Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Баев Дмитрий Олегович

Разработка системы построения отчетов автотестов, написанных на разных языках программирования

Научный руководитель: Ерошенко Артем Михайлович, старший инженер
по автоматизации тестирования, компания Яндекс

Санкт-Петербург
2014

Содержание

Введение

Бывают модульные тесты, а бывают высокоуровневые. И когда их количество начинает расти, анализ результатов тестов становится проблемой. Дело в том, что высокоуровневые тесты сильно отличаются от модульных, и обладают рядом особенностей:

- они затрагивают гораздо больше функциональности, что затрудняет локализацию проблемы;
- такие тесты воздействуют на систему через посредников, например, браузер;
- таких тестов очень много, и зачастую приходится вводить дополнительную категоризацию. Это могут быть компоненты, области функциональности, критичность.

В рамках стандартной модели xUnit анализировать результаты таких тестов достаточно проблематично. Например, в ошибка «Can not click on element «Search Button»» тесте на web-интерфейс может произойти по следующим причинам:

- сервис не отвечает;
- на странице нет элемента «Search Button»;
- элемент «Search Button» есть, но не получается на него кликнуть.

А имея дополнительную информацию о ходе выполнения теста, например, лог работы сервиса и скриншот страницы, локализовать проблему гораздо легче.

Отсюда возникает следующая задача: разработать такую систему, которая позволяет агрегировать дополнительную информацию о ходе выполнения тестов и строить отчет.

В данной работе будет описан процесс разработки такой системы.

Во второй главе, на основании анализа различных систем построения отчетов автотестов, а также опыта написания тестирования, сформулированы основные принципы для организации системы.

В третьей главе приведена подробная архитектура Allure, позволяющая легко интегрироваться с любыми существующими тестовыми фремворками и расширять имующийся функционал. Подробно описана интеграция новых фремворков, и новых систем сборки.

В заключении дано описание текущего состояния разработки и перспективы ее развития.

Глава 1. Постановка задачи

1.1. Термины и понятия

В данном разделе описаны термины, используемые других частях представленной работы. При этом смысл многих терминов сужен, по сравнению, с их обычным смыслом. Это связано, с тем, что данная работа ориентирована в первую очередь, разработку системы построения отчетов автотестов. В дальнейшем приведенные термины будут использоваться в указанных значениях, если не оговорено обратное.

1.1.1. Тестирование

Аттачмент (attachment) — любая информация, например, скриншот или лог, которую надо сохранить вместе с результатами теста.

История (user story, story) — модуль, часть функциональности, из которых может состоять требование.

Контекст теста (test context, test fixture) — все, что нужно тестируемой системе чтобы мы могли ее протестировать. Например, наглядно понятно, что такое контекст теста в тестовом фремворке RSpec:

- контекст — множество фруктов содержащих = яблоко, апельсин, грушу;
- экспертиза — удалим апельсин из множества фруктов;
- проверка — множество фруктов содержит = яблоко, груша.

Ошибка теста (test error) — ошибка, возникающая в ходе выполнения теста. Например, ошибка может возникнуть в проверяемой системе, или в самом тесте. Также ошибка может возникнуть в окружении (например, в операционной системе, виртуальной машине). Как правило, ошибка в самом тесте, а не в проверяемой системе.

Падение теста (test failure) — тест падает, когда в проверке утверждений актуальное значение не совпадает с ожидаемым. Обычно означает наличие ошибки в проверяемой системе.

Проблемно-ориентированное проектирование (DDD) — набор принципов и схем, помогающих разработчикам создавать изящные системы объектов. При правильном применении оно приводит к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит сложная бизнес-логика, устраняющая промежуток между реальными условиями области применения продукта и кодом.

Продуктовый тест — в данной работе автор под данным термином подразумевает высокоуровневые тесты, например, интеграционные и системные.

Разработка через тестирование (TDD, test-driven development) — техника разработки программного обеспечения, которая основывается на повторении очень которких циклов:

- написание теста на новую/изменяемую функциональность;
- имплементация функциональности. Тест должен пройти;
- рефакторинг кода под соответствующие стандарты разработки.

