----- Автоматизация (beta) -----

Это черновик большого раздела автоматизации тестирования. Здесь не будет подробных гайдов по инструментам с практикой и кодом. Вместо этого, в рамках своих скромных компетенций, я бы хотел рассмотреть азы, дающие представление об общей картине: что это, зачем оно надо, какая бывает автоматизация, что следует автоматизировать, общие сведения о CI/CD, инфраструктуре, месте автотестов в общем пайплайне, механизме их запуска и отчетности, а также темы, полезные для поиска работы автоматизатором.

Общее

- *Автоматизация тестирования (test automation): Использование программного обеспечения для осуществления или помощи в проведении определенных тестовых процессов, например, управление тестированием, проектирование тестов, выполнение тестов и проверка результатов. (ISTQB)*
- *Автоматизация выполнения тестов (test execution automation): Использование программного обеспечения (например, средств захвата/воспроизведения) для контроля выполнения тестов, сравнения полученных результатов с эталонными, установки предусловий тестов и других функций контроля тестирования и организации отчетов. (ISTQB)*
- *Автоматизированное тестирование (scripted testing): Выполнение тестов, реализуемое при помощи заранее записанной последовательности тестов. (ISTQB)*
- *Автоматизированное тестовое обеспечение (automated testware): Тестовое обеспечение, используемое в автоматизированном тестировании, например, инструментальные сценарии. (ISTQB)*
- *Автоматизированный сценарий тестирования (test script): Обычно используется как синоним спецификации процедуры тестирования, как правило, автоматизированной. (ISTQB)*

Автоматизированное тестирование (automated testing, test automation) - набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования.

Многие задачи и действия, описанные в ИСО/МЭК/ИИЭР 29119-2 как процессы Менеджмента Тестирования и Динамического Тестирования, а также многие аспекты методов тестирования, представленные в ИСО/МЭК/ИИЭР 29119-4, могут быть автоматизированы. Автоматизация тестирования требует использования программного обеспечения, обычно называемого инструментами тестирования.

Области применения автоматизации:

- **Регрессионное тестирование**: необходимость выполнять вручную тесты, количество которых неуклонно растет с каждым билдом, но вся суть которых сводится к проверке того факта, что ранее работавшая функциональность продолжает работать корректно.
- **Инсталляционное тестирование и настройка тестового окружения**: множество часто повторяющихся рутинных операций по проверке работы инсталлятора, размещения файлов в файловой системе, содержимого конфигурационных файлов, реестра и т.д. Подготовка приложения в заданной среде и с заданными настройками для проведения основного тестирования.
- **Конфигурационное тестирование и тестирование совместимости**: выполнение одних и тех же тест-кейсов на большом множестве входных данных, под разными платформами и в разных условиях. Классический пример: есть файл настроек, в нём сто параметров, каждый может принимать сто значений: существует 100 100 вариантов конфигурационного файла - все их нужно проверить.
- **Использование комбинаторных техник тестирования** (в т.ч. доменного тестирования): генерация комбинаций значений и многократное выполнение тест-кейсов с использованием этих сгенерированных комбинаций в качестве входных данных.

- **Модульное тестирование**: проверка корректности работы атомарных участков кода и элементарных взаимодействий таких участков кода - практически невыполнимая для человека задача при условии, что нужно выполнить тысячи таких проверок и нигде не ошибиться.
- **Интеграционное тестирование**: глубокая проверка взаимодействия компонентов в ситуации, когда человеку почти нечего наблюдать, т.к. все представляющие интерес и подвергаемые тестированию процессы проходят на уровнях более глубоких, чем пользовательский интерфейс.
- **Тестирование безопасности**: необходимость проверки прав доступа, паролей по умолчанию, открытых портов, уязвимостей текущих версий ПО и т.д., т.е. быстрое выполнение очень большого количества проверок, в процессе которого нельзя что-то пропустить, забыть или «не так понять».
- **Тестирование производительности**: создание нагрузки с интенсивностью и точностью, недоступной человеку. Сбор с высокой скоростью большого набора параметров работы приложения. Анализ большого объема данных из журналов работы системы автоматизации.
- **Дымовой тест для крупных систем**: выполнение при получении каждого билда большого количества достаточно простых для автоматизации тест-кейсов.
- **Приложения (или их части) без графического интерфейса**: проверка консольных приложений на больших наборах значений параметров командной строки (и их комбинаций). Проверка приложений и их компонентов, вообще не предназначенных для взаимодействия с человеком (веб-сервисы, серверы, библиотеки и т.д.).
- **Длительные, рутинные, утомительные для человека и/или требующие повышенного внимания операции**: проверки, требующие сравнения больших объёмов данных, высокой точности вычислений, обработки большого количества размещенных по всему дереву каталогов файлов, ощутимо большого времени выполнения и т.д. Особенно, когда такие проверки повторяются очень часто.
- **Проверка «внутренней функциональности» веб приложений** (ссылок, доступности страниц и т.д.): автоматизация предельно рутинных действий (например, проверить все 30’000+ ссылок на предмет того, что все они ведут на реально существующие страницы). Автоматизация здесь упрощается в силу стандартности задачи - существует много готовых решений.
- **Стандартная, однотипная для многих проектов функциональность**: даже высокая сложность при первичной автоматизации в таком случае окупится за счёт простоты многократного использования полученных решений в разных проектах.
- **“Технические задачи”**: проверки корректности протоколирования, работы с базами данных, корректности поиска, файловых операций, корректности форматов и содержимого генерируемых документов и т.д.

Преимущества автоматизации:

- Скорость выполнения тест-кейсов может в разы и на порядки превосходить возможности человека. Если представить, что человеку придётся вручную сверять несколько файлов размером в несколько десятков мегабайт каждый, оценка времени ручного выполнения становится пугающей: месяцы или даже годы. При этом 36 проверок, реализуемых в рамках дымового тестирования командными скриптами, выполняются менее чем за пять секунд и требуют от тестировщика только одного действия - запустить скрипт;
- Отсутствует влияние человеческого фактора в процессе выполнения тест кейсов (усталости, невнимательности и т.д.). Продолжим пример из предыдущего пункта: какова вероятность, что человек ошибется, сравнивая (посимвольно!) даже два обычных текста размером в 100 страниц каждый? А если таких текстов 10? 20? И проверки нужно повторять раз за разом? Можно смело утверждать, что человек ошибется гарантированно. Автоматика не ошибется;
- Средства автоматизации способны выполнить тест-кейсы, в принципе непосильные для человека в силу своей сложности, скорости или иных факторов. И снова наш пример со сравнением больших текстов является актуальным: мы не можем позволить себе потратить годы, раз за разом выполняя крайне сложную рутинную операцию, в которой мы к тому же будем гарантированно допускать ошибки. Другим прекрасным примером непосильных для человека тест-кейсов является исследование

производительности, в рамках которого необходимо с высокой скоростью выполнять определённые действия, а также фиксировать значения широкого набора параметров. Сможет ли человек, например, сто раз в секунду измерять и записывать объем оперативной памяти, занимаемой приложением? Нет. Автоматика сможет;

- Средства автоматизации способны собирать, сохранять, анализировать, агрегировать и представлять в удобной для восприятия человеком форме колоссальные объемы данных. В нашем примере с дымовым тестированием «Конвертера файлов» объем данных, полученный в результате тестирования, невелик - его вполне можно обработать вручную. Но если обратиться к реальным проектным ситуациям, журналы работы систем автоматизированного тестирования могут занимать десятки гигабайт по каждой итерации. Логично, что человек не в состоянии вручную проанализировать такие объёмы данных, но правильно настроенная среда автоматизации сделает это сама, предоставив на выход аккуратные отчеты в 2-3 страницы, удобные графики и таблицы, а также возможность погружаться в детали, переходя от агрегированных данных к подробностям, если в этом возникнет необходимость;
- Средства автоматизации способны выполнять низкоуровневые действия с приложением, операционной системой, каналами передачи данных и т.д. В одном из предыдущих пунктов мы упоминали такую задачу, как «сто раз в секунду измерить и записать объем оперативной памяти, занимаемой приложением». Подобная задача сбора информации об используемых приложением ресурсах является классическим примером. Однако средства автоматизации могут не только собирать подобную информацию, но и воздействовать на среду исполнения приложения или само приложение, эмулируя типичные события (например, нехватку оперативной памяти или процессорного времени) и фиксируя реакцию приложения. Даже если у тестировщика будет достаточно квалификации, чтобы самостоятельно выполнить подобные операции, ему всё равно понадобится то или иное инструментальное средство - так почему не решить эту задачу сразу на уровне автоматизации тестирования?

Недостатки автоматизации:

- Необходимость наличия высококвалифицированного персонала в силу того факта, что автоматизация - это «проект внутри проекта» (со своими требованиями, планами, кодом и т.д.). Даже если забыть на мгновение про «проект внутри проекта», техническая квалификация сотрудников, занимающихся автоматизацией, как правило, должна быть ощутимо выше, чем у их коллег, занимающихся ручным тестированием.
- Разработка и сопровождение как самих автоматизированных тест-кейсов, так и всей необходимой инфраструктуры занимает очень много времени. Ситуация усугубляется тем, что в некоторых случаях (при серьёзных изменениях в проекте или в случае ошибок в стратегии) всю соответствующую работу приходится выполнять заново с нуля: в случае ощутимого изменения требований, смены технологического домена, переработки интерфейсов (как пользовательских, так и программных) многие тест-кейсы становятся безнадежно устаревшими и требуют создания заново.
- Автоматизация требует более тщательного планирования и управления рисками, т.к. в противном случае проекту может быть нанесен серьезный ущерб (см. предыдущий пункт про переделку с нуля всех наработок).
- Коммерческие средства автоматизации стоят ощутимо дорого, а имеющиеся бесплатные аналоги не всегда позволяют эффективно решать поставленные задачи. И здесь мы снова вынуждены вернуться к вопросу ошибок в планировании: если изначально набор технологий и средств автоматизации был выбран неверно, придётся не только переделывать всю работу, но и покупать новые средства автоматизации.
- Средств автоматизации крайне много, что усложняет проблему выбора того или иного средства, затрудняет планирование и определение стратегии тестирования, может повлечь за собой дополнительные временные и финансовые затраты, а также необходимость обучения персонала или найма соответствующих специалистов.

Исходя из вышеперечисленного, существуют **случаи, в которых автоматизация, скорее всего, приведёт только к ухудшению ситуации**. Вкратце - это все те области, где требуется человеческое мышление, а также некоторый перечень технологических областей:

- Планирование, разработка тест-кейсов, написание отчетов о дефектах, анализ результатов тестирования и отчётность: компьютер пока не научился думать;
- Функциональность, которую нужно (достаточно) проверить всего несколько раз, тест-кейсы, которые нужно выполнить всего несколько раз (если человек может их выполнить): затраты на автоматизацию не окупятся;
- Низкий уровень абстракции в имеющихся инструментах автоматизации: придётся писать очень много кода, что не только сложно и долго, но и приводит к появлению множества ошибок в самих тест-кейсах.
- Слабые возможности средства автоматизации по протоколированию процесса тестирования и сбору технических данных о приложении и окружении: есть риск получить данные в виде «что-то где-то сломалось», что не помогает в диагностике проблемы.
- Низкая стабильность требований: придётся очень многое переделывать, что в случае автоматизации обходится дороже, чем в случае ручного тестирования.
- Сложные комбинации большого количества технологий: высокая сложность автоматизации, низкая надёжность тест-кейсов, высокая сложность оценки трудозатрат и прогнозирования рисков.
- Проблемы с планированием и ручным тестированием: автоматизация хаоса приводит к появлению автоматизированного хаоса, но при этом ещё и требует трудозатрат. Сначала стоит решить имеющиеся проблемы, а потом включаться в автоматизацию.
- Нехватка времени и угроза срыва сроков: автоматизация не приносит мгновенных результатов. Поначалу она лишь потребляет ресурсы команды (в том числе время). Также есть универсальный афоризм: «лучше руками протестировать хоть что-то, чем автоматизированно протестировать ничего».
- Области тестирования, требующие оценки ситуации человеком (тестирование удобства использования, тестирование доступности и т.д.): в принципе, можно разработать некие алгоритмы, оценивающие ситуацию так, как её мог бы оценить человек. Но на практике живой человек может сделать это быстрее, проще, надёжнее и дешевле.

Далее копипаста различных вопросов-ответов из цикла статей на хабре.

Зачем нужна автоматизация, если наши ручные тестировщики и так справляются?

Действительно, прежде чем вводить автоматизацию на проекте, потому что это “стильно, модно, молодежно”, и у всех оно есть, стоит учесть следующие факторы:

- Как долго будет жить проект, каков его масштаб, насколько сильно и часто он меняется;
- Оценить, насколько трудозатратно будет писать автотесты в каждом конкретном случае. Возможно, что автоматизировать ваш функционал будет гораздо сложнее, дольше и дороже, чем тестировать его руками;
- Также, если на проекте ожидается “выпиливаем всё старое и пишем новое с нуля”, время автотестов еще не пришло.

Но если ваш проект:

- стабильно развивается и растет;
- увеличивается количество разработчиков;
- приложение начинает обрастать новым функционалом.

То прямо пропорционально начнет увеличиваться и время тестирования. Рано или поздно оно дойдет до критического момента, когда регресс будет занимать больше времени, чем велась разработка. Это значит, что пришло время задуматься об автоматизации тестирования.

Автотесты станут гарантией, что ничего из старого не отвалилось, пока разработчики делали новые крутые штуки. Особенно, если эти новые штуки делались в совершенно других частях кода. Ручные регрессы, в данном случае, это, конечно, хорошо, но тут и человеческий фактор, и замыленный глаз, и непредсказуемость мест, в которых могут всплыть баги.При этом ручное тестирование полностью никуда не уходит, так как новый функционал все еще будет проверяться руками. Нет смысла автоматизировать то, что возможно не взлетит и уже через неделю будет выпилено или отправлено на доработки.

Но вместо того, чтобы занимать ручного тестировщика написанием тест-кейсов на новый функционал, поддержку документации и постоянными регрессами, можно выделить ему хотя бы один день в неделю на автоматизацию, помочь на старте с настройками инфраструктуры, и уже через несколько месяцев у вас начнут гонять первые автотесты, ваши задачи станут в разы быстрее доходить до релизов а заодно станет больше уверенности в стабильности выпускаемого кода.

Еще автоматизированный регресс позволяет тестировщикам больше времени уделять полезным задачам, например, исследовательскому тестированию. Кроме того, появится лучшее понимание, как работает приложение под капотом, что точно пригодится даже в ручном тестировании.

Можно ли начать писать автотесты за один день?

Писать автотесты можно начать прямо в этот самый момент, но лучше с этим не торопиться и подойти к вопросу осмысленно.

Если на проекте еще не было автоматизации, то нужно обязательно начать с ресерча существующих инструментов, фреймворков, подходов. Почитать о том, с какими проблемами сталкиваются пользователи, и как быстро они решаются. Оценить все плюсы и минусы, выбрать самое подходящее, с чем будет легко и приятно работать. Цена ошибки на старте намного меньше, чем когда вы уже активно автоматизируете и вдруг понимаете, что выбранный вами фреймворк никому не удобен и совершенно вам не подходит.

Если вы никогда раньше не занимались автотестами, первое время может быть сложно.. Непросто начать писать автотесты сразу, если еще нет никаких примеров и не на что опереться. Благо, сейчас в интернете очень много различных статей с массой хороших примеров кода, которые можно брать, подстраивать под себя и разбираться, как они работают.

В конце концов, чтобы научиться автоматизировать, надо просто начать автоматизировать. Это действительно работает.

Идеально, если со стороны разработчиков будут те, кто готов на первых порах вам помогать разбираться в коде, отвечать на все ваши вопросы. В hh именно так и было - первое время, когда мы только начинали писать свои первые автотесты, мы просто заваливали наших ребят всевозможными вопросами. Благодаря им мы и пришли к тому, что имеем сегодня

Но вполне возможно, что у вашей команды не будет времени помогать вам с автотестами. К сожалению, такое возможно, и не стоит их за это винить. Если вы попали в такую ситуацию, не отчаивайтесь! В интернете очень мощное комьюнити тестировщиков. В том же самом телеграме есть чаты, где 24/7 вам ответят на все вопросы и помогут разобраться.

Можно ли заставить разработчиков писать автотесты?

В теории, конечно можно. Но скорее всего они будут не сильно этому рады, да и вряд ли кто-то будет рад в принципе.

Во-первых, это замедлит саму разработку. Вместо того, чтобы писать код, разрабатывать продуктовый функционал, разработчики будут заниматься написанием автотестов.

Во-вторых, это дорого.

Ну и в-третьих, у каждого должна быть своя зона ответственности. Разработчики создают функционал, тестировщики - его проверяют. С автотестами все точно так же, как и в обычном тестировании. Вот юнит-тесты - да, зона ответственности разработчиков, но UI - это уже тестировщеское.

По факту, этот вопрос равен “Зачем нам тестировщики, если можно попросить разработчиков самим проверять то, что они написали?”.

Не замедляют ли автотесты разработку?

Начнем с того, что в первую очередь автотесты мощно сокращают время на тестирование и регрессы, а значит помогают выпускать фичи в релизы быстрее.

Но, действительно, иногда они увеличивают время разработки. Например, когда разработчик меняет покрытый автотестами функционал, в связи с чем приходится перед мержем своего кода править падающие автотесты.

Чтобы это не становилось проблемой, при оценке и декомпозиции задачи нужно закладывать время на возможные фиксы автотестов. В идеале на таких встречах должен присутствовать тестировщик, который сможет сказать, какие автотесты могут быть затронуты новым функционалом. Тогда время задачи будет предсказуемым и не увеличится.

Но чтобы это нормально работало, нужно, чтобы на проекте была стабильная инфраструктура, и автотесты падали по понятным причинам. Тогда разработчикам не придется тратить свое драгоценное время, чтобы в каждой задаче разбираться, из-за чего именно упал автотест - из-за его кода или из-за проблем на тестовом стенде.

Можно ли писать 1 автотест сразу на обе платформы?

Конечно, можно. Существуют кроссплатформенные фреймворки, и многие их используют. Но мы сразу выбрали для себя нативные, как более стабильные, надежные и легко поддерживаемые.

В чем главные проблемы кроссплатформенных фреймворков?

Это отдельно живущий от вашего приложения код. Если вам понадобится в нем что-то доработать или настроить под себя, то придется идти к другим разработчикам, делать запросы и ждать, пока они сделают и выпустят это на своей стороне, что не всегда удобно и продуктивно. Частые проблемы при выходе новых ОС и баги. Нужно ждать поддержку со стороны разработчиков фреймворка, и это ожидание весьма непредсказуемое. Вы не знаете, когда всё поправят и оптимизируют, и когда оно у вас наконец заработает. То есть вы зависимы от других людей. Возможно, выбранный вами кроссплатформенный фреймворк написан на языке программирования, который не знаком вашим разработчикам. Тогда помощь от них будет получить сложнее, если она вам вдруг понадобится. Также разработчики в этом случае не смогут писать и править автотесты самостоятельно.

