# Содержание

# Введение

В настоящее время многие IT-проекты имеют сложную разнообразную функциональность и отличаются очень короткими промежутками между релизами. Поэтому разработчики вынуждены часто выполнять большое количество повторяющихся проверок (регрессионных тестов). По всей вероятности, именно этот факт является одной из причин активного развития автоматизации тестирования. Всё больше IT-компаний приходят к решению оптимизации процесса тестирования, что позволит им сократить финансовые и временные затраты.

Для упрощения автоматизации тестирования применяют вспомогательные инструменты для записи тестов, так называемые, рекодеры. Их принцип действия следующий. Пользователь совершает какие-то манипуляции в браузере, а его действия записываются в виде последовательности незамысловатых команд. Далее сохраняя этот тест, его можно запускать для самостоятельного повторения браузером. Благодаря таким рекодерам, легко производится автоматизация рутинных проверок и уменьшается количество ручной работы.

В рамках выпускной квалификационной работы была поставлена задача – разработать для предприятия НПО «Криста» инструмент, позволяющий фиксировать действия пользователей и генерировать код автоматизированных тестов, адаптированных под корпоративную библиотеку autotest-lib, для проверки функций интерфейса в программных продуктах компании с применением веб-технологий. При его разработке требуется решить такие задачи, как проектирование плагина (расширения) для браузера с механизмом регистрации действий в веб-приложении и их трансформацию в код автоматизированных тестов, проектирование базы данных, их реализация, тестирование и документирование.

Ожидается, что разрабатываемый плагин упростит процесс автоматизации тестирования для интерфейса веб-приложений, сократит время на написания автоматизированных тестов и увеличит количество создаваемых тестов.

Для теоретического обоснования выдвинутой гипотезы предполагается проведение анализа проблемы и существующих решений с дальнейшим синтезированием для получения модели предметной области и формирования требований к разрабатываемому модулю. Подобные процессы анализа и синтеза применяются на каждом этапе разработки для постановки задачи и нахождении наиболее оптимального способа ее решения.

# 1. Описание предметной области

## 1.1. Общие сведения о предметной области

### 1.1.1. О тестирование и его видах

Тестирование программного обеспечения – это проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом [1]. Тестирование необходимо как разработчику программного продукта, чтобы убедиться в его готовности, так и заказчику, чтобы удостовериться, что их деньги не пропали впустую.

Все виды тестирования ПО, в зависимости от преследуемых целей, условно делятся на следующие группы:

- Функциональные;
- Нефункциональные;
- Связанные с изменениями [1].

Подробнее рассмотрим функциональное тестирование, поскольку основной задачей разработки будет являться генерирование, именно, функциональных тестов для интерфейса, проверяя бизнес-функции проектов.

Функциональное тестирование обозревает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента и системы в целом. Данные тесты основываются на функциях, выполняемых системой, и могут быть представлены на всех уровнях тестирования: компонентном или модульном (Component/Unit testing), интеграционном (Integration testing), системном (System testing) и приемочном (Acceptance testing).

Тестирование функциональности может проводиться с двух сторон:

- требования;
- бизнес-процессы.

Тестирование в ракурсе «бизнес-процессы» использует знание этих самых бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии опираются на случаях использования системы (use cases) [1].

В зависимости от того, используют ли тестировщики дополнительные программные средства для тестирования ПО, выделяют следующие виды тестирования:

- Мануальное (ручное) – сам разработчик подготавливает тестовые данные и «вручную», без использования дополнительных программных средств, проводит тестирование;

- Автоматизированное – разработка автоматизированных тестов с и

спользованием программных средств для повышения эффективности тестирования.

Каждый подход имеет свои преимущества и недостатки. Автоматизация требует базовых навыков программирования, позволяет получить более точный результат и по возможности исключает «человеческий фактор». Ручное же тестирование проще освоить, считается более универсальным вариантом, который применяется на проектах всех типов. Также мануальные проверки отличаются однообразием и медлительностью [2].

Чтобы определить необходимо ли нам писать автоматизированные тесты, нужно ответить на вопрос «перевешивают ли в нашем случае преимущества автоматизации перед её недостатками?» − хотя бы для некоторой функциональности нашего проекта. Автоматизация тестирования имеет такие достоинства как:

- Повторяемость – все написанные тесты будут всегда выполняться однообразно, т. е. тестировщик не пропустит тест по неосмотрительности и не ошибется в результатах;

- Быстрое выполнение − автоматизированному скрипту не нужно сверяться с инструкциями и документациями, что сильно экономит время выполнение тестов;

- Меньше затрат на поддержку – на проведение того же объёма тестирования вручную уходит куда больше времени, чем на поддержку и анализ уже написанных результатов;

- Отчёты – автоматически рассылаемые и сохраняемые отчёты о результатах тестирования;

- Выполнение без вмешательств – тесты могут выполняться в нерабочее время (например, ночью, когда нагрузка на локальные сети максимальна снижена) или во время выполнения инженер-тестировщик может заниматься другими продуктивными делами.

Недостатками автоматизации тестирования являются:

- Повторяемость – в тоже время является и минусом автоматизации, поскольку, выполняя тест вручную, можно обратить внимание на дополнительные детали и, проведя, несколько вспомогательных операций, найти неисправность. Скрипт на это пока не способен;

- Большие затраты на разработку – разработка автоматизированных тестов — это сложный процесс, так как собственно идёт разработка системы, которая тестирует другое приложение. Для написания тестов также используются фреймворки, библиотеки, утилиты и прочее. Их, естественно, необходимо тестировать и отлаживать, на что требуется достаточное количество времени;

- Затраты на поддержку − затраты на ручное тестирование больше, чем на автоматизированное того же функционала, но они всё же есть. Чем чаще изменяется приложение, тем они выше.

Автоматизацию целесообразно применять в следующих ситуациях:

- Часто используемая функциональность, риски от ошибок в которой достаточно высоки. При автоматизации проверки критических функций ПО, можно гарантировать быстрое нахождение ошибок и впоследствии их решение;

- Рутинные операции (автоматизировать перебор данных, заполнение различных полей данными, их проверку и сохранение и тому подобное);

- Валидационные сообщения (автоматизировать заполнение полей корректными данными и проверку на появление той или иной валидации);

- Проверка данных, нуждающихся в точных математических расчётах;

- Проверка правильности поиска данных.

И многое другое, зависящее от требований к тестируемой системы и выбранного инструмента тестирования.

Для автоматизации тестирования используется трёхуровневая модель.

1. Уровень модульного тестирования (Unit Test layer).

Автоматизация модульных или компонентных тестов. Наличие подобных тестов на ранних стадиях разработки программного продукта и дальнейшее их пополнение новыми тестами убережёт проект от многих серьёзных проблем.

2. Уровень функционального тестирования (Functional Tests Layer (Non-UI)).

Позволяет, при необходимости, тестировать непосредственно бизнес логику приложения, не принимая во внимание пользовательский интерфейс.

3. Уровень тестирования через пользовательский интерфейс (GUI Test Layer).

Происходит тестирование функциональности системы, имитируя действия конечного пользователя, через графический интерфейс [1].

Для обеспечения лучшего качества продукта, рекомендуется автоматизировать все 3 уровня, но в данной работе мы коснёмся только третьего уровня.

### 1.1.2. О плагине

Для рассмотрения что такое плагин и его возможностей, нам необходимо расшифровать следующие термины:

- Скрипт (сценарий) — это последовательность действий, описанных с помощью скриптового языка программирования (JavaScript, PHP, Perl, Python и др.) для автоматического выполнения определенных задач [3];
- API (программный интерфейс приложения) — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой [3].

Плагин — это независимо компилируемый программный модуль, динамически подключаемый к основной программе и предназначенный для расширения и/или использования её возможностей. API плагин для средств разработки в браузере Google Chrome ─ DevTools ─ это программа, которая позволяет создавать, тестировать и отлаживать программное обеспечение. Она позволяет просмотреть исходный код сайта. С её помощью можно просматривать и отлаживать HTML разметку сайта, его CSS стили и скрипты, написанные на языке JavaScript. Также можно проверить сетевой трафик, потребляемый сайтом, его быстродействие и много других параметров.

Расширение DevTools добавляет функциональность в Chrome DevTools. Оно может добавлять новые пользовательские панели и боковые панели, взаимодействовать с проверяемой страницей, получать информацию о сетевых запросах и многое другое.

Расширение DevTools структурировано как любое другое расширение: оно может иметь фоновую страницу, скрипты содержимого и другие элементы. Кроме того, каждое расширение DevTools имеет страницу DevTools, которая имеет доступ к API DevTools.

Для создания расширения в первую очередь создаётся файл manifest.json он представляет собой файл формата JSON. Это единственный файл, который обязательно должен быть в каждом расширении, использующем API Веб-расширения (WebExtension APIs).

Используя manifest.json, мы определяем базовые метаданные о расширении, такие как имя и версия. Также можно определить некоторые

аспекты функционала (такие, как фоновые скрипты, контент скрипты и действия браузера).

Далее создаётся каркас плагина (HTML – код) и пишутся все необходимые скрипты для взаимодействия со страницей браузера. Один из таких, скрипт содержимого (content-script) - это «файл JavaScript, который выполняется в контексте веб-страниц». Это означает, что скрипт содержимого может взаимодействовать с веб-страницами, которые посещает браузер. Скрипт содержимого имеет доступ к текущей странице, но ограничен в API, к которым он имеет доступ. Например, он не может прослушивать клики на действие браузера. Необходимо добавить другой тип сценария к создаваемому расширению, фоновый сценарий (background), который имеет доступ к каждому API Chrome, но не может получить доступ к текущей странице. Поэтому скрипт содержимого сможет извлечь URL-адрес из текущей страницы, но ему нужно будет передать этот URL в фоновый сценарий, чтобы сделать с ним что-то полезное [4]. Для общения будет использоваться то, что Google называет передачей сообщений, что позволяет скриптам отправлять и прослушивать сообщения. Это единственный способ взаимодействия скриптов содержимого и фоновых скриптов (*Рисунок 1.1*).

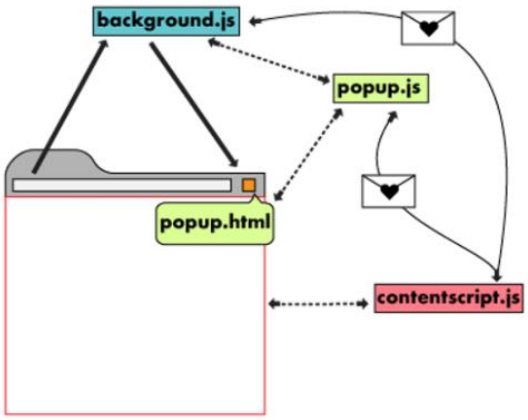


Рисунок 1.1 - Процесс взаимодействия между скриптами расширения

Чтобы опубликовать расширение в браузере надо во вкладке «Расширения» Google Chrome включить режим разработчика и «загрузить распакованное расширения», т.е. выбрать папку, где хранятся все необходимые скрипты для расширения, и где обязательно должен присутствовать файл manifest.json (*Рисунок 1.2*) [5].

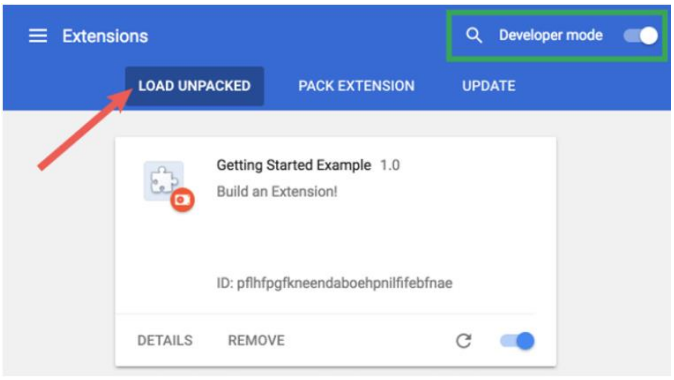


Рисунок 1.2 - Пример опубликования созданного расширение в Google Chrome

## 1.2. Постановка задачи

Разработать плагин для браузера, который регистрирует действия пользователя в веб-системах «НПО «Криста», для последующего преобразования в программный код, способный повторить эти действия в автоматизированном режиме.

## 1.3. Формулировка проблемы

В ввиду того, что компания «НПО «Криста» разрабатывает большие многофункциональные веб-приложения имеет смысл тестировать пользовательский интерфейс, так как главная задача разработчиков – выпустить полезный, функциональный и удобный продукт. Поскольку нажатие функциональных кнопок, открытие вкладок, заполнение полей ввода и тому подобные операции являются рутинными, то для тестирования UI (User Interface) веб-приложений в компании разработали специальную библиотеку autotest-lib, способную воспроизвести действия пользователя в автоматизированном режиме. С помощью данной библиотеки разработчики пишут функциональные автоматизированные тесты, проверяя бизнес-функции приложения через графический интерфейс.

Написания тестов-проверок для таких однообразных операций даже с уже разработанной библиотекой автоматизации autotest-lib всё ещё является монотонной задачей, которая затрачивает большое количество рабочего времени разработчика. Чтобы автоматизировать также и текущий процесс, было принято решение записывать макросы (микрокоманды) действий пользователя для последующего преобразования их в программный код библиотеки autotest-lib, что в последствии ускорит процесс тестирования веб-приложений, позволяя увеличить количество автоматизированных функциональных тестов, а, следовательно, и улучшить качество программного продукта.

## 1.4. Обоснование актуальности задачи

В современном мире скорость выпуска программных продуктов на рынок является неотъемлемой частью конкурентной борьбы. Разработка любой программы состоит из нескольких обязательных этапов, грамотное следование которым становится основополагающим критерием для занимающихся созданием ПО компаний. Методология разработки ПО включает следующие стадии: анализ требований, кодирование, тестирование, отладка и внедрение, необходимых для получения правильно выполняющей свои функции программы.

Тестирование продукта и последующая отладка позволяют ликвидировать ошибки программирования и добиться полнофункциональной работы разработанной программы. Чем раньше начинается тестирование, тем быстрее продукт попадет в реальную эксплуатацию. Современное программное обеспечение является сложным многофункциональным объектом. Его ручная проверка требует значительных трудовых, денежных и временных затрат. Поэтому на помощь и приходят средства автоматизации тестирования. Благодаря автоматизации повышается качество тестирования, так как «человеческий фактор» не оказывает влияния на качество тестирования. Также автоматизация ускоряет процесс тестирования без потери качества. Использование средств автоматизации для тестирования позволяет запускать уже написанные скрипты без дальнейших доработок. Так же остро стоит вопрос создания функциональных тестов. Задача разрабатываемой системы устранить из этого процесса программиста.

Таким образом, с помощью автоматизации тестирования можно резко увеличить количество создаваемых тестов для систем и как можно быстрее выпускать новые программные продукты и улучшать их.

## 1.5. Обзор аналогов

В ходе исследования предметной области проекта было выявлено два аналога разрабатываемой системы: Selenium IDE и Wildfire.ai.

Selenium IDE является интегрированной средой разработки тестов Selenium (*Рисунок 1.3*). Реализован как плагин к браузеру Firefox и Chrome. Он предназначен для записи взаимодействия пользователя с веб-сайтами для создания и поддержания автоматизации веб-приложений, тестов и устранения необходимости вручную выполнять повторяющиеся действия. В Selenium IDE можно выполнять такие функции как: запись и воспроизведение тестов в браузерах Firefox и Chrome, организация тестов в наборы для удобного

управления, сохранение и загрузка скриптов для последующего воспроизведения [6].

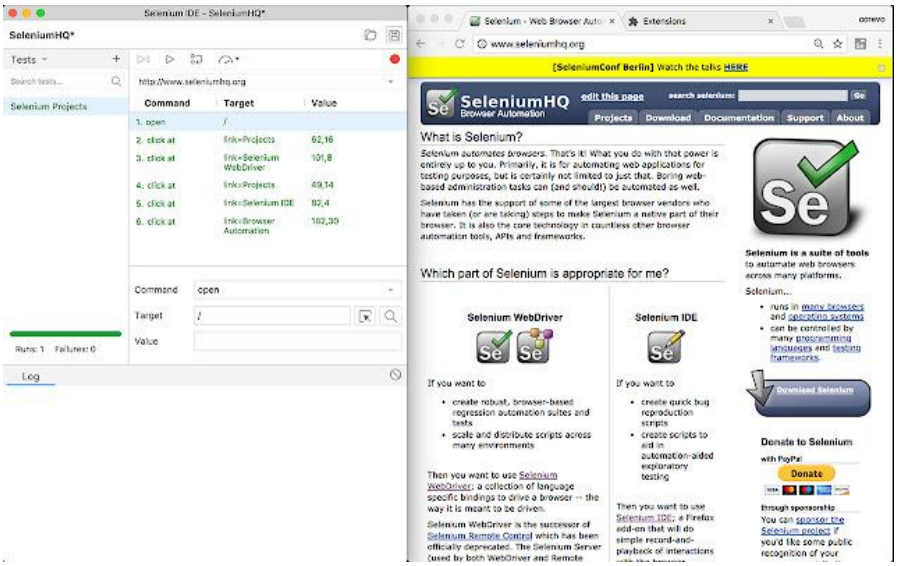


Рисунок 1.3 - Обзор аналога Selenium IDE

Wildfire.ai − это расширение Chrome и Firefox, которое позволяет записывать действия пользователей на посещённых им веб-страницах и воспроизводить их с помощью симулятора (*Рисунок 1.4*). При записи или моделирование действий, создаётся журнал, который можно просматривать. Затем можно использовать редактор рабочего процесса для управления поведением симуляции. Оно также может использоваться таксировщиками ПО, которые используют такой инструмент как Selenium [7].

Сравнение существующих решений для создания системы регистрации действий пользователя сведено в *таблице 1.1*.

