Правительство Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный университет»

Кафедра информационно-аналитических систем

Сапурина Лилия Юрьевна

Разработка системы автоматизированного тестирования front-end части приложения для операторов сотовой связи Мегафон

Бакалаврская работа

Научный руководитель:
К. ф.-м. н., доцент Михайлова Е. Г.

Рецензент:
ведущий инженер Степанов Д. С.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Analytical Information Systems

Lilia Sapurina

System development of automated testing application front-end for mobile MegaFon network operators

Bachelor's Thesis

Scientific supervisor:
assoc. professor Mikhailova Elena

Reviewer:
senior Developer Stepanov Dmitry

Saint-Petersburg
2016

Оглавление

Введение

Автоматизированное тестирование является неотъемлемой частью современного подхода к разработке программного обеспечения. Организация системы автоматического тестирования позволяет идентифицировать ошибки функциональности и дизайна приложения на всех этапах его создания.

Преимущества, которые предоставляет система:

- При любом изменении, внесенном программистом в код, запускается тестовый сценарий, проверяющий корректную работу прежней и новой функциональности. Таким образом, разработчик всегда уверен в том, что его действия не повлекли за собой ошибок работы приложения. А если ошибка возникла, то отчёт о прохождении тестов позволит точно идентифицировать место и причину сбоя.

- Автоматизированное тестирование значительно сокращает время разработки ПО, так как заменяет процесс ручного тестирования. Кроме того, автономные тесты способны работать для разных форм и приложений. Таким образом, один раз написанный тест может служить инструментом для поиска ошибок многократно.

- Случайный подход генерации данных, используемых в тестах, позволяет найти непредвиденные ошибки. Так как очень часто машина способна сгенерировать тестовый пример, который ручной тестировщик мог не предусмотреть.

- История прохождения тестов на разных этапах разработки даёт ценную статистическую информацию: среднее количество ошибок при разработке одной формы, прохождении одного спринта (если речь идёт о компаниях, поддерживающих гибкие методологии разработки), за всё время создания продукта; время, потраченное на исправление ошибок; типы ошибок, их процентное соотношение. Такого рода информация может быть представлена заказчику, а также активно использоваться при дальнейшей разработке,

чтобы на её основании улучшить процесс, устраняя недостатки предыдущих релизов. Кроме того, информация может быть использована при прохождении проверок компании на соответствие тому или иному мировому стандарту.

Не смотря на все перечисленные преимущества автоматизированного тестирования, в большинстве IT компаний понятие ручного тестирования не превратилось в атавизм. Это происходит из-за того, что создание подобной системы является нетривиальной и очень специфической задачей. В зависимости от функциональности разрабатываемого ПО, языков и средств программирования, задача организации системы кардинально меняется, приобретая свои персональные проблемы.

1. Постановка задачи

Целью данной работы является разработка и реализация универсальной системы автоматизированного тестирования web-приложений, написанных при помощи фреймворка AngularJS, а также корректирования общего плана процесса разработки, с учётом создания данной системы. Рассматриваться задача будет на примере приложения для операторов сотовой связи «Мегафон», разрабатываемым в компании «PETER-SERVICE».

Задача включает в себя:

1. Формулирование требований, которыми должна обладать система.
2. Выбор оптимальных фреймворков для тестирования.
3. Разработка способов расширения стандартных методов этих фреймворков.
4. Реализация системы.
5. Оценка влияния системы на процесс разработки.

На начальном этапе разработки системы были выделены следующие требования:

- максимально избежать влияния изменений в разрабатываемом продукте на работу тестов
- унифицировать разрабатываемые тесты, организовать их многократное переиспользование
- совместить процесс создания тестовых сценариев и их реализации. Разработать систему так, чтобы сценарии можно было однозначно сопоставлять с тестами
- создать библиотеки пользовательских методов, уменьшающих затраты на решение специфических проблем

- организовать оптимальную систему отчётности для выявления и определения мест возникших ошибок функциональности
- скомбинировать систему с инструментами непрерывной интеграции - Continuous Integration [2]
- автоматически генерировать документацию для каждого элемента системы

2. Обзор

В настоящий момент существует огромное количество тестовых фреймворков для автоматического тестирования, написанных для того или иного языка. Главной задачей в этой работе являлось избежать так называемого «зоопарка продуктов», который может возникнуть если для различных целей в проекте использовать совершенно различные по своей сути программные решения. Например, приложение написать на языке javascript, тестировать его на языке Python, а сборку делать при помощи скриптов на shell. Такой подход совершенно неприемлем в команде. При отсутствии сотрудника, занимающегося той или иной частью процесса, работа может остановиться. Поэтому необходимо было максимально универсализировать каждый шаг жизненного цикла ПО. Именно поэтому для написания автоматических тестов было принято решение выбрать фреймворк Protractor [13], написанный непосредственно для использования в комбинации с AngularJS. Кроме того, при таком выборе, вся команда, включая разработчиков и дизайнеров, может принимать участие в создании тестов, вносить какие-то поправки и давать советы по улучшению.

