МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Информационные системы и технологии корпоративного управления
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему Моделирование среды валидации и тестирования фронт-енд разработчика

Студент А.Ю. Кузьмин
(И.О. Фамилия) (личная подпись)

Научный руководитель к.п.н, доцент, Е.В. Панюкова
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Оглавление

Введение

Важнейшей задачей фронт-енд разработчика программного обеспечения является создание интерфейса пользователя, отвечающего современным требованиям эргономики и дизайна.

Как показывает практика проектирования программного обеспечения (ПО), одной из причин снижения производительности приложений является неоптимизированный код на стороне клиента.

Так, по мнению аналитиков, повышение производительности интерфейса на 50% приведет к повышению общей производительности приложения на 40% [42].

Как известно, при проверке правильности программ и систем рассматриваются процессы верификации, валидации и тестирования программ, которые регламентированы в стандарте ISO/IEC 12207 жизненного цикла ПО [11].

Валидация (Validation) – это процесс оценки конечного ПО на предмет соответствия ожиданиям и требованиям клиента. Это динамический механизм проверки и тестирования фактического продукта [7].

Фронт-енд тестирование (Front-end Testing) - это тестирование графического интерфейса пользователя (GUI), функциональности и удобства использования веб-сайта или приложения.

Среда валидации и тестирования фронт-енд разработчика обеспечивает автоматизированную поддержку данных процессов.

Как показал анализ, известные средства валидации и тестирования функционально избыточны и недостаточно эффективны для задач фронт-енд разработки ПО, т.к. не учитывают специфику последней.

Совершенно очевидно, что для создания эффективной среды валидации и тестирования фронт-енд разработчика необходимо предварительно разработать адекватную модель указанной среды.

Таким образом, разработка модели эффективной среды валидации и тестирования фронт-енд разработчика ПО представляет **актуальность** и научно-практический интерес.

Объектом исследования магистерской диссертации является среда валидации и тестирования фронт-енд разработчика.

Предметом исследования является модель среды валидации и тестирования фронт-енд разработчика.

Целью работы является – теоретическое обоснование и практическая реализация модели среды валидации и тестирования, обеспечивающей высокую эффективность решения задач фронт-енд разработчика.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Проанализировать современное состояние проблемы исследования.
2. Проанализировать методологии и технологии моделирования среды валидации и тестирования фронт-енд разработчика.
3. Разработать модель автоматизированной среды валидации и тестирования фронт-енд разработчика.
4. Подтвердить функциональную эффективность среды валидации и тестирования фронт-енд разработчика, разработанной на основе предлагаемой модели.

Гипотеза исследования: применение предлагаемой модели в качестве основы для построения среды валидации и тестирования позволит повысить эффективность решения задач фронт-енд разработки ПО.

Методы исследования. В процессе исследования использованы следующие методы и подходы: программная инженерия, методологические подходы к тестированию ПО, объектно-ориентированный подход к анализу и проектированию программного обеспечения.

Новизна исследования заключается в разработке модели среды валидации и тестирования, которая позволит повысить эффективность решения задач фронт-енд разработки ПО.

Практическая значимость исследования заключается в возможности практического применения предлагаемой модели для построения эффективной среды валидации и тестирования фронт-енд разработчика ПО.

Теоретической основой диссертационного исследования являются научные труды российских и зарубежных ученых, занимающихся проблемами моделирования и повышения эффективности систем управления тестированием ПО.

Основные этапы исследования: исследование проводилось с 2018 по 2020 год в несколько этапов:

На первом этапе (констатирующем этапе) – формулировалась тема исследования, выполнялся сбор информации по теме исследования из различных источников, проводилась формулировка гипотезы, определялись постановка цели, задач, предмета исследования, объекта исследования и выполнялось определение проблематики данного исследования.

Второй этап (поисковый этап) – в ходе проведения данного этапа осуществлялся анализ методологий моделирования сред валидации и тестирования, была разработана модель эффективной среды валидации и тестирования фронт-енд разработчика, подготовлены и опубликованы научные статьи по теме исследования в научных журналах и сборниках.

Третий этап (оценка эффективности) – на данном этапе осуществлялась оценка эффективности и проверка адекватности предлагаемой модели среды валидации и тестирования фронт-енд разработчика, сформулированы выводы о полученных результатах по проведенному исследованию.

На защиту выносятся:

1. Модель среды валидации и тестирования фронт-енд разработчика.
2. Результаты проверки адекватности предлагаемой модели среды валидации и тестирования фронт-енд разработчика.

По теме исследования опубликованы 2 статьи:

1. Кузьмин А.Ю. Сравнительный анализ фреймворков для автоматизации фронтенд-тестирования // Вестник научных конференции.

2020, №9-3(61). С. 65-66.

2. Кузьмин А.Ю. Модель методики автоматизации фронтенд-тестирования программного обеспечения // Вестник научных конференции. 2020, №12.

Диссертация состоит из введения, трех глав, заключения и списка литературы.

В первой главе проанализировано современное состояния проблемы повышения управления фронт-енд валидацией и тестирования. Рассмотрены методологические и технологические основы фронт-енд валидации ПО. Описаны методологические основы и дан обзор современных технологий фронт-енд тестирования.

Во второй главе рассмотрены принципы построения интегрированной среды валидации и фроент-енд тестирования разработчика. Рассмотрены технологии автоматизации валидации и фроент-енд тестирования разработчика. Разработан алгоритм автоматизации фроент-енд тестирования. Дан обзор и анализ инструментов автоматизации фронтенд-тестирования.

Третья глава посвящена разработке и проверки адекватности модели среды валидации и фроент-енд тестирования разработчика. Подтверждена функциональная эффективность среды валидации и фроент-енд тестирования разработчика, реализованной на основе предлагаемой модели.

В заключении приводятся результаты исследования.

Работа изложена на 73 страницах и включает 24 рисунка, 7 таблиц, 43 источника.

Глава 1 Анализ современного состояния проблемы управления фронт-енд валидацией и тестированием

Фронт-енд (Front-end) – это клиентская сторона пользовательского интерфейса к программной части сервиса или вариант архитектуры ПО [23].

Фронт-енд разработка — это разработка пользовательского интерфейса и функций, которые работают на клиентской стороне веб-сайта или приложения.

На рисунке 1.1 представлена схема процесса фронденд-разработки в концепции компании UpWork [39].

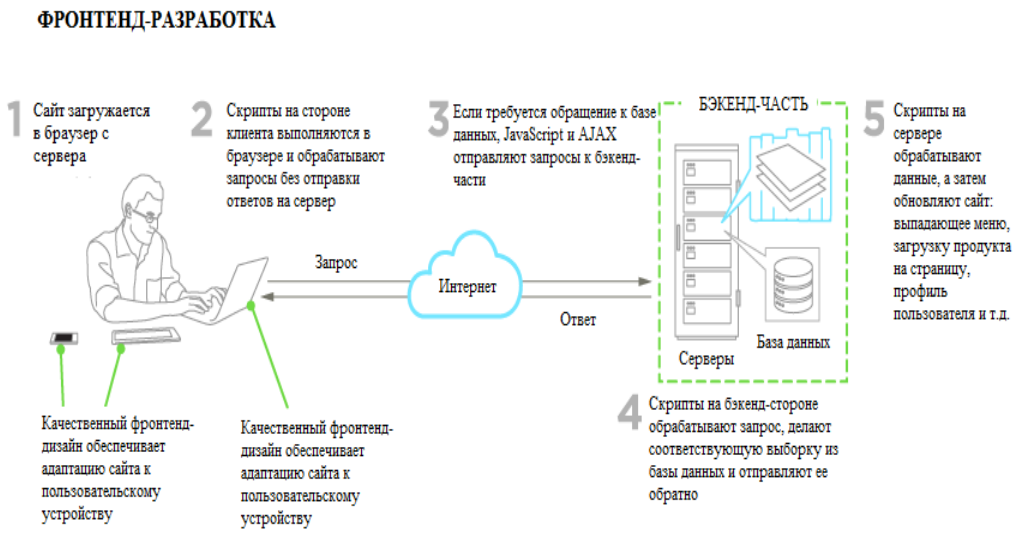

Рисунок 1.1 - Схема процесса фронденд-разработки в концепции компании UpWork

Проблематика фронтенд - валидации и тестирования довольно широко освящена в работах отечественных ученых и ИТ-специалистов таких, как А. Савин, Г. Тайяр, М. Тредер, М. Фаулер, K. C. Dodds, N. Michas, M. Paulasaari, E. Yerburgh и др.

Теоретическим аспектам фронт-енд тестирования посвящены работы E. Yerburgh и М.Фаулера [43].

В работах А.Савина, Г. Тайяра и M. Paulasaari освящаются вопросы, связанные с автоматизацией отдельных уровней фронт-енд тестирования и созданием автотестов [13, 35, 39].

N. Michas и M. Тредер рассматривают в своих работах алгоритмы и требования к валидации, основанные на принципах юзабилити [33].

K. C. Dodds предлагает принципы организации эффективного тестирования приложений JavaScript [26].

Вместе с тем, анализ литературы и интернет-источников по теме исследования подтвердил недостаточность работ, посвященных проблематике моделирования и проектирования интегрированных сред фронт-енд валидации и тестирования разработчика.

1.1 Методологические и технологические основы фронт-енд валидации

Валидацию фронт-енд тестирования можно условно разделить на валидацию программного кода и валидацию данных.

Разработка форм — один из самых ответственных и сложных этапов создания веб-интерфейсов.

Проект должен получить пользовательские данные, проверить их и дать пользователю обратную связь.

Перечислим цели валидации данных:

1) получение корректных данных в корректном формате для последующей их обработки;

2) защита пользователя от разного рода перехвата данных;

3) защита приложения от взлома и доступа к данным пользователя, которые он ввел в форму.

На рисунке 1.2 изображена диаграмма деятельности UML, представляющая сценарий валидации данных в форме [33].

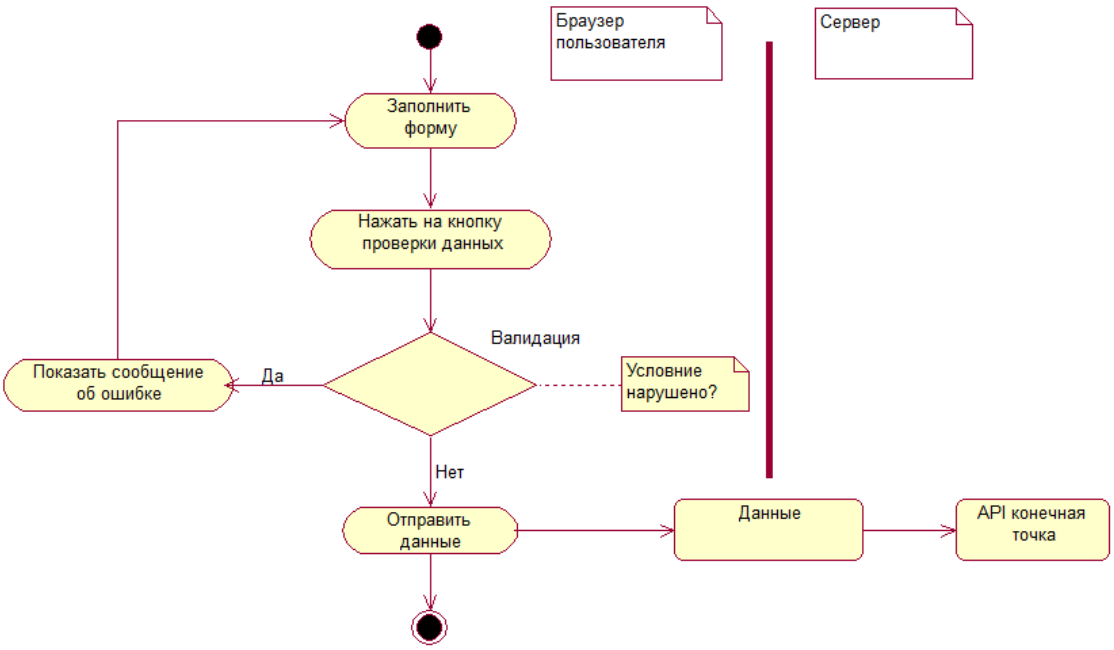

Рисунок 1.2 – Пример сценария валидации данных

Следует отметить, что современные браузеры предоставляют разработчику встроенный API, позволяющий поэтапно реализовать валидацию данных методом Progressive enhancement (прогрессивное улучшение) - от HTML/CSS к JS [12].

Требования к валидации программного кода по М. Тредеру, основанные на принципах юзабилити:

1) валидация должна быть в правильном месте. Имеется в виду форма сообщения о некой ошибке;

2) валидация должна быть в правильное время - инлайн-валидация: пользователь не должен заполнять всю форму и узнавать об ошибках только после отправки. Еще более критична потеря какие-то данных;

3) подходящий цвет для вывода сообщений об ошибке;

4) понятный язык сообщения об ошибке.

Следует констатировать, что не существует общепринятого регламента проведения валидации программного кода.

Вместе с тем, можно выделить следующие этапы валидации кода ПО:

1) Выбор инструмента валидации кода.

2) Проверка программного кода.

3) Составление отчета об ошибках.

Технологии фронт-енд валидации кода основаны на следующих подходах:

− проверка кода в текстовых редакторах или интегрированных средах разработки [1].

В качестве текстовых редакторов для проверки кода используются программные продукты: Atom, Notepad++, Netbeans и др.

Преимуществом такого подхода является относительная простота реализации. К недостаткам можно отнести ограниченные возможности инструментария.

Интегрированная среда разработки (Integrated Development Environment, IDE) – это программный пакет, который объединяет основные инструменты, необходимые для написания и тестирования программного обеспечения.

Примеры IDE, используемых для фронт-енд разработки: WebStorm, Zend Studio, Komodo IDE и др.

Преимуществом такого подхода является широкие возможности для проверки кода. К недостаткам относительную дороговизну рекомендуемых IDE и их функциональная изыточность.

− применение специализированных инструментов для валидации кода, например, JSLint и ESLint.

JSLint - это инструмент статического анализа кода, используемый при разработке программного обеспечения для проверки соответствия исходного кода JavaScript правилам кодирования. Он предоставляется в основном в виде веб-приложения на основе браузера, доступного через домен jslint.com, но есть также модификации для командной строки [30].

ESLint - это инструмент для выявления шаблонов, обнаруженных в коде ECMAScript / JavaScript, и составления отчетов о них, с целью сделать код более согласованным и избежать ошибок [27].

Основные отличия инструментария ESLint от JSLint:

- ESLint использует Espree для синтаксического анализа JavaScript;
- ESLint использует AST для оценки шаблонов в коде;
- ESLint полностью подключаемый, каждое правило - это подключаемый модуль, и вы можете добавить больше во время выполнения.

С точки зрения фроент-енд тестировщиков и разработчиков применение специализированных инструментов для валидации кода является самым оптимальным решением.

1.2 Методологические основы фронт-енд тестирования

Следует отметить, что методологической основой фронт-енд тестирования являются базовые методы и подходы, используемые в классическом тестировании ПО.

Соответственно фронт-енд тестирование подразделяется на статическое и динамическое тестирование [13].

В статическом тестировании используются обзоры кода, пошаговые руководства и проверки синтаксиса.

Статическое тестирование обычно используется как корректура для поиска ошибок в коде, и его можно использовать вместе с текстовыми редакторами для проверки исходного кода на предмет структуры, синтаксиса и потока данных для статического анализа [26].

Статическое тестирование часто применяется во время написания кода, так как динамическое тестирование происходит во время выполнения программы или ее частей.

Другой основной подход к тестированию, называемый динамическим тестированием, означает выполнение программы и ее запуск в предварительно определенных тестовых случаях.

При динамическом тестировании определенных разделов кода, таких как модули или функции, обычно выполняется программа в отлаженной среде и используются заглушки и драйверы в процессе. Еще один способ провести

различие между методами статического и динамического тестирования - представить статическое тестирование как проверку, а динамическое тестирование - как проверку. Обычно методы обоих подходов необходимы при создании комплексного плана тестирования для программы.

Методы фронт-енд тестирования ПО также обычно подразделяются на методы «белого ящика» и «черного ящика».

Эти два метода используются для иллюстрации перспективы, которую принимают программисты, пишущие тесты, и степени их воздействия на исходный код при создании тестовых случаев.

Кроме того, существует третья категория методов, которая представляет собой комбинацию из первых двух, называемая «серым» тестированием.

Тестирование методом белого ящика основано на возможности просмотра исходного кода ПО. Тесты белого ящика проверяют внутреннюю работу программы в отличие от тестирования только функциональности, которая доступна конечному пользователю.

При разработке тестовых случаев белого ящика тестировщик самостоятельно решает, какие входные данные будут использоваться для тестов, и, какими должны быть соответствующие выходные данные.

Поэтому ему необходимы некоторые навыки программирования, чтобы понимать программную архитектуру системы и разрабатывать для нее содержательные тесты.

Метод тестирования белого ящика может быть реализован на уровнях интеграции и тестирования системы, но чаще всего он используется при тестировании на уровне устройства.

Тесты белого ящика можно использовать для модульного и интеграционного тестирования ПО.

Этот тип тестирования позволяет выявить множество ошибок и проблем, но он может не определить недостающие части спецификации или незавершенные требования к ПО.

Методики фронт-енд тестирования, основанные на методе белого ящика:

– тестирование API приложения с использованием его частных и общедоступных интерфейсов программирования приложений. Целью тестирования API является проверка правильности ответа API на конкретный запрос;

– покрытие кода - это уровень требований, который приложение должно выполнять на стадии тестирования. Например, предварительно определенный процент функций в приложении должен быть протестирован, чтобы соответствовать уровню покрытия кода;

– внедрение целенаправленных ошибки в приложение для определения эффективности стратегий тестирования;

– мутационное тестирование - это метод, при котором исходный код видоизменяется, чтобы увидеть, насколько хорошо тестовые примеры могут выявить проблему;

– статические методы испытаний и др.