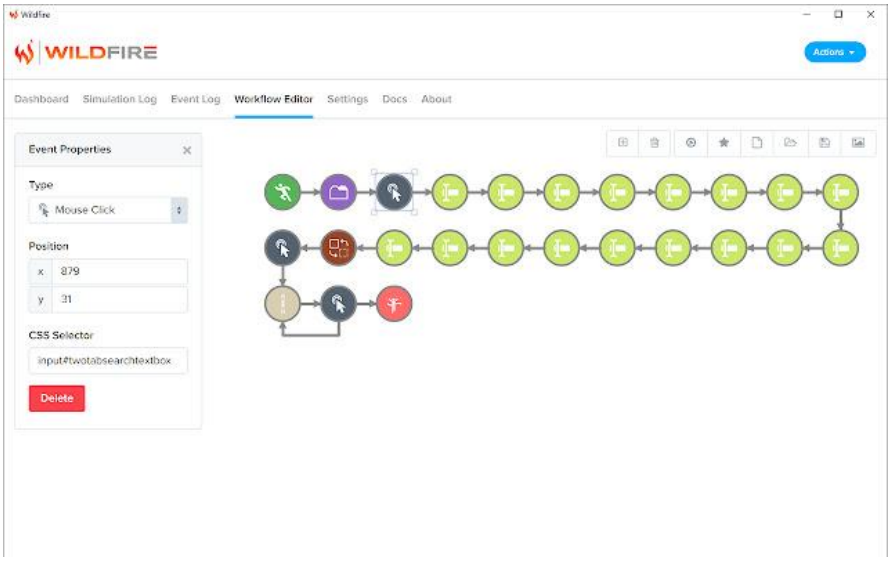


Рисунок 1.4 - Обзор аналога Wildfire

Таблица 1.1 - Сравнение аналогов

| Сравнительные критерии | Selenium IDE | Wildfire.ai |
|---|---|---|
| Поддерживает браузеры Google Chrome и Firefox | + | + |
| Записывает взаимодействия пользователя с веб-страницей | + | + |
| Запись и воспроизведение теста | + | + |
| Сохранение и загрузка скриптов для последующего воспроизведения | + | + |
| Возможность просматривания и моделирования журнала симуляции | — | + |
| Использование инструмента Selenium | + | + |
| Генерация кода теста | — | — |

При анализе целесообразности создания системы регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования был сделан вывод о том, что аналоги, представленные выше, не подходят для поставленной задачи. Во-первых, это обуславливается тем, что интерфейсы веб-систем в компании «НПО «Криста» состоят из собственно разработанных UI-компонентов и для тестирования данных компонентов используется также разработанная компанией библиотека «autotest-lib», что и требует создание собственного плагина, который будет регистрировать действия пользователя в веб-системах «НПО «Криста» и генерировать код автоматизированных тестов под корпоративную библиотеку autotest-lib. А во-вторых, ни один из перечисленных аналогов не имеет возможности трансформировать записанные действия пользователя в программный код, что позволяет запускать тесты внутри самих веб-приложений компании.

# 2. Программная документация

## 2.1. Техническое задание на программное обеспечение

### 2.1.1. Общие сведения

#### 2.1.1.1. Полное наименование системы и её условные обозначения

Полное наименование системы: «Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования».

Краткое название системы: «Система регистрации действий пользователя с генерации кода автотестов».

#### 2.1.1.2. Плановые сроки начала и окончания работ

Плановый срок начала работ по созданию плагина с генерацией автотестов 1 июля 2020 года. Плановый срок окончания работ по разработке 15 июня 2021 года.

#### 2.1.1.3. Перечень документов, на основании которых создаётся система, кем и когда утверждены эти документы

Основанием для разработки системы регистрации действий пользователя с генерацией кода автотестов являются следующие документы и нормативные акты:

- Приказ по темам ВКР № 550-04 от 30.11.2020;
- Функциональные требования.

#### 2.1.1.4. Порядок оформления и предъявления заказчику результатов работ

Система предоставляется в виде готового плагина для браузера Google Chrome, с соответствующей необходимой документацией в сроки, установленные календарным планом.

#### 2.1.1.5. Перечень нормативно-технических документов, методических материалов, использованных при разработке ТЗ

При разработке системы и создании проектно-эксплуатационной документации необходимо руководствоваться требованиями следующих нормативных документов:

- ГОСТ 19.201-78. Техническое задание. Требования к содержанию и оформлению;
- РД 50-34.698-90. Методические указания. Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов.

## 2.1.2. Назначение и цели создания системы

### 2.1.2.1. Назначение системы

Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования представляется как плагин для браузера, который регистрирует действия пользователя в веб-системах «НПО «Криста», для последующего преобразования в программный код, способный повторить эти действия в автоматизированном режиме.

Плагин предназначен, для ускорения разработки функциональных автоматизированных тестов, что должно позволить увеличить количество тестов и, соответственно, качество программных продуктов.

### 2.1.2.2. Цели создания системы

Основными целями создания программного продукта являются упрощение процесса тестирования ПО, автоматизация процесса тестирования и отслеживание пользовательских действий в веб-системах «НПО «Криста».

## 2.1.3. Требования к системе

### 2.1.3.1. Требования к структуре и функционированию системы

В состав системы регистрации действий пользователя с генерации кода автотестов должны входить следующие компоненты:

- Плагин ─ предназначен для скриптов расширения, т. е. то, что будет загружаться в браузер для формирования полноценного расширения и встраивания в инструменты разработчика DevTools. Обязательно должен иметь следующие файлы и скрипты:
    - manifest.json ─ некий конфигурационный файл, который предоставляет необходимую информацию о расширении. Например, такую как: название, версия, разрешения, используемые url и так далее;
    - content script ─ файлы, которые запускаются в контексте веб-страниц. Используя стандартную объектную модель документа

(DOM), они могут читать сведения о веб-страницах, которые посещает браузер, вносить в них изменения и передавать информацию в их родительское расширение;
    - background script ─ обработчик событий расширения, он содержит прослушиватели событий браузера, которые важны для расширения;
    - devtools.js и devtools.html ─ файлы, которые позволяют встроить плагин в DevTools.
- Подсистема Agent ─ предназначена для взаимодействия с окном веб-приложения, включает в себя такие функции как:
    - Прослушивает события веб-приложения;
    - Анализирует и обрабатывает «пойманные» события, формируя лог объекта данных;
    - Подписывается на модуль компании для сбора статистики о взаимодействии пользователя с элементами интерфейса ─ CoreAnalytics и обрабатывает события, приходящие от него;
    - Работает с NoSQL хранилищем данных IndexedDB: сохраняет информацию о событиях, получает данные, очищает БД от записей при создании новой сессии.
- Подсистема DevTools page (UI) ─ предназначена для визуализации информации о произошедших событиях на веб-странице и сгенерированного java-кода автоматизированного теста, преобразовывает полученные объекты данных о действиях пользователя из IndexedDB в java-код автоматизированных тестов.

Agent и DevTools page (UI) общаются между собой посредством отправки сообщений через content-script и background page: Agent ↔ content-script ↔ background ↔ DevTools. Проверка установления связи между подсистемами происходит таким образом:

- Agent ← content-script ← background ← [DevTools page - 'connect'];
- [Agent - 'connect'] → content-script → background → DevTools page.

Доступ к базе данных IndexedDB асинхронный, то есть клиент делает запрос и фиксирует функции обратного вызова. Как только станет известен результат запроса, он будет проинформирован об этом. В случае успеха операции запрос будет выполнен, в случае ошибки – отобразится причина ошибки.

В общем виде архитектура системы должна выглядеть следующим образом (*Рисунок 2.5*):

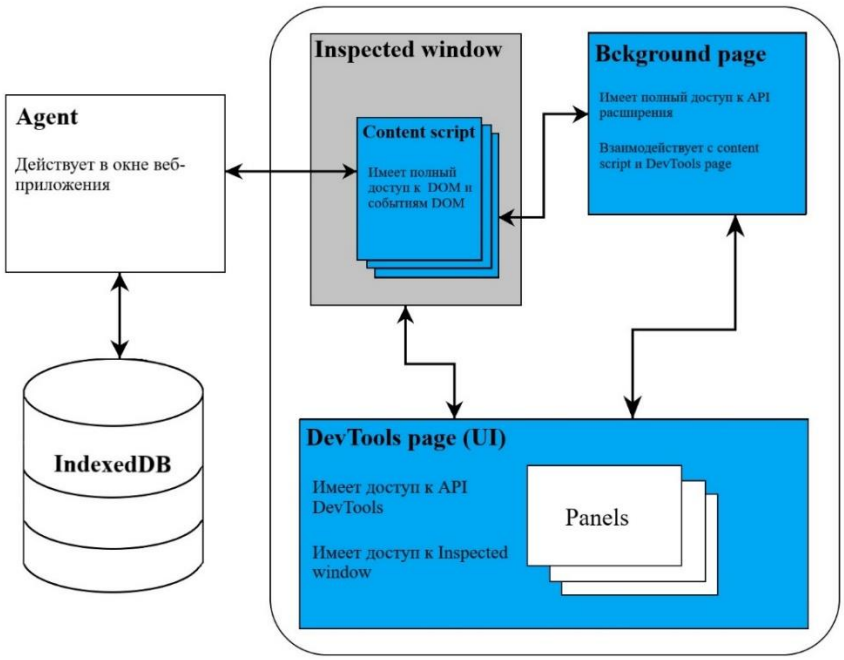


Рисунок 2.5 - Архитектура системы

### 2.1.3.2. Требования к реализации программы

Функциональные требования системы:

- Работа системы должна соответствовать определённым, рассмотренным и проанализированным ранее бизнес-процессам;
- Система должна предоставлять пользователю через графический интерфейс следующие функции:
    - Начать генерацию кода автоматизированного теста;
    - Закончить генерация кода автоматизированного теста;
    - Обновить лог событий;
    - Скопировать сгенерированный код;
    - Посмотреть сгенерированный код автотеста и лог событий выбранной сессии.
- Система должна регистрировать действия пользователя при работе в веб-приложении на странице браузера;
- Система должна анализировать и обрабатывать полученные события с веб-страницы приложения;
- Система должна определять с какими ui-компонентами и элементами веб-приложения взаимодействовал пользователь;
- Система должна формировать объекты данных (логи) о произошедших событиях на страницах. Лог должен содержать следующие поля:
    - Дата и время;

    - Тип события;
    - Идентификатор или название элементов, с которыми взаимодействовал пользователь;
    - Значение элемента формы (если имеется);
    - Путь до элемента в DOM-дереве веб-страницы.
- Система должны сохранять собранные данные о событиях в клиентское хранилище IndexedDB для последующей с ними работы;
- Система должна превращать собранные действия в код на языке программирования java, адаптированный под корпоративную библиотеку автоматизации тестирования autotest-lib;
- Система должна отображать лог и сгенерированный код автоматизированных тестов на созданной панели в DevTools;
- Система должна создавать сессии генераций кода. Каждая сессия должна отображать следующую информацию:
    - Дата и время окончании генерации кода автотеста;
    - Java-код автоматизированного теста;
    - Лог событий, т.е. данные по которым генерировался автотест.

### 2.1.3.3. Требования к надёжности

Система должна сохранять работоспособность и обеспечивать восстановление своих функций при возникновении следующих внештатных ситуаций:

- при сбоях в системе электроснабжения аппаратной части, приводящих к перезагрузке операционной системы (ОС), восстановление программы должно происходить после перезапуска ОС и запуска исполняемого файла системы;
- при ошибках в работе аппаратных средств (кроме носителей данных и программ) восстановление функции системы возлагается на ОС;
- при ошибках, связанных с программным обеспечением (ОС и драйверы устройств), восстановление работоспособности возлагается на ОС;
- при ошибках, связанных с браузером, восстановление программы должно происходить после перезапуска браузера;
- при ошибках с сетью, восстановление работоспособности возлагается на провайдера сети.

Оценка и контроль показателей надежности ПО выполняется только при ее эксплуатации. Для оценки и контроля используется метод мониторинга, т.е. непрерывного наблюдения и регистрации параметров работы программы.

### 2.1.3.4. Требование к безопасности

Система не должна запрашивать излишние разрешения для работы и замедлять работу браузера.

Также программа не должна наносить вред, как аппаратной, так и программной части устройства, на котором она функционирует, а также не должна замедлять работу других программ.

### 2.1.3.5. Требование к интерфейсу

Интерфейс должен быть простым и интуитивно понятным пользователю. Не должно возникать перегруженности на экране, для комфортной работы с данными.

### 2.1.3.6. Требования к информационной и программной совместимости

1. Требования к клиентской части системы:
   - Входные данные:
     - события от веб-приложения, с которым взаимодействует пользователь;
     - события от корпоративного модуля сбора статистики CoreAnalytics.
   - Выходные данные:
     - информация о событиях (лог), произошедших на веб-странице;
     - сгенерированный java-код автоматизированных тестов, адаптированный под корпоративную библиотеку autotest-lib;
     - сохранённый лог и сгенерированный код текущей сессии.
- Система управления версиями – Git.
- Язык программирования – TypeScript (компилятор – Babel 7) и JavaScript.
- Среда установки – DevTools в браузере Google Chrome версии 43 и выше.

2. Требования к базе данных:

- Использование объектного хранилища данными внутри браузера Indexed Database API;
- Хранимая о каждом взаимодействии с веб-приложением информация: уникальный идентификатор, дата и время, тип события, идентификатор или название ui-компонента (с которым работал пользователь), значение элемента формы (если пользователь вводил какие-то данные), xpath элемента (путь в DOM-дереве веб-страницы до элемента).

#### 2.1.3.7. Требования к организационному обеспечению

К работе с системой должны допускаться сотрудники, имеющие навыки работы на персональном компьютере, ознакомленные с правилами эксплуатации и прошедшие обучение по работе с системой.

#### 2.1.3.8. Требования к документации

Программная документация:

- Техническое задание на программное обеспечение;
- Пояснительная записка к программному обеспечению;
- Описание программы;
- Программа и методика испытаний.

Эксплуатационная документация на программный продукт – Руководство пользователя.

## 2.2. Пояснительная записка к программному обеспечению

### 2.2.1. Назначение и область применения

Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования предназначена для ускорения разработки функциональных автоматизированных тестов, что должно позволить увеличить количество тестов и, соответственно, качество программных продуктов.

Вид автоматизируемой деятельности – тестирование.

Объект автоматизации – процесс написания тестов.

Цель создания – упростить процесс автоматизации тестирования, сократить время на написания автоматизированных тестов и увеличить количество тестов.

Область применения программы – автоматизация тестирования интерфейса программного продукта.

## 2.2.2. Технические характеристики

Разрабатываемое ПО должно фиксировать действия пользователей в веб-системе и превращать эти действия в java-код автоматизированных тестов. На *рисунке 2.6* представлены входные и выходные данные, характеристики которых приведены в *таблице 2.2*.

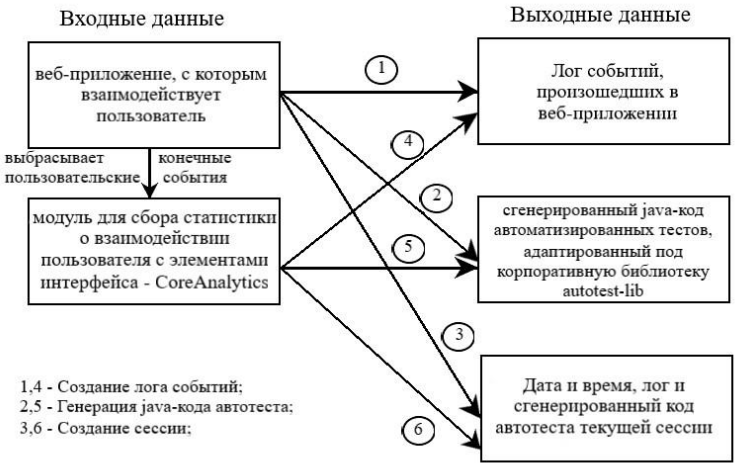


Рисунок 2.6 - Входные и выходные данные системы

Таблица 2.2 - Описание входных и выходных данных системы

| Характеристики | Входные данные | Выходные данные |
|---|---|---|
| Представляемая информация | - События от веб-приложения;<br>- События от модуля CoreAnalytics | - Характеристика события: уникальный идентификатор, дата и время (когда произошло событие), тип события, ui-компонент, значение элемента формы, xpath элемента;<br>- Java-код автоматизированного теста;<br>- Характеристика сессии: дата и время, лог, код теста |
| Формат представления | - DOM-модель;<br>- Observer pattern | - Запись в таблице IndexedDB;<br>- Запись в хранилище состояний приложения (Redux) |

Окончание таблицы 2.2

| Характеристики | Входные данные | Выходные данные |
|---|---|---|
| Носитель | - Сервер приложений | - IndexedDB API;<br>- Хранилище состояний приложения (Redux) |
| Способ передачи | DevTools API | DevTools API |
| ПО для получения | Веб-браузер Google Chrome | - NoSQL хранилище IndexedDB;<br>- Хранилище состояний приложения (Redux) |

Для получения входных данных необходимо, чтобы разрабатываемая программа работала в браузере. Система должна получать доступ к открытому в браузере веб-приложению для регистрации действий пользователя. Поэтому наша система предполагает расширение функциональных возможностей браузера, т. е. создание плагина или, иначе говоря, расширение для браузера. Расширения состоят из разных, но связанных компонентов. Компоненты могут включать фоновые сценарии, сценарии содержимого, страницу параметров, элементы пользовательского интерфейса и различные файлы логики. Все элементы плагина взаимодействуют между собой посредством отправки сообщений друг другу. Компоненты расширения создаются с помощью технологий веб-разработки: HTML, CSS и JavaScript [5].

Для JavaScript было принято решение использовать надстройку TypeScript для создания более жесткой архитектуры программного кода с контролем изменений, приводящих к ошибкам несоответствия типов. Программа, написанная на TypeScript, переводится при помощи Babel 7 в программный код, который может выполняться в браузере [8].

Архитектура плагина позволит просматривать DOM-модель веб-приложения и отлавливать изменения его состояний.

Для генерации кода автотестов для интерфейса веб-приложения нам необходимо хранить информацию о произошедших событиях за текущую сессию. Поэтому для удобства было выбрано объектное хранилище IndexedDB, где можно хранить данные в формате JSON прямо внутри браузера.

Таким образом, разработка включает в себя создание плагина и базы данных IndexedDB, что представлено на *рисунке 2.7*.

Для удобства представления пользователю функций системы и их поддержки были выбраны такие библиотеки, как React и Redux. Они задают

общую архитектуру интерфейса приложения. React позволяет представить весь пользовательский интерфейс как набор компонентов, зависящих от внутреннего состояния приложения. Redux позволяет организовать хранение данных внутри приложения и отправку сообщений об изменениях, влияющих на представление react-компонентов. Благодаря данному подходу можно декларативно описать пользовательский интерфейс и смоделировать его возможные состояния [9]. Такая архитектура упрощает читабельность и масштабируемость разрабатываемого программного кода.

