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

Процесс проектирования и разработки больших программных систем зачастую весьма сложен и тестно связан с процессом контроля его качества. Один из способов контроля качества программного обеспечения это тестирование. Тестирование — это процесс исследования, испытания программного продукта, имеющий две различные цели:

- продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
- выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Часто процесс тестирования автоматизируют. Это полезно, например, для регрессионного тестирования. Регрессионное тестирование — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Написание автотестов в таком случае помогает избежать многочисленных проверок той же функциоанльности в будущем. Однако с процессом развития программной системы количество таких тестов может сильно возрасти.

Есть несколько уровней тестирования:

- Модульное тестирование (юнит-тестирование) — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто модульное тестирование осуществляется разработчиками ПО.
- Интеграционное тестирование — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.

- Системное тестирование — тестируется интегрированная система на ее соответствие требованиям.

Большие программные системы для обеспечения высокого уровня качества зачастую проходят несколько этапов тестирования. А сложность высокоуровневых тестов на такие системы может быть сравнима со сложностью тестируемых систем. 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.

Процесс тестирования состоит из двух этапов: непосредственно проведение тестов и анализ результатов. Если речь идет о модульном тестировании программных систем, то анализ результатов занимает незначительное время. Но если говорить о функциональном тестировании, то это не так. Функциоанльное тестирование программных систем — достаточно сложная задача. И часто существенную часть времени тестировщика занимает именно анализ результатов тестирования. В рамках данной работы разработывается система, позволяющая сократить время анализа результатов тестирования программных систем.

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

В третьей главе приведена подробная архитектура Allure, позволяющая легко интегрироваться с любыми существующими тестовыми фреймворками и расширять имующийся функционал. Подробно описана интеграция новых фреймворков, 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. Программирование

Веб-браузер, браузер (web browser) — программное обеспечение для просмотра веб-сайтов.

Веб-интерфейс (web-interface) — это совокупность средств, при помощи которых пользователь взаимодействует с веб-сайтом или любым другим приложением через браузер. Веб-интерфейсы получили широкое распространение в связи с ростом популярности всемирной паутины и соответственно — повсеместного распространения веб-браузеров.

Программный интерфейс (application programming interface, API) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

Фреймворк (framework) — структура программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

1.1.2. 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 фреймворк чтобы мы могли ее протестировать. Например, наглядно понятно, что такое контекст теста в тестовом фреймворке 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 быть запущены несколько раз, и каждый раз возвращать 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 mectex 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.

Тестовый фреймворк (testing framework) — фреймворк, который используется для написания и запуска тестов.

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.

1.1.3. 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 в 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 [1] (в дальнейшем будет рассматриваться именно JUnit, как основа фреймворков xUnit). JUnit предоставляет систему для запуска тестов, также предоставляет отчет для анализа результатов. фреймворк был разработан Кент Беком (Kent Beck [2]), автором таких методологий разработки ПО как экстремальное программирование (XP) и разработка через тестирование (TDD), в 2002 году. Данный фреймворк 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 фреймворк 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 появляться фреймворки, основанные на BDD. Как правило, в данных фреймворках идет абстрагирование от кода тестов, и вынесение спецификаций к фреймворку на уровень описания.

Например, следующим образом выглыдит спецификация в JBehave [3]:

Листинг 1.2.1: Пример спецификации 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 фреймворка, получившего название Allure.

1.2.3. Allure

Первым, и самым важным отличием разработываемого фреймворка было то, что он не выполняет тесты, а просто собирает информацию о ходе их выполнения. Также, разрабатываемый фреймворк 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. Обзор существующих систем

На момент написания автором работы, систем, решающих поставленную задачу не было. Однако, есть системы, которые решают эту задачу частично. В данной главе рассматриваются такие системы.

2.1. JUnit

JUnit — тестовый фреймворк, для написания автотестов на языке программирования Java. Данный фремворк прежде всего ориентирован на написание модульных тестов. JUnit 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) [4]:

- тесты представляют из себя набор проверок утверждений;
- тесты могут быть сгруппированы в суиты, для совместного запуска;
- у каждого теста или суита может быть подготовка (set up) или завершение (tear down);
- суиты объеденяются в тест ран.

Ниже можно посмотреть пример простейщего JUnit теста.

