют детализированную информацию по каждому из компонентов. Изучив параметры запросов, можно обнаружить ряд проблем, приводящих к ухудшению скорости отображения страницы. К примеру, подгружается слишком большая картинка и Javascript, или отправляется значительное количество запросов.

Другая необходимая проверка направлена на анализ заголовков кэширования, поскольку корректность его выполнения при повторном посещении страницы позволяет повысить скорость загрузки страницы до 80%.

Тестирование серверной части направлено на анализ выполнения запросов и получения соответствующего ответа от Back-end.

Цели данного вида тестирования:

* Измерить время отклика самых важных бизнес-транзакций;
* Определить предельный уровень допустимой нагрузки;
* Выявить «узкие» места в производительности системы;
* Составить рекомендации по улучшению производительности;
* Найти возможные дефекты, проявляющиеся только при одновременной работе большого количества пользователей.

**В общем виде что такое тестирование производительности?**

Это класс тестирования ПО, который фокусируется на производительности системы при определенной нагрузке. Он не ищет напрямую ошибки или дефекты. Он производит аналитику на основе эталонных тестов и предоставляет разработчику всю диагностическую информацию, необходимую для выявления проблем производительности и узких мест. При этом происходит:

* измерение времени выполнения выбранных операций при определенных 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

Важно понимать, что все подвиды тестирования производительности это, грубо говоря, одно и то же, просто в зависимости от конкретного подвида выбираются разные параметры (показатели нагрузки/пользователей, длительности и т.п.) и собираются соответствующие метрики. Точкой отсчета принято брать результаты Capacity testing.

В реальном мире проводят только часть из перечисленных подвидов в соответствии с бюджетом и приоритетами данного ПО, а параметры тестов и метрики могут корректироваться в разных ситуациях.

Небольшая заметка. Несмотря на необходимость понимания многих математических и статистических концепций, многие тестировщики и менеджеры либо не имеют достаточных знаний в области математики и статистики, либо не пользуются ими. Это приводит к значительным искажениям и неверной интерпретации результатов тестирования производительности. Поэтому хороший специалист должен обладать и смежными знаниями.

**Что такое тестирование емкости/способностей? (Capacity)**

Capacity — базовый тест, который обычно выполняется первым. Все последующие тесты на среднее время ответа, регенерацию системы, утилизацию ресурсов нужно выполнять с оглядкой на результаты Capacity. Емкость системы измеряется в rps (requests per second). Используемый подход: ступенчато повышаем нагрузку до момента, когда время ответа начнет расти экспоненциально. Экспоненциальный рост времени ответа, как правило, связан с полной утилизацией одного из ресурсов, из-за которого запросы вместо моментальной обработки выстраиваются друг за другом и ждут своей очереди на обработку.

Capacity point – точка, где перестает расти Throughput, но увеличивается Response Time, то есть система перестаёт справляться с запросами.

Исходя из этого тестирования выбираются значения для stress, load и soak/endurance тестов. Для stress берётся значение близкое к capacity point, но немного меньше. Для load количество пользователей из зоны деградации.

Важно, ваша цель, не получить кол-во rps, при котором всё взрывается, а понять, какой именно ресурс станет узким местом, при повышении нагрузки. Поэтому, если обстреливаемый вами сервер не покрыт мониторингом — можете даже не начинать тест. Общий подход к сбору телеметрии — чем больше метрик собирается, тем лучше.

В некоторых случаях Capacity называют так же Scalability (масштабируемость), но в целом это не равнозначные понятия.

**Что означает тестирование масштабируемости? (Scalability)**

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?

**Разница между тестированием ёмкости/способностей и тестированием масштабируемости? (Capacity vs Scalability)**

Тестирование масштабируемости — это тестирование программного приложения для измерения его способности увеличивать или уменьшать масштаб с точки зрения любых его нефункциональных возможностей. Тестирование производительности, масштабируемости и надежности обычно группируется аналитиками качества ПО.

Тестирование емкости измеряет, сколько пользователей может обработать приложение. Это подмножество тестирования масштабируемости, в котором при тестировании масштабируемости вы получите меру емкости приложения. Тестирование масштабируемости измеряет, насколько хорошо приложение справляется с растущим числом пользователей. Если вы тестируете масштабируемость до тех пор, пока приложение не выйдет из строя, у вас будет мера того, сколько пользователей (емкость) может обработать приложение.

**Расскажите о стрессовом тестировании? (Stress testing)**

Стрессовое тестирование выполняется самым первым, если нет отдельного Capacity тестирования, хотя по факту это всё равно будет Capacity, т.к. нагрузка берется «с потолка». Позволяет проверить насколько приложение и система в целом работоспособны в условиях высокой нагрузки. Нагрузка на систему будет возрастать непрерывно до тех пор, пока не будет достигнут один из критериев его остановки. Пример: стресс-тест банковской системы был остановлен при превышении отметки в 1500 пользователей, когда высокая загруженность процессора (более 80%) привела к увеличению среднего времени отклика в пять раз и массовому появлению ошибок HTTP(S).

**Расскажите о нагрузочном тестировании? (Load)**

Нагрузочное тестирование - это тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе. Этот тип тестирования производительности выполняется для диагностики поведения системы при увеличении рабочей нагрузки.

Нагрузка на систему обычно подается на протяжении 1-2 часов (в некоторых источниках: 4-8 часов, хотя это уже больше endurance), количество пользователей для нагрузочного теста берется из зоны деградации (в некоторых источниках: определяется в количестве 80% от результата максимальной производительности). 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.

**Что такое объемное тестирование? (Volume testing)**

Объемное тестирование предназначено для прогнозирования того, может ли система / приложение обрабатывать большой объем данных. Это тестирование сосредоточено на наполнении базы данных продукта в реальных сценариях использования.

Пример 1: отправка через POST-запросы очень большого количества данных.

Пример 2: как изменится производительность приложения спустя X лет, если аудитория приложения вырастет в Y раз?

**Тестирование выносливости/стабильности/надежности (Soak/Endurance/Stability/Reliability testing)**

Задачей тестирования стабильности является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит является проверка на утечки памяти, время отклика, правильность подключения и закрытия подключения к модулям (например, БД) и т.п. в течение длительного времени, чтобы гарантировать, что после длительного периода время отклика системы останется таким же или лучше, чем на начало теста. Этот тип тестирования выполняется в самом конце (а где-то по ночам). Так же он помогает управлять будущими нагрузками, ведь нам необходимо понять, сколько дополнительных ресурсов (таких как ЦП, емкость диска, использование памяти или пропускная способность сети) необходимы для поддержки использования в будущем.

**Что такое спайк/шиповое тестирование? (Spike)**

Этот вид тестирования предназначен для определения поведения системы при внезапном увеличении нагрузки (большого количества пользователей) на систему. Например, дни распродаж в интернет-магазине.

**Что такое тестирование устойчивости? (Resilence)**

Проверка устойчивости проводится для того, чтобы убедиться, что система способна вернуться в исходное состояние после кратковременного напряжения. Например, если в интернет-магазине действует скидка на определенные товары на короткое время, скажем, один час в день.

**Что такое тестирование времени отклика? (Response time testing)**

Время ответа относится ко времени, которое требуется одному системному узлу для ответа на запрос другого. Среднее время ответа — это среднее время, затрачиваемое на каждый запрос в оба конца. Пиковое время отклика помогает нам определить, какие компоненты потенциально проблематичны. Коэффициент ошибок - это математический расчет, который отображает процент проблемных запросов. Три критических значения времени отклика: 0,1 секунды, 1,0 секунды и 10 секунд.

**Что такое Ramp тестирование?**

Это метод тестирования, который предлагает ступенчато поднимать нагрузку до тех пор, пока система не выйдет из строя.

**Что такое тестирование хранилища? (Storage testing)**

Тестирование хранилища определяется как тип тестирования программного обеспечения, который проверяет, сохраняет ли тестируемое приложение соответствующие данные в соответствующих каталогах и достаточно ли у него места для предотвращения неожиданного завершения из-за недостатка дискового пространства. Это также называется тестированием производительности хранилища (Storage Performance testing).

Зачем оно нужно?

* Медленное хранилище означает медленное время отклика, длительные запросы и более низкую availability приложений.
* Медленное хранилище — это накладные расходы на обслуживание серверной инфраструктуры.
* Также помогает найти практическое ограничение хранилища перед деплоем.
* Помогает понять, как система будет реагировать при замене или обновлении аппаратного обеспечения.

Типы:

* Application testing: Тестирование приложений с примерами запросов в production like environment
  + Сравните время ответа OLTP
  + Сравните время выполнения batch
  + Сравните rates непрерывной потоковой передачи
* Application Simulation: Проведите тестирование с использованием стандартного программного обеспечения, аналогичного целевому приложению
  + Протестировать на пиковые значения IOPS для баз данных
  + Тест пика для data streaming environments
  + Проверка задержек хранилища для обмена сообщениями или других однопоточных приложений
* Benchmarking: Провести тестирование с использованием стандартного программного обеспечения.
  + Проверка на повреждение данных.

**Что такое тестирование на отказ и восстановление? (Failover and Recovery testing)**

Тестирование на отказ и восстановление (Failover and Recovery testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками ПО, отказами оборудования или проблемами связи/сети. Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу «24x7». Если Вы создаете продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования Вам просто не обойтись. Т.к. 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
* Незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации).
* Объявление или внесение в массивы данных невозможных или ошибочных элементов.
* Отказ носителей данных.

Стоит заметить, что тестирование на отказ и восстановление – это весьма продукт-специфичное тестирование. Разработка тестовых сценариев должна производиться с учетом всех особенностей тестируемой системы. Принимая во внимание довольно жесткие методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта.

**Что вы знаете о Тестировании удобства пользования? (Usability testing)**

Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам:

* производительность, эффективность (efficiency) — сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т. д. ? (меньше - лучше)
* правильность (accuracy) — сколько ошибок сделал пользователь во время работы с приложением? (меньше - лучше)
* активизация в памяти (recall) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени? (повторное выполнение операций после перерыва должно проходить быстрее чем у нового пользователя)
* эмоциональная реакция (emotional response) – как пользователь себя чувствует после завершения задачи — растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (положительная реакция - лучше)

Проверка удобства использования может проводиться как по отношению к готовому продукту, посредством тестирования черного ящика (black box testing), так и к интерфейсам приложения (API), используемым при разработке — тестирование белого ящика (white box testing). В этом случае проверяется удобство использования внутренних объектов, классов, методов и переменных, а также рассматривается удобство изменения, расширения системы и интеграции ее с другими модулями или системами. Использование удобных интерфейсов (API) может улучшить качество, увеличить скорость написания и поддержки разрабатываемого кода, и как следствие улучшить качество продукта в целом.

Отсюда становится очевидно, что тестирование удобства пользования может производиться на разных уровнях разработки ПО: модульном, интеграционном, системном и приемочном.

**Отличия тестирование на удобство пользования и тестирования доступности? (Usability Vs. Accessibility testing)**

USABILITY testing показывает, насколько проста в использовании и удобна система программного обеспечения. Здесь небольшой набор целевых конечных пользователей «использует» программную систему для выявления дефектов юзабилити. Основное внимание в этом тестировании уделяется простоте использования приложения пользователем, гибкости в управлении средствами управления и способности системы выполнять свои задачи. Это также называется тестированием пользовательского опыта (UX – user expirience). Это тестирование рекомендуется на начальном этапе разработки SDLC, что позволяет лучше понять ожидания пользователей. Исследования (Virzi, 1992 и Neilsen Landauer, 1993) показывают, что 5 пользователей достаточно для выявления 80% проблем с юзабилити, хотя некоторые исследователи предлагают другие цифры.

Тестирование доступности (accessibility testing) — это подмножество юзабилити-тестирования. Его цель — убедиться в том, что наш продукт удобен в использовании для людей с различными видами инвалидности или особенностей восприятия. Это могут быть проблемы со зрением, слухом или ограничения в подвижности рук.

Ваш продукт должен правильно работать с соответствующим ПО. Примеры такого программного обеспечения:

* Speech Recognition Software — ПО преобразует произнесенное слово в текст, который служит вводом для компьютера.
* Программа для чтения с экрана — используется для озвучивания текста, отображаемого на экране
* Программное обеспечение для увеличения экрана — используется для увеличения масштаба элементов и облегчения чтения для пользователей с нарушениями зрения.
* Специальная клавиатура, облегчающая ввод для пользователей, у которых проблемы с двигательными функциями.

Еще один из примеров — люди с цветовой слепотой (дальтонизмом). Эта особенность довольно широко распространена. Различными видами цветовой слепоты страдают около 8 % мужчин и 0,4 % женщин - не так уж мало!

Цвет не должен быть единственным способом передачи информации. Если вы используете цвет для того, чтобы, допустим, отобразить статус, эту информацию стоит продублировать еще каким-то образом — геометрическими фигурами, иконками или текстовым комментарием. Хорошая контрастность. Хорошая контрастность обеспечивает нормальную видимость элементов управления и текста даже для людей, не различающих те или иные оттенки. Есть отличный инструмент для тестирования веб-сайтов на предмет доступности для людей с различными формами цветовой слепоты: Color Blind Web Page Filter.

Если вы хотите сократить количество тестов, можно ограничиться только тремя фильтрами: дейтеранопия, протанопия и тританопия. Это наиболее выраженные формы цветовой слепоты (не считая крайне редкого черно-белого зрения). Остальные люди с особенностями цветовосприятия видят больше оттенков, и если ваш UI достаточно хорошо виден с этими тремя фильтрами, то и для остальных будет отображаться корректно.

Пример чек-листа:

* Предоставляет ли приложение клавиатурные эквиваленты для всех действий мышью и окон?
* Предоставляются ли инструкции как часть пользовательской документации или руководства? Легко ли понять и использовать приложение, используя документацию?
* Упорядочены ли вкладки логически для обеспечения плавной навигации?
* Предусмотрены ли сочетания клавиш для меню?
* Поддерживает ли приложение все операционные системы?
* Четко ли указано время отклика каждого экрана или страницы, чтобы конечные пользователи знали, как долго ждать?
* Все ли надписи правильно написаны?
* Являются ли цвета подходящим для всех пользователей?
* Правильно ли используются изображения или значки, чтобы их было легко понять конечным пользователям?
* Есть ли звуковые оповещения?
* Может ли пользователь настроить аудио или видео элементы управления?
* Может ли пользователь переопределить шрифты по умолчанию для печати и отображения текста?
* Может ли пользователь настроить или отключить мигание, вращение или перемещение элементов?
* Убедитесь, что цветовое кодирование никогда не используется в качестве единственного средства передачи информации или указания на действие
* Видна ли подсветка с инвертированными цветами?
* Тестирование цвета в приложении путем изменения контрастности
* Правильно ли слышат люди с ограниченными возможностями всё имеющее отношение к аудио и видео?
* Протестируйте все мультимедийные страницы без мультимедиа-оборудования.
* Предоставляется ли обучение пользователям с ограниченными возможностями, что позволит им ознакомиться с программным обеспечением или приложением?

**Что такое тестирование интерфейса? (UI testing)**

Это тип интеграционного теста, который проверяет, правильно ли установлена связь между двумя различными программными системами. Соединение, которое объединяет два компонента, называется интерфейсом. Этот интерфейс в компьютерном мире может быть чем угодно, как API, так и веб-сервисами и т. д. Тестирование этих подключаемых сервисов или интерфейса называется Тестированием интерфейса. На самом деле интерфейс — это программное обеспечение, состоящее из наборов команд, сообщений и других атрибутов, которые обеспечивают связь между устройством и пользователем.

Тестирование интерфейса включает в себя тестирование двух основных сегментов:

* Интерфейс веб-сервера и сервера приложений
* Интерфейс сервера приложений и базы данных

Во время тестирования интерфейса выполняются различные типы тестирования, которые могут включать:

* Workflow: он гарантирует, что механизм интерфейса обрабатывает ваши стандартные рабочие процессы как и ожидалось.
* Крайние случаи (Edge cases -unexpected values) — непредвиденные значения: это учитывается, когда тестирование включает дату, месяц и день в обратном порядке.
* Тестирование производительности, нагрузки и сети (Performance, load, and network testing): интерфейс с большим объемом может потребовать больше нагрузочного тестирования, чем интерфейс с низким объемом, в зависимости от механизма интерфейса и инфраструктуры подключения
* Отдельные системы (Individual systems): это включает в себя тестирование каждой системы в отдельности. Например, биллинговая система и система управления запасами для розничного магазина должны работать отдельно.

**Что такое тестирование рабочего процесса/воркфлоу? (Workflow testing)**

Это тип тестирования программного обеспечения, который проверяет, что каждый software workflow точно отражает данный бизнес-процесс. Workflow — это серия задач для получения желаемого результата, которая обычно включает несколько этапов или шагов. Для любого бизнес-процесса тестирование этих последовательных шагов определяется как «WorkFlow testing».