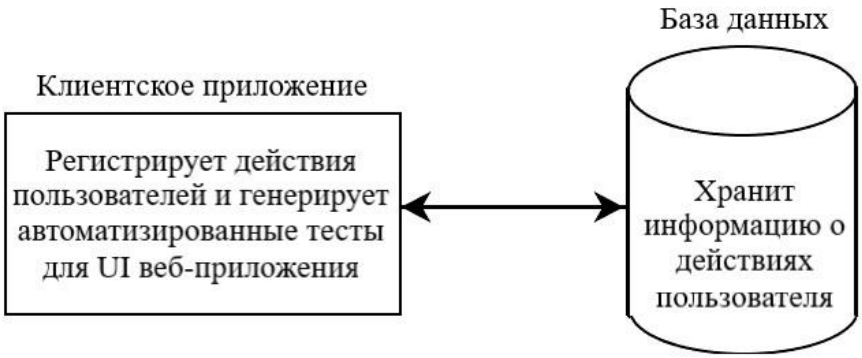


Рисунок 2.7 - Структурная схема разрабатываемой системы

Для упрощения разработки компонентов пользовательского интерфейса используются различные библиотеки, основанные на React, такие как, React Bootstrap, которая используется для стилизации компонентов.

Так как создаваемое расширение должно фиксировать действия пользователей на веб-странице и впоследствии генерировать автотест по этим данным, было принято решение добавить его в инструменты разработчика браузера DevTools в качестве новой панели.

Основные инструменты, используемые при реализации ПО, представлены в *таблице 2.3*.

Таблица 2.3 - Инструменты, выбранные для программирования системы

| Инструменты | Приложение |
|---|---|
| Система управления версиями | Git |
| Среда разработки | WebStorm |
| Языки разработки | TypeScript, ES6, JSX, HTML, CSS |
| Транслятор | Babel 7.5 |
| Инструмент сборки модулей | Webpack |
| Инструмент управления зависимостями | Npm |
| Основные библиотеки | React, Redux, React Bootstrap |

## 2.3. Описание программы

### 2.3.1. Общие сведения

Программа функционирует в браузере Google Chrome версии 43 и выше. В качестве среды разработки плагина используется WebStorm. Скачивания пакетов всех необходимых библиотек происходит из облачного сервера npm. В качестве языка программирования был выбран TypeScript, так как он обеспечивает статическую проверку типов. При помощи babel весь программный код компилируется в чистый JS, который впоследствии выполняется интерпретатором браузера. Также применяются языки разработки, такие как: ES6, JSX, HTML, CSS.

Пользовательский интерфейс плагина строится на основе отдельных компонентов с использованием библиотеки React. Так как в React не рекомендуется реализовывать прямое взаимодействие компонент-компонент, для управления состоянием компонентов используется Redux. Он позволяет хранить все состояние приложения в одном месте, называемом «store» («хранилище»). Компоненты «отправляют» изменение состояния в хранилище, а не напрямую другим компонентам. Компоненты, которые должны быть в курсе этих изменений, «подписываются» на хранилище.

Все модули плагина упаковываются при помощи webpack и загружаются в браузер.

### 2.3.2. Функциональное назначение

Программа является инструментом для упрощения процесса автоматизации тестирования интерфейса веб-приложений. С помощью данной системы тестирощики компании «НПО «Кристы» могут генерировать java-код автоматизированных тестов для UI, производя действия с интерфейсом веб-приложения.

### 2.3.3. Описание логической структуры

Плагин содержит в себе две части: подсистему Agent и подсистему UI. Первая часть программы встраивается в страницу веб-приложения, которая реализует функционал прослушивания событий, происходящих на этой странице, фиксирует их и сохраняет информацию о действии пользователя в хранилище IndexedDB (БД встраивается также в веб-страницу). Вторая часть функционирует на странице DevTools. Эта часть программы выполняет трансформация записанных действий пользователя в код автоматизированных тестов, использующих функционал корпоративной библиотеки autotest-lib, и

отображает лог событий и код автотестов на отдельной панели инструмента разработчика. Контент-скрипт и фоновая страница представляют собой модули, которые служат посредниками при передаче сообщений между подсистемами. Общая структура системы продемонстрирована на *рисунке 2.8*, сплошной линией показаны части программы.


Рисунок 2.8 - Взаимосвязь подсистем программы

Обе подсистемы загружаются в браузер в виде файлов под название Agent и UI. Они поучаются путём сборки соответствующих модулей. Общий замысел процесса работы приложения представлен в виде диаграммы деятельности на *рисунке 2.9*.

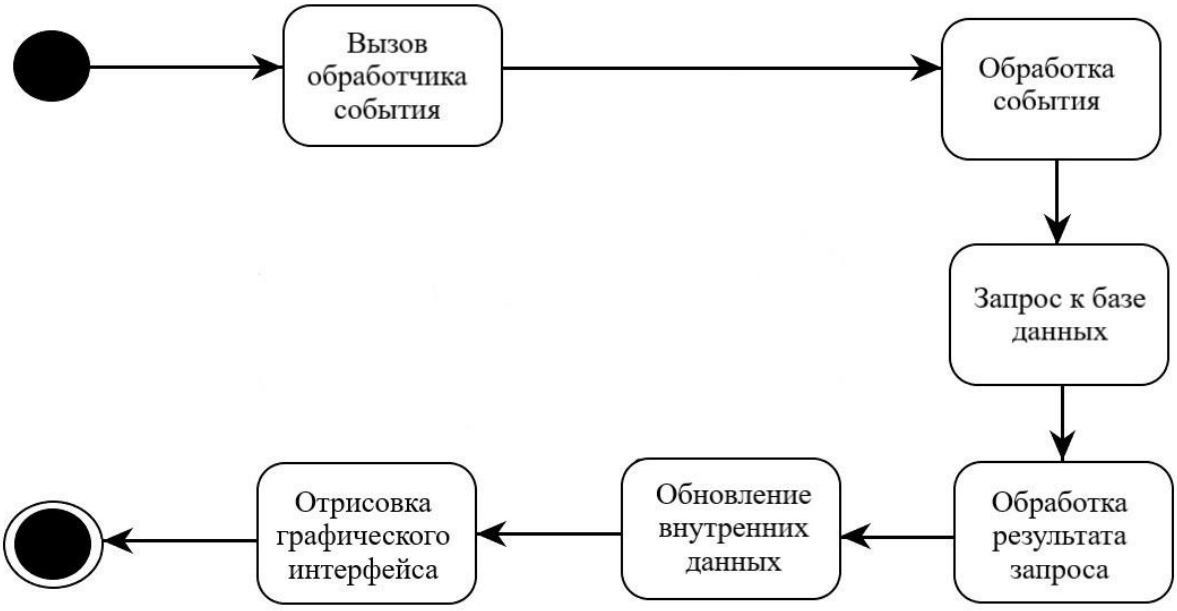


Рисунок 2.9 - Алгоритм работы приложения

Подсистема Agent встраивается в страницу веб-приложения и в начале работы плагина отправляет сообщение для подсистемы UI, уведомляя о том, что связь между ними установлена. Далее перехватывает события, которые совершил пользователь. Так же подсистема Agent подписывается на

корпоративный модуль CoreAnalytics, который выбрасывает сообщением данные о взаимодействии пользователя с элементами интерфейса. Эта подписка необходима для получения конечных пользовательских действий, что помогает получать актуальные данные о нужных событиях, когда веб-страница не успевает окончательно прогрузиться.

После того как событие было «поймано» класс Agent анализирует и обрабатывает либо полученный элемент DOM-дерева, либо полученное сообщение от CoreAnalytics. В процессе анализа определяет по каким UI-компонентам веб-приложения было произведено действие. В основном, распознавание происходит по определённым атрибутам, прописанным в DOM-модели приложения, таким как "class" или "data-control-type" (*Приложение А*). Подсистема может определять следующие UI-компоненты веб-приложения:

- Навигатор;
- Форма;
- Контейнер вкладок формы;
- Панель кнопок формы;
- Грид (некая область для работы с вычислительными и другими данными);
- Панель инструментов грида;
- Ячейки грида;
- Быстрый фильтр грида;
- Фильтр формы;
- Модальное окно.

У UI-компонента берутся все необходимые данные, которые пригодятся для создания строки автотеста, например, атрибуты "id", "title", "value" элемента. В итоге, после обработки действия, формируется объект данных для сохранения его в хранилище IndexedDB, чтобы потом превратить его в строчки java-кода автоматизированного теста, который будет проверять UI-компонент, с которым работал пользователь. Объект содержит следующие сведения о произошедшем событии:

- Дата и время, когда пользователь совершил то или иное действие;
- Тип события (например, клик по кнопке формы, открытие вкладки, ввод значений в быстрый фильтр грида и т.д.);
- Необходимые атрибуты UI-компонента, с которым работал пользователь;

- Значение (при вводе текста в ячейки грида, в поля фильтра формы и быстрого фильтра);
- Путь до элемента, с которым работал пользователь, в DOM-дереве веб-страницы (xpath).

На *рисунке 2.10* продемонстрировано как храниться созданный объект с данными о событии в IndexedDB.


Рисунок 2.10 - Сохранённый объект данных в IndexedDB

Подсистема Agent работает с базой данных IndexedDB, которая встроена в веб-страницу приложения (*Рисунок 2.11*), и может отправлять ей следующие запросы:

- Подключиться к БД (getConnection);
- Сохранить данные (saveItem);
- Получить данные (getDataFromDB);
- Очистить БД от всех записей (deleteItems).

В случае успеха операции запрос будет выполнен, в случае ошибки – отобразится причина ошибки.

Алгоритм процесса сбора, анализа и обработки действий пользователя изображён на *рисунке 2.12*.

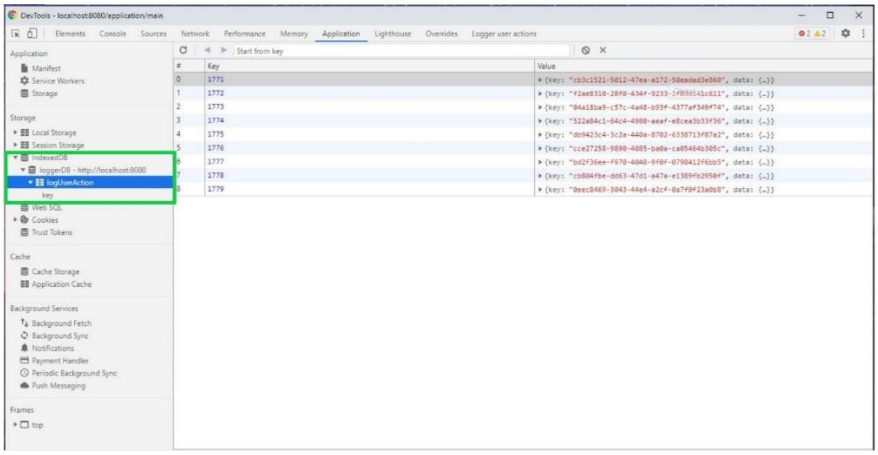

Рисунок 2.11 - Хранилище IndexedDB

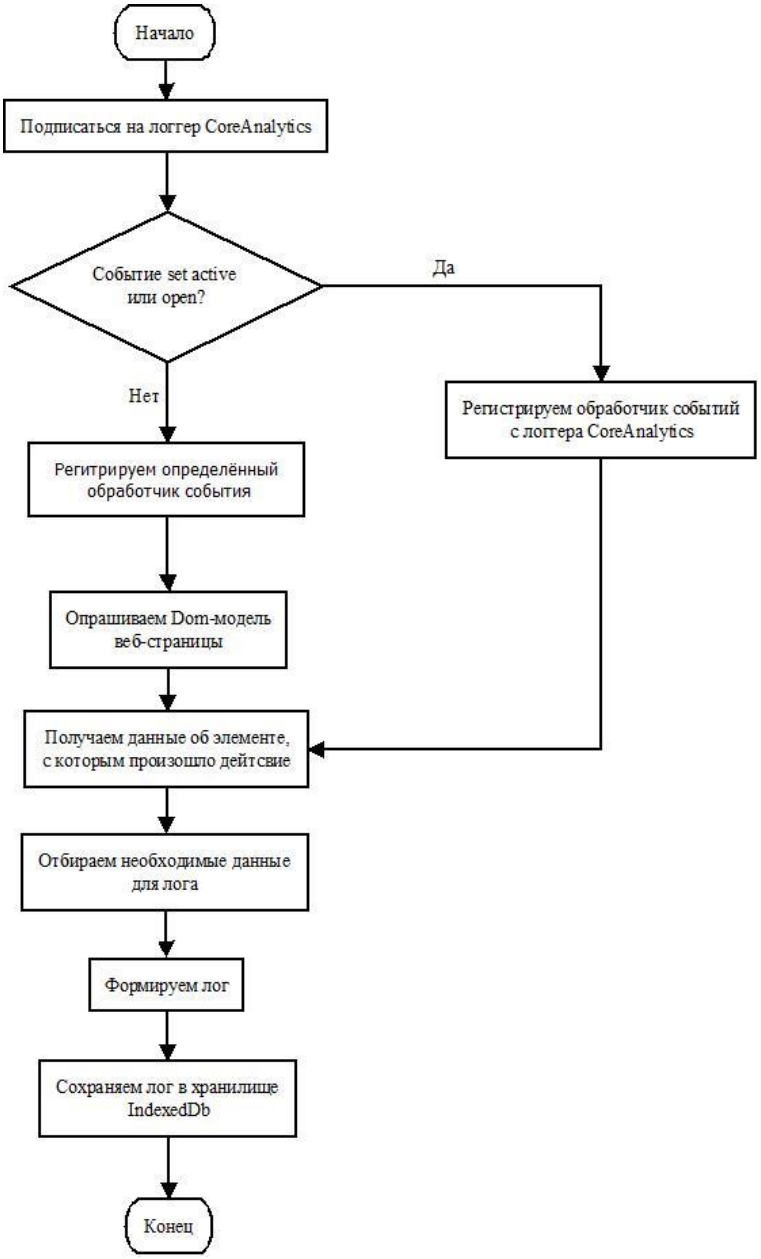


Рисунок 2.12 - Схема процесса сбора, анализа и обработки пользовательских данных

Интерфейс плагина (*Рисунок 2.13*) реализован в подсистеме UI, то есть в инструменте разработчика DevTools будет создана новая панель. Данная страница DevTools разделена на три части:

1. Окно «Лог» отображает информацию о произошедших событиях в веб-приложении;
2. Окно «Код» отображает сгенерированный код автоматизированного теста;
3. Окно «Сессии» отображает текущую и предыдущие сессии работы с плагином.

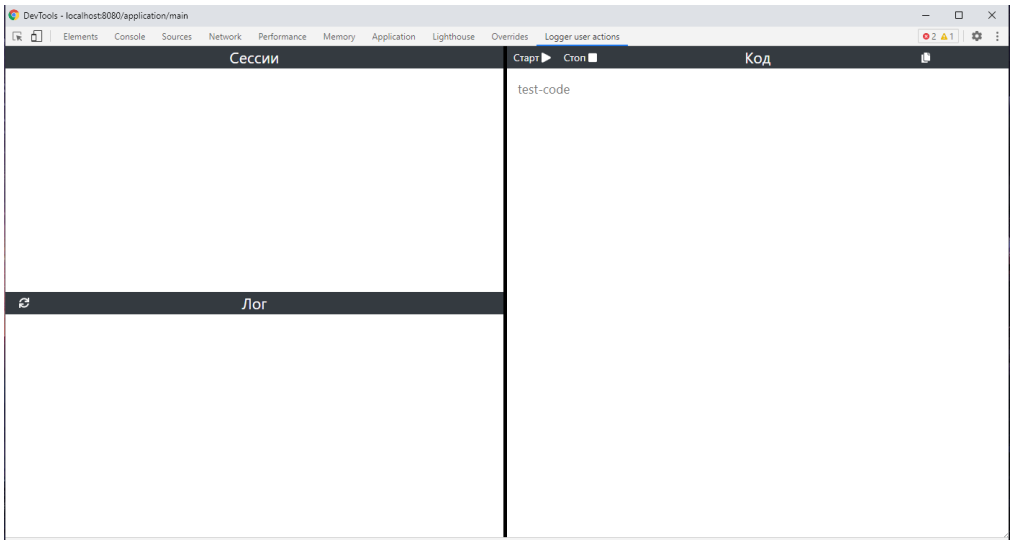


Рисунок 2.13 - Интерфейс плагина

Пользователь имеет пять функциональных возможностей:

1. Начать генерацию кода автоматизированного теста;
2. Остановить генерацию автоматизированного теста;
3. Обновить таблицу логов событий;
4. Скопировать сгенерированный код автоматизированного теста;
5. Посмотреть сгенерированный код теста и лог событий выбранной сессии.

Они представлены на *рисунке 2.14*.

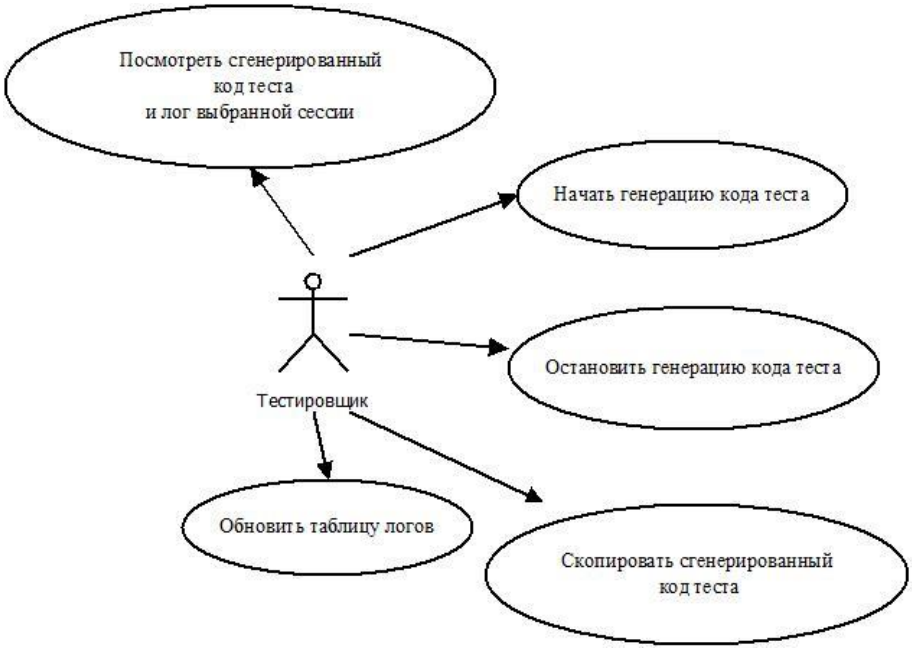


Рисунок 2.14 - Диаграмма вариантов использования системы

Процесс работы каждой функциональной возможность можно представить в виде диаграммы последовательности (*Рисунок 2.15*, *Рисунок 2.16*, *Рисунок 2.17*, *Рисунок 2.18*).