В процессе разработки системы тестирования были использованы методологии, позаимствованные из других фреймворков.
Protractor – молодой и развивающийся проект. Крупных систем или руководств по его использованию нет.
Для понятия сути разработки систем тестирования был изучен фреймворк PyTest языка Python. В данном языке хорошо развиты методологии объектно-ориентированного программирования, создания фикстур (конфигураций) и параметризации тестов. Данные методы помогли лучше осознать и определиться в направлении действий. В процессе подготовки к основной работе был создан небольшой проект на GitHub, в котором использовался фреймворк PyTest для тестирования простого web-приложения - адресной книги [22]. Данная система была разработана при помощи руководств [24], а также онлайн-курса Алексея Баранцева по созданию автоматических тестов на языке Python [21].

3. Используемые инструменты

Для создания системы необходимо было точно определиться с выбором ряда используемых инструментов таких как:

- фреймворк для автоматизированного тестирования
- библиотека для создания отчётов
- генератор документации

Также в компании активно применяются следующие инструменты и технологии:

- JavaScript фреймворк: AngularJS [5]
- сервер непрерывной интеграции: TeamCity [9]
- плагин для инспектирования качества кода: SonarQube [16]

Таким образом, необходимо было предусмотреть оптимальную интеграцию с вышеупомянутыми продуктами, чтобы внедрить систему в общий процесс разработки.

3.1. Фреймворк для автоматизированного тестирования

Для написания автоматических тестов был выбран фреймворк Protractor. Protractor полностью написан на языке JavaScript, что существенно упрощает взаимодействие разработчиков и тестировщиков, тем самым устраняя недопонимания и объединяя команду в процессе разработки автотестов.
Помимо выбора библиотеки тестирования необходимо было определиться с выбором применяемой надстройки из следующего списка:

- Jasmine [8]
- Cucumber [20]

- Mocha [10]

После изучения вышеупомянутых надстроек было принято решение использовать Jasmine. Больших отличий от Mocha у этой надстройки нет. Cucumber больше нацелен на тестирование приложений на Java. Тестировать приложения на других языках с помощью Cucumber необходимо тогда, когда у разработчиков должна поддерживаться постоянная связь с заказчиком. Такой цели в данной задаче не было.

Плюсом Jasmine является тот факт, что unit-тесты [12], которыми занимаются разработчики пишутся именно с использованием фреймворка Jasmine.

	Jasmine	**Mocha**	**Cucumber**
Синтаксис	Тесты описываются в блоках describe, сами проверки в блоках it	Аналогично Jasmine. Но, библиотека Chai [1] даёт возможность производить проверки тремя различными стилями: should, assert, expect (аналогичен Jasmine)	Написание тестов состоит из 2-х этапов с различным синтаксисом для каждого. 1.Описание особенностей (feature). (Можно на русском языке) 2.Определение поступков - на половину генерируется после первого этапа.

	Jasmine	**Mocha**	**Cucumber**
Mocking	Есть в Jasmine по умолчанию. createSpy()	Отсутствует. Требуется загрузка библиотеки Sinon.js [17]. В свою очередь в данной библиотеке предусмотрены три способа мокинга: spies, stubs (аналогично spy в Jasmine), mocks.	Используется stubs [7]
Асинхронные процессы	До версии Jasmine 2.3 процесс обработки асинхронных процессов был затруднён. В ранних версиях для работы с процессами применялись задержки (timeouts).	Есть в библиотеке Sinon.js	Применяются методы задержек

Таблица 1: Сравнительный анализ свойств надстроек библиотеки Protractor

3.2. Библиотека для создания отчётов

Необходимые свойства отчёта о прохождении тестов:

- каждый прогон автоматических сценариев сохраняется с указанием точного времени и даты окончания
- результаты проверок хранятся в строгой иерархии, повторяющей структуру хранения файлов с тестами
- каждая проверка фиксируется в виде скриншота страницы в момент проверки
- отчёт предоставляет статистические данные, оформленные в виде таблиц и графиков

Практически вся перечисленная функциональность реализована в библиотеке protractor-screenshot-reporter [15]. Для нее было написано небольшое расширение, позволяющее отправлять полученный отчёт на почту определённому кругу лиц и строить графики.