Может показаться, что написать один кроссплатформенный автотест быстрее и проще, чем два нативных. Но на самом деле это не так.

Во-первых, всё “сэкономленное” время в последствии может уйти на поддержку такого автотеста для двух платформ. Это может стать проблемой из-за обновляемой специфики разных ОС. Кроме того, часто поведение одной и той же фичи или кнопки может различаться для iOS и Android.

Во-вторых, если при выборе между кроссплатформенным фреймворком и нативными, главным аргументом является “выучить один язык программирования проще, чем два”, не спешите принимать решение. Синтаксис языков Kotlin и Swift очень похожи (по-крайней мере, в вопросах написания автотестов). Написав автотест на одном из языков, вы без труда напишете точно такой же для второй платформы.

Когда автотесты начнут приносить больше пользы, чем проблем?

Ровно тогда, когда:

- на вашем проекте будет настроена стабильная инфраструктура,
- автотесты будут встроены в ваш CI/CD, а не запускаться локально у кого-то, когда об их существовании вспомнили,
- автотесты будут падать приемлемое для вас количество раз, то есть не будут флаковать.

Даже если тестов совсем немного, но они уже гоняются хотя бы перед мержем ветки разработчика, то это уже польза: минус какое-то количество ручных проверок и гарантия того, что ваше приложение как минимум запускается.

За сколько месяцев можно прийти к 100% покрытию?

Вопрос 100% покрытия - очень скользкий. Не хочется вдаваться в бесполезные споры о том, существует ли оно вообще и нужно ли к нему стремиться. Но поскольку его задают очень часто, давайте разбираться.

Во-первых, стоит рассматривать покрытие каждой фичи отдельно.

Во-вторых, глобально ответ на вопрос “на сколько у тебя покрыта эта фича” будет основываться на опыте и знании продукта тестировщика.

Оценивая покрытие фичи, я рассуждаю следующим образом:

все позитивные проверки - это 60-70% от всего покрытия,

далее добавляем негативные проверки (всевозможные прерывания, сворачивания-разворачивания экранов, зероскрины и т.п.) - получаем 90%.

Оставшиеся 10% - это сложновоспроизводимые сценарии, зависимости от других фичей и т.д., которые покрывать либо вообще не стоит (подумайте, выдержит ли такую нагрузку ваш CI и стоит ли того время на поддержку таких автотестов), либо оставлять это на самую последнюю очередь. Также добавлю, что не все фичи в принципе нужно автоматизировать. Иногда трудозатраты на автоматизацию какой-то невероятно сложной проверки сильно преувеличивают возможность один раз перед релизом быстро проверить ее руками. Чтобы не оставлять вопрос менеджера без ответа - выявляем среди всей функциональности приложения наиболее критичный и в нём смотрим, какое количество позитивных и негативных проверок нам нужно автоматизировать в первую очередь, чтобы быть уверенными, что функционал работает. Далее смотрим какое количество тестировщиков сколько времени готовы уделять автоматизации и считаем, как долго мы будем это автоматизировать.

Сколько нужно тестировщиков, чтобы автоматизировать весь проект?

Расскажу историю об автоматизации мобильных приложений в hh.ru.

В разные периоды времени у нас было от 2 до 4 тестировщиков на всё мобильное направление. Примерно за год мы пришли к практически полному покрытию регрессов на обе платформы. И это с учетом того, что мы продолжали тестировать функционал руками и проводить ручные регрессы, а на андроид дважды переезжали на новый фреймворк. В итоге, кстати, мы остановились на Kaspresso, с которым тесты стали сильно стабильнее и проще в написании. Каждый переезд был большим объемным рефакторингом, так как до этого мы уже успели написать какое-то значительное количество автотестов. Но тем не менее, за один год регрессы были автоматизированы.

Иными словами, любым количеством тестировщиков можно автоматизировать проект, если выделить на это достаточно времени. Но возникает вопрос: как тестировщику успевать и руками тестировать, и автотесты писать? Иногда это действительно сложно. Особенно если идет большой поток нового функционала, который срочно нужно тестировать. Всем хочется успеть в релиз, и задачам ставят больший приоритет. Знакомая ситуация. Что делать? Садимся. Понимаем, что автотесты в недалеком будущем принесут нам максимум

пользы и выделяем каждому тестировщику, например, один день в неделю под автоматизацию. Мы такое практикуем, когда у тестировщиков завал с ручным тестированием, и это работает. Для автоматизации можно выбрать день после релиза, когда уже ничего не горит, и тестировщик с чистой совестью может писать автотесты, не отвлекаясь на другие задачи.

Как вам живется с автоматизированными регрессами?

За два года, что мы живем с автоматизированным регрессом, у нас надежно устаканился недельный релиз-трейн. Релизы регулярно отправляются в прод без проведения ручных регрессов благодаря количеству, качеству и, главное, доверию к автотестам.

Помимо того, что автотесты запускаются на релизных ветках, весь набор автотестов всегда прогоняется на фиче-ветках разработчиков перед мерджем. В девелоп/мастер не допускается код, который уронил тесты. Благодаря этому мы поддерживаем стабильность девелопа, мы уверены в нем и можем отрезать релизную ветку в любой момент.

Для тестировщика автоматизированный регресс может означать больше свободного времени для более важных и полезных задач - это, например, написание новых автотестов, увеличение стабильности существующих, развитие своего направления и тому подобные крутые задачи.

Много ли времени уходит на поддержку автотестов?

Поддержка автотестов встроена в наши процессы разработки. Как это работает - каждую неделю назначается дежурный тестировщик, одной из задач которого является следить за стабильностью тестов. Если тест упал, дежурный должен разобраться в причине падения и назначить ответственного на фикс. Если тест упал из-за собственной нестабильности - задача на тестировщика, если тест нашел баг - на разработчика, код которого повлиял на этот автотест. По опыту, чаще всего это не занимает много времени - один тестировщик тратит несколько часов за неделю.

Более сложные задачи - развитие, улучшение и оптимизация, фиксы инфраструктуры. Такие задачи идут наравне с разработкой продуктового функционала и имеют соответствующий флоу (проработка, декомпозиция, оценка, разработка и тд). Это случается не слишком часто, в среднем раз в квартал на каждую платформу. Тут важно отметить, что чаще всего это не самые высокоприоритетные задачи, и делаются они, когда у команды появляется на это ресурс.

Если всё четко настроить, научиться работать с флакующими тестами и по-максимуму их не допускать, то поддержка автотестов занимает мало времени и приносит много пользы.

Насколько сложно найти мобильного автоматизатора?

Порой сложно найти просто хорошего мобильного тестировщика с релевантным опытом, а с опытом автоматизации в конкретном стеке технологий дела обстоят еще сложнее.

В мобильные команды hh.ru мы обычно ищем просто крутых ручных тестировщиков, которые хотят развиваться в сторону автоматизации. Во время собеседований базовые знания в программировании, конечно, являются плюсом, но не решающим фактором. Автоматизации мы готовы обучать у себя. А вот что действительно важно - чтобы человек стремился развиваться, умел и хотел обучаться и был проактивным в этих моментах. Конечно, сложно точно определить такое во время собеседований, но и не совсем невозможно.

Источники:

- Святослав Куликов “Тестирование программного обеспечения. Базовый курс”. Раздел 3: Автоматизация тестирования
- ТОП-5 вопросов ручных тестировщиков про автоматизацию
- ТОП-5 вопросов менеджера про автоматизацию

- ТОП-5 вопросов технического директора про автоматизацию

Доп. материал:

- Введение в автоматизированное тестирование
- 10 Tips You Should Read Before Automating Your Testing Work
- Manual And Automation Testing Challenges
- How To Translate Manual Test Cases Into Automation Scripts? - A Step By Step Guide With Example
- Test Automation - Is It A Specialized Career? Can Normal Testers Do Automation Also?

Полезные ссылки

Telegram

- Подборка сообществ в помощь QA automation инженеру
- Большая подборка чатов @it_chats
- Чат для студентов БЕСПЛАТНОЙ ШКОЛЫ QA automation для всех + F.A.Q.
- Чат для начинающих автоматизаторов @aqa_chatka

Репозитории

- Awesome Selenium
- Awesome Appium
- Awesome Visual Regression Testing
- Awesome JMeter
- Awesome k6
- Awesome Playwright
- Awesome Gatling
- Awesome JMeter
- 10 интересных репозиториев на GitHub, полезных любому разработчику

Youtube

- Eugene Suleimanov
- alishev
- Тестировщик
- Антон Семенченко
- dmdev
- Лёша Маршал
- SDET- QA Automation Techie
- Автоматизация CI/CD - Полный Курс на Простом Языке
- Плейлист “Автоматизация с нуля”
- Плейлист “Python - Starter”
- Плейлист Курс программирования - "Python Essential"
- Плейлист “Вебинары Python”
- Плейлист “Практики и инструменты DevOps”
- Годные туториалы на YouTube

Курсы

- Лекции от Всеволода Брекелова
- Питонтьютор - бесплатный курс по программированию с нуля на Python
- Программирование на Python
- Python: основы и применение
- Автоматизация тестирования с помощью Selenium и Python

- Бесплатный курс “Введение в программирование”
- Test Automation University
- Continuous Integration with Jenkins
- Автоматизированное тестирование с нуля / Полный курс за 3 часа / selenium + testng
- Python за месяц
- Как научиться разработке на Python: новый видеокурс Яндекса
- Python.org рекомендует: Программирование для НЕпрограммистов

Сборники материалов по автоматизации

- What Is Automation Testing (Ultimate Guide To Start Test Automation)

Книги

- Swaroop Chitlur - A Byte of Python (есть в переводе)
- Цикл статей с переводом Okken Brian - Python Testing with pytest
- Топ-10 книг для разработчика

Площадки для тренировки

- Лучшие сайты для практики автоматизации тестирования
- Сайты-песочницы, на которых можно практиковать написание автотестов
- Skillotron QA Auto Tests
- https://academybugs.com/find-bugs/
- Selenium Certification Training
- Codewars, HackerRank, Coderbyte, CodinGame, LeetCode, Topcoder, Project Euler, CodeFights
- Шесть бесплатных автоматизированных платформ для изучения программирования
- Песочница и шпаргалка по изучению Python
- Где порешать реальные задачи для кандидатов в Яндекс: тренировка на Codeforces и разбор

Как стать автоматизатором и вопросы с собеседований

Можно ли стать автоматизатором без опыта ручного тестирования?

Можно, если у вас есть опыт в программировании. В каких-то компаниях это действительно так и работает. Ручные тестировщики пишут тест-кейсы (шаги + ожидаемый результат), автоматизатор их берет и переносит в код. В принципе, такой подход вполне валиден и работает, но я вижу в нем некоторые недостатки.

Во-первых, когда сам пишешь автотесты на функционал, который хорошо знаешь, ты можешь по ходу добавлять какие-то проверки, которые мог пропустить во время написания тест-кейсов. Плюс знаешь другие автотесты, которые можно дополнить. Соответственно, тебе легче поддерживать актуальность автотестов.

Во-вторых, автоматизация тестирования - это интересно и полезно. Ты начинаешь изучать код, расширяешь свои знания о продукте, понимаешь, как всё работает изнутри. Это полезно и для ручного тестирования в том числе. Начинаешь чуть лучше понимать разработчиков.

Можно ли писать автотесты автоматически? Не хочется учиться программированию.

Попробовать можно. Мы пробовали. Для таких дел существуют рекордеры. Но те тесты, которые ими создаются - это монструозные и неподдерживаемые куски кода.

Возможно, это будет работать, если, допустим, в приложении есть какая-то кнопка, которая никогда не будет меняться. Не изменится ни путь до нее, ни ее функциональность и положение. Тогда код этого теста никогда не нужно будет менять, и пусть этот тест будет жить. Но увы, на практике так не работает. Тесты должны быть легко поддерживаемыми, понятными, читаемыми. Рекордером такого не добьешься.

Можно использовать рекордеры в каких-нибудь сложных местах приложения, чтобы посмотреть, как можно повзаимодействовать с каким-нибудь труднонаходимым элементом. То есть использовать его как помощника, как вспомогательный инструмент, но не как основное средство автоматизации.

За сколько тестировщик превращается в автотестировщика

Опять же, по нашему опыту, мы нанимаем человека без опыта автоматизации и на испытательный срок (3 месяца) ему ставится задача - написать свой первый автотест на любую из платформ, которая ему понравится больше или покажется проще. И у нас еще никто не провалил испытательный срок.

Естественно, большую роль играет то, что человек пишет автотесты не совсем с нуля. У нас уже есть и готовые автотесты, которые можно смотреть и писать по аналогии, и люди, которые готовы помогать и отвечать на вопросы.

По итогу за 3 месяца мы получаем человека, который уже понимает, как писать автотесты минимум для одной из платформ. Следующим шагом будет написать такой же тест для второй платформы. Еще через 3-4 месяца мы получим самостоятельного автоматизатора мобильных приложений под обе платформы, которому еще какое-то время, возможно, нужна будет помощь с какими-то сложными вещами. Но вот свободно писать легкие автотесты под обе платформы он будет уже через полгода.

Карьерный путь автоматизатора

- Что учить, чтоб стать автоматизатором тестирования
- Карьерный путь автоматизатора

Роадмапы в основном включают и мануал и авто, их можно посмотреть в теме “Что должен знать и уметь Junior? Что спросят на собеседовании?”.

Вопросы для подготовки к собеседованию можно условно поделить на 3 большие группы:

- джуна наверняка всё-равно будут спрашивать общую теория тестирования по мануалу, хотя бы по верхам;
- всё то, что касается непосредственно автоматизации: какая бывает, инструменты в общем и конкретно под вакансию, представление об инфраструктуре CI/CD, лучшие практики автоматизации и т.п.;
- core языка программирования, указанного в вакансии и всё, что вокруг этого.

Вопросы по автоматизации:

- Что такое автоматизация и зачем она нужна?
- Когда нужно начинать автоматизацию на проекте?
- Какая бывает автоматизация (виды, методы, платформы и т.п.)?
- Популярные фреймворки и инструменты автоматизации, запуска тестов и генерации отчетности;
- Инфраструктура CI/CD, пайплайн, место автотестов в нем;
- Что следует автоматизировать в первую очередь?
- Какая тестовая документация нужна для автоматизированного тестирования?

Вопросы по языкам программирования:

Java:

- дизайн-паттерны;
- дата-типы;
- коллекции, Map...;
- модификаторы доступа. Public, Private, Abstract классы и методы;
- Что такое Интерфейс?
- Что такое лямбда функция?

- дженерики;
- коллекции;
- методы класса object;
- больше тут или в гугле.

Вопросы общие по типу:

- Разница между библиотекой и фреймворком?
- Что означает слово SNAPSHOT в версии библиотеки?
- Что такое SDK?

Практические навыки:

- уметь писать код;
- Git;
- консоль;
- типовые инструменты для платформы;
- моки запросов (Swifter/Wiremock);
- инструменты отчетности (Allure);
- инструменты CI.

Пример вопросов от кандидата работодателю:

- Сколько IOS разработчиков в приложении?
- Сколько Unit Test'ов и сколько UI Test'ов на данный момент?
- С какой периодичностью запускаются тесты?
- Какой релизный цикл? Сколько сейчас времени на регресс?
- Кто добавляет Accebility Identifier'ы в приложение?
- Какая минимальная версия IOS поддерживается?
- Сколько времени тратится на сборку приложения локально?
- Какая система сборки используется на проекте?
- Автоматизируете ли разрешение конфликтов в project.pbxproj ?

Источники:

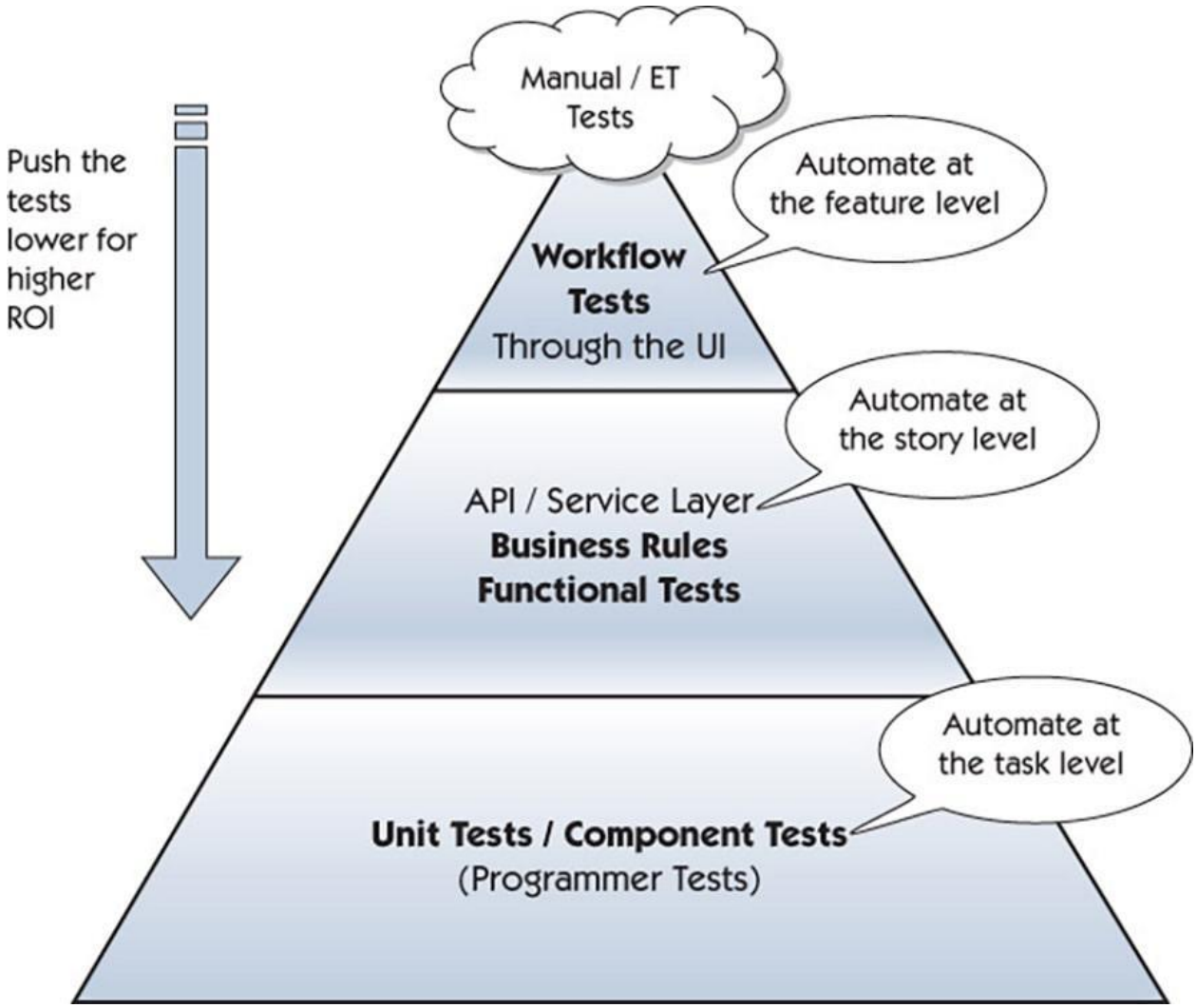
- ТОП-5 вопросов ручных тестировщиков про автоматизацию
- ТОП-5 вопросов технического директора про автоматизацию

Доп. материал:

- Чек-лист для начинающего автотестера на Java
- **39 TOP Automation Testing Interview Questions And Answers**
- **50 Most Popularly Asked Selenium Interview Questions And Answers**
- **Automation Testing Interview Questions And Answers (Updated 2022)**
- **Interview Prep Questions**
- 50 вопросов по Docker, которые задают на собеседованиях, и ответы на них
- Как начать карьеру QA Automation Engineer: один простой совет
- Нужно ли знать программирование для qa автоматизатора?
- Как стать QA AUTOMATION engineer с нуля самостоятельно
- Дмитрий Бормотов - Трансформация из Manual QA в Automation
- Как стать автоматизатором тестирования?
- Какие вопросы ожидать на позицию автоматизатора и причем тут сортировка?
- Что спрашивают на собеседовании у джуна, или как я искала свою вторую работу в ИТ
- 10 Awesome Tips To Become A Better Automation Tester

- Какие ошибки совершает начинающий QA Automation Engineer? Как их избежать?
- Три типичных ошибки автоматизатора
- QAGuild#54: Что должен знать тестировщик? Топ 3 навыка для QA Automation engineer
- Как стать Java разработчиком за 1,5 года
- Как я изучал структуры данных и алгоритмы для собеседования в FAANG
- Как я готовился к собеседованию в Google

Что нужно автоматизировать?