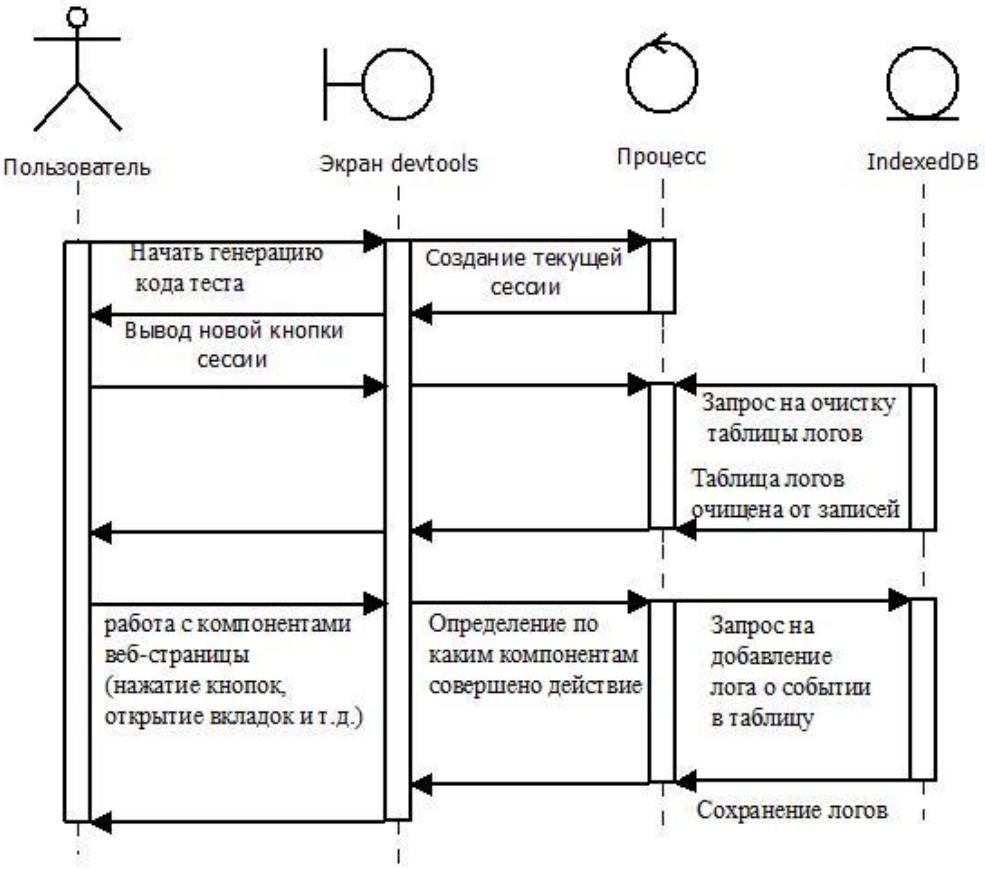


Рисунок 2.15 - Диаграмма последовательности старта генерации автотеста

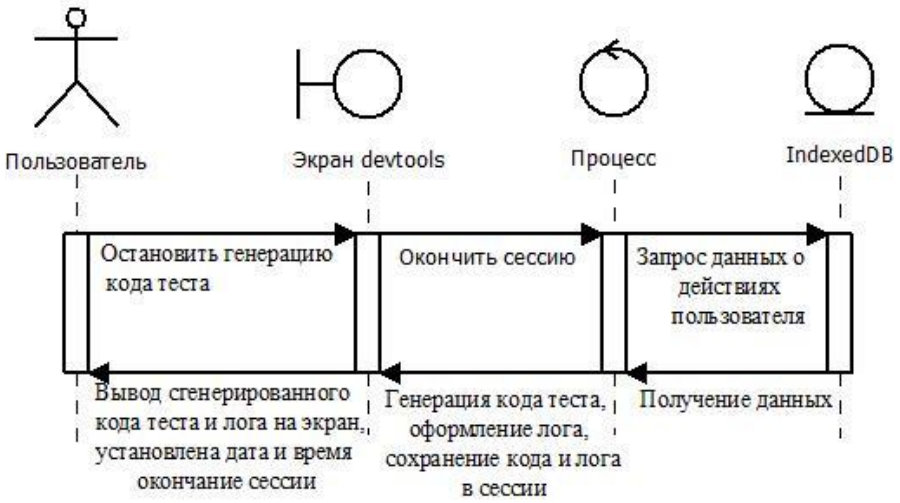


Рисунок 2.16 - Диаграмма последовательности окончания генерации автотеста

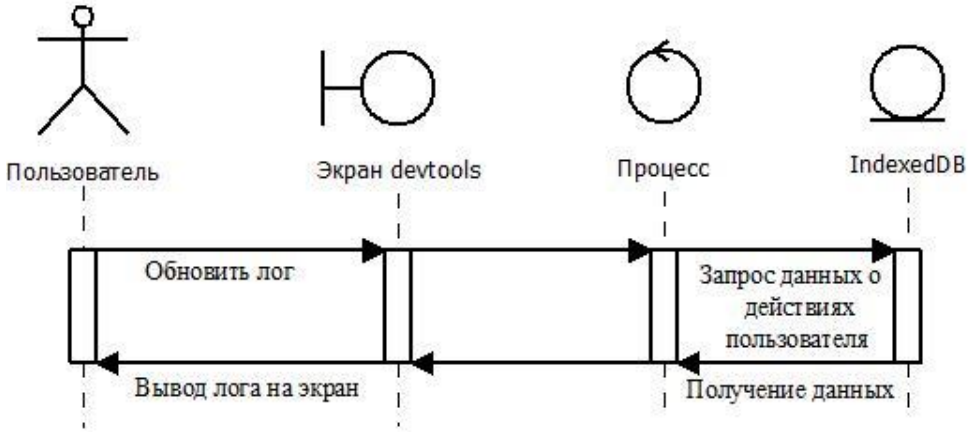


Рисунок 2.17 - Диаграмма последовательности обновления лога

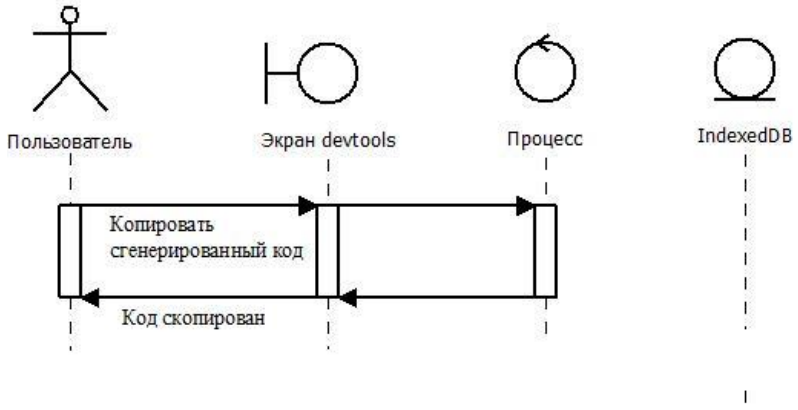


Рисунок 2.18 - Диаграмма последовательности копирования кода автотеста

Когда пользователь нажимает на кнопку «Стоп» происходит генерация java-кода автоматизированного теста. Из базы данных IndexedDB мы получаем все записи и начинаем их трансформировать в код, адаптированный под корпоративную библиотеку autotest-lib.

Для удобства превращения был создан модуль entities, где представлены, так называемые, классы сущности, которые отражают какие

действия произошли с определёнными UI-компонентами. Каждый класс сущности имеет два основных метода: isEntity() – проверяет, что по полученной записи из БД нам подходит именно эта сущность и getCode() – возвращает нам строку кода для этой сущности. А также два дополнительных метода: getPrevCode() и getPostCode(), которые возвращают нам строки кода до или после настоящей сущности необходимые для последующей правильной работы автотеста (*Приложение Б*). Алгоритм процесса генерации продемонстрирован на *рисунке 2.19*.

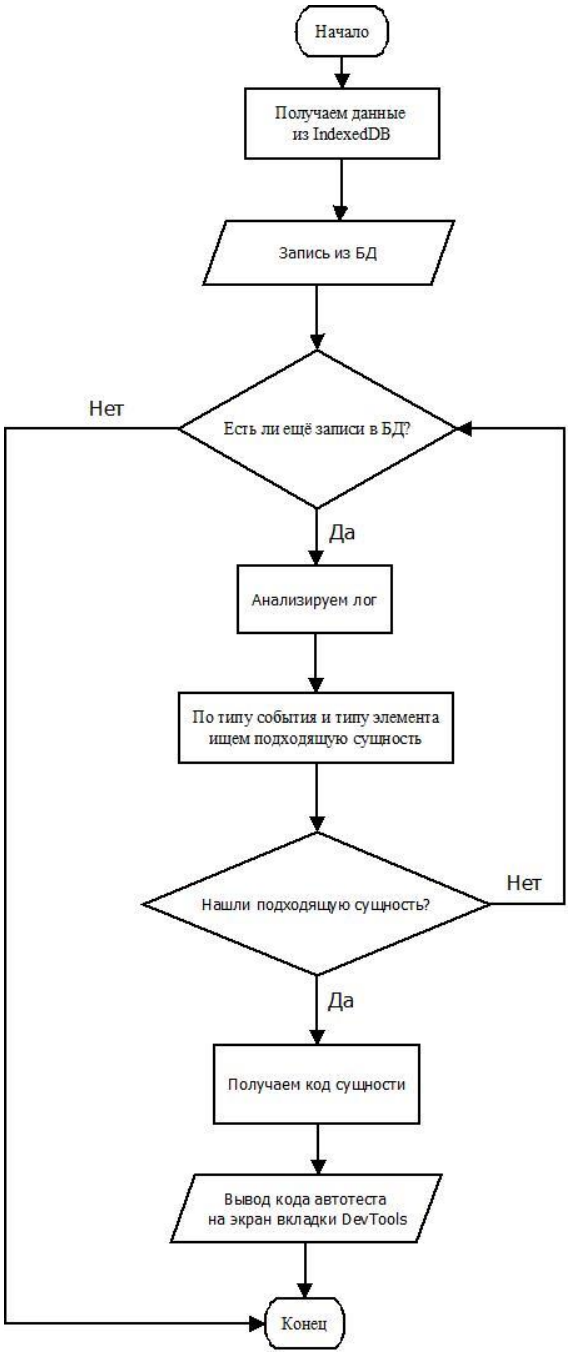


Рисунок 2.19 - Схема процесса генерации кода автотеста

Генерация можно описать следующим образом. При анализе полученного объекта лога, а конкретнее по его типу события и UI-компонентам, из списка имеющихся сущностей по условию метода isEntity() находится подходящая. В список классов сущностей входят:

- Открытие формы через навигатор (NavigatorOpenForm);
- Открытие формы по специальному идентификатору (OpenFormByDFD);
- Нажатие кнопки формы (ClickBigButton);
- Нажатие кнопки формы с выпадающим списком функций (ClickToolbarBigButton);
- Сделать активной вкладку формы (SetActiveTab);
- Закрыть вкладку формы (CloseTab);
- Обновить вкладку формы (RefreshTab);
- Открыть на полный экран вкладку формы (FullScreenTab);
- Установить размер вкладки по умолчанию (MinimizeTab);
- Нажатие кнопки на панели инструмента грида (ClickToolboxGridButton);
- Установить значение в быстрый фильтр (SetValueInQuickFilter);
- Установить значение в ячейку грида (SetValueInGridCell);
- Очистить ячейку грида (ClearGridCell);
- Сделать активным вкладку фильтра формы (ActivateFilter);
- Установить значение в параметр фильтра формы (SetFilterParameter);
- Очистить параметр фильтра формы (ClearFilterParameter);
- Применить фильтр формы (ApplyFilter);
- Сбросить фильтр формы (ResetFilter);
- Применить модальное окно (ApplyModalWindow);
- Закрыть модальное окно (CloseModalWindow).

После чего мы получаем шаблон кода автоматизированного теста. Отфильтрованный код, где убираются некоторое дублирование кода и пустые строки, выводится на интерфейс плагина (*Рисунок 2.20*).

Структура подсистемы UI может быть представлена на основании концепции «Model-View-Controller» (Модель-Представление-Контроллер). Это схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо (*Рисунок 2.21*) [3].

```
Старт ▶   Стоп ■                    Код

mainPage.getNavigator().openForm("Пакетная замена","Конфигурация пакетной замены").waitForLoadingWindow();
Form form = mainPage.getWorkArea().findForm("Конфигурация пакетной замены");
Grid grid = GridUtils.getGridByTabName(form,"Конфигурация пакетной замены");
GridToolbox toolbox = grid.toolbox();
toolbox.refresh();
Button btn = toolbar.getButton("Переместить вниз");
btn.shouldBe(new Condition[]{Condition.visible}).click();
Button btn = toolbar.getButton("Переместить вниз");
btn.shouldBe(new Condition[]{Condition.visible}).click();
Button btn = toolbar.getButton("Переместить вниз");
btn.shouldBe(new Condition[]{Condition.visible}).click();
Grid grid = GridUtils.getGridByTabName(form,"Конфигурация пакетной замены");
GridToolbox toolbox = grid.toolbox();
toolbox.editRow(3);
GridUtils.setColumnValueInSelectedRow("124", grid, "Минимальная сила препятствующих документов");
toolbox.rollback();
```

Рисунок 2.20 - Пример сгенерированного кода автоматизированного теста

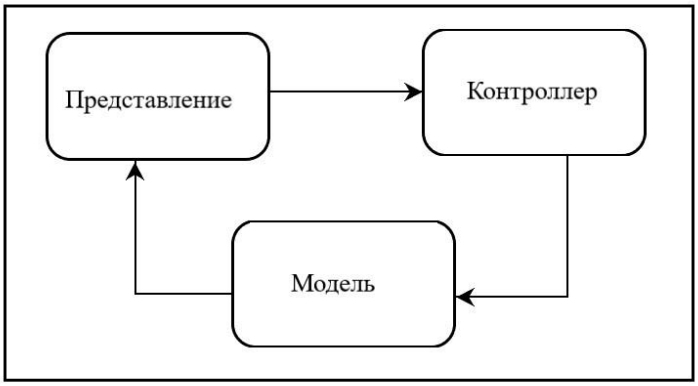


Рисунок 2.21 - Схема MVC

Пользователь взаимодействует с графическим интерфейсом (иначе говоря, представлением), осуществляет действия, события перехватываются контроллером и им обрабатываются. В соответствии с логикой обработки меняется состояние модели приложения. При изменении состояния компоненты представления перерисовываются.

В программе представление подсистемы UI реализовано по технологии React и предназначено для отображения графического интерфейса пользователю (*Рисунок 2.22*). В react-компонентах (расширение JSX) описана логика реагирования на действия пользователя для передачи надлежащей информации в контроллер, а также логика отображения с учётом состояния модели плагина.

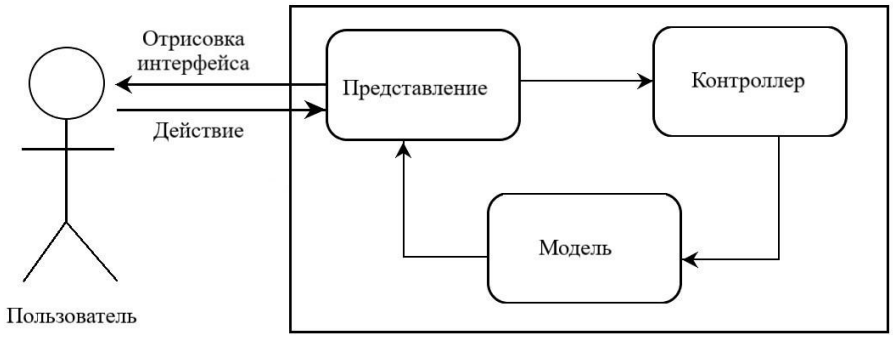


Рисунок 2.22 - Взаимосвязь пользователя с представлением

Модуль components является представлением подсистемы UI (Рисунок 2.23) и содержит в себе следующие react-компоненты:

1. Окно «Лог»:
   - Таблица логов (LogTable). Отображает информацию о произошедших событиях в веб-приложении. Информация содержит дату и время события, какое действие было совершено и по каким ui-компонентам. Данные получает из базы данных IndexedDB;
   - Кнопка обновления лога (RefreshButton). Обновляет таблицу логов;
2. Окно «Код»:
   - Текстовое поле (только для чтения) для сгенерированного java-кода автоматизированного теста (GeneratedCode). Отображает шаблон автотеста, для генерации кода берёт данные из IndexedDB;
   - Кнопка «начать генерацию кода» (StartGenerationButton). Очищает базу данных, фиксирует дату начала генерации и передаёт подсистеме Agent сообщение о том, что можно прослушивать действия пользователя;
   - Кнопка «закончить генерацию кода» (StopGenerationButton). Фиксирует дату окончания генерации и передаёт сообщение подсистеме Agent о том, что прослушивание действий пользователя можно завершить;
   - Кнопка копирования сгенерированного кода автотеста (CopyButton). Копирует из текстового поля весь сгенерированный шаблон кода автотеста;
3. Окно «Сессии»:
   - Блок сессий (Session). Отображает дату и время окончания каждой сессии генерации;

- Кнопка сессии (SessionButton). Отображает каждую сессию по-отдельности. При нажатии отображает код и лог данной сессии в соответствующих окнах;

4. Общее:
   - Заголовки (TitleRow). Отображает название окон для интуитивного понимания пользователя.

Пример react-компонента продемонстрирован в *приложение В*.

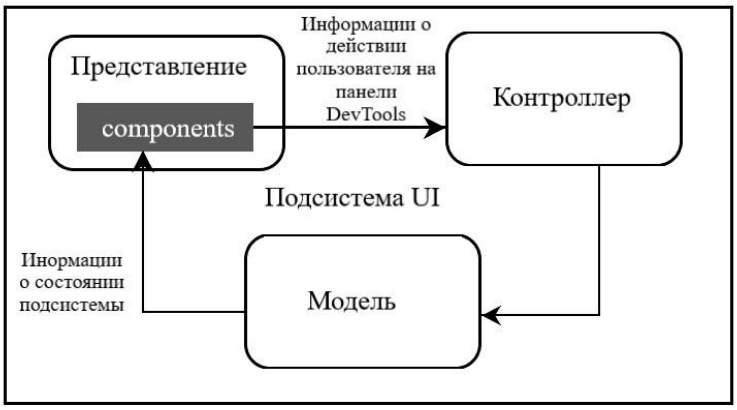


Рисунок 2.23 - Представление подсистемы UI

При взаимодействии пользователя с интерфейсом плагина из представления поступает информация о совершённом действии в модуль actions контроллера. В результате работы данного модуля происходит отправка информации для обновления модели, что представлено на *рисунке 2.24*.