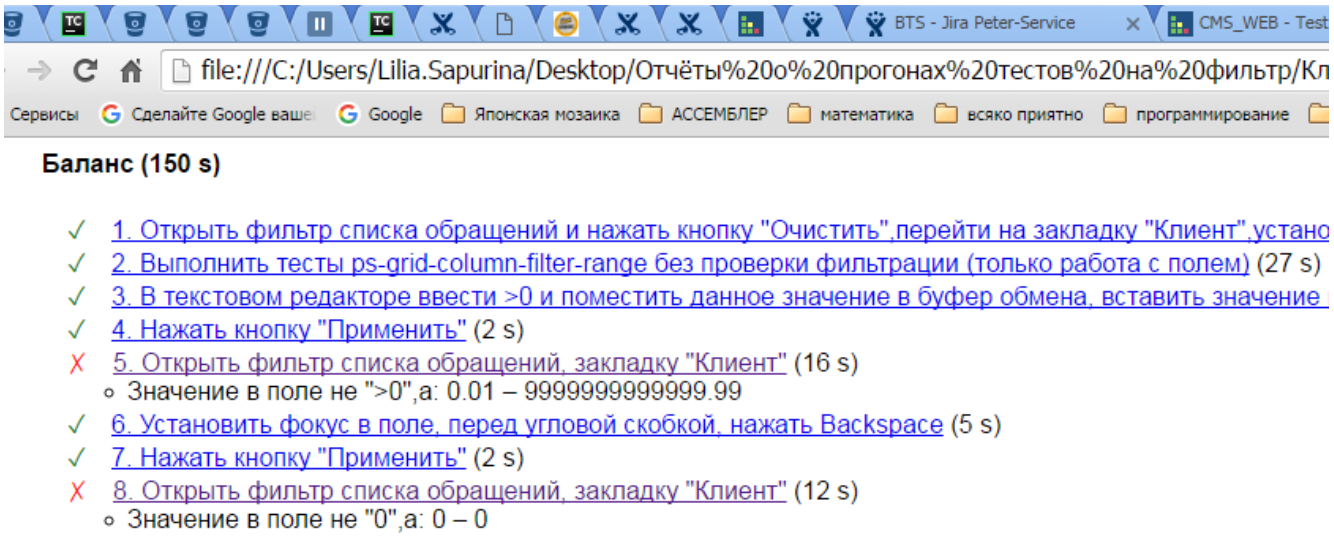

Рис. 1: Пример отчёта

3.3. Генератор документации

В качестве библиотеки для автоматической генерации документации используется JSDoc [19]. Библиотека предназначена для работы с JavaScript файлами. В результате получается цельная и информативная документация в формате HTML.

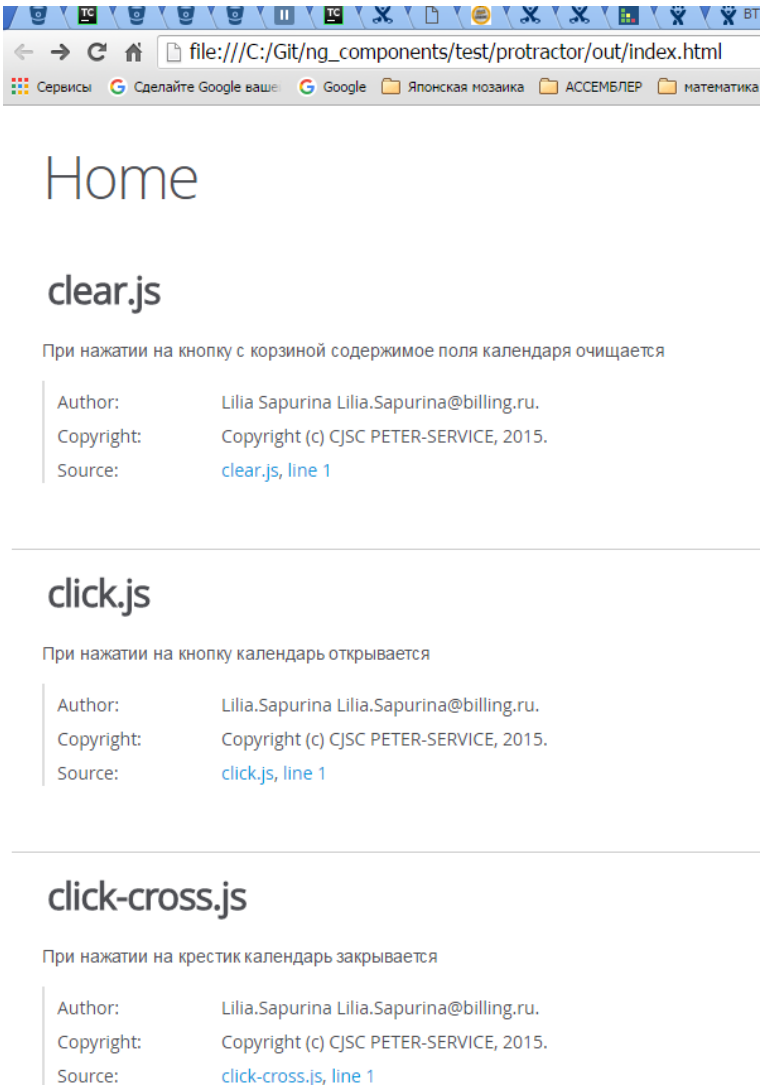

Рис. 2: Пример страницы документации

4. Структура тестируемого приложения

Тестируемый продукт — web-приложение, тестируемые объекты — группы элементов на странице. Доступ к этим элементам происходит через их идентификаторы, а также css, xpath и jQuery запросы к html-разметке. В любом случае взаимодействие с элементом тесно связано с концепцией разрабатываемой страницы. Это значит, что при плохо разработанной системе доступа, любое изменение программиста может повлечь за собой потерю связи с элементом и тем самым нарушить работу теста.