Например, убедитесь, что система может быть установлена на платформе пользователя и работает правильно. Тестирование рабочего процесса проводится поэтапно. Вот как вы будете выполнять Workflow testing:

* Начальная фаза (Inception phase): эта фаза включает начальное планирование испытаний и тестирование прототипа
* Фаза разработки (Elaboration phase): Эта фаза включает базовую архитектуру тестирования
* Фаза строительства (Construction phase): эта фаза включает в себя значительные испытания в каждой сборке
* Фаза перехода (Transition phase): Эта фаза включает в себя регрессионные тесты и повторные тесты исправлений

Тестирование workflow выполняется:

* Test engineer: планирует цели теста и график. Определяет Test case и процедуры. Оценивает результаты теста.
* Component engineer: Разработка тестовых компонентов. Автоматизирует некоторые тестовые процедуры.
* Integration Tester: Выполнение интеграционных тестов и выявление дефектов
* System Testers: Выполнение системных тестов и отчеты о дефектах

**Что вы знаете о пользовательском приемочном тестировании? (UAT – User Acceptance testing)**

Пользовательское приемочное тестирование (UAT) — это тип тестирования, выполняемый конечным пользователем или клиентом для проверки / принятия ПО перед его перемещением в production. UAT выполняется на заключительном этапе тестирования после выполнения функциональных, интеграционных и системных испытаний. Основной целью UAT является проверка end-to-end business flow. Он не фокусируется на косметических ошибках, орфографических ошибках или тестировании системы. Приемочное тестирование пользователя выполняется в отдельной среде тестирования с настройкой данных, аналогичных производственным. Это своего рода тестирование черного ящика, в котором будут участвовать два или более конечных пользователя. Этапы:

* Анализ бизнес-требований
* Создать плана тестирования UAT
* Определить Test Scenario
* Создать Test case UAT
* Подготовить Test Data (Production like Data)
* Запустить Test case
* Записать результаты
* Подтвердить бизнес-цели

**Что такое эксплуатационное приемочное тестирование? (OAT — Operational Acceptance testing)**

ИСПЫТАНИЕ НА ЭКСПЛУАТАЦИЮ (OAT) — это тип тестирования программного обеспечения, который оценивает операционную готовность программного приложения до его выпуска в производство. Целью эксплуатационного тестирования является обеспечение бесперебойной работы системы в ее стандартной операционной среде (SOE - standard operating environment). Это также называется Оперативное тестирование (Operational testing). Эксплуатационное приемочное тестирование обеспечивает соответствие системы и компонентов в стандартной операционной среде приложения (SOE). Типы OAT:

* Installation testing
* Load & Performance Test Operation
* Backup and Restore testing
* [Security testing](https://www.guru99.com/what-is-security-testing.html)
* Code Analysis
* Fail over testing
* Recovery testing
* End-to-End[Test Environment](https://www.guru99.com/test-environment-software-testing.html)Operational testing
* Operational Documentation Review

Примеры Test case:

* Резервные копии, сделанные на одном сайте, могут быть развернуты на тот же сайт
* Резервные копии, сделанные на одном сайте, можно развернуть на другом сайте.
* Внедрение любых новых функций в живую производственную среду не должно отрицательно влиять на целостность текущих производственных услуг.
* Процесс внедрения может быть воспроизведен с использованием действующей документации
* Каждый компонент может быть отключен и успешно запущен в согласованные сроки.
* Для оповещений — все критические оповещения должны идти в TEC и ссылаться на документ правильного разрешения.
* Оповещения созданы и выдаются при превышении согласованных пороговых значений
* Любая документация по восстановлению, созданная или измененная, включая сервисные диаграммы, действительна
* Это должно быть передано в соответствующие области поддержки.
* Любой компонент, на который влияет сбой, должен показывать рекомендуемый порядок перезапуска, время завершения и т. д.

**Расскажите об инсталляционном тестировании?**

Тестирование инсталляции (установки) направленно на проверку успешной инсталляции и настройки, а также обновления или удаления ПО, как десктопного, так и мобильного.

Тестирование инсталляции в большинстве своем не входит в Веб-тестирование, являясь специализированным тестированием установки приложений на различные операционные системы.

Следующие проверки должны быть выполнены для этапов:

Установка.

* Установка должна начаться при клике по кнопке, подтверждающей данное действие
* Установки во всех поддерживаемых окружениях и на всех поддерживаемых платформах
* Установки в неподдерживаемых окружениях, а также в нужных окружениях с некорректными настройками
* Права, которые требует инсталляция (чаще всего они должны быть админскими), проверить установить приложение как гость
* Установки в clean state (при отсутствии любых возможных связанных файлов и предыдущих версий)
* Подсчитывается ли при установке количество свободного места на диске и выдается ли предупреждение если места недостаточно
* Установки загруженного ранее приложения, а также прямая установка с использованием сети/беспроводного соединения
* Восстановится ли процесс установки при внезапном его прерывании (отключение устройства, отказ сети, отключение беспроводного соединения)
* Установка приложения, его запуск, удаление приложения должны возвращать систему в исходное состояние
* Распознается ли наличие в системе приложений/программ, необходимых для корректной работы устанавливаемого приложения
* Повторный запуск установки приложения при уже текущем должен выдавать корректное сообщение, двойная установка должна быть исключена
* Процесс установки может быть настраиваемый/дефолтный. Убедиться, что оба корректно работают
* Наличие кнопки, которая предложит сохранить приложение в определенную папку, а также указывает дефолтное местоположение («C:/programs/.»)
* Правильно ли установлены, сохранены ли в корректных папках файлы приложения
* Наличие созданных ярлыков, корректно ли они расположены
* После установки в системной вкладке « Программы и компоненты» должны быть доступны: название приложения, иконка, имя издателя, размер приложения, дата установки и номер версии
* Настройки переменных сред PATH
* Убедиться, что лицензионный ключ сохраняется в Windows Registry library
* Поддерживает ли приложение функции ‘UnInstall’, ‘Modify’, ‘ReInstall’ и корректно ли они работают
* Работа приложения с уже существующими DLL-файлами, с DLL-файлами приложений, которые необходимы для корректной работы устанавливаемого приложения
* Наличие информации/сообщение о том, когда истекает срок действия установленной пробной версии приложения

Обновление:

* Поддерживает ли приложение функцию обновления/автообновления
* При попытке установить ранее установленную версию приложения система должна ее распознать и выдать корректное сообщение
* Сохраняются ли пользовательские настройки при попытке загрузить новую версию/обновить старую версию
* При попытке обновить версию должны быть доступны функции удалить приложение и восстановить приложение
* Стандартные проверки как при первичной установке приложения
* Убедиться, что номер версии приложения сменился новым
* Запустить приложение и убедиться, что оно работает корректно

Откат до предыдущей версии:

* Попробовать установить старую версию на более новую
* Наличие корректного сообщения при попытке отката
* Убедиться, что приложение работает корректно

Удаление приложения:

* Не остается ли в системе никаких папок/файлов/ярлыков/ключей реестра после полного удаления приложения
* Корректно ли работает система после установки и последующего удаления приложения

**Что вы знаете о тестировании безопасности? (Security and Access Control testing)**

Это тип тестирования ПО, который выявляет уязвимости, угрозы и риски. Целью тестов безопасности является выявление всех возможных лазеек и слабых мест в ПО, которые могут привести к потере информации, доходов, репутации компании, сотрудников или клиентов. Общая стратегия безопасности основывается на трех основных принципах:

* Конфиденциальность — сокрытие определенных ресурсов или информации
* Целостность – ресурс может быть изменен только в соответствии с полномочиями пользователя
* Доступность — ресурсы должны быть доступны только авторизованному пользователю, внутреннему объекту или устройству

Тестирование безопасности обычно выполняет отдельный специалист по безопасности. В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

* попытки узнать пароль с помощью внешних средств;
* атака системы с помощью специальных утилит, анализирующих защиты;
* подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
* целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
* просмотр несекретных данных в надежде найти ключ для входа в систему.

При неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

Типы тестирования безопасности:

* Сканирование уязвимостей/оценка защищенности (Vulnerability Scanning) выполняется с помощью автоматизированного ПО для сканирования системы на наличие известных сигнатур уязвимостей.
* Сканирование безопасности (Security Scanning) включает в себя выявление слабых сторон сети и системы, а затем предоставляет решения для снижения этих рисков. Это сканирование может быть выполнено как ручным, так и автоматизированным.
* Тестирование на проникновение (Penetration testing) — этот тип тестирования имитирует атаку злоумышленника. Это тестирование включает анализ конкретной системы для проверки потенциальных уязвимостей при попытке внешнего взлома.
* Оценка рисков (Risk Assessment) тестирование включает анализ рисков безопасности, наблюдаемых в организации. Риски классифицируются как Низкие, Средние и Высокие. Это тестирование рекомендует меры по снижению риска.
* Аудит безопасности (Security Auditing) - внутренняя проверка приложений и операционных систем на наличие уязвимостей. Аудит также может быть выполнен путем построчной проверки кода
* Этический взлом (Ethical hacking) — совершается с целью выявления проблем безопасности в системе. Это делается White Hat хакерами - это специалисты по безопасности, которые использует свои навыки законным способом для помощи в выявлении дефектов системы, в отличии от Black Hat (преступников) или Gray Hat (что-то между).
* Оценка состояния (Posture Assessment) объединяет сканирование безопасности, этический взлом и оценки рисков, чтобы показать общее состояние безопасности организации.

|  |  |
| --- | --- |
| SDLC фаза | Security Processes |
| Requirements | Анализ безопасности для требований и проверка случаев злоупотребления / неправильного использования |
| Design | Анализ рисков безопасности для проектирования. Разработка плана тестирования с учетом тестирования безопасности |
| Coding and Unit testing | Статическое и динамическое тестирование безопасности и тестирование белого ящика |
| Integration testing | Тестирование черного ящика |
| System testing | Тестирование черного ящика и сканирование уязвимостей |
| Implementation | Тестирование на проникновение, сканирование уязвимостей |
| Support | Анализ воздействия патчей |

**Что означает оценка уязвимости/защищенности? (Vulnerability Assessment)**

Это процесс оценки рисков безопасности в программной системе с целью уменьшения вероятности угрозы. Уязвимость — это любые ошибки или слабости в процедурах безопасности системы, разработке, реализации или любом внутреннем контроле, которые могут привести к нарушению политики безопасности системы. Целью оценки уязвимости является снижение возможности несанкционированного доступа для злоумышленников (хакеров). Анализ проникновения зависит от двух механизмов, а именно от оценки уязвимости и тестирования на проникновение (VAPT - Vulnerability Assessment and Penetration testing).

Методы:

* Активное тестирование
  + Inactive testing, тестер вводит новые test data и анализирует результаты.
  + В процессе тестирования тестеры создают интеллектуальную модель процесса, и она будет расти и дальше во время взаимодействия с тестируемым программным обеспечением.
  + Выполняя тест, тестер будет активно вовлекаться в процесс обнаружения новых Test case и новых идей. Вот почему это называется Active testing.
* Пассивное тестирование
  + Пассивное тестирование — отслеживание результатов запуска тестируемого программного обеспечения без введения новых Test case или data.
* Тестирование сети
  + Тестирование сети — это процесс измерения и записи текущего состояния работы сети за определенный период времени.
  + Тестирование в основном проводится для прогнозирования работы сети под нагрузкой или для выявления проблем, создаваемых новыми сервисами.
  + Нам нужно проверить следующие характеристики сети:
    - Уровни утилизации
    - Количество пользователей
    - Использование приложения
* Распределенное Тестирование
  + Распределенные тесты применяются для тестирования распределенных приложений, то есть приложений, работающих с несколькими клиентами одновременно. По сути, тестирование распределенного приложения означает тестирование его клиентской и серверной частей по отдельности, но с помощью метода распределенного тестирования мы можем протестировать их все вместе.
  + Части теста будут взаимодействовать друг с другом во время теста. Это делает их синхронизированными соответствующим образом. Синхронизация является одним из наиболее важных моментов в распределенном тестировании.

**Расскажите подробнее о тестировании на проникновение? (Penetration testing)**

PENETRATION testing — это тип тестирования безопасности, который выявляет уязвимости, угрозы, риски в программном приложении, сети или веб-приложении, которые может использовать злоумышленник. Тестирование на проникновение показывает реальную картину существующей угрозы в системе безопасности и определяет уязвимости организации к ручным атакам. Проведение пентеста на регулярной основе позволит определить технические ресурсы, инфраструктуру, физические и кадровый арсенал содержащие в себе слабые аспекты, которые требуют развития и усовершенствования. Данный вид тестирования выполняется как вручную, так и автоматически и может быть как Black Box, так и Grey и White. Ввиду необходимости наличия специфических знаний и опыта для выполнения этого вида тестирования привлекается отдельный специалист – пентестер. Примеры кейсов тестирования на проникновение:

* Тест на проникновение в сеть:
  + выявлении уязвимостей сетевого и системного уровня;
  + определение неправильных конфигураций и настроек;
  + выявление уязвимости беспроводной сети;
  + мошеннические услуги;
  + отсутствие [надежных паролей](http://withsecurity.ru/kak-pridumat-nadezhnyj-parol-i-zapomnit-ego) и наличие слабых протоколов.
* [Тест на проникновение приложений](http://withsecurity.ru/pentest-mobilnyh-prilozheniy-na-chto-obratit-vnimanie):
  + выявление недостатков прикладного уровня;
  + подделка запросов;
  + применение злонамеренных скриптов;
  + нарушение работы управления сеансами;
  + и т.п.
* Тест на физическое проникновение:
  + взлом физических барьеров;
  + проверка и взлом замков;
  + нарушения работы и обход датчиков;
  + вывод из строя камер видеонаблюдения;
  + и т. д.

**Отличия Vulnerability Assessment от Penetration testing?**

Оба этих вида отличаются друг от друга по силе и задачам, которые они выполняют. Однако для составления исчерпывающего отчета по тестированию уязвимостей рекомендуется сочетание обеих процедур.

|  |  |  |
| --- | --- | --- |
|  | Vulnerability Assessment | Penetration testing |
| Working | Нахождений уязвимостей | Выявление и использование уязвимостей |
| Mechanism | Обнаружение и сканирование | Симуляция |
| Focus | Ширина | Глубина |
| Coverage of Completeness | Высокое покрытие | Низкое |
| Cost | Низкая стоимость | Высокая |
| Performed By | Внутренний персонал | Атакующий или пентестер |
| How often to Run | После каждой сборки | Реже, зависит от политики безопасности компании и продукта |
| Result | Предоставит частичную информацию об уязвимостях | Предоставит полную информацию об уязвимостях |
|  |  |  |

**Что такое Fuzz тестирование?**

FUZZ testing (fuzzing) – это метод тестирования ПО методом черного ящика, один из типов тестирования безопасности, который вводит недействительные или случайные данные, называемые FUZZ, в систему программного обеспечения для обнаружения ошибок кодирования и лазеек в безопасности. Данные вводятся с использованием автоматических или полуавтоматических методов тестирования, после чего система отслеживается на предмет различных исключений, таких как сбой системы или сбой встроенного кода и т. д.

Обычно fuzzing обнаруживает наиболее серьезные ошибки или дефекты безопасности. Это очень экономически эффективный метод тестирования. Fuzzing — один из самых распространенных методов хакеров, используемых для обнаружения уязвимости системы (сюда относятся популярные SQL- или скриптовые инъекции). Примеры фаззеров:

* Mutation-Based Fuzzers: изменяет существующие образцы данных для создания новых test data. Это очень простой и понятный подход, он начинается с действительных образцов и постоянно корректирует каждый байт или файл.
* Generation-Based Fuzzers: определяет новые данные на основе ввода модели. Он начинает генерировать ввод с нуля на основе спецификации.
* PROTOCOL-BASED-fuzzer: самый успешный фаззер — это детальное знание тестируемого формата протокола. Понимание зависит от спецификации. Это включает в себя запись массива спецификации в инструмент, а затем с помощью метода генерации тестов на основе модели проходится спецификация и добавляется неравномерность в содержимое данных, последовательность и т. д. Это также известно как синтаксическое тестирование, грамматическое тестирование, тестирование надежности, и т. д. Fuzzer может генерировать Test case из существующего или использовать допустимые или недействительные входные данные.

Типы ошибок, обнаруживаемых Fuzz testing:

* Сбои ассертов и утечки памяти (Assertion failures and memory leaks). Эта методология широко используется для больших приложений, где ошибки влияют на безопасность памяти, что является серьезной уязвимостью.
* Некорректный ввод (Invalid input). Фаззеры используются для генерирования неверного ввода, который используется для тестирования процедур обработки ошибок, и это важно для программного обеспечения, которое не контролирует его ввод. Простой фаззинг может быть способом автоматизации отрицательного тестирования.
* Исправление ошибок (Correctness bugs). Fuzzing также может использоваться для обнаружения некоторых типов ошибок «правильности». Например, поврежденная база данных, плохие результаты поиска и т. д.

**Можно ли отнести тестирование безопасности или нагрузочное тестирование к функциональным видам тестирования?**

Данные виды во многих источниках относят к нефункциональным видам тестирования, но если это является основной функцией приложения, то можно отнести и к функциональным. Пример: менеджер паролей, основная функция приложения – безопасность.

**Что вы знаете о конфигурационном тестировании? (Configuration testing)**

Конфигурационное тестирование (Configuration testing) — специальный вид тестирования, направленный на проверку работы ПО при различных аппаратных и программных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т. д. )

В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

* Проект по профилированию работы системы Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.
* Проект по миграции системы с одной платформы на другую Цель Тестирования: Проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Для клиент-серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для некоторых типов приложений может быть актуален только один):

* Серверный
* Клиентский

На первом (серверном) уровне, тестируется взаимодействие выпускаемого ПО с окружением, в которое оно будет установлено:

* Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т. д.)
* Программные средства (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т. д.)

Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность).

На следующем (клиентском) уровне, ПО тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

* Тип, версия и битность операционной системы (подобный вид тестирования называется кроссплатформенное тестирование)
* Тип и версия Web барузера, в случае если тестируется Web приложение (подобный вид тестирования называется кросс-браузерное тестирование)
* Тип и модель видео адаптера (при тестировании игр это очень важно)
* Работа приложения при различных разрешениях экрана
* Версии драйверов, библиотек и т. д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки)

и т. д.

Перед началом проведения конфигурационного тестирования рекомендуется:

* создавать матрицу покрытия (матрица покрытия - это таблица, в которую заносят все возможные конфигурации),
* проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится),
* шаг за шагом, в соответствии с расставленными приоритетами, проверять каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов нам необходимо будет провести. В связи с этим, рекомендуем, по возможности, автоматизировать этот процесс, так как именно при конфигурационном тестировании автоматизация реально помогает сэкономить время и ресурсы. Конечно же автоматизированное тестирование не является панацеей, но в данном случае оно окажется очень эффективным помощником.

В итоге: конфигурационным называется тестирование совместимости выпускаемого продукта (ПО) с различным аппаратным и программным средствами.

Основные цели — определение оптимальной конфигурации и проверка совместимости приложения с требуемым окружением (оборудованием, ОС и т. д.). Автоматизация конфигурационного тестирования позволяет избежать лишних расходов

Примечание: в ISTQB вообще не говорится о таком виде тестирования как конфигурационное. Согласно глоссарию, данный вид тестирования рассматривается там как тестирование портируемости: configuration testing: See portability testing. portability testing: The process of testing to determine the portability of a software product.

**Что вы знаете про регрессионное тестирование? (Regression testing)**

При корректировках программы необходимо гарантировать сохранение качества. Для этого используется регрессионное тестирование - дорогостоящая, но необходимая деятельность в рамках этапа сопровождения, направленная на перепроверку корректности измененной программы. В соответствии со стандартным определением, регрессионное тестирование - это выборочное тестирование, позволяющее убедиться, что изменения не вызвали нежелательных побочных эффектов, или что измененная система по-прежнему соответствует требованиям.