Какие модули и места следует подвергать автоматизации?

- Участки кода, исполнение которых трудно визуализировать и получить осязаемую информацию о протекающих процессах (back-end процессы, занесение в базу данных, занесение логов в файл);
- Функциональность продукта, которая будет использоваться наиболее часто и возникновение ошибок которой связано с достаточно высоким риском. Автоматизированное тестирование узловых моментов функциональности потребует меньше времени для поиска ошибок. И соответственно, сократит время на их устранение;
- Типовые часто выполняемые операции, которые обычно связаны с обработкой данных (CRUD). Например - формы, в которых количество заполняемых граф и полей довольно значительное. Цель - автоматизировать занесение требуемых данных в нужное поле и проверить правильность выполнения задачи после сохранения результата;
- Сообщения об ошибках. Требуется автоматизация разнесения некорректных данных по соответствующим полям и тестирование корректности проверки правильности данных и сообщений об ошибках;

- Комплексная проверка поведения всей системы, как целостного объекта (end-to-end testing);
- Проверка числовых массивов, нужных для достоверных математических операций;
- Тестирование корректности отображаемых результатов поиска в ответ на запрос по нужным данным;
- Предложенный список только ориентировочный. Всё зависит от предъявляемых к проверяемой системе требований, возможностей, которые позволяет реализовать выбранный для автоматического тестирования инструмент.

Источники:

- Какие места в проекте нужно автоматизировать в первую очередь?

Доп. материал:

- Автоматизировать или нет: спорные кейсы, плюсы и минусы автотестов
- How To Select Correct Test Cases For Automation Testing (And Ultimately Achieve A Positive Automation ROI)
- How To Implement Efficient Test Automation In The Agile World
- Right Tests for Automation
- Решаем, что и когда автоматизировать, и нужно ли
- Не автоматизируйте test cases
- Автоматизация тестирования: что можно, а что не нужно
- Лучшие практики автоматизации тестирования: решение, что и когда автоматизировать

Виды и инструменты автоматизации

Одним из самых частых вопросов новичков в автоматизации является вопросы о том, какой язык программирования выбрать для изучения, а также какой инструмент лучше. Эти вопросы на самом деле стоит рассматривать в другой плоскости, а язык программирования является следствием, а не причиной (а где-то и вообще не нужен).

При всем многообразии того, что можно считать автоматизацией, можно попытаться выделить категории:

- Функциональное или нефункциональное тестирование;
- Уровни: unit, integration, UI/E2E;
- Метод: white box, black box;
- Уровень в сетевой модели: frontend, backend, database;
- Платформы: desktop, mobile, web, cross platform;
- ОС: windows, macos, linux, cross platform;
- Масштабируемость: однопоточные, многопоточные;
- Программа, скрипт, фреймворк или библиотека;
- Работа непосредственно с тестами, как вспомогательное средство (генераторы данных), инфраструктура (CI/CD, раннеры, отчетность);
- Если это создание тестов, то каким образом: нативный язык программирования, универсальный ЯП, тестовая модель и ключевые слова, рекордеры (Capture & Playback);
- и т.д. и т.п. до бесконечности.

Каждый инструмент создан и подходит для определенных задач и сочетает в себе несколько таких критериев. Именно поэтому нет лучшего инструмента и единственно правильного языка программирования для старта.

Технологии автоматизации тестирования

Начнём с краткого обзора эволюции высокоуровневых технологий, при этом подчеркнув, что «старые» решения по-прежнему используются (или как компоненты «новых», или самостоятельно в отдельных случаях).

	Подход	Суть	Преимущества	Недостатки
1	Частные решения	Для решения каждой отдельной задачи пишется отдельная программа	Быстро, просто	Нет системности, много времени уходит на поддержку. Почти невозможно повторное использование
2	Тестирование под управлением данными (DDT)	Из тест-кейса вовне выносятся входные данные и ожидаемые результаты.	Один и тот же тест кейс можно повторять многократно с разными данными	Логика тест-кейса по прежнему строго определяется внутри, а потому для ее изменения тест кейс надо переписывать
3	Тестирование под управлением ключевыми словами (KDT)	Из тест-кейса вовне выносится описание его поведения	Концентрация на высокоуровневых действиях. Данные и особенности поведения хранятся вовне и могут быть изменены без изменения кода тест-кейса	Сложность выполнения низкоуровневых операций
4	Использование фреймворков	Конструктор, позволяющий использовать остальные подходы	Мощность и гибкость	Относительная сложность (особенно - в создании фреймворка)
5	Запись и воспроизведение (Record & Playback)	Средство автоматизации записывает действия тестировщика и может воспроизвести их, управляя тестируемым приложением	Простота, высокая скорость создания тест-кейсов.	Крайне низкое качество, линейность, неподдерживаемость тест-кейсов. Требуется серьёзная доработка полученного кода
6	Тестирование под управлением поведением (BDT)	Развитие идей тестирования под управлением данными и ключевыми словами. Отличие - в концентрации на бизнес сценариях без выполнения мелких проверок	Высокое удобство проверки высокоуровневых пользовательских сценариев	Такие тест-кейсы пропускают большое количество функциональных и нефункциональных дефектов, а потому должны быть дополнены классическими более низкоуровневыми тест-кейсами

На текущем этапе развития тестирования представленные в таблице технологии иерархически можно изобразить следующей схемой:

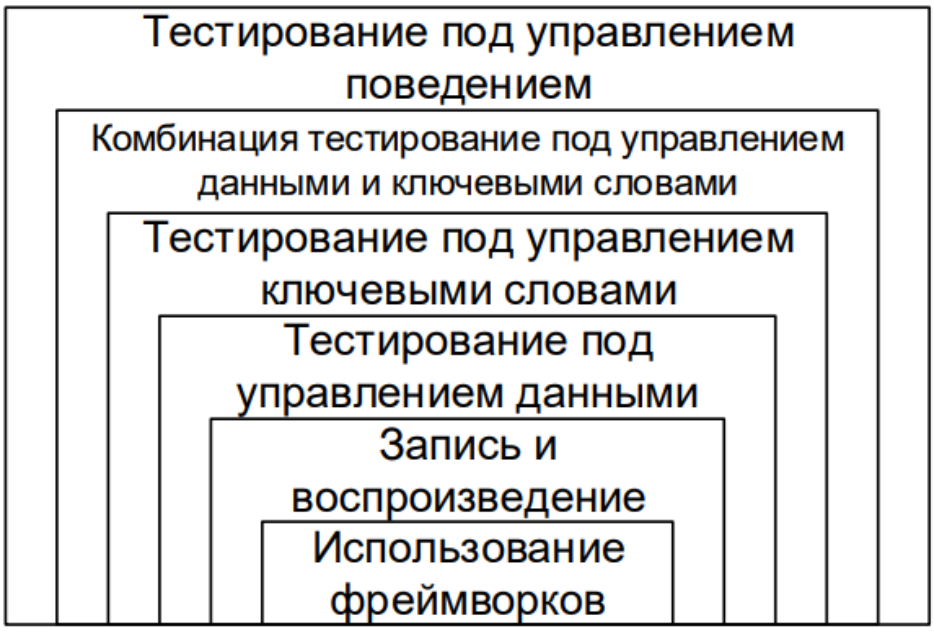


Рисунок 3.2.b — Иерархия технологий автоматизации тестирования

Технологии: Частные решения

Иногда перед тестировщиком возникает уникальная (в том плане, что такой больше не будет) задача, для решения которой нет необходимости использовать мощные инструментальные средства, а достаточно написать небольшую программу на любом из высокоуровневых языков программирования (Java, C#, PHP и т.д.) или даже воспользоваться возможностями командных файлов операционной системы или подобными тривиальными решениями. Также сюда можно отнести задачи вида:

- Подготовить базу данных, наполнив её тестовыми данными (например, добавить в систему миллион пользователей со случайными именами).
- Подготовить файловую систему (например, создать структуру каталогов и набор файлов для выполнения тест-кейсов).
- Перезапустить набор серверов и/или привести их в требуемое состояние.

Удобство частных решений состоит в том, что их можно реализовать быстро, просто, «вот прямо сейчас». Но у них есть и огромный недостаток - это «кустарные решения», которыми может воспользоваться всего пара человек. И при появлении новой задачи, даже очень похожей на ранее решенную, скорее всего, придётся всё автоматизировать заново.

Преимущества:

- Быстрота и простота реализации;
- Возможность использования любых доступных инструментов, которыми тестировщик умеет пользоваться;
- Эффект от использования наступает незамедлительно;
- Возможность нахождения очень эффективных решений в случае, когда основные инструменты, используемые на проекте для автоматизации тестирования, оказываются малопригодными для выполнения данной отдельной задачи;
- Возможность быстрого создания и оценки прототипов перед применением более тяжеловесных решений

Недостатки:

- Отсутствие универсальности и, как следствие, невозможность или крайняя сложность повторного использования (адаптации для решения других задач).
- Разрозненность и несогласованность решений между собой (разные подходы, технологии, инструменты, принципы решения).
- Крайне высокая сложность развития, поддержки и сопровождения таких решений (чаще всего, кроме самого автора никто вообще не понимает, что и зачем было сделано, и как оно работает).
- Является признаком «кустарного производства», не приветствуется в промышленной разработке программ.

Технологии:Тестирование под управлением данными (DDT)

Обратите внимание, как много раз в командных файлах повторяются строки, выполняющие одно и то же действие над набором файлов (и нам ещё повезло, что файлов немного). Ведь куда логичнее было бы каким-то образом подготовить список файлов и просто передать его на обработку. Это и будет тестированием под управлением данными. Теперь нам достаточно подготовить CSV-файл с любым количеством имён сравниваемых файлов, а код тест-кейса не увеличится.

К другим типичным примерам использования тестирования под управлением данными относится:

- Проверка авторизации и прав доступа на большом наборе имен пользователей и паролей;
- Многократное заполнение полей форм разными данными и проверка реакции приложения;
- Выполнение тест-кейса на основе данных, полученных с помощью комбинаторных техник.

Данные для рассматриваемого подхода к организации тест-кейсов могут поступать из файлов, баз данных и иных внешних источников или даже генерироваться в процессе выполнения тест-кейса (см. описание источников данных для автоматизированного тестирования).

Преимущества:

- Устранение избыточности кода тест-кейсов;
- Удобное хранение и понятный человеку формат данных;
- Возможность поручения генерации данных сотруднику, не имеющему навыков программирования;
- Возможность использования одного и того же набора данных для выполнения разных тест-кейсов;
- Возможность повторного использования набора данных для решения новых задач;
- Возможность использования одного и того же набора данных в одном и том же тест кейсе, но реализованном под разными платформами.

Недостатки:

- При изменении логики поведения тест-кейса его код всё равно придётся переписывать;
- При неудачно выбранном формате представления данных резко снижается их понятность для неподготовленного специалиста;
- Необходимость использования технологий генерации данных;
- Высокая сложность кода тест-кейса в случае сложных неоднородных данных;
- Риск неверной работы тест-кейсов в случае, когда несколько тест-кейсов работают с одним и тем же набором данных, и он был изменён таким образом, на который не были рассчитаны некоторые тест-кейсы;
- Слабые возможности по сбору данных в случае обнаружения дефектов;
- Качество тест-кейса зависит от профессионализма сотрудника, реализующего код тест кейса.

Технологии: Тестирование под управлением ключевыми словами

Логическим развитием идеи о вынесении вовне тест-кейса данных является вынесение вовне тест-кейса команд (описания действий). Идею можно развить, добавив в CSV-файл ключевые слова, являющиеся описанием выполняемой проверки. Ярчайшим примером инструментального средства автоматизации

тестирования, идеально следующего идеологии тестирования под управлением ключевыми словами, является Selenium IDE, в котором каждая операция тест-кейса описывается в виде: Действие (ключевое слово) - Необязательный параметр 1 - Необязательный параметр 2.

Тестирование под управлением ключевыми словами стало тем переломным моментом, начиная с которого стало возможным привлечение к автоматизации тестирования нетехнических специалистов. Согласитесь, что нет необходимости в знаниях программирования и тому подобных технологий, чтобы наполнять подобные только что показанному CSV-файлы или (что очень часто практикуется) XLSX файлы.

Вторым естественным преимуществом тестирования под управлением ключевыми словами (хотя она вполне характерна и для тестирования под управлением данными) стала возможность использования различных инструментов одними и теми же наборами команд и данных. Так, например, ничто не мешает нам взять показанные CSV-файлы и написать новую логику их обработки не на PHP, а на C#, Java, Python или даже с использованием специализированных средств автоматизации тестирования.

Преимущества:

- Максимальное устранение избыточности кода тест-кейсов;
- Возможность построения мини-фреймворков, решающих широкий спектр задач;
- Повышение уровня абстракции тест-кейсов и возможность их адаптации для работы с разными техническими решениями;
- Удобное хранение и понятный человеку формат данных и команд тест-кейса;
- Возможность поручения описания логики тест-кейса сотруднику, не имеющему навыков программирования;
- Возможность повторного использования для решения новых задач;
- Расширяемость (возможность добавления нового поведения тест-кейса на основе уже реализованного поведения)

Недостатки:

- Высокая сложность (и, возможно, длительность) разработки;
- Высокая вероятность наличия ошибок в коде тест-кейса;
- Высокая сложность (или невозможность) выполнения низкоуровневых операций, если фреймворк не поддерживает соответствующие команды;
- Эффект от использования данного подхода наступает далеко не сразу (сначала идет длительный период разработки и отладки самих тест-кейсов и вспомогательной функциональности);
- Для реализации данного подхода требуется наличие высококвалифицированного персонала;
- Необходимо обучить персонал языку ключевых слов, используемых в тест-кейсах

Технологии: Фреймворки

Фреймворк автоматизации тестирования - это интегрированная система, которая устанавливает правила автоматизации конкретного продукта. Эта система объединяет библиотеки функций, источники тестовых данных, сведения об объектах и различные повторно используемые модули. Эти компоненты действуют как небольшие строительные блоки, которые необходимо собрать для представления бизнес-процесса. Платформа обеспечивает основу для автоматизации тестирования и упрощает процесс автоматизации.

Фреймворки автоматизации тестирования представляют собой не что иное, как успешно развившиеся и ставшие популярными решения, объединяющие в себе лучшие стороны тестирования под управлением данными, ключевыми словами и возможности реализации частных решений на высоком уровне абстракции.

Фреймворков автоматизации тестирования очень много, они очень разные, но их объединяет несколько общих черт:

- высокая абстракция кода (нет необходимости описывать каждое элементарное действие) с сохранением возможности выполнения низкоуровневых действий;
- универсальность и переносимость используемых подходов;
- достаточно высокое качество реализации (для популярных фреймворков).

Как правило, каждый фреймворк специализируется на своём виде тестирования, уровне тестирования, наборе технологий. Существуют фреймворки для модульного тестирования (например, семейство xUnit), тестирования веб-приложений (например, семейство Selenium), тестирования мобильных приложений, тестирования производительности и т.д.

Существуют бесплатные и платные фреймворки, оформленные в виде библиотек на некотором языке программирования или в виде привычных приложений с графическим интерфейсом, узко- и широко специализированные и т.д.

Преимущества:

- Широкое распространение;
- Универсальность в рамках своего набора технологий;
- Хорошая документация и большое сообщество специалистов, с которыми можно проконсультироваться;
- Высокий уровень абстракции;
- Наличие большого набора готовых решений и описаний соответствующих лучших практик применения того или иного фреймворка для решения тех или иных задач.

Недостатки:

- Требуется время на изучение фреймворка;
- В случае написания собственного фреймворка де-факто получается новый проект по разработке ПО;
- Высокая сложность перехода на другой фреймворк;
- В случае прекращения поддержки фреймворка тест-кейсы рано или поздно придётся переписывать с использованием нового фреймворка;
- Высокий риск выбора неподходящего фреймворка.

Технологии: Запись и воспроизведение (Record & Playback)

Технология записи и воспроизведения (Record & Playback) стала актуальной с появлением достаточно сложных средств автоматизации, обеспечивающих глубокое взаимодействие с тестируемым приложением и операционной системой. Использование этой технологии, как правило, сводится к следующим основным шагам:

- Тестировщик вручную выполняет тест-кейс, а средство автоматизации записывает все его действия;
- Результаты записи представляются в виде кода на высокоуровневом языке программирования (в некоторых средствах - специально разработанном);
- Тестировщик редактирует полученный код;
- Готовый код автоматизированного тест-кейса выполняется для проведения тестирования в автоматизированном режиме.

Сама технология при достаточно высокой сложности внутренней реализации очень проста в использовании и по самой своей сути, потому часто применяется для обучения начинающих специалистов по автоматизации тестирования.

Преимущества:

- Предельная простота освоения (достаточно буквально нескольких минут, чтобы начать
- использовать данную технологию);

- Быстрое создание «скелета тест-кейса» за счет записи ключевых действий с тестируемым приложением;
- Автоматический сбор технических данных о тестируемом приложении (идентификаторов и локаторов элементов, надписей, имен и т.д.);
- Автоматизация рутинных действий (заполнения полей, нажатий на ссылки, кнопки и т.д.);
- В отдельных случаях допускает использование тестировщиками без навыков программирования.