Кроме того, запросы идут к серверу приложений, поэтому получение некоторых результатов происходит с задержкой.

В основе разработки приложения лежит компонентный подход. Его страницы разделены на так называемые компоненты, которые состоят из одного или нескольких элементов, связанных некоторой функциональностью. Каждый такой блок изначально отдельно разрабатывается дизайнерами и программистами.

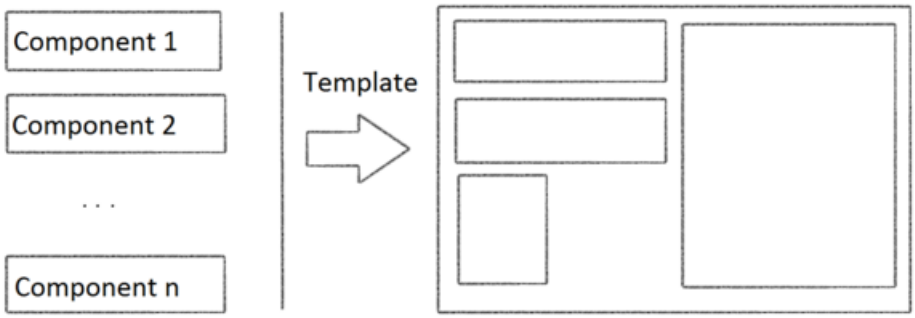

Рис. 3: Схема компонентного подхода

Поэтому задача тестирования приложения сводится к тестированию каждого компонента в отдельности, а затем проверки их совместного функционирования. Очевидно, что именно тесты на проверку отдельных компонентов следует сделать максимально независимыми друг от друга, а также от разметки страницы, на которую этот компонент был помещён.

5. Структура системы автоматизированного тестирования

Protractor — это фреймворк, разработанный на основе WebDriverJS [18]. Он запускает реальный браузер и производит определенные действия над элементами, совершает проверки. Главная идея всех расширений заключалась в том, чтобы обеспечить работу тестов не над всей web-страницей целиком, а непосредственно с каждым её компонентом. А далее объединить все автономные действия и проверки для компонентов в единый тестовый сценарий, обеспечивающий проверку страницы.

5.1. Пример

Рассмотрим два компонента на странице — **toolbar** (набор кнопок) и **table** (таблица).
Пусть тестовый сценарий устроен следующим образом:

1. Открыть страницу в браузере.
2. Посчитать количество строк таблицы и проверить, что оно соответствует определенному числу.
3. Найти в списке кнопок кнопку очистки и нажать на неё.
4. Проверить, что в таблице нет ни одной записи.

С учётом расширений выполнение такого сценария будет осуществляться в следующем виде:

1. Открываем страницу в браузере; от корневого элемента страницы ищем компонент **table** по некоторому идентификатору или запросу:

   table = psTable(by.id('tableId'));

   Обёртка psTable обеспечивает подключение к объекту **table** всех специфических методов, описанных в классе для компонентов-таблиц.

2. Используем пользовательский метод из класса psTable для подсчёта количества строк в таблице; делаем проверку:

```
expect(table.rowCount()).toEqual(number);
```

3. Снова от корневого элемента страницы ищем объект **toolbar**:

```
toolbar = psToolbar(by.id('toolbarId'));
```

   При помощи пользовательского метода получаем из списка кнопку очистки и нажимаем на неё:

```
toolbar.clickClearButton();
```

4. Делаем проверку для количества строк в таблице:

```
expect(table.rowCount()).toEqual(0);
```

5.2. Схема наследования

Все основные методы работы с элементами в Protractor описаны в классе ElementFinder. Соответственно расширять необходимо было именно этот класс.
Была разработана следующая схема наследования:
создан класс-наследник ElementFinder-a: psComponent. В этом классе определяются общие методы для всех компонентов страницы. Например, извлечение ширины элемента или нажатие на кнопку по её названию в компоненте.
Для каждого описанного в отдельности компонента страницы создан свой собственный класс-наследник psComponent. В этих классах описаны специфические методы для компонентов. Например, нажатие кнопки раскрытия выпадающего списка в компоненте dropDown или подсчёт количества строк в компоненте table.
Были созданы оболочки для компонентов, чтобы создавать в тестовых сценариях объекты определённых типов. Эти оболочки описаны в пользовательском плагине, объединяющем все описанные классы-расширения и обрабатывающем пользовательские запросы к их методам.