Листинг 2.1.1: Простой JUnit тест.

```
public class SampleTest {фреймворк

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

Для семейства xUnit существует стандартный отчет Surefire [5]. Есть несколько разных видов этого отчета, но суть у всех одна: отображается список всех тестов, у каждого теста есть статус, время выполнения и сообщение об об ошибке (в случае, если тест не прошел). Для большинства

современных языков программирования существует своя имплементация xUnit фреймворка.

Основным недостатком xUnit является атомарность теста, невозможность отобразить тестовый сценарий. Основаный на JUnit тестовый фреймворк Thucydides [6] решает эту проблему путем разбиения тестов на шаги.

2.2. Thucydides

Thucydides — это фреймворк 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 [7].

В свое время данный фреймворк произвел революцию. Прежде всего, фреймворк предлагает 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. Фреймворк анализирует структуру данных методов и отображает информацию о них в отчете. Мало того, это помогало сильно сократить код тестов — появляется возможность вынести шаги в отдельные библиотеки и переиспользовались в множестве проектов [8].

Отчет, который строит 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 контроля тестировщиков.

Еще одной важной возможностью Thucydides является 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.

Не обошлось в данном фреймворке без недостатков. Прежде всего, предложеная структура слишком ограничивает возможности тестирования. С использованием Thucydides можно писать тесты на веб-интерфейс, а это лишь малая часть функционального тестирования. Для тестов, например, на API данный фреймворк не подходит.

Также важным недостатком является ограничение в используемых технологиях: тесты можно писать используя только тестовый фреймворк JUnit и только систему сборки Maven, не смотря на то, что в любой большой компании тесты пишутся на разных языках программирования, в зависимости от специфики поставленной задачи.

Кроме того, Thucydides предлагает очень много возможностей, из-за он чего уже давно превратился в большой, сложный фреймворк с множеством ошибок и проблем. Более правильным решением было бы сделать модульный проект, с возможностью подмены некоторых компонент.

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 фреймворк, 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 фреймворку 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.

В следующей главе рассказывается о процессе разработки фреймворка, удовлетворяющего данным требованиям, и о проблемах, с которыми пришлось столкнуться автору данной работы в процессе разработки. Также описывается текущая архитектура проекта.

Глава 3. Разработка

3.1. Разработка Allure Framework

В этой главе можно узнать про процесс разработки фреймворка Allure [9].

3.1.1. Первые шаги

После исследования была поставлена задача написать первый прототип. В первую очередь прототип должен работать с тестовым фреймворком JUnit и системой сборки Maven [10], так как именно эти технологии по большей части использовались в компании Яндекс.

Проект поделен на два модуля: адаптер для JUnit, который собирает данные о ходе теста, и плагин для Maven, который генерирует по этим данным отчет. Адаптер использует механизм JUnit Rules. Для того, чтобы начать собирать информацию о тесте, необходимо добавить в код теста следующие строки:

Листинг 3.1.1: Пример подключения к тестам JUnit Rules.

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

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

Генерация отчета происходит с помощью шаблонизатора freemarker. Freemarker — технология, которая позволяет генерировать любые текстовые данные, основаные на шаблонах. Прежде всего, данный инструмент ориентирован на написание фреймворкпростых HTML-страниц, основаных на MVC шаблоне [11].

Реализация шагов и аттачментов в прототипе достаточно сложная — используется технология MethodInterceptor из библиотеки cglib. Данная технология позволяет подменять выполнение оригинальных методов через

наследование, тем самым накладывая множество ограничений на подменяемые методы. Например, они не могут быть приватными, статическими или объявлены как final [12].

Прототип был написан в сентябре 2013 года, сразу после чего 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. Для того, чтобы добавить в каждый тест (а в небольших тестовых проектах их около 500) две строчки надо было затратить существенное количество времени и сил.

Тогда было решено воспользоваться инструментацией кода. Инструментация — это некоторое изменение байт-кода программы. Для решения этой был использован фреймворк ASM OW2 [13]. Также написан Maven-плагин, который после компиляции проекта во все тесты инжектировал необходимые для подключения Allure поля. Тем самым появилась возможность генерировать отчет для больших проектов.

3.1.3. PyTest

Сразу после окончания разработки прототипа поступил запрос от команды тестировщиков, которые писали тесты на языке программирования Python с использованием тестового фреймворка PyTest [14]. Дело в том, что для python'а, а в частности для фреймворка PyTest на момент написания автором работы не существовало фреймворков, позволяющих строить отчет о результатах тестирования, кроме стандартного surefire. Но возможностей surefire тестировщикам не хватало, и большую часть времени тестирования занимал именно анализ результатов тестов.

С этого момента начался следующий цикл разработки Allure. Были предприняты первые попытки написать адаптер для 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 несинтезируемые, чистые данные, а второй — содержать данные в удобном для отображения формате. Также появился новый модуль, получивший название Report Generator (генератор отчета). В данный модуль была вынесена общая логика из JUnit и PyTest адаптеров.

Дальнейшим этапом развития было написание Jenkins плагина. Jenkins это проект с открытым исходным кодом, предоставляющий сервисы для непрерывной интеграции. Это очень популярный в мире инструмент. Именно Jenkins использовался как система сборки и тестирования у тестировщиков, которые писали тесты на python'е.

3.1.4. AspectJ

Несмотря на то, что фреймворк уже начали использовать в некоторых проектах, большая часть проектов отказывалась подлючать Allure из-за сложностей с подключением. 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 фреймворк AspectJ для сбора информации о пройденых шагах. Данный фреймворк 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):

Листинг 3.1.2: Пример описание точек входа и аспектов.