Главной задачей этапа сопровождения является реализация систематического процесса обработки изменений в коде. После каждой модификации программы необходимо удостовериться, что на функциональность программы не оказал влияния модифицированный код. Если такое влияние обнаружено, говорят о регрессионном дефекте. Для регрессионного тестирования функциональных возможностей, изменение которых не планировалось, используются ранее разработанные тесты. Одна из целей регрессионного тестирования состоит в том, чтобы, в соответствии с используемым критерием покрытия кода (например, критерием покрытия потока операторов или потока данных), гарантировать тот же уровень покрытия, что и при полном повторном тестировании программы. Для этого необходимо запускать тесты, относящиеся к измененным областям кода или функциональным возможностям.

Другая цель регрессионного тестирования состоит в том, чтобы удостовериться, что программа функционирует в соответствии со своей спецификацией, и что изменения не привели к внесению новых ошибок в ранее протестированный код. Эта цель всегда может быть достигнута повторным выполнением всех тестов регрессионного набора, но более перспективно отсеивать тесты, на которых выходные данные модифицированной и старой программы не могут различаться. Важной задачей регрессионного тестирования является также уменьшение стоимости и сокращение времени выполнения тестов.

Можно заключить, что регрессионное тестирование выполняется чтобы минимизировать регрессионные риски. То есть, риски того, что при очередном изменении продукт перестанет выполнять свои функции. С регрессионным тестированием плотно связана другая активность - импакт анализ (или иначе, анализ влияния изменений). Обычно под импакт анализом имеют в виду одно из следующих:

1. Попытку оценить регрессионные риски ещё на этапе планирования изменений (этим определением, по моему опыту, чаще пользуются менеджеры и разработчики);
2. Попытку определить объем регрессионного тестирования с учетом изменений, которые уже произошли (это определение чаще используют сами тестировщики). У Пола Джеррарда есть серия статей, где более детально раскрывается понятие импакт анализа, причем не только с позиции тестировщика.  
   Очевидно, что от эффективности импакт анализа зависит эффективность регрессионного тестирования. Но не всегда тщательно проведенный импакт анализ позволяет сократить затраты на последующее тестирование.

Обоснование корректности метода отбора тестов. Перечислим некоторые особенности реализации регрессионного тестирования:

* Некоторые участки кода программы не получают управление при выполнении некоторых тестов.
* Если участок кода реализует требование, но измененный фрагмент кода не получает управления при выполнении теста, то он и не может воздействовать на значения выходных данных программы при выполнении данного теста.
* Даже если участок кода, реализующий требование, получает управление при выполнении теста, это далеко не всегда отражается на выходных данных программы при выполнении данного теста. Действительно, если изменяется первый блок программы, например, путем добавления инициализации переменной, все пути в программе также изменяются, и, как следствие, требуют повторного тестирования. Однако может так случиться, что только на небольшом подмножестве путей действительно используется эта инициализированная переменная.
* Не каждый тест, проверяющий код, находящийся на одном пути с измененным кодом, обязательно покрывает этот измененный код.
* Код, находящийся на одном пути с измененным кодом, может не воздействовать на значения выходных данных измененных модулей программы.
* Не всегда каждый оператор программы воздействует на каждый элемент ее выходных данных.

Предположим, что изменения в программе ограничиваются одним оператором. Если при выполнении какого-либо теста на исходной программе этот оператор никогда не получает управление, можно с уверенностью сказать, что он не получит управление и в ходе выполнения теста на новой программе, а результаты тестирования новой и старой программ будут совпадать. Следовательно, нет необходимости выполнять этот тест на новой программе. Указанный метод легко можно обобщить для случая нескольких изменений: если тест не задействует ни одного измененного оператора, и его входные данные не изменились, код, выполняемый им в измененной программе, будет в точности таким же, как в первоначальной версии. Такой тест не выявляет различий между двумя версиями системы; следовательно, нет необходимости прогонять его повторно. Если тест не затрагивает ни одного оператора вывода, поведение которого зависит от измененных операторов, это означает, что, несмотря на изменения в программе, все операторы, которые получают управление при выполнении этого теста, не изменят вывод системы по отношению к предыдущей версии. Таким образом, нет необходимости повторно прогонять и тесты такого рода.

Следовательно, необходимо ориентироваться на выбор только тех тестов, которые покрывают измененный код, воздействующий, в свою очередь, на вывод программы. Такой подход гарантирует, что будут выбраны только тесты, обнаруживающие изменения, и метод будет, как говорят, точным.

Классификация тестов при отборе.

Создание наборов регрессионных тестов рекомендуется начинать с множества исходных тестов. При заданном критерии регрессионного тестирования все исходные тесты подразделяются на четыре подмножества:

* Множество тестов, пригодных для повторного использования. Это тесты, которые уже запускались и пригодны к использованию, но затрагивают только покрываемые элементы программы, не претерпевшие изменений. При повторном выполнении выходные данные таких тестов совпадут с выходными данными, полученными на исходной программе. Следовательно, такие тесты не требуют перезапуска.
* Множество тестов, требующих повторного запуска. К ним относятся тесты, которые уже запускались, но требуют перезапуска, поскольку затрагивают, по крайней мере, один измененный покрываемый элемент, подлежащий повторному тестированию. При повторном выполнении такие тесты могут давать результат, отличный от результата, показанного на исходной программе. Множество тестов, требующих повторного запуска, обеспечивает хорошее покрытие структурных элементов даже при наличии новых функциональных возможностей.
* Множество устаревших тестов. Это тесты, более не применимые к измененной программе и непригодные для дальнейшего тестирования, поскольку они затрагивают только покрываемые элементы, которые были удалены при изменении программы. Их можно удалить из набора регрессионных тестов.
* Новые тесты, которые еще не запускались и могут быть использованы для тестирования.

Сразу после создания тест вводится в структуру базы данных как новый. После исполнения новый тест переходит в категорию тестов, пригодных для повторного использования либо устаревших. Если выполнение теста способствовало увеличению текущей степени покрытия кода, тест помечается как пригодный для повторного использования. В противном случае он помечается как устаревший и отбрасывается. Существующие тесты, повторно запущенные после внесения изменения в код, также классифицируются заново как пригодные для повторного использования или устаревшие в зависимости от тестовых траекторий и используемого критерия тестирования.

Классификация тестов по отношению к изменениям в коде требует анализа последствий изменений. Тесты, активирующие код, затронутый изменениями, могут требовать повторного запуска или оказаться устаревшими. Чтобы тест был включен в класс тестов, требующих повторного запуска, он должен быть затронут изменениями в коде, а также должен способствовать увеличению степени покрытия измененного кода по используемому критерию. Затронутым элементом теста может быть траектория, выходные значения, или и то, и другое. Чтобы тест был включен в класс тестов, пригодных для повторного использования, он должен вносить вклад в увеличение степени покрытия кода и не требовать повторного запуска.

Степень покрытия кода определяется для тестов, пригодных для повторного использования, поскольку к этому классу относятся тесты, не требующие повторного запуска и способствующие увеличению степени покрытия до желаемой величины. Если имеется компонент программы, не задействованный пригодными для повторного использования тестами, то вместо них выбираются и выполняются с целью увеличения степени покрытия тесты, требующие повторного запуска. После запуска такой тест становится пригодным для повторного использования или устаревшим. Если тестов, требующих повторного запуска, больше не осталось, а необходимая степень покрытия кода еще не достигнута, порождаются дополнительные тесты и тестирование повторяется.

Окончательный набор тестов собирается из тестов, пригодных для повторного использования, тестов, требующих повторного запуска, и новых тестов. Наконец, устаревшие и избыточные тесты удаляются из набора тестов, поскольку избыточные тесты не проверяют новые функциональные возможности и не увеличивают покрытие.

Дополнительный материал.  
Регрессионное тестирование: разновидности метода отбора тестов  
<https://www.intuit.ru/studies/courses/48/48/lecture/1446?page=1>  
Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов  
<https://www.intuit.ru/studies/courses/48/48/lecture/1448>  
Регрессионное тестирование: алгоритм и программная система поддержки  
<https://www.intuit.ru/studies/courses/48/48/lecture/1450>

**Типы регрессии по Канеру?**

Сэм Канер описал 3 основных типа регрессионного тестирования:

* Регрессия багов (Bug regression) — попытка доказать, что исправленная ошибка на самом деле не исправлена
* Регрессия старых багов (Old bugs regression) — попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.
* Регрессия побочного эффекта (Side effect regression) — попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения

**Что подразумевается под проверкой дыма/дымовым тестированием? (Smoke testing)**

Дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для каждой новой сборки для подтверждения того, что ПО стартует и выполняет основные функции без критических и блокирующих дефектов. В случае отсутствия таковых дефектов дымовое тестирование объявляется пройденным, и команда QA может начинать дальнейшее тестирование полного цикла, в противном случае, сборка объявляется дефектной, что делает дальнейшее тестирование пустой тратой времени и ресурсов. Сборка возвращается на доработку и исправление.

Аналогами дымового тестирования являются Build Verification testing и Acceptance testing, выполняемые на функциональном уровне командой тестирования, по результатам которых делается вывод о том, принимается или нет установленная версия ПО в тестирование, эксплуатацию или на поставку заказчику.

Если мы говорим про сайт интернет-магазина, то сценарий будет примерно следующим:

* Сайт открывается
* Можно выбрать случайный товар и добавить его в корзину
* Можно оформить и оплатить заказ

Если мы говорим про мобильное приложение, например, messenger, то:

* Приложение устанавливается и запускается
* Можно авторизоваться
* Можно написать сообщение случайном контакту

Синонимом в некоторых источниках указан breath testing.

**Что такое тестирование встряхиванием? (Shake out testing)**

Стандартного определения ISO для теста на встряхивание для ПО не существует. Говорят, что это синоним к Smoke тестированию.

**Что подразумевается под санитарным тестированием/согласованности/исправности? (Sanity testing)**

Санитарное тестирование - это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

**Отличие санитарного тестирования от дымового? (Sanity vs Smoke testing)**

Эти виды тестирования имеют «вектора движения», направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

**В чем разница между повторным и регрессионным тестированием?**

* Регрессионное тестирование проводится для подтверждения того, что недавнее изменение программы или кода не оказало неблагоприятного воздействия на существующие функции. Повторное тестирование проводится для подтверждения того, что тест-кейсы, которые не прошли, проходят после устранения дефектов.
* Цель регрессионного тестирования подтвердить, что новые изменения кода не должны иметь побочных эффектов для существующих функций. Повторное тестирование проводится на основе исправлений дефектов.
* Проверка дефектов не является частью регрессионного тестирования. Проверка дефекта является частью повторного тестирования
* В зависимости от проекта и наличия ресурсов, регрессионное тестирование может проводиться параллельно с повторным тестированием. Приоритет повторного тестирования выше, чем регрессионное тестирование, поэтому оно проводится перед регрессионным тестированием.
* Вы можете сделать автоматизацию для регрессионного тестирования, ручное тестирование может быть дорогим и трудоемким.
* Регрессионное тестирование называется общим (generic) тестированием. Повторное тестирование — это плановое (planned) тестирование.
* Регрессионное тестирование проводится для пройденных Test case. Повторное тестирование проводится только для неудачных тестов.
* Регрессионное тестирование проверяет наличие неожиданных побочных эффектов. Повторное тестирование гарантирует, что первоначальная ошибка была исправлена.
* Регрессионное тестирование проводится только тогда, когда есть какие-либо изменения или изменения становятся обязательными в существующем проекте. Повторное тестирование выполняет дефект с теми же данными и той же средой с разными входными данными с новой сборкой (build).
* Test case для регрессионного тестирования могут быть получены из функциональной спецификации, user tutorials and manuals, а также defect reports в отношении исправленных проблем. Test case для повторного тестирования не могут быть получены до начала тестирования.

**Объясните, что такое тестирование N+1?**

Вариант регрессионного тестирования представлен как N+1. В этом методе тестирование выполняется в несколько циклов, в которых ошибки, обнаруженные в тестовом цикле «N», устраняются и повторно тестируются в тестовом цикле N + 1. Цикл повторяется, пока не будет найдено ни одной ошибки.

**Что означает подтверждающее тестирование? (confirmation/re-testing)**

Повторное тестирование — это тип тестирования, выполняемый для проверки того, что конкретный дефект устранён после исправления кода.

**Что вы знаете о тестировании сборки? (Build Verification Test)**

Тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

**Что такое тестирование файлов cookie?**

Тестирование файлов cookie определяется как тип тестирования программного обеспечения, который проверяет файлы cookie, созданные в вашем веб-браузере.

Файл cookie — это небольшой фрагмент информации, который веб-сервер хранит в текстовом файле на жестком диске пользователя (клиента). Эта часть информации затем отправляется обратно на сервер каждый раз, когда браузер запрашивает страницу с сервера. Обычно cookie-файлы содержат персонализированные пользовательские данные или информацию, которые используются для связи между различными веб-страницами. Другими словами, куки - это не что иное, как личность пользователя, и они используются для отслеживания того, где пользователь перемещался по страницам сайта. Цель файла cookie - обеспечить быстрое взаимодействие между пользователями и веб-сайтами. Приложения, в которых можно использовать файлы cookie, - это создание корзины покупок, персонализированный веб-интерфейс, отслеживание пользователей, маркетинг, пользовательские сессии и т. д.

Куки состоят в основном из трех вещей:

* Имя сервера, с которого был отправлен куки
* Время жизни (Cookies Lifetime)
* Случайно сгенерированный уникальный номер

Обычно на компьютеры пользователей пишутся два типа файлов cookie:

* Сеансовые куки: Эти куки активны до тех пор, пока открыт браузер. Когда мы закрываем браузер, этот файл cookie сеанса удаляется.
* Постоянные файлы cookie: эти файлы cookie постоянно хранятся на компьютере пользователя и сохраняются в течение нескольких месяцев или лет.

Где хранятся файлы cookie? Когда какое-либо приложение веб-страницы записывает cookie-файл, оно сохраняется в текстовом файле на жестком диске пользователя. Путь сохранения файлов cookie зависит от браузера.

Примеры Test case для Cookie testing:

* Отключение файлов cookie: отключите все файлы cookie и попытайтесь использовать основные функции сайта
* Поврежденные файлы cookie: вручную отредактируйте файл cookie в блокноте и измените параметры на несколько случайных значений
* Шифрование куки: конфиденциальная информация, такая как пароли и имена пользователей, должна быть зашифрована
* Тестирование файлов cookie в нескольких браузерах. Убедитесь, что с вашего веб-сайта правильно записываются cookie в разных браузерах
* Проверка удаления куки с веб-сайта
* Удаление файлов cookie: удалите все файлы cookie для веб-сайтов и посмотрите, как веб-сайт среагирует
* Доступ к файлам cookie: файлы cookie, написанные одним сайтом, не должны быть доступны другим
* Не допускайте чрезмерного использования файлов cookie: если тестируемое приложение является общедоступным веб-сайтом, не следует злоупотреблять файлами cookie.
* Тестирование с другими настройками. Тестирование должно выполняться правильно, чтобы убедиться, что веб-сайт работает хорошо с другими настройками файлов cookie.
* Категоризируйте куки отдельно: куки не должны храниться в той же категории вирусов, спама или шпионских программ

**Что такое тестирование потоков? (Thread testing)**

Тестирование потоков определяется как тип тестирования программного обеспечения, который проверяет основные функциональные возможности конкретной задачи (потока). Обычно проводится на ранней стадии фазы интеграционного тестирования. Тестирование на основе потоков является одной из дополнительных стратегий, принятых в ходе тестирования системной интеграции. Поэтому его, вероятно, следует более правильно назвать «тестом взаимодействия потоков» (thread interaction test).

Тестирование на основе потоков подразделяется на две категории:

* Однопоточное тестирование включает одну транзакцию приложения за раз
* Многопоточное тестирование включает одновременно несколько активных транзакций

Как производить:

* Тестирование на основе потоков является обобщенной формой тестирования на основе сеансов (session-based testing), в котором сеансы являются формой потока, но поток не обязательно является сеансом.
* Для тестирования потока, поток или программа (небольшая функциональность) интегрируются и тестируются постепенно как подсистема, а затем выполняются для всей системы.
* На самом низком уровне оно предоставляет интеграторам лучшее представление о том, что тестировать.
* Вместо непосредственного тестирования программных компонентов требуется, чтобы интеграторы сосредоточились на тестировании логических путей выполнения в контексте всей системы.

**Что такое тестирование документации? (Documentation testing)**

Плохая документация может повлиять на качество продукта. Хорошая документация по продукту играет решающую роль в конечном продукте. Тестирование артефактов, разработанных до, во время и после тестирования продукта, называется тестированием документации. Это нефункциональный тип тестирования программного обеспечения. Мы знаем, что дефекты, обнаруженные на этапе тестирования, более дорогостоящие, чем если бы они были обнаружены на этапе требований. Стоимость исправления ошибки увеличивается тем больше, чем позже вы найдете ее. Таким образом, тестирование документации может начаться с самого начала процесса разработки программного обеспечения, чтобы сэкономить большую сумму денег. Некоторые часто проверяемые артефакты:

* Requirement documents
* Test Plan
* Test case
* Traceability Matrix (RTM)

**Какие вы знаете уровни тестирования данных?**

Тесты группируются в зависимости от того, где они добавлены в SDLC, или от уровня детализации, который они содержат. В целом, существует четыре уровня или слоя тестирования: модульное тестирование, интеграционное тестирование, системное тестирование и приемочное тестирование. Целью уровней тестирования является систематизация тестирования программного обеспечения и простота выявления всех возможных Test case на определенном уровне.

Здесь работает хорошая аналогия с пирамидой тестирования. Это распределение по количеству тестов на разных уровнях приложения.