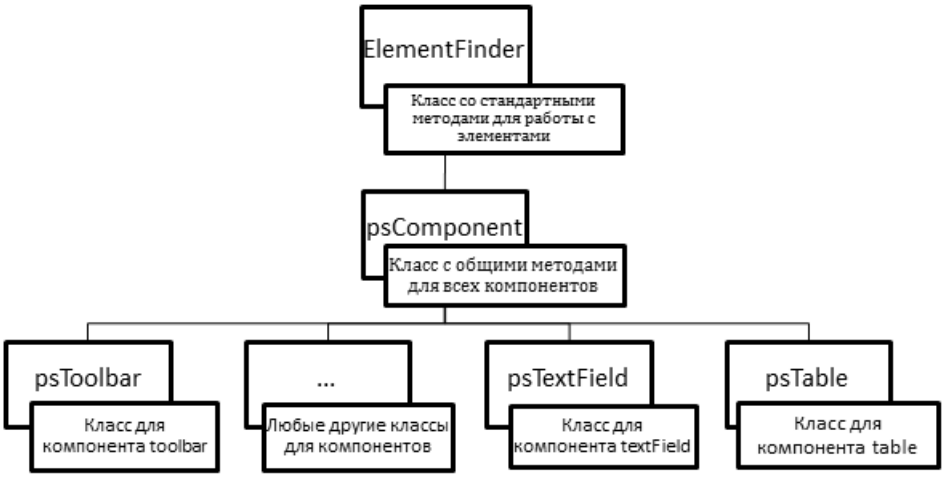

Рис. 4: Схема наследования

5.3. Расширение matchers

Помимо перечисленных расширений были созданы библиотеки так называемых matchers. Это методы, используемые для специфических проверок, не описанных в Protractor. Например, проверка, что объект имеет определённый цвет.

expect(object).hasColor(“Green”);

5.4. Расширение пользовательских локаторов

Ещё одним важным созданным расширением являются пользовательские локаторы.

Очень удобно использовать конструкцию by.customLocator(), так как стандартный набор локаторов ограничен. Для этого была создана ещё одна автономно подключаемая библиотека, которая представляет собой набор пользовательских методов для поиска элементов по специфическим локаторам. Например, поиск тела колонки таблицы по названию её колонки или поиск поля ввода по его названию.

5.5. Подключение пользовательских методов

Очень полезным для упрощения написания тестов было создание библиотеки с расширенными javascript-методами. Например, парсеры строк, работа с датами и временем, методы ожидания или вывод сообщений в лог. Все эти методы разбиты по их функциональности и подключаются автоматически.

6. Применение паттерна Page Object

В целом, все абстракции методов на уровне компонентов представляют из себя приближение к паттерну автоматического тестирования Page Object [3]. Но всё-таки им не являются. Они упрощают работу по поиску и универсализации локаторов элементов. Применение Page Object в свою очередь направлен скорее на упрощение работы по написанию тестов, уменьшению их размера и улучшение читаемости. После его применения тестовый сценарий, написанный на любом языке, можно читать как текст, состоящий из названий применяемых методов и проверок.

6.1. Суть методологии

Каждая страница в приложении представляется как отдельный класс с набором собственных специфических методов. При обращении к такой странице, в тесте создаётся объект данного класса и применяются необходимые методы для осуществления действий на этой странице и каких-то специальных проверок. При переходе на другие страницы приложения аналогичным образом создаются экземпляры соответствующих им классов. Таким образом, описание действий и проверок происходит не непосредственно в тесте, а в отдельно созданном классе. В тесте указывается лишь говорящее название метода, по которому точно можно определить то, что реализует этот метод.

Внутренние методы классов реализуются при помощи использования разработанных и описанных выше расширений. Таким образом получается, что абстракция методов производится дважды - на уровне отдельных компонентов на странице, а затем на уровне всей страницы целиком.

Важно заметить, что в данном подходе не исключено наследование. Часто возникают случаи, когда страница в зависимости от применения каких-то динамических действий может полностью менять своё содержимое. Например, если на странице представлено несколько вкладок и на каждой из них находится своё собственное содержимое со специ-

фической функциональностью. В этом случае необходимо определить класс всей страницы с методами для действий на ней и в том числе с методами перехода на вкладки. В свою очередь каждая вкладка будет представлена в виде своего собственного класса, отнаследованного от класса родительской страницы.

6.2. Плюсы применения

- **устранение повторений в коде:**
  параметризация методов помогает избежать многочисленных повторений одинаковых частей кода в тесте.

- **улучшение читаемости:**
  тест представляет собой цельный и последовательный текст, состоящий из названий методов и проверок. По этому тексту можно однозначно установить что делает тест даже без использования документации.

- **структурированное хранение тестовых методов:**
  классы хранятся в отдельной папке в строгой иерархии, соответствующей структуре приложения. Такая организация упрощает поиск необходимых методов.