Недостатки:

- Линейность тест-кейсов: в записи не будет циклов, условий, вызовов функций и прочих характерных для программирования и автоматизации явлений;
- Запись лишних действий (как просто ошибочных случайных действий тестировщика с тестируемым приложением, так и (во многих случаях) переключений на другие приложения и работы с ними);
- Так называемый «хардкодинг», т.е. запись внутрь кода тест-кейса конкретных значений, что потребует ручной доработки для перевода тест-кейса на технологию тестирования под управлением данными;
- Неудобные имена переменных, неудобное оформление кода тест-кейса, отсутствие комментариев и прочие недостатки, усложняющие поддержку и сопровождение тест кейса в будущем;
- Низкая надёжность самих тест-кейсов в силу отсутствия обработки исключений, проверки условий и т.д

Технологии: Тестирование под управлением поведением

Рассмотренные выше технологии автоматизации максимально сфокусированы на технических аспектах поведения приложения и обладают общим недостатком: с их помощью сложно проверять высокоуровневые пользовательские сценарии (а именно в них и заинтересованы заказчики и конечные пользователи). Этот недостаток призвано исправить тестирование под управлением поведением, в котором акцент делается не на отдельных технических деталях, а на общей работоспособности приложения при решении типичных пользовательских задач.

Такой подход не только упрощает выполнение целого класса проверок, но и облегчает взаимодействие между разработчиками, тестировщиками, бизнес-аналитиками и заказчиком, т.к. в основе подхода лежит очень простая формула «given-when-then»:

- Given («имея, предполагая, при условии») описывает начальную ситуацию, в которой находится пользователь в контексте работы с приложением;
- When («когда») описывает набор действий пользователя в данной ситуации;
- Then («тогда») описывает ожидаемое поведение приложения.

Такой принцип описания проверок позволяет даже участникам проекта, не имеющим глубокой технической подготовки, принимать участие в разработке и анализе тест-кейсов, а для специалистов по автоматизации упрощается создание кода автоматизированных тест-кейсов, т.к. такая форма является стандартной, единой и при этом предоставляет достаточно информации для написания высокоуровневых тест-кейсов. Существуют специальные технические решения (например, Behat, JBehave, NBehave, Cucumber), упрощающие реализацию тестирования под управлением поведением.

Преимущества:

- Фокусировка на потребностях конечных пользователей;
- Упрощение сотрудничества между различными специалистами;
- Простота и скорость создания и анализа тест-кейсов (что, в свою очередь, повышает полезный эффект автоматизации и снижает накладные расходы).

Недостатки:

- Высокоуровневые поведенческие тест-кейсы пропускают много деталей, а потому могут не обнаружить часть проблем в приложении или не предоставить необходимой для понимания обнаруженной проблемы информации;
- В некоторых случаях информации, предоставленной в поведенческом тест-кейсе, недостаточно для его непосредственной автоматизации

К классическим технологиям автоматизации тестирования также можно отнести разработку под управлением тестированием (Test-driven Development, TDD) с её принципом «красный, зелёный, улучшенный» (Red-Green-Refactor), разработку под управлением поведением (Behavior-driven Development), модульное тестирование (Unit Testing) и т.д. Но эти технологии уже находятся на границе тестирования и разработки приложений, потому выходят за рамки данной темы.

Автоматизация вне прямых задач тестирования

На протяжении данного раздела мы рассматривали, как автоматизация может помочь в создании и выполнении тест-кейсов. Но все те же принципы можно перенести и на остальную работу тестировщика, в которой также бывают длительные и утомительные задачи, рутинные задачи или задачи, требующие предельного внимания, но не связанные с интеллектуальной работой. Всё перечисленное также можно автоматизировать. Да, это требует технических знаний и первоначальных затрат сил и времени на реализацию, но в перспективе такой подход может экономить до нескольких часов в день. К самым типичным решениям из данной области можно отнести:

- Использование командных файлов для выполнения последовательностей операций - от копирования нескольких файлов из разных каталогов до развертывания тестового окружения. Даже в рамках многократно рассмотренных примеров по тестированию «Конвертера файлов» запуск приложения через командный файл, в котором указаны все необходимые параметры, избавляет от необходимости вводить их каждый раз вручную;
- Генерация и оформление данных с использованием возможностей офисных приложений, баз данных, небольших программ на высокоуровневых языках программирования. Нет картины печальнее, чем тестировщик, руками нумерующий три сотни строк в таблице;
- Подготовка и оформление технических разделов для отчётов. Можно тратить часы на скрупулезное вычитывание журналов работы некоего средства автоматизации, а можно один раз написать скрипт, который будет за мгновение готовить документ с аккуратными таблицами и графиками, и останется только запускать этот скрипт и прикреплять результаты его работы к отчёту;
- Управление своим рабочим местом: создание и проверка резервных копий, установка обновлений, очистка дисков от устаревших данных и т.д. и т.п. Компьютер всё это может (и должен!) делать сам, без участия человека;
- Сортировка и обработка почты. Даже раскладывание входящей корреспонденции по подпапкам гарантированно занимает у вас несколько минут в день. Если предположить, что настройка специальных правил в вашем почтовом клиенте сэкономит вам полчаса в неделю, за год экономия составит примерно сутки;
- Виртуализация как способ избавления от необходимости каждый раз устанавливать и настраивать необходимый набор программ. Если у вас есть несколько заранее подготовленных виртуальных машин, их запуск займёт секунды. А в случае необходимости устранения сбоев разворачивание виртуальной машины из резервной копии заменяет весь процесс установки и настройки с нуля операционной системы и всего необходимого программного обеспечения

Классификация инструментов автотестирования мобильных приложений

Драйвер. Утилиты автотестирования, как и другие программы, могут взаимодействовать с приложением только через программный интерфейс - по-другому они не умеют. Для работы через другие интерфейсы существуют специальные программы - драйверы. Драйвер - программа, которая предоставляет API для одного из интерфейсов приложения. Для каждого интерфейса, кроме, собственно, API, необходим свой

драйвер. Например, когда вы даёте драйверу для GUI команду “Нажать на кнопку Menu”, он воспринимает её через API и отсылает в тестируемое приложение, где эта команда превращается в клик по графической кнопке Menu. Для взаимодействия с API приложения драйверы не нужны или почти не нужны - взаимодействие программное. А вот при работе с остальными интерфейсами без них не обойтись. Наиболее сложными обычно являются драйверы для GUI, так как этот интерфейс сильно отличается от обычного для программы общения кодом. При этом в автоматизированном тестировании мобильных приложений GUI наиболее актуален, так как в интеграционном тестировании использовать чаще всего приходится именно его. Наиболее популярные драйвера для GUI в мобильном тестировании - UIAutomator и Espresso для Android, XCUITest - для iOS.

Надстройка. Когда функционала драйвера не хватает или он неудобен и сложен, над ним появляется еще один уровень, который я буду называть надстройкой.Надстройка - программа, которая взаимодействует с приложением через один или несколько драйверов, повышая удобство их использования или расширяя их возможности.

У надстройки могут быть следующие функции:

- Модификация поведения (без изменения API). Например:
    - дополнительное протоколирование,
    - валидация данных,
    - ожидание выполнения действия в течение определенного времени.
- Повышение удобства и/ или уровня абстракции API через:
    - использование синтаксического сахара - удобных названий функций, более коротких обращений к ним, унифицированного стиля написания тестов;
    - неявное управление драйвером, когда, например, он инициализируется автоматически, без необходимости прописывать каждое такое действие вручную;
    - упрощение сложных команд вроде выбора события из календаря или работы с прокручивающимися списками;
    - реализацию альтернативных стилей программирования, таких как процедурный стиль или fluent.
- Унификация API драйверов. Здесь надстройка предоставляет единый интерфейс для работы сразу с несколькими драйверами. Это позволяет, например, использовать один и тот же код для тестов на iOS и Android, как в популярной надстройке Appium.

Фреймворк. С другой стороны тестов находится фреймворк запуска. В рамках данной статьи я буду коротко называть его “фреймворк”. Фреймворк - это программа для формирования, запуска и сбора результатов запуска набора тестов.

В задачи фреймворка входят:

- формирование, группировка и упорядочение набора тестов,
- распараллеливание набора (опционально),
- создание фикстур,
- запуск тестов,
- сбор результатов их выполнения,
- формирование отчётов о выполнении (опционально).

Можно заметить, что эти функции не связаны с тестированием только мобильных приложений - их можно успешно применять и в тестировании десктоп- и веб-приложений. Дело в том, что фреймворк не должен обеспечивать взаимодействие тестов и приложения - он работает только с тестами, и тип приложения не имеет значения. Если драйверы и надстройки находятся между тестами и приложением, то фреймворк находится над тестами, организуя их запуск. Поэтому важно не путать понятия “драйвер” и “фреймворк”.

Конечно, в некоторых фреймворках есть собственные драйверы для работы с приложениями, но это вовсе не обязательное условие. Самые заметные фреймворки в мобильном тестировании - xUnit и Cucumber.

Комбайны. Наконец, еще одна группа утилит, использующихся для автоматизации тестирования мобильных приложений, - это комбайны, объединяющие в себе и фреймворки, и драйверы (причём не только мобильные), и даже возможности разработки. Xamarin.UITest, Squish, Ranorex - все они поддерживают автоматизацию тестирования iOS-, Android-, веб-приложений, а два последних - ещё и десктоп-приложений.

Android: Инструменты для автоматизации тестирования

Тесты под андроид бывают локальные и инструментальные:

- Instrumented tests выполняются на устройстве Android, физическом или эмулированном. Тест собирается в APK и устанавливается вместе с тестовым приложением. Instrumented tests обычно представляют собой тесты пользовательского интерфейса, запуск приложения и последующее взаимодействие с ним;
- Local tests выполняются на вашей машине разработки или сервере, поэтому их также называют тестами на стороне хоста (host-side tests). Для запуска тестов используется виртуальная машина Java (JVM). Обычно они маленькие и быстрые, изолируя тестируемый объект от остальной части приложения.

Не все модульные тесты являются локальными, и не все E2E тесты выполняются на устройстве. Например:

- Большой локальный тест: вы можете использовать симулятор Android, который работает локально, например Robolectric;
- Небольшой инструментальный тест: вы можете убедиться, что ваш код хорошо работает с функцией платформы, такой как база данных SQLite. Вы можете запустить этот тест на нескольких устройствах, чтобы проверить интеграцию с несколькими версиями SQLite.

UI тесты обращаются к GUI-драйверам. GUI-драйверы являются наиболее сложным компонентом всего стека тестирования. Они решают базовые, низкоуровневые задачи. Когда вы даете GUI-драйверу (Espresso, UiAutomator, Robotium, Selendroid) команду «кликнуть на кнопку», он обрабатывает ее, взаимодействует с приложением, и эта команда превращается в клик по элементу.

Все драйверы работают на Android Instrumentation Framework - базовом API Android для взаимодействия с системой. Самые популярные - Espresso и UiAutomator. Они оба разрабатываются и поддерживаются компанией Google. Их без труда можно использовать одновременно в пределах одного теста.

UiAutomator поставляется вместе с Android SDK начиная с 16 версии API. Как GUI-драйвер он служит для поиска элементов интерфейса на экране девайса и эмуляции различных действий: кликов, тачей, свайпов, ввода текста, проверок на видимость. Рекордер для записи тестов он не предоставляет, зато предоставляет утилиту для просмотра древовидной структуры экрана - Ui Automator Viewer.

UiAutomator позволяет писать тесты по модели черного ящика (black-box). Он живет в собственном процессе и работает без доступа к исходному коду приложения. Значит, тест с его использованием может взаимодействовать практически с любыми приложениями, в том числе системными. Это открывает доступ к множеству функций, например, становятся возможны:

- набор номера и совершение звонка;
- отправка и чтение смс;
- взаимодействие с нотификациями;
- управление настройками сети, геолокации;
- снятие скриншотов.

Наиболее подходящим сценарием для использования UiAutomator является не работа с тестируемым продуктовым приложением, а взаимодействие со сторонними или системными приложениями. Более подходящим инструментом для работы с продуктовым приложением является **Espresso**.

Это также официальный фреймворк для UI-тестирования от Google, но тесты с его использованием работают уже по модели белого ящика (white-box). Espresso поддерживает Android API с 10 версии, хотя появился значительно позже. Предоставляет рекордер для записи тестов. Фактически Espresso является лидером и даже стандартом в индустрии. Такой успех может быть связан с тем, что он обеспечивает повышенную скорость и стабильность тестов в сравнении с конкурентами. Espresso решает низкоуровневую задачу - найти необходимый элемент на экране по заданным параметрам (ViewMatcher), произвести с ним действия (ViewAction) или выполнить проверки (ViewAssertion). Синтаксис Espresso реализован на основе фреймворка Hamcrest. Он построен на использовании иерархий вложенных матчеров - сущностей, описывающих требования к объекту, в случае Espresso - к элементам интерфейса. Они используются при задании параметров поиска элемента на экране, а также в проверках как описание свойств, которыми элемент должен обладать. Вложенность матчеров часто приводит к тому, что код тестов становится трудно читать и поддерживать. Стабильность и скорость тестов на Espresso обусловлены его внутренним устройством - все команды Espresso выполняются в том же процессе, в котором работает само тестируемое приложение. Действия и проверки преобразовываются и кладутся в очередь сообщений главного потока приложения. Выполняются они только когда приложение готово к этому, то есть его главный поток находится в состоянии ожидания пользовательского ввода (idle).

Итак, с драйверами и устройством наиболее популярных из них мы разобрались. Мы поняли, что все они решают низкоуровневые задачи: поиск элемента на экране и выполнение с ним какого-либо действия. Из-за этого они предоставляют невыразительный API, которым неудобно пользоваться для решения более высокоуровневых проблем. Например, ни один из них не предоставляет собственный механизм для реализации повторов неудачных действий или логирования. Более того, часто существует необходимость работать в тестах сразу с несколькими драйверами, например, с Espresso и UiAutomator.

На помощь приходят **обертки** (надстройки). Именно к API оберток мы будем обращаться в наших тестах, и именно обертки предоставляют конечный арсенал приемов для наших тестов.

- **Kakao** - простой и удобный Kotlin DSL для Espresso. Он позволяет упростить код тестов на Espresso и повысить его читаемость;
- **Barista** - это объемная библиотека, содержащая большое количество полезных решений и приемов при работе с Espresso;
- **Kaspresso** - это фреймворк-обертка для UI-тестирования, который претендует на то, чтобы избавить мир от зла и стать практически единственной зависимостью в вашем проекте для работы с тестами. Kaspresso расширяет лаконичный Kakao DSL - он предоставляет собственный Kotlin DSL для описания тестовых секций и шагов, внутри которых продолжает жить Kakao.

Подробнее в источнике по первой ссылке далее.

- На чем писать Android UI-тесты
- Автотесты на Android. Картина целиком
- An open-source documentation about Android UI testing
- Android Developers - Test apps on Android
- Kaspresso: фреймворк для автотестирования, который вы ждали
- Kaspresso tutorials. Часть 1. Запуск первого теста
- Adb-server в Kaspresso
- Светлана Смельчакова - UI Automator deep diving
- Selendroid Tutorial: Android Mobile Test Automation Framework (Part 1)

iOS: Инструменты для автоматизации тестирования

Под iOS существуют следующие инструменты:

- TestProject - фреймворк для iOS тестирования на базе Appium и Selenium. Подходит для тестирования веба, Android-приложений и API. При этом не обязательно использовать XCode;
- EarlGrey - open-source инструмент, разработанный компанией Google для тестирования своих собственных iOS-приложений, таких как YouTube, Google Calendar и т.д.;
- XCTest - инструмент, созданный Apple для iOS тестирования. Он полностью совместим с XCode. XCTest универсален - с помощью него можно проводить как unit-тесты, так и тестировать производительность и пользовательский интерфейс;
- Detox - это инструмент end-to-end тестирования, который используется для тестирования методом серого ящика;
- OCMock - проект с открытым исходным кодом, который упрощает iOS-тестирование с помощью mock-объектов. OCMock расшифровывается как Objective-C Mock. С его помощью тестировщик может создавать mock-объекты, используя язык Objective-C;
- KIF - это фреймворк для интеграционного тестирования iOS-приложений. KIF расшифровывается как “keep it functional” и используется для тестирования пользовательского интерфейса, как и XCTest.

Подробнее в источнике по первой ссылке далее.

- Обзор фреймворков для iOS тестирования
- Погружение в автотестирование на iOS. Часть 1. Как работать с accessibilityidentifier объектов
- Погружение в автотестирование на iOS. Часть 2. Как взаимодействовать с ui-элементами iOS приложения в тестах

Кроссплатформенные обертки

- Appium - это довольно популярный кросс-платформенный open source инструмент для автоматизации тестирования десктоп и мобильных приложений под Android и iOS. Архитектура Appium схожа с Selenium WebDriver, широко распространенным и ставшим стандартом в web-тестировании. Кроссплатформенность достигается за счет использования разных драйверов для разных платформ. Именно драйверы транслируют клиентский Appium-код в команды, непосредственно исполняемые на устройствах. Для написания тестов Appium предоставляет клиентские библиотеки с фиксированным API;
- Calabash пользуется большой популярностью среди разработчиков благодаря стабильности и подходу к построению тестов. Calabash использует Cucumber, благодаря чему его могут использовать даже люди, незнакомые с программированием. Поддерживает много языков программирования, может работать как с iOS, так и с Android приложениями.

Доп. материал:

- Appium Studio Tutorial For Mobile Automation (15+ Hands-On Tutorials)
- Appium Tutorial For Testing Android And IOS Mobile Apps
- Автоматизация мобильных приложений на базе Appium

WEB: Инструменты для автоматизации тестирования

- Selenium: Старейший фреймворк, все еще самый популярный. Поддерживает Java, Python, C#, Ruby, JavaScript. Эмулирует почти все возможные действия пользователя;
- Cucumber: Фреймворк ориентирован на behavior-driven development (BDD) - “разработка через поведение”; удобный как для разработчиков, так и QA. Главное преимущество: простота;
- Cypress: И наконец Cypress, делающий жизнь тестировщиков проще. В нем нет некоторых проблем, существующих в Selenium, потому что он имеет другую архитектуру - построен на основе JS-фреймворка Mocha. Преимущества: простота и удобство асинхронных тестов. Применяется BDD/TDD-библиотека (TDD = test-driven development);

- Playwright enables reliable end-to-end testing for modern web apps;
- Puppeteer is a Node library which provides a high-level API to control Chrome or Chromium over the DevTools Protocol

Подробнее в источнике по первой ссылке далее.

- E2E-тестирование и Cypress
- Александр Самойлов, Сергей Львов - Визуальные проверки в End-to-End-тестах
- QA Wolf handles your end-to-end testing
- Selenium vs Puppeteer vs Cypress vs Playwright
- 30+ Best Selenium Tutorials: Learn Selenium With Real Examples
- Что такое Cypress: Введение и архитектура

API: Инструменты для автоматизации тестирования

По постману и соап есть множество ссылок в инструментах для мануальщика.