```
@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 одним из главных достоинств фреймворка является прозрачная интеграция с существующими тестовыми системами. Рассмотрим простейший JUnit тест:

Листинг 3.1.3: Простой 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.

Листинг 3.1.4: Простой JUnit тест с добавлением шагов.

```
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 можем добавлять к тесту аттачменты, указывать параметры, группировать тесты по требованиям и историям, и так далее.

3.1.6. TestNG

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

3.1.7. Report Face

Как только фреймворк начал набирать популярность, стало появляться все больше требований к самому отчету. Из-за сильно возросшей сложности отчета было решено заменить freemarker на AngularJS [15] и сделать его в виде Single-Page-Application [16].

3.1.8. Alluredides

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

3.1.9. Остальные фреймворки

По мере развития Allure появлялась поддержка новых фреймворков:

- TestNG
- RSpec
- PHPUnit
- ScalaTest
- Karma (Jasmine и другие)

и способов построения отчета:

- TeamCity Plugin
- Command Line Interface

3.1.10. 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. В данном разделе описывается текущее состояние фреймворка.

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

3.2.1. Listener

В большинстве случаев тесты пишутся с использованием тестового фреймворка. И чтобы собрать информацию о ходе выполнения тестов, нужно уметь взаимодействовать с этим тестовым фреймворком. Именно для этого используется механизм листнеров. В каждом конкретном случае взаимодействие происходит по-своему. Например, в случае JUnit используется RunListener:

Листинг 3.2.5: С помощью JUnit RunListener можно собирать информацию о ходе выполения теста.

```
public class RunListener {
    /**
     * Called before any tests have been run. This may be called on an
     * arbitrary thread.
     *
     * @param description describes the tests to be run
     */
    public void testRunStarted(Description description) throws Exception {
    }

    /**
```

```
     * Called when all tests have finished. This may be called on an
     * arbitrary thread.
     *
     * @param result the summary of the test run, including all the tests that failed
     */
    public void testRunFinished(Result result) throws Exception {
    }

    /**
     * Called when an atomic test is about to be started.
     *
     * @param description the description of the test that is about to be run
     * (generally a class and method name)
     */
    public void testStarted(Description description) throws Exception {
    }

    /**
     * Called when an atomic test has finished, whether the test succeeds or fails.
     *
     * @param description the description of the test that just ran
     */
    public void testFinished(Description description) throws Exception {
    }

    /**
     * Called when an atomic test fails, or when a listener throws an exception.
     *
     * <p>In the case of a failure of an atomic test, this method will be called
     * with the same {@code Description} passed to
     * {@link #testStarted(Description)}, from the same thread that called
     * {@link #testStarted(Description)}.
     *
     * <p>In the case of a listener throwing an exception, this will be called with
     * a {@code Description} of {@link Description#TEST_MECHANISM}, and may be called
     * on an arbitrary thread.
     *
     * @param failure describes the test that failed and the exception that was thrown
     */
    public void testFailure(Failure failure) throws Exception {
    }

    /**
     * Called when an atomic test flags that it assumes a condition that is
     * false
     *
     * @param failure describes the test that failed and the
     * {@link org.junit.AssumptionViolatedException} that was thrown
     */
    public void testAssumptionFailure(Failure failure) {
    }