7. Итоговая структура каталогов системы

После разработки всех описанных расширений была организована следующая структура хранения:

- **doc**
  Автоматически сгенерированная документация в виде html-страниц.
- **resource**
  Тестовые сценарии. Внутри папки организована строгая иерархия, соотвествующая хранению тестовых сценариев в TestRail [11].
- **javascript**
  Вспомогательные методы, написанные на javascript.
- **helper**
  Параметры тестов.
- **component**
  Классы-расширения стандартных методов для каждого используемого компонента.
- **plugin**
  Пользовательские плагины.
- **config**
  Конфигурационные файлы.
- **PO**
  Классы для реализации паттерна Page Object.
- **result**
  Автоматически генерирующийся отчёт о выполнении.
- **lib**
  Подключаемые библиотеки-расширения (matchers, locators).

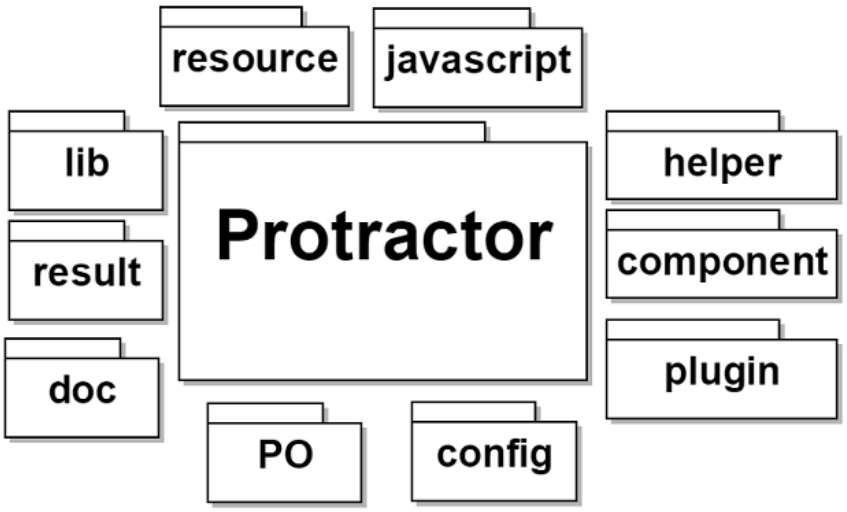

Рис. 5: Структура каталогов системы

8. Тестовые сценарии и их реализация

Все ручные тесты описываются в объединённых по функциональности тестовых сценариях. Они представляют из себя пошаговые инструкции действий, которые пользователь должен произвести над страницей и результат реакции системы на эти действия. Сценарии описываются в иерархическом порядке при помощи инструмента TestRail [11].

Автоматические тесты нужно было организовывать таким образом, чтобы написанный кейс максимально соответствовал программному коду. Необходимо было организовать хранение и отчетность о прохождении тестов таким образом, чтобы шаг и причину ошибки можно было бы выявить максимально точно. Это было реализовано следующим образом. Тесты разбивались на блоки. Блок describe содержал в себе все шаги кейса и назывался в точности так же как и сам тестовый сценарий. В свою очередь внутри такого блока поочерёдно для каждого шага описывался блок it, в котором непосредственно выполнялись описанные в шаге действия и проверки. Этот метод организации тестов не соответствует предложенному в документации, но зато позволяет избежать падения и задержек тестов.

9. Взаимодействие с другими продуктами

Запуск тестовых сценариев может происходить вручную или автоматически. Ручной запуск можно производить как локально (через консоль или настроив конфигурацию в среде разработки), так и через сервер TeamCity. Автоматически тесты запускаются при каждом новом коммите в ветку разработки. Но помимо автотестов в системе выполняются ещё unit-тесты [12], провал которых не допускает попадания изменений в систему, а также проверки форматирования и корректности кода при помощи SonarQube [16]. Чтобы все действия выполнялись в нужном порядке и не противоречили друг другу был использован инструмент для автоматизации запуска заданий – Grunt [4]. Этот инструмент так же был выбран по той причине, что он написан на javascript. Следовательно, улучшает взаимопонимание в команде web-разработчиков.

10. Влияние системы на процесс разработки

Разработанная система заметно упростила жизнь ручным тестировщикам. Рутинная работа по многократному ручному прогону тесткейсов стала редким исключением. Но, конечно, чтобы оценить и проанализировать влияние внедрённой системы на общий процесс разработки были изучены метрики автоматизированного тестирования. На их основе были проанализированы полученные за период функционирования системы (3 месяца) данные.