Разработка через требования (BDD, behavior-driven development) — Разновидность разработки через тестирование, сфокусированная на тестах в которых четко описаны ожидаемые требования к тестируемой системе. Упор делается на то, что тесты используются как документация работы системы.

Результат теста (test result) — тест, или тест суит могут быт ьзапущены несколько раз, и каждый раз возращать различные результаты проверок.

Тест — некоторая процедура, котороая может быть выполена вручную или автоматически, и может быть использована для проверки ожидаемых требований к тестируемой системы. Тест часто называют тесткейсом.

Тест кейс (test case) — обычно синоним для понятия "тест". В xUnit это также может обозначать тестовый класс, как мectex boldto в которое содержит тестовые методы.

Тест прошел (test success) — ситуация, в которой проверка каждого утверждения в тесте прошла успешна (актуальные значения совпали с ожидаемыми), и в процессе выполенения теста не произошло никаких ошибок теста.

Тест ран (test run) — запуск некоторого числа тестов или тестсуитов. После выполнения тестов из тестрана, мы можем получить их результаты.

Тест суит (test suite) — способ наименования некоторого числа тестов, которые могут быть запущены вместе.

Тестируемая система (System Under Test) — любая вещь, которую мы проверяем, например, метод, класс, объект, приложение.

Требование (feature) — часть функциональности развивающейся системы, которая может быть протестирована.

Шаг (step) — некоторая логическая часть теста. Каждый тест может состоять из одного или нескольких шагов. Как правило, шаги отображают сценарий теста.

Шаг теста (test step) — смотри "Шаг".

Экстремальное программирование (XP) — одна из гибких методологий разработки программного обеспечения

xUnit — под этим термином подразумевается любой член семейства инфраструктур автоматизации тестов (Test Automation Framework), применяемых для автоматизации созданных вручную сценариев тестов. Для большинства современных языков программирования существует как минимум одна реализация xUnit. Обычно для автоматизации применяется тот же язык, который использовался для написания тестируемой ситстемы. Хотя это не всегда так, использовать подобную стратегию проще, поскольку тесты легко получают доступ к программному интерфейсу тестируемой

системы.

WebDriver — утилита, позволяющая эмулировать действия пользователя в различных браузерах.

Большинство членов xUnit реализованы с использованием объектно-ориентированной парадигмы.

1.1.2. Сокращения

SUT — System Under Test, смотри "Тестируемая система".

1.2. Проблематика

В современном мире развитие идет очень быстро. Требования к продуктам часто меняются, и надо уметь успевать за этими изменениями. Для этого, в частности, важно сокращение длительности релизного цикла программ. И последнее время все чаще узким местом является тестирование. Для того, чтобы ускорить процесс тестирования, надо ускорить выполнение тестов, и сократить время анализа результатов тестирования. В данной работе рассматривается инструмент, который помогает решить вторую задачу - ускорение анализа результатов тестирования. Но обо всем по порядку.

1.2.1. JUnit

Последние 12 лет тесты пишутся с использованием фреймворков xUnit, в частности JUnit (в дальнейшем будет рассматриваться именно JUnit, как основа фремворков xUnit). JUnit предоставляет систему для запуска тестов, также предоставляет отчет для анализа результатов. Фремворк был разработан Кент Беком (Kent Beck), автором таких методологий разработки ПО как экстремальное программирование (XP) и разработка через тестирование (TDD), в 2002 году. Данный фремворк ориентирован прежде всего на написание модульных тестов, однако последнее время сильно увеличилось количество функциональных тестов. Это

связано, прежде всего, с сильным развитием интерфейсов (в частности, web-интерфейсов). И в случае функциональных тестов данный фремворк предоставляет мало информации. Решением данной проблемы являлось появления методологии разработки через требования.

1.2.2. Разработка через тестирование

Методология разработки через тестирования комбинирует в себе основные техники и практики из TDD с идеями из DDD и объектно-ориентированным проектированием. В данном подходе основная задача ставится в описании требований (спецификаций) к тестируемой системе и дальнейшей проверки системы на удовлетворение этим требованиям.