    /**
     * Called when a test will not be run, generally because a test method is annotated
     * with {@link org.junit.Ignore}.
     *
     * @param description describes the test that will not be run
     */
    public void testIgnored(Description description) throws Exception {
    }
}
```

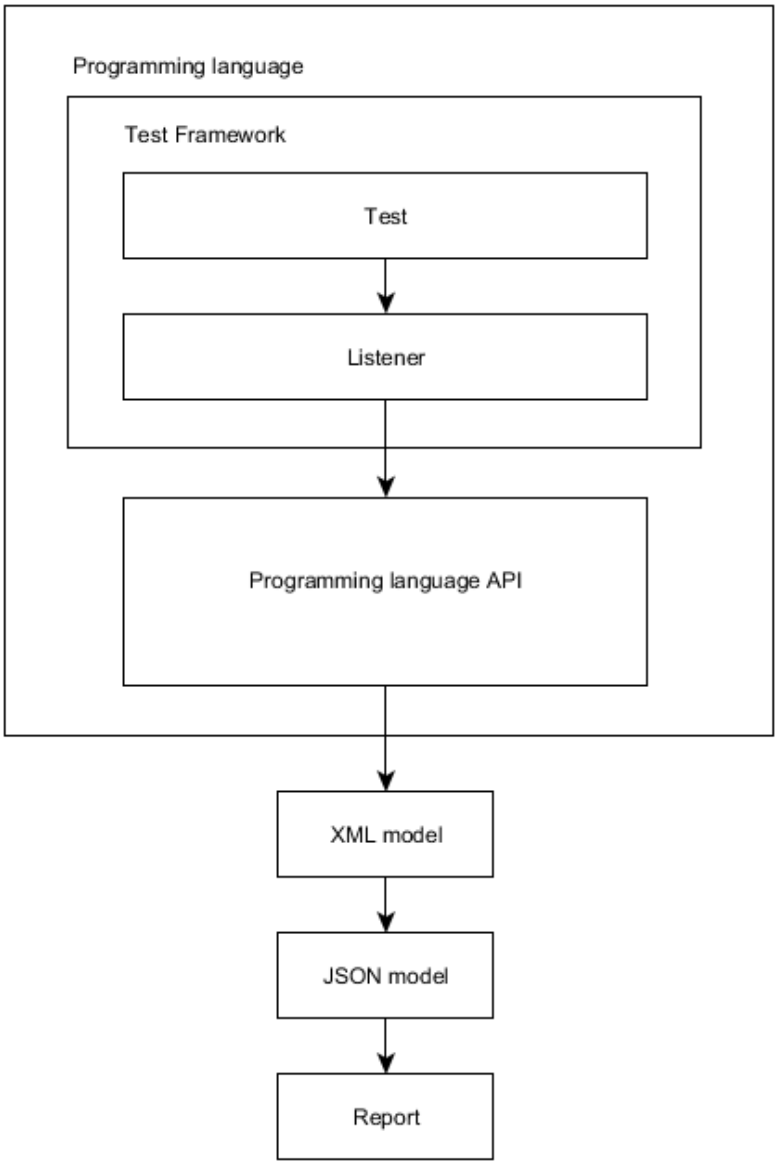

Рис. 3.1: Общая схема фреймворка Allure

Используя свою реализацию RunListener можно построить собрать основную информацию о ходе выполнения теста.

Подключение листенера, как правило, вынесено на уровень конфигурации запуска, что полностью удавлетворяет требованиям работы. Для адаптации тестового фреймворка достаточно реализовать 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 терминологией 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 интерфейс 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 фреймворков. Вся собранная информация о ходе тестов сохраняется в XML модель.

3.2.3. XML model

Собранная о тесте информация сериализуется 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:

Листинг 3.2.6: Пример простой XML-модели

```
<?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 отображения формат. Например, данные заранее группируются для различных отображений в отчете (xUnit, BDD, Defects). Также считается статистическая информация и генерируются данные для графиков.

Преборазование между моделями происходит с помощью XSLT.

3.2.5. Report

Отображает результаты разными способами:

- 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.
- BDD — данный таб используется для отображения результатов проверки требований к продукту. Сначала тесты группируются по тре-

бованию, потом по истории. Позволяет сразу понять, какие требования нарушаются в тестируемом продукте.

- Defects — группирует тесты по тексту сообщения. Также разбивает все ошибки на две группы — продуктовые дефекты и ошибки тестов.
- Timeline — отображает ход выполнения тестов с течением времени. Помогает находить ошибки типа "вчера в 15:30 сервис не работал".

Также есть различные графики, отображающие картину выполнения тестов в целом. Каждое из этих отображений результатов полезно разным людям. Менеджеру — BDD, разработчику — xUnit, тестировщику Defects. Отчет хорошо работает с большим количеством тестов (десятки тысяч).

3.3. Сравнение

Как было сказано ранее, на момент написания автором работы, систем, решающих поставленную задачу не было. Были рассмотрены системы, которые решали эту задачу частично. В данном разделе сравниваются эти системы с Allure Framework.