* Unit-слой — это когда тестируется один модуль программы, чаще всего это одна функция или метод. Таких тестов должно быть больше всего. Unit-тест для данных — это когда мы определяем требования для каждой ячейки. Нет смысла тестировать дальше, если у нас есть ошибки на уровне ячеек. Если, например, в фамилии содержатся цифры, то какой смысл проверять что-то дальше? Возможно, там должны быть буквы, похожие на эти цифры. И тогда нам нужно всё исправлять и проверять следующий уровень, чтобы у нас всё было в единственном числе и не было дубликатов, если так сказано в требованиях.
* Integration-слой — это когда несколько кусков программы тестируются вместе. Слой API для данных — это когда мы говорим о всей таблице. Допустим, у нас могут быть дубликаты, но не больше ста штук. Если у нас город-миллионник, то на одной улице не может жить миллион человек. Поэтому если мы сделаем выборку по улице, то количество адресов должно быть десять тысяч или тысяча — это надо определить. А если у нас миллион, то с данными что-то не так.
* System-слой — это когда вся программа тестируется полностью. В случае с данными этот слой означает, что тестируется вся система. Здесь включается статистика. Например, мы говорим, что у нас не может быть больше 30% мужчин, рожденных после 1985 года. Или мы говорим, что 80% данных должны быть одного типа.
* Приемочное тестирование — это тест, проводимый для определения того, удовлетворяются ли требования спецификации или контракта в соответствии с его поставкой. Приемочное тестирование в основном выполняется пользователем или заказчиком. Тем не менее, другие акционеры могут быть вовлечены в этот процесс.

**Что такое подкожный тест? (Subcutaneous test)**

Тест, который выполняется не для конечного пользовательского интерфейса, а для интерфейса, расположенного чуть ниже поверхности (пример — API).

**Расскажите о локализации, глобализации и интернационализации? (Localization/ globalization/internationalization testing)**

Глобализированное ПО — это ПО, функционирующее одинаково независимо от географической, культурной и национальной среды. Тестирование глобализации концентрируется на выявлении потенциальных проблем в дизайне продукта, которые могут испортить глобализацию. Например, разработчик должен заложить в CSS основу для вертикального текста, если в будущем планируется локализовать продукт на язык с вертикальным письмом или обработку почтовых индексов для разных стран (где-то цифры, где-то цифры с буквами и т.п.). Оно гарантирует, что код может обрабатывать желаемую международную поддержку без нарушения какой-либо функциональности. А также, что не будет никакой потери данных и проблем с отображением.

Тестирование глобализации, в свою очередь, можно разделить на два подвида:

* Тестирование интернационализации (118N)
* Тестирование локализации (L10N)

**Интернализация** ПО – это особый процесс, при котором веб-софт создается таким образом, чтобы оно было равноудаленным от какой-либо культуры и (или) специфики определенного географического региона.

Например, одна из задач по интернационализации ПО– корректное редактирование логики всех подключенных параметров форматирования (формат даты, времени, цифровое и валютное форматирование).

Также, тестеры во время проверки на соответствие ПО требованиям 118N тестируют работу продукта на равномерность работы в разных регионах и культурах мира.

Основной задачей тестирования интернациональности является проверка того, может ли программный код работать со всей международной поддержкой без нарушения функциональности, что может привести к потере данных или проблемам целостности информации.

В основном, фокус тестирования интернациональности направлен на:

* Тестирование языковой совместимости. Проводятся проверки того, может ли веб-продукт корректно работать в определенной языковой среде;
* Проверка функциональности: полное выполнение регрессионных тестов в разнообразных языковых средах (корректное отображение специфической информации – даты, времени и валюты);
* Тестирование на корректность отображения графического интерфейса;
* Проверка удобства пользования: тестирование простоты использования ПО на различных операционных системах, разнообразных устройствах и прочее.

Локализация ПО – деятельность по модификации ПО в соответствии с определенными региональными настройками (языком, географической территорией, кодировкой и прочим), которая должна бесперебойно работать.

В данный вид проверки входит необходимость выполнения работ по переводу всего контента программного обеспечения для конечного пользователя. Во время перевода должны учитываться иконки, информационная графика, справочные материалы, техническая документация и иные культурные особенности регионов, где в будущем будет доступен это программное обеспечение.

На что обратить внимание:

* Длина переведенных слов
* Параметры шрифта пользовательского интерфейса
* Ввод текста в разных локализациях
* RTL-языки (справа-налево) или вертикальных
* Перевод сокращений или аббревиатур
* Мета-теги (проблемы с SEO или отображением имени вкладки (title, description, keywords))
* Соответствие мер исчисления, валюты, postal code и т.п.

**Что такое исследовательское тестирование? (Exploratory testing)**

Исследовательское Тестирование — одновременно является и техникой, и видом тестирования. Такое тестирование подразумевает под собой одновременно изучение проекта, функционала, проектирование тест кейсов в уме и тут же их исполнение, не записывая и не создавая тестовую документацию.

Такой вид тестирования обычно не предусматривается в тест плане и тест кейсы выполняются и модифицируются динамически. Эфективность такого тестирования напрямую зависит от опыта тестировщика ранее имевшим дело с этим приложением, платформой, знанием мест скопления возможных багов и рисками кототые относятся к конкретному продукту. Цель данного тестирования — это углубление в познании продукта, приложения и нахождения «на лету» возможных багов. Также такое тестирование помогает в дальнейшем проектировании тест кейсов для покрытия функционала данного приложения. Исследовательское тестирование широко используется в Agile-моделях.

**Что вы знаете о турах Виттакера в исследовательском тестировании?**

Чтобы систематизировать исследовательское тестирование можно использовать идею туров. Туры – это идеи и инструкции по исследованию программного продукта, объединенные определённой общей темой или целью. Туры, как правило, ограничены по времени – длительность тестовой сессии не должна превышать 4 часа. Идею туров развивали в своих работах Канер, Бах, Хендриксон, Болтон, Кохл и другие. Джеймс Виттакер (James A. Whittaker), хоть и не придумал саму идею туров, но предложил свой подход к исследовательскому тестированию с использованием туров и в своей книге “Exploratory Software Testing” в доступной форме озвучил идею туров и описал сами туры.

Тур – это своего рода план тестирования, он отражает основные цели и задачи, на которых будет сконцентрировано внимание тестировщика во время сессии исследовательского тестирования. При этом Виттакер использует метафору, что тестировщик – это турист, а тестируемое приложение – это город. Обычно у туриста (тестировщика) мало времени, поэтому он выполняет конкретную задачу в рамках выбранного тура, ни на что другое не отвлекаясь. Город (ПО) разбит на районы: деловой центр, исторический район, район развлечений, туристический район, район отелей, неблагополучный район.  
Переводы и подробности "туров" есть тут:  
<https://habr.com/ru/post/328990/>  
<https://www.software-testing.ru/library/testing/testing-for-beginners/2965-exploratory-software-testing>  
<https://www.software-testing.by/blog/exploratory-testing-exploratory-tours/>

**Что такое Свободное или Интуитивное тестирование? (Adhoc)**

Часто его путают с другим видом тестирования «Exploratory testing» – «Исследовательское тестирование».

Свободное тестирование (ad-hoc testing) – это вид тестирования, который выполняется без подготовки к тестированию продукта, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Оно не требует никакой документации, планирования, процессов, которых следует придерживаться при выполнении тестирования. Такой способ тестирования в большинстве случаев дает большее количество заведенных отчётов об ошибке. Это обусловлено тем, что тестировщик на первых шагах приступает к тестированию основной функциональной части продукта и выполняет как позитивные, так и негативные варианты возможных сценариев.

Чаще всего такое тестирование выполняется, когда владелец продукта не обладает конкретными целями, проектной документацией и ранее поставленными задачами. При этом тестировщик полагается на свое общее представление о продукте, сравнение с похожими продуктами, собственный опыт. Однако при тестировании ad-hoc имеет смысл владеть общей информацией о продукте, особенно если проект очень сложный и большой. Поэтому нужно хорошее представление о целях проекта, его назначении и основных функциях, и возможностях. А дальше уже можно приступать к ad-hoc тестированию.

Виды свободного тестирования (ad-hoc testing):

* Buddy testing – процесс, когда 2 человека, как правило разработчик и тестировщик, работают параллельно и находят дефекты в одном и том же модуле тестируемого продукта. Такой вид тестирования помогает тестировщику выполнять необходимые проверки, а разработчику исправлять множество дефектов на ранних этапах.
* Pair testing – процесс, когда 2 тестировщика проверяют один модуль и помогают друг другу. К примеру, один может искать дефекты, а второй их документировать. Таким образом, у одного тестера будет функция, скажем так, обнаружителя, у другого – описателя.
* Monkey testing – произвольное тестирование продукта с целью как можно быстрее, используя различные вариации входных данных, нарушить работу программы или вызвать ее остановку (простыми словами – сломать).

Различия между Buddy testing и Pair testing:

* Buddy testing (Совместное тестирование) – это сочетание модульного тестирования и системного тестирования между разработчиком и тестировщиком.
* Pair testing (Парное тестирование) – выполняется только тестировщиками с разным уровнем знаний и опыта (такое сочетание поможет поделиться взглядами и идеями).

Основные преимущества ad-hoc testing:

* нет необходимости тратить время на подготовку документации;
* самые важные дефекты зачастую обнаруживаются на ранних этапах;
* часто применяется, когда берут нового сотрудника. С помощью этого метода, человек усваивает за 3 дня то, что, разбираясь тестовыми случаями, разбирал бы неделю – это называется форсированное обучение новых сотрудников;
* возможность найти трудновоспроизводимые и трудноуловимые дефекты, которые невозможно было бы найти, используя стандартные сценарии проверок.

Если нам нужно провести ad-hoc тестирование интернет-магазина, то этот краткий список может помочь с тем, что нужно проверить:

* все возможности сайта доступны без регистрации;
* корректность отображения анимаций и картинок;
* все возможности сайта доступны после регистрации;
* процесс регистрации;
* процесс добавления/удаления из корзины;
* процесс оплаты покупок;
* удобство в пользовании для новичков, простота, подсказки, помощь.

**Что вы знаете о мутационном тестировании? (Mutation testing)**

Mutation testing — это тип тестирования программного обеспечения, в котором мы мутируем (меняем) определенные выражения в исходном коде и проверяем, способны ли Test case найти ошибки. Это тип тестирования белого ящика, который в основном используется для модульного тестирования. Изменения в мутантной программе сохраняются крайне небольшими, поэтому это не влияет на общую цель программы. Цель Mutation testing - оценить качество Test case, которые должны быть достаточно надежными, чтобы не выполнять мутантный код. Этот метод также называется стратегией тестирования на основе ошибок, так как он включает в себя создание ошибки в программе.

* Шаг 1: Ошибки вводятся в исходный код программы путем создания множества версий, называемых мутантами. Каждый мутант должен содержать одну ошибку, и цель состоит в том, чтобы заставить версию мутанта потерпеть неудачу, что демонстрирует эффективность Test case.
* Шаг 2: Test case применяются к исходной программе, а также к программе мутанта.
* Шаг 3: Сравните результаты оригинальной и мутантной программы.
* Шаг 4: Если исходная программа и программы-мутанты генерируют разные выходные данные, то этот мутант уничтожается by the Test case. Следовательно, Test case достаточно хорош, чтобы обнаружить изменение между оригинальной и мутантной программой.
* Шаг 5: Если исходная программа и программа-мутант генерируют одинаковые выходные данные, мутант остается в живых. В таких случаях необходимо создать более эффективные Test case, которые убивают всех мутантов.

Что изменить в программе мутантов? Есть несколько методов, которые могут быть использованы для создания мутантных программ:

* Операторы замены операндов (Operand replacement operators) – например, в условии if (x> y) поменять местами значения x и y
* Операторы модификации выражений (Expression Modification Operators) – например, в условии if (х == у) Мы можем заменить == на >=
* Операторы модификации операторов (Statement modification Operators) – например, удалить часть else в конструкции if-else или удалить целиком конструкцию if-else, чтобы проверить, как ведет себя программа

Оценка мутации = (убитые мутанты / общее количество мутантов) \* 100

**Что означает механизм тестирования по ключевым словам? (Keyword Driven testing Framework)**

Это скриптовая техника, которая использует файлы данных, которые содержат ключевые слова, связанные с тестируемым ПО. Эти ключевые слова описывают набор действий, необходимых для выполнения определенного шага. Тест на основе ключевых слов состоит из ключевых слов высокого и низкого уровня, включая аргументы ключевых слов, которые составлены для описания действия Test case. Это также называется тестированием на основе таблиц (table-driven testing) или тестированием на основе action word (action word based testing). В тестировании по ключевым словам вы сначала идентифицируете набор ключевых слов, а затем связываете действие (или функцию), связанную с этими ключевыми словами. Здесь каждое действие тестирования, такое как открытие или закрытие браузера, щелчок мыши, нажатия клавиш и т. д., описывается ключевым словом, таким как openbrowser, click, Typtext и т. д. Тестирование на основе ключевых слов обычно выполняется с помощью автоматического тестирования.

**Что вы знаете о тестировании интерфейса прикладного программирования (API — Application Programming Interface)?**

API действует как интерфейс между двумя программными приложениями и позволяет им связываться друг с другом на оговорённых правилах и не залезая в реализацию предоставляемых функций. В реальных условиях API может содержать десятки и сотни методов, при этом не нужно задумываться, как это работает. У вас есть документ с перечнем того, что и как можно спросить и что вам на это придет в ответ. Часто взаимодействие с API скрыто под интерфейсом пользователя и вы, взаимодействуя с данным ресурсом, даже не знаете, что используете его API. Взаимодействовать с API можно и с веб-страницы, и с помощью специальных инструментов и напрямую из кода приложения.

Простой пример: вы можете встроить на свою главную страницу сайта маленький виджет прогноза погоды, который будет отправлять определенный правилами запрос к API некоего сервиса погоды и получать определенный правилами ответ, содержащий посылку с данными.

Еще более простой пример: примите официанта в качестве API ресторана. В ресторане вы даете заказ на основе блюд, определенных в меню. Официант принимает ваш заказ и на этом ваше участие заканчивается и вам не интересно, что там прозойдёт дальше. От официанта вы ожидаете только итог – вам приносят заказанное блюдо.

Тестирование API — это тип тестирования белого ящика, который включает в себя тестирование API напрямую, а также в рамках интеграционного тестирования, чтобы проверить, соответствует ли API ожиданиям с точки зрения функциональности, надежности, производительности и безопасности приложения. В тестировании API наш основной упор будет сделан на уровне бизнес-логики архитектуры программного обеспечения.

Типы тестирования API:

* Unit testing: Для проверки функциональности отдельной операции
* Functional testing: Чтобы проверить функциональность более широких сценариев с помощью блока результатов unit-тестирования, протестированных вместе
* Load testing: Чтобы проверить функциональность и производительность под нагрузкой
* Runtime/Error Detection: Мониторинг приложения для выявления проблем, таких как исключения и утечки ресурсов
* Security testing: Чтобы гарантировать, что реализация API защищена от внешних угроз
* UI testing: Это выполняется как часть end-to-end integration тестов, чтобы убедиться, что каждый аспект пользовательского интерфейса функционирует должным образом.
* Interoperability and WS Compliance testing: Совместимость и WS Compliance testing — это тип тестирования, который применяется к SOAP API. Функциональная совместимость между API-интерфейсами SOAP проверяется путем обеспечения соответствия профилям функциональной совместимости веб-служб. Соответствие WS- \* проверено, чтобы гарантировать, что стандарты, такие как WS-Addressing, WS-Discovery, WS-Federation, WS-Policy, WS-Security и WS-Trust, должным образом реализованы и используются
* Penetration testing: Чтобы найти уязвимости при атаках злоумышленников
* Fuzz testing: Для проверки API путем принудительного ввода в систему некорректных данных для попытки принудительного сбоя

Примеры для тестирования API:

* Возвращаемое значение на основе входных условий: его относительно легко проверить, поскольку входные данные могут быть определены и результаты могут быть проверены.
* Ничего не возвращает: при отсутствии возвращаемого значения проверяется поведение API в системе.
* Вызов другого API / события / прерывания: если выход API инициирует какое-либо событие или прерывание, то эти события и прерывания listeners следует отслеживать
* Обновление структуры данных: Обновление структуры данных будет иметь определенный эффект или влияние на систему, и это должно быть проверено
* Изменение определенных ресурсов: если вызов API изменяет некоторые ресурсы, его следует проверить путем доступа к соответствующим ресурсам.

Чаще всего для тестирования API сейчас используют инструмент Postman.

Postman предназначен для проверки запросов с клиента на сервер и получения ответа от бэкенда.

Позволяет писать тесты внутри себя на JS.

Функции:

* отправка запросов
* тесты автоматических запросов
* окружения
* тест-коллекции
* тест-раннер

**Как протестировать API без документации/черным ящиком?**

Если Вам по какой-то причине предстоит проделать эту неблагодарную работу, определитесь, насколько всё плохо и какая у Вас есть информация об объекте тестирования.

Известно ли какие порты для Вас открыты? Знаете ли Вы нужные endpoints?

Если дело совсем плохо — просканируйте порты, например, с помощью netcat. Открытые порты сохраните в файл.

Эта операция займёт довольно много времени. Можете почитать советы по работе с Nmap и Netcat.

Если Вам известен нужный порт и соответствующий endopoint — переберите все возможные HTTP методы. Начните с наиболее очевидных POST, PUT, GET. Для ускорения процесса напишите скрипт, например, на Python.

В худшем случае, когда ни порт ни endpoints неизвестны Вам, скорее всего придётся перебирать все открытые порты и генерировать endpoints, которые подходят по смыслу.

Разработчики обычно не особо заморачиваются и закладывают минимально-необходимую информацию. Так что включите воображение и попробуйте придумать endpoints опираясь на бизнес логику и принятые в Вашей компании стандарты.

Если ни endpoints ни бизнес логика Вам неизвестны, то у меня есть подозрение, что Вы тестируете API с не самыми хорошими намерениями.

**А что такое endpoint?**

Смысл в том, что сайт, написанный на любом языке, поддерживающем HTTP запросы, не посылает на сервер никаких PHP/C/Python команд, а общается ним с помощью запросов, описанных в API.