Тестирование методом черного ящика рассматривает ПО как непрозрачный ящик, в котором исходный код не виден тестировщику.

При таком подходе функциональность ПО исследуется в условиях отсутствия знаний о внутренних функциях или структурах программы.

Единственное знание о программе, которое есть у инженера-тестировщика - это ожидаемый результат, но ничего не говорится о том, как программа его достигает.

Преимущество «черного ящика» заключается в том, что к тестировщику не предъявляются требования по наличию навыков программирования.

Поэтому мышление тестировщиков может отличаться от мышления программистов, и они пишут тесты, которые подчеркивают разные функциональные возможности ПО.

Результатом являются более надежные кейс-тесты.

С другой стороны, повышается риск написания излишне большого количества тестов для случая, который можно было бы проверить проще с помощью метода тестирования белого ящика.

Кроме того, тестировщики могут пропустить написание тестов для некоторых частей ПО в целом. Методы черного ящика можно применять ко всем уровням тестирования ПО, но чаще всего они используются в высокоуровневых тестах и в модульном тестировании.

Примеры методик, основанных на методе черного ящика:

– разделение входных данных на эквивалентные разделы данных для кейс-тестов;

– анализ граничных значений (применяется для разработки тестов, использующих диапазоны граничных значений входных данных);

– метод угадывания ошибок (предназначен для опытных тестировщиков, которые использует приобретенные знания о распространенных ситуациях сбоя ПО обеспечения для создания кейс-тестов);

– методика тестирования перехода состояния (основана на проверке изменения состояния программы для различных типов входных данных);

– методика тестирование прецедентов (применяется для поиска кейс-тестов, которые используют всю программу от начала и до конца. Каждый прецедент описывает взаимодействие пользователя с программой для достижения желаемого результата);

– методика тестирование на основе таблицы решений (объединяет возможные входы и выходы в таблице, чтобы продемонстрировать, какие результаты выдают те или иные сценарии).

Тестирование серого ящика - это комбинация методов черного и белого ящиков.

При тестировании в «сером ящике» тестировщик может обладать знаниями о внутренней структуре и функциях ПО, но тесты создаются с точки зрения конечного пользователя.

Этот метод тестирования может использоваться на любом уровне тестирования ПО, но чаще всего он используется на уровне интеграционного тестирования.

На рисунке 1.3 представлена пирамида фронт-енд тестирования, которая состоит из модульных, интеграционных и сквозных (end-to-end, E2E) тестов [43].

Рисунок 1.3 – Пирамида фронт-енд тестирования

Примеры методик, основанных на методе черного ящика:

– матричное тестирование (используется, чтобы убедиться в том, что ПО соответствует требованиям, установленным в его спецификации);

– регрессионное тестирование (гарантирует, что новые изменения в ПО не нарушат работу существующих функций);

– тестирование ортогональных массивов - это статистический способ тестирования системы со всеми возможными входами для выявления логических ошибок.

На основании произведенного анализа можно сделать вывод, что методологической основой фронт-енд тестирования являются базовые методы и подходы, используемые в классическом тестировании ПО.

Вместе с тем, использование того или иного метода тестирования зависит от уровня (вида) фроент-енд тестирования.

Так, на уровне модульного тестирования применяется метод «белого ящика» ящика.

На уровне интеграционного тестирования рекомендуется применение метода «серого» ящика. Соответственно, на уровне E2E тестирования используется метод «черного ящика»

1.3 Обзор методик фронтенд-тестирования

Большинство существующих технологий фронт-енд тестирования основано на использовании фреймворков.

Фреймворк - это программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

1.3.1 Модульное тестирование в среде фреймворка Jasmine

Jasmine - это фреймворк для разработки на основе поведения для JavaScript, который стал самым популярным выбором для тестирования приложений AngularJS [24].

Jasmine предоставляет функции, которые помогают структурировать ваши тесты, а также делать утверждения. По мере роста ваших тестов важно сохранять их хорошо структурированными и задокументированными, и Jasmine помогает в этом.

В Jasmine мы используем функцию описания, чтобы сгруппировать наши тесты вместе:

```
описать ("сортировка списка пользователей", function () {
  // отдельные тесты идут сюда
                                                    });
```

А затем каждый отдельный тест определяется в вызове it-функции:

```
описать ('сортировка списка пользователей', function () {
  it ('по умолчанию сортирует по убыванию', function () {
    // ваше тестовое утверждение находится здесь
  });
});
```

Группирование связанных тестов в описательных блоках и описание каждого отдельного теста в рамках it-вызова позволяет самодокументировать ваши тесты.

Наконец, Jasmine предоставляет сопоставители, которые позволяют делать утверждения:

```
описать ('сортировка списка пользователей', function () {
  it ('по умолчанию сортирует по убыванию', function () {
    var users = ['jack', 'igor', 'jeff'];
    var sorted = sortUsers (пользователи);
    ожидать (отсортировано) .toEqual (['Джефф', 'Джек', 'Игорь']);
  });
});
```

Jasmine поставляется с рядом сопоставителей, которые помогут вам делать различные утверждения. Вы должны прочитать документацию Jasmine, чтобы узнать, что это такое. Чтобы использовать жасмин с кармой, мы используем тестовый бегун карма-жасмин.

AngularJS также предоставляет модуль ngMock, который предоставляет имитацию для ваших тестов. Это используется для внедрения и имитации сервисов AngularJS в модульных тестах. Кроме того, он может расширять другие модули, чтобы они были синхронными.

Синхронизация тестов делает их намного чище и упрощает работу. Одна из самых полезных частей ngMock - это $ httpBackend, который позволяет нам имитировать запросы XHR в тестах и вместо этого возвращать образцы данных.

1.3.2 Модульное тестирование в среде фреймворка Jest

Jest — это фреймворк для тестирования JavaScript, разработанный для обеспечения уверенности в правильной работе любого JavaScript кода [18].

Он позволяет писать тесты с приемлемым, знакомым и функциональным API, и быстро достигать желаемых результатов.

Jest хорошо документирован, требует минимальной настройки и может быть расширен, чтобы соответствовать требованиям разработчиков.

Jest был разработан Facebook как библиотека для тестирования кода JavaScript. Первая версия среды была выпущена в 2014 году и быстро вызвала большой интерес, но потребовалось некоторое время, чтобы сообщество разработчиков полностью приняло ее.

Основной принцип Jest - предложить простую платформу для тестирования, которая не требует настройки.

Для достижения высокой эффективности Jest имеет смысл использовать в сочетании с библиотекой React, так как это также разработка Facebook.

React (иногда React.js или ReactJS) - JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов [6].

Как показывает практика, не рекомендуется тестировать слишком много компонентов React.

Если принять во внимание парадигму функционального программирования, большая часть бизнес-логики, которую следует тщательно протестировать, не должна включаться в сами компоненты, а должна быть написана как отдельные функции.

Тем не менее, бывают случаи, когда становится полезным тестировать взаимодействия React, например, проверять, что при нажатии на элемент вызывается правильная функция с ожидаемыми аргументами.

Чтобы проверить внутреннюю функциональность React, Facebook выпустил библиотеку ReactTestUtils, которая предоставляет базовую функциональность для тестирования приложений React.

Facebook также рекомендует использовать другую тестирующую утилиту, разработанную AirBnB, называемую Enzyme, которая предлагает простое исследование, манипулирование и просмотр выходных данных компонента React.

Когда компоненты React монтируются и пересекаются с использованием их свойств, они легко доступны для состояний и дочерних объектов.

Существует два способа монтажа компонентов с помощью Enzyme: монтаж и неглубокий монтаж. Разница между ними заключается в том, что при монтировании компонента загружается все дерево DOM, а при мелком монтировании только корневой компонент загружается в память.

1.3.3 Тестирование при помощи снимков в среде фреймворка Jest

Тестирование с использованием снимков - это очень полезный инструмент в ситуациях, когда необходимо быть уверены, что пользовательский интерфейс не изменяется неожиданным образом.

Тестирование компонентов React может занять много времени, даже для некоторых простых задач, таких как проверка того, что определенный текст был воспроизведен. Вместо того чтобы тестировать выходные данные компонентов React по отдельности, Jest предлагает тесты моментальных снимков.

Тесты моментальных снимков с Jest не особенно полезны при тестировании взаимодействий, но они являются очень эффективным способом проверки правильности вывода компонентов или представлений.

При выполнении теста моментальных снимков Jest визуализирует компонент или представление React и сохраняет выходные данные в отдельном файле в формате JavaScript-нотации объектов (JSON). Затем при каждом запуске тестов Jest создает новый вывод JSON и сравнивает его со снимком, который был сохранен ранее.

Если поведение компонента было изменено и снимки больше не совпадают, программист должен предпринять соответствующие действия для устранения проблемы.

Тестирование моментальных снимков в приложении React дает программисту возможность тестировать поведение и визуализацию компонентов без написания большого количества утверждений. Они также действуют как отказоустойчивые, чтобы убедиться, что поведение компонента не изменилось случайно. Это, однако, не означает, что каждый компонент должен иметь свой собственный тест снимка.

Лучший способ использовать тесты моментальных снимков - это выбрать компоненты, обладающие чрезвычайно важной функциональностью и наиболее важные для продолжения работы в соответствии с ожиданиями.

Создание снимков слишком большого количества компонентов может замедлить тестирование и создать дополнительную работу с постоянными обновлениями снимков.

Еще одним важным моментом является то, что React, разрабатываемая крупной технологической компанией, такой как Facebook, очень тщательно протестирована его разработчиками, и поэтому пользователям Jest не следует заканчивать тестирование инфраструктуры вместо собственного кода.

1.3.4 Покрытие кода в среде фреймворка Jest

Покрытие кода - мера, используемая при тестировании программного обеспечения. Она показывает процент исходного кода программы, который был выполнен в процессе тестирования.

Jest имеет встроенный инструмент покрытия кода под названием Istanbul.

Istanbul был разработан Кришнаном Анантесвараном из Yahoo и является популярным клиентским инструментом покрытия кода JavaScript.

Интеграция с Jest стала простой и не требует настройки от пользователя. Для сбора покрытия при выполнении тестов должно использоваться ключевое слово *--coverage*.

По умолчанию Jest выводит отчет о покрытии как в формате HTML, так и в консоли, но покрытие также может быть интегрировано для отображения покрытия в реальном времени с помощью плагинов текстового редактора.

Пример HTML-вывода покрытия кода в среде Jest представлен на рисунке 1.4 [35].

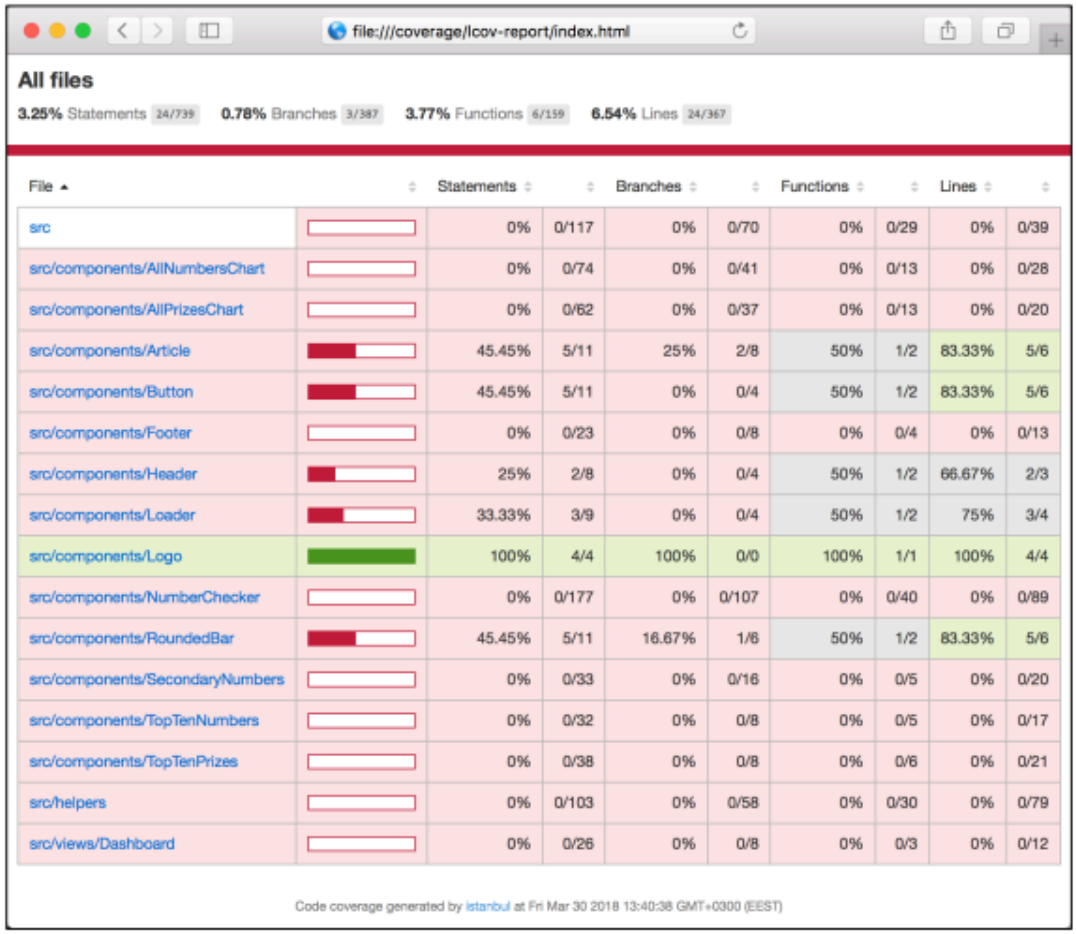

Рисунок 1.4 - Пример HTML-вывода покрытия кода в среде Jest

Раскраска строк таблицы показывает, насколько хорошо были протестированы определенные части программы.

Зеленый цвет указывает на хорошее покрытие, а красный указывает на отсутствие покрытия. Покрытие было разделено далее на операторы, ветви,

функции и строки, каждый из которых указывает на несколько разные части программы.

В представленном примере покрытие кода показывает, что большинству компонентов вообще не хватает тестов, а тестируемые также написаны только частично.

Единственный компонент с достаточным количеством тестов для достижения 100% покрытия - это логотип.

1.3.5 E2E тестирование в среде фреймворка Robot

Robot Framework - это фреймворк для разработки приемочных автотестов (ATDD) [21].

Robot был разработан Nokia Networks в 2005 году и выпущен как программное обеспечение с открытым исходным кодом в 2008 году.

Синтаксис кода применяет подход ключевых слов в тестах и поддерживает множество различных форматов для написания тестовых случаев. Поддерживаемые форматы для тестов: HTML-коды, разделенные вкладками, или в текстовом формате, использующем разделение пробелов или разделение каналов для полей.

Фреймворк может быть дополнительно расширен с помощью библиотек тестирования, которые реализованы на Python или Java, и предоставляет широкую экосистему, состоящую из нескольких готовых к использованию библиотек тестирования и инструментов, которые являются результатами различных проектов.

Поскольку платформа содержит модульную архитектуру, пользователи также могут создавать свои собственные ключевые слова, используя существующие, и, следовательно, создавать свои собственные служебные библиотеки для повторного использования.

Robot не зависит от конкретной операционной системы. Он был реализован с Python, но может также использоваться с Jython, который

является реализацией Python, предназначенной для работы на Java, или IronPython, который является реализацией Python для платформ .NET и Mono.

Модульная структура методики представлена на рисунке 1.5.

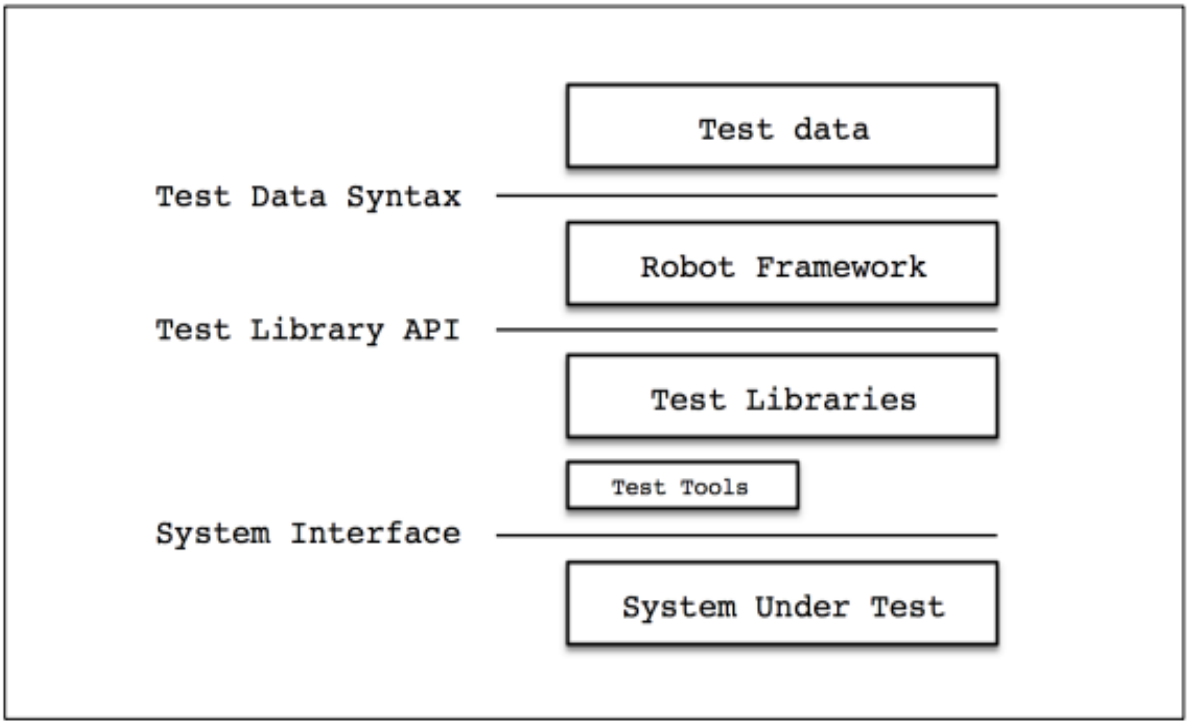

Рисунок 1.5 – Модульная структура методики E2E тестирование в среде Robot

Когда запускается Robot Framework, он получает тестовые данные и начинает выполнение кейс-тестов.