10.1. Метрики автоматизированного тестирования

После изучения [14], [6], [23] были сформулированы следующие метрики для обоснования тестирования:

- доставка бага до разработчика с момента внесения правки в код (единицы: время)
- время прогона автотестов по сравнению с ручной итерацией
- количество багов, найденных автотестами
- трудоёмкость тестирования

Значения всех метрик измерялись на фиксированном наборе тестов. Очевидно, что в зависимости от сложности проверок и переходов эти значения могут варьироваться, но сами соотношения времён автоматизированного и ручного тестирования должны оставаться неизменными. Все нижеперечисленные эксперименты ставились на машине со следующими характеристиками:

- операционная система: Windows 7 64-bit
- процессор: Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz (2 processors)
- оперативная память: 16,0 GB

10.1.1. Доставка бага до разработчика

После внесений изменений в код, разработчик отправляет их на оценку определённому кругу своих коллег, которые имеют отношение к тому продукту, который он создаёт. Изменения вносятся в основную ветку разработки и добавляются в исходный код лишь тогда, когда они просмотрены и одобрены всеми задействованными сотрудниками - ревьюверами. Время оценки правильности изменений может варьироваться. Ревьюверы могут просить разработчика дорабатывать или исправлять свои изменения. После одобрения, изменения анализируются автоматическим скриптом, который проверяет работу юнит-тестов и стиля кода при помощи SonarQube. Но все эти проверки в совокупности не дают гарантий, что при внесении изменений в код, не пропадёт часть нужной функциональности.

До внедрения системы автоматизированного тестирования, тестировщики обязаны были вручную проверять одни и те же тест-кейсы с некоторой периодичностью. Для форм с обширной функциональностью, такие проверки могут занимать до 10 человеко-дней, т.е. целый спринт. При нахождении ошибки в функциональности тестировщик оформляет баг, который потом отправляет как задачу разработчику, вносившему изменения. Таким образом, доставка бага в случае ручного тестирования замедлена.

Удобство наличия системы автоматизированного тестирования в том, что их можно запускать после любых, даже самых незначительных изменений. Прогон происходит в фоновом режиме и через некоторое время возвращает подробный отчёт о проверках. В отчёте тестировщик явно может проверить наличие ошибок и оформить баги. Очевидно, что и сами автотесты могут содержать какие-то недочёты и ошибки. В случае сомнения правильности работы тестировщик может вручную прогнать ту или иную часть сценария. Это может занять какое-то время, но при этом это будет намного быстрее, чем прогонять все тесты в совокупности. По статистике последних прогонов было выяснено, что доставка бага до разработчика в этом случае происходит в течении од-

ного дня.

Кроме того, из-за трудоёмкости прогона сценариев, ручные тесты в реальности прогоняются лишь за несколько недель до отгрузки нового релиза. Таким образом, баги к разработчикам приходят в огромных количествах и все сразу. Это создаёт большие неудобства в дальнейшей разработке и затормаживает процесс. Ошибки исправляются не сразу, а в самом конце разработки. После внедрения системы автоматизированного тестирования эта проблема пропадает и ошибки исправляются в момент их появления, что значительно оптимизирует процесс создания продукта.

	min	**max**
Ручное	1	10
Авто	0	1

Таблица 2: Время доставки багов в человеко-днях

10.1.2. Время прогона автотестов

Время прохождения тестов зависит от сложности сценариев и количества автоматизированных тест-кейсов, которые нужно запустить. Все задержки, вызываемые загрузкой значений из базы данных, обновления страниц приложения, загрузки динамических элементов и т.п. не превышают аналогичных им задержек при ручном прогоне. Кроме того, ввод данных при автоматизированном подходе происходит намного быстрее. При ручном тестировании пользователь может допустить ошибку в последовательности выполнения шагов тест-кейса, что приведёт в потере логики проверок, а, следовательно, к необходимости повторного прогона теста с самого начала. При автоматизированном подходе такие ситуации исключены.

Таким образом, автоматизированный прогон тестов значительно превышает ручной прогон по скорости выполнения.
Эксперимент проводился на сценариях, не содержащих ожиданий дополнительных загрузок. Любое действие в них мгновенно приводит к той или иной реакции приложения.

Количество сценариев	**1**	**5**	**10**	**50**	**100**
Ручное	10 мин	1 ч	3 ч	15 ч	48 ч
Авто	30 сек	2 мин	5 мин	35 мин	2 ч

Таблица 3: Время прогона тестовых сценариев

10.1.3. Количество багов, найденных автотестами

Преимущество в нахождении багов автоматическими тестами в первую очередь заключается в том, что за счёт случайного подхода ввода данных есть возможность находить непредвиденные ошибки. Таким образом, количество найденных багов при автоматизированном подходе больше.

За три месяца системой автоматизированного тестирования было найдено 96 багов, ручным тестированием - 89. Разница представляет собой описанный случай - непредвиденные ошибки.

10.1.4. Трудоёмкость тестирования

Здесь уместным будет сделать замечание о том, что улучшения за счёт внедрения системы автоматизированного тестирования становятся заметными по данной метрике лишь после определенного времени функционирования системы. В начальный момент внедрения системы трудоёмкость по созданию автотестов значительно превышает ручное тестирование. Причём чем больше тестов разрабатывается, тем быстрее

происходит процесс их создания. Обуславливается это тем, что в процессе разработки тестов для определённых форм, расширяются пользовательские библиотеки методов. На эти расширения может уйти немало времени, но затем ими можно будет просто пользоваться, не задумываясь об их реализации.