Адрес, на который посылаются сообщения называется Endpoint. Обычно это URL (например, название сайта) и порт. Если я хочу создать веб сервис на порту 8080, Endpoint будет выглядеть так: [http://vladislaveremeev.ru:8080](http://vladislaveremeev.ru:8080/)

Если моему Web сервису нужно будет отвечать на различные сообщения я создам сразу несколько URL (interfaces) по которым к сервису можно будет обратиться. Например:

* [https://vladislaveremeev.ru:8080](https://vladislaveremeev.ru:8080/) /resource1/status
* [https://vladislaveremeev.ru:8080](https://vladislaveremeev.ru:8080/) /resource1/getserviceInfo
* [https://vladislaveremeev.ru:8080](https://vladislaveremeev.ru:8080/) /resource1/putID
* [http://vladislaveremeev.ru:8080](http://vladislaveremeev.ru:8080/) /resource1/eventslist
* [https://vladislaveremeev.ru:8080](https://vladislaveremeev.ru:8080/) /resource2/putID

Как видите у моих эндпойнтов (Enpoints) различные окончания. Такое окончание в Endpoint называются Resource, а начало Base URL. Такое определение Endpoint и Resource используется, например, в SOAP UI для RESTful интерфейсов [https://vladislaveremeev.ru:8080](https://vladislaveremeev.ru:8080/) - это Base URL /resource1/status - это Resource

Endpoint = Base URL + Resource

Понятие Endpoint может использоваться в более широком смысле. Можно сказать, что какой-то определённый роутер или компьютер является Endpoint. Обычно это понятно из контекста. Также следует обратить внимание на то, что понятие Endpoint выходит за рамки RESTful и может использовать как в SOAP так и в других протоколах. Термин Resource также связан с RESTful, но в более широком смысле может означать что-то другое.

Итак, простейший запрос состоит из метода и Enpoint

Request = Method + Endpoint

**Frontend testing Vs. Backend testing?**

**Frontend testing** — это тип тестирования, который проверяет уровень представления 3-уровневой архитектуры. С точки зрения непрофессионала, вы проверяете GUI - все, что видно на экране, на стороне клиента. Для веб-приложения интерфейсное тестирование будет включать проверку функциональных возможностей, таких как формы, графики, меню, отчеты и т. д., а также связанный Javascript. Frontend testing - это термин, охватывающий различные стратегии тестирования, включая оценку производительности фронтенда, которая является хорошей практикой перед тестированием приложения с высокими пользовательскими нагрузками. Тестер должен хорошо понимать бизнес-требования для выполнения этого типа тестирования. Ранее оптимизация производительности означала оптимизацию на стороне сервера. Это было связано с тем, что большинство веб-сайтов были в основном статичными, а большая часть обработки выполнялась на стороне сервера. Однако сегодня веб-приложения становятся более динамичными и в результате код на стороне клиента нередко становится причиной низкой производительности.

**Backend testing** — это тип тестирования, который проверяет уровень приложений и базы данных 3-уровневой архитектуры. В сложном программном приложении, таком как ERP, внутреннее тестирование повлечет за собой проверку бизнес-логики на уровне приложений. Для более простых приложений бэкэнд-тестирование проверяет серверную часть или базу данных. Это означает, что данные, введенные в интерфейс, будут проверены в базе данных. Базы данных проверяются на наличие свойств ACID, операций CRUD, их схемы, соответствия бизнес-правилам. Базы данных также проверяются на безопасность и производительность. Производится проверка целостности данных, Проверка достоверности данных, Тестирование функций, процедур и триггеров. При внутреннем тестировании нет необходимости использовать графический интерфейс. Вы можете напрямую передавать данные с помощью браузера с параметрами, необходимыми для функции, чтобы получить ответ в некотором формате по умолчанию. Например, XML или JSON. Вы также подключаетесь к базе данных напрямую и проверяете данные с помощью SQL-запросов.

**Что подразумевают под эталонным тестированием? (Baseline testing)**

Это подход к тестированию, в котором за точку отсчета берется базовая линия — это показатель конкретного ориентира, который служит основой для нового тестирования. В базовом тестировании тесты собирают и сохраняют все результаты, полученные в исходном коде, и сравнивают с эталонным базовым уровнем. Этот базовый уровень относится к последним принятым результатам испытаний. Если в исходном коде есть новые изменения, то для повторного выполнения тестов необходимо сформировать текущий базовый уровень. Если последние результаты будут приняты, то текущая базовая линия станет эталонной. По большей части Baseline testing относят к тестированию производительности.

**В чем разница между Baseline и Benchmark testing?**

* Baseline предназначено для оценки производительности приложения. Benchmark сравнивает производительность приложения с отраслевым стандартом.
* Baseline тестирование использует данные, собранные для повышения производительности. Benchmark возвращает информацию о целевом приложении по сравнению с другими приложениями.
* Baseline тестирование сравнивает текущую производительность с предыдущей производительностью приложения, тогда как Benchmark сравнивает производительность нашего приложения с производительностью конкурентов.

**Что такое параллельное/многопользовательское тестирование? (Concurrency/Multi-user testing)**

Параллельное тестирование определяется как метод тестирования для обнаружения дефектов в приложении, когда в систему вошли несколько пользователей. Другими словами, отслеживание эффекта, когда несколько пользователей выполняют одно и то же действие одновременно. Параллельное тестирование также называется многопользовательским тестированием. Тестирование параллельной программы является более сложной задачей, чем тестирование последовательной программы, из-за недетерминированности и проблем синхронизации. Зачем оно нужно:

* Определяет влияние одновременного доступа к одним и тем же записям базы данных, модулям или коду приложения.
* Определяет и измеряет уровень взаимоблокировки, блокировки и использования однопоточного кода и ограничения доступа к общим ресурсам

**Как вы думаете, что такое тестирование на переносимость?**

Тестирование переносимости — это тип тестирования программного обеспечения, который проводится для определения степени легкости или сложности, с которой программное приложение может быть эффективно и эффективно перенесено с одного аппаратного обеспечения, программного обеспечения или среды на другое.

Результаты тестирования переносимости представляют собой измерения того, насколько легко программный компонент или приложение будут интегрированы в среду, и затем эти результаты будут сравниваться с нефункциональным требованием переносимости программной системы. Измерение основано на сравнении стоимости адаптации программного обеспечения к новой среде и стоимости реконструкции.

Атрибуты тестирования переносимости:

* Адаптивность: Адаптируемость определяется как способность программного приложения адаптироваться к конкретной среде без каких-либо усилий. Общие стандарты связи между несколькими системами помогают повысить адаптивность системы в целом.
* Installability: Устанавливаемость определяется как способность программного приложения быть установленным в желаемой среде без использования дополнительных ресурсов. Устанавливаемость выполняется на программном обеспечении, которое должно быть установлено в целевой среде.
* Заменяемость: Возможность замены определяется как способность программного приложения заменять другое программное обеспечение в конкретной среде. Приложение, которое заменяет предыдущее приложение, должно давать одинаковые результаты во всех целевых средах.
* Сосуществование: Сосуществование определяется как способность программного приложения работать с другим программным приложением в системе, не мешая друг другу и совместно используя один и тот же ресурс. Специально это тестирование используется в больших системах, которые включают в себя несколько подсистем.

**Что такое тестирование графического интерфейса/визуальное тестирование? (GUI — Graphical User Interface)**

Существует два типа интерфейсов для компьютерного приложения. Интерфейс командной строки, где вы вводите текст, и компьютер отвечает на эту команду и GUI — графический интерфейс пользователя, где вы взаимодействуете с компьютером, используя изображения, а не текст.

Цель тестирования графического интерфейса пользователя (GUI) — проверить функциональность интерфейса пользователя.

Примеры:

* Тестирование размера, положения, ширины, высоты элементов.
* Тестирование сообщений об ошибках, которые отображаются.
* Тестирование разных разделов экрана.
* Проверка шрифта, читаемый ли он или нет.
* Тестирование экрана в разных разрешениях с помощью увеличения и уменьшения масштаба, например, 640 x 480, 600x800 и т. д.
* Проверка выравнивания текстов и других элементов, таких как значки, кнопки и т. д. , находятся на своем месте или нет.
* Тестирование цветов шрифтов.
* Проверка цветов сообщений об ошибках, предупреждающих сообщений.
* Проверка, имеет ли изображение хорошую четкость или нет.
* Тестирование выравнивания изображений.
* Проверка орфографии.
* Пользователь не должен разочаровываться при использовании системного интерфейса.
* Тестирование, является ли интерфейс привлекательным или нет.
* Тестирование полос прокрутки в соответствии с размером страницы, если таковые имеются.
* Тестирование отключенных полей, если таковые имеются.
* Тестирование размера изображений.
* Проверка заголовков, правильно ли они выровнены или нет.
* Тестирование цвета гиперссылки.

Визуальное тестирование проверяет корректность отображения пользователю web-сайта, мобильного или десктопного приложения, дизайн-системы, PDF-файла или отдельного изображения на наличие расхождений с спецификацией дизайна (рендеринг страниц, шрифтов, изображений и т. д.). Раньше выполнялось вручную, т.к., например, в случае веб-сайта классическое автоматизированное тестирование было бесполезно – большинство инструментов сверялись с DOM и не видели те ошибки, что человек мог увидеть вживую. Сейчас визуальное тестирование выполняется автоматизированными инструментами создания скриншотов и сверки их с эталоном. Там всё еще очень много нюансов, но это уже не относится к статье о ручном тестировании.

**Что такое A/B тестирование?**

A / B-тестирование также называется сплит-тестированием (split). При тестировании AB мы создаем и анализируем два варианта приложения, чтобы найти, какой вариант работает лучше с точки зрения пользовательского опыта, потенциальных клиентов, конверсий или любой другой цели, а затем в конечном итоге сохранить наиболее эффективный вариант.

Давайте попробуем понять это на примере. Предположим, у нас есть интернет магазин и каталог отображается определенным образом. В какой-то момент (новые маркетинговые исследования/пожелания клиента и т. д.) решено изменить дизайн выдачи товаров в каталоге. Независимо от того, сколько проведено анализа, выпуск нового пользовательского интерфейса будет большим изменением и может иметь неприятные последствия.

В этом случае мы можем использовать A / B-тестирование. Мы создадим интерфейс нового варианта и выпустим его для некоторого процента пользователей. Например — мы можем распределить пользователей в соотношении 50:50 или 80:20 между двумя вариантами - A и B. После этого в течение определенного периода времени мы будем наблюдать эффективность обоих вариантов. Таким образом, тестирование A/B помогает принять решение о выборе лучшего варианта.

**Что означает сквозное тестирование? (E2E — End–to–End)**

Сквозное тестирование — это стратегия тестирования для выполнения тестов, которые охватывают все возможные потоки приложения от его начала до конца; проверяет программную систему вместе с ее интеграцией с внешними интерфейсами. Целью сквозного тестирования является создание полного производственного сценария, выявление программных зависимостей и утверждение, что между различными программными модулями и подсистемами передается правильный ввод. Сквозное тестирование обычно выполняется после функционального и системного тестирования. Оно использует реальные данные, такие как данные и тестовая среда, для имитации настроек в реальном времени. Сквозное тестирование также называется цепным тестированием (Chain testing).

Для чего оно нужно? Современные программные системы являются сложными и взаимосвязаны с несколькими подсистемами. Подсистема может отличаться от текущей системы или может принадлежать другой организации. Если какая-либо из подсистем выйдет из строя, вся система программного обеспечения может рухнуть. Это серьезный риск, и его можно избежать путем сквозного тестирования.

**В чем разница между E2E и системным тестированием?**

|  |  |
| --- | --- |
| End to End testing | System testing |
| Проверяет программную систему, а также взаимосвязанные подсистемы | Проверяет только программную систему в соответствии со спецификациями требований. |
| Проверяет весь E2E flow | Проверяет функциональные возможности и функции системы. |
| Все интерфейсы, бэкэнд-системы | Функциональное и нефункциональное тестирование |
| Выполняется после завершения System testing | Выполняется после завершения Integration testing |
| Сквозное тестирование включает проверку внешних интерфейсов, которые могут быть сложными для автоматизации. Следовательно, ручное тестирование является предпочтительным. | Как ручное, так и автоматическое могут быть выполнены для тестирования системы |

**Что такое параллельное тестирование? (Parallel testing)**

Это тип тестирования ПО, который одновременно проверяет несколько приложений или подкомпонентов одного приложения, чтобы сократить время выполнения теста. При параллельном тестировании тестер запускает две разные версии программного обеспечения одновременно с одним и тем же вводом. Цель состоит в том, чтобы выяснить, ведут ли себя прежняя система и новая система одинаково или по-разному. Это гарантирует, что новая система достаточно способна для эффективной работы программного обеспечения.

Пример: когда какая-либо организация переходит от старой системы к новой, legacy является важной частью. Передача этих данных является сложным процессом. При тестировании программного обеспечения проверка совместимости вновь разработанной системы со старой системой осуществляется посредством «параллельного тестирования».

|  |  |
| --- | --- |
| Это Parallel testing | Это НЕ Parallel testing |
| * Тестирование обновленного приложения по сравнению с предыдущим приложением. * Запуск старого сценария с новым программным обеспечением с зарезервированными условиями ввода. * Цель состоит в том, чтобы узнать, соответствует ли результат предыдущей системе. * Должен иметь знания о старой и недавно разработанной системе |  Тестирование только одного ПО   Кросс-браузерное или кроссплатформенное тестирование.   Цель состоит в том, чтобы выяснить проблему проектирования.   Знать разницу не обязательно. |

**Тест дизайн**

**Тест дизайн? (Test Design)**

Этап процесса тестирования ПО, на котором проектируются и создаются Test case (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования. План работы над тест дизайном:

* анализ имеющихся проектных артефактов: документация (спецификации, требования, планы), модели, исполняемый код и т. д.
* написание спецификации по тест дизайну ([Test Design Specification](http://www.protesting.ru/documentation/sqe_test_design_specification_template.zip))
* проектирование и создание [Test case](http://www.protesting.ru/testing/testcase.html)

Роли, ответственные за тест дизайн:

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

Попросту говоря, задача тест аналитиков и дизайнеров сводится к тому, чтобы используя различные стратегии и техники тест дизайна, создать набор Test case, обеспечивающий оптимальное тестовое покрытие тестируемого приложения. Однако, на большинстве проектов эти роли не выделяется, а доверяется обычным тестировщикам, что не всегда положительно сказывается на качестве тестов, тестировании и, как из этого следует, на качестве ПО (конечного продукта).

**Перечислите известные техники тест-дизайна?**

* Cтатические (Static):
  + Review:
    - Неформальное ревью (Informal review)
    - Прохождение (Walkthrough)
    - Техническое ревью (Technical Review)
    - Инспекция (Inspection)
  + Статический анализ (Static Analysis):
    - Поток данных (Data Flow)
    - Поток управления (Control Flow)
    - Путь (Path)
    - Стандарты (Standards)
* Динамические (Dynamic):
  + Белый ящик (White-box)
    - Выражение (Statement)
    - Решение (Decision)
    - Ветвь (Branch)
    - Условие (Condition)
    - Конечный автомат (FSM)
  + Основанные на опыте (Expirience-based):
    - Предугадывание ошибки (Error Guessing — EG)
    - Исследовательское тестирование (Exploratory testing)
    - Ad-hoc testing
  + Черный ящик (Black-box):
    - Эквивалентное Разделение (Equivalence Partitioning — EP)
    - Анализ Граничных Значений (Boundary Value Analysis — BVA)
    - Комбинаторные техники ([Combinatorial Test Techniques](https://sysgears.com/articles/test-design-techniques-overview/" \l "combinatorial))
    - Переходы между состояниями (State transition)
    - Случаи использования (Use case testing)

Альтернативный источник:

* [Specification-Based Testing Techniques (or Black Box techniques):](https://sysgears.com/articles/test-design-techniques-overview/#black-box)
  + [Equivalence Partitioning](https://sysgears.com/articles/test-design-techniques-overview/#equivalence-classes)
  + [Boundary Value Analysis](https://sysgears.com/articles/test-design-techniques-overview/#boundary-values)
  + [Combinatorial Test Techniques:](https://sysgears.com/articles/test-design-techniques-overview/#combinatorial)
    - [All Combinations](https://sysgears.com/articles/test-design-techniques-overview/#all-combinations)
    - [Pairwise Testing](https://sysgears.com/articles/test-design-techniques-overview/#pairwise)
    - [Each Choice Testing](https://sysgears.com/articles/test-design-techniques-overview/#each-choice)
    - [Base Choice Testing](https://sysgears.com/articles/test-design-techniques-overview/#base-choice)
  + [Decision Table Testing](https://sysgears.com/articles/test-design-techniques-overview/#decision-table)
  + [Classification Tree Method](https://sysgears.com/articles/test-design-techniques-overview/#classification-tree)
  + [State Transition Testing](https://sysgears.com/articles/test-design-techniques-overview/#state-transition)
  + [Cause-Effect Graphing](https://sysgears.com/articles/test-design-techniques-overview/#cause-effect)
  + [Scenario Testing](https://sysgears.com/articles/test-design-techniques-overview/#scenario)
  + [Random Testing](https://sysgears.com/articles/test-design-techniques-overview/#random)
  + [Syntax Testing](https://sysgears.com/articles/test-design-techniques-overview/#syntax)
* [Structure-Based Testing Techniques (or White Box techniques):](https://sysgears.com/articles/test-design-techniques-overview/#white-box)
  + [Statement Testing](https://sysgears.com/articles/test-design-techniques-overview/#statement)
  + [Decision Testing](https://sysgears.com/articles/test-design-techniques-overview/#decision)
  + [Condition Testing:](https://sysgears.com/articles/test-design-techniques-overview/#conditions)
    - [Branch Condition Testing](https://sysgears.com/articles/test-design-techniques-overview/#branch-condition)
    - [Branch Condition Combination Testing](https://sysgears.com/articles/test-design-techniques-overview/#branch-condition-combination)
    - [Modified Condition Decision Coverage (MCDC) Testing](https://sysgears.com/articles/test-design-techniques-overview/#mcdc)
  + [Data Flow Testing](https://sysgears.com/articles/test-design-techniques-overview/#data-flow)
* [Experience-Based Testing Techniques:](https://sysgears.com/articles/test-design-techniques-overview/#experience)
  + [Error Guessing](https://sysgears.com/articles/test-design-techniques-overview/#error-guessing)

Все методы тестирования на основе спецификаций (черного ящика) могут быть удобно описаны и систематизированы с помощью следующей таблицы:

|  |  |  |
| --- | --- | --- |
| **Группа** | **Техника** | **Когда используется** |
| Элементарные методы: - сфокусированы на анализе входных / выходных параметров — могут быть объединены для обеспечения лучшего покрытия - обычно не используют и не зависят от других приемов | Equivalence Partitioning | Входные и выходные параметры имеют большое количество возможных значений |
| Boundary Value Analysis | Значения параметров имеют явные (например, четко определенные в документации) границы и диапазоны или неявные (например, известные технические ограничения) границы |  |
| Комбинаторные стратегии: - объединить возможные значения нескольких входных / выходных параметров — можно использовать элементарные приемы, чтобы уменьшить количество возможных значений | All Combinations | Количество возможных комбинаций входов достаточно мало, или каждая отдельная комбинация входов приводит к определенному выходу |
| Pairwise Testing | Количество входных комбинаций чрезвычайно велико и должно быть сведено к приемлемому набору cases |  |
| Each Choice Testing | У вас есть функциональность, при которой конкретное значение параметра чаще вызывает ошибку чем комбинация значений |  |
| Base Choice Testing | Вы можете выделить набор значений параметров, которые имеют наибольшую вероятность использования |  |
| Продвинутые техники: — помочь проанализировать Систему с точки зрения бизнес-логики, иерархических отношений, сценариев и т. д. - анализ основан на данных, организованных в виде таблиц, диаграмм и шаблонов - может полагаться на элементарные и комбинаторные методы для разработки Test case | Decision Table Testing | Существует набор комбинаций параметров и их выводов, описываемых бизнес-правилами или другими правилами. |
| Classification Tree Method | У вас есть иерархически структурированные данные, или данные могут быть представлены в виде иерархического дерева |  |
| State Transition Testing | В функциональности есть очевидные состояния, у которых переходы регулируются правилами (например, потоками) |  |
| Cause-Effect Graphing | Причины (входы) и следствия (выходы) связаны большим количеством сложных логических зависимостей |  |
| Scenario Testing | Есть четкие сценарии в функционале |  |
| Другие | Random Testing | Вы должны подражать непредсказуемости реальных входных данных, или функциональность имеет несистематические дефекты |
| Syntax Testing | Функциональность имеет сложный синтаксический формат для ввода (например, коды, сложные имена электронной почты и т. д.) |  |

Методы тестирования на основе структуры (Structure-Based Testing Techniques): также известны как методика тестирования White Box, это означает, что мы знакомы с кодом, который собираемся тестировать. Чтобы понять эти методы, мы должны определить, что такое покрытие в контексте разработки теста. Вот хорошее определение из Книги ISTQB: Тестирование покрытия определенным образом измеряет количество тестов, выполненных набором тестов (полученных другим способом, например, с использованием методов, основанных на спецификациях). Везде, где мы можем посчитать вещи и сказать, была ли каждая из этих вещей проверена каким-либо тестом, мы можем измерить охват. Основная мера покрытия:

Number of coverage items exercised

Coverage = -------------------------------------- x 100%

Total number of coverage items

где «coverage item» — это то, что мы смогли подсчитать и посмотреть, выполнил ли тест этот элемент или использовал его.