Затем он выводит результаты тестов в виде отчетов и журналов.

Сам фреймворк не имеет никаких знаний о тестируемой системе, так как связь проходит через тестовые библиотеки.

Библиотеки могут явно использовать интерфейсы приложений или дополнительно расширять использование с помощью инструментов тестирования, которые используются в качестве драйверов.

SeleniumLibrary, ранее известная как Selenium2Library, - это библиотека тестирования Robot Framework, предназначенная для веб-приложений.

Она использует модули Selenium WebDriver для автоматического управления веб-браузером.

В дополнение к WebDriver, Selenium состоит из многих других инструментов автоматизации браузера, которые можно использовать как плагин для браузера для записи и повторения взаимодействий в браузере для исследовательского тестирования, или как часть более широкой библиотеки, которая использует определенные его части, как в случай WebDriver в SeleniumLibrary.

С помощью SeleniumLibrary легко создавать E2E-тесты, которые выполняются в реальной среде браузера для веб-приложений.

Написание качественных E2E-тестов с помощью Robot Framework и автоматизация тестовых прогонов с помощью рабочего процесса непрерывной интеграции (Continuous integration, CI) обеспечивает надежное регрессионное тестирование для веб-приложения.

Сервер CI можно настроить таким образом, чтобы тесты становились частью конвейера развертывания, а затем запускались каждый раз, когда в хранилище помещается новый код.

Это гарантирует, что никакие критические изменения не повлияют на состояние системы пока выполняются тесты.

Анализ известных технологий фронт-енд тестирования показал, что их авторы отдают предпочтение фреймворкам, которые считаются наиболее успешными среди разработчиков для отдельных слоев пирамиды фронт-енд тестирования.

Так, для модульного тестирования рекомендуется использовать методику на основе фреймворка Jasmine, E2E тестирования - среда Robot, а тестирование при помощи снимков - среда фреймворка Jest.

Вместе с тем все рассмотренные методики и среды не содержат механизмы интеграции функций валидации кода, что снижает их эффективность использования в контексте темы исследования.

Выводы к первой главе

1. Следует констатировать недостаточность работ, посвященных проблематике моделирования и проектирования интегрированных сред фронт-енд валидации и тестирования разработчика.

2. Методологии и технологии фронт-енд валидации основаны на проверке кода в текстовых редакторах или интегрированных средах разработки и применении специализированных инструментов для валидации кода.

3. Методологической основой фронт-енд тестирования являются базовые методы и подходы, используемые в классическом тестировании ПО.

4. Пирамида фронт-енд тестирования состоит из модульных, интеграционных и сквозных или E2E тестов.

5. Большинство известных технологий фронт-енд тестирования основано на использовании фреймворков.

6. Рассмотренные методики и среды не содержат механизмы интеграции функций валидации кода и функционально избыточны на уровне фроент-енд тестирования, что снижает их эффективность использования в контексте темы исследования.

Глава 2 Принципы построения интегрированной среды валидации и фроент-енд тестирования разработчика

Ввиду того, что в специальной литературе и источниках нет четкого определения среды валидации и фронт-енд тестирования разработчика, предлагается рассматривать данную среду, как интегрированную, включающую в себя средства автоматизации валидации и фронт-енд тестирования.

Рассмотрим известные технологии автоматизации валидации и фронт-енд тестирования ПО.

2.1 Технологии автоматизированной валидации ПО

Следует констатировать недостататочность работ по автоматизированной валидации ПО.

Существует два подхода к валидации данных формы API: шаблон-ориентированный и реактивный [16].

При шаблон-ориентированном подходе формы полностью запрограммированы в шаблоне компонента. Он определяет структуру формы, формат ее полей и правила валидации.

Данный подход проще, так как его легче конфигурировать.

В отличие от шаблон-ориентированного подхода процесс создания реактивной формы состоит из двух этапов. Сначала необходимо создать модель в программном коде, а затем привязать элементы HTML к данной модели с помощью директив в шаблоне.

Используя реактивный подход, разработчик создает базовую структуру данных непосредственно в коде (а не в шаблоне). После создания модели связываете элементы шаблона HTML с моделью, задействуя специальные директивы с префиксом form*. В отличие от шаблон-ориентированных реактивные формы можно протестировать без участия браузера.

Реактивный подход легче тестировать, он более гибок и предоставляет возможность управления формой.

Необходимо учесть, что валидация на стороне клиента не может заменить валидацию на сервере.

Валидация на стороне клиента считается способом предоставления пользователю мгновенную обратную связь с минимизацией количества запросов на сервер, содержащих недопустимые данные.

Помимо описанных в разделе 1.1 инструментов ESLint и JSLint в качестве средства автоматизации валидации фроент-енд тестирования рекомендуется использовать альтернативный инструментарий.

В работе [36] описан алгоритм валидации HTML-страниц на сайте, построенный на применении такого инструмента, как Services_W3C_HTMLValidator.

Данный алгоритм состоит из следующих шагов:

Шаг 1. Для получения HTML-кода используется дополнительная программа веб-парсер Goutte [28]. Клиент работает только под PHP 5.3.

Шаг 2. Goutte возвращает HTML-код запрашиваемой страницы.

Шаг 3. Выполняется валидация полученного HTML-кода с помощью Services_W3C_HTMLValidator.

Пример валидации HTML-страницы на PHP приведен на рисунке 2.1.

```php
<?php
require_once 'Services/W3C/HTMLValidator.php';

$validator = new Services_W3C_HTMLValidator();
$validator->validator_uri = 'http://localhost/w3c-validator/check'; // http://validator.w3.org/check
$r = $validator->validate('http://google.com/');
if ($r->isValid()) {
  echo $r->uri . ' is valid!';
}
else {
  echo $r->uri . ' is NOT valid!';
}
```

Рисунок 2.1 – Пример кода валидации HTML-страницы

Некоторые разработчики используют JSON Formatter & Validator [31].

Данный инструментарий обеспечивает:

− фиксирование багов валидации;

− улучшенную логику валидации, в частности валидации строк;

− проверку нескольких стандартов JSON, включая текущие спецификации RFC 8259 и ECMA-404 и т.д.

Вместе с тем, в большинстве публикаций в качестве инструмента, который позволяет проводить анализ качества JavaScript кода рекомендуется использовать ESLint [15, 29].

Поэтому в качестве средства автоматизации валидации для интегрированной среды валидации и фроент-енд тестирования разработчика выбираем инструмент ESLint.

2.3 Принципы автоматизации тестирования

Автоматизированное тестирование предложено М. Коном, который представлял процесс автоматизации системы и проверки ПО в форме пирамиды [41].

Структура пирамиды тестирования изображена на рисунке 2.1.

Рисунок 2.2 – Пирамида автоматизированного тестирования

Пирамида автоматизированного тестирования состоит из следующих основных слоев (снизу-вверх):

1) автоматизированное тестирование модулей;

2) автоматизированное интеграционное тестирование;

3) автоматизированное сквозное (end-to-end) тестирование.

Как следует из рисунка, сквозное тестирование находится на самой вершине пирамиды тестирования, а модульное тестирование образует ее основание.

Верхний компонент пирамиды также включает тесты пользовательского интерфейса.

Элементы этой пирамиды отличаются друг от друга размером, поскольку они содержат различное количество тестовых случаев, необходимых для выполнения того или иного типа тестирования.

Представленная пирамида типична для тестирования программного обеспечения. Но есть много его модификаций.

Все зависит от типа тестируемого приложения.

Составные элементы пирамиды, а также их расположение могут быть изменены.

Иногда пирамида может быть изменена полностью.

Автоматизация тестирования программных продуктов рассматривается, как правило, в контексте оптимизации бизнес-процессов и сокращения затрат. В этом нет ничего удивительного: перевод тестов в автоматический режим порой выводит бизнес на принципиально новый уровень эффективности.

Как известно, тестирование пользовательского интерфейса имеет важное значение для комплексной стратегии тестирования, поскольку оно обеспечивает критическую обратную связь с точки зрения пользователя.

Однако это требует значительных усилий: проверку визуальных деталей, таких как изображения, цвета и шрифты, а также каждый аспект функционального поведения приложения, включая элементы управления, навигацию, сообщения об ошибках, обработку ввода данных и многое другое.

Комплексное тестирование GUI отнимает много времени и стоит дорого, особенно когда тесты должны повторяться как часть набора регрессии или для обеспечения совместимости между браузерами и между устройствами.

Автоматизированные тесты экономят время и затраты, выполняя часть времени, требуемую для ручного тестирования.

Автоматизация тестирования экономит системные ресурсы, работая в одночасье и параллельно во многих браузерах и на разных платформах.

Автоматизация также освобождает тестируемый персонал от рутинных испытаний, чтобы они могли сосредоточиться на более сложных и предварительных испытаниях.

Улучшенный охват тестами, возможный благодаря автоматизации тестирования, создает уверенность в том, что приложение готово к выпуску с качеством, требуемым пользователями.

Рассмотрим основные этапы автоматизации фронт-енд тестирования.

1) Подготовка — выбор бизнес-операций, подлежащих автоматизации тестирования, определение требований к системе автоматизированного функционального тестирования, согласование проектных сроков, выбор инструмента автоматизации, оценка возможных рисков.

2) Проведение — производится запуск автоматизированных тестов и проведение регрессионного автоматизированного тестирования, если необходимо.

3) Отчет — составляется итоговый документ с результатами тестирования, который содержит обнаруженные дефекты, отклонения от нормативов и предложения по улучшению системы. Создаются руководство пользователя и инструкции по настройке и сопровождению системы автоматизированного функционального тестирования.

Основные направлением автоматизации фронтенд-тестирования является GUI-тестирование — оценка функционирования графического интерфейса приложения.

Автоматизированное тестирование GUI предназначено для проверки работы приложения через графический интерфейс пользователя. Любые приложения, имеющие графический интерфейс пользователя, нуждаются в тестировании GUI, в том числе: web-приложения, клиентские настольные приложения, банковские приложения, платежные системы.

Основные задачи автоматизированного тестирования GUI:

– анализ графического интерфейса системы;

– разработка автоматизированных GUI тест-кейсов и подготовка тестовых данных;

– запуск автоматизированных тестов и составление отчета;

– поддержка автоматизированных тестов.

Преимущества автоматизированного тестирования GUI:

– имитация работы реальных пользователей системы при помощи автоматизированных скриптов;

– проверка многочисленных сценариев работы приложения, в том числе трудно поддающихся ручному тестированию;

– проверка корректности работы приложения на больших объемах тестовых данных без существенного увеличения трудозатрат.

Автоматизированное тестирование GUI позволяет эмулировать работу реальных пользователей с интерфейсом системы. При этом проверяется работа системы на соответствие техническому заданию при помощи современных инструментов автоматизированного тестирования.

В настоящее время общепринятым считается следующее определение автоматизированного тестирования ПО - это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования [3].

Автоматизация тестирования обеспечивает преимущества:

– экономический эффект от уменьшения затрат по сравнению с выполнением ручного тестирования;

– улучшение качества за счет исключения человеческого фактора, повторного использования автотестов на разных этапах разработки, регулярной проверки функциональности в процессе разработке и того, что QA-специалисты смогут уделить больше внимания проверкам, не покрытым автотестами;

– оптимизация объема тестирования – возможность проверить больший объем функциональности за то же время, а также применить более креативные методы тестирования;

– сокращение времени тестирования – регулярный запуск регрессионных тестов позволяет быстро обнаруживать старые дефекты, а уменьшение времени за счет автоматизации дает возможность быстрее выводить программный продукт на рынок.

При этом обеспечиваетсчя оптимизация ключевых задач тестирования:

– тестирование критически важной функциональности (например: функциональности, наличие ошибок в которой связано со многими рисками с точки зрения бизнес-логики или безопасности пользовательских данных);

– проведение регрессии – объем дымового тестирования увеличивается с выпуском каждой новой версии, но вся его суть сводится к проверке того факта, что ранее работавшая функциональность продолжает работать корректно;

– установка и настройка тестового окружения – автотесты пишутся для часто повторяющихся рутинных операций (например, проверка содержимого конфигурационных файлов или реестра) и подготовки приложения для запуска в определенной среде и с определенными настройками для проведения основного тестирования;

– тестирование безопасности – проверка прав доступа, паролей, уязвимостей текущих версий ПО и так далее, то есть, быстрое выполнение

очень большого количества проверок, в процессе которого нужно учесть большое количество параметров;

– тестирование скорости и надежности работы приложения под разной нагрузкой – создание нагрузки с точностью и интенсивностью, которые недоступны человеку, а также быстрый анализ большого объема данных или сбор большого набора параметров работы приложения;

– модульное тестирование – проверка правильности работы большого количества атомарных участков кода и взаимодействий таких участков кода;

– интеграционное тестирование – проверка взаимодействия компонентов в ситуации, когда почти нечего наблюдать, так как все тестируемые процессы проходят на более глубоких уровнях, чем пользовательский интерфейс;

– выполнение проверок, которые требуют сложных и точных математических расчетов;

– использование комбинаторных техник тестирования – генерация комбинаций значений и многократное выполнение тест-кейсов с использованием полученных комбинаций в качестве входных данных;

– различные «технические задачи» (например, проверка работы с базами данных, корректности протоколирования, файловых операций, поиска, корректности форматов и содержимого генерируемых приложением документов).

Несмотря на все плюсы автоматизации стоит помнить и о ее потенциальных рисках.

Частые изменения функционала, кода, структуры баз данных и других компонентов приложения подразумевают регулярное обновление написанных ранее автотестов.

В результате автоматизация тестирования ПО по времени нередко сопоставима с проведением ручных проверок.

Вопреки названию автоматизированное тестирования не является полностью автономным процессом и подразумевает активное участие

человека. Помимо запуска автотестов жизненный цикл автоматизированного тестирования включает оценку выгоды применения автотестов, поиск инструментальных средств, написание скриптов и подготовку тестовых данных, анализ результатов выполнения автотестов.

Исходя из сказанного выше, эффективность работы команды тестирования во многом зависит от того, какие именно задачи было решено автоматизировать и как эта автоматизация была проведена [10].

Любое приложение (в том числе – мобильное) можно условно разделить на 3 уровня:

– уровень компонентов (Unit layer) включает код приложения (например, переменные, функции, методы, библиотеки);

– уровень функциональности (Functional layer) – это бизнес-логика приложения, то есть, практический результат работы, для получения которого оно создано;

– уровень графического интерфейса пользователя (GUI layer) включает компоненты приложения, видимые конечному пользователю (например, экраны, кнопки, выпадающие списки).

Автоматизация приносит максимальный эффект процессу тестирования, если она затрагивает все уровни тестируемого приложения.

Автоматизация приносит максимальный эффект процессу тестирования, если она затрагивает все уровни тестируемого приложения (таблица 2.1).

Таблица 2.1 – Таблица соответствия уровней тестируемого приложения и типов автоматизированных тестов

Уровень	Тип автотеста
Уровень компонентов	Unit Tests
Уровень функциональности	Service Tests
Уровень интерфейса	App Tests, App Integration Tests, E2E tests

Автоматизация начинается на уровне компонентов. Автоматизированные юнит-тесты (Unit Tests) создаются для каждой новой возможности, добавленной в приложение.

Они позволяют быстро обнаруживать ошибки в коде.

Следующая ступень – уровень функциональности. Ему соответствуют сервисные автотесты (Service Tests), которые направлены на тестирование классов, образующих компонент в составе нового функционала.

Такие тесты запускаются только после успешного завершения юнит-тестирования.

Это тесты, предназначенные для проверки функциональности «в чистом виде». Обычно они запускаются на уровне API без использования графического интерфейса.

Если нужно протестировать взаимодействие с внешними сервисами, которые не могут гарантировать предоставление данных или по каким-то причинам недоступны, используются эмуляторы внешних сервисов.

На уровне интерфейса выполняются автотесты для проверки приложения в целом (App Tests), интеграции приложений (App Integration Tests) и полные сценарные тесты (End-to-End Test).

Автотесты для проверки приложения в целом отличаются глубиной проработки и большим объемом. Их цель – убедиться в корректности работы всего приложения.

Если приложение имеет объемный функционал, для тестирования оно может быть разбито на несколько отдельных приложений, предоставляющие пользователю разные возможности.

В описанном выше случае также применяются автотесты интеграции приложений, предназначенные для проверки взаимодействия «приложений внутри приложения» и корректности переключения между ними.

Полные сценарные тесты представляют собой автоматизированные GUI-тесты, которые запускаются для всей системы, воспроизводят типичные пользовательские пути или полные сценарии взаимодействия.

2.3 Разработка алгоритма автоматизации фроент-енд тестирования

С учетом вышеизложенного предлагается алгоритм автоматизации тестирования, который изображен в виде диаграммы деятельности UML (рисунок 2.3).