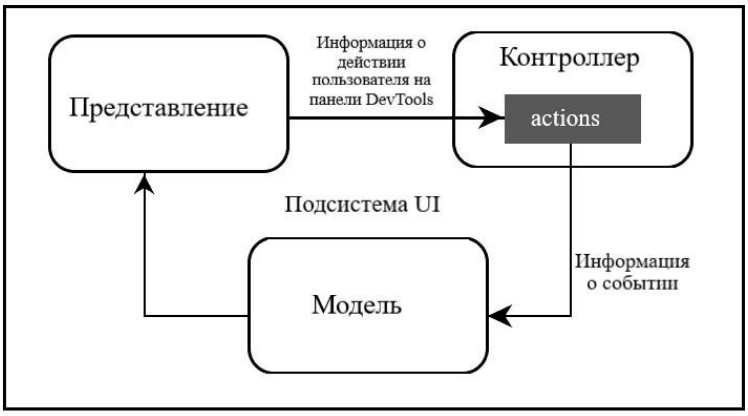


Рисунок 2.24 - Контроллер подсистемы UI

В контроллер подсистемы UI могут поступать информационные сообщения от Agent. Модуль AgentHandler отвечает за их обработку (*Рисунок 2.25*). Определена обработка следующих сообщений:

- уведомление об установлении связи;

- уведомление об окончание генерации кода автоматизированного теста;
- установление сессионных данных.

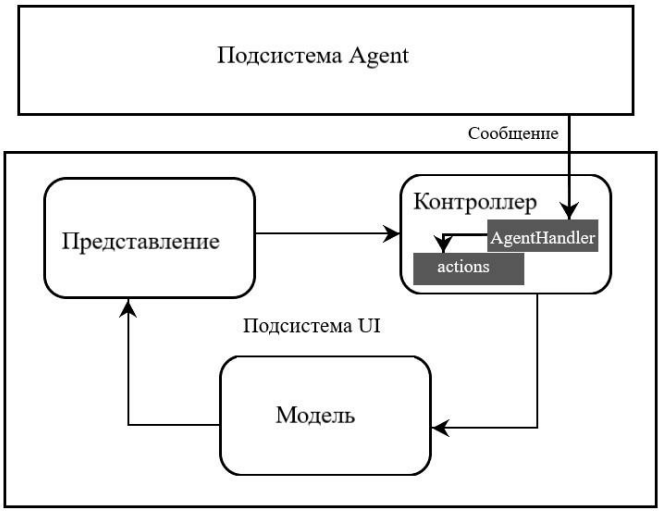


Рисунок 2.25 - Схема получение данных от подсистемы Agent

Получаемые из сообщения данные, в том числе информация, получаемая из представления, поступают на модуль actions. В нём определены следующие события:

- Основные события (mainAction):
  - успешно установлена связь с Agent;
- Информация о сессиях (sessionAction):
  - началась генерация кода автотеста;
  - сессия создана;
  - лог и код автоматизированного теста сессии сохранены;
  - установление сведений о сессии по её ключу;
  - установление сведений о текущей сессии;
  - закончилась генерация кода автотеста.

Модель подсистемы UI включает в себя хранимые данные и правила работы с ними. Структура состояния плагина построена по технологии Redux, и предназначена для хранения общих внутренних данных о подсистеме в модуле Store, в виде наборов состояний, определённых в модуле Reducers:

- Основное состояние (MainState):
  - наличие связи с Agent;
- Состояние сессии (SessionState):
  - номер текущей сессии;
  - характеристика сессии (хранит лог и сгенерированный код автотеста);

- дата начала сессии;
- дата окончания сессии.

Для каждого набора состояний в модуле Reducers определены реакции на события, получаемые от контроллера. После обработки полученного события происходит обновление данных в модуле Store. Информация об этом поступает в компоненты представления, состояние которых соответственно меняется. На *рисунке 2.26* представлена общая схема данного процесса.

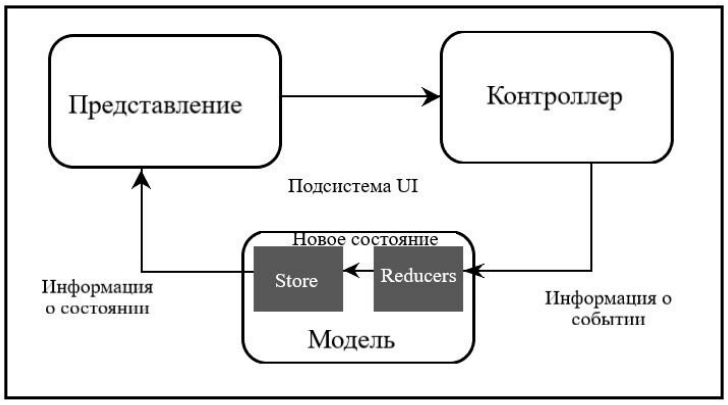


Рисунок 2.26 - Модель подсистемы UI

## 2.3.4. Используемые технические средства

Программа предназначена для установки на персональных компьютерах (ПК) в офисах предприятия «НПО «КРИСТА». В свою очередь на ПК должен быть установлен браузер Google Chrome версии 43 и выше.

Минимальные аппаратные средства:

- процессор двухъядерный с тактовой частотой 2Гц и выше;
- оперативная память не менее 4 Гб;
- свободное место на жестком диске – 300 Мб.

## 2.3.5. Вызов и загрузка

Клиентское приложение – расширение в DeveloperTools (DevTools) браузера Google Chrome. Такая реализация предполагает, что в браузере загружаются:

- страница с веб-приложением, для которого будут генерироваться шаблоны автоматизированных тестов;
- страница DevTools, в которой открывается расширение.

## 2.3.6. Входные и выходные данные

Входные данные:

- События от веб-приложения;
- События от корпоративного модуля CoreAnalytics.

Выходные данные:

- Характеристика события;
- Java-код автоматизированного теста;
- Характеристика сессии: дата и время, лог, код теста.

## 2.4. Программа и методика испытаний

### 2.4.1. Объект испытаний

Наименование программы – «Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования».

Область применения программы – автоматизация тестирования интерфейса программного продукта.

### 2.4.2. Цель испытаний

Целью предварительных испытаний является проверка системы на соответствие основным требованиям, прописанным в техническом задании и выявление ошибок в программе.

### 2.4.3. Требование к программе

1. Система должна предоставлять пользователю через графический интерфейс следующие функции:
   - Начать генерацию кода автоматизированного теста;
   - Закончить генерация кода автоматизированного теста;
   - Обновить лог событий;
   - Скопировать сгенерированный код;
   - Посмотреть сгенерированный код автоматизированного теста и лог событий выбранной сессии.
2. Система должна генерировать правильные строки кода автотеста под конкретную сущность ui-компонента веб-приложения;
3. Система должна устанавливаться в инструменты разработчика в браузере Google Chrome.

### 2.4.4. Состав и порядок испытаний

Технические средства, используемые во время испытаний:

- Офисный компьютер НПО «Криста»;
- Сервер НПО «Криста».

Программные средства, используемые во время испытаний:

- WebStorm 2021.1.1;
- Jest 24.9.0;
- Google Chrome 90;
- IntelliJ IDEA Ultimate 2021.1.1;
- JDK 1.8;
- Apache Tomcat 9;
- дистрибутив, предоставленный НПО «Криста»;
- библиотека для автоматизации тестирования интерфейса autotest-lib, предоставленная НПО «Криста».

Порядок проведения испытаний совпадает порядку, в котором перечислены проверяемые требования.

### 2.4.5. Методы испытаний

Список проводимых для требований испытаний представлен в *таблице 2.4*. Примеры автоматизированных тестов для модульного и функционального тестирования приведены в *приложение Д*. Примеры результатов других видов тестирования приведены в *таблице 2.5*.

Таблица 2.4 - Описание тестирования требований

| № требования | Вид тестирования | Описание | Ожидаемый результат |
|---|---|---|---|
| 1 | Модульное тестирование | Автоматизированная проверка корректности работы основных модулей системы | Успешное выполнение написанных тестов |
| | Функциональное тестирование | Ручное тестирование функционала системы посредством взаимодействия с графическим интерфейсом системы | Соответствие требованию № 1 |

Окончание таблицы 2.4

| № требования | Вид тестирования | Описание | Ожидаемый результат |
|---|---|---|---|
| 2 | Функциональное тестирование | Автоматизированная проверка корректности функционала системы, связанной с генерацией java-кода автоматизированного теста | Успешное выполнение написанных тестов |
| 3 | Тестирование совместимости | Ручная проверка установки системы | Соответствие требованию № 3 |

Таблица 2.5 - Пример оформления результатов испытаний

| Действие | Реакция системы | Результат испытания |
|---|---|---|
| Нажатие на кнопку «Закончить генерацию» | В окне «Лог» появляется таблица логов.<br>В окне «Код» появляется сгенерированный код автоматизированного теста.<br>В окне «Сессии» кнопка сессии с названием «Текущая» меняется на дату и время. | Соответствует требованию № 1 |
| Нажатие на кнопку определённой сессии | В окне «Лог» появляется таблица логов этой сессии.<br>В окне «Код» появляется сгенерированный код автоматизированного теста этой сессии. | Соответствует требованию № 1 |
| В расширения браузера добавлена папка с клиентской частью системы | Плагин находится в инструменте разработчика | Соответствует требованию № 3 |

# 3. Эксплуатационная документация на программный продукт

## 3.1. Руководство пользователя

### 3.1.1. Назначение программы

Программа предназначена для ускорения разработки функциональных автоматизированных тестов для интерфейса веб-приложений. Основными функциями системы являются:

- Начать генерацию кода автоматизированного теста;
- Закончить генерацию кода автоматизированного теста;
- Обновить лог событий;
- Скопировать сгенерированный код теста;
- Просмотреть сгенерированный код и лог событий выбранной сессии.

### 3.1.2. Условия выполнения программы

Плагин с пользовательским интерфейсом функционирует на персональных компьютерах с браузером Google Chrome версии 43 и выше. Для получения входных и выходных данных, необходимо наличие связи с веб-приложением, для которого будет генерироваться автоматизированный тест.

### 3.1.3. Установка и настройка программы

Для установки плагина необходимо:

1. Открыть страницу расширений в Google Chrome одним из двух способов:
    a. прейти по URL-адресу − «chrome://extensions»;
    b. открыть меню браузера, нажав по кнопке в правой верхней части экрана, далее выбрать пункт «Дополнительные инструменты» → «Расширения»;
2. Включить режим разработчика (перетащить вправо ползунок, находящийся в правой верхней части экрана);
3. Нажать на кнопку «Загрузить распакованное расширение» и выбрать папку с названием «chrome-extension» в появившемся диалоговом окне.

Программа появится среди установленных расширений (*Рисунок 3.27*).

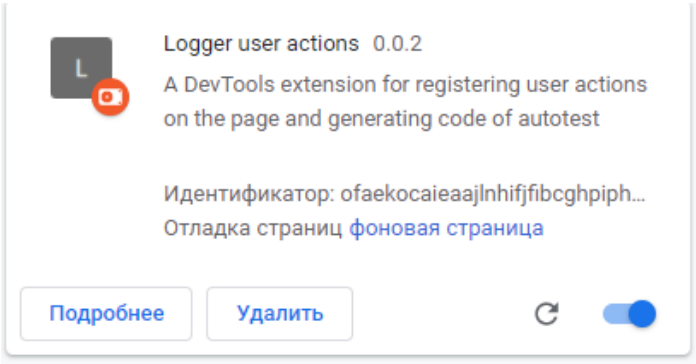


Рисунок 3.27 - Пример установленной программы в окне "Расширения"

### 3.1.4. Загрузка программы

Для загрузки установленной программы необходимо выполнить следующий ряд действий в браузере Google Chrome:

1. Открыть страницу с веб-приложением, для которого будут генерироваться функциональные автоматизированные тесты для интерфейса;
2. Открыть инструменты разработчика DevTools одним из трёх способов:
    a. нажать на клавиатуре F12 или Control + Shift+ I для Windows, Linux и Chrome O или Command + Option+ I для Mac;
    b. открыть меню браузера, нажав по кнопке в правой верхней части экрана, далее выбрать пункт «Дополнительные инструменты» → «Инструменты разработчика»;
    c. щёлкнуть правой кнопкой мыши по открытой странице и выбрать из появившегося окошка пункт «Просмотреть код»;
3. Открыть в DevTools вкладку с названием «Logger user actions» (*Рисунок 3.28*).

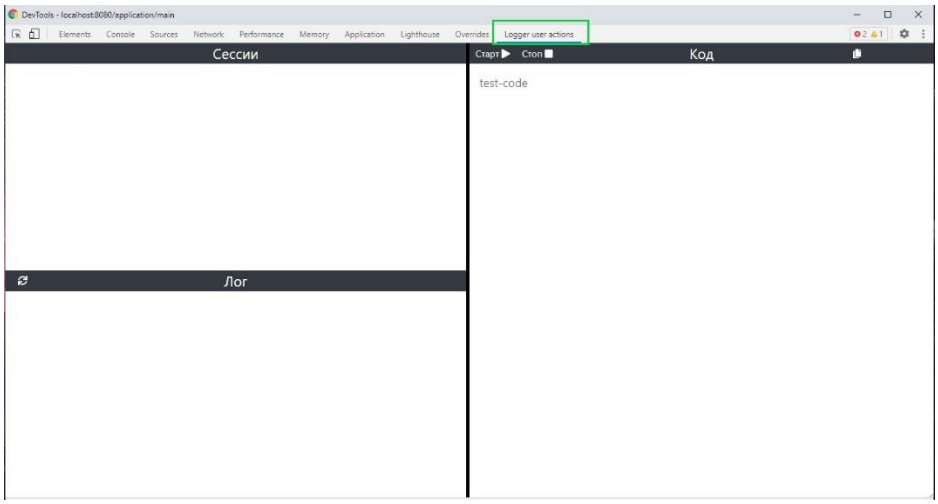


Рисунок 3.28 - Интерфейс, доступный после загрузки плагина

### 3.1.5. Управление выполнением программы

Общая схема доступа к функционалу программы изображена на *рисунке 3.29*.

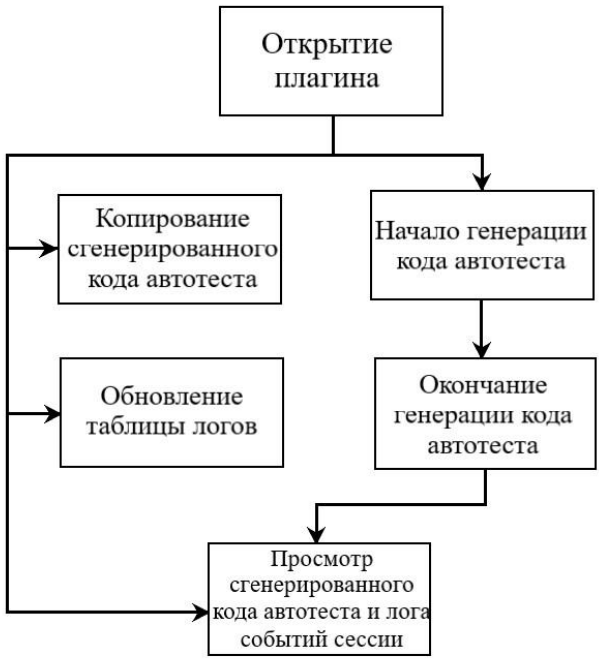


Рисунок 3.29 - Схема доступа к функционалу плагина

Интерфейс плагина разделён на три небольших функциональных окна:

1. Окно «Лог» отображает информацию о произошедших событиях на странице проекта. Информация содержит дату и время события, какое действие было совершено и по каким ui-компонентам. Имеется возможность обновления лога по соответствующей кнопке в верхнем левом углу окна (*Рисунок 3.30*).


Рисунок 3.30 - Окно "Лог"

2. Окно «Код» отображает сгенерированный код автоматизированного теста. В данном окне имеются кнопки «Старт» и «Стоп» для начала и

окончания генерации кода. Также есть возможность выделить весь код, щёлкнув левую кнопку мыши в любом месте текущей области (фокусирование), или копировать код для последующей с ним работы, нажав по кнопке в верхнем правом углу окна (*Рисунок 3.31*).

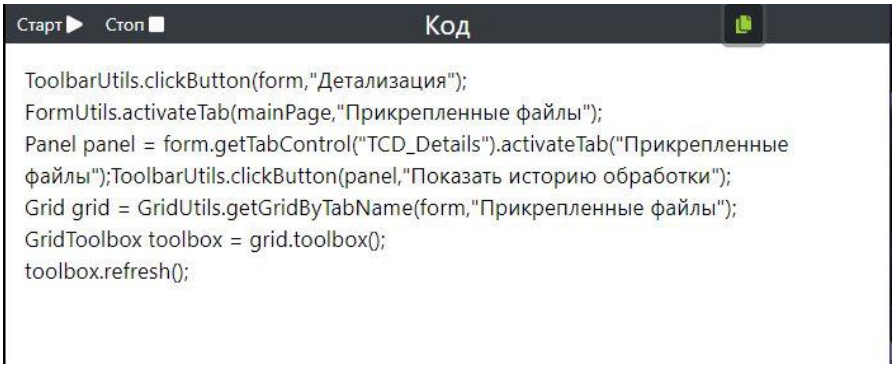


Рисунок 3.31 - Окно "Код"

3. Окно «Сессии» отображает текущую и предыдущие сессии работы с плагином. По нажатию на кнопки сессий в окнах «Код» и «Лог» будут отображаться соответственно сгенерированный код автоматизированного теста и лог событий выбранной сессии (*Рисунок 3.32*).