В скором времени начали появлятся фремворки, основанные на BDD. Как правило, в данных фремворках идет абстрагирование от кода тестов, и вынесение спецификаций к фремворку на уровень описания. Например, следующим образом выглыдит спецификация в JBehave:

```
Given a 5 by 5 game
When I toggle the cell at (2, 3)
Then the grid should look like
.....
.....
.....
..X..
.....
When I toggle the cell at (2, 4)
Then the grid should look like
.....
.....
.....
..X..
..X..
When I toggle the cell at (2, 3)
Then the grid should look like
.....
.....
.....
.....
..X..
```

Именно идеи BDD были взяты в основу разработаного автором фремворка, получившего название Allure.

1.2.3. Allure

Первым, и самым важным отличием разработываемого фремворка было то, что он не выполняет тесты, а просто собирает информацию о ходе их выполнения. Также, разрабатываемый фремворк должен уметь предоставлять результаты как в виде BDD, так и в виде xUnit. Еще одной важной идеей было то, что отчет должен быть простым и понятным каждому. Это позволит ввести дополнительный уровень контроля над тестировщиками. В больших компаниях часто возникает проблема, когда тестировщик не в полной мере ответственно подходит к анализу результатов. А другому человеку будет сложно понять, что же конкретно тестируется, не разбираясь в коде тестов.

Взяв за основу данные идеи было проведено исследование, которое позволило сформулировать более подробные требования к разрабатываемой системе.

Глава 2. Исследование

В данной главе рассматривается теоритические аспекты разработки фремворка. Проводится исследование существующих систем, описываются требования к разрабатываемой системе.

2.1. JUnit

Рассмотрим более подробно тестовый фреймворк JUnit. Данный фреймворк впервые показал, как надо устраивать процесс тестирования (на самом деле, основные идеи были сформированы Кент Беком при разработке SUnit, но именно в лице JUnit эти идеи получили широкое распространение):

- тесты представляют из себя набор проверок утверждений;
- тесты могут быть сгруппированы в суиты, для совместного запуска;
- суиты объеденяются в тест ран.

Для семейства xUnit существует стандартный отчет surefire. В приложении [номер] можно посмотреть на разные виды surefire-отчета. Ниже можно посмотреть пример простейщего JUnit теста.

```
public class SampleTest {

    @Test
    public void sampleTest() throws Exception {
        assertThat(4, is(2 + 2));
    }
}
```

Основным недостатком xUnit является атомарность теста, невозможность отобразить тестовый сценарий. Основаный на JUnit тестовый фремворк Thucydides решает эту проблему путем разбиения тестов на шаги.

2.2. Thucydides

Thucydides это фремворк для написание тестов на веб-интерфейс с использованием webDriver, написанный Джоном Смартом (John Ferguson Smart). Джон Смарт — специалист в BDD, в оптимизации жизненного цикла процесса разработки. Хорошо известеный спикер множества интернациональных конференций, автор множества статей.

В свое время данный фреймворк произвел революцию. Прежде всего, фремворк предлагал структуру для тестов на веб-интерфейс, концепцию разбиения тестов на шаги и возможность сохранять скриншоты каждого шага. Шагом теста являлся любой метод, проаннотированный аннотацией @Step. Фремворк парсил структуру данных методов и отображал информацию о них в отчете. Мало того, это помогало сильно сократить код тестов — шаги выносились в отдельные библиотеки и переиспользовались в множестве проектов.

Отчет, который строил Thucydides для тестов, могли посмотреть другие люди, и понять, что происходит в тесте. В Яндексе это позволило разделить тестировщиков на "автоматизаторов"и ответственных за релиз. Первые писали тесты, а вторые эти тесты запускали, просматривали результаты и, в случае необходимости, дополнительно проводили ручное тестирование продукта. Это позволило сильно увеличить качество тестирования за счет появления дополнительного уровня качества тестов.

Также важной фозможностью Thucydides являлось сохранение скриншотов. В тестах на веб-интерфейс очень важно иметь возможно увидеть, в чем проблема. Однако, не всегда хватает возможности сохранения только скриншотов. Хотелось бы уметь приклеплять к отчету и другие типы данных, например, логи.

Однако, есть в данном фреймворке и недостатки. Прежде всего, предложеная структура была слишком жесткой. С использованием Thucydides можно писать тесты на веб-интерфейс, но большая часть функциональных тестов тестирует не его, а, например, API.