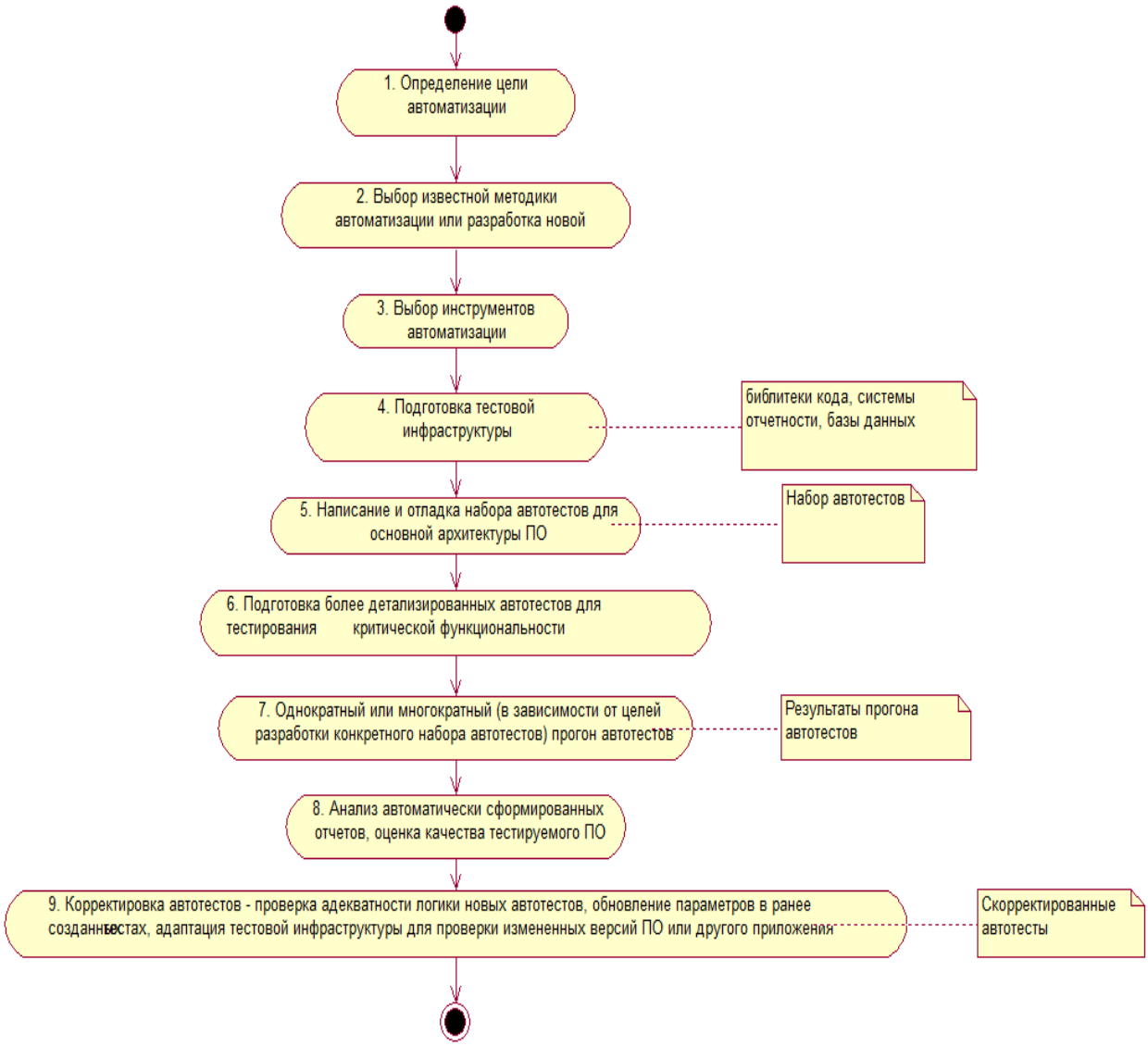

Рисунок 2.3 – Алгоритм автоматизации тестирования

Применение описанного алгоритма позволяет провести автоматизацию тестирования с учетом особенностей конкретного ПО.

Шаг 2 предлагаемого алгоритма предусматривает выбор методики автоматизации тестирования.

В настоящее время применяются 4 методики автоматизации тестирования:

– запись и воспроизведение скриптов;

– написание сценария;

– тестирование под управлением данными;

– тестирование на основе ключевых слов.

Запись и воспроизведение скриптов (Record and Play) – подразумевает использование утилит для записи действий пользователя в приложении. Программа преобразует запись в код и генерирует автотесты, которые в последствии выполняются без участия человека.

Основные преимущества – простота применения, не требуются знания в области программирования.

Главный недостаток – необходимость создания новых автотестов после внесения любых изменений в интерфейс ПО.

Обычно эта методика активно применяется для проведения дымового тестирования, а также однократного прогона однотипных тестов в разных окружениях.

Написание сценария (Scripting) – методика заключается в использовании тестовых сценариев, написанных на языках, специально разработанных для автоматизации тестирования ПО [5].

Основное отличие от методики «запись и воспроизведение скриптов» – код тестов создается людьми, а не программой.

Это требует серьезных временных и финансовых затрат, поскольку разработкой занимаются программисты или тестировщики с высокой квалификацией.

С другой стороны – такие тесты проще поддерживать и масштабировать, поскольку при написании кода вручную автор учитывает возможные изменения в названиях структурных элементов ПО, а также может согласовать эти изменения с остальными участниками проектной команды.

Тестирование под управлением данными (Data-driven testing).

Это методология создания скриптов и их верификации на основе данных, которые содержатся в базе данных или хранилище. Используется в случаях, когда нужно реализовывать однотипные проверки для различных комбинаций входных данных.

Тестирование на основе ключевых слов (Keyword-based testing) – методика написания автоматизированных тестовых сценариев, использующая подающиеся на вход файлы не только для хранения тестовых данных и ожидаемых результатов, но и ключевых слов, относящихся к тестируемому приложению.

Ключевые слова интерпретируются специальными процедурами, вызываемыми из управляющего сценария для данного теста.

Тесты, подготовленные в рамках этого подхода, представляют собой не программный код, а последовательность действий с их параметрами.

Как и первый подход, позволяет создавать автотесты тестировщикам, которые не имеют навыков программирования.

При этом автотесты под управлением ключевыми словами стабильнее и легче в поддержке, чем тесты типа Record and Play.

Для описания Keyword-based тестов используются ключевые слов, для реализации применяются фреймворки.

При выборе методики автоматизации тестирования для конкретного продукта нужно учитывать такие факторы: особенности предметной области и тип ПО, а также методологию разработки приложения.

Представленный алгоритм может быть использован для автоматизации фронт-енд тестирования.

2.4 Обзор и анализ инструментов автоматизации фронтенд-тестирования

Автоматическое тестирование упрощает обработку повторяющихся сценариев. Поэтому многие команды разработчиков включают ее в процесс [4].

Как результат возрастающего спроса на автоматизацию появились JavaScript-фреймворки для разных целей.

Рассмотрим известные инструменты автоматизации фронтенд-тестирования [8, 38].

Фреймворк Mocha.

Mocha - это многофункциональный тестовый фреймворк JavaScript, работающий на Node.js и в браузере, что делает асинхронное тестирование простым и увлекательным [19].

Тесты Mocha запускаются последовательно, обеспечивая гибкую и точную отчетность и отображая неперехваченные исключения в правильные тестовые случаи. Размещен на GitHub.

Mocha можно использовать с большинством известных библиотек утверждений JavaScript, включая:

- should.js
- express.js
- chai
- better-assert
- unexpected

Тесты в Mocha имеют улучшенное качество трассировки исключений и могут прогоняться сериями.

Преимущества:

- гибкое open-source приложение;
- легко поддерживает кодогенераторы;
- большое количество туториалов и документов в открытом доступе;

− последовательное выполнение тест кейсов с гибким составлением отчетов;

− простое сопоставление исключений с соответствующими кейсами.

Недостатки:

− слабый инструментарий;

− невысокая производительность и ограниченный набор функций;

− долгая установка и настройка.

Несмотря на это, многие разработчики используют этот фреймворк из-за развитой экосистемы.

Фреймворк Jasmine.

Jasmine - это поведенческая среда разработки для тестирования кода JavaScript. Jasmine не зависит от других JavaScript-фреймворков и не требует DOM [17].

И у него чистый, очевидный синтаксис, позволяющий относительно просто писать тесты.

Jasmine чаще всего используется для тестинга асинхронного кода, но у него много возможностей. Работает на Node.js и составляет точные и гибкие отчеты, периодически запуская кейсы.

Преимущества:

− гибкость. Главный плюс этого сервиса заключается в его совместимости со всеми JS-фреймворками и библиотеками;

− крупное комьюнити с множеством библиотек, туториалов и блог постов;

− простота освоения;

− удобные шаблоны.

Недостатки:

− требует долгой настройки. Пользователь должен выбрать и установить mocking фреймворк и библиотеку утверждений самостоятельно;

− библиотека для снапшот-тестинга сложна в интеграции;

− падающая популярность.

Следует отметить, что Jasmine хорошо подходит для масштабных проектов с интеграцией большого количества внешних библиотек.

Фреймворк Jest.

Jest - это высококачественный фреймворк для тестирования JavaScript с упором на простоту.

Он работает с проектами, использующими: Babel, TypeScript, Node, React, Angular, Vue и др.

Разработан Фейсбуком и используется для тестирования Javascript кода, особенно React JS приложений.

Jest позволяет писать тесты с доступным, знакомым и многофункциональным API, который быстро дает результаты.

Jest хорошо документирован, требует небольшой настройки и может быть расширен в соответствии с требованиями пользователя.

На рисунке 2.4 представлен пример отчета по тестированию покрытия кода.

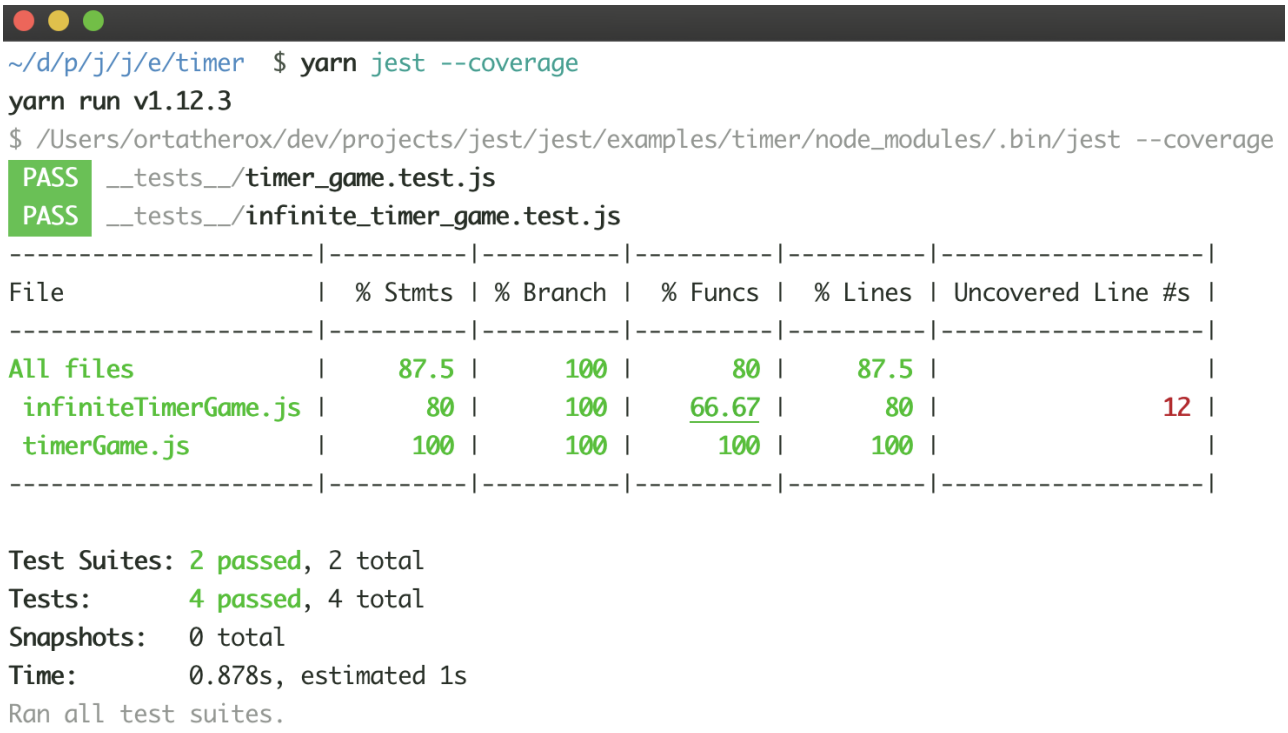

Рисунок 2.4 – Пример отчета теста покрытия кода

Преимущества:

- самый главный плюс Jest заключается в его в его простой настройке. Он идет в комплекте с mock-объектами и библиотекой утверждений. Тест кейсы написаны с BDD подходом;
- хорошая документация;
- предоставляет надежный инструментарий.
- позволяет проводить регрессионные тесты. Это особенно удобно для исправления багов с UI при react разработке. Делает скриншоты и сравнивает их с предыдущими, чтобы убедиться в том, что интерфейс не поехал.

Недостатки:

- поддерживает не так много библиотек;
- сложен в освоении;
- тестирование больших снапшотов невозможно.

Необходимо отметить, что Jest пользуется популярностью у фронтенд-тестировщиков и разработчиков.

Фреймворк Selenium WebDriver.

Selenium WebDriver - это веб-фреймворк, позволяющий выполнять кросс-браузерные тесты. Этот инструмент используется для автоматизации тестирования веб-приложений.

Selenium WebDriver позволяет выбрать язык программирования для создания тестовых сценариев.

Благодаря простой настройке WebDriver можно использовать со всеми основными браузерами (Firefox, Safari, Edge, Chrome, Internet Explorer и др.).

Обмен данными осуществляется двумя способами: WebDriver передает команды браузеру через драйвер и получает информацию обратно по тому же маршруту (рисунок 2.5) [2].

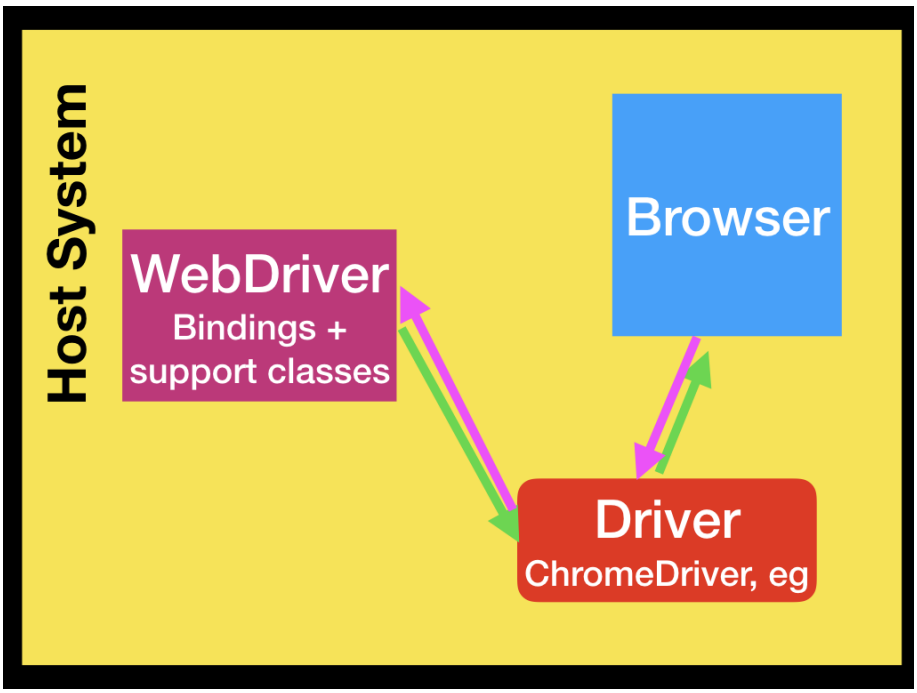

Рисунок 2.5 – Схема обмена данными Selenium WebDriver

Selenium WebDriver относится как к языковым привязкам, так и к реализации кода, управляющего отдельным браузером.

Поддерживает следующие типы тестирования:

- приемочное тестирование;
- функциональное тестирование;
- тестирование производительности, включая нагрузочное тестирование, и стресс-тестирование;
- регрессное тестирование;
- TDD и BDD.

Преимущества:

- простой в освоении IDE с открытым исходным кодом;
- долго присутствует на рынке. Поэтому у него большое комьюнити, где всегда можно найти решение для любой проблемы;
- несмотря на то, что у него свой язык, поддерживает связь с Java, JavaScript, PHP и др.

Недостатки:

− отсутствие официальной службы поддержки;

− для использования всего набора функций необходимо устанавливать плагины;

− ограниченная масштабируемость. Невозможно проводить параллельный тестинг. Поэтому специально для этой цели выпущен продукт Selenium Grid.

Комьюнити JavaScript разработчиков высоко оценивает Selenium и считает лучшим фреймворком для веб-приложений. Его автоматическое кросс-браузерное тестирование более полное, чем у конкурентов. Открытый исходный код закрепляет его место среди лидеров.

Кроме того, он работает не только с JavaScript, но и с другими языками программирования.

Фреймворк Cucumber.

По мнению разработчиков, Cucumber является лучшим инструментом для интеграционного тестирования. Cucumber разработан для облегчения разработки, управляемой поведением (BDD).

Фреймворк Protractor.

Для Angular-приложений разработан фреймворк Protractor [20]. Он автоматизирует End-to-End тестинг для SPA, написанных на Angular и Angular JS. Взаимодействует с приложением как настоящий пользователь, используя веб-браузер.

Protractor - это программа Node.js. Для запуска необходимо установить Node.js.

По умолчанию Protractor использует инфраструктуру тестирования Jasmine для своего интерфейса тестирования.