Для методов, основанных на спецификациях, это могут быть случаи использования, разделы эквивалентности, граничные значения, состояния из диаграммы перехода состояний, процент бизнес-правил из таблицы решений и т. д. Для методов, основанных на структуре, элементы покрытия представлены структурные элементы кода. В принципе, оценка покрытия означает, что мы должны решить, какие структурные элементы мы будем использовать (например, заявления или решения). Затем найдите общее количество этих элементов в коде. Введите дополнительные операторы (например, ведение журнала) рядом с каждым структурным элементом, чтобы выяснить, использовался ли этот элемент во время выполнения Test case. И, наконец, измерьте охват, выполнив тесты и используя формулу, упомянутую выше. Но, как правило, вы не должны думать обо всех этих шагах, потому что есть много автоматизированных инструментов для измерения покрытия. Методы структурного тестирования обычно подразумевают, что вы должны измерить покрытие для существующих наборов тестов (включая наборы черного ящика), а затем разработать дополнительные Test case белого ящика, основанные на элементах структурного кода, для достижения максимально возможного покрытия.

**Что такое статическое тестирование, когда оно начинается и что оно охватывает?**

При статическом тестировании код не выполняется. Вы вручную проверяете код, документы требований и проектные документы на наличие ошибок. Отсюда и название «статичный». Основная цель этого тестирования — повысить качество программных продуктов путем выявления ошибок на ранних этапах цикла разработки. Это тестирование также называется Non-execution техникой или verification. Проверки могут также производиться автоматически (например, используя линтеры).

В статическом тестировании проверяются следующее:

* Unit Test cases
* Business Requirements Document (BRD)
* Use Cases
* System/Functional Requirements
* Prototype
* Prototype Specification Document
* DB Fields Dictionary Spreadsheet
* Test Data
* Traceability Matrix Document
* User Manual/Training Guides/Documentation
* Test Plan Strategy Document/Test cases
* Automation/Performance Test Scripts

**Что такое динамическое тестирование, когда оно начинается и что оно охватывает?**

При динамическом тестировании выполняется код. Оно проверяет функциональное поведение ПО, использование памяти / процессора и общую производительность системы. Основная цель этого тестирования — подтвердить, что программный продукт работает в соответствии с требованиями бизнеса. Это тестирование также называется Execution technique или validation. Динамическое тестирование выполняется на всех уровнях тестирования, и это может быть либо тестирование черного, либо белого ящика.

Виды динамического тестирования: Модульное тестирование. Интеграционное тестирование: Системное тестирование. Кроме того, нефункциональное тестирование, такое как производительность, тестирование безопасности.

На примере: Предположим, мы тестируем страницу входа в систему, где у нас есть два поля с именем «Имя пользователя» и «Пароль», а имя пользователя ограничено буквенно-цифровым форматом. Когда пользователь вводит имя пользователя «VA610», система принимает это. Но когда пользователь вводит «VA610 @ 123», тогда приложение выдает сообщение об ошибке. Этот результат показывает, что код действует динамически на основе пользовательского ввода.

**Какие виды Review вы знаете?**

* Неофициальные reviews: это один из видов рецензирования, который не сопровождается каким-либо процессом поиска ошибок в документе. При использовании этого метода вы просто просматриваете документ и оставляете неофициальные комментарии к нему.
* Технические reviews: команда, состоящая из ваших коллег, изучает технические характеристики программного продукта и проверяет, подходят ли он для проекта. Они пытаются найти любые несоответствия в спецификациях и стандартах. Этот обзор концентрируется главным образом на технической документации, связанной с программным обеспечением, такой как: Стратегия тестирования, План тестирования и спецификации требований.
* Пошаговое руководство. Автор рабочего продукта объясняет продукт своей команде. Участники могут задавать вопросы, если таковые имеются. Встреча проводится автором.
* Инспекция: Основная цель — найти дефекты, а встречу проводит обученный модератор. Этот обзор является формальным типом обзора, где следует строгий процесс поиска дефектов. У рецензентов есть контрольный список для проверки рабочих продуктов. Они фиксируют дефект и информируют участников об устранении этих ошибок.
* Code Walk Through: Это неформальный анализ исходного кода программы для выявления дефектов и проверки методов кодирования.

**Что вы знаете о Data Flow testing?**

Тестирование потока данных — это еще один набор методов / стратегий белого ящика, который связан с анализом потока управления, но с точки зрения жизненного цикла переменной. Переменные определяются, используются и уничтожаются, когда в них больше нет необходимости. Аномалии в этом процессе, такие как использование переменной без ее определения или после ее уничтожения, могут привести к ошибке. Существуют условные обозначения, которые могут помочь в описании последовательных во времени пар в жизненном цикле переменной:

* ~ — переменная еще не существует или предыдущий этап был последним
* d — определено, создано, инициализировано
* k — не определено, убито
* u — используется (c - использование вычислений; p - использование предикатов)

Таким образом, ~ d, du, kd, ud, uk, uu, k ~, u ~ являются вполне допустимыми комбинациями, когда ~ u, ~ k, dd, dk, kk, ku, d ~ являются аномалиями, потенциальными или явными ошибками. В настоящее время практически все они эффективно обнаруживаются компиляторами или, по крайней мере, IDE, и нам редко требуется выполнять статический анализ для обнаружения этих аномалий. То же самое относится и к динамическому анализу, который сфокусирован на исследовании / выполнении du пар — современные языки программирования снижают вероятность возникновения проблем, связанных с du. Так что в настоящее время такая проверка в основном не стоит усилий.

**Что вы знаете о Control Flow testing?**

Тестирование потоков управления (Control Flow Testing) — это одна из двух техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

Фундаментом для тестирования потоков управления является построение графов потоков управления (Control Flow Graph), основными блоками которых являются:

* блок процесса — одна точка входа и одна точка выхода
* точка альтернативы — одна точка входа, две и более точки выхода
* точка соединения — две и более точек входа, одна точка выхода

При тестировании потока управления определяются различные уровни покрытия тестами. Под «покрытием» мы подразумеваем процент кода, который был протестирован, по сравнению с тем, который есть для тестирования. В тестировании потока управления мы определяем покрытие на нескольких различных уровнях. (Обратите внимание, что эти уровни охвата представлены не по порядку. Это связано с тем, что в некоторых случаях проще определить более высокий уровень охвата, а затем определить более низкий уровень охвата с точки зрения более высокого.):

|  |  |  |
| --- | --- | --- |
| Уровень | Название | Краткое описание |
| Уровень 0 | -- | «Тестируй все что протестируешь, пользователи протестируют остальное» На английском языке это звучит намного элегантнее: «Test whatever you test, users will test the rest» |
| Уровень 1 | Покрытие операторов | Каждый оператор должен быть выполнен как минимум один раз. |
| Уровень 2 | Покрытие альтернатив / Покрытие ветвей | Каждый узел с ветвлением (альтернатива) выполнен как минимум один раз. |
| Уровень 3 | Покрытие условий | Каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз. |
| Уровень 4 | Покрытие условий альтернатив | Тестовые случаи создаются для каждого условия и альтернативы |
| Уровень 5 | Покрытие множественных условий | Достигается покрытие альтернатив, условий и условий альтернатив (Уровни 2, 3 и 4) |
| Уровень 6 | «Покрытие бесконечного числа путей» | Если, в случае зацикливания, количество путей становится бесконечным, допускается существенное их сокращение, ограничивая количество циклов выполнения, для уменьшения числа тестовых случаев. |
| Уровень 7 | Покрытие путей | Все пути должны быть проверены |

Подробный разбор с примерами доступен в источнике:

flylib.com/books/en/2.156.1/control\_flow\_testing.html

**Что такое Loop coverage?**

Loop testing определяется как тип тестирования программного обеспечения методом белого ящика, который полностью фокусируется на валидности конструкций цикла. Это одна из частей тестирования структуры управления (Control Structure testing): тестирование пути, проверка данных, тестирование условий (path testing, data validation testing, condition testing).

Этот показатель сообщает, выполняли ли вы каждое тело цикла ноль раз, ровно один раз и более одного раза (последовательно). Для циклов do-while покрытие цикла сообщает, выполняли ли вы тело ровно один раз и более одного раза. Ценным аспектом этой метрики является определение того, выполняются ли циклы while и for более одного раза, т.к. эта информация не сообщается другими метриками.

**Что такое Race coverage?**

Этот показатель показывает, выполняет ли несколько потоков один и тот же код одновременно. Это помогает обнаружить сбой при синхронизации доступа к ресурсам. Это полезно для тестирования многопоточных программ, например, в операционной системе.

**Тестирование пути и тестирование базового пути? (Path testing & Basis Path testing)**

Path testing — это метод структурного тестирования, который включает использование исходного кода программы для нахождения каждого возможного исполняемого пути. Это помогает определить все ошибки, лежащие в части кода. Здесь Test case выполняются таким образом, что каждый путь проходится по крайней мере один раз. Все возможные control paths, включая все loop paths. Test case подготавливаются на основе логической сложности. При таком типе тестирования каждый оператор в программе гарантированно выполняется как минимум один раз. Flow Graph, Cyclomatic Complexity и Graph Metrics используются для достижения basis path.

Любая программа включает в себя несколько точек входа и выхода. Тестирование каждого из этих пунктов является сложным и трудоемким. Для сокращения избыточных тестов и достижения максимального охвата тестов используется Basis Path testing. Basis Path testing такое же, но оно основано на методе White Box testing и определяет Test case на основе потоков или логического пути, которые могут быть пройдены через программу. В программной инженерии Basis Path testing включает в себя выполнение всех возможных блоков в программе и достижение максимального охвата пути с наименьшим количеством Test case. Это гибрид branch testing и path testing. Цель Basis Path testing состоит в том, что оно определяет количество независимых путей, таким образом, число необходимых Test case может быть определено явно (максимизирует охват каждого тестового случая).

Тесты критического пути (Critical path tests) запускаются для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности. Многие пользователи обычно используют определенный набор функций приложения, который необходимо проверить, как только фаза smoke будет успешно завершена. Здесь метрический предел немного ниже, чем при smoke, и он соответствует 70-80-90% в зависимости от цели проекта при ста процентах у smoke. Чаще всего на практике, на данном уровне тестирования проверяется основная масса требований к продукту. Пример: выбор шрифта, возможность набора текста, вставки картинок и т. д.

**Что вы знаете о Statement coverage?**

Охват операторов — это метод проектирования теста белого ящика, который включает в себя выполнение всех исполняемых операторов (if, for и switch) в исходном коде как минимум один раз. Он используется для вычисления и измерения количества операторов в исходном коде, которые могут быть выполнены с учетом требований. Другими словами, тестер будет концентрироваться на внутренней работе исходного кода, касающегося control flow graphs или flow charts. Как правило, в любом программном обеспечении, если мы посмотрим на исходный код, будет множество элементов, таких как операторы, функции, циклы, обработчики исключений и т. д. В зависимости от входных данных программы некоторые части кода могут не выполняться. Цель покрытия Statement - охватить все возможные пути, строки и операторы в коде. Тестируются операторы потока управления, такие как.

Покрытие операторов позволяет найти:

* Неиспользованные выражения (Unused Statements)
* Мертвый код (Dead Code)
* Неиспользуемые ветви (Unused Branches)
* Недостающие операторы (Missing Statements)

**Что вы знаете о Decision coverage?**

«Решение» — это программная точка, в которой control flow имеет два или более альтернативных маршрута. control flow- это последовательность событий (paths) при выполнении через компонент или систему. Таким образом, если быть точным, «решение» - это узел потока управления (например, оператор if, оператор цикла или оператор case), который имеет две или более ссылок на отдельные ветви выполнения. 100% покрытие решений означает, что все возможные результаты решения были выполнены по крайней мере один раз. Для утверждений if это правда или ложь.

Иногда этот метод называют Branch testing.

**Что вы знаете о Branch coverage?**

В покрытии ветвей проверяется каждый результат из модуля кода. Например, если результаты являются бинарными, вам необходимо протестировать как истинные, так и ложные результаты. Это помогает вам гарантировать, что каждая возможная ветвь из каждого условия решения выполняется по крайней мере один раз. Используя метод покрытия Branch, вы также можете измерить долю независимых сегментов кода. Это также поможет вам узнать, какие разделы кода не имеют ветвей.

Если тесты имеют полный branch coverage, то мы можем сказать, что они также имеют полный statement coverage, но не наоборот. Причина заключается в том, что в branch coverage, кроме выполнения всех statements, мы также должны проверить, выполняют ли тесты все ветви, что можно интерпретировать как охват всех ребер в control flow branch.

**Что вы знаете о Condition coverage?**

Покрытие условий (Condition/Toggle Coverage) — рассматриваются только выражения с логическими операндами, например, AND, OR, XOR. Условное покрытие обеспечивает лучшую чувствительность к control flow, чем decision coverage. Condition Coverage не дает гарантии о полном decision coverage.

Multiple Condition Coverage: Множественное покрытие условий сообщает, встречается ли каждая возможная комбинация условий. Test Case, необходимые для полного охвата решения несколькими условиями, приведены в таблице истинности логического оператора для решения. Как и в случае покрытия условий, покрытие нескольких условий не включает покрытие решений.

**Что вы знаете о FSM coverage?**

Покрытие FSM (FSM Coverage) — Покрытие конечного автомата, безусловно, является наиболее сложным методом покрытия кода. В этом методе покрытия вам нужно посмотреть, как много было переходов/посещений определенных по времени состояний (time-specific states). Оно также проверяет, сколько последовательностей включено в конечный автомат.

**Что такое Function coverage?**

Этот показатель показывает, вызывали ли вы каждую функцию или процедуру. Во время предварительного тестирования полезно обеспечить хотя бы некоторое покрытие во всех областях программного обеспечения. Широкое неглубокое тестирование быстро обнаруживает серьезные недостатки в наборе тестов.

**Что такое Call coverage?**

Этот показатель показывает, выполняли ли вы каждый вызов функции. Гипотеза состоит в том, что ошибки обычно возникают в интерфейсах между модулями. Также известен как покрытие пары вызовов (call pair coverage).

**Что означает LCSAJ coverage?**

LCSAJ (linear code sequence and jump) «линейная последовательность кода и переход». Эта вариация path coverage учитывает только подпути, которые могут быть легко представлены в исходном коде программы, не требуя flow graph. LCSAJ — это последовательность строк исходного кода, выполняемых последовательно. Эта «линейная» последовательность может содержать decisions, пока control flow фактически продолжается от одной строки к следующей во время выполнения. Подпути создаются путем объединения LCSAJ. Исследователи ссылаются на коэффициент покрытия путей длиной n LCSAJ как на коэффициент эффективности теста (TER) n + 2.

Его основное применение при динамическом анализе программного обеспечения, чтобы помочь ответить на вопрос «Сколько тестирования достаточно?». Динамический анализ программного обеспечения используются для измерения качества и эффективности тестовых данных программного обеспечения, где количественное определение выполняются в терминах структурных единиц кода при тестировании. В более узком смысле, LCSAJ является хорошо определенным линейным участком коды программы. При использовании в этом смысле, LCSAJ также называют JJ-путь, стоя для скачка к скачку-пути. 100% LCSAJ означает 100% Statement Coverage, 100% Branch Coverage, 100% procedure или Function call Coverage, 100% Multiple condition Coverage.

**Сравнение некоторых метрик**

Вы можете сравнить «относительные силы»? (relative strengths), когда более сильная метрика включает более слабую метрику.

* Decision coverage включает в statement coverage, поскольку выполнение каждой ветви должно приводить к выполнению каждого statement. Эта связь сохраняется только тогда, когда control flows непрерывно до конца всех основных блоков. Например, функция C / C ++ может никогда не вернуться для завершения вызывающего базового блока, потому что она использует throw, abort, семейство exec, exit или longjmp.
* Path coverage включает в себя Decision coverage.
* Predicate coverage включает в себя Path coverage и multiple condition coverage, а также большинство других метрик.