Еще одним важным недостатком было ограничение в технологиях: только JUnit, только Maven. Наверное, в любой большой компании тесты пишутся на разных языках программирования, в зависимости от специфики поставленной задачи.

Кроме того, Thucydides пытался все написать сам, со временем превращаясь в большой проект с множеством проблем.

2.2.1. Итого

Thucydides имеет следующие плюсы:

- задается единая структура тестов;
- разбиение тестов на шаги, отображение сценариев тестов;
- сохранение скриншотов;
- хороший отчет, с возможностью группировки тестов по требованиям.

Но минусов тоже много:

- слишком много ограничений, заданных структурой тестов;
- слишком много ограничений на структуру шагов;
- невозможность использовать другие типы аттачментов;
- большое количество проблем и ошибок;
- отчет понятен только тестировщикам.

Вывод: отличный фремворк, если речь идет только о простом тестировании веб-интерфейсов с использованием JUnit и Maven. В иных случаях не подходит.

2.3. Уточненные требования к работе

Основываясь на анализе существующих решений, были выработаны уточненные требования к фремворку Allure:

- умение оперировать как в терминах xUnit, так и в терминах BDD;

- отображение сценария теста, разбиение теста на шаги;
- возможность приклеплять к результатам теста произвольные данные;
- независимость от стека используемых технологий;
- простой и понятный отчет, который смогут смотреть не только разработчики тестов.

Глава 3. Реализация Allure Framework

3.1. Разработка Allure Framework

В этой главе можно узнать про процесс разработки фремворка Allure.

3.1.1. Первые шаги

После исследования была поставлена задача написать первый прототип. Решено было писать прототип для связки JUnit + Maven, так как именно эти технологии в основном использовались в компании Яндекс.

Проект был поделен на два модуля: адаптер для JUnit, который собирал данные о ходе теста, и плагин для Maven, который генерировал по этим данным отчет. Адаптер представлял из себя две JUnit рулы, одна собирала информацию о тесте, другая о тест суите. Для того, чтобы начать собирать информацию о тесте, необходимо было добавить в код теста:

```
@ClassRule
public static TestSuiteReportRule testSuite = new TestSuiteReportRule();

@Rule
public TestCaseReportRule testCase = new TestCaseReportRule(testSuite, this);
```

Генерация отчета происходила с помощью шаблонизатора freemarker.

Реализация шагов и аттачментов в прототипе была достаточно сложной — использовалась библиотека cglib, которая накладывала много ограничений на методы шагов и методы сохранения аттачментов.

Прототип был написан в сентябре 2013 года, начал внедряться в некоторые новые тестовые проекты в компании Яндекс.

3.1.2. Подключение прототипа к существующим проектам

Прототип уже использовался в некоторых новых тестовых проектах, но подключение к существующим тестам все еще было большой проблемой. Ведь для того, чтобы добавить в каждый тест (а в небольших тестовых проектах их около 500) две строчки надо было затратить существенное количество времени.

Тогда было решено воспользоваться инструментацией кода. Инструментация — это некоторое изменение байт-кода программы. Для решения этой задачи выбор пал на фреймворк ASM OW2. Автором работы был написан Maven плагин, который после компиляции проекта во все тесты инжектировал Allure рулы. Тем самым появилась возможность генерировать отчет для больших уже написанных проектов.

3.1.3. PyTest

Сразу после окончания разработки прототипа поступил запрос от команды тестировщиков, которые писали тесты на Python с использованием PyTest. Дело в том, что для языка программирования python, а в частности фремворка PyTest не было возможности построить отчет, кроме стандартного surefire. Но возможностей surefire тестировщикам не хватало, и большую часть времени тестирования занимал анализ логов тестов.

С этого момента начался следующий цикл разработки Allure. Автор данной работы начал пытаться адаптировать текущий прототип под Python. Произошли существенные изменения в модели — стало понятно, что большинство логики JUnit-адаптера будет дублироваться в PyTest-адаптере. Было решено разделить модель на два уровня. Первый уровень должен содержать только несентезируемые, чистые данные, а второй — содержать данные в удобном для отображения формате. Появился новый модуль, получивший название Report Generator (генератор отчета). В данный модуль была вынесена общая логика из JUnit и PyTest адаптеров.