Преимущества:

− благодаря использованию Selenium WebDriver упрощается кросс-браузерное тестирование;

− наличие дополнительных локаторов (как repeater, model, binding и др.);

− легкое создание и управление объектами страницы;

− встроенная waits функция, которой нет, например, у WebDriver;

− если приложение разработано на основе BDD методологии, этот фреймворк идеален, так как он поддерживает Cucumber, Jasmine, Mocha и др.;

− автоматическая съемка скриншотов и быстрое их сравнение;

− поддержка параллельного выполнения кейсов с нескольких рабочих станций.

Недостатки:

− хорошо работает в Chrome, но есть проблемы поддержи других браузеров;

− нет поддержки Robot классов;

− маленькое комьюнити.

По мнению разработчиков, Protractor хорош для Angular-приложений, но из-за того, что Protractor работает только на Chrome, он не подходит для кросс-браузерного тестирования.

Для сравнительного анализа характеристик инструментов автоматизации фронтенд-тестирования используем таблицу 2, составленную на основе опроса разработчиков и тестировщиков [40].

Таблица 2.2 – Сравнительный анализ фреймворков для фронтенд-тестирования

Характеристика (1-5)	Mocha	Jasmine	WebDriver	Jest	Protractor
Простота настройки	2	3	4	5	5
Простота освоения	4	5	4	3	4
Комьюнити	3	5	5	4	3
Функциональность	3	4	4	5	4
Документация	4	5	3	5	4
Производительность	2	2	3	5	5
Итого	18	24	23	27	25

Как показал анализ, наилучшие характеристики, по мнению специалистов, у фреймворка Jest, что вполне объясняет его популярность

среди тестировщиков. Вместе с тем, использование того или иного инструмента для автоматизации тестирования зависит от объекта тестирования и требований к тестовым сценариям. На основе анализа составлена таблица рекомендуемых фреймворков по видам фроент-енд тестирования (таблица 1).

Таблица 2.3 Рекомендуемые фреймворки по видам фроент-енд тестирования

Вид тестирование	Фреймворк
Модульное тестирование	Jasmine
Интеграционное тестирование	Cucumber
E2E тестирование	Selenium

Как показывает практика, наилучшие результаты достигаются при применении комплекса инструментов для тестирования различных уровней фронтенд-разработки.

Выводы ко второй главе

1. Среда валидации и фроент-енд тестирования разработчика представляет собой интеграцию методов и средств автоматизации валидации и фронт-енд тестирования.
2. В большинстве публикаций в качестве средства автоматизированной поддержки валидации JavaScript кода рекомендуется использовать ESLint.
3. Эффективность среды валидации и фроент-енд тестирования зависит от того, как организована автоматизация отдельных уровней фронт-енд тестирования и насколько она позволит снизить влияние человеческого фактора на результаты тестирования.
4. При автоматизации тестирования по технологии проектирования Agile чаще всего используется методика Scripting, которая не охватывает проверки графического интерфейса.
5. Как показывает практика, наилучшие результаты достигаются при применении комплекса инструментов для тестирования различных уровней фронтенд-разработки.

Глава 3 Разработка и проверка адекватности модели среды валидации и тестирования фронт-енд разработчика

3.1 Разработка модели среды валидации и тестирования фронт-енд разработчика

Для разработки модели среды валидации и тестирования фронт-енд разработчика (далее – СВТФР) используем язык UML [14].

Для отражения функционального аспекта СВТФР построим диаграмму вариантов использования.

Диаграмма вариантов использования является одной из базовых диаграмм языка UML. Она инкапсулирует функциональность системы, включая варианты использования, участников (акторов) и их отношения.

Диаграмма вариантов использования моделирует задачи, сервисы и функции, требуемые системой / подсистемой приложения, а также показывает, как пользователь обращается с системой.

В ней сосредоточены требования к системе, включая внутренние и внешние воздействия.

Диаграмма вариантов использования представляет, как сущность из внешней среды может взаимодействовать с частью системы.

Таким образом, основная цель диаграммы вариантов использования состоит в решении следующих задач:

- отображение функционального аспекта системы;
- определения требований к системе;
- отображения внешний взгляда на систему;
- представления внутренних и внешних факторов, влияющих на систему.
- представления взаимодействия между акторами.

Для упрощения процесса построения диаграммы вариантов использования используется методология RUP (Rational Unified Process) [37].

RUP - это метод Agile-разработки программного обеспечения, при котором жизненный цикл проекта или разработка программного обеспечения делится на четыре фазы. На этих этапах проводятся различные мероприятия: моделирование, анализ и проектирование, внедрение, тестирование и применение.

Процесс RUP является итеративным (все основные действия процесса повторяются на протяжении всего проекта) и гибким (различные компоненты могут быть скорректированы, и фазы цикла могут повторяться, пока программное обеспечение не отвечает требованиям и целям).

Очень важно проанализировать всю систему перед тем, как начать создавать диаграмму вариантов использования, а затем определить функциональные возможности системы.

И как только все функциональные возможности идентифицированы, они затем преобразуются в варианты (сценарии) использования.

Далее следует выделить акторов.

Актор - это человек (действующее лицо) или объект (подсистема), взаимодействующий с системой.

Как только акторы и варианты использования определены, устанавливаются связи между конкретным актором и вариантом использования / системой.

Акторами в процессе фронт-енд тестирования являются: Тестировщик, фреймворки, инструментарий валидации кода.

Варианты использования (прецеденты) представлены в таблицах 3.1-3.4.

Таблица 3.1 – Модульное фронт-енд тестирование

Прецедент: Модульное фронт-енд тестирование
ID: 1
Краткое описание: выполнение модульного фронт-енд тестирования ПО
Главный актор: Тестировщик
Второстепенный актор: фреймворк Jasmine
Предусловие: разработка тест-кейсов

Продолжение таблицы 3.1

Постусловие: нет
Основной поток: Тестировщик выполняет модульное фронт-енд тестирование ПО
Альтернативные потоки: нет

Таблица 3.2 – Интеграционное фронт-енд тестирование

Прецедент: Интеграционное фронт-енд тестирование
ID: 2
Краткое описание: выполнение интеграционного фронт-енд тестирования ПО
Главный актор: Тестировщик
Второстепенный актор: фреймворк Cucumber
Предусловие: разработка тест-кейсов
Основной поток: Тестировщик выполняет интеграционное фронт-енд тестирования ПО
Постусловие: нет
Альтернативные потоки: нет

Таблица 3.3 – E2E фронт-енд тестирование

Прецедент: E2E фронт-енд тестирование
ID: 3
Краткое описание: выполнение E2E фронт-енд тестирования ПО
Главный актор: Тестировщик
Второстепенный актор: фреймворк Selenium
Предусловие: разработка тест-кейсов
Основной поток: Тестировщик выполняет E2E фронт-енд тестирования ПО
Постусловие: нет
Альтернативные потоки: нет

Таблица 3.4 – Описание прецедента: Валидация кода ПО

Прецедент: Валидация кода ПО
ID: 4
Краткое описание: выполнение валидации кода ПО
Главный актор: Тестировщик
Второстепенный актор: инструментарий ESLint

Продолжение таблицы 3.4

Предусловие: разработка тест-кейсов
Основной поток: Тестировщик выполняет валидаци. кода ПО
Постусловие: нет
Альтернативные потоки: нет

На рисунке 3.1 изображена диаграмма вариантов использования СВТФР, построенная на основе описанных рекомендаций.

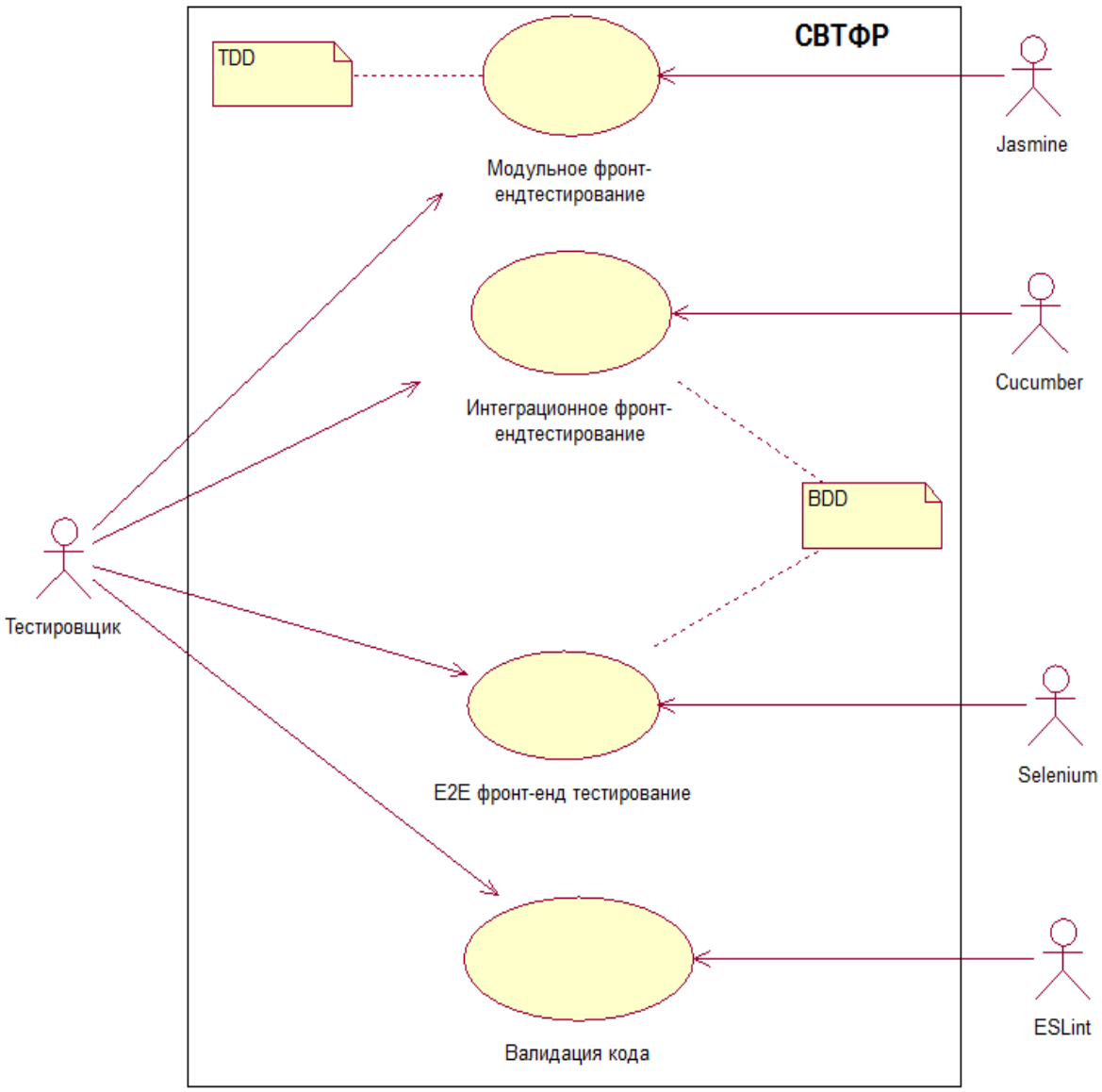

Рисунок 3.1 - Диаграмма вариантов использования СВТФР

Для поддержки модульного тестирования используются метод TDD (Test-Driven Development, Разработка через тестирование).

Диаграмма деятельности метода TDD представлена на рисунке 3.2

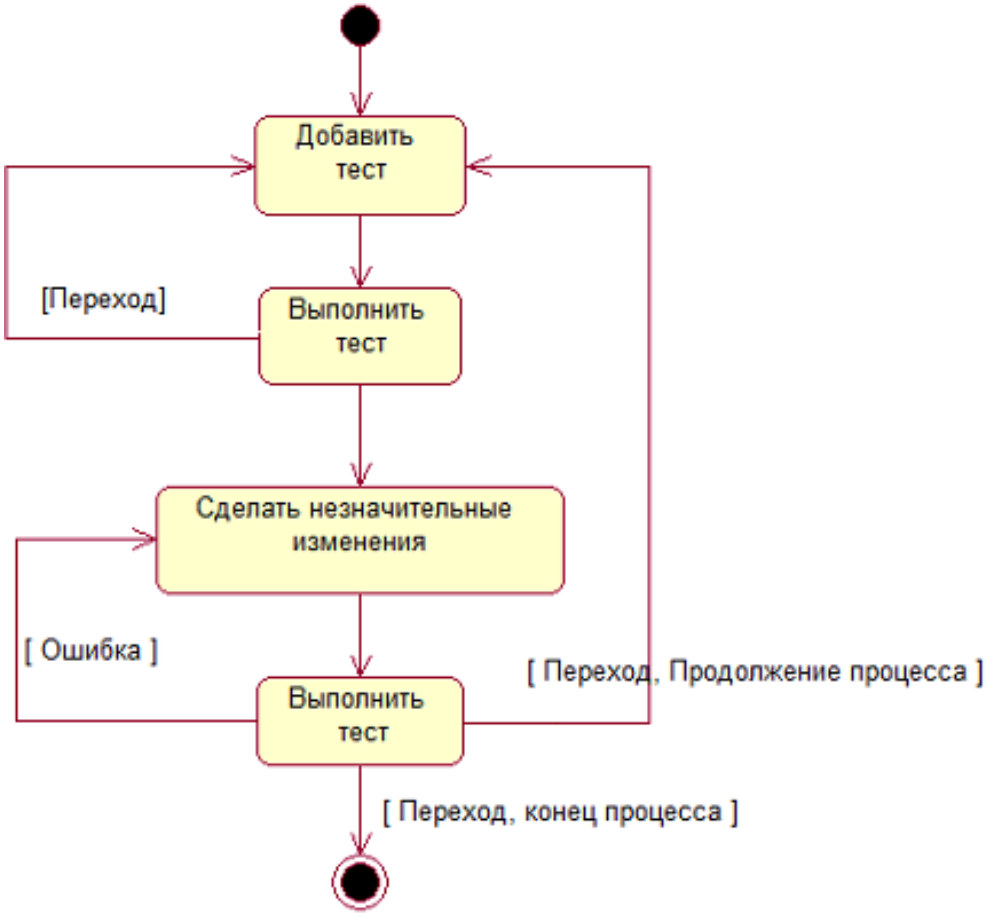

Рисунок 3.2 - Диаграмма деятельности метода TDD

В качестве средства автоматизации тестирования используется фреймворк Jasmine.

Для поддержки интеграционного тестирования используется метод BDD (Behavior-Driven Development, Разработка через поведение).

Блок-схема метода BDD представлена на рисунке 3.3

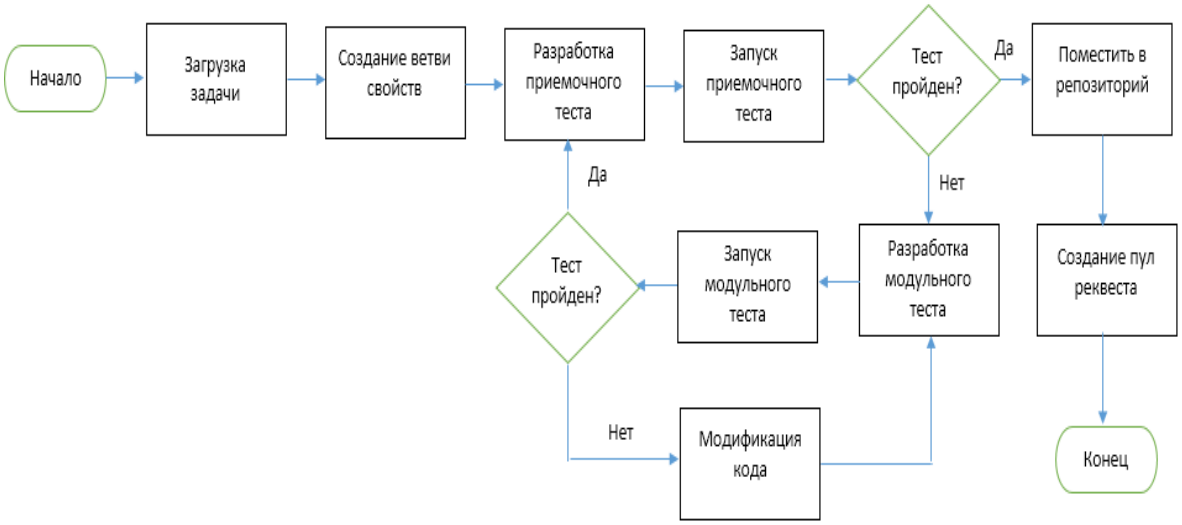

Рисунок 3.3 - Блок-схема метода BDD

В качестве средства автоматизации тестирования используется фреймворк Cucumber.

Для поддержки E2E тестирования используются метод BDD и фреймворк Selenium.

При разработке данной методики принят во внимание успешный личный опыт автора, а также опыт других разработчиков и тестировщиков, работающих в области фронт-енд тестирования.

Для отражения статистического аспекта СВТФР разработана диаграмма классов.

Диаграмма классов используется для визуализации, описания, документирования различных аспектов системы, а также для построения исполняемого программного кода.

Она показывает атрибуты, классы, функции и отношения, чтобы дать общее представление о системе программного обеспечения.

Диаграмма классов СВТФР представлена на рисунке 3.4.