Показатели покрытия нельзя сравнивать количественно.

**Что такое Equivalence Partitioning?**

Хорошо известная техника и одна из самых используемых. Часто вы можете найти такое объяснение: Например, у вас есть диапазон допустимых значений от 1 до 10, поэтому вам просто нужно проверить одно значение из диапазона — «5» и одно вне диапазона - «0». Это очень простое и понятное объяснение, но оно может дать узкий взгляд на эту технику. Что если у нас нет диапазона чисел? Концепция разделения эквивалентности возникла из математики и понимания того, что такое класс эквивалентности и отношение эквивалентности, что может быть трудно быстро понять без математического бекграунда. Но для целей тестирования, я думаю, это можно упростить, определив его следующим образом: эквивалентное разделение - это подмножество элементов из определенного набора, которые обрабатываются Системой (тестируются) одинаково. Таким образом, вам не нужно выполнять тесты для каждого элемента подмножества, и достаточно одной проверки, чтобы охватить все подмножество. Следовательно, методика может быть описана как разделение всего набора данных ввода / вывода на такие разделы. И если у вас есть, например, набор данных, содержащий около 100 элементов, которые можно разделить на 5 разделов, вы можете уменьшить количество кейсов до 5. Хитрость заключается в том, чтобы увидеть и идентифицировать разделы. Их можно найти в наборах без номеров (например, листья деревьев, разделенные по цвету - желтый, зеленый и т. д.), или даже один элемент может быть классом эквивалентности (например, лифт обычно более полон на первом этаже, чем на других этажах). По мере того, как люди выходят из здания, поднимается лифт, поэтому первый этаж - это отдельный класс эквивалентности). Очень эффективная, экономящая время техника.

**Что такое Boundary Value Analysis?**

Анализ Граничных Значений (Boundary Value Analysis — BVA). Второй известный метод, который часто используется в паре с предыдущим. Идея этого метода заключается в проверке граничных значений для разделов эквивалентности, когда результат изменяется с действительного на недействительный (для этого раздела). Test case, основанные на этих значениях, чувствительны к ошибкам и имеют высокую вероятность обнаружения ошибок при использовании логических операторов (> вместо >=, < вместо >, || вместо && и т. l.).

Это описание немного неоднозначно. Для проверки границ мы должны проверять допустимые значения границ диапазона, плюс одно значение ниже и одно значение вне диапазона. Таким образом, для диапазона больше 1 и меньше или равного 10, это 1, 2 и 10, 11.

**Что такое Error Guessing?**

Предугадывание ошибки (Error Guessing — EG). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать» при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «пользователь должен ввести код». Тест аналитик, будет думать: «Что, если я не введу код?», «Что, если я введу неправильный код? «, и так далее. Это и есть предугадывание ошибки.

**Что такое Cause/Effect?**

Причина / Следствие (Cause/Effect — CE). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие), то есть Простая проверка базовых действий и их результата. Например, если нажать крестик в правом верхнем углу окна (причина), оно закроется (следствие). Или вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как «Имя», «Адрес», «Номер Телефона» а затем, нажать кнопку «Добавить» - это «Причина». После нажатия кнопки «Добавить», система добавляет клиента в базу данных и показывает его номер на экране - это «Следствие».

Граф причинно-следственных связей похож на Decision Table и также использует идею объединения условий. И иногда они описываются как один метод. Но если между условиями существует много логических зависимостей, может быть проще их визуализировать на cause-effect graph. Здесь можно выделить три этапа:

* Определить условия и последствия
* Нарисовать график со всеми логическими зависимостями и ограничениями
* Преобраовать график в Decision Table, отслеживая каждую комбинацию причин, которые приводят к эффекту от эффекта (tracing each combination of causes that lead to an effect from the effect).

**Что такое Exhaustive testing?**

Исчерпывающее тестирование (Exhaustive testing — ET) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

**Какие вы знаете комбинаторные техники тест-дизайна?**

Или, скорее, стоит рассматривать их как комбинаторные стратегии. Их основное назначение — создавать комбинации входных параметров на основе одного из приведенных ниже алгоритмов.

Все комбинации (All combinations): как видно из названия, этот алгоритм подразумевает генерацию всех возможных комбинаций. Это означает исчерпывающее тестирование и имеет смысл только при разумном количестве комбинаций. Например, 3 переменные с 3 значениями для каждой дают нам матрицу параметров 3х3 с 27 возможными комбинациями.

Попарное тестирование (Pairwise testing) — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров. Смысл метода не в том, чтобы перебрать все возможные пары параметров, а в том, чтобы подобрать пары, обеспечивающие максимально эффективную проверку при минимальном количестве выполняемых тестов. С этой задачей помогают справиться математические методы, называемые ортогональными таблицами (OAT). Также существует ряд инструментов, которые помогают автоматизировать этот процесс.

Тестирование каждого выбора (Each choise testing): эта стратегия означает, что каждое значение каждого конкретного параметра должно использоваться как минимум один раз в тестовом наборе. Таким образом, полученное количество случаев будет равно количеству значений параметра с наибольшим диапазоном. Каждый выбор — это минимальная стратегия покрытия.

|  |  |  |
| --- | --- | --- |
| param1 | param2 | param3 |
| a | 1 | X |
| b | 2 | Y |
| c | 3 |  |
|  | 4 |  |
|  | 5 |  |
| Test Case | param1 | param2 | param3 |
| 1 | a | 1 | X |
| 2 | b | 2 | Y |
| 3 | c | 3 | X |
| 4 | a | 4 | Y |
| 5 | b | 5 | X |

Тестирование базового выбора (Base choise testing): для этой стратегии мы должны определить наши базовые значения для каждого параметра. Это могут быть самые распространенные, самые маленькие / самые большие, самые простые или значения по умолчанию. После того, как мы сделали наш базовый выбор, мы должны изменять значение каждого параметра по одному, сохраняя при этом значения других параметров фиксированными при базовом выборе. Пусть a, 2 и Y будут нашим базовым выбором. Тогда кейсы будут:

|  |  |  |
| --- | --- | --- |
| param1 | param2 | param3 |
| a | 1 | X |
| b | 2 | Y |
| c | 3 | Z |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | param1 | param2 | param3 |
| 1 | a | 2 | Y |
| 2 | b | 2 | Y |
| 3 | c | 2 | Y |
| 4 | a | 1 | Y |
| 5 | a | 3 | Y |
| 6 | a | 2 | X |
| 7 | a | 2 | Z |

**Что такое тестирование ортогональных массивов? (OAT — Orthogonal Array testing)**

Оно является техникой тестирования, которая использует ортогональные массивы для создания Test case. Это особенно полезно, когда тестируемая система имеет огромные входные данные. Например, когда необходимо проверить билет на поезд, необходимо проверить такие факторы, как — количество пассажиров, номер билета, номера мест и номера поездов. Один за другим проверка каждого фактора / ввода является громоздкой. Это более эффективно, когда инженер QA объединяет больше входных данных и проводит тестирование. В таких случаях мы можем использовать метод тестирования Orthogonal Array. Этот тип сопряжения или объединения входов и тестирования системы для экономии времени называется попарным тестированием (pairwise testing). Метод OATS используется для попарного тестирования. Сформулировать суть pairwise можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

В реальном мире у тестировщиков не будет свободного времени для выполнения всех возможных Test case, чтобы выявить дефекты, поскольку существуют другие процессы, такие как документация, предложения и обратная связь с заказчиком, которые необходимо учитывать на этапе тестирования. Следовательно, test managers хотели оптимизировать количество и качество Test case, чтобы обеспечить максимальное покрытие тестами с минимальными усилиями. Это усилие называется Test case Optimisation.

* Систематический и статистический способ тестирования парных взаимодействий
* Точки взаимодействия и интеграции являются основным источником дефектов.
* Выполните четко определенные, краткие Test case, которые могут выявить большинство ошибок.
* Ортогональный подход гарантирует попарное покрытие всех переменных.

Формула для расчета ОАТ:

Runs (N) — количество строк в массиве, что выражается в количестве тестовых случаев, которые будут сгенерированы. Factors (K) - Количество столбцов в массиве, что выражается в максимальном количестве переменных, которые могут быть обработаны. Levels (V) - максимальное количество значений, которые могут быть приняты для любого отдельного фактора. Один фактор имеет от 2 до 3 входов для тестирования. Это максимальное количество входов определяет уровни.

**Что такое Domain analysis/testing?**

Доменный анализ: Это тип функционального тестирования, направленный разбиение диапазона возможных значений переменной (или переменных) на поддиапазоны (или домены), с последующим выбором одного или нескольких значений из каждого домена для тестирования, то есть на анализ показательных значений и взаимосвязи элементов. Основная цель Domain testing: предоставить стратегию по выбору минимального набора показательных тестов. Кроме того, оно проверяет, что система не должна принимать входные данные, условия и индексы за пределами указанного или действительного диапазона. Во многом доменное тестирование пересекается с известными нам техниками разбиения на классы эквивалентности и анализа граничных значений. Но доменное тестирование не ограничивается перечисленными техниками. Оно включает в себя как анализ зависимостей между переменными, так и поиск тех значений переменных, которые несут в себе большой риск (не только на границах).

**Что такое Cyclomatic Complexity в тестировании ПО?**

Это метрика ПО, используемая для измерения сложности программы. Это количественная мера независимых путей в исходном коде программы. Независимый путь определяется как путь, имеющий хотя бы одно ребро, которое ранее не проходило ни в одном другом пути. Цикломатическая сложность может быть рассчитана относительно функций, модулей, методов или классов в программе. Эта метрика основана на представлении программы как control flow. Поток управления изображает программу в виде графа, который состоит из вершин и ребер. На графе вершины представляют обработку задачи, а ребра представляют поток управления между вершинами.

Тестирование базового пути является одним из методов «белого ящика» и гарантирует выполнение хотя бы одного оператора во время тестирования. Он проверяет каждый линейно независимый путь в программе, что означает, что число тестовых примеров будет эквивалентно цикломатической сложности программы.  V(G) = E — N + 2 где E – количество ребер, N –количество вершин

или

V(G) = P + 1 где P — Количество предикатных узлов (узел, содержащий условие)

Сложность 1-10 говорит о хорошо написанном коде, легкой тестируемости и экономичности. 10-20 и 20-30 говорят об усложнённом коде и трудностях с тестированием такого кода вкупе с высокими затратами. Сложность больше 40 практически не поддаётся проверке и стоит очень дорого. Подсчитывается вручную в случае небольшого ПО или с помощью автоматизированного ПО для крупного.

Цикломатическая Сложность может оказаться очень полезной:

* Помогает разработчикам и тестировщикам определять независимые пути выполнения
* Разработчики могут быть уверены, что все пути были протестированы хотя бы раз
* Помогает нам больше сосредоточиться на открытых (uncovered) путях
* Улучшение покрытия кода
* Оценка рисков
* Использование этих метрик в начале цикла снижает риски

**Что такое State Transition testing?**

Тестирование переходов между состояниями определяется как метод тестирования ПО, при котором изменения входных условий вызывают изменения состояния в тестируемом приложении (AUT). Это метод тестирования черного ящика, в котором тестировщик анализирует поведение тестируемого приложения для различных входных условий в последовательности. В этом методе тестер предоставляет как положительные, так и негативнве входные значения теста и записывает поведение системы. Это модель, на которой основаны система и тесты. Любая система, в которой вы получаете разные выходные данные для одного и того же ввода, в зависимости от того, что произошло раньше, является системой конечных состояний. Техника тестирования переходов между состояниями полезна, когда вам нужно протестировать различные системные переходы. Этот подход лучше всего подходит там, где есть возможность рассматривать всю систему как конечный автомат

* Состояние (state, представленное в виде круга на диаграмме) – это состояние приложения, в котором оно ожидает одно или более событий. Состояние помнит входные данные, полученные до этого, и показывает, как приложение будет реагировать на полученные события. События могут вызывать смену состояния и/или инициировать действия.
* Переход (transition, представлено в виде стрелки на диаграмме) – это преобразование одного состояния в другое, происходящее по событию.
* Событие (event, представленное ярлыком над стрелкой) – это что-то, что заставляет приложение поменять свое состояние. События могут поступать извне приложения, через интерфейс самого приложения. Само приложение также может генерировать события (например, событие «истек таймер»). Когда происходит событие, приложение может поменять (или не поменять) состояние и выполнить (или не выполнить) действие. События могут иметь параметры (например, событие «Оплата» может иметь параметры «Наличные деньги», «Чек», «Приходная карта» или «Кредитная карта»).
* Действие (action, представлено после «/» в ярлыке над переходом) инициируется сменой состояния («напечатать билет», «показать на экране» и др.). Обычно действия создают что-то, что является выходными/возвращаемыми данными системы. Действия возникают при переходах, сами по себе состояния пассивны.
* Точка входа обозначается черным кружком.
* Точка выхода показывается на диаграмме в виде мишени.

Для наглядности возьмем классический пример покупки авиабилетов. Все начинается с точки входа. Мы (клиенты) предоставляем авиакомпании информацию для бронирования. Служащий авиакомпании является интерфейсом между нами и системой бронирования авиабилетов. Он использует предоставленную нами информацию для создания бронирования. После этого наше бронирование находится в состоянии «Создано». После создания бронирования система также запускает таймер. Если время таймера истекает, а забронированный билет еще не оплачен, то система автоматически снимает бронь.

Каждое действие, выполненное над билетом, и соответствующее состояние (отмена бронирования пользователем, оплата билета, получение билета на руки, и т. д.) отображаются в блок-схеме. На основании полученной схемы составляется набор тестов, в котором хотя бы раз проверяются все переходы.

Некоторым исследователям представляется более удобным свести весь процесс в таблицу состояний и переходов. Конечно, таблица не так наглядна, как схема, но зато она получается более полной и систематизированной, так как определяет все возможные State-Transition варианты, а не только валидные.

Еще пример с банкоматом. После того, как мы визуализировали все переходы, мы просто выполняем определенные пути из диаграммы в качестве Test case:

**Что такое Scenario (use case) testing?**

Use Case описывает сценарий взаимодействия двух и более участников (как правило – пользователя и системы). Пользователем может выступать как человек, так и другая система. Юзкейсы могут быть позитивными (Sunny day use case) – в приоритете, и негативными (Rainy day use case).

Целью этого тестирования является нахождение дефектов, которые трудно найти при тестировании частей/модулей отдельно. Но нужно понимать, что это не аналог приемочного тестирования (но может быть его частью) и оно не охватывает все возможные варианты путей.

Как правило, Test case представлен очень небольшим количеством действий и одним результатом. Сценарий, с другой стороны, представляет собой последовательность действий (с промежуточными результатами), которые приводят к достижению какой-то конкретной цели. Сценарии могут быть частью документации разработчика (scenario diagrams). Довольно часто они задокументированы в требованиях как «use cases» — сценарии, которые обычно описывают взаимодействие пользователя с Системой. Но часто эти сценарии не очень подробны. Кроме того, прежде чем использовать их для создания Test case, их необходимо подробно описать с помощью шаблона. Шаблоны могут варьироваться от проекта к проекту. Но среди таких обычных полей, как имя, цель, предварительные условия, актер (ы) и т. д., всегда есть основной успешный сценарий и так называемые расширения (плюс иногда подвариации). Расширения - это условия, которые влияют на основной сценарий успеха. А подвариации - это условия, которые не влияют на основной flow, но все же должны быть рассмотрены. После того, как шаблон заполнен данными, мы создаем конкретные Test case, используя методы эквивалентного разделения и граничных значений. Для минимального охвата нам нужен как минимум один тестовый сценарий для основного сценария успеха и как минимум один Test case для каждого расширения. Опять же, этот метод соответствует общей формуле «получите условия, которые меняют наш результат, и проверьте комбинации». Но способ получить это - проанализировать поведение Системы с помощью сценариев.

Пример: Рассмотрим сценарий, когда пользователь покупает товар с сайта онлайн-покупок. Пользователь сначала войдет в систему и начнет поиск. Пользователь выберет один или несколько товаров, отображаемых в результатах поиска, и добавит их в корзину. После всего этого он проверит. Так что это пример логически связанного ряда шагов, которые пользователь выполнит в системе для выполнения задачи. В этом тестировании проверяется поток транзакций во всей системе от конца до конца. Варианты использования — это, как правило, путь, который пользователи чаще всего используют для достижения конкретной задачи. Таким образом, это упрощает использование прецедентов при обнаружении дефектов, поскольку включает в себя путь, с которым пользователи чаще всего сталкиваются при первом использовании приложения.

Как создать хорошие сценарии (Сэм Канер):

* 1. Напишите истории жизни для объектов в системе.
  2. Перечислите возможных пользователей, проанализируйте их интересы и цели.
  3. Подумайте об отрицательных пользователях: как они хотят злоупотреблять вашей системой?
  4. Перечислите «системные события». Как система справляется с ними?
  5. Перечислите «особые события». Какие приспособления система делает для них?
  6. Перечислите преимущества и создайте сквозные задачи, чтобы проверить их.
  7. Интервью пользователей об известных проблемах и сбоях старой системы.
  8. Работайте вместе с пользователями, чтобы увидеть, как они работают и что они делают.
  9. Читайте о том, что должны делать подобные системы.
  10. Изучите жалобы на предшественника этой системы или ее конкурентов.
  11. Создать фиктивный бизнес. Относитесь к нему как к реальному и обрабатывайте его данные.
  12. Попробуйте преобразовать реальные данные из конкурирующего или предшествующего приложения.

**Что такое Decision Table testing?**

Этот простой метод заключается в документировании бизнес-логики в таблице как наборов условий и действий. Например, если у вас есть набор переменных, которые трудно запомнить и управлять ими, таблица решений поможет упорядочить их, чтобы упростить идентификацию правильных случаев.