- Что такое REST Assured? - REST Assured API Testing - 18+
- Тестирование API с помощью REST Assured (Java)
- Cucumber и Spock для автоматизации API-тестов. В чем польза этих фреймворков
- Cypress API

Unit

- List of unit testing frameworks
- Перевод книги Python Testing with pytest

Инструменты по performance/security и всему не вошедшему смотри в разделе полезных ссылок по мануальному тестированию.

Инструменты, касающиеся инфраструктуры CI/CD вынесены в следующую тему.

Доп. материал:

- Автоматизация UI-тестирования в приложении Недвижимости на Android. Доклад Яндекса
- Как мы автоматизировали тестирование бэкенда
- Browser Automation Testing For Start-Ups And Small-Sized Teams
- Доказательная разработка или как data-driven подход добавил смысла работе
- Why Do We Need Framework For Test Automation?

По инструментам:

- Путеводитель по инструментам автотестирования мобильных приложений
- Top 20 Best Automation Testing Tools In 2022 (Comprehensive List)
- 100+ лучших инструментов для тестирования программного обеспечения
- How To Choose The Best Automation Testing Tool (A Complete Guide)
- How To Develop Test Scripts Using Top 5 Most Popular Test Automation Frameworks (Examples)
- Record-and-Replay тестирование - сочетание достоинств юнит и интеграционных тестов
- LEADERSHIP IN TEST: TEST EXECUTION TOOLS
- Фреймворки для тестирования: личный опыт и новые методы
- 10 LATEST SOFTWARE TESTING TOOLS QAS ARE USING IN 2022
- Применение автотестов в ежедневных релизах
- Разработка и сценарное тестирование с Vanessa-ADD. Концепция, теория и сквозной пример создания сценария
- THE 10 BEST QA AUTOMATION TOOLS FOR SOFTWARE TESTING IN 2022
- Выбираем правильные инструменты автоматизации (с таблицей)

- Борьба с задержкой тестов в Selenium: пример из практики
- Все про Mobile QA Automation - Appium, XCUITest, Espresso
- Дмитрий Буздин - Как построить свой фреймворк для автотестов?
- Одновременное использование нескольких тест-фреймворков
- Почему я больше не использую Cucumber при автоматизации тестирования
- Selenoid UI
- 50 приложений для эффективной организации удалённой работы
- Selenium, Selenoid, Selenide, Selendroid… Что все это значит?
- 6 антипаттернов для UI тестирования на JavaScript, которых следует избегать
- Эмуляторы и симуляторы vs реальные устройства для автоматизации тестирования
- FlaNium: как сделать тестирование Desktop-приложений под Windows проще
- Black box API testing with server logs
- Veslo - расширение Retrofit для тестирования (Java)
- The Essential Guide to Automated Visual Regression Testing
- QTP Tutorials - 25+ Micro Focus Quick Test Professional (QTP) Training Tutorials
- 15+ SoapUI Tutorials: The Best Web Services API Testing Tool
- LoadRunner Tutorial For Beginners (Free 8-Day In-Depth Course)
- Most Popular Test Automation Frameworks With Pros And Cons Of Each - Selenium Tutorial #20
- Introduction To Sikuli GUI Automation Tool (Automate Anything You See On Screen) - Sikuli Tutorial #1
- PowerShell UIAutomation Tutorial: UI Automation Of Desktop Applications
- Katalon Automation Recorder (Selenium IDE Alternative): Hands-On Review Tutorial
- Geb Tutorial - Browser Automation Testing Using Geb Tool
- AutoIt Tutorial - AutoIt Download, Install & Basic AutoIt Script
- Automation Testing Using Cucumber Tool And Selenium - Selenium Tutorial #30
- Protractor Testing Tool For End-To-End Testing Of AngularJS Applications
- Ranorex Tutorial: A Powerful Desktop, Web, And Mobile Automation Testing Tool
- Осваиваем Data-driven Testing в Selenium
- Борьба с задержкой тестов в Selenium: пример из практики
- Доклад: Как достичь Pixel Perfect с Jetpack Compose и Figma / Владимир Иванов (Tinkoff)
- Ускоряем свои тесты с помощью cypress-grep
- QAGuild#53: Тестирование api на python и schemathesis
- Selenide vs Selenium - подробное сравнение
- Grafana и автотесты: учимся измерять работу тестов
- Playwright: веб-тестирование без драмы
- Рецепты применения техник тест-дизайна с помощью многофункц-ного Citrus Testing Framework часть 1, 2
- Создание фреймворка для автоматических тестов: пошаговая инструкция
- Как в Яндексе используют PyTest и другие фреймворки для функционального тестирования
- Единственно верный способ загружать и скачивать файлы в Selenium тестах
- Пишем автотест с использованием Selenium Webdriver, Java 8 и паттерна Page Object
- Простой и удобный шаблон тестового фреймворка на selenide для UI автотестов
- Тесты в Python: все основные подходы, плюсы и минусы. Доклад Яндекса
- Home видео для Selenium aka WebDriver. Или чем записать экран, если у вас есть java, поломанные тесты и немного времени
- Эффективное тестирование верстки
- Автоматизация тестирования мобильных приложений. Часть 2: предусловия, верификация элементов и независимость шагов

Инфраструктура и пайплайн (CI/CD)

В этой теме у меня мало опыта, вероятность белиберды увеличена втрое.

Сейчас компетентность в сфере TestOps является таким же базовым требованием к QA-инженерам, как и написание автоматизированных тестов. Причина - в активном развитии CI/CD в проектах и необходимости QA-инженерам работать с пайплайнами

Многие начинающие автоматизаторы бросаются учить язык программирования, пишут первые учебные тесты и даже успешно делают тестовые задания по автоматизации тест-кейсов. Однако не все задумываются о том, что с этими тестами в реальной работе делать дальше. Кто их будет запускать? Когда? Каким образом? Здесь я бы хотел рассказать о пайплайне CI, месте автотестов в нем, а так же как и на чем тесты запускаются.

Прежде всего, откуда этот пайплайн взялся и что за CI/CD.

Непрерывная интеграция (Continuous Integration, CI) и непрерывная поставка (Continuous Delivery, CD) представляют собой культуру, набор принципов и практик, которые позволяют разработчикам чаще и надежнее развертывать изменения программного обеспечения.

CI/CD - это одна из DevOps-практик. Она также относится и к agile-практикам: автоматизация развертывания позволяет разработчикам сосредоточиться на реализации бизнес-требований, на качестве кода и безопасности.

Непрерывная интеграция - это методология разработки и набор практик, при которых в код вносятся небольшие изменения с частыми коммитами. И поскольку большинство современных приложений разрабатываются с использованием различных платформ и инструментов, то появляется необходимость в механизме интеграции и тестировании вносимых изменений. С технической точки зрения, цель CI - обеспечить последовательный и автоматизированный способ сборки, упаковки и тестирования приложений. При налаженном процессе непрерывной интеграции разработчики с большей вероятностью будут делать частые коммиты, что, в свою очередь, будет способствовать улучшению коммуникации и повышению качества программного обеспечения.

Непрерывная поставка начинается там, где заканчивается непрерывная интеграция. Она автоматизирует развертывание приложений в различные **окружения**: большинство разработчиков работают как с продакшн-окружением, так и со средами разработки и тестирования.

Инструменты CI/CD помогают настраивать специфические параметры окружения, которые конфигурируются при развертывании. А также CI/CD-автоматизация выполняет необходимые запросы к веб-серверам, базам данных и другим сервисам, которые могут нуждаться в перезапуске или выполнении каких-то дополнительных действий при развертывании приложения.

Непрерывная интеграция и непрерывная поставка нуждаются в непрерывном тестировании, поскольку конечная цель - разработка качественных приложений. Непрерывное тестирование часто реализуется в виде набора различных автоматизированных тестов (регрессионных, производительности и других), которые выполняются в **CI/CD-конвейере** (pipeline, build chain и т.д.). Зрелая практика CI/CD позволяет реализовать непрерывное развертывание: при успешном прохождении кода через CI/CD-конвейер, сборки автоматически развертываются в продакшн-окружении. Команды, практикующие непрерывную поставку, могут позволить себе ежедневное или даже ежечасное развертывание.

Типичный CD-**конвейер** состоит из этапов сборки, тестирования и развертывания. Более сложные конвейеры включают в себя следующие этапы:

- Получение кода из системы контроля версий и выполнение сборки;
- Настройка инфраструктуры, автоматизированной через подход “инфраструктура как код”;
- Копирование кода в целевую среду;
- Настройка переменных окружения для целевой среды;
- Развертывание компонентов приложения (веб-серверы, API-сервисы, базы данных);

- Выполнение дополнительных действий, таких как перезапуск сервисов или вызов сервисов, необходимых для работоспособности новых изменений;
- Выполнение тестов и откат изменений окружения в случае провала тестов;
- Логирование и отправка оповещений о состоянии поставки.

Например, в Jenkins конвейер определяется в файле Jenkinsfile, в котором описываются различные этапы, такие как сборка (build), тестирование (test) и развертывание (deploy). Там же описываются переменные окружения, секретные ключи, сертификаты и другие параметры, которые можно использовать в этапах конвейера. В разделе post настраивается обработка ошибок и уведомления. В более сложном CD-конвейере могут быть дополнительные этапы, такие как синхронизация данных, архивирование информационных ресурсов, установка обновлений и патчей. CI/CD-инструменты обычно поддерживают плагины. Например, у Jenkins есть более 1500 плагинов для интеграции со сторонними платформами, для расширения пользовательского интерфейса, администрирования, управления исходным кодом и сборкой.

Многие команды, использующие CI/CD-конвейеры в облаках используют **контейнеры**, такие как Docker, и **системы оркестрации**, такие как Kubernetes. Контейнеры позволяют стандартизировать упаковку, поставку и упростить масштабирование и уничтожение окружений с непостоянной нагрузкой. Есть множество вариантов совместного использования контейнеров, инфраструктуры как код (**Iaas**) и CI/CD-конвейеров. Архитектура бессерверных вычислений представляет собой еще один способ развертывания и масштабирования приложений. В бессерверном окружении инфраструктурой полностью управляет поставщик облачных услуг, а приложение потребляет ресурсы по мере необходимости в соответствии с его настройками. Например, в AWS бессерверные приложения запускаются через функции AWS Lambda, развертывание которых может быть интегрировано в CI/CD-конвейер Jenkins с помощью плагина.

Больше о CI/CD можно почитать тут и тут.

Этапы конвейера

В момент, когда триггерится сборка, например, когда разработчик сделал коммит в свою ветку, запускается процесс, который выполняется специально написанными скриптами и утилитами. Этот процесс состоит из нескольких обязательных шагов. Простой пример для PR:

- При открытии каждого Pull Request, Git-сервер отправляет уведомление CI-серверу;
- CI-сервер клонирует репозиторий, проверяет исходную ветку (например bugfix/wrong-sorting), и сливает код с кодом master-ветке;
- Тогда запускается билд-скрипт (сценарий сборки). Например ./gradlew build;
- Если эта команда возвращает код ответа “0”, то билд успешно выполнен. (Другой ответ означает ошибку);
- CI-сервер направляет уведомление об успешном билде на Git-сервере;
- Если билд был успешен, то Pull Request разрешается слить с существующим кодом. (Если не успешен, то, соответственно, не разрешается).

Ошибка в любом из шагов приводит к полному падению всей сборки. Ну и, само собой разумеется, шаги расположены в таком порядке, чтобы сужать воронку потенциальных проблем. Если Quality Gate предыдущего этапа не пройдет, то на проверку следующего уже можно не тратить ресурсы.

Пример Quality Gates, которые встроены в pipeline отсюда:

- Сборка сервиса:
    - Проверка наличия конфигурации корректного формата;
    - Проверка стандартов оформления кода;
    - Проверка на необходимое покрытие Unit-тестами;
    - Генерации и публикации контрактов (контроль обратной совместимости).
- Запуск Beta-тестов;

- Обязательный code-review;
- Сканирование на уязвимости.

Пример сферического пайплайна в вакууме отсюда:

- Code scanning: код проверяется на соответствие общему гайдлайну (linters), уязвимости (code security) и качество (code quality);
- Unit tests;
- Build: этап для сборки artifacts/packages/images и т.д. Здесь уже можно задуматься о том, каким будет стратегия версионирования всего приложения. Во времена контейнеризации, в первую очередь интересуют образы для контейнеров и способы их версионирования;
- Scan package: пакет/образ собрали. Теперь нужно просканировать его на уязвимости. Современные registry уже содержат инструментарий для этого;
- Deploy: стадия для развертывания приложения в различных окружениях;
- Integration testing: приложение задеплоили. Оно где-то живет в отдельном контуре. Наступает этап интеграционного тестирования. Тестирование может быть как ручным, так и автоматизированным;
- Performance testing (load/stress testing): данный вид тестирования имеет смысл проводить на stage/pre-production окружениях. С тем условием, что ресурсные мощности на нем такие же, как в production;
- Code Review / Approved: одним из важнейших этапов являются Merge Request. Именно в них могут производиться отдельные действия в pipeline перед слиянием, а также назначаться группы лиц, требующих одобрения перед слиянием.

Больше про build-agent можно почитать тут: TeamCity: настраиваем CI/CD в вашей команде и тут Что такое сборщик продукта, про окружения тут Создаем инфраструктуру для интеграционных тестов

E2E автотесты

Теперь пора поговорить непосредственно про то, что чаще всего касается рядового автоматизатора - **E2E автотесты**. Как мы выяснили выше, до прогона E2E сборка сначала проходит несколько шагов, а условиями запуска чаще являются ключевые моменты: commit, pull request и merge request. В моей прошлой компании был очень простой конвейер, где на коммит в фича-ветку запускались тесты разработчика, на вливание в develop стартовали критичные E2E тесты, а на вливание в main уже всё что есть, включая регрессионные тесты. Теперь, когда известны места автотестов в конвейере, нужно еще понять, на чем эти тесты будут прогоняться и как имея скрипт автотеста его запустить в конвейере.

Все эти моменты конфигурируются в самом CI-агенте, в jenkins это были джобы (job). Пример: How to Add your First Android Job to Jenkins. За запуск кода тестов, когда вы инициируете запуск из конвейера или локально, отвечает **Test Runner**. Это лишь одна из задач, в зону ответственности раннера входят:

- подготовка окружения;
- формирование набора тестов для исполнения;
- запуск тестов;
- получение результатов выполнения тестов;
- подготовка отчетов о прохождении тестов;
- сбор статистики.

Несмотря на то, что с фреймворком поставляется также и раннер, существует возможность использования более совершенных сторонних раннеров.

Параллельно с вопросом о том, какой раннер выбрать для тестов, перед вами встает другой: а на чем лучше запускать тесты? Есть три опции:

- Настоящий девайс.

    - Плюсы. Покажет проблемы, специфичные для конкретных устройств и прошивок. Многие производители меняют Android под себя - как UI, так и логику работы ОС. И бывает полезно проверить, что ваше приложение корректно работает в таком окружении.
    - Минусы. Необходимо где-то добыть ферму устройств, организовать специальное помещение под них - необходима низкая температура, нежелательно попадание прямых солнечных лучей и т. д. Кроме того, аккумуляторы имеют свойство вздуваться и выходить из строя. А еще сами тесты могут менять состояние устройства, и вы не можете просто взять и откатиться на какой-то стабильный снепшот.
- Чистый эмулятор. Под «чистым» мы подразумеваем, что вы запускаете эмулятор у себя или где-то на машине, используя установленный на эту машину AVD Manager.
    - Плюсы. Быстрее, удобнее и стабильнее настоящего устройства. Создание нового эмулятора занимает считаные минуты. Никаких проблем с отдельным помещением, аккумуляторами и прочим.
    - Минусы. Отсутствие упомянутых выше device specifics. Однако зачастую количество тестовых сценариев, завязанных на специфику устройства, ничтожно мало, и они не высокоприоритетные. Но самый главный минус - это плохая масштабируемость. Простая задача залить новую версию эмулятора на все хосты превращается в мучение.
- Docker-образ Android-эмулятора. Docker решает недостатки чистых эмуляторов.
    - Плюсы. Docker и соответствующая обвязка в виде подготовки и раскатки образа эмулятора - это полноценное масштабируемое решение, позволяющее быстро и качественно готовить эмуляторы и раскатывать их на все хосты, обеспечивая их достаточную изолированность.
    - Минусы. Более высокий входной порог.

В сети есть разные Docker-образы Android-эмуляторов:

- Docker image от Avito;
- Docker image от Google;
- Docker image от Agoda.

Предстоит сделать сложный выбор между облачным решением, локальным решением с нуля и локальным решением на базе чего-то, если в компании есть своя инфраструктура по запуску тестов других платформ. Вся эта инфраструктура уже связывается с раннером для запуска тестов, после чего уже решаются остальные моменты, такие как вывод отчета по прогону тестов (например, Allure) и внедрение/синхронизация с TMS.

Источники:

- Что такое CI/CD? Разбираемся с непрерывной интеграцией и непрерывной поставкой
- Разбираемся в CI/CD
- Автотесты на Android. Картина целиком

Доп. материал:

- DevOps инструменты не только для DevOps. Процесс построения инфраструктуры автоматизации тестирования с нуля
- Руководство для начинающих: создаем DevOps-пайплайн
- Зачем CI/CD тестировщикам?
- Jenkins Pipeline. Что это и как использовать в тестировании
- ГДЕ И КАК ПРОГОНЯТЬ UI ТЕСТЫ
- Инфраструктура + тестирование = любовь
- Leadership in test: a guide to infrastructure and environments
- Leadership in test: testing tools
- Создаем инфраструктуру для интеграционных тестов
- HOW TO RUN API SMOKE TESTS IN YOUR CONTINUOUS DEPLOYMENT PIPELINE

- Разбираемся в CI/CD
- Что такое CI (Continuous Integration)
- Как мы настроили CI/CD, чтобы релизить часто и без страха
- Как настроить Pipeline для Jenkins, Selenoid, Allure
- CI-билд - это не елка
- Идеальный пайплайн в вакууме
- Основные подходы к CI/CD для целей тестирования ПО
- How To Have Seamless Script Execution Planning And Reporting For Success Of An Automation Project
- Что такое сборщик продукта
- #8 QA инфраструктура компании на локальной машине. Docker, Jenkins, Jira, Selenoid, Allure.
- Автотесты и Docker
- Интеграция с Allure: структурировать, упростить, стабилизировать
- Автоматизация расчета покрытия требований и его визуализация в Allure Report
- Что такое Docker
- Continuous Integration with Jenkins
- Стратегии деплоя в Kubernetes: rolling, recreate, blue/green, canary, dark (A/B-тестирование)
- Как e2e автотесты на Selenide помогают QA-команде при частых релизах
- Как ускорить автотесты
- Тестируем CI Pipeline
- Switchboard: набор инструментов для управления авто-тестами
- Централизованное логирование в Docker с применением ELK Stack
- Строим домашний CI/CD при помощи GitHub Actions и Python
- Нагрузочное тестирование как CI-сервис для разработчиков
- Автоматическое тестирование микросервисов в Docker для непрерывной интеграции
- Тесты на pytest с генерацией отчетов в Allure с использованием Docker и Gitlab Pages и частично selenium
- Автоматизация системных тестов на базе QEMU (Часть 1/2)
- Как сократить время сборки образов Docker в GitLab CI
- Crash-crash, baby. Автоматический мониторинг фатальных ошибок мобильных приложений
- git-flow, GitHub flow, GitLab Flow

Процессы и автоматизация проекта с нуля

Есть множество статей про технологии и те или иные подходы к автоматизации. Но почему-то нет статей про «обратную сторону» автоматизации. Как вообще всё зарождается на проекте? И как это «всё» организовать? Ниже будет рассмотрен пример компании Россельхозбанк.