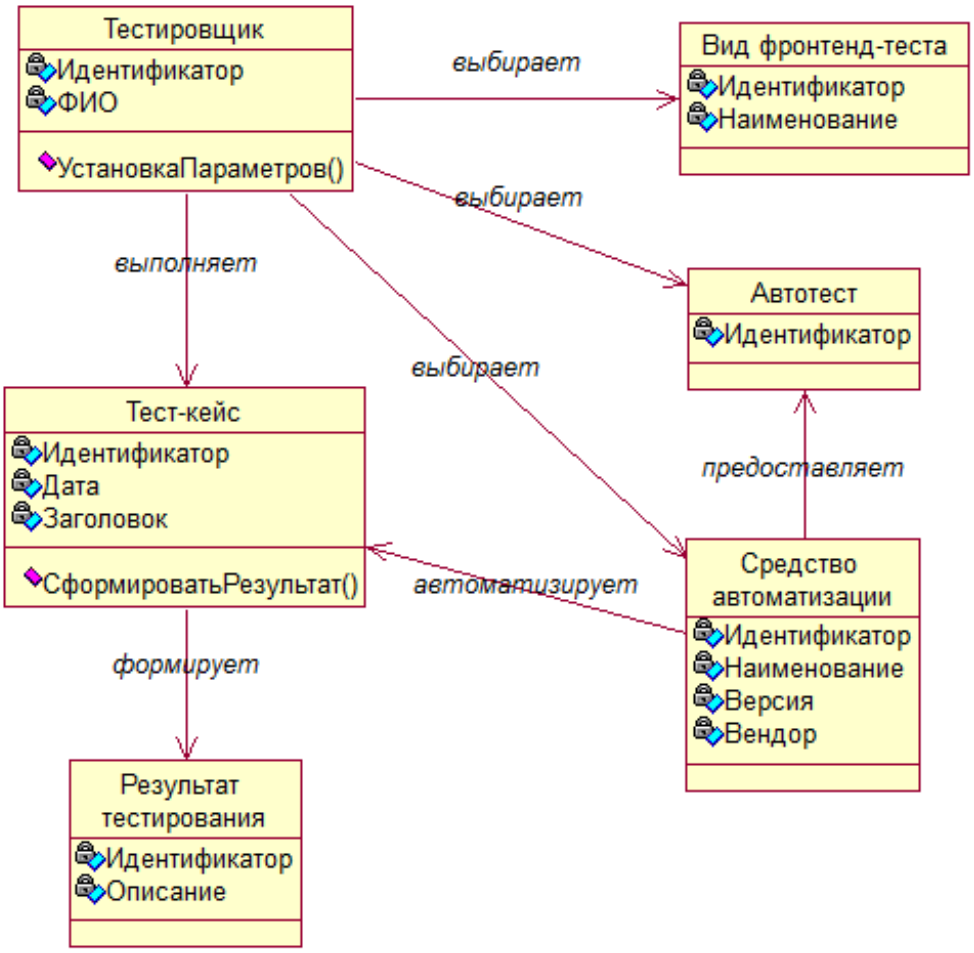

Рисунок 3.4 Диаграмма классов СВТФР

Для отражения динамического аспекта СВТФР использована диаграмма последовательности.

Диаграмма последовательности - это диаграмма взаимодействия, которая показывает объекты, участвующие в конкретном взаимодействии, и сообщения, которыми они обмениваются, упорядоченные во временной последовательности.

Использование диаграммы последовательности позволяет показать участников или объекты, участвующие во взаимодействии, и генерируемые ими события для отражения динамического разрабатываемой системы.

На рисунке 3.5 изображена диаграмма последовательности сценария выполнения фроент-енд-тестирования по видам тестирования.

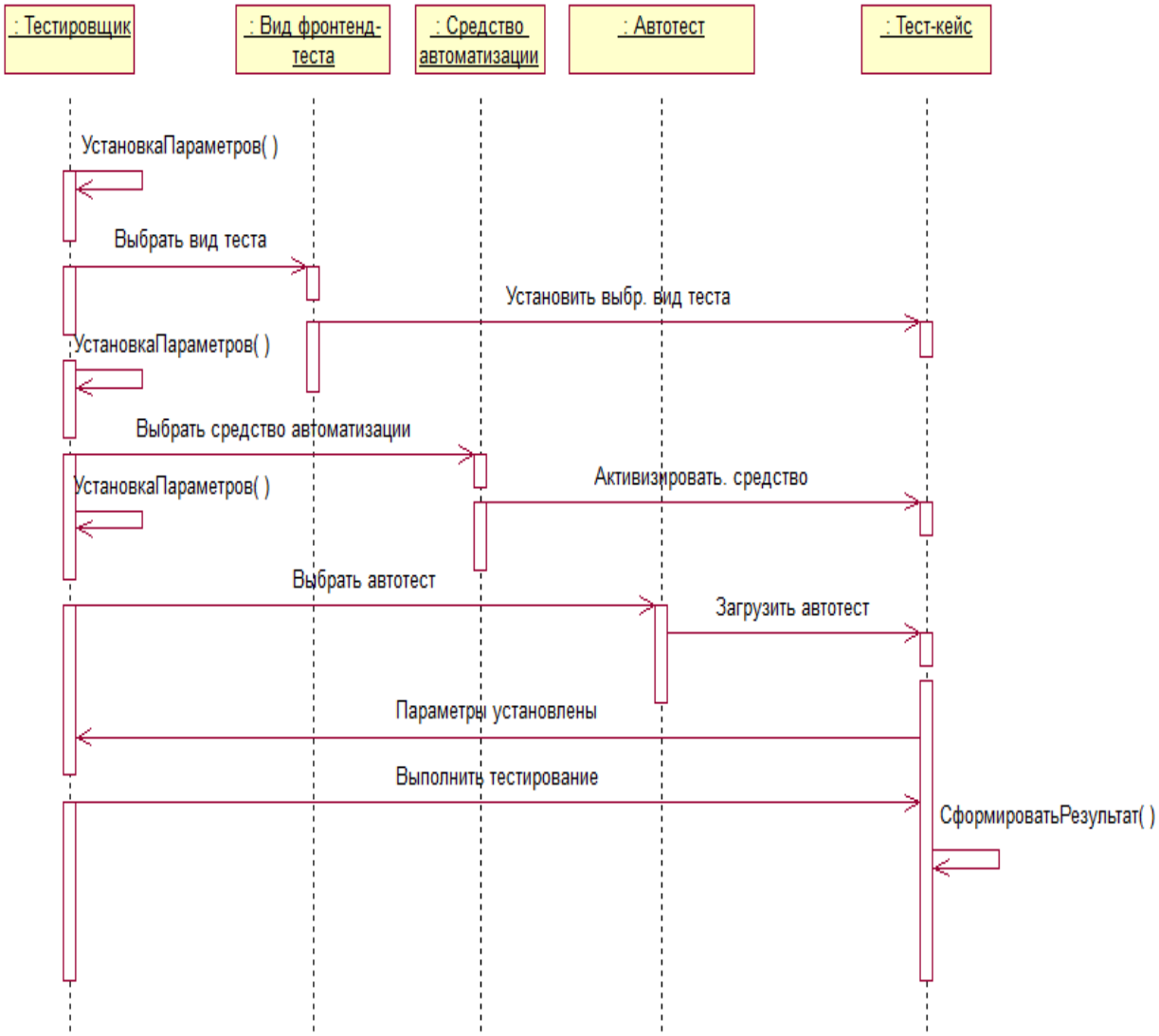

Рисунок 3.5 - Диаграмма последовательности сценария выполнения фроент-енд-тестирования по видам тестирования

На основе диаграммы классов построена логическая модель данных СВТФР фронтенд-тестированием.

Были выделены следующие основные сущности:

Тестировщик, Тест-кейс, Тестируемое ПО, Средство тестирования, Вид фронт-теста, Результат тестирования.

Логическая модель данных СВТФР представлена на рисунке 3.6.

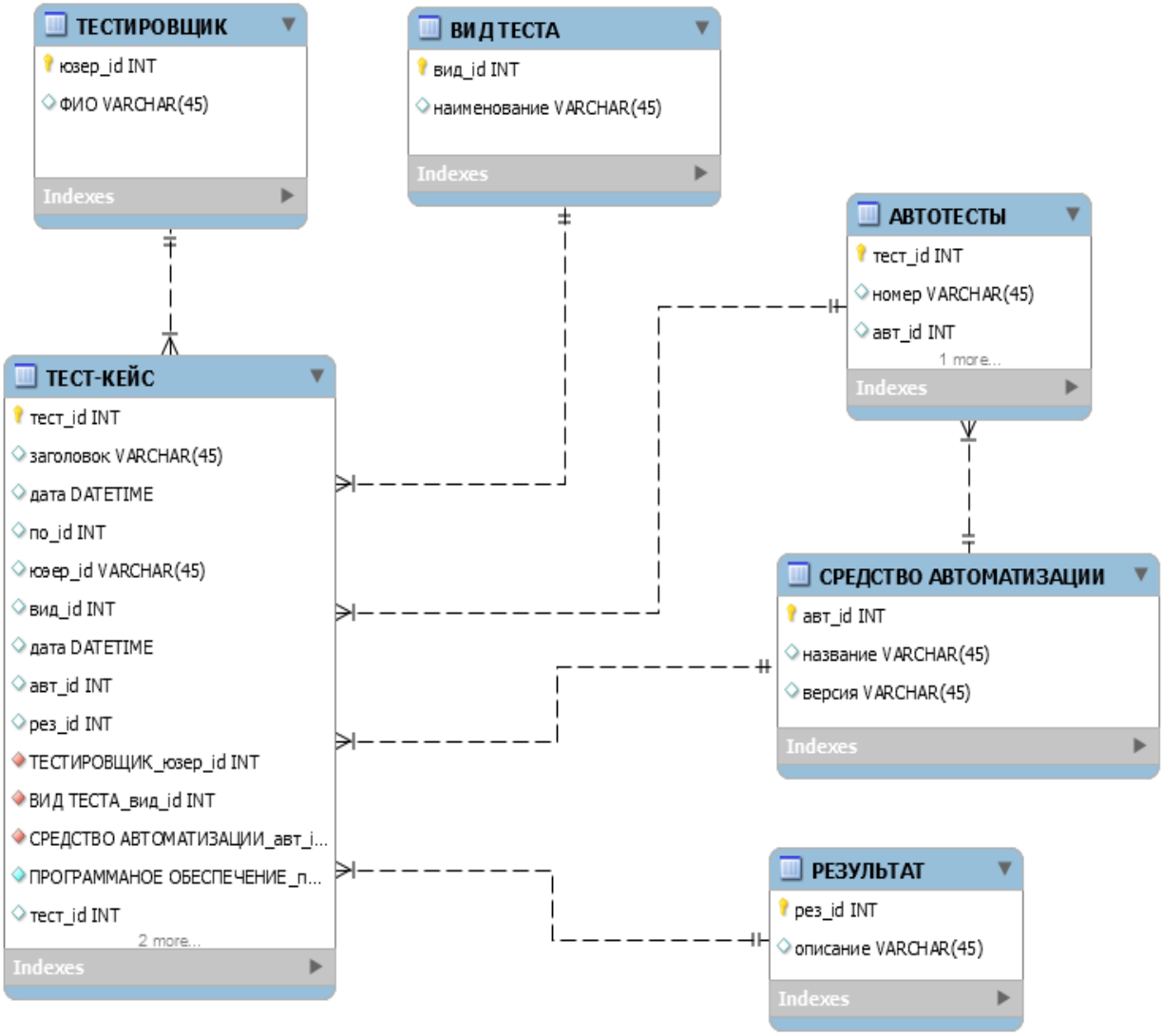

Рисунок 3.6 - Логическая модель данных СВТФР

Логическая модель данных системы фронтенд-тестирования разработана в среде MySQL Workbench [34].

3.2 Проверка адекватности модели среды валидации и тестирования фронт-енд разработчика

Для проверки адекватности предлагаемой модели СВТФР необходимо реализовать указанную среду и провести с ее помощью экспериментальное тестирование ПО.

Предварительно необходимо построить логическую архитектуру СВТФР.

Логическая архитектура - это структурный дизайн, который дает как можно больше деталей, не ограничивая архитектуру конкретной технологией или средой. Например, диаграмма, иллюстрирующая взаимосвязь между программными компонентами.

Для этого используем диаграмму пакетов UML.

Диаграммы пакетов UML используются для отражения организации ПО и его элементов.

Чаще всего диаграммы пакетов используются для организации диаграмм вариантов использования и диаграмм классов, хотя использование диаграмм пакетов не ограничивается этими элементами UML.

Элементы, содержащиеся в пакете, используют одно и то же пространство имен. Следовательно, элементы, содержащиеся в определенном пространстве имен, должны иметь уникальные имена.

Пакеты могут быть построены для представления физических или логических отношений.

При выборе включения классов в определенные пакеты полезно назначить классы с одинаковой иерархией наследования одному и тому же пакету.

Также есть веский аргумент в пользу включения классов, связанных посредством композиции, и классов, которые взаимодействуют с ними, в один пакет.

Пакеты представлены в UML 2.1 как папки и содержат элементы, совместно использующие пространство имен; все элементы в пакете должны быть идентифицируемыми и, следовательно, иметь уникальное имя или тип.

Пакет должен отображать имя пакета и, при желании, могут отображать элементы внутри пакета в дополнительных отсеках.

Логическая архитектура СВТФР изображена на рисунке 3.7.

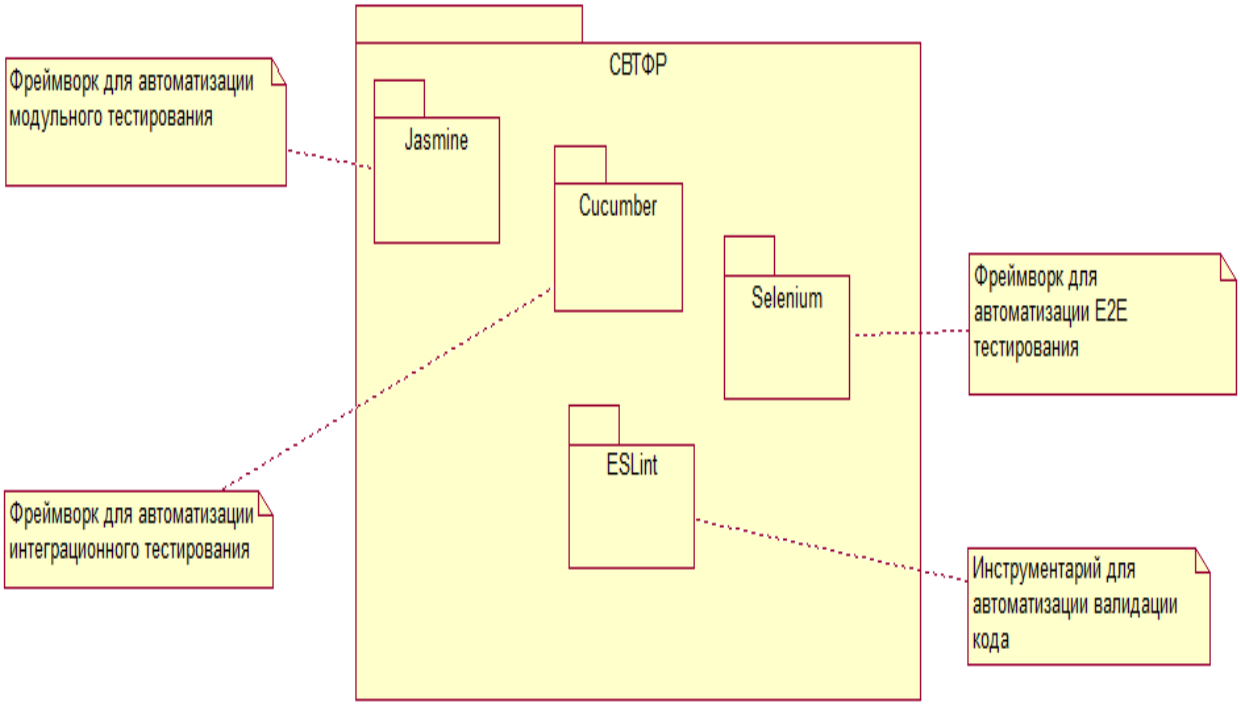

Рисунок 3.7 – Логическая архитектура СВТФР

Для управления СВТФР разработано веб-приложение, диаграмма развертывания которого представлена на рисунке 3.8.

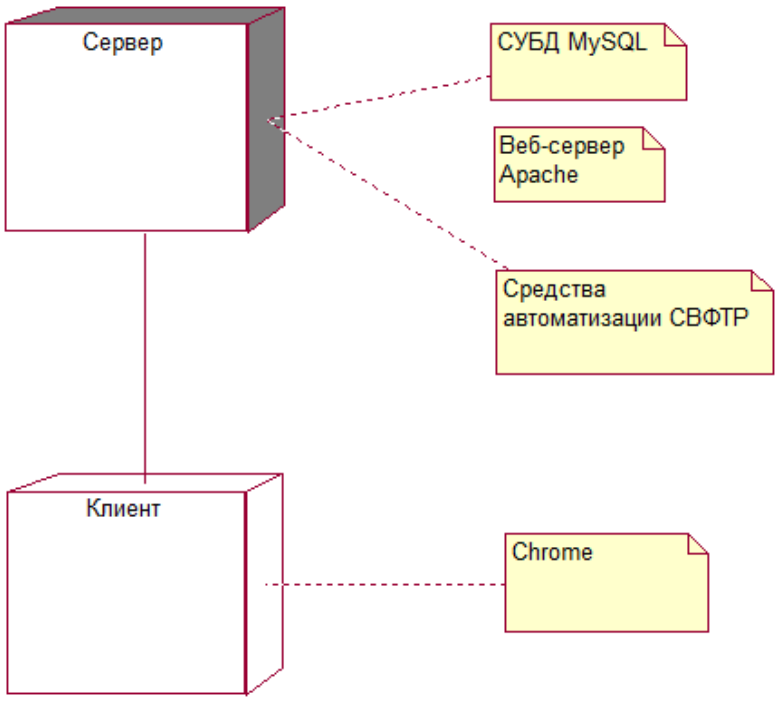

Рисунок 3.8 – Диаграмма развертывания веб-приложения СВФТР

Разработан алгоритм функционирования СВТФР (рисунок 3.9).