Следующим этапом было написание Jenkins плагина. Дело в том,

что тесты на Python не используют Maven в своем жизненном цикле. В конкретно нашем случае они запускались с использованием Jenkins. Чтобы не устанавливать Maven на виртуальные машины, на которых запускались тесты, было решено написать плагин для Jenkins.

3.1.4. AspectJ

Несмотря на то, что фремворк уже начали использовать в некоторых проектах, большая часть проектов отказывалась подлючать Allure из-за сложностей с шагами. Дело в том, что шаги определялись очень сложным образом, и перевести существующие библиотеки шагов на предлагаемый способ подключения не предоставлялось возможным. Тогда было решено использовать фремворк AspectJ для сбора информации о пройденых шагах. Данный фремворк позволяет встроить некоторый код в байт-код загружаемого ClassLoader'ом класса. Притом, не надо думать об устройстве и структуре байт кода, достаточно просто описать точки входа (pointcuts) и аспекты (aspects):

```
@Pointcut("@annotation(ru.yandex.qatools.allure.annotations.Step)")
public void withStepAnnotation() {
    //pointcut body, should be empty
}

@Pointcut("execution(* *(..))")
public void anyMethod() {
    //pointcut body, should be empty
}

@Before("anyMethod() && withStepAnnotation()")
public void stepStart(JoinPoint joinPoint) {
    ...
}

@AfterThrowing(pointcut = "anyMethod() && withStepAnnotation()", throwing = "e")
public void stepFailed(JoinPoint joinPoint, Throwable e) {
    ...
}

@AfterReturning(pointcut = "anyMethod() && withStepAnnotation()", returning = "result")
public void stepStop(JoinPoint joinPoint, Object result) {
    ...
}
```

В итоге для добавления шага надо проаннатировать метод аннотацией @Step.

3.1.5. Примеры работы фреймворка Allure для JUnit тестов

Уже на данном этапе одно из основных достоинств разработаного автором фреймворка является прозрачная интеграция с существующими тестовыми системами. Рассмотрим простейший JUnit тест:

```
public class SimpleTest {

    @Test
    public void simpleTest() throws Exception {
        assertThat(4, is(2 + 2));
    }

    public int sum(int a, int b) {
                return a + b;
        }

    public void check(int a, int b, int c) {
        assertThat(c, is(a + b));
    }
}
```

Для данного теста уже можно построить отчет. Достаточно воспользоваться одним из инструментов для генерации отчета.

Чтобы отобразить информацию о тестовом сценарии достаточно проаннотировать соответствующие методы аннотацией @Step.

```
public class SimpleTest {

    @Test
    public void simpleTest() throws Exception {
        int c = sum(2, 2);
        check(2, 2, c);
    }

    @Step("Считаем сумму'{0}' и'{1}'")
    public int sum(int a, int b) {
                return a + b;
        }

    @Step("Проверяем, чтосумма '{0}' и'{1}' равна'{c}'")
    public void check(int a, int b, int c) {
        assertThat(c, is(a + b));
    }
}
```

Так же просто мы может добавлять к тесту аттачменты, указывать параметры, группировать тесты по требованиям и историям, и так далее.

3.1.6. TestNG

В качестве эксперимента автором работы был написан адаптер для TestNG. то второй поддерживаемый тестовый фреймворк для Java, написание которого показало необходимость в новом слое абстракции, API для языка программирования. После этого код самих адаптеров сильно упростился. Также пропала необходимость использовать тестовые фремворки для проверок. Отчет можно построить по результатам выполнения любого кода, достаточно лишь определить, что является тестом, а что проверкой.

3.1.7. Report Face

Как только фреймворк начал набирать популярность, стало появляться все больше требований к самому отчету. Было решено заменить freemarker на AngularJS и сделать отчет в виде One-Page-Application.

3.1.8. Report Face 2.0

Автор данной работы на момент написания работы не является профессиональным Java Script разработчком, из-за чего возникало множество проблем в отчете. Поэтому к разработке присоеденился Борис Сердюк, который полностью переработал структуру отчета, и воплотил в жизнь все задумки.