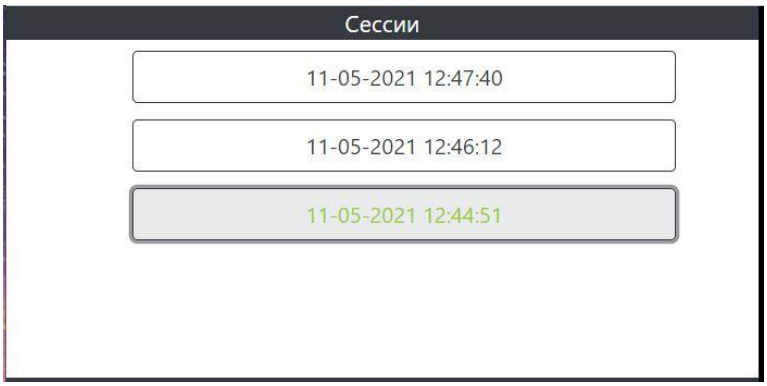


Рисунок 3.32 - Окно "Сессии"

Для того чтобы начать генерацию кода автоматизированного теста необходимо нажать кнопку *«Старт»* в окне «Код» в верхнем левом углу (*Рисунок 3.33*). В окне «Сессии» отобразится *текущая* сессия работы с плагином.

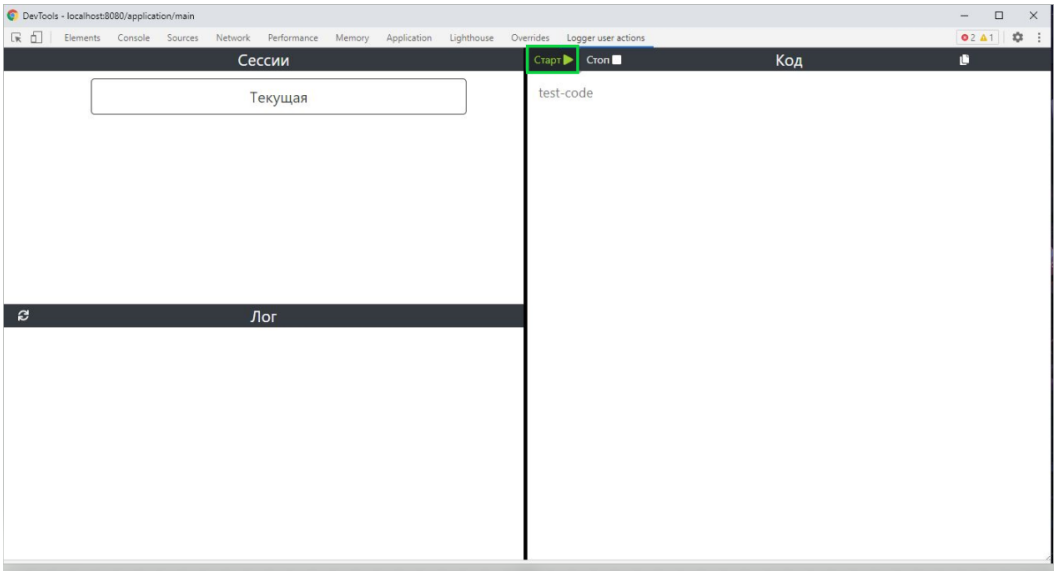


Рисунок 3.33 - Начало генерации кода автоматизированного теста

После этого переходим в веб-приложение (не закрывая страницу DevTools, а только сворачивая её). Совершив набор определённых действий на веб-странице проекта, необходимо вернутся на страницу DevTools и нажать кнопку *«Стоп»*, в верхнем левом углу окна «Код», для окончания генерации кода автоматизированного теста для текущей сессии. В итоге, получим сгенерированный код автоматизированного теста, написанный на языке программирования Java, в окне «Код» и лог событий в окне «Лог». В окне «Сессии» отобразится дата и время окончание сессии (*Рисунок 3.34*).

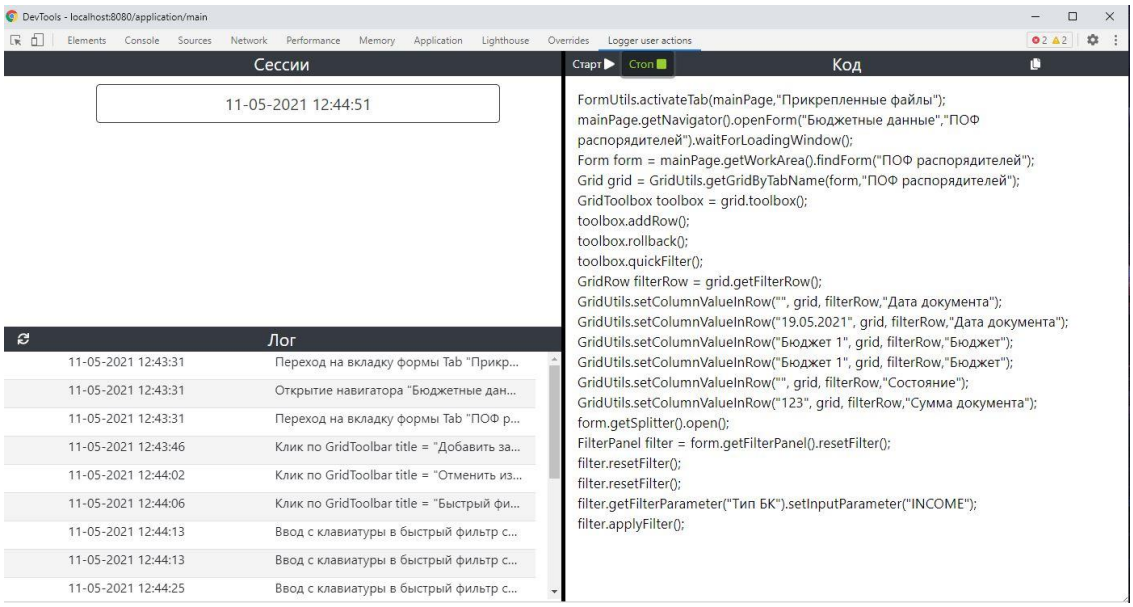


Рисунок 3.34 - Окончание генерации кода автоматизированного теста

Далее, чтобы создать ещё один шаблон автоматизированного теста, т.е. новую сессию, повторяем все те же действия, с нажатия кнопки «Старт».

### 3.1.6. Завершение программы

Завершение работы программы происходит при закрытии DevTools.

# 4. Акт испытаний программного продукта

## 4.1. Объект испытаний

Наименование программы – «Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования».

Область применения программы – автоматизация тестирования интерфейса программного продукта.

## 4.2. Цель испытаний

Целью предварительных испытаний является проверка системы на соответствие основным требованиям, прописанным в техническом задании и выявление ошибок в программе.

## 4.3. Результаты испытаний

Результаты модульного тестирования системы, как и функционального тестирования генерации кода автоматизированного теста, прилагаются к коду программы в виде тестовых файлов, которые содержат в себе имя тестируемого модуля, оканчивающееся на «.test». Например, имя файла с тестами для модуля SessionReducer, где определены реакции на события, получаемые от контроллера компонентов системы, описанного в SessionReducer.ts, – SessionReducer.test.ts (*Приложение Д*).

Результаты функционального тестирования системы через её графический интерфейс приведены в *таблице 4.6*.

Результаты установочного тестирования приведены в *таблице 4.7*. Плагин устанавливался на персональном компьютере сотрудника НПО «Криста».

Таблица 4.6 - Результаты функционального тестирования системы

| Действие | Реакция системы | Результат испытаний |
|---|---|---|
| Нажатие на кнопку «Начать генерацию» | - В окне «Сессии» появится кнопка сессии с названием «Текущая»;<br>- Система начнёт фиксировать и записывать действия пользователя. | Соответствует требованию № 1 |

Окончание таблицы 4.6

| Действие | Реакция системы | Результат испытаний |
|---|---|---|
| Нажатие на кнопку «Закончить генерацию» | - Система генерирует код автоматизированного теста по записанным действиям пользователя;<br>- В окне «Лог» появляется таблица логов;<br>- В окне «Код» появляется сгенерированный код автоматизированного теста;<br>- В окне «Сессии» кнопка сессии с названием «Текущая» меняется на дату и время. | Соответствует требованию № 1 |
| Нажатие на кнопку «Обновить лог» | Происходит обновление лога. В окне «Лог» появляются недостающие строки таблицы логирования, если они имеются. | Соответствует требованию № 1 |
| Нажатие на кнопку «Скопировать код» | В буфер обмена скопирован сгенерированный код, который вывелся в окне «Код». | Соответствует требованию № 1 |
| Нажатие на кнопку определённой сессии | - В окне «Лог» появляется таблица логов этой сессии;<br>- В окне «Код» появляется сгенерированный код автоматизированного теста этой сессии. | Соответствует требованию № 1 |

Таблица 4.7 - Результаты установочного тестирования

| Действие | Реакция системы | Результат испытаний |
|---|---|---|
| В расширения браузера добавлена папка с клиентской частью системы | Плагин находится в инструментах разработчика | Соответствует требованию № 3 |

## 4.4. Выводы

На основании результатов испытаний можно сделать вывод, что система соответствует основным требованиям, прописанным в техническом задании, и никаких ошибок не было выявлено.

Разработчик _____ Куликова Э. В.
Руководитель работы _____ Смирнов Н. В.

# 5. Экономическое обоснование

## 5.1. Экономическое обоснование разработки ПО

В настоящее время многие IT-проекты имеют сложную разнообразную функциональность и отличаются очень короткими промежутками между релизами. Поэтому разработчики вынуждены часто выполнять большое количество повторяющихся проверок. Именно этот факт является одной из причин активного развития автоматизации тестирования. Всё больше IT-компаний приходят к решению оптимизации процесса тестирования, что позволит им сократить финансовые и временные затраты.

Компания «НПО «Криста» разрабатывает большие многофункциональные веб-приложения, поэтому они занимаются тестированием пользовательского интерфейса. Поскольку нажатие функциональных кнопок, открытие вкладок, заполнение полей ввода и тому подобные операции являются рутинными, то для тестирования UI (User Interface) веб-приложений в компании разработали специальную библиотеку autotest-lib, способную воспроизвести действия пользователя в автоматизированном режиме.

Написания тестов-проверок для таких однообразных операций даже с уже разработанной библиотекой автоматизации autotest-lib всё ещё является монотонной задачей, которая затрачивает большое количество рабочего времени разработчика. Чтобы автоматизировать также и текущий процесс, было принято решение записывать макросы (микрокоманды) действий пользователя для последующего преобразования их в программный код библиотеки autotest-lib, что в последствии ускорит процесс тестирования веб-приложений, позволяя увеличить количество автоматизированных функциональных тестов, а, следовательно, и улучшить качество программного продукта.

Для упрощения автоматизации тестирования применяют вспомогательные инструменты для записи тестов, так называемые, рекодеры, такие как, например, Selenium IDE и Wildfire.ai. Их принцип действия следующий. Пользователь совершает какие-то манипуляции в браузере, а его действия записываются в виде последовательности незамысловатых команд. Далее сохраняя этот тест, его можно запускать для самостоятельного повторения браузером. Благодаря таким рекодерам, легко производится автоматизация рутинных проверок и уменьшается количество ручной работы.

При анализе целесообразности использования рекордеров, представленных выше, был сделан вывод о том, что они не подходят для поставленной задачи. Во-первых, это обуславливается тем, что интерфейсы

веб-систем в компании «НПО «Криста» состоят из собственно разработанных UI-компонентов и для тестирования данных компонентов используется также разработанная компанией библиотека «autotest-lib». А во-вторых, ни один из перечисленных аналогов не имеет возможности трансформировать записанные действия пользователя в программный код, что позволяет запускать тесты внутри самих веб-приложений компании.

Таким образом, отсюда возникает потребность создание собственного плагина, который будет регистрировать действия пользователя в веб-системах «НПО «Криста» и генерировать код автоматизированных тестов под корпоративную библиотеку autotest-lib.

## 5.2. Описание ПО

Система регистрации действий пользователя в браузере с генерацией кода для автоматизации тестирования предназначена для ускорения разработки функциональных автоматизированных тестов, что должно позволить увеличить количество тестов и, соответственно, качество программных продуктов.

Вид автоматизируемой деятельности – тестирование.

Объект автоматизации – процесс написания тестов.

Цель создания – упростить процесс автоматизации тестирования, сократить время на написания автоматизированных тестов и увеличить количество тестов.

Область применения программы – автоматизация тестирования интерфейса программных продуктов компании «НПО «Криста».

Потенциальными пользователями системы будут являться тестировщики компании «НПО «Криста».

## 5.3. Расчёт затрат на разработку программного обеспечения

Для расчета затрат на разработку программного обеспечения, для начала необходимо расписать длительность всех этапов, которые будут пройдены в ходе выполнения работы.

В *таблице 5.8* представлены необходимые работы, их максимальная, минимальная и ожидаемая длительность. Как видно по таблице ожидаемая длительность всех работ составляет 93 дней. Данный срок вполне приемлем для разработки системы.

Таблица 5.8 - Расчёт длительности работ

| Наименование работ | Длительность работ (в рабочих днях) | | |
|---|---|---|---|
| | Минимум | Максимум | Ожидаемая |
| Изучение функциональных требований | 1 | 2 | 2 |
| Изучение литературы | 2 | 4 | 3 |
| Разработка технического задания | 3 | 10 | 6 |
| Разработка алгоритмов | 3 | 6 | 5 |
| Программирование | 30 | 60 | 40 |
| Тестирование | 5 | 10 | 7 |
| Отладка и исправление ошибок | 5 | 20 | 13 |
| Разработка документации | 10 | 25 | 17 |
| Общая длительность работ | 59 | 137 | 93 |

Исходя из данных о работах и их длительности, необходимо рассчитать затраты на разработку ПО. В *таблице 5.9* приводится расчет затрат на основную заработную плату.

Расчет величины основной заработной платы $З_о$, руб. участников команды производится по формуле ([10], с. 16, формула (7)):

$$З_о = \sum_{i=1}^{n} Т_{чi} \times t_i \ , \qquad (5.1)$$

где n – количество исполнителей, занятых разработкой конкретного ПО;

$Т_{чi}$ – часовая заработная плата i-го исполнителя, руб;

$t_i$ – трудоемкость работ, выполняемых i-м исполнителем, ч.

Трудоемкость рассчитывается по формуле ([10], с. 17, формула (8)):

$$Т_{ож} = (3 \cdot T_{min} + 2 \cdot T_{max})/5 \ , \qquad (5.2)$$

где $Т_{ож}$ – ожидаемая продолжительность работ;

$Т_{min}$ и $Т_{max}$ – соответственно наименьшая и наибольшая по мнению эксперта длительность работы.

Для разработки любой системы или подсистемы необходимо несколько человек, так как один разработчик со всем справиться не сможет. Поэтому в данном случае в команду входят один программист, один аналитик и один тестировщик.

На примере программиста покажем, как рассчитывается его заработная плата. Трудоемкость работы программиста, равна следующему значению:

$$T_{\text{ож}} = (3 \cdot 40 + 2 \cdot 90) / 5 = 60 \text{ д.} = 480 \text{ ч.}$$

Поскольку месячная заработная плата программиста составляет 25 000 рублей, то часовая зарплата составит:

$$T_{\text{ч}i} = 25000 / (22 \cdot 8) = 142 \text{ руб./ч.}$$

Таблица 5.9 - Расчёты основной заработной платы

| № | Участник команды | Выполняемые работы (табл. 5.8) | Месячная з/п, р. | Часовая з/п, р. | Трудоемкость работ, часов | Основная з/п,р. |
|---|---|---|---|---|---|---|
| 1 | Программист | 2, 4, 5, 7 | 25 000 | 142 | 480 | 68 160 |
| 2 | Аналитик | 1, 3, 8 | 30 000 | 171 | 192 | 32 832 |
| 3 | Тестировщик | 6 | 20 000 | 114 | 56 | 6 384 |
| Итого затраты на основную заработную плату разработчиков | | | | | 107 376 | |

Затраты на дополнительную заработную плату разработчика включают выплаты, предусмотренные законодательством о труде (оплата отпусков, льготных часов, времени выполнения государственных обязанностей и других выплат, не связанных с основной деятельностью исполнителей), и определяются по формуле ([10], с. 18, формула (9)):

$$З_{\text{д}} = \frac{З_{\text{о}} \times Н_{\text{д}}}{100} \quad (5.3)$$

где $З_{\text{о}}$ – затраты на основную заработную плату с учетом премии (руб.);

$Н_{\text{д}}$ – норматив дополнительной заработной платы, который равен 10%.

Тогда затраты на дополнительную заработную плату составят:

$$З_{\text{д}} = \frac{107\,376 \times 10}{100} = 10\,738 \text{ руб.}$$

Социальные отчисления (в фонд социальной защиты населения и на обязательное страхование) определяются в соответствии с действующими законодательными актами по формуле ([10], с. 18, формула (10)):

$$З_{\text{соц}} = \frac{(З_{\text{о}} + З_{\text{д}}) \times Н_{\text{соц}}}{100}, \quad (5.4)$$

где $Н_{\text{соц}}$ – норматив отчислений на социальные нужды (согласно действующему законодательству) равный 30%.

Тогда социальные отчисления равны:

$$З_{соц} = \frac{(107\,376 + 10\,738\,)\times 30}{100} = 35\,434 \text{ руб.}$$

Рассчитаем затраты на эксплуатацию оборудования по формуле ([10], с. 19, формула (15)):

$$Z_{зэ} = Z_{зр} + Z_{ам} + Z_{эт} + Z_{вм} + Z_{тр} + Z_{пр} \quad (5.5)$$

где $Z_{зр}$ – издержки на заработную плату обслуживающего персонала, руб./год;

$Z_{ам}$ – годовые издержки на амортизацию, руб./год;

$Z_{эт}$ – годовые издержки на электроэнергию, потребляемую ЭВМ, руб./год;

$Z_{вм}$ – годовые издержки на вспомогательные материалы, руб./год;

$Z_{тр}$ – затраты на текущий ремонт компьютера, руб./год;

$Z_{пр}$ – прочие и накладные расходы, руб./год;

$Z_{зр}$ в данной формуле принимаем равной нулю.

Сумма годовых амортизационных отчислений определяется по формуле ([10], с. 20, формула (16)):

$$Z_{ам} = C_{ба} \times H_{ам}, \quad (5.6)$$

где $С_{ба}$ – балансовая стоимость компьютера, р.;

$Н_{ам}$ – норма амортизации, %.

Для вычислительной техники норма амортизации допускается в размере 25 %. $C_{ба}$ в данном случае равно 40 000 руб. Тогда:

$$Z_{ам} = 40\,000 \times 0{,}25 = 10\,000 \text{ руб.}$$

Стоимость электроэнергии, потребляемой за год, определяется по формуле ([11], с. 15):

$$Z_{эт} = C_{эт} \times T_{эф} \times P_{сп} \times A, \quad (5.7)$$

где $С_{эт}$ – стоимость 1 кВт/ч электроэнергии, равная 3,84 руб.;

$Т_{эф}$ – действительный годовой фонд времени работы ЭВМ, час/год;

$Р_{сп}$ – суммарная потребляемая мощность ЭВМ, равная 0,5 кВт;

$A$ – коэффициент интенсивного использования мощности машины.