Общая информация по проекту

- Большой проект с командами, разрабатывающими общий продукт;
- В командах есть QA Manual (Ручной тестировщик);
- Направления тестирования - Frontend, Backend.

Изучаем проект

Понимаем цель разработки продукта и кто потребитель: ответы на эти вопросы помогают понять, на что делать упор в автоматизации. Например, если работаем с юридическими лицами, то делаем упор на тестирование «исполнения законодательства» и переводы по платежам и тд. Если это физические лица, то на основные выполняемые операции: переводы с карты на карту, оплату услуг и тп. Так же важно, чтобы направление автоматизации «смотрело» в целом на продукт, а не на отдельные в нем команды.

Знакомимся с участниками разработки продукта: в нашем случае нужно быть знакомым со всеми, так как необходимо будет взаимодействовать тем или иным образом. Выделю ключевых людей:

- Владельцы продуктов (Product Owners), они заказчики автоматизации в команде;
- QA каждой команды. Они - это клиенты инструмента автоматизации, их уровень счастья - это показатель успеха;
- Лидер ручного тестирования. Помогает в организации процесса и во взаимодействии с ручным тестированием;
- Лидер разработки Frontend. Он влияет на стабильность и качество автотестов;
- Специалист по закупке. Решит вопросы выделения техники, в основном с серверным железом.

Изучаем каждую команду: собираем информацию о том, какую часть проекта разрабатывает команда.

- Разбираемся в направлении - Frontend, Backend или всё сразу;
- Разбираемся с тем, как QA тестируют свою часть продукта;
- Выясняем, на каком уровне QA manual знаком с автоматизацией;
- Находим боли в тестировании и что приоритетно закрыть автоматизацией.

Формируем требования к автоматизации и заказываем ресурсы

Что мы собрали?

- У нас есть 8 команд;
- Есть 11 QA инженеров;
- Есть направления тестирования Frontend, Backend + будем «ходить» в Базу данных;
- 90% QA manual никогда не занимались автоматизацией.

Исходя из данных, формируем требования к автоматизации

У нас нет задачи делать какие-то инновационные решения, поэтому придерживаемся классики:

- В качестве языка программирования выбираем Java - так будет проще найти специалистов;
- Нужно тестировать Frontend. Берем Selenium;
- Нужно тестировать Backend. Взаимодействуем с ним через REST. Берем REST-assured;
- Нужно «ходить» в базу данных. Возьмем стандартные библиотеки из Java;
- Нужно либо обучить QA manual разработке автотестов, либо нанять армию из N автоматизаторов. Мы думаем о расходах проекта на тестирование, поэтому убиваем 2-х зайцев сразу. Берем Cucumber;
- Нам нужна отчетность с красивыми графиками. Берем Allure.

Получился следующий стек технологий: Java, Selenium, REST-assured, Cucumber.

Оцениваем силы автоматизаторов

Поскольку их нет, берем 1 автоматизатора на 5 команд (больше 5-и не потянет).

Автотесты нужно где-то запускать

Что нам понадобится?

- Машина для Jenkins. 1 штука. Через него реализуем удаленный запуск;
- Машина под Slave Jenkins. Как agent для Jenkins;
- Машина под Selenoid. Для параллельного запуска тестов.

Делаем демо нашего инструмента

Наше демо будут смотреть все: владельцы продуктов, QA инженеры, разработчики, аналитики и тд. Значит ориентируемся на понимание для всех.

Берем команду в качестве первой «жертвы» автоматизации. Ребята разрабатывают Frontend. Нам нужна наглядность.

- Делаем 5-10 автотестов. Записываем на видео;
- Обязательно после прогона показываем Allure. Красивые графики любят все;
- Рисуем «инфраструктуру автотестов». Главное, чтобы было просто и понятно;
- Обозначаем главную цель автоматизации;
- Демонстрируем эффект от автоматизации;
- Делаем сравнительные тесты. Ручное прохождение и автоматизированное;
- Показываем, как создаются сценарии;
- Показываем планы автоматизации (когда появятся тот или иной функционал);
- Показываем, как будет происходить внедрение автоматизации в команды.

Подготавливаем UI к автоматизации

Для обеспечения надежности, стабильности и качества автотестов, необходимо раскидать якоря на UI. Централизованно через лидера Frontend и дополнительно через владельцев продукта проставляем атрибут для UI-элементов “data-test-id“ или с любым другим названием. Смысл в том, чтобы со стороны UI проставить этот атрибут во всех элементах типа «Таблица, поле для ввода, кнопка» и тд. Если коротко, то на всех элементах, с которыми взаимодействуем, это даст +300% к надежности автотестов. Переезд элемента в другое место или изменение его содержимого никак не повлияют на проверку автотестом. Делаем этот момент обязательным для всех новых фич.

Разрабатываем автотесты

- Делим команды между автотестерами;
- Разрабатываем каркас проекта для автоматизации. Все по шаблону;
- Подготавливаем набор Cucumber шагов для работы с Frontend, основным направлением в тестировании. Выносим шаги в отдельный проект и делаем из него подключаемый артефакт;
- Настраиваем Selenoid и Jenkins;
- Начинаем подключать команды к автоматизации. При подключении команды все сводится к тому, чтобы создать репозиторий, загрузить каркас, создать Job в Jenkins по шаблону, обучить QA работе с Cucumber, работе с Git и средой разработки;
- После обучения QA manual самостоятельно пишут автотесты. Один раз за спринт автоматизатор приходит в команду и принимает написанные автотесты, делает правки и вливает в основную ветку. На этом этапе ребята качают уровень написания правильных сценариев, а мы получаем качественные проверенные сценарии;
- После встречи со всеми командами у автоматизатора в спринте остается еще 5-6 свободных дней. Это время тратим на разработку Cucumber шагов;
- В конце спринта на продуктовом демо показываем результаты по командам и делаем анонсы новых фич в инструменте.

Результаты

6 спринтов (60 дней), 8 команд, 2 автотестера и 11 ручных тестировщиков - имеем 50% покрытия регресса проекта автотестами. Это с учетом подключения команд (подключались постепенно) и разработки шагов.

Вещи, о которых нужно помнить при разработке с таким подходом

Ручной тестировщик - это клиент. QA инженер - прямой потребитель инструмента автоматизации. Их уровень комфорта, счастья и количество автотестов - это показатель качества нашей работы. Если один из показателей падает, значит нужно что-то менять. Иначе все посыпется.

Ориентация на выгоду для проекта. Нужно всегда помнить, что содержание разработчиков, тестировщиков, аналитиков и других специалистов стоит денег. В конечном итоге наша цель их сэкономить. Как мы их экономим?

- Автотестами: находим дефекты запуская их каждый день.
- На каждом этапе тестирования сокращаем время регресса.

Все это приводит к более быстрому выпуску продукта в промышленную эксплуатацию.

Не работает? Меняй! Не нужно бояться ошибок. Их будет очень много в разработке, в общении с командами, во взаимодействии с QA инженерами, в демо. Главное увидеть, что «что-то» не работает и менять подход до тех пор, пока все не будут в восторге. Всё делаем для людей. Не для себя.

Источники:

- Автоматизация тестирования «с нуля» (нетехническая сторона вопроса)

Доп. материал:

- Гайд внедрения автоматизации тестирования, если ты рядовой QA инженер
- Путь к автоматизации тестирования в SuperJob: инструменты, проблемы и решения
- Как мы автоматизировали тестирование бэкенда
- Готовим приложение для автоматизации тестирования
- О чем спрашивать, унаследовав тест-автоматизацию
- Андрей Солнцев - Воркшоп: Как начать свой проект автоматизации с нуля. Продолжение (часть 1)
- Андрей Солнцев - Воркшоп: Как начать свой проект автоматизации с нуля. Продолжение (часть 2)
- AQA Checklist \ Score для интеграции Автоматизации тестирования в существующие Agile процессы
- 5 заклинаний для команды автоматизации
- Организация мониторинга бизнес-процессов (с нуля до автоматизации)
- Вместо 100 запусков приложения - один автотест, или как сэкономить QA-инженеру 20 лет жизни
- Автоматизация тестирования в микросервисной архитектуре
- Как (авто)тестировать Монстра
- Как автоматизировать тестирование на проекте, где по 100 релизов в месяц
- Процесс автоматизированного тестирования за 10 шагов
- How Does Test Planning Differ For Manual And Automation Projects?
- Step By Step Guide To Implement Proof Of Concept (POC) In Automation Testing
- How To Perform Automation Testing Of JAVA/J2EE Applications (Part 2)

Лучшие практики автоматизации

В каждой статье свой топ лучших практик, характеристик хороших тестов и подобного, так что я просто скопировал наиболее полезные на мой взгляд списки.

7 характеристик отлично написанных тестов

- **Тест полностью автоматизирован** (очевидно): Иногда попадаются такие тесты, которые автоматизированы не полностью. Самые распространенные причины: либо это очень сложно, либо просто невозможно;
- **Тест повторяем**: тест не ломается, если приложение не поменялось: Это относится к основам генерации уникальных данных. Например, мы тестируем регистрацию. Очевидно, что если не генерировать уникальный емейл, то на продакшене такой тест, скорее всего, не будет работать;
- **Тест заканчивается валидацией**: За исключением случаев, где нужно выполнить действия по зачистке данных и подобных действий, завершение валидацией является лучшей практикой. Это помогает убедиться, что последнее действие прошло успешно;
- **Тест достаточно стабилен, чтобы его использовать в CI/CD**: Если тест регулярно ломается, то он недостаточно стабилен, чтобы использовать его в CI/CD. Так как практически любая компания пытается добиться CI или даже CD, то часто такой тест не просто бесполезен, но даже вреден, так как отнимает большое количество времени и все равно не может использоваться в CI автоматически;

- **Тест очень легко читать**: Мы обычно не пишем тесты в одиночку. Часто это команда людей и нашим коллегам тоже приходится поддерживать наши тесты. Крайне важно, чтобы любой член команды мог разобраться в структуре теста, не тратя на это излишнее количество времени. Даже если мы пишем тесты в одиночку, иногда бывает очень тяжело понять, что тест делает и как, если он не написан специально для облегчения понимания;
- **Тест требует минимальной поддержки**: Пункт очевидный, но не всегда соблюдаемый. Чем меньше мы тратим времени на поддержку, тем больше у нас времени на то, чтобы сделать что-то полезное, как, например, написать больше тестов;
- **Тест работает параллельно с другими тестами и не ломается**: В какой-то момент, особенно для end-to-end тестов, мы сталкиваемся с тем, что прогон тестов занимает слишком много времени, это снижает скорость разработки и приводит к таким эффектам, как неоттестированный патч. На этом этапе мы обычно задумываемся о параллелизации, чтобы ускорить исполнение тестов. Если тесты были написаны так, что они могут быть запущены параллельно в любой последовательности и не пересекаться друг с другом, то это делает задачу параллельного исполнения просто задачей настройки инфраструктуры, а не задачей переписывания тестов.

Еще десять заповедей автоматизации

1. **Не автоматизируй только "по тест-кейсам"**

Существует распространенное заблуждение, что тест-автоматизация обязана произрастать из тест-кейсов. Автоматизаторы берут существующие или свежесозданные тест-кейсы и превращают их в автоматизированные сценарии. Это называется "автокафе". В этом подходе может быть смысл, но другие подходы принесут не меньше, а то и больше выгоды. Расширяя определение автоматизации за рамки "тест-кейс - инструмент - тест-скрипт" до "продуманного применения технологии с целью помочь людям выполнять свою работу", мы можем использовать компьютерные мощности для задач, для которых они предназначены, а люди - тестировщики - пусть делают все остальное. К счастью, в большинстве задач, где хороши люди, компьютеры выступают плохо - и наоборот.

2. **Обращайся с разработкой автоматизации, как с разработкой ПО**

Разработка автоматизации - это и есть разработка ПО. Даже если мы используем интерфейс перетаскивания или записи и воспроизведения для создания автоматизации, то где-то под капотом или за занавесом над нашими действиями работает код. Мы должны начать обращаться с автоматизацией как с разработкой ПО, иначе мы окажемся с чем-то неподдерживаемым на руках, и проект умрет, едва родившись. Подход к автоматизации как к разработке ПО означает, что мы должны учитывать большинство (если не все) процессов и видов деятельности, требуемых при разработке ПО. Включая:

- Дизайн - мы должны решить, что внедрять и как это делать, чтобы это было поддерживаемым и пригодным к эксплуатации.
- Внедрение - надо написать код.
- Хранение - код и его артефакты должны где-то храниться.
- Тестирование - тестировать тесты? Естественно! Мы должны быть достаточно уверены, что автоматизация ведет себя так, как нам хочется. Если мы не доверяем автотестам, в них нет смысла.
- Баги - в любом ПО есть баги; автоматизация, как ПО, не исключение. Тестирование поможет нам, но не отловит все баги на свете. Выделите время на исправление багов.
- Логи - это артерия автоматизации. Без них мы не поймем, что автоматизация делает, и не сможем ее починить, когда она сломается. К тому же мы не сможем сказать, в автоматизации ли кроется проблема, или же в тестируемом ПО.

3. **Следуй стандартам и идиомам программирования**

Помимо обращения с автоматизацией как с разработкой ПО, мы должны встраивать в нее соответствующие идеи по внедрению. Каждый инструмент и язык имеют свои собственные идиомы и хитрости, но

общепринятые подходы к проектированию и внедрению обычно годятся для всего. Эта статья про инкапсуляцию и абстракцию рассказывает об образцовых практиках, которое можно переносить в другие специфичные области.

4. **Не забывай про поддержку и обслуживание**

ПО не идеально и никогда не бывает полностью готово; с автоматизацией то же самое. В ней будут баги. С изменением тестируемого приложения нужно будет соответственно менять автоматизацию. Мы не можем это предотвратить, но можем разработать наше ПО так, чтобы снизить затраты на его поддержку и обслуживание; и на них тоже надо выделить время. Эта статья и этот пост в блоге рассказывают о факторах, которые надо учитывать, планируя будущую поддержку.

5. **Не делай скрипты зависимыми друг от друга**

Создание тест-скрипта, зависимого от результатов другого теста - как правило, мощный анти-паттерн. Если скрипты зависят друг от друга, их нельзя прогонять поодиночке, и это усложняет дебаг автоматизации и проблем приложения. К тому же зависимые скрипты нельзя запускать параллельно с теми, от которых они зависят. Тут есть граничный случай, когда абсолютно все скрипты зависят от одного-единственного. Этот единичный скрипт обычно настраивает тест-окружение и тестовые данные. В условиях современных фреймворков автоматизации и непрерывной развертки это все большая редкость, но этот сценарий может быть уместен в ситуациях, когда фреймворки недоступны или не подходят для специфической автоматизационной задачи.

6. **Внедряй грамотное логирование и отчеты**

Как описано здесь, грамотные логи, результаты и сообщения об ошибках критично важны для понимания, доверия к, и поддержки автоматизации. Эти логи - наш детализированный образ прогона автотестов: что запускалось, как именно, что упало, как оно упало, как оно преуспело, и т. д. Конечно, если мы тщательно внедрили грамотное логирование, которое дает нам эту информацию в читабельной форме.

7. **Влияй на тестируемость и автоматизируемость**

Тестируемость, та степень, до которой приложение или фича могут быть протестированы, и автоматизируемость, степень, до которой тест-деятельность может выполняться автоматически - это не то, что могут создавать тестировщики, QA и QE, но, конечно, есть вещи, на которые мы можем повлиять. Осуществление этого влияния - наша обязательная задача. Разработчики не всегда знают, что нам нужно для тестирования и для создания автотестов. Мы должны донести это до них. Статьи здесь и здесь рассказывают о некоторых аспектах тестируемости и автоматизируемости.

8. **Не попадайся в ловушку невозвратных затрат**

Иногда мы делаем ошибки. Иногда это крупные ошибки. Мы извлекаем максимум из информации, которой располагаем в моменте, но это не всегда срабатывает. Когда что-то в нашем плане идет не так, мы инстинктивно стараемся это исправить и продолжать исправлять. Однако иногда стоит начать заново, иначе мы рискуем выбросить хорошие деньги вслед за плохими. Это называется "ловушкой невозвратных затрат". Если кратко, она заключается в мнении, что мы уже потратили столько денег на этот проект, что обязаны потратить еще для его реабилитации, дабы извлечь выгоду из уже потраченных, невозвратных затрат на него. Возможно, мы эмоционально привязаны к нашему программному "ребенку"; мы потратили на него столько времени и денег, что не в силах выбросить его и начать заново. Возможно, мы боимся; руководство, скорее всего, не придет в восторг, захоти мы бросить все и снова сделать "то же самое". Мы должны рассматривать такие ситуации с точки зрения бизнеса, а не предаваться эмоциям.

9. **Опасайся хитроумных приспособлений**

Машины Руба Голдберга - это сложные машины, выполняющие сравнительно простые задачи, вроде "Автономного платочка". В мире автоматизации создание таких машин - очень веселое дело, и они могут делать крутые штуки вроде увязывания вместе разных инструментов для выполнения тест-задач. Минус в том, что это сложно понимать и поддерживать; надо опасаться создания того, что сложнее поддерживать, нежели выполнять эти задачи вручную. Эта статья расскажет больше об автоматизированных машинах Руба Голдберга; этот пост размышляет о состоянии "автоматизированности".

10. **Не делай тестовые данные зависимыми от временных данных**

Скрипт, с которым я столкнулся недавно, в один прекрасный день начал падать. Исследуя вопрос, мы выяснили, что падал он из-за смены месяца с июля на август. Скрипт был написан таким образом, что оракул проверял даты в июле, и в июле все было хорошо - приложение возвращало даты текущего месяца, июля. Когда дата сменилась на август, приложение начало возвращать даты августа, и тест падал. В этом случае надо не жестко кодировать даты июля, а динамически генерировать данные на основании текущего месяца.