3.1.9. Alluredides

В отделе тестирования Яндекс разрабатываемый автором фремворк настолько понравился тестировщикам, что ими был написан инструмент, позволяющий мигрировать тесты с Thucydides на Allure, который получил название Alluredides

3.1.10. Остальные фреймворки

По мере развития Allure появлялась поддержка новых фреймворков и способов построения отчета:

- TestNG — адаптер для данного языка был написан автором данной работы.
- RSpec – адаптер написан Ильей Садыковым.
- PHPUnit — адаптер написан Иваном Крутовым.
- ScalaTest — адаптер написан Иваном Крутовым.
- Karma — адаптер написал Борис Сердюк, данный адаптер позволяет строить отчет для тестов, спользующих Karma, например, Jasmine-тестов.
- TeamCity Plugin — плагин написан Иваном Крутовым. Добавляет возможность строить отчет в TeanCity.
- Command Line Interface —написан Иваном Крутовым совместно с автором данной работы. Позволяет строить отчет используя командную строку.

3.1.11. Report Generation API

На данный момент в связи с появлением большого количества инструментов, позволяющих генерировать отчет, разрабатывается библиотека, генерирующая отчет. В ней будут описаны все методы, которые нужны для построения отчета.

3.2. Общая схема работы

После разработки прототипа было еще много изменений в структуре проекта. Весь код переписывался, четыре раза. На данный момент автор работает над версией 1.4. В данном разделе описывается текущее состояние фреймворка.

Общая схема работы Allure показана на рисунке 3.1. Рассмотрим подробнее назначение отдельных частей.

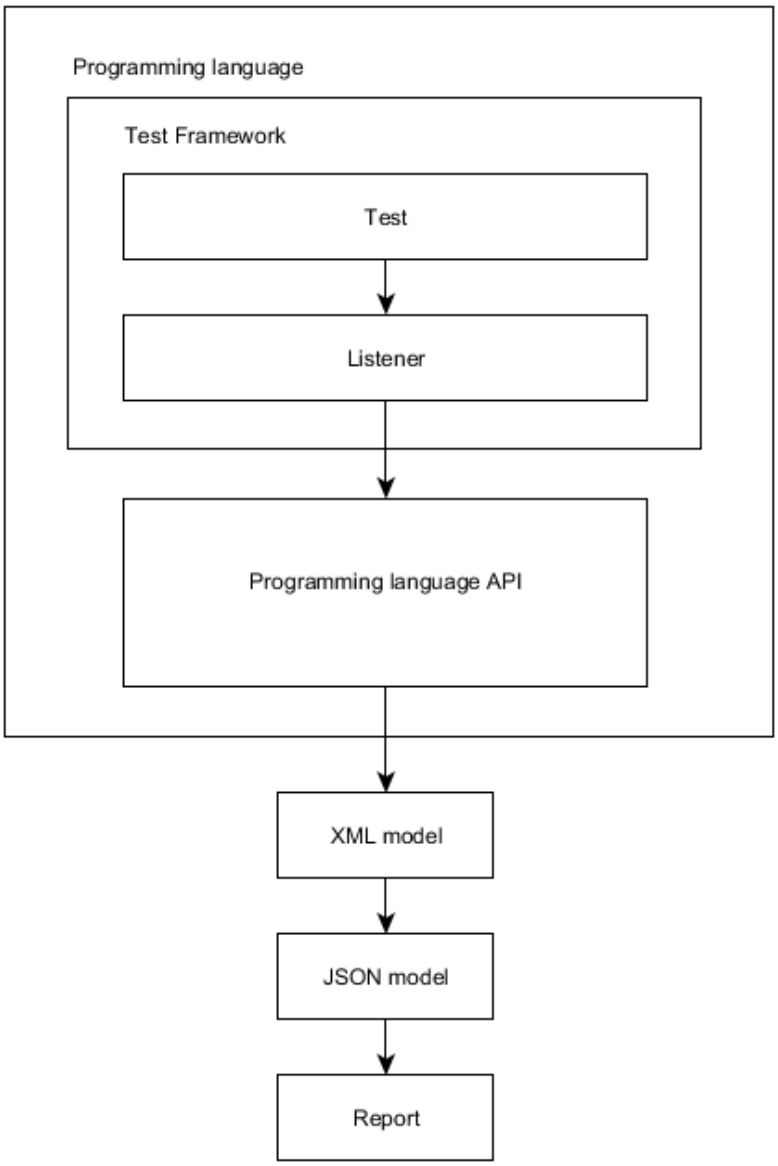

Рис. 3.1: Общая схема фремворка Allure

3.2.1. Listener

Для большинства тестовых фремворков xUnit есть возможность подключить листенер для сбора информации о ходе тестов. Мало того, подключение листенера, как правило, вынесено на уровень конфигурации запуска, что полностью удавлетворяет требованиям работы. Для адаптации тестового фремворка достаточно реализовать тест листенер используя соответсвующее API языка программирования.