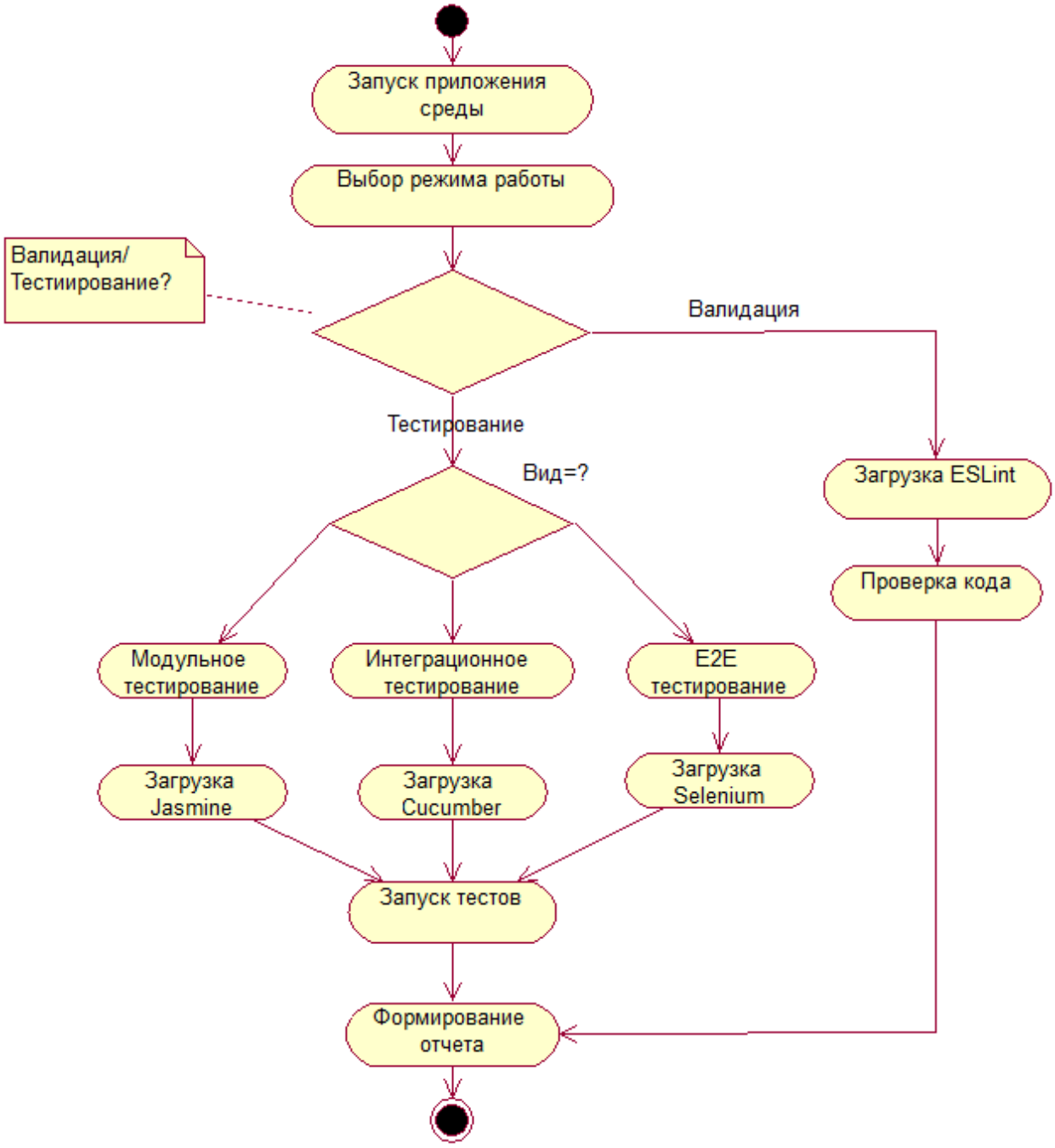

Рисунок 3.9 – Алгоритм функционирования СВТФР

Приложение реализовано на платформе CMS Wordpress [25].

Выбор платформы обусловлен Wordpress тем, что это инструмент для создания веб-сайтов с открытым исходным кодом, написанный на PHP в комплекте с СУБД MySQL или MariaDB.

По мнению разработчиков, это, вероятно, самая простая и мощная из

существующих сегодня CMS.

Экран главного окна приложения представлен на рисунке 3.10.

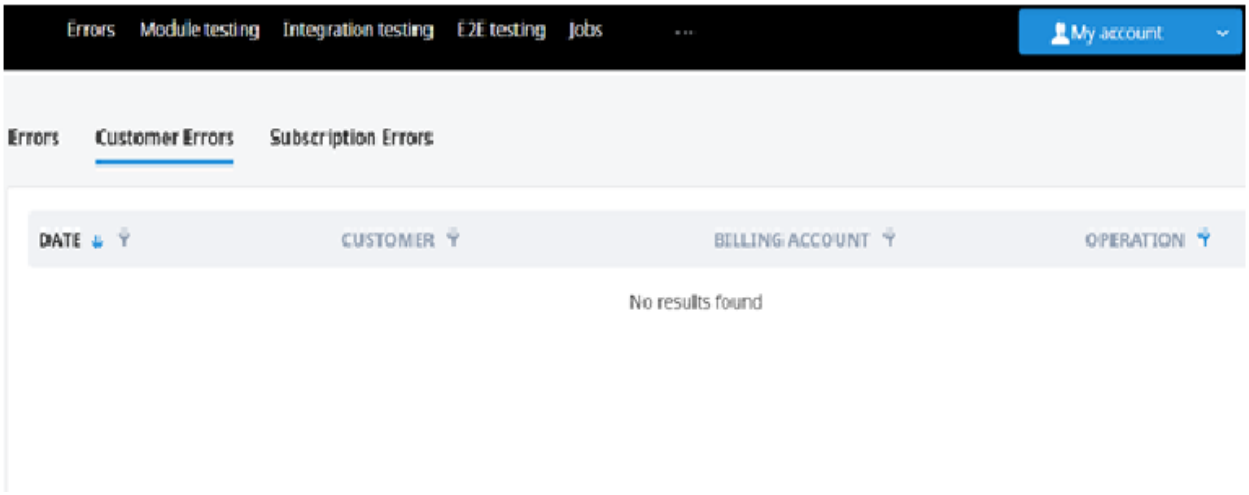

Рисунок 3.10 – Экран главного окна приложения

Программное обеспечение серверной части должно соответствовать требованиям:

– операционные системы FreeBSD, Windows или Linux;

– СУБД MySQL 7.1 или выше;

– Веб-сервер Apache версии 2.4 или выше;

– Язык программирования PHP версии 5.6 или выше.

Аппаратное обеспечение серверной части должно обеспечивать поддержку указанного программного обеспечения.

Программное обеспечение клиентской части должно включать веб-браузеры Mozilla Firefox, Google Chrome и др.

Аппаратное обеспечение клиентской части должно обеспечивать поддержку указанного программного обеспечения.

База данных приложения разработана на основе представленной выше логической модели данных СВФТР.

Как было отмечены выше, СВФТР должна обеспечивать следующую функциональность:

– автоматизированная поддержка фронт-енд валидации кода;

– автоматизированная поддержка фронт-енд тестирования.

Для валидации кода используется валидатор ESLint.

Пример результата проверки кода приведен на рисунке 3.11.

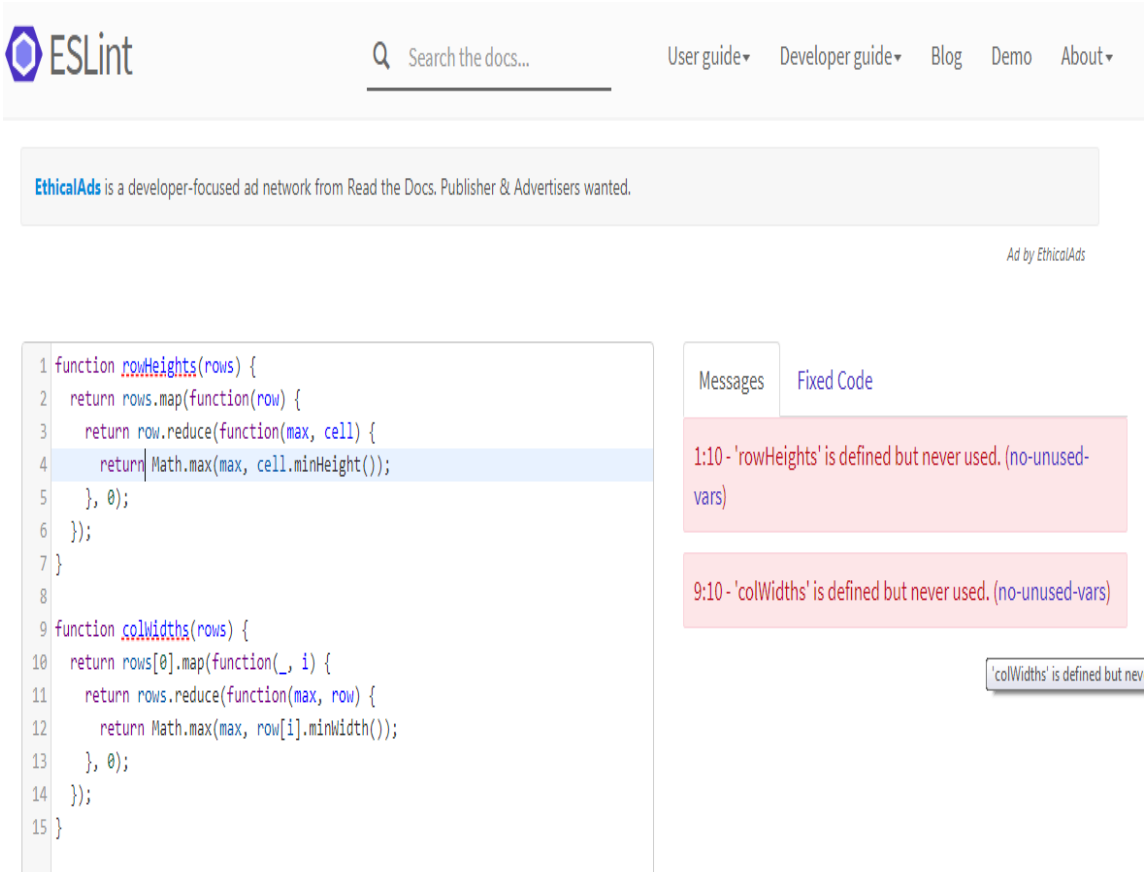

Рисунок 3.11 – Результаты проверки кода Javascript в валидаторе ESLint

Разработаны автотесты, предназначенные для всех видов фронт-енд тестирования.

Фрагменты автотестов по видам тестирования представлены ниже:

1) Тест для модульного тестирования с использованием в качестве средства автоматизации фреймворка Jasmine.

```
import {inject, TestBed} from "@angular/core/testing";
import {LiferayIntegrationService} from "../shared/services/liferay-integration/liferay-integration.service.api";
import {NotificationService} from "../shared/services/notification/notification.service";
```

```
import {CheckLocaleService} from "./check-locale.service";
import {LogService} from "../shared/services/log/log.service";
describe("CheckLocaleService", () => {
   let spyNotificationService;
   let spyLogService;
   let spyLiferayIntegrationService;
   beforeEach(() => {
      spyNotificationService = jasmine.createSpyObj("NotificationService",
["errorLocalize"]);
      spyLogService = jasmine.createSpyObj("LogService", ["error"]);
      spyLiferayIntegrationService =
jasmine.createSpyObj("LiferayIntegrationService", ["getHeaderPortletParams"]);
      TestBed.configureTestingModule({
         providers: [
            CheckLocaleService,
            {provide: NotificationService, useValue: spyNotificationService},
            {provide: LogService, useValue: spyLogService},
            {provide: LiferayIntegrationService, useValue:
spyLiferayIntegrationService}
         ]
      });
   });
   it("should be created", inject([CheckLocaleService], (service:
CheckLocaleService) => {
      expect(service).toBeTruthy();
   }));
   it("should notify", inject([CheckLocaleService], (service: CheckLocaleService)
=> {
spyLiferayIntegrationService.getHeaderPortletParams.and.returnValue({localeErro
rFlag: false});
```

```
        service.checkLocaleError();
        expect(spyNotificationService.errorLocalize).not.toHaveBeenCalled();
spyLiferayIntegrationService.getHeaderPortletParams.and.returnValue({localeErro
rFlag: true});
        service.checkLocaleError();
        expect(spyNotificationService.errorLocalize).toHaveBeenCalled();
    }));
});
```

2) Тест для интеграционного тестирования с использованием в качестве средства автоматизации фреймворка Cucumber.

```
        const { setWorldConstructor } = require('cucumber')

        class CustomWorld {
          constructor() {
            this.variable = 0
          }
            setTo(number) {
            this.variable = number
          }
            incrementBy(number) {
            this.variable += number
          }
        }
        setWorldConstructor(CustomWorld)
```

3) Тест для E2E тестирования с использованием в качестве средства автоматизации фреймворка Selenium WebDriver.

```
        /in e2e/tests/autocomplete.js
        const webdriver = require("selenium-webdriver");
        const driver = new webdriver.Builder().forBrowser("firefox").build();
        describe("login form", () => {
```

```
// e2e tests are too slow for default Mocha timeout
this.timeout(10000);
before(function(done) {
  driver
    .navigate()
    .to("http://path.to.test.app/")
    .then(() => done());
});

it("autocompletes the name field", function(done) {
  driver.findElement(By.css(".autocomplete")).sendKeys("John");
  driver.wait(until.elementLocated(By.css(".suggestion")));
  driver
    .findElement(By.css(".suggestion"))
    .click()
    .then(() => done());
});

after(function(done) {
  driver.quit().then(() => done());
});
});
```

Для вывода результатов тестирования использовано программное средство Karma [32].

Karma - это инструмент командной строки JavaScript, который можно использовать для создания веб-сервера, который загружает исходный код вашего приложения и выполняет ваши тесты. Вы можете настроить Karma для работы с несколькими браузерами, что полезно для уверенности в том, что ваше приложение работает во всех браузерах, которые вам необходимо поддерживать.

Karma выполняется в командной строке и отображает результаты тестов в командной строке после их запуска в браузере.

Это приложение NodeJS, которое следует устанавливать через npm / yarn.

Результаты по видам тестирования в среде Karma представлены на рисунках 3.12-3.14.