Пример: если пользователь вводит правильное имя пользователя и пароль, он будет перенаправлен на домашнюю страницу. Если какой-либо из вводимых данных неправильный, появится сообщение об ошибке.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Условие | 1 | 2 | 3 | 4 |
| Имя пользователя (+/-) | - | + | - | + |
| Пароль (+/-) | - | - | + | + |
| Итог (E/H) | E | E | E | H |

Условные обозначения: «+» — правильное имя пользователя / пароль «-» - Неверное имя пользователя / пароль E - Сообщение об ошибке отображается H - отображается главный экран

* + Случай 1 — имя пользователя и пароль были неверны. Пользователю показывается сообщение об ошибке.
  + Случай 2 — Имя пользователя было правильным, но пароль был неправильным. Пользователю показывается сообщение об ошибке.
  + Случай 3 — Имя пользователя было неверным, но пароль был правильным. Пользователю показывается сообщение об ошибке.
  + Случай 4 — имя пользователя и пароль были правильными, и пользователь перешел на домашнюю страницу Преобразуя это в тестовый пример, мы можем создать 2 сценария:
    1. Введите правильное имя пользователя и правильный пароль и нажмите на кнопку входа, и ожидаемый результат будет пользователь должен перейти на домашнюю страницу
    2. И один из сценария ниже:
       1. Введите неправильное имя пользователя и неправильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке
       2. Введите правильное имя пользователя и неправильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке
       3. Введите неправильное имя пользователя и правильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке

**Что такое Random testing?**

Техника функционального тестирования (Black box), также известный как тестирование с произвольным вводом, это, вероятно, самый недооцененный метод, основная идея которого заключается в выборе случайных входных данных из возможных значений для определенной функциональности. Так что нет никакой системы в выборе входных данных. Этот метод также называется monkey testing, и если мы говорим о ручном тестировании, он может быть менее эффективным, чем другие методы черного ящика. Но если мы добавим автоматизацию, она станет мощным инструментом. Просто представьте, что Test case (с различными наборами входных данных) генерируются, выполняются и оцениваются автоматически в непрерывном цикле, что позволяет вам запускать тысячи и миллионы случаев в течение разумного времени. Создание «Случайного тестера» — довольно интересная тема, но также довольно сложная и требует более глубокого изучения. Здесь я опишу это только концептуально.

В Monkey testing тестером (иногда и разработчиком) считается «Обезьяна». Если обезьяна использует компьютер, она будет произвольно выполнять любую задачу в системе из своего понимания. Точно так же, как тестер будет применять случайные Test case в тестируемой системе, чтобы находить bugs/errors без предварительного определения тестового примера. В некоторых случаях Monkey testing также посвящен модульному тестированию или GUI-тестированию. Основная задача: попытаться сломать систему.

Gorilla testing — это метод тестирования ПО, при котором модуль программы многократно тестируется, чтобы убедиться, что он работает правильно и в этом модуле нет ошибок. Модуль может быть проверен более ста раз одним и тем же способом. Gorilla testing также известен как «раздражающее тестирование».

|  |  |
| --- | --- |
| Monkey testing | Gorilla testing |
| Тестирование выполняется случайным образом без каких-либо заранее определенных Test case | Тоже |
| Тестирование выполняется на всей системе может иметь несколько Test case | Тестирование выполняется на нескольких отдельных модулях с небольшим количеством Test case. |
| Целью Monkey testing является проверка на сбой системы | Целью тестирования Gorilla является проверка правильности работы модуля. |

|  |  |
| --- | --- |
| Monkey testing | Ad-hoc testing |
| Тестирование выполняется случайным образом без каких-либо заранее определенных Test case | Тестирование выполняется без планирования и документации (контрольные примеры и SRS) |
| В Monkey testing тестеры могут не знать, что за ПО и для чего оно предназначено. | В Ad-hoc testing тестер должен понять ПО перед выполнением тестирования |
| Целью Monkey testing является проверка на сбой системы | Целью специального тестирования является случайное разделение системы на части и проверка их функциональности. |

Типы Обезьян:

* Тупая обезьяна: тестеры не имеют представления о системе и ее функциональных возможностях, а также не имеют никаких гарантий относительно достоверности Test case.
* Умная обезьяна: Тестер имеет четкое представление о системе, ее назначении и функциональности. Тестер перемещается по системе и предоставляет действительные данные для выполнения тестирования.
* Выдающаяся обезьяна: тестеры выполняют тестирование в соответствии с поведением пользователя и могут указать некоторые вероятности возникновения ошибок.

**Что такое Syntax testing?**

Синтаксическое тестирование — это метод проверки «черного ящика». Этот метод помогает при разработке Test case для входных форматов. Конечно, если наши правила синтаксиса звучат как «должны быть только цифры или буквы», нам не нужен этот метод. Но если у нас есть какой-то сложный формат (например, почтовый индекс, телефон, конкретный адрес электронной почты), это может быть удобно. Прежде всего, мы должны определить наш формат и описать его формально, используя форму Бэкуса-Наура (или расширенный BNF). Это очень важный момент, поэтому я бы посоветовал вам ознакомиться с BNF, прежде чем читать дальше. После того, как мы создали определение BNF для нашего формата, пришло время генерировать положительные и отрицательные кейсы:

Для положительных случаев мы находим возможные варианты значений, допускаемые отдельными элементами определения BNF, а затем разрабатываем варианты, чтобы просто охватить эти варианты.

Для случаев с недопустимым синтаксисом мы определяем и применяем возможные мутации (например, отсутствующий элемент, нежелательный дополнительный элемент, недопустимое значение для элемента и т. д.) к отдельным элементам определения BNF. Затем мы разрабатываем наши кейсы, применяя мутации, которые могут давать отличительные результаты (случаи, которые приводят к действительным комбинациям, исключаются).

**Что вы знаете о Classification tree method?**

Большинство ресурсов выделяют два основных шага для этой техники:

* Определение относящихся к тесту аспектов (аспектов, которые влияют на функциональность — так называемые классификации) и их соответствующих значений (называемых классами, это могут быть точные значения или классы эквивалентности).
* Объединение разных классов из всех классификаций в Test case.

Оба эти шага справедливы для большинства методов тестирования и могут быть перефразированы следующим образом: — определить параметры ввода / вывода, а затем объединить их, чтобы получить ваши кейсы.

Теперь давайте добавим некоторые детали. Классификационное дерево — это графический метод, который помогает нам визуализировать относящиеся к тесту аспекты (аспекты, которые влияют на поведение тестового объекта во время теста) в форме иерархического дерева.

Как вырастить это дерево? Это похоже на интеллектуальные карты с небольшими отличиями, если это вам о чем-то говорит. У нас есть тестовый объект (целое приложение, определенная функция, абстрактная идея и т. д.) вверху как корень. Мы рисуем ответвления от корня как классификации (проверяем соответствующие аспекты, которые мы определили). Затем, используя распределение по эквивалентности и анализ граничных значений, мы определяем наши листья как классы из диапазона всех возможных значений для конкретной классификации. И если некоторые из классов могут быть классифицированы далее, мы рисуем под-ветку / классификацию с собственными листьями / классами. Когда наше дерево завершено, мы делаем проекции листьев на горизонтальной линии (Test case), используя одну из комбинаторных стратегий (все комбинации, каждый выбор и т. д.), и создаем все необходимые комбинации.

В приведенном выше примере использовалась комбинаторная стратегия «Каждый выбор». Наиболее очевидные преимущества здесь — это большая наглядность и ясность объема тестового объекта и идей Test case. Если у вас есть сложные, иерархически структурированные данные, и вы можете позволить себе тратить время на создание и поддержку дерева, этот метод будет чрезвычайно удобен. И для эффективного применения метода вы можете, например, рассмотреть возможность использования специального инструмента, такого как Classification Tree Editor.

**Как мы узнаем, что код соответствует спецификациям?**

Матрица прослеживаемости — это интуитивно понятный инструмент, который обеспечивает соответствие требований тестовым примерам. Когда выполнение всех Test case заканчивается успешно, это указывает на то, что код соответствует требованиям.

**Что включает в себя матрица отслеживания требований? (RTM — Requirement Traceability Matrix)**

Матрица прослеживаемости — это документ, который связывает любые два базовых документа, которые требуют отношения «многие ко многим» для проверки полноты отношений. В случае тестирования это матрица покрытия функциональных требований тест-кейсами. Есть даже такое понятие как Requirement based testing, которое имеет место быть, когда есть требования к продукту, на их основе составляются тест-сценарии и выполняется тестирование. Зачем нужна эта матрица? Например, для того чтобы: - при разработке тестов четко ориентироваться какие из требований уже покрыты тестами, а какие еще нет; - при выполнении тестирования ориентироваться какие из требований прошли все написанные для них тесты успешно, а какие — еще нет.

Матрица трассировки может служить одновременно в качестве матрицы покрытия. Наличие такой матрицы позволяет объективно оценить, какая часть продукта покрыта тестами, а какая нет. Это необходимое условие, чтобы оценить, какой объем работы мы уже выполнили и что еще осталось сделать — и по части создания, и по части выполнения тестов. Еще одно преимущество traceability matrix – ее наглядность. Если она поддерживается в актуальном состоянии, то можно сразу увидеть «белые пятна» и сосредоточиться на них. Traceability matrix также позволяет сравнивать тесты между собой по критерию количества требований, которые они покрывают. Одни тесты могут покрывать несколько требований, другие – только одно.

**В чем разница между Test matrix и Traceability matrix?**

Тестовая матрица: тестовая матрица используется для определения фактического качества, усилий, плана, ресурсов и времени, необходимых для охвата всех этапов тестирования программного обеспечения.

Матрица прослеживаемости: сопоставление между Test case и требованиями.

**Что такое анализ GAP?**

Анализ пробелов выявляет любые отклонения между функциями, доступными для тестирования, и восприятием их клиентом. Матрица прослеживаемости — это инструмент тестирования, который тестировщики могут использовать для отслеживания пробелов.

**Что такое граф причинно-следственных связей? (Cause Effect Graph)**

Это графическое представление входных данных и связанных с ними выходных эффектов, которые помогают в разработке Test case.

**В чем разница между предугадыванием ошибок и посевом? (Error guessing and error seeding)**

Предугадывание - методика разработки Test Suite, в которой тестировщики должны предполагать возможные дефекты и писать тестовые наборы для их представления.

Seeding. Это процесс добавления известных ошибок в программу для отслеживания скорости обнаружения и удаления. Это также помогает оценить количество неисправностей, оставшихся в программе.

**Стили тестов?**

\*источник утерян, аналогов не нашёл

* Output based verification (смотрим что возвращает)
* State based verification (смотрим состояние)
* Communication based verification (общение и зависимости)

**Техники тестирования требований?**

* Взаимный просмотр:
  + беглый просмотр — автор показывает свою работу коллегам, они в свою очередь дают свои рекомендации, высказывают свои вопросы и замечания;
  + технический просмотр — выполняется группой специалистов;
  + формальная инспекция — привлекается большое количество специалистов, представляет собой структурированный, систематизированный и документированный подход. Минус такого варианта — тратится много времени.
* Вопросы — если возникают вопросы, то можно спрашивать у представителей заказчика, более опытных коллег.
* Тест-кейсы и чек-листы — хорошее требование должно быть проверяемым, чтобы это определить можно использовать чек-листы или полноценные тест-кейсы. Если можно быстро придумать несколько пунктов чек-листа — это уже хороший знак.
* Исследование поведения системы — необходимо мысленно смоделировать процесс работы пользователя с системой, созданной по тестируемым требованиям, после этого определить неоднозначные варианты определения системы.
* Рисунки — графическое представление дает наглядное представление приложения, на рисунке проще увидеть, что какие-то элементы не стыкуются, где-то чего-то не хватает и т. д.
* Прототипирование — сделав наброски пользовательского интерфейса, легко оценить применение тех или иных пользовательских решений

**Что такое эвристики?**

Эвристики – это быстрые, недорогие способы решения проблемы или принятия решения. Эвристики подвержены ошибкам, то есть они могут как сработать, так и не сработать. Эвристики недостаточно абстрактны, они могут перекрываться и пересекаться друг с другом. Также эвристики зависят от контекста

Процесс тестирования на основе эвристик – это такая технология тестирования алгоритмов, приложений и программ, при использовании которой стратегия тестирования основывается на предыдущем опыте и данных о вероятности наступления различных событий.

* Эвристика «Время вышло!» мы останавливаем тестирование, когда заканчивается выделенное на него время.
* Эвристика [пиньяты](http://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D0%BD%D1%8C%D1%8F%D1%82%D0%B0) (The Piñata Heuristic). Мы прекращаем ломать программу, когда начинают выпадать конфеты – мы останавливаем тестирование, когда видим первую достаточно серьезную проблему.
* Эвристика «мертвой лошади». В программе слишком много ошибок, так что продолжение тестирования не имеет смысла. Мы знаем, что все изменится настолько, что сведет на нет результаты текущего тестирования.
* Эвристика «Задание выполнено» останавливаем тестирование, когда найдены ответы на все поставленные вопросы.
* Эвристика «Отмена задания» Наш клиент сказал нам: «пожалуйста, прекратите тестирование». Это может произойти по причине перерасхода бюджета, или вследствие отмены проекта, и по любой другой причине. Какова бы ни была причина, нам поручили остановить тестирование. (На самом деле эвристика «Время вышло!» может быть частным случаем более общей «Отмены задания», в том случае, если предпочтительнее, чтобы не мы сами, а заказчик принял решение о том, что время вышло.)
* Эвристика «Я зашел в тупик!» По какой бы то ни было причине мы останавливаемся, поскольку обнаруживаем некое препятствие. У нас нет информации, которая нам требуется (например, многие люди заявляют, что не могут тестировать без достаточного количества спецификаций). Имеется блокирующая ошибка, и таким образом мы не можем перейти в ту область продукта, которую необходимо протестировать, у нас нет необходимого оборудования или инструментария, у команды нет квалификации, требуемой для выполнения некоторых специальных тестов.
* Эвристика «освежающей паузы» Вместо прекращения тестирования мы приостанавливаем его на некоторое время. Мы можем остановить тестирование и сделать перерыв, когда мы устали, когда нам стало скучно или пропало вдохновение. Мы можем сделать паузу на то, чтобы выполнить некоторые исследования, разработать планы, поразмыслить над тем, что мы делали в прошлом и понять, что делать дальше. Идея заключается в том, что нам требуется определенный перерыв, после которого мы сможем вернуться к продукту со свежим взглядом или свежими мыслями. Также есть и другой вид паузы: мы можем остановить тестирование какой-либо функции, поскольку в настоящий момент другая имеет более высокий приоритет.
* Эвристика «Отсутствие продвижения» Что бы мы ни делали, мы получаем тот же самый результат. Это может происходить в случае, когда программа падает определенным способом или перестает отвечать, но также мы можем не продвигаться, когда программа в основном ведет себя стабильно: «выглядит хорошо!»
* Эвристика Привычного завершения. Мы останавливаем тестирование тогда, когда мы обычно останавливаем тестирование. Имеется протокол, задающий определенное количество идей для тестирования, или тест-кейсов, или циклов тестирования, или как вариант – имеется определенный объем работ по тестированию, который мы выполняем и после этого останавливаемся. Agile-команды, например, часто применяют такой подход: «когда выполнены все приемочные тесты, мы знаем, что продукт готов к поставке. Отличие от эвристики «Время вышло!» в том, что временные ограничения могут изменяться более гибко, чем некоторые другие.
* Больше нет интересных вопросов В этот момент мы решаем, что не осталось вопросов, ответы на которые были бы достаточно ценными, чтобы оправдать стоимость продолжения тестирования, и поэтому мы останавливаемся. Эта эвристика используется в основном как дополнение к другим эвристикам, помогая принять решение о том, есть ли какие-то вопросы или риски, которые отменяют действие этих эвристик (примеры таких вопросов я привожу после каждой эвристики). Кроме того, если одна эвристика советует нам прекратить тестирование, следует проверить, нет ли интересных вопросов или серьезных рисков в других областях, и если они есть, то мы скорее продолжим тестирование, чем остановимся.
* Эвристика уклонения/безразличия. Иногда людей не интересует дополнительная информация, либо они не хотят знать, что происходит в программе. Тестируемое приложение может быть первой версией, которую, как мы знаем, скоро заменят. Некоторые люди прекращают тестирование по причине лени, злого умысла или отсутствия мотивации. Иногда бизнес-критичность выпуска нового релиза настолько высока, что никакая мыслимая проблема не остановит выход программы, и поэтому никакие новые результаты тестирования не будут иметь значения.

Дополнение: Кем Канер (Cem Kaner) предложил еще одну эвристику: «Отказ от выполнения задания» (Mission Rejected), в которой тестировщик сам отказывается от продолжения тестирования. Подробнее читайте здесь.

Процесс тестирования на основе эвристик – это такая технология тестирования алгоритмов, приложений и программ, при использовании которой стратегия тестирования основывается на предыдущем опыте и данных о вероятности наступления различных событий.

Мнемоника – это набор правил и приёмов, которые помогают эффективно запоминать необходимые сведения (информацию).

Большинство экспертов с мировым именем в сфере тестирования настоятельно рекомендуют использовать проверенную схему SFDPOT, автором которой является Джеймс Бах. По их словам, это самый качественный и правильный инструмент для быстрой генерации тестовых идей и наработок.

Итак, что такое SFDPOT?

SFDPOT – Structures, Functions, Data, Platforms, Operations, Times:

* Structures – архитектура приложения (продукта), которая проверяется по частям. На этом этапе создаются тестовые идеи и шаги, неразрывно связанные со структурой веб-продукта;
* Function – производительность приложения (продукта). Проверка того, что может выполнять приложение (продукт). На этом этапе проводится функциональное тестирование ПО;
* Data – взаимодействие с данными. Проверка приложения на взаимодействие с данными. QA специалисты должны определить по какой логике продукт взаимодействует с данными, как проходит их получение, тип обработки и виды информации;
* Platform – экосистема, платформа. Выполнение проверки того, как именно приложение (продукт) потенциально взаимодействует с платформой, на которой оно создано и запущенно. Тестировщик должен определить, на каких именно платформах и внутри каких систем необходимо провести процедуру ручного и автоматизированного тестирования;
* Operations – проверка созданных сценариев для разработанного приложения. На этом этапе перед тестировщиками постает задача по выяснению потенциального круга будущих пользователей, которые будут взаимодействовать с создаваемым продуктом;
* Time – период времени, проверка того, как продукт ведет себя в зависимости от наступления или завершения каких-либо временных промежутков.

Что же касается CRUCSPIC STMP, то этот метод может рассматриваться как составная часть Operations внутри методики SFDPOT.

Если вкратце сказать о функциональности CRUCSPIC STMP, то можно отметить, что это своего рода атрибуты системы. Наработки CRUCSPIC STMP позволяют выделить основные объекты тестирования, его целей, а также построить карту эффективности взаимодействия между собой.