Однако стоит заметить, что не всю необходимую информацию о ходе теста можно собрать используя листенер, так как он оперирует терминоло-

гией xUnit. Сбор остальной информации о тестах, например информацию о пройденных шагах и сделанных аттачментах, будет реализован на уровне API языка программирования.

3.2.2. Programming language API

API для языка программирования представляет из себя набор обработчиков событий и сами события, используя которые можно полностью описать жизненный цикл теста. Программный инетрфейс содержит в себе следующие события:

- начало/конец тестового запуска;
- начало/конец тест суита;
- начало/конец тест кейса;
- начало/конец шага;
- сохранение аттачмента;
- добавление параметров запуска/тест суита/тест кейса;
- изменение статуса теста/шага;
- добавление пометок к тесту.

С использованием API для языка программирования сильно упрощается написание и поддержка листнеров для тестовых фремворков. Вся собранная информация о ходе тестов сохраняется в XML модель.

3.2.3. XML model

Собранная о тесте информация серелизуется в виде XML файлов. Для каждого теста создается свой файл. Сохраняются только те данные, которые нельзя синтезировать, что упрощает реализацию и поддержку интерфейса для языка программирования. Простейший пример сохранненной информации об одном тесте:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:test-suite xmlns:ns2="urn:model.allure.qatools.yandex.ru" start="1400681607876"
    stop="1400681627123">
```

```
    <name>my.company.SampleTest</name>
    <test-cases>
        <test-case start="1400681608883" stop="1400681608891" status="passed">
            <name>test_pass</name>
        </test-case>
    </test-cases>
    <labels/>
</ns2:test-suite>
```

3.2.4. JSON model

На следующем этапе данные конвертируются в более удобный для обоажения формат. Например, данные заранее группируются для различных отображений в отчете (xUnit, BDD, Defects). Также считается статистическая информация и генерируются данные для графиков.

3.2.5. Report

Отображает результаты разными способами.

3.2.5.1. xUnit

В данном табе используется терминология xUnit. Сначала тесты группируются по тест суитам, затем по самим тестам. Также присутствует возможность сортировать тесты по имени, важности, статусу и времени выполнения.

3.2.5.2. BDD

Данный таб используется для отображения результатов проверки требований к продукту. Сначала тесты группируются по требованию, потом по истории. Позволяет сразу понять, какие требования нарушаются в тестируемом продукте.

3.2.5.3. Defects

Группирует тесты по тексту сообщения. Также разбивает все ошибки на две группы — продуктовые дефекты и ошибки тестов.

3.2.5.4. Timeline

Отображает ход выполнения тестов с течением времени. Помогает находить ошибки типа "вчера в 15:30 сервис не работал".

Заключение

В ходе работы поставленные задачи были достигнуты. Allure фреймворк стал незаменимой частью отдела тестирования компании Яндекс, и на данный момент активно продолжает развиваться.

Allure был адаптирован под основные тестовые фреймворки, которые используются для функционального тестирования, за исключением C sharp, и под основные системы выполнения тестов.

Была разработана уникальная архитектура, позволяющая писать самодокументирующиеся тесты.

В ходе работы автор изучил следующие технологии:

- Инструментирование кода: cglib, Spring Aspects, aspectJ, OW2 ASM.
- Написание плагинов: Maven, Jenkins, TeamCity.
- Java SPI.
- JUnit (автор работы является контрибутором).
- Java шаблонизатор Freemarker.
- XSLT, XQuery, XSD, JAXB.
- написание Command Line Interface.
- AngularJS, Jasmine.
- Непрерывная интеграция на примере Github + TeamCity.