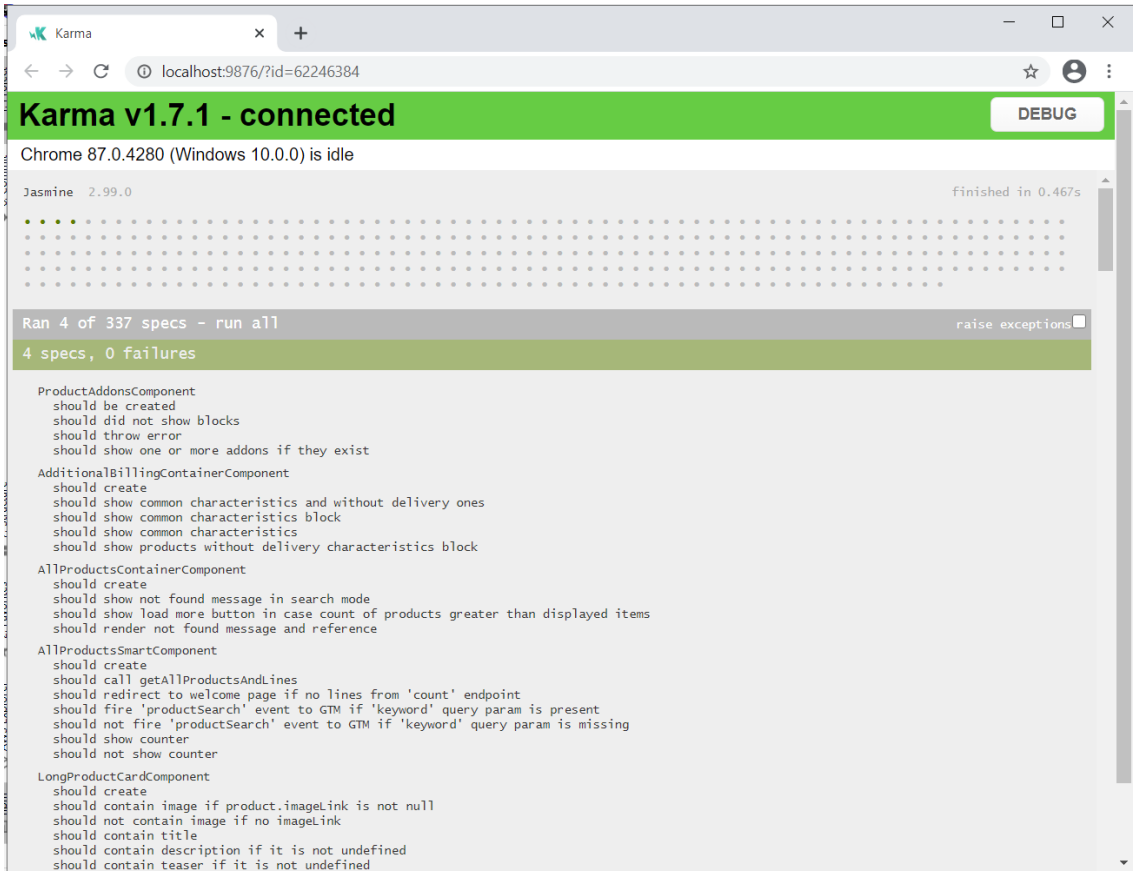

Рисунок 3.12 – Результат модульного тестирования

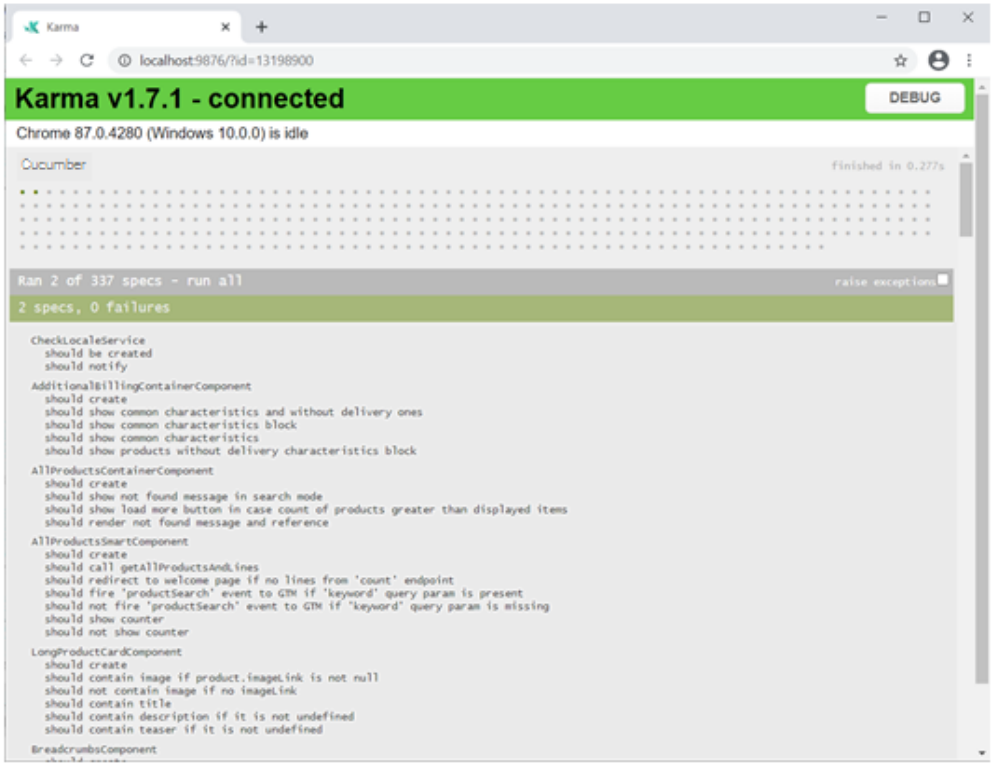

Рисунок 3.13 – Результат интеграционного тестирования

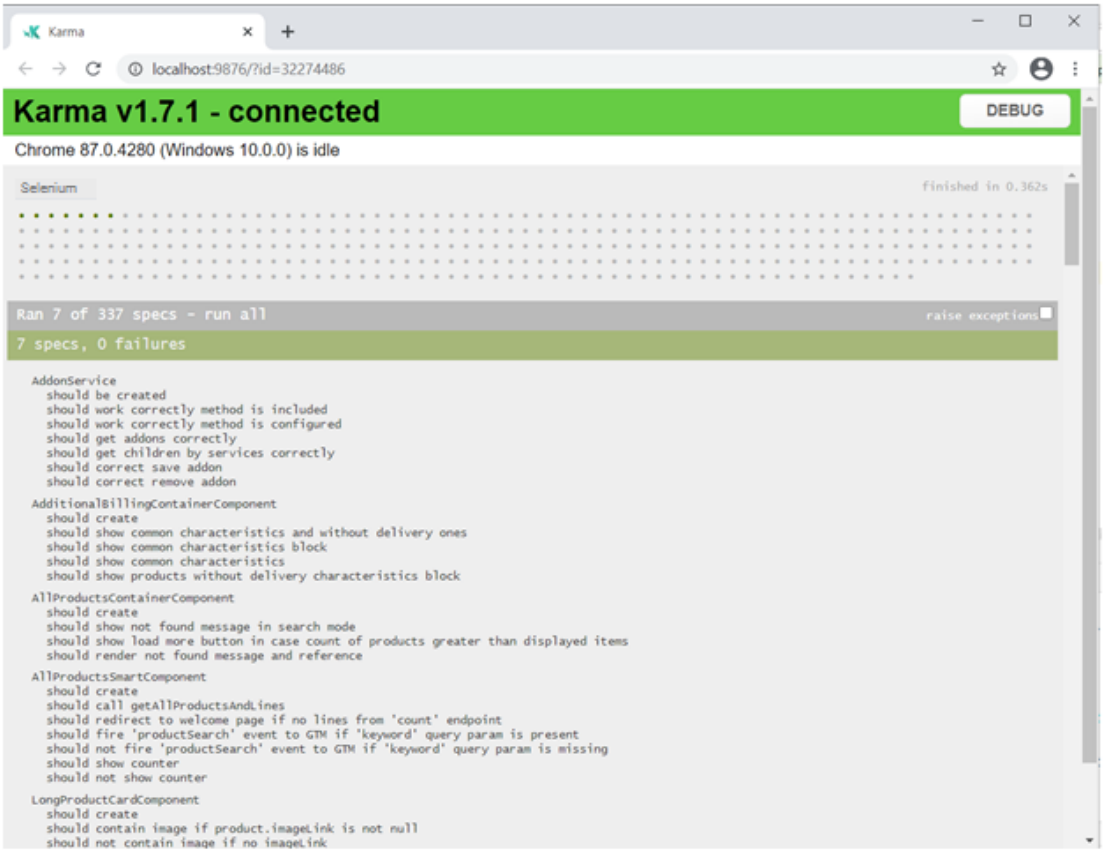

Рисунок 3.14 – Результат E2E тестирования

Таким образом, СВТФР, реализованная на основе предлагаемой модели позволяет решить задачи автоматизации фроент-енд тестирования разработчика.

Это подтверждает адекватность разработанной модели СВТФР.

Выводы к третьей главе

1. Для разработки модели СВТФР выбран язык UML. Для отражения функционального аспекта СВТФР построим диаграмму вариантов использования. Она инкапсулирует функциональность системы, включая варианты использования, акторов и их отношения. Для упрощения процесса построения диаграммы вариантов использования используется методология RUP. Для отражения статистического аспекта СВТФР разработана диаграмма классов. Диаграмма классов используется для визуализации, описания, документирования различных аспектов системы, а также для построения исполняемого программного кода. Для отражения динамического аспекта СВТФР использована диаграмма последовательности. На основе диаграммы классов построена логическая модель данных СВТФР фронтенд-тестированием.

2. Для проверки адекватности предлагаемой модели СВТФР выполнена ее реализация и проведено экспериментальное тестирование ПО. Предварительно разработана диаграмма пакетов UML, изображающая логическую архитектуру СВТФР. Для управления средой разработано веб-приложение.

3. СВТФР, реализованная на основе предлагаемой модели позволяет решить задачи автоматизации фроент-енд тестирования разработчика. Это подтверждает адекватность разработанной модели СВТФР.

Заключение

Важнейшей задачей фронт-енд разработчика программного обеспечения является создание интерфейса пользователя, отвечающего современным требованиям эргономики и дизайна.

Как показывает практика проектирования ПО, одной из причин снижения производительности приложений является неоптимизированный код на стороне клиента. Как показал анализ, известные средства валидации и тестирования функционально избыточны и недостаточно эффективны для задач фронт-енд разработки ПО, т.к. не учитывают специфику последней.

Совершенно очевидно, что для создания эффективной среды валидации и тестирования фронт-енд разработчика необходимо предварительно разработать адекватную модель указанной среды.

Магистерская диссертация посвящена актуальной проблеме разработки модели среды валидации и тестирования фронт-енд разработчика.

Выполненные в работе научные исследования представлены следующими основными результатами:

1. Проанализировано современное состояние проблемы исследования. Как показал анализ, методологической основой фронт-енд тестирования являются базовые методы и подходы, используемые в классическом тестировании ПО Большинство известных технологий фронт-енд тестирования основано на использовании фреймворков. Вместе с тем следует констатировать недостаточность работ, посвященных проблематике моделирования и проектирования интегрированных сред фронт-енд валидации и тестирования разработчика.

2. Дан обзор и выбраны методология и технология моделирования валидации и тестирования фронт-енд разработчика. Предложено рассматривать среду валидации и фроент-енд тестирования разработчика как интеграцию средств автоматизации валидации и фронт-енд тестирования.

При выборе методики автоматизации тестирования для конкретного ПО

нужно учитывать такие факторы: особенности предметной области и тип приложения, а также методологию, по которой оно разрабатывается. Как показал анализ, наилучшие результаты достигаются при применении комплекса фреймворков для решения различных задач фронтенд-тестирования.

3. Разработана модель автоматизированной среды валидации и тестирования фронт-енд разработчика. Для разработки модели среды выбран язык UML. Модель представляет собой комплекс базовых диаграмм UML – диаграммы вариантов использования, классов и последовательности.

4. Выполнена оценка функциональной эффективности среды валидации и тестирования фронт-енд разработчика, разработанной на основе предлагаемой модели. Для этого выполнена ее реализация и проведено экспериментальное тестирование ПО. Среда, реализованная на основе предлагаемой модели позволяет решить задачи автоматизации фроент-енд тестирования разработчика. Это подтверждает адекватность разработанной модели среды валидации и тестирования фронт-енд разработчика.

Таким образом, в работе решена актуальная научно-практическая проблема разработки модели эффективной среды валидации и тестирования фронт-енд разработчика.

Гипотеза исследования подтверждена.

Значение диссертационной работы определяется тем, что в ее рамках исследованы возможности повышения эффективности валидации и тестирования фронт-енд разработчика.

Список используемой литературы и используемых источников

1. 34 лучших инструмента для frontend-разработчика [Электронный ресурс]. URL: https://skillbox.ru/media/code/34_luchshikh_instrumenta_dlya_frontend_razrabotchika/ (дата обращения: 27.09.2020).

2. 8 лучших JavaScript-фреймворков для тестирования в 2019 г. [Электронный ресурс]. URL: https://digitalskynet.ru/blog/8-best-javascript-frameworks-for-testing-2019 (дата обращения27.09.2020).

3. Автоматизированное тестирование программного обеспечения – основные понятия [Электронный ресурс]. URL: http://www.protesting.ru/automation/ (дата обращения: 27.09.2020).

4. Автоматизированное тестирование, автоматизация тестирования приложений [Электронный ресурс]. URL: https://daglab.ru/avtomatizirovannoe-testirovanie-avtomatizacija-testirovanija-prilozhenij/ (дата обращения: 20.01.2020).

5. Артефакты, необходимые для тестирования [Электронный ресурс]. URL: https://habr.com/ru/post/39056/ (дата обращения: 27.09.2020).

6. Библиотека React [Электронный ресурс]. URL:https://reactjs.org/blog/2017/09/26/react-v16.0.html (дата обращения: 27.09.2020).

7. Верификация и валидация [Электронный ресурс]. https://qalight.com.ua/baza-znaniy/verifikatsiya-i-validatsiya/#:~:text=%D0%92%D0%B0%D0%BB%D0%B8%D0%B4%D0%B0%D1%86%D0%B8%D1%8F%20(validation)%20%E2%80%93%20%D1%8D%D1%82%D0%BE%20%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81,%D0%B2%20%D1%81%D0%B5%D0%B1%D1%8F%20%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA%20%D0%BA%D0%BE%D0%B4%D0%B0%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B (дата обращения: 27.09.2020).

8. ГОСТ Р 53622-2009 Информационные технологии. Информационно-вычислительные системы. Стадии и этапы жизненного цикла, виды и комплектность документов.

9. ГОСТ Р 56922-2016. Системная и программная инженерия. Тестирование программного обеспечения.

10. Дастин Э. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация: Пер. с англ. / Э. Дастин, Дж. Рэшка, Дж. Пол. М.: из-во «Лори». 2003, 289 с.

11. Методы проверки и тестирования программ и систем [Электронный ресурс]. URL: https://intuit.ru/studies/courses/2190/237/lecture/6130 (дата обращения: 27.09.2020).

12. Нативная валидация как фреймворк [Электронный ресурс]. URL: https://habr.com/ru/company/yandex/blog/348240/ (дата обращения: 27.09.2020).

13. Савин А. Тесты - фронтенд [Электронный ресурс]. URL: https://medium.com/@oxmap/%D1%82%D0%B5%D1%81%D1%82%D1%8B-%D1%84%D1%80%D0%BE%D0%BD%D1%82%D0%B5%D0%BD%D0%B4-%D1%87%D0%B0%D1%81%D1%82%D1%8C-i-%D1%8E%D0%BD%D0%B8%D1%82-31d328a2407b (дата обращения: 27.09.2020).

14. Самуйлов С. В. Объектно-ориентированное моделирование на основе UML : учебное пособие / С. В. Самуйлов. Саратов : Вузовское образование, 2016. - 37 с. URL: http://www.iprbookshop.ru/47277.html (дата обращения: 27.09.2020).

15. Сложно о простом: ESLint в команде [Электронный ресурс]. URL: https://habr.com/ru/post/322550/ (дата обращения: 27.09.2020).

16. Файн Я., Моисеев А. Angular и TypeScript. Сайтостроение для профессионалов. СПб.: Питер, 2018. 464 с.

17. Фреймворк Jasmine. Официальный сайт [Электронный ресурс]. URL: https://jasmine.github.io/ (дата обращения: 27.09.2020).

18. Фреймворк Jest. Официальный сайт [Электронный ресурс]. URL: https://jestjs.io/ (дата обращения: 27.09.2020).

19. Фреймворк Mocha. Официальный сайт [Электронный ресурс]. URL: https://mochajs.org/ (дата обращения: 27.09.2020).

20. Фреймворк Protractor. Официальный сайт [Электронный ресурс]. https://www.protractortest.org/#/ (дата обращения: 27.09.2020).

21. Фреймворк Robot. Официальный сайт [Электронный ресурс]. URL: https://robotframework.org/ (дата обращения: 27.09.2020).

22. Фреймворк Selenium WebDriver. Официальный сайт [Электронный ресурс]. URL: https://www.selenium.dev/projects/ (дата обращения: 27.09.2020).

23. Фронтенд и бэкенд [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%BE%D0%BD%D1%82%D0%B5%D0%BD%D0%B4_%D0%B8_%D0%B1%D1%8D%D0%BA%D0%B5%D0%BD%D0%B4 (дата обращения: 27.09.2020).

24. An Introduction to Jasmine Unit Testing [Электронный ресурс]. URL: https://www.freecodecamp.org/news/jasmine-unit-testing-tutorial-4e757c2cbf42/ (дата обращения: 27.09.2020).

25. CMS WordPress [Электронный ресурс]. URL: https://ru.wordpress.org/ (дата обращения: 27.09.2020).

26. Dodds K. C. Learn the smart, efficient way to test any JavaScript application [Электронный ресурс]. URL: https://testingjavascript.com/ (дата обращения: 27.09.2020).

27. ESLint [Электронный ресурс]. URL: https://eslint.org/ (дата обращения: 27.09.2020).

28. Goutte, a simple PHP Web Scraper [Электронный ресурс]. URL: https://goutte.readthedocs.io/en/latest/ (дата обращения: 27.09.2020).

29. How to set up Eslint with Typescript in VS Code [Электронный ресурс]. URL: https://thesoreon.com/blog/how-to-set-up-eslint-with-typescript-in-vs-code (дата обращения: 27.09.2020).

30. JSLint [Электронный ресурс]. URL: https://www.jslint.com/ / (дата обращения: 27.09.2020).

31. JSON Formatter & Validator [Электронный ресурс]. URL: https://jsonformatter.curiousconcept.com/ (дата обращения: 27.09.2020).

32. Karma [Электронный ресурс]. URL: http://karma-runner.github.io/0.12/intro/installation.html (дата обращения: 27.09.2020).

33. Michas N. Backend Data Validations and Why You Need Them [Электронный ресурс]. URL: https://medium.com/better-programming/back-end-data-validations-73ea9004c6d7 (дата обращения: 27.09.2020).

34. MySQL Workbench [Электронный ресурс]. URL: https://www.mysql.com/products/workbench/ (дата обращения: 27.09.2020).

35. Paulasaari M. Tools for Code Quality in Front-end Software Development, Helsinki Metropolia University of Applied Sciences, 2018.

36. QA: автоматизация валидации HTML-страниц [Электронный ресурс] URL: http://tigor.com.ua/blog/2011/06/16/qa-automating-validate-frontend-html/ (дата обращения: 27.09.2020).

37. Rational Unified Process [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Rational_Unified_Process (дата обращения: 27.09.2020).

38. Testing Framework and Tools [Электронный ресурс]. URL: https://geekflare.com/javascript-unit-testing/ (дата обращения: 27.09.2020).

39. The Role of a Front-End Web Developer: Creating User Experience & Interactivity [Электронный ресурс]. URL: https://www.upwork.com/hiring/development/front-end-developer/ (дата обращения: 27.09.2020).

40. The State of JavaScript 2018: Testing overview [Электронный ресурс]. URL: https://2018.stateofjs.com/testing/overview/ (дата обращения: 27.09.2020).

41. What Is Automation Testing Pyramid? [Электронный ресурс]. URL: https://ru.qatestlab.com/resources/knowledge-center/test-automated-pyramid/ (дата обращения: 27.09.2020).

42. What is Front End Testing? Tools & Frameworks [Электронный ресурс]. URL: https://www.guru99.com/frontend-testing.html (дата обращения: 27.09.2020).

43. Yerburgh E. The frontend testing pyramid [Электронный ресурс]. URL: https://livebook.manning.com/book/testing-vue-js-applications/the-frontend-testing-pyramid/42 (дата обращения: 27.09.2020).