	Surefire	Thucydides	Allure Framework
Статус теста	Да	Да	Да
Сообщение об ошибке	Да	Да	Да
Длительность теста	Да	Да	Да
Сценарий теста	Нет	Да	Да
Возможность сохранения аттачментов	Нет	Только скриншоты	Да
Параметры теста	Нет	Нет	Да
Простое подключение	Да	Нет	Да
Типы тестов	В основном модульные	Тесты на веб-интерфейс с использованием WebDriver	Любые
BDD	Нет	Да	Да
Поддерживает основные языки программирования и тестовые фремворки	Да	Только JUnit (Java)	Да

Заключение

В ходе работы поставленные задачи были успешно решены.

- Отчет Allure может группировать результаты выполнения тестов как в терминах xUnit, так и в терминах BDD. Также есть возможность группировки по тексту ошибок.
- В отчете есть возможность отображать сценарий теста, разбивать тест на шаги.
- Еще предоставляется возможность приклеплять к результатам теста произвольные данные.
- Allure не зависит от стека используемых технологий, есть возможность использовать его с различными языками программирования, тестовыми фреймворками и системами сборки (интеграции).
- Одним из основных преимуществ отчета является прозрачное и легкое подключение к тестам.
- Получившийся отчет действительно является простым и понятным. Отчет могут смотреть как тестировщики, так и разработчики или менеджеры. Мало того, в отчете может разобраться человек, далекий от программирования.

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), TestNG, PyTest.
- Java шаблонизатор Freemarker.
- XSLT, XQuery, XSD, JAXB.
- написание Command Line Interface.
- AngularJS, Jasmine.
- Непрерывная интеграция на примере Github + TeamCity.

фреймворк

Список литературы

1. JUnit Home Page. http://junit.org/.
2. Kent Beck Wiki Page. http://en.wikipedia.org/wiki/Kent˙Beck.
3. JBehave Home Page. http://jbehave.org/.
4. *Meszaros G.* xUnit Test Patterns: Refactoring Test Code.
5. Maven Surefire Report Plugin. http://maven.apache.org/surefire/maven-surefire-report-plugin/.
6. Thucydides Home Page. http://www.thucydides.info/.
7. John's biography. http://www.wakaleo.com/about-us/about-wakaleo-consulting.
8. Thucydides Documentation Page. http://www.thucydides.info/docs/thucydides.pdf.
9. Allure Home Page. https://github.com/allure-framework/allure-core.
10. Maven Home Page. http://maven.apache.org/.
11. Freemarker Home Page. http://freemarker.org/.
12. Cglib Home Page. https://github.com/cglib/cglib.
13. OW2 ASM Home Page. http://asm.ow2.org/.
14. PyTest Home Page. http://pytest.org/latest/.
15. AngularJS Home Page. https://angularjs.org/.
16. Single Page Application Wiki. http://en.wikipedia.org/wiki/Single-page˙application.

Приложения

Листинг 0.0.7: Пример подключения Allure к JUnit тестам с использованием Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.sonatype.oss</groupId>
        <artifactId>oss-parent</artifactId>
        <version>7</version>
    </parent>

    <groupId>ru.yandex.qatools.allure</groupId>
    <artifactId>allure-junit-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <allure.version>1.3.6</allure.version>
        <aspectj.version>1.7.4</aspectj.version>

        <compiler.version>1.7</compiler.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <name>Allure Example Unsing jUnit and WebDriver</name>
    <description>Allure Example Unsing jUnit and WebDriver</description>

    <dependencies>
        <dependency>
            <groupId>ru.yandex.qatools.allure</groupId>
            <artifactId>allure-junit-adaptor</artifactId>
            <version>${allure.version}</version>
        </dependency>
        <dependency>
            <groupId>com.github.detro.ghostdriver</groupId>
            <artifactId>phantomjsdriver</artifactId>
            <version>1.0.4</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-archetype-plugin</artifactId>
                <version>2.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.0</version>
                <configuration>
                    <source>${compiler.version}</source>
                    <target>${compiler.version}</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.14</version>
                <configuration>
                    <testFailureIgnore>false</testFailureIgnore>
```

```
                    <argLine>
                        -javaagent:${settings.localRepository}/org/aspectj/aspectjweaver/
                        ${aspectj.version}/aspectjweaver-${aspectj.version}.jar
                    </argLine>
                    <properties>
                        <property>
                            <name>listener</name>
                            <value>ru.yandex.qatools.allure.junit.AllureRunListener</value>
                        </property>
                    </properties>
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjweaver</artifactId>
                        <version>${aspectj.version}</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>

    <reporting>
        <excludeDefaults>true</excludeDefaults>
        <plugins>
            <plugin>
                <groupId>ru.yandex.qatools.allure</groupId>
                <artifactId>allure-maven-plugin</artifactId>
                <version>${allure.version}</version>
            </plugin>
        </plugins>
    </reporting>

</project>
```