Действительный годовой фонд времени ЭВМ равняется числу рабочих часов для участников команды, за вычетом времени на профилактику и ремонт ЭВМ. Время профилактики: ежемесячная – пять часов, ежегодная – 7 суток. Следовательно, $T_{гф}$ = 260 × 8 – (7 * 8 + 5 * 12) = 2 080 – 116 = 1 964 ч. Так как фактическое время работы ЭВМ совпадает с плановым временем работы

участников команды, следовательно, коэффициент интенсивного использования мощности машины принимается равным единице. Тогда по формуле 1.7 находится значение

$$Z_{эт} = C_{эт} \times T_{эф} \times P_{сп} \times A = 3{,}84 \times 660 \times 0{,}5 \times 1 \approx 1\,267 \text{ руб./год.}$$

Затраты на материалы, необходимые для обеспечения нормальной работы ЭВМ, составляют примерно 1% от стоимости ЭВМ:

$$Z_{вм} = 0.01 \times 40000 = 400 \text{ руб./год.}$$

Затраты на текущий и профилактический ремонт принимаются равными 5% от стоимости ЭВМ:

$$Z_{тр} = 0.05 \times 40000 = 2000 \text{ руб./год.}$$

Прочие косвенные затраты, связанные с эксплуатацией ЭВМ, состоят из амортизационных отчислений здания, стоимости услуг сторонних организаций и составляют около 5 % от стоимости ЭВМ:

$$Z_{пр} = 0.05 \times 40000 = 2000 \text{ руб./год.}$$

В таком случае, получаем:

$$Z_{зэ} = 10\,000 + 1\,267 + 400 + 2000 + 2000 = 15\,667 \text{ руб./год.}$$

Цену машино-часа можно определить по формуле ([10], с. 19, формула (14)):

$$C_{мч} = Z_{зэ} / T_{гф}\,, \tag{5.8}$$

где $Z_{зэ}$ – полные затраты на эксплуатацию ЭВМ в течение года;

$T_{гф}$ – действительный годовой фонд времени работы ЭВМ, час/год.

$$C_{мч} = 15\,667 / 1\,964 \approx 8 \text{ руб./ч.}$$

Для расчета фактического времени отладки программы используется формула ([10], с. 19, формула (13)):

$$t_{фв} = t_{бс} + t_{д} + t_{о}\,, \tag{5.9}$$

где $t_{бс}$ – затраты труда на составление программы по готовой блок-схеме, ч;

$t_{д}$ – затраты труда на подготовку документации задачи, ч;

$t_{о}$ – затраты труда на отладку программы на ЭВМ при комплексной отладке задачи, ч.

$$t_{фв} = 480 + 192 + 56 = 728 \text{ ч}$$

Тогда затраты на машинное время, составят ([10], с. 19, формула (12)):

$$Z_{омв} = C_{мч} \times t_{фв} = 8 \times 728 = 5\,824 \text{ руб.}$$

Расчет прочих затрат осуществляется в процентах от затрат на основную заработную плату разработчика с учетом премии по формуле ([10], с. 20, формула (18)):

$$З_{пз} = \frac{З_о \times Н_{пз}}{100} \quad (5.10)$$

где $Н_{ПЗ}$ – норматив прочих затрат. Норматив рекомендуется брать в пределах 100 – 150 %, в данном случае возьмем его равным 120%.

Подставим значения в формулу 1.10 для расчета прочих затрат:

$$З_{пз} = \frac{З_о \times Н_{ПЗ}}{100} = \frac{107\,376 \times 120}{100} = 128\,851 \text{ руб.}$$

Полная сумма затрат на разработку программного обеспечения находится путем суммирования всех рассчитанных статей затрат (*Таблица 5.10*).

Таблица 5.10 - Затраты на разработку ПО

| Статья затрат | Сумма, р |
|---|---|
| Основная заработная плата команды разработчиков ($З_о$) | 107 376 |
| Дополнительная заработная плата команды разработчиков ($З_д$) | 10 738 |
| Социальные отчисления ($З_{соц}$ ) | 35 434 |
| Машинное время ($Z_{омв}$ ) | 5824 |
| Прочие затраты ($З_{пз}$) | 128 851 |
| Общая сумма затрат на разработку | 288 223 |

Таким образом, себестоимость разработки программного обеспечения составляет 288 223 руб.

## 5.4. Оценка результата от внедрения ПО

Экономический эффект представляет собой прирост чистой прибыли, полученный организацией в результате использования разработанного ПО. В данном случае он может быть достигнут за счет того, что снижается время на написание автоматизированных тестов тестировщиками компании для программных продуктов. Процесс написания тестов автоматизируется. Тесты генерируются сами, достаточно просто произвести нужный набор действий в программном продукте для интерфейсных компонентов системы, которые необходимо проверить.

Повышение производительности труда можно рассчитать по формуле ([10], с. 22, формула (21)):

$$P_i = \left(\frac{\Delta T_j}{F_j - \Delta T_j}\right) \times 100 \quad (5.11)$$

где Fj – время, которое планировалось пользователем для выполнения работы j-го вида до внедрения программы, ч;

$\Delta T_i$ - количество часов, которые экономит пользователь.

Примерные виды работ, которыми занимаются тестировщики: сконструировать тест, нахождение необходимых идентификаторов компонентов интерфейса для его проверки, а также собственно написание самого автоматизированнго теста в целом. Если сравнить сколько минут выполняется работа до автоматизации и сколько позже, то можно увидеть сколько экономиться времени и насколько повышается производительность труда.

В *таблице 5.11* приведено повышение производительности труда при различных видах работ пользователей.

Таблица 5.11 - Повышение производительности выполнения работ

| Вид работ | До автоматизации, мин (*Fj)* | Экономия времени, мин. | Повышение производительности труда $P_i$, % |
|---|---|---|---|
| Сконструировать тест | 15 | 10 | 200 |
| Нахождение идентификаторов компонентов интерфейса для его проверки | 15 | 12 | 400 |
| Написание автоматизированного теста | 30 | 20 | 200 |

Из данной *Таблица 5.11* видно, что самые трудоемкие виды работ: нахождение идентификаторов компонентов интерфейса для его проверки и проектирование и написание самого автотеста. Автоматизация данных видов работ позволяет сэкономить большое количество времени, и значительно повышает производительность труда.

Среднемесячная зарплата тестировщика компании составляет 25 000 рублей, тогда часовая зарплата составит 142 руб./ч. В день он успевает сделать примерно 4 задачи, связанных с проверкой интерфейсных компонентов систем и на каждую задачу у него уходит примерно час времени. Следовательно, за выполнение именно этой части работы, его заработок составляет: (142 руб./ч × 4 ч) × 22 дня = 12 496 руб.

Тогда, с учётом того, что плагин предположительно будет пользоваться как минимум 10-ми тестировщиками, то экономия, связанная с повышением производительности труда будет составлять ([10], с. 23, формула (22)):

$$\Delta P = Z_{\text{п}} \times \sum_i \frac{P_i}{100} = (12\,496 \times 10) \times 3{,}3 = 412\,368 \text{ руб.}$$

Годовая экономия $Э_р$ (руб./год) складывается из экономии эксплуатационных расходов и экономии в связи с повышением производительности труда пользователя ([10], с. 22, формула (20)):

$$Э_р = (P_1 - P_2) + \Delta P_п, \quad (5.12)$$

где $P_1$ и $P_2$ – соответственно эксплуатационные расходы до и после внедрения разрабатываемой программы;

$\Delta P_{\text{п}}$ – экономия от повышения производительности труда дополнительных пользователей.

Если эксплуатационные расходы на тестирование интерфейсов проекта компании составляют примерно 5 000 руб. за один проект в месяц, а таких крупных проектов 3, то за год ожидается экономия в ((5 000 руб. × 3) × 12) / 3,3 = 54 545,45 руб.

Следовательно, по формуле 5.12 годовая экономия будет:

$$Эр = 55\,545{,}45 + 412\,368 = 466\,913{,}45 \text{ руб./год.}$$

Критерием эффективности создания и внедрения новых средств автоматизации является ожидаемый экономический эффект. Он определяется по формуле ([10], с. 22, формула (19)):

$$Э = Э_р - E_н \times K_п, \quad (5.13)$$

где $Э_{\text{р}}$ – годовая экономия, р.;

$E_{\text{н}}$ – нормативный коэффициент ($E_{\text{н}} = 0{,}15$);

$K_{\text{п}}$ – капитальные затраты на проектирование и внедрение, включая первоначальную стоимость программы, р.

По формуле 5.13 выполняется расчет ожидаемого экономического эффекта:

$$Э = 466\,913{,}45 - 0{,}15 \times 288\,223 = 423\,680 \text{ руб.}$$

По данному ожидаемому экономическому эффекту видно, что внедрение плагина благоприятно скажется для предприятия. Сократит количество расходов и при этом цена от внедрения подсистемы окупится в будущем, за счёт экономии средств.

До внедрения данной системы большинство тип работ выполнялись вручную. Из-за этого время значительно увеличивалось, тестировщикам

компании приходилось более тщательно и внимательно разбираться с малейшими элементами, чтобы не совершать ошибок. Любая ошибка приводила к долгому поиску причины и времени её исправления.

После автоматизации время, требуемое на выполнение работ, значительно сократилось. Это позволило сотрудникам выполнять гораздо большее количество работ в день. Сократилось также и количество ошибок при работе, например, при написание автоматизированных тестов. И теперь ошибки являются не такими фатальными их можно поправить быстрее, чем ранее.

# Заключение

В процессе выполнения ВКР был проанализирован процесс тестирования и его автоматизация. На основании полученных знаний были сформированы требования для разработки системы.

Для реализации требований был спроектирован и разработан плагин (расширение) для браузера и создана база данных в IndexedDB, в которой хранятся основные данные о пользовательских действиях. Полученная система на основании работы пользователя в веб-приложение позволяет фиксировать его действия на веб-странице приложения и генерировать строки кода автоматизированных тестов, адаптированных под корпоративную библиотеку autotest-lib. В результате проведённого тестирования плагина было установлено, что система соответствует всем указанным требованиям.

К созданному плагину была разработана соответствующая программная документация, а именно, техническое задание, пояснительная записка, описание программы, а также программа и методика испытания. В качестве эксплуатационной документации было разработано руководство пользователя.

Таким образом, был получен плагин для браузера, который регистрирует действия пользователя в веб системах «НПО «Криста», для последующего преобразования в программный код, способный повторить эти действия в автоматизированном режиме с учетом различных тонкостей, разрабатываемых предприятием программных продуктов.

Полученная система может использоваться специалистами предметной области для ускорения разработки функциональных автоматизированных тестов для интерфейса веб-приложений НПО «Криста».

# Список используемых источников


1. ПроТестинг.ru: сайт. – 2008. – URL: http://www.protesting.ru (дата обращения: 26.04.2021);
2. Классификация видов тестирования: статья. – 2018. – URL: https://qa-academy.by/qaacademy/news/klassifikaciya-vidov-testirovaniya/ (дата обращения: 26.04.2021);
3. Свободная энциклопедия: сайт. URL: https://ru.wikipedia.org (дата обращения: 28.05.2021 – 5.05.2021);
4. Как создать расширение для Chrome: статья. – 2019. – URL: https://devacademy.ru/article/kak-sozdat-rasshirenie-dlya-chrome (дата обращения: 29.04.2021);
5. Документация о расширениях для Google Chrome: документация. URL: https://developer.chrome.com/docs/extensions (дата обращения: 29.04.2021);
6. Обзор расширения Selenium IDE: интернет-магазин chrome. URL: https://chrome.google.com/webstore/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd (дата обращения: 30.04.2021);
7. WildFire: сайт. – 2017. – URL: https://wildfire.ai (дата обращения: 30.04.2021);
8. TypeScript and Babel 7: статья. – 2018. – URL: https://devblogs.microsoft.com/typescript/typescript-and-babel-7/ (дата обращения: 02.05.2021);
9. Redux: документация. – 2018. – URL: https://redux.js.org/introduction (дата обращения: 02.05.2021);
10. Экономические основы рынка программного обеспечения и вычислительной техники: Программа учебной дисциплины и методические указания к выполнению контрольной работы /Сост. Н. А. Клементьева; РГАТУ имени П. А. Соловьева. – Рыбинск, 2016. – 27 с;
11. Расчет годового фонда времени работы ПК: статья. – 2015. – URL: https://mydocx.ru/9-62729.html (дата обращения: 02.06.2021).