Как можно и нельзя автоматизировать

Нельзя:

- **Автоматизировать все ручные тесты**: ручные тесты прекрасны - они находят проблемы, о которых вы даже не помышляли. Но ручной подход к тестированию не всегда можно напрямую перенести на автоматизированные проверки. Возможно, стоит создать новые сценарии специально для автоматизации.
- **Делать автотесты задачей одного-единственного человека**: когда вы только начинаете, найм консультанта по автоматизации - хорошая мысль, однако убедитесь, что соответствующие знания доступны всем участникам проекта. В один прекрасный день консультант или ключевой исполнитель покинет вас, и разбираться с тест-набором придется всем остальным.
- **Автоматизировать все на свете**: сценарий автоматизируется не просто так - на это есть причины, его автоматизация каким-то образом ценна для проекта. Начните с рутинных задач, которые все равно выполняются ежедневно - и вы сразу увидите первую, мгновенную выгоду от усилий по автоматизации.
- **Автоматизировать, просто чтобы автоматизировать**: подумайте, что вам нужно, и что нужно прямо сейчас. Если вы настроены на 80-100% автоматизацию во всех проектах просто потому, что "это круто и так хочет генеральный директор, или "это хорошо звучит на собраниях совета директоров", то вы попусту тратите время и деньги.
- **Покупать лидирующий по рынку, наилучший инструмент автоматизации**: найдите инструмент, который подходит под ваши задачи, и убедитесь, что он действительно делает то, что обещает. Не поддавайтесь на уговоры продавцов купить дорогостоящую программу, которая очень крута прямо сейчас в этом конкретном проекте или на этой конкретной платформе. И не заставляйте другие проекты использовать инструмент, если он им не нужен.
- **Думать в терминах "прошел" и "упал"**: у вас будут упавшие кейсы. Это не значит, что там есть ошибка. Не воспринимайте упавший тест как сигнал о баге - вам нужно разобраться, ПОЧЕМУ тест упал. Также подумайте, действительно ли категории "прошел/упал" подходят для отчетности об автотестах? Возможно, стоит подумать о другой классификации, которая лучше опишет происходящее?
- **Запускать все подряд каждый божий раз**: подумайте о цели прогона ваших тестов. Если вам не нужно тестировать отдельную область - не делайте этого, тестируйте только ту функциональность, которая в этом нуждается. Прочее тестирование - это лишняя трата времени и сил, которая замусорит ваш отчет ненужными данными. Конечно, некоторые проверки могут гоняться постоянно, но убедитесь в том, что это оправданные меры.
- **Забывать тестировать собственные тесты**: привыкайте к идее тестировать свои тесты. Автоматизированный тест-сценарий не должен выпускаться в мир, пока вы не увидели, как он

несколько раз упал. Запускайте тест в обстоятельствах, когда он безусловно упадет, чтобы учесть это, когда он будет выпущен в релиз.
- **Недооценивать усилия по поддержке автоматизированного набора тестов**: автоматизированный тест-сьют - не та штука, которую сделал и забыл, а она бегает себе там. Ваша система постоянно меняется, и тест-набор нужно обновлять и пополнять беспрестанно. Не сваливайте эту ответственность на единственного тех-тестировщика в вашей компании.

Нужно:

- **Дробить тесты на независимые сценарии**: куда легче определить, где гнездятся проблемы и ошибки, если ваш тест-набор состоит из кратких, конкретных тест-сценариев. Не пытайтесь впихнуть все в один сценарий. Очень, ОЧЕНЬ трудно разобраться, что же пошло не так, если вам надо докопаться до всего на свете, проверяющегося в вашем тесте. Пусть тесты будут краткими и простыми.
- **Сделать автоматизацию задачей всей команды проекта**: почему бы не дать возможность всем, а не только тестировщикам, влиять на набор автотестов? Разработчики, менеджеры проекта, продакт-оунеры - всем им есть что сказать на предмет автоматизированного тестирования, и они могут принести ему пользу.
- **Донести информацию о результатах автоматизированного тестирования до всех - и подумать, кто получает эти результаты**: предоставьте разную информацию разным людям в проекте. Разработчику нужна специфическая информация о прогоне автотестов, и она безразлична менеджеру проекта. Опросите всех заинтересованных лиц, выясните, какая информация им требуется.
- **Начинать с простых вещей и пополнять тест-сьюты по ходу дела**: начните с того, что вам необходимо, и следуйте по воркфлоу проекта. Проекты постоянно меняются - не надо тратить время, силы и деньги на то, что не будет использовано или изменится в последний момент. Создавайте набор тестов по кусочкам, шаг за шагом.
- **Заставить ваш фреймворк работать на износ**: вы приобрели дорогую программу, настроили ее и потратили время на создание и обновление наборов тестов. Пользуйтесь этим! Создайте набор, который может прогоняться часто И при этом ценен для проекта! Набор автотестов, который не запускается - это пустая трата денег.
- **Учитывать проектные и организационные изменения**: найдите программу, которая способна расти вместе с вашими проектами и вашей организацией, и выясните, что вы можете - и не можете - с ней делать. Если она вам не подходит, отбросьте ее. Нет смысла тратить силы и время на неподходящие вам ресурсы.
- **Разгрузить ручных тестировщиков**: думайте о наборе автотестов как о помощи мануальным тестировщикам. Освободите их от нудных повторяющихся задач, пусть они делают то, в чем они звезды - тестируют, а не тупо следуют сценариям.
- **Сделать запуск тестом простым и быстрым**: настройка и старт набора автотестов должна быть простой задачей и давать мгновенную обратную связь. Если ваш тест-набор замедлен или очень сложен, люди просто не будут заморачиваться с запуском.
- **Постоянно обновлять тест-набор**: если меняется ваш код - меняются автотесты, и это ваше золотое правило. Оно верно для ВСЕХ изменений базы кода. Правки багов, внедрение фич - все это ведет к изменениям тест-набора.

О паттернах проектирования/архитектуре есть ссылки в доп. материалах ниже. О создании фреймворка в теме про виды и инструменты.

Источники:

- 7 характеристик хороших тестов
- Еще десять заповедей автоматизации
- Как можно и нельзя автоматизировать

Доп. материал:

- Святослав Куликов “Тестирование программного обеспечения. Базовый курс” 3.2.2. Особенности тест-кейсов в автоматизации
- Чистая архитектура в автотестах
- Элементы хорошего автотеста
- Ловушки автоматизации (и 9 советов, как в них не попасть)
- 7 характеристик хороших тестов
- UI-автотесты: как делать не стоит
- 7 способов повысить эффективность автоматизации тестирования в Agile разработке
- UI-автотесты: как делать не стоит
- ТОП-5 вопросов начинающего автоматизатора про автотесты
- Top 10 Test Automation Strategies And Best Practices
- Пожалуй, лучшая архитектура для UI тестов
- Распространенные паттерны и методологии UI-автоматизации: реальные примеры
- SOLID и другие принципы объектно-ориентированного проектирования в контексте автоматизации
- ООП, «святая троица» и SOLID: некоторый минимум знаний о них
- Принцип открытости-закрытости

Что такое flaky tests?

Флейки-тесты, они же “флаки”, они же нестабильные, они же ненадежные, они же “моргающие”, они же “случайно успешные”

Flaky-тест, буквально “хлопчатый”, “рассыпающийся на кусочки”, в индустрии ИТ-тестирования означает нестабильный, ненадежный тест, который иногда “pass”, иногда “fail”, и трудно понять, по какой закономерности. Убийца времени тестировщика, источник нервозности в команде.

На такие тесты тратится много времени. Возникает задержка, пока команда не разберется, в чем дело. Конечно, страдает продуктивность.

Причины кратко

- Тестовый фреймворк изначально плохо собран. Его код не проверен на соответствие задачам. Совет: код фреймворка должен быть в достаточной мере чистым; должны соблюдаться принципы DRY, использоваться Object design pattern страницы, или Factory design pattern.
- В тесте слишком много hardcoded-данных. Нестабильность результатов тестов возникает, когда эти тесты запускаются в другом окружении. Надо откорректировать hardcoded-данные (практика показывает, что это чаще всего неправильно прописанные пути к чему-то, неправильные IP-адреса и т.п.).
- Кэширование предыдущего состояния теста и запуск нового теста с кэшированными данными. Лучше чистить кэш для каждого выполнения тестов. Проверяй данные перед каждым тестом, и очищай их состояние после каждого теста.
- По внешней причине, не связанной с самими тестами. Плохое подключение к интернету или к конкретной базе данных; неподходящая версия браузера; утечки памяти.
- “Потеря связи” со сторонним (3rd party) компонентом также приводит к ненадежности; здесь поможет тщательная проверка сторонних компонентов перед запуском.
- Неэффективные селекторы элементов (например XPATH), или некорректные координаты элементов. Селекторы можно поменять достаточно легко, корректируя дизайн страницы. Рекомендуется работать с более надежными селекторами (например “id” или “name”).
- Злоупотребление “sleep”-командами, останавливающими выполнение в ожидании чего-то. Здесь помогает замена “sleep”-ожидания на более надежное в этом плане “wait”-ожидание (пока элемент появится). Это экономит время и (почти) устраняет “моргание” тестов во многих случаях.

Что делать

Если есть время, надо документировать такие тесты, и отправлять в “карантин”. После выполнения тестов, проверь выведенные данные. Проверь логи, дампы памяти, текущее состояние системы. Возможно скриншоты, если это UI-тесты. В 90% случаев причина выяснится уже на этом этапе! Если нет, создай тикеты насчет этих тестов, и проверь их по одному в карантине, тщательно. Исключи тесты в карантине из пайплайна до конца проверки, пока не добьешься стабильности. В Google тоже бывают flaky-тесты, говорит Hala Samir из Google; как они решают эту проблему? Стандартно, например анализируют выведенные данные, проверяя корреляцию с функциями возможно вызвавшими нестабильность, по возможности без перезапуска тестов.

Еще раз о причинах

Если разделить причины по происхождению:

- Тесты сами по себе имеют “склонность к нестабильности”
- Проблемы с фреймворком
- Проблемы в подключаемых библиотеках/сервисах
- Проблемы в операционной системе / в устройстве

Подробнее.

Как уже говорилось выше, нестабильность результатов часто возникает из-за неправильной инициализации, или «очистки». Реже, но тоже случается - от неправильно подобранных тестовых данных. Головная боль тестировщиков - асинхронные действия, на это следует обращать особое внимание. Более простая причина - неправильный порядок запуска тестов.

Проблемы с фреймворком: часто фреймворк не сконфигурирован на выделение достаточных системных ресурсов; или неправильно планирует очередность запуска тестов, из-за чего они “перекрываются” между собой.

Если в проекте много сторонних библиотек или подключаемых сервисов, у них может быть свое “дерево зависимостей”, где и таятся причины нестабильности. Например переменные некорректно инициализируются; память “утекает”; какой-то ресурс не отвечает на запрос; и т.п. Чтобы исключить эти проблемы, в идеале надо стремиться к “герметичности” тестовой среды, то есть тестовая среда не должна иметь внешних зависимостей.

Операционная система и тестовые устройства. Банально, но причиной иногда бывает нестабильное интернет-подключение. Также банальная причина - ошибки чтения/записи на физический носитель данных.

Статистика

Статистически (со слов тестировщиков), основные причины нестабильности это: на первом месте по «сложности разбора» асинхронные операции (async wait); затем многопоточные операции; затем неправильный порядок запуска тестов; затем аппаратные проблемы (с сетью и синхронизацией времени, или с памятью).

Ну, а если найти причину нестабильных тестов и исправить их - не получается никак, то, как говорил вице-президент Unity3D Алан Берд, “нестабильные тесты это еще хуже, чем вообще без тестов”.

Источники:

- Нестабильные тесты. Почему они существуют и что с ними делать

Доп. материал:

- Blog: Flaky Testing
- Flaky-тесты: Откуда ноги растут. Опыт Uber
- Flaky тесты (они же моргающие или "случайно успешные")

Мутационное тестирование (Mutation testing)

Юнит тесты помогают нам удостовериться, что код работает так, как мы этого хотим. Одной из метрик тестов является процент покрытия строк кода (Line Code Coverage). Но насколько корректен данный показатель? Имеет ли он практический смысл и можем ли мы ему доверять? Ведь если мы удалим все assert строки из тестов, или просто заменим их на assertSame(1, 1), то по-прежнему будем иметь 100% Code Coverage, при этом тесты ровным счетом не будут тестировать ничего. Насколько вы уверены в своих тестах? Покрывают ли они все ветки выполнения ваших функций? Тестируют ли они вообще хоть что-нибудь? Ответ на этот вопрос даёт мутационное тестирование.

Мутационное тестирование (MT, Mutation Testing, Mutation Analysis, Program mutation, Error-based testing, Fault-based testing strategy) - это вид тестирования ПО методом белого ящика, основанный на всевозможных изменениях (мутациях) частей исходного кода и проверке реакции на эти изменения набора автоматических юнит тестов. Изменения в мутантной программе сохраняются крайне небольшими, поэтому это не влияет на общее исполнение программы. Если тесты после изменения кода не падают (failed), значит либо этот код недостаточно покрыт тестами, либо написанные тесты бесполезны. Критерий, определяющий эффективность набора автоматических тестов, называется Mutation Score Indicator (MSI).

Введем некоторые понятия из теории мутационного тестирования:

Для применения этой технологии у нас, очевидно, должен быть исходный код (source code), некоторый набор тестов (для простоты будем говорить о модульных - unit tests). После этого можно начинать изменять отдельные части исходного кода и смотреть, как реагируют на это тесты. Одно изменение исходного кода будем называть Мутацией (Mutation). Например, изменение бинарного оператора "+" на бинарный "-" является мутацией кода. Результатом мутации является Мутант (Mutant) - то есть это новый мутированный код. Каждая мутация любого оператора в вашем коде (а их сотни) приводит к новому мутанту, для которого должны быть запущены тесты. Кроме изменения "+" на "-", существует множество других мутационных операторов (Mutation Operator, Mutator, faults or mutation rules), каждый из которых имеет свою цель и применение:

- Мутация значений (Value mutation): изменение значения параметра или константы;
- Мутация операторов (Statement mutation): реализуется путем редактирования, удаления или перестановки оператора;
- Мутация решения (Decision Mutation): изменение логических, арифметических и реляционных операторов.

В зависимости от результата теста мутанты делятся на:

- Выжившие мутанты (Survived Mutants): мутанты, которые все еще живы, то есть не обнаруживаются при выполнении теста. Их также называют live mutants;
- Убитые мутанты (Killed Mutants): мутанты, обнаруженные тестами;
- Эквивалентные мутанты (Equivalent Mutants): мутанты, которые изменив части кода не привели к какому-либо фактическому изменению в выводе программы, т.е. они эквивалентны исходному коду;
- Нет покрытия (No coverage): в этом случае мутант выжил, потому что для этого мутанта не проводились тесты. Этот мутант находится в части кода, не затронутой ни одним из ваших тестов. Это означает, что наш тестовый пример не смог его охватить;
- Тайм-аут (Timeout): выполнение тестов с этим активным мутантом привело к тайм-ауту. Например, мутант привел к бесконечному циклу в вашем коде. Не обращайте внимание на этого мутанта. Он считается «обнаруженным». Логика здесь в том, что если этот мутант будет внедрен в ваш код, ваша CI-сборка обнаружит его, потому что тесты никогда не завершатся;
- Ошибка выполнения (Runtime error): выполнение тестов привело к ошибке (а не к провалу теста). Это может произойти, когда средство запуска тестов не работает. Например, когда средство выполнения теста выдает ошибку OutOfMemoryError или для динамических языков, когда мутант привел к

неразборчивому коду. Не тратьте слишком много внимания на этого мутанта. Он не отображается в вашей оценке мутации;
- Ошибка компиляции (Compile error): это состояние возникает, когда это компилируемый язык. Мутант привел к ошибке компиляции. Он не отражается в оценке мутации, поэтому вам не нужно уделять слишком много внимания изучению этого мутанта;
- Игнорируется (Ignored): мы можем видеть это состояние, когда пользователь устанавливает конфигурации для его игнорирования. Он будет отображаться в отчетах, но не повлияет на оценку мутации;
- Тривиальные мутанты (Trivial Mutants): фактически ничего не делают. Любой тестовый пример может убить этих мутантов. Если в конце тестирования остались тестовые примеры, значит, это недопустимый мутант (invalid mutant).

Метрики:

- Обнаруженные (Detected): это количество мутантов, обнаруженных нашим тестом, то есть убитых мутантов. Detected = Killed mutants +Timeout;
- Необнаруженные (Undetected): это количество мутантов, которые не были обнаружены нашим тестом, то есть выживших мутантов. Undetected = Survived mutants + No Coverage;
- Покрытые (Covered): это количество мутантов покрытых тестами. Covered = Detected mutants + Survived mutants;
- Действительные (Valid): это количество действительных мутантов, не вызвавших ошибки компиляции или рантайма. Valid = Detected mutants + Undetected mutants;
- Недействительные (Invalid): это количество всех недействительных мутантов, т.е. они не могли быть протестированы, так как вызывали ошибку компиляции или выполнения. Invalid = Runtime errors + Compile errors;
- Всего мутантов (Total mutants): содержит всех мутантов. Total = Valid + Invalid + Ignored;
- Оценка мутации на основе покрытого кода (Mutation score based on covered code): оценивает общий процент убитых мутантов на основе покрытия кода. Mutation score based on covered code = Detected / Covered * 100;
- Неправильный синтаксис (Incorrect syntax): их называют мертворожденными мутантами, это представлено как синтаксическая ошибка. Обычно эти ошибки должен обнаруживать компилятор;
- Оценка мутации (Mutation score): это оценка, основанная на количестве мутантов. В идеале равна 1 (100%). Mutation score = Detected / Valid * 100 (%).