Период функционарования	1 месяц	2 месяца	3 месяца
Время автоматизации тест-кейса	2 дня	4 часа	1 ч

Таблица 4: Время автоматизации тест кейса

Заключение

Был реализован полный жизненный цикл построения системы автоматизированного тестирования от создания схемы хранения и расширения стандартных методов до запуска и получения отчетов о прохождении тестов.
Данная система позволяет:

- легко расширять набор стандартных методов для работы с элементами страницы, пользуясь разработанными классами для компонентов. А также расширять стандартные методы для проверок, внедрять новые локаторы для поиска.

- описывать тест-кейсы, «переводя» их на язык программирования. Схема написания тестов позволяет однозначно сопоставлять шаги сценария с программными блоками автотестов

- многократно использовать автономные тесты компонентов для различных страниц и приложений

- легко ориентироваться в созданных методах благодаря автоматически генерирующейся документации

- получать исчерпывающий отчёт о прохождении тестов, позволяющий однозначно определить шаг, в котором произошла ошибка. Также отчёт позволяет посмотреть лог и скриншот ошибки, что значительно уменьшает сложность поиска причины возникновения ошибки

- Проверять корректность внесённых изменений за счёт автоматически запускающихся сценариев

В настоящий момент система полностью реализована. Процесс написания тестовых сценариев максимально упрощен. Им занимается уже целая команда тестировщиков. Автоматические тесты помогают найти большой процент программных багов и сократить трудозатраты на ручной прогон сценариев.

Планы на будущее

Продолжая тему максимальной автоматизации в будущем планируется работа по внедрению системы автоматического создания тестовых багов по отчету о прохождении тестов. В настоящий момент для того, чтобы известить разработчиков о возникшей ошибке тестировщикам приходится вручную просматривать тестовые отчёты и создавать задачи по их исправлению. В будущем этот процесс планируется автоматизировать. Проблемы возникают лишь с определением того, на кого назначать эту работу. Обычно тестировщики сами знают кто из разработчиков занимался той или иной функциональностью и сами решают на кого назначать работу. При автоматизированном подходе придётся немного изменить и сам процесс разработки, для того, чтобы эту информацию можно было бы извлечь непосредственно из программного кода.

Список литературы

[1] Chai Assertion Library. — URL: http://chaijs.com/.

[2] Duvall Paul, Matyas Stephen M., Glover Andrew. Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). — Addison-Wesley Professional, 2007. — ISBN: 0321336380.

[3] Fowler Martin. — PageObject, 2013. — URL: http://martinfowler.com/bliki/PageObject.html.

[4] GRUNT. The JavaScript Task Runner. — URL: http://gruntjs.com/.

[5] Google. — AngularJS. — URL: https://angularjs.org.

[6] Graham Dorothy. — ROI of test automation:benefit and cost, 2010. — URL: http://www.dorothygraham.co.uk/downloads/generalPdfs/ProfTesterROI.pdf.

[7] Head First Design Patterns / Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra. — O' Reilly & Associates, Inc., 2004. — ISBN: 0596007124.

[8] Jasmine. Behavior-Driven JavaScript. — URL: http://jasmine.github.io/.

[9] JetBrains. — TeamCity - Powerful Continuous Integration out of the box. — URL: https://www.jetbrains.com/teamcity.

[10] Mocha: simple, flexible, fun. — URL: https://mochajs.org/.

[11] Modern Test Case Management Software for QA and Development Teams. — URL: http://www.gurock.com/testrail/.

[12] Osherove Roy. The Art of Unit Testing: With Examples in .Net. — 1st edition. — Greenwich, CT, USA : Manning Publications Co., 2009. — ISBN: 1933988274, 9781933988276.

[13] Protractor - end-to-end testing for AngularJS. — URL: http://www.protractortest.org.

[14] ROI в автоматизации тестирования, 2013. — URL: http://bugscatcher.net/archives/2920.

[15] Screenshot Reporter for Protractor. — URL: https://www.npmjs.com/package/protractor-screenshot-reporter.

[16] SonarSource. — SonarQube. — URL: http://www.sonarqube.org.

[17] Standalone test spies, stubs and mocks for JavaScript. — URL: http://sinonjs.org/.

[18] UWEBDRIVER I/O Selenium 2.0 bindings for NodeJS. — URL: http://webdriver.io/.

[19] Use JSDoc. — URL: http://usejsdoc.org/.

[20] ltd Cucumber. — Cucumber. Simple, human collaboration. — URL: https://cucumber.io.

[21] Алексей Баранцев. Онлайн-курс «Программирование на Python для тестировщиков». — URL: http://software-testing.ru/trainings/schedule?task=3&cid=233.

[22] Лилия Сапурина. Система автоматизированного тестирования web-приложений на Python. — URL: https://github.com/liliasapurina/python_training.

[23] Метрики в тестировании, 2012. — URL: http://blog.shumoos.com/archives/271.

[24] Тестирование и качество ПО. — URL: http://software-testing.ru/.