# Приложение А. Код модуля Agent

```
import sendMessage from "./utils/sendMessage";
import { MessageTypeAgent } from "../ui/util/inspectWindow/sendMessage";
import {
   deleteItem,
   deleteItems,
   getConnection,
   getDataFromDB,
   IUserActionType,
   saveItem
} from "../indexedDB/indexeddb";
import {
   checkEventChange,
   defineParentsObject,
   getChildrenOfObject,
   getGridCellColumnName,
   getQuickFilterColumnName,
   getXPath,
   Types
} from "./utils/forDefineUIComponents";

//для поиска нужных элементов data-control-type
enum ControlTypes {
   grid = "Grid",
   tab = "TabContainer"
}
//для поиска элементов по class
enum ClassTypes {
   bigButton = "big-small-button",
   bigButtonCombo = "big-small-combo",
   bigButtonScroll = "big-float",
   gridToolbar = "grid-toolbar-button",
   cell = "cell",
   quickFilter = "filter",
   quickFilterCol = "[object Object]",
   consolidationFilterToolbar = "propertyfilter-toolbar",
   consolidationFilterCaption = "propertyfilter-table",
   filterToolbar = "button__toolbar",
   filterCaption = "propertyTableRow",
   placeHolderDate = "datepicker-placeHolder",
   check = "checked",
```

```
    uncheck = "unchecked"
}

//для поиска элементов по id
enum IdTypes {
    modalWindowOk = "mw_ok",
    modalWindowClose = "mw_close"
}

const DATA_TYPE: string = "data-control-type";
const CLASS_TYPE: string = "class";
const ID_TYPE: string = "id";

let formName: string;
let tabActive: string | undefined;
let tabContainer: string | null | undefined;
let formsPathNavigator: string = "";
let isConsolidation: boolean = false;

type EventFunction = (e: Event) => void;
type GetFunction = (between: [number, number]) => void;

interface AgentHandlersType {
    connect: EventFunction;
    getData: GetFunction;
    clearData: VoidFunction;
}

/**класс Agent**/
export default class Agent {
    private readonly window: Window;
    handlers: AgentHandlersType;

    constructor(c: Window) {
        this.window = c;
        console.log("Agent constructor (c) >> ", c);

        this.initIndexedDB();
        this.initDevtoolsMessageListener();
        this.getDataFromLogger();

        this.handlers = {
```

```
        // Broadcast when the dev tools are opened
        connect: function() {
          console.log("Browser connect handler -> connected");
          sendMessage("connected", "");
        },

        //получит данные из indexedDB
        getData: async function(between: [number, number]) {
          console.log("agent getData");
          sendMessage("setDateEnd", between[1]); //отправить конечную дату сессии
          let listCode: IUserActionType[] = await getDataFromDB();

          const from = between[0];
          const to = between[1];
          listCode = listCode.filter(value => from <= value.date && value.date <= to);

          sendMessage("setData", listCode);
        },

        //очистить данные в БД
        clearData: function() {
          console.log("ClearData");
          deleteItems();
        }
      };
    }

    //обработчик событий, приходящих от логгера CoreAnalytics
    analyticHandler(data: any) {
      console.log("MYEVENT:", data);

      let type: string | undefined;
      let dataMap: Map<string, string | undefined | boolean> = new Map();

      //открытие/обновление/закрытие формы
      if (
        (data.controlType === "Form" && data.operation === Types.open) ||
        (data.controlType === "Container" && (data.operation === Types.refresh || data.operation ===
Types.close))
      ) {
        formName = data.controlLabel;
        dataMap.set("formName", data.controlLabel);
```

```
    dataMap.set("formDFDName", data.formName);
    type = data.operation;
}
console.log("FORM:", formName, "TYPE:", type);

//открытие формы по навигатору
if (data.controlType === "Navigator" && data.operation === Types.open) {
    formsPathNavigator += data.controlLabel + '","';
    type = data.operation;
}
console.log("NAVIGATOR:", formsPathNavigator, "TYPE:", type);
if (formsPathNavigator !== "" && formsPathNavigator.includes(formName?.replace(/ё/g, "e"))) {
    dataMap.set("navigator", formsPathNavigator);
    type = Types.open;
    formsPathNavigator = "";
}

//сделать активным/масштабирование таб
if (
    data.controlType === "Container" &&
    (data.operation === Types.setActive ||
        data.operation === Types.minimize ||
        data.operation === Types.fullscreen ||
        data.operation === Types.refresh)
) {
    let reg = /[A-Z]/gi;
    if (data.controlLabel.match(reg) !== null) {
        tabActive = data.formName;
    } else tabActive = data.controlLabel;
    dataMap.set("tab", tabActive);
    dataMap.set("formName", data.formName);
    type = data.operation;
    console.log("TAB:", tabActive, "TYPE:", type);
}

//открытие вкладки фильтра
if (
    data.controlLabel?.includes("Фильтр") &&
    (data.operation === Types.slamming || data.operation === Types.setActive)
) {
    dataMap.set("filter", true);
    type = data.operation;
```

```
    }

    const newData: any | IUserActionType = setDataObjectFromCoreAnalyticInIndexedDB(dataMap,
type); //формирование объекта данных для indexedDB
    if (newData === null) return; //если не было никакого взаимодействия пользователя ни с одним
ui-компонентом
    saveItem(newData); //сохраняем объект в indexedDB
  }

  //подписываться на события логгера CoreAnalytics
  getDataFromLogger() {
    // @ts-ignore
    if (window.CoreAnalytics._applicationName === "consolidation") isConsolidation = true;
//проверяем является ли открытый проект "Консолидацией"
    // @ts-ignore
    window.CoreAnalytics._observer.subscribe(this.analyticHandler);
    // @ts-ignore
    //window.CoreAnalytics._observer.unsubscribe(analyticHandler);
  }

  initDevtoolsMessageListener() {
    window.addEventListener("message", event => {
      // Only accept messages from same frame
      // event.source - источник события
      if (event.source !== window) {
        return;
      }

      const message = event.data;

      // Only accept messages of correct format (our messages)
      if (typeof message !== "object" || message === null || message.source !== "coquette-inspect-
devtools") {
        return;
      }

      this.handleMessage(message);
    });
  }

  initIndexedDB() {
    getConnection(); //подключаемся к indexedDB
```

```
  }

  //отклик на события
  callback(type: Types, e: any) {
    let mapOfComponents: Map<string, HTMLElement> = new Map(); //хранит ui-компоненты, с которыми взаимодействовал пользователь

    try {
      //нахождение ui-компонентов, с которыми взаимодействовал пользователь

      this.findTabContainer(e, mapOfComponents);
      this.findBigButton(e, mapOfComponents);
      this.findGridToolbar(e, mapOfComponents);
      type = this.findCheckBox(e, mapOfComponents, type);
      this.findGridCell(e, mapOfComponents, type);
      this.findQuickFilter(e, mapOfComponents, type);
      this.findModalWindow(e, mapOfComponents, type);
      this.findFilter(e, mapOfComponents, type);
      this.findGrid(e, mapOfComponents);

      //для полей дат
      const placeHolder: any = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.placeHolderDate)
      );

      $(placeHolder).bind(Types.change, () => {
        console.log("DATEPICKER", e.target.value, type);
        if (e.target.value !== undefined && type === Types.blur) {
          this.callback(Types.change, e);
        }
        $(placeHolder).unbind();
      });

      const data: any | IUserActionType = setDataObjectInIndexedDB(mapOfComponents, type, e); //формирование объекта данных для indexedDB
      if (data === null) return; //если не было никакого взаимодействия ни с одним ui-компонентом
      saveItem(data); //сохраняем объект в indexedDB
      if (checkEventChange(data) || data.type === Types.focus) deleteItem(data); //удаляем записи из indexedDB, которые не несут полезной информации
    } catch (error) {
      if (error instanceof TypeError) {
```

```
            console.log("TypeError");
        }
    }
}

//включен режим инспектора
attachSelectClickHandler = () => {
    this.window.addEventListener(
        Types.click,
        (e: MouseEvent) => {
            if (e.detail === 1) {
                console.log("CLICK");
                this.callback(Types.click, e);
            } else if (e.detail === 2) {
                this.callback(Types.dblclick, e);
            }
        },
        true
    );
    this.window.addEventListener(
        Types.change,
        (e: Event) => {
            this.callback(Types.change, e);
        },
        true
    );
    this.window.addEventListener(
        Types.focus,
        (e: Event) => {
            console.log("FOCUS");
            this.callback(Types.focus, e);
        },
        true
    );
    this.window.addEventListener(
        Types.blur,
        (e: Event) => {
            console.log("BLUR");
            this.callback(Types.blur, e);
        },
        true
    );
```

```
};

removeSelectClickHandler() {
   this.window.removeEventListener(
      Types.click,
      function() {
         console.log("CLICK END");
      },
      true
   );
   this.window.removeEventListener(
      Types.change,
      function() {
         console.log("CHANGE END");
      },
      true
   );
   this.window.removeEventListener(
      Types.focus,
      function() {
         console.log("FOCUS END");
      },
      true
   );
   this.window.removeEventListener(
      Types.blur,
      function() {
         console.log("BLUR END");
      },
      true
   );
}

findTabContainer(e: Event, map: Map<string, HTMLElement>) {
   const container: HTMLElement | null = defineParentsObject(e, node =>
      node.getAttribute(DATA_TYPE)?.includes(ControlTypes.tab)
   );

   if (container !== null) {
      tabContainer = container?.getAttribute("id");
   }
}
```

```
  findGrid(e: Event, map: Map<string, HTMLElement>) {
    const grid: HTMLElement | null = defineParentsObject(e, node =>
      node.getAttribute(DATA_TYPE)?.includes(ControlTypes.grid)
    );

    if (grid !== null && (map.has("gridToolbar") || map.has("gridCell") || map.has("quickFilter"))) {
      map.set("grid", grid);
    }
  }

  findBigButton(e: Event, map: Map<string, HTMLElement>) {
    const bigButton: HTMLElement | null =
      defineParentsObject(e, node => node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.bigButton))
||
      defineParentsObject(e, node =>
node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.bigButtonCombo));

    if (bigButton !== null) {
      map.set("bigButton", bigButton);
    }
  }

  findGridToolbar(e: Event, map: Map<string, HTMLElement>) {
    const gridToolbar: HTMLElement | null = defineParentsObject(e, node =>
      node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.gridToolbar)
    );

    if (gridToolbar !== null) {
      map.set("gridToolbar", gridToolbar);
    }
  }

  findCheckBox(e: Event, map: Map<string, HTMLElement>, type: Types) {
    const checkBox: HTMLElement | null = defineParentsObject(e, node =>
      node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.check)
    );

    if (checkBox !== null && type === Types.blur) {
      let classCheckBox = checkBox.getAttribute(CLASS_TYPE);
      if (classCheckBox?.includes("un")) {
        type = Types.unchecked;
```

```
        } else type = Types.checked;
      }
      return type;
    }

    findGridCell(e: Event, map: Map<string, HTMLElement>, type: Types) {
      let gridCell: HTMLElement | null = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.cell)
      );

      if (gridCell !== null) {
        if (type === Types.click || type === Types.dblclick) return;
        map.set("gridCell", gridCell);
        getGridCellColumnName(map, gridCell, e);
      }
    }

    findQuickFilter(e: Event, map: Map<string, HTMLElement>, type: Types) {
      const quickFilter: HTMLElement | null = defineParentsObject(
        e,
        node => node.getAttribute(CLASS_TYPE) === ClassTypes.quickFilter
      );

      const quickFilterCol: HTMLElement | null = defineParentsObject(e, node =>
node.getAttribute("col"));

      if (quickFilter !== null) {
        if (type === Types.click || type === Types.dblclick) return;
        map.set("quickFilter", quickFilter);
        if (quickFilterCol !== null) {
          getQuickFilterColumnName(map, quickFilterCol, e);
        }
      }
    }

    findModalWindow(e: Event, map: Map<string, HTMLElement>, type: Types) {
      const modalWindowButton: HTMLElement | null =
        defineParentsObject(e, node =>
node.getAttribute(ID_TYPE)?.includes(IdTypes.modalWindowOk)) ||
        defineParentsObject(e, node =>
node.getAttribute(ID_TYPE)?.includes(IdTypes.modalWindowClose));
```

```
    if (modalWindowButton !== null && type === Types.click) {
      let modalWindow: HTMLElement | null = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes("modal_window")
      );
      if (modalWindow !== null) {
        let modalWindowCaption: Element | null = getChildrenOfObject(modalWindow, "core-modal-
header-title");
        map.set("modalWindow", <HTMLElement>modalWindowCaption);
        map.set("modalWindowButton", modalWindowButton);
      }
    }
  }

  findFilter(e: Event, map: Map<string, HTMLElement>, type: Types) {
    let filterToolbar: HTMLElement | null;
    let filterCaption: HTMLElement | null;

    if (isConsolidation) {
      filterToolbar = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.consolidationFilterToolbar)
      );
      filterCaption = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.consolidationFilterCaption)
      );
    } else {
      filterToolbar = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.filterToolbar)
      );
      filterCaption = defineParentsObject(e, node =>
        node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.filterCaption)
      );
    }

    if (filterToolbar !== null) {
      if (isConsolidation) {
        map.set("filterToolbar", <HTMLElement>e.target);
      } else map.set("filterToolbar", filterToolbar);
    }

    if (filterCaption !== null && (type === Types.change || type === Types.checked || type ===
Types.unchecked)) {
      if (isConsolidation) {
```

```
            let name =
              defineParentsObject(e, node => node.getAttribute(CLASS_TYPE)?.includes("even")) ||
              defineParentsObject(e, node => node.getAttribute(CLASS_TYPE)?.includes("odd"));
            map.set("filterCaption", <HTMLElement>name);
          } else {
            let name = getChildrenOfObject(filterCaption, "panel-caption");
            map.set("filterCaption", <HTMLElement>name);
          }
        }
      }

      handleMessage(message: MessageTypeAgent) {
        const handler: any = this.handlers[message.name];
        if (!handler) {
          console.warn("Не найден обработчик для события ", message.name);
          this.removeSelectClickHandler();
          return;
        }

        handler.call(this, message.data);
      }
    }

    //Формируем объект данных от логгера CoreAnalytics для сохранения в indexedDB
    function setDataObjectFromCoreAnalyticInIndexedDB(
      map: Map<string, string | undefined | boolean>,
      type: string | undefined
    ) {
      let formN, tab, formD, isFilter, navigator: string | boolean | undefined;
      let f: boolean = false;

      if (map.has("formName")) {
        formN = map.get("formName");
        f = true;
      } else formN = formName;

      if (map.has("formDFDName")) {
        formD = map.get("formDFDName");
        f = true;
      }

      if (map.has("navigator")) {
```

```
    navigator = map.get("navigator");
    f = true;
  }

  if (map.has("tab")) {
    tab = map.get("tab");
    f = true;
  } else tab = tabActive;

  if (map.has("filter")) {
    isFilter = map.get("filter");
    f = true;
  } else isFilter = false;

  if (!f) return null;

  return {
    date: Date.now(),
    type: type,
    nameForm: formN,
    dfdNameForm: formD,
    tab: tab,
    tabContainer: tabContainer,
    isFilter: isFilter,
    navigator: navigator
  };
}

//Формируем объект данных  для сохранения в indexedDB
function setDataObjectInIndexedDB(map: Map<string, HTMLElement>, type: string, e: any) {
  let tab,
    bigButton,
    grid,
    gridToolbar,
    gridCellRow,
    gridCellCol,
    quickFilter,
    filterToolbar,
    filterCaption,
    modalWindowCaption,
    modalWindowButton,
    value: string | undefined;
```

```
let isFilter: boolean = false,
   f: boolean = false;

if (map.has("gridCell")) {
   gridCellRow = map.get("gridCell")?.getAttribute("row");
   if (map.has("gridColGrandParent")) {
      gridCellCol = map.get("gridColGrandParent")?.getAttribute("title") + '" , "';
      gridCellCol += map.get("gridColParent")?.getAttribute("title") + '" , "';
      gridCellCol += map.get("gridColName")?.getAttribute("title");
   } else if (map.has("gridColParent")) {
      gridCellCol = map.get("gridColParent")?.getAttribute("title") + '" , "';
      gridCellCol += map.get("gridColName")?.getAttribute("title");
   } else gridCellCol = map.get("gridColName")?.getAttribute("title");
   f = true;
}

if (map.has("quickFilter")) {
   if (map.has("quickFilterColGrandParent")) {
      quickFilter = map.get("quickFilterColGrandParent")?.getAttribute("title") + '" , "';
      quickFilter += map.get("quickFilterColParent")?.getAttribute("title") + '" , "';
      quickFilter += map.get("quickFilterCol")?.getAttribute("title");
   } else if (map.has("quickFilterColParent")) {
      quickFilter = map.get("quickFilterColParent")?.getAttribute("title") + '" , "';
      quickFilter += map.get("quickFilterCol")?.getAttribute("title");
   } else quickFilter = map.get("quickFilterCol")?.getAttribute("title");

   f = true;
}

if (map.has("modalWindow")) {
   modalWindowCaption = map.get("modalWindow")?.innerText;
   if (map.has("modalWindowButton")) {
      modalWindowButton = map.get("modalWindowButton")?.innerText;
      f = true;
   }
}

if (map.has("gridToolbar")) {
   gridToolbar = map.get("gridToolbar")?.getAttribute("title");
   f = true;
}
```

```
  if (map.has("bigButton")) {
    if (defineParentsObject(e, node =>
node.getAttribute(CLASS_TYPE)?.includes(ClassTypes.bigButtonScroll))) {
      bigButton = map.get("bigButton")?.innerText + " v";
    } else bigButton = map.get("bigButton")?.getAttribute("title");
    f = true;
  }

  if (map.has("grid")) {
    grid = map.get("grid")?.getAttribute("id");
    f = true;
  }

  if (map.has("filterToolbar")) {
    if (isConsolidation) {
      if (map.get("filterToolbar")?.innerHTML.includes("Применить")) filterToolbar = "Применить";
      else if (map.get("filterToolbar")?.innerHTML.includes("Очистить")) filterToolbar = "Очистить";
    } else filterToolbar = map.get("filterToolbar")?.getAttribute("title");
    isFilter = true;
    f = true;
  }

  if (map.has("filterCaption")) {
    if (isConsolidation) filterCaption = map.get("filterCaption")?.getAttribute("displayname");
    else filterCaption = map.get("filterCaption")?.innerText;
    isFilter = true;
    f = true;
  }

  if (map.has("grid") || map.has("bigButton") || map.has("filterToolbar") || map.has("filterCaption")) {
    tab = tabActive;
    f = true;
  }

  switch (type) {
    case Types.change:
    case Types.blur:
      value = e.target.value;
      if (value === "...") value = undefined;
      break;
    case Types.checked:
      value = "1";
```

```
            break;
        case Types.unchecked:
            value = "0";
            break;
        default:
            value = undefined;
    }

    if (!f) {
        return null;
    }

    return {
        date: Date.now(),
        type: type,
        tab: tab,
        tabContainer: tabContainer,
        nameForm: formName,
        bigButton: bigButton,
        grid: grid,
        gridToolbar: gridToolbar,
        gridCellRow: gridCellRow,
        gridCellCol: gridCellCol,
        quickFilter: quickFilter,
        isFilter: isFilter,
        filterToolbar: filterToolbar,
        filterCaption: filterCaption,
        modalWindow: modalWindowCaption,
        modalWindowButton: modalWindowButton,
        value: value,
        xpath: getXPath(e.target)
    };
}
```

# Приложение Б. Пример генерации кода автотеста

```
import { CodeEntity } from "../CodeEntity";
import { Types } from "../../../../agent/utils/forDefineUIComponents";

//установить введённые значения в ячейки грида
export class SetValueInGridCell extends CodeEntity {
  //получить текущий грид и метод редактирования ячейки грида
  getPrevCode(): string {
    if (this.prevRecord?.grid === undefined || this.record.grid !== this.prevRecord?.grid) {
      return [
        `Grid grid = GridUtils.getGridByTabName(form,"${this.record.tab}");`,
        `GridToolbox toolbox = grid.toolbox();`,
        `toolbox.editRow();\n`
      ].join("\n");
    }
    if (
      this.prevRecord?.gridToolbar !== "Добавить запись" &&
      this.prevRecord?.gridCellRow !== this.record.gridCellRow
    ) {
      return `toolbox.editRow();\n`;
    }
    return "";
  }

  isEntity(): boolean {
    return (
      (this.record.type === Types.change || this.record.type === Types.blur) &&
      this.record.gridCellRow !== undefined &&
      this.record.gridCellCol !== undefined &&
      this.record.value !== "" &&
      this.record.value !== undefined
    );
  }

  getCode(): string {
    return (
      this.getPrevCode() +
      `GridUtils.setColumnValueInSelectedRow("${this.record.value}", grid,
"${this.record.gridCellCol}");`
    );
  }
}
```

# Приложение В. Пример компонента подсистемы UI

```
import React, { FC } from "react";
import { useSelector } from "react-redux";
import { StoreType } from "../reducers";

const GeneratedCode: FC = () => {
  const currentSession = useSelector<StoreType, number | null>(({ SessionState }) =>
SessionState.currentSession);
  const records = useSelector<StoreType, string>(({ SessionState }) => {
    const index = currentSession === null ? SessionState.sessionData.length - 1 : currentSession;

    if (!SessionState.sessionData.length) {
      return "";
    }

    return SessionState.sessionData[index].generatedCode.join("\n");
  });

  //выделить весь текст в textarea
  const handleSelect = (e: any) => {
    e.target.select();
  };

  return (
    <div className="generation-code">
      <textarea
        className="generation-code-text syntax-highlight"
        placeholder="test-code"
        value={records}
        readOnly
        onFocus={handleSelect}
      />
    </div>
  );
};

export default GeneratedCode;
```

# Приложение Д. Пример тестирования модуля sessionAction и sessionReducer

```
import sessionReducer from "../SessionReducer";
import * as Action from "../../actions/sessionAction";
import { ISessionData } from "../../components/Session";

describe("SessionReducer", () => {
  const initialState = {
    currentSession: null,
    sessionData: [
      {
        dateOfStart: 1617971927343,
        generatedCode: [],
        log: [],
        title: "Текущая"
      }
    ],
    startSessionDate: 0,
    stopSessionDate: 0
  };

  it("Должен менять состояния", () => {
    expect(sessionReducer(initialState, Action.setCurrentSession(1))).toEqual({
      ...initialState,
      currentSession: 1
    });

    let sessionData: ISessionData[] = [];
    expect(sessionReducer(initialState, Action.setSessionData(sessionData))).toEqual({
      ...initialState,
      sessionData: []
    });

    let session: ISessionData = {
      dateOfEnd: 1617968897352,
      generatedCode: [],
      log: [],
      title: "09-04-2021 12:23:45"
    };
    expect(sessionReducer(initialState, Action.setSessionByKey(session))).toEqual({
      ...initialState,
      sessionData: [
        {
          dateOfEnd: 1617968897352,
          generatedCode: [],
```

```
            log: [],
            title: "09-04-2021 12:23:45"
          }
        ]
      });
    });
});



describe("sessionAction", function() {
  it("should return correct type", function() {
    expect(sessionAction.createSession()).toEqual({
      type: "CREATE_SESSION"
    });

    let sessionData: ISessionData[] = [];
    expect(sessionAction.setSessionData(sessionData)).toEqual({
      payload: [],
      type: "SET_SESSION_DATA"
    });

    let session: ISessionData = {
      dateOfEnd: 1617968897352,
      generatedCode: [],
      log: [],
      title: "09-04-2021 12:23:45"
    };

    expect(sessionAction.setSessionByKey(session)).toEqual({
      payload: {
        dateOfEnd: 1617968897352,
        generatedCode: [],
        log: [],
        title: "09-04-2021 12:23:45"
      },
      type: "SET_SESSION_BY_KEY"
    });

    let currentSession: number = 2;
    expect(sessionAction.setCurrentSession(currentSession)).toEqual({
      payload: 2,
      type: "SET_CURRENT_SESSION"
    });

    expect(sessionAction.setStartSession()).toEqual({
```

```
        type: "SET_START_GENERATION"
      });

      expect(sessionAction.setStopSession()).toEqual({
        type: "SET_STOP_GENERATION"
      });
    });
});
```