Источники:

- Мутационное тестирование
- Mutation Testing Guide: What You Should Know

Доп. материал:

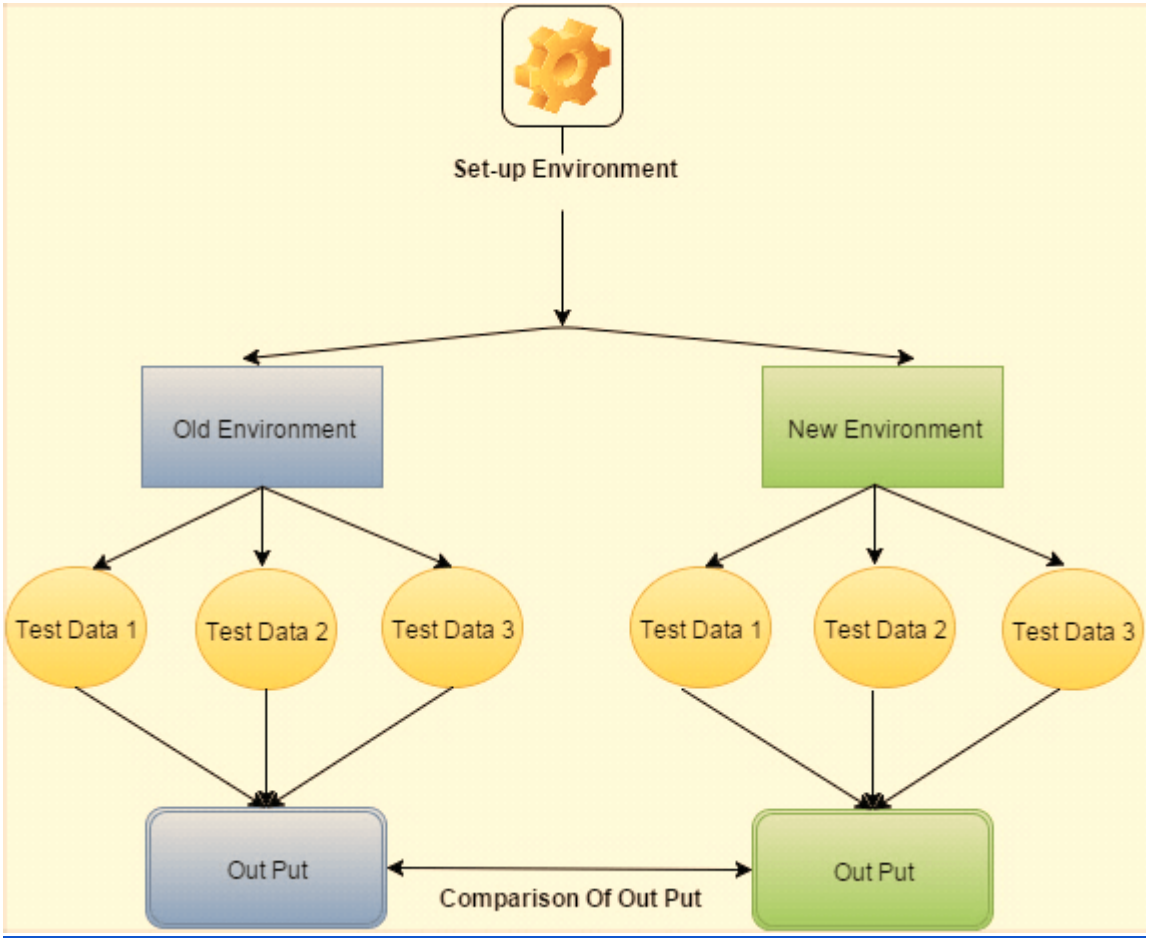
- Mutation testing
- Mutation Testing
- What Is Mutation Testing: Tutorial With Examples
- Мутанты, убийства - это о чем? О тестировании
- Изучаем mutmut - инструмент для мутационного тестирования на Python

Параллельное тестирование (Parallel testing)

Обычно большинство команд тестирования выполняют свои тесты по одному. Этот процесс называется последовательным тестированием или серийным тестированием (sequential testing or serial testing). В процессе последовательного тестирования каждый тестовый пример запускается один за другим и следующий тест не начинается, пока не завершится предыдущий. Последовательный процесс тестирования

требует много времени, усилий и ресурсов. Чтобы выпускать качественные продукты в кратчайшие сроки, более эффективным будет параллельное тестирование.

Параллельное тестирование включает в себя в основном автоматизированное тестирование нескольких версий одного и того же приложения или разных компонентов приложения одновременно и с одинаковыми входными данными, чтобы сократить общее время выполнения теста путем интеграции фреймворка автоматизации с облачными решениями и виртуализацией.



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».

Источники:

- Parallel Testing Guide: How To Perform Parallel Testing
- What is Parallel Testing? Definition, Approach, Example

Доп. материал:

- Parallel testing: get feedback earlier, release faster
- What Is Parallel Testing And Why Is It Important?

Подкожный тест (Subcutaneous test)

Впервые упоминание встречается в 2010 году у Jimmy Bougard в блоге: ”В то время как модульный тест фокусируется на мелкомасштабном дизайне, подкожный тест не касается дизайна, а вместо этого проверяет основные входы и выходы системы в целом”, затем в докладе Matt Davies and Rob Moore - “Microtesting - How We Set Fire To The Testing Pyramid While Ensuring Confidence”, а позже и в презентации Daniel Lafeir and Michael

Lawrie. Термин «подкожный» означает автоматизированный тест непосредственно под слоем пользовательского интерфейса. В приложении MVC это будут тесты для всего, что находится непосредственно под контроллером. Для веб-службы все, что находится ниже ендпоинта (endpoint).

Идея состоит в том, что самый верхний уровень в приложении не выполняет никакой реальной бизнес-логики, а просто соединяет внешние интерфейсы с базовыми службами. Как подчеркивает Martin Fowler в своем сообщении о подкожном тестировании, такие тесты особенно полезны при попытке выполнить функциональные тесты, когда вы хотите реализовать сквозной сценарий, избегая при этом некоторых трудностей, связанных с тестированием через сам пользовательский интерфейс:

- UI-тесты медленные. От этого никуда не деться. Их можно запускать параллельно, допиливать напильником и делать чуть-чуть быстрее, но они останутся медленными;
- UI-тесты нестабильные. Отчасти потому, что они медленные. А еще потому, что Web-браузер и интерфейс пользователя не были созданы для того, чтобы ими управлял компьютер (в настоящее время данный тренд меняется, но не факт, что это хорошо);
- UI-тесты - это наиболее сложные тесты в написании и поддержке. Они просто тестируют слишком много. (Это усиливается тем фактом, что, зачастую, люди берут «ручные» тест-кейсы и начинают их автоматизировать как есть, без учета разницы в ручном и автоматическом тестировании);
- Нам говорят, что, якобы, UI-тесты эмулируют реального пользователя. Это не так. Пользователь не будет искать элемент на странице по ID или XPath локатору. Пользователь не заполняет форму со скоростью света, и не «упадет» если какой-то элемент страницы не будет доступен в какую-то конкретную миллисекунду. И даже теперь, когда браузеры разрабатываются с учетом того, что браузером можно программно управлять - это всего-лишь эмуляция, даже если очень хорошая;
- Кто-то скажет, что некоторый функционал просто нельзя протестировать иначе. Я скажу, что если есть функционал, который можно протестировать только UI тестами (за исключением самой UI логики) - это может быть хорошим признаком архитектурных проблем в продукте.

Таким образом определение можно сформулировать так: “Если модульный тест тестирует наименьший тестируемый компонент кода приложения, то подкожный тест представляет собой единый рабочий процесс (workflow), который можно идентифицировать и протестировать в коде приложения. Подкожное тестирование рассматривает рабочий процесс как тестируемую единицу”.

Стоит помнить, что это не прямая замена автоматизации тестирования через UI, а просто более целенаправленное тестирование функциональности на нужном уровне приложения. Это позволяет команде создавать более совершенные сквозные тесты с использованием существующего кода, фреймворка и / или библиотек, доступных для внешнего приложения. Основная цель этого вида тестирования - уменьшить нестабильность теста и сосредоточиться на функциональности. Это позволяет группе различать функциональные сбои из-за проблем с кодом и сбои приложений из-за проблем совместимости или проблем с внешними зависимостями. Поскольку подкожные тесты можно запускать с той же тестовой средой, что и модульные тесты, они не имеют доступа к внутреннему коду или вызовам API во время работы. Изоляция этих тестов и использование имитирующих инструментов, которые могут имитировать вызовы API и заглушки для различных взаимодействий служб, могут дать целенаправленную, изолированную оценку функциональности во внешнем стеке. Такое тестирование функциональности позволяет сделать любую автоматизацию через браузер и пользовательский интерфейс более легкими. Это означает, что традиционные тесты пользовательского интерфейса могут быть очень минимальными и проверять только то, что связи работают между уровнями приложения. Вместо использования дорогостоящих тестов графического интерфейса и пользовательского интерфейса для получения информации о функциональности они могут быть поверхностными проверками работоспособности или дымовыми тестами. Избегая использования автоматизации пользовательского интерфейса, утверждающей ожидание чего-то функционально работающего, тесты могут утверждать, правильно ли приложение обменивается данными на своих уровнях интеграции.

Поскольку при подкожном тестировании используются все компоненты на странице, создание интегрированных тестов, которые тестируют только несколько компонентов, или тестирование одного компонента на уровне модуля может не потребоваться. Модульные тесты могут быть нацелены на более сложную логику, а не пытаться протестировать всю логику вокруг одного компонента, тестирование базовой логики облегчает нагрузку на модульное тестирование. Подкожное тестирование использует ту же структуру тестирования (например, Jest), что и модульные тесты. Это сохраняет функциональные тесты внутренними и ближе к коду, что дает команде больше шансов на более быструю обратную связь и более быструю настройку теста, чем тестовая среда, поддерживаемая отдельно. Это означает, что командам не нужно выполнять дополнительную работу по сопровождению нескольких фреймворков, репозиториев, а иногда и языков для выполнения функционального тестирования пользовательского интерфейса. Теперь, когда подкожное тестирование позволяет проводить функциональное тестирование кода, а не через пользовательский интерфейс, любые тесты пользовательского интерфейса могут быть сокращены до небольшой части того, что было необходимо ранее. UI-тесты можно использовать как дымовые тесты. Таким образом они могут доказать, что приложение и все его уровни находятся в работоспособном состоянии связи.

Поскольку подкожные тесты ориентированы на поведение высокого уровня (high-level behavior), а не на дизайн, они идеально подходят для стратегий тестирования на основе сценариев (scenario-based testing strategies), таких как BDD или паттерн Testcase Class per Fixture.

К сожалению, наряду со всеми плюсами subcutaneous подхода мы можем получить и снижение покрытия (coverage), в частности glue code (Связующий код - программный код, который служит исключительно для «склеивания» разных частей кода, и при этом не реализует сам по себе никакую иную прикладную функцию). Насколько важна\существенна потеря покрытия в данном случае? Зависит от ситуации. Мы потеряли немного glue code, который может быть (а может и не быть) важным (рекомендую в качестве упражнения определить, какой код потерялся). Оправдывает ли данная потеря покрытия введения тяжеловесного тестирования на уровне UI? Это тоже зависит от ситуации. Мы можем, например:

- Добавить один UI-тест для проверки glue code, или
- Если мы не ожидаем частых изменений glue code - оставить его без автотестов, или
- Если у нас есть какой-то объем «ручного» тестирования - есть отличный шанс, что проблемы с glue code будут замечены тестировщиком, или
- Придумать что-то еще (тот же канареечный релиз, Canary deployment)

Один из способов избежать потери покрытия - “Feature Tests Model”

Источники:

- Introduction To Subcutaneous Testing
- Пишем автотесты эффективно - Subcutaneous tests + англ. версия + видеоверсия
- Subcutaneous Testing in ASP.NET Core

Разница между coupling и cohesion

Это термины из принципов разработки ПО.

Целью этапа проектирования в жизненном цикле разработки программного обеспечения является создание решения проблемы, указанной в документе SRS (Спецификация требований к программному обеспечению). Результатом этапа проектирования является проектный документ программного обеспечения (SDD).

По сути, проектирование представляет собой итеративный процесс, состоящий из двух частей. Первая часть - это концептуальный проект, который сообщает заказчику, что будет делать система. Во-вторых, это техническое проектирование, которое позволяет сборщикам систем понять, какое аппаратное и программное обеспечение необходимо для решения проблемы клиента.

Концептуальный проект системы:

- Написано простым языком, т.е. понятным для клиента языком.
- Подробное объяснение характеристик системы.
- Описывает функциональность системы.
- Это не зависит от реализации.
- Связан с документом требования.

Технический проект системы:

- Аппаратная составляющая и дизайн.
- Функциональность и иерархия программных компонентов.
- Архитектура программного обеспечения
- Сетевая архитектура
- Структура данных и поток данных.
- Компонент ввода/вывода системы.
- Показывает интерфейс.

Модуляризация: модульность - это процесс разделения программной системы на несколько независимых модулей, где каждый модуль работает независимо. Есть много преимуществ модуляризации в разработке программного обеспечения. Некоторые из них приведены ниже:

- Легко понять систему.
- Обслуживание системы простое.
- Модуль может использоваться много раз в соответствии с их требованиями. Нет необходимости писать это снова и снова.

Связь (Coupling): Связь является мерой степени взаимозависимости между модулями. Хорошее программное обеспечение будет иметь низкую связность.

Типы связности:

- Связность данных (**Data Coupling**): если зависимость между модулями основана на том факте, что они обмениваются данными, передавая только данные, то говорят, что модули связаны данными. При соединении данных компоненты независимы друг от друга и взаимодействуют через данные. Коммуникации модулей не содержат бродячих данных (?tramp data). Пример - система расчетов с клиентами;
- Связность штампов (**Stamp Coupling**): При соединении штампов полная структура данных передается от одного модуля к другому. Следовательно, он включает в себя tramp data. Это может быть необходимо из соображений эффективности - этот выбор сделал проницательный дизайнер, а не ленивый программист;
- Связность управления (**Control Coupling**): если модули взаимодействуют, передавая управляющую информацию, то говорят, что они связаны управлением. Может быть плохо, если параметры указывают совершенно другое поведение, и хорошо, если параметры позволяют факторизовать и повторно использовать функциональность. Пример - функция сортировки, которая принимает функцию сравнения в качестве аргумента;
- Внешняя связность (**External Coupling**): при внешней связности модули зависят от других модулей, внешних по отношению к разрабатываемому программному обеспечению или аппаратному обеспечению определенного типа. Ex - протокол, внешний файл, формат устройства и т. д.
- Общая связность (**Common Coupling**): модули имеют общие данные, такие как глобальные структуры данных. Изменения в глобальных данных означают отслеживание всех модулей, которые обращаются к этим данным, для оценки эффекта изменения. Таким образом, у него есть недостатки, такие как

сложность повторного использования модулей, ограниченная способность контролировать доступ к данным и меньшая ремонтопригодность.
- Связность содержимого (**Content Coupling**): при связности содержимого один модуль может изменять данные другого модуля, или поток управления передается от одного модуля к другому модулю. Это наихудшая форма связности, и ее следует избегать.

Сцепление/единство/сплоченность (Cohesion): это мера степени, в которой элементы модуля функционально связаны. Это степень, в которой все элементы, направленные на выполнение одной задачи, содержатся в компоненте. По сути, сцепление - это внутренний клей, который удерживает модуль вместе. Хороший дизайн программного обеспечения будет иметь высокое сцепление.

Types of Cohesion:

- **Functional Cohesion**: в компоненте содержится каждый важный элемент для отдельного вычисления. Функциональная сплоченность выполняет задачу и функции. Это идеальная ситуация.
- **Sequential Cohesion**: элемент выводит некоторые данные, которые становятся входными данными для другого элемента, т. е. поток данных между частями. Это происходит естественным образом в функциональных языках программирования.
- **Communicational Cohesion**: два элемента работают с одними и теми же входными данными или вносят свой вклад в одни и те же выходные данные. Пример - обновить запись в базе данных и отправить ее на принтер.
- **Procedural Cohesion**: Элементы процедурного единства обеспечивают порядок исполнения. Действия по-прежнему слабо связаны и маловероятно, что их можно будет использовать повторно. Вычислить средний балл студента, распечатать студенческий отчет, рассчитать совокупный средний балл, распечатать совокупный средний балл.
- **Temporal Cohesion**: элементы связаны по своему времени. В модуле, связанном с временной связностью, все задачи должны выполняться в один и тот же промежуток времени. Эта связность содержит код для инициализации всех частей системы. Происходит множество различных действий, и все они происходят в единицу времени.
- **Logical Cohesion**: элементы связаны логически, а не функционально. Компонент Ex-A считывает входные данные с ленты, диска и сети. Весь код этих функций находится в одном компоненте. Операции родственные, но функции существенно различаются.
- **Coincidental Cohesion**: элементы не связаны (несвязаны). Элементы не имеют никакой концептуальной связи, кроме местоположения в исходном коде. Это случайность и худшая форма сплоченности. Вывести следующую строку и поменять местами символы строки в одном компоненте.

Источники:

- Coupling and Cohesion

Доп. материал:

- Low Coupling и High Cohesion
- ООП для ООП: GRASP

Другое (ссылки)

Всякое из сохраненного, что пока не определилось в другие темы

Тренды автоматизации тестирования

- AI/ML in Software Test Automation
- Top 10 Automation Testing Trends for 2021
- Топ тренды автоматизации тестирования, на которые следует обратить внимание в 2021 году
- Заменит ли автоматизация мануальщиков

- Может ли автоматизированное тестирование заменить ручное?

DevOps & TestOps

- A Brief Guide to Testing in DevOps
- Артем Ерошенко - TestOps: DevOps для тестировщиков
- Почему TestOps - это DevOps для тестировщиков
- Обновленный обзор Allure TestOps
- Управление тестами в TestOps: храните информацию, а не выводы
- Готовим QA-команду к внедрению DevOps: 5 советов

Test case as a code

- Артем Ерошенко - Тест-кейсы как код
- Test as Text Approach in IntelliJ IDEA
- Test case-as-code BDD, или управление ручными и автоматическими тестами

Разное

- Сколько стоит избавиться от ручного тестирования?
- When To Opt For Automation Testing?
- По следам приложения - мониторинг
- Внедряем Snapshot testing в UI-тесты iOS
- Интеграционная шина и автоматизация бп
- QA митап SuperJob
- Manual Testing Vs. Automated Testing: Which One Is The Best Fit For You?
- AUTOMATED TESTING W/ TESTRIGOR CEO ARTEM GOLUBEV & PAUL GROSSMAN
- THE GENERATION OF AUTONOMOUS AUTOMATION AND WHAT IT LOOKS LIKE (WITH BERTOLD KOLICS FROM MABL)
- How To Make End-to-End Testing Your Friend
- TEAMWORK, AI, AND CONTAINERIZATION (WITH NASA'S MICHAEL RITCHSON)
- «Автотестеры, которые не ошибаются: как это сделать»
- Автотесты на расширениях
- MVVM и MBT в контексте автоматизации UI
- You shall not pass, или Как мы настроили мониторинг тестовых окружений
- Тест дизайн методом Interface - Model - State
- Автоматическое тестирование аналитики в браузере
- Тестирование производительности приложений как часть ежедневного цикла разработки
- Паттерны и Методологии Автоматизации UI: Примеры из жизни
- Три паттерна для улучшения работы с автотестами
- Свидетели DevOps: мифы и байки про девопсов и тех, кто их нанимает
- Так как же не страдать от функциональных тестов?
- Как оценить покрытие автоматизации
- Координация автоматизированного и ручного тест-менеджмента
- THE PITFALLS OF TEST AUTOMATION
- Три паттерна для улучшения работы с автотестами
- Software Tester or Lazy Developer?
- Проверка эффективности автотестов
- Test Markup in QA Automation
- Алексей Баранцев: Десять правил построения хороших локаторов
- Did You Test the Right Things? Test Gap Analysis Can Tell You
- Using Gap Analysis In The Testing Process To Align Devs and Testers
- Семь самых распространенных ошибок при переходе на CI/CD

- Как сократить издержки на автотестах
- Меньше, чем пара. Еще один способ сокращения количества тестов