Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«Уральский федеральный университет

имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий - РТФ

Школа бакалавриата

ДОПУСТИТЬ К ЗАЩИТЕ ПЕРЕД ГЭК

РОП 09.03.01 Спиричева Н.Р.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

                           (подпись)

«\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_2025 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

Пояснительная записка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| Руководитель:                          \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_                                                                подпись | ФИО |
| Консультант                     \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_                                                                   подпись  Нормоконтролёр                     \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_                                                                   подпись | ФИО    ФИО |
| Студент группы РИ-410910    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_                                                    подпись | Ф.И.О\_\_\_\_\_\_\_\_\_\_ |

Екатеринбург

2025

РЕФЕРАТ

СОДЕРЖАНИЕ

[РЕФЕРАТ 2](#_Toc197719750)

[СОДЕРЖАНИЕ 3](#_Toc197719751)

[ВВЕДЕНИЕ 4](#_Toc197719752)

[1 Анализ предметной области 6](#_Toc197719753)

[1.1 Основные виды архитектур web-приложений 6](#_Toc197719754)

[1.1.1 Монолитная архитектура 6](#_Toc197719755)

[1.1.2 Архитектура Single Page Application 7](#_Toc197719756)

[1.1.3 Микросервисная архитектура 8](#_Toc197719757)

[1.2 Виды тестирования web-приложений 8](#_Toc197719758)

[1.2.1 Функциональное тестирование 8](#_Toc197719759)

[1.2.2 Нефункциональное тестирования 10](#_Toc197719760)

[1.2.3 Регрессионное тестирование 12](#_Toc197719761)

[1.3 Обзор существующих решений для тестирования 13](#_Toc197719762)

[1.3.1 Решения для тестирования пользовательских интерфейсов 13](#_Toc197719763)

[1.3.2 Решения для модульного тестирования 18](#_Toc197719764)

[1.4 Обзор современных платформ для автоматического тестирования 23](#_Toc197719765)

[2 Ход работы 26](#_Toc197719766)

[2.1 Написание сценариев тестирования 26](#_Toc197719767)

[2.2 Автоматизация тестовых сценариев 27](#_Toc197719768)

[2.3 Разработка модульных тестов 29](#_Toc197719769)

[2.4 Разработка автоматической системы для запуска тестов и формирования отчетов 30](#_Toc197719770)

[СПИСОК ИСТОЧНИКОВ 33](#_Toc197719771)

[**ПРИЛОЖЕНИЕ А** 37](#_Toc197719772)

[**ПРИЛОЖЕНИЕ Б** 39](#_Toc197719773)

[**ПРИЛОЖЕНИЕ В** 42](#_Toc197719774)

ВВЕДЕНИЕ

В современном мире цифровые технологии развиваются стремительными темпами, а веб-приложения становятся все более сложными и многофункциональными. Они включают в себя множество компонентов: динамические интерфейсы, распределенные серверные системы, интеграции со сторонними API, сложную бизнес-логику и высокие требования к безопасности и производительности. В таких условиях традиционные методы ручного тестирования оказываются недостаточно эффективными: они требуют значительных временных и трудовых затрат, а также не всегда способны обеспечить достаточное покрытие тестами.

Рост сложности веб-приложений приводит к увеличению числа потенциальных уязвимостей и ошибок, которые могут негативно сказаться на пользовательском опыте, безопасности и стабильности системы. Особенно критично это для проектов с частыми обновлениями, где изменения в одном модуле могут неожиданно повлиять на работу других компонентов. В таких условиях автоматизация тестирования становится не просто полезным инструментом, а необходимостью, позволяющей: сократить время проверки функциональности, повысить точность обнаружения ошибок, минимизировать человеческий фактор и снизить затраты на поддержку качества продукта.

Цель работы **-** разработать и интегрировать автоматизированную систему тестирования для веб-приложения, способную эффективно проверять его функциональность на разных уровнях.

Для достижения данной цели планируется выполнение следующих шагов:

1. Изучить существующие решения и подходы для автоматизации тестирования.
2. Спроектировать сценарии для тестирования пользовательского графического интерфейса.
3. Разработать программу для проверки работоспособности приложения с формированием отчета.
4. Настроить и развернуть систему автоматизированного тестирования.
5. Применить систему автоматизированного тестирования для тестирования веб-приложения на практике.

Объект исследования – процесс тестирования веб-приложений.

Предметом исследования являются методы и инструменты для автоматизированного тестирования.

1 Анализ предметной области

1.1 Основные виды архитектур web-приложений

За последние десятилетия веб-приложения очень сильно развились и прошли путь от простых веб-страниц до полноценных программных решений., обладающих сложной архитектурой и богатой функциональностью. Их ключевой особенностью является высокая степень интерактивности и возможность динамического обновления контента без необходимости полной перезагрузки страницы.

Это стало возможным благодаря развитию JavaScript-фреймворков и появлению новых веб-стандартов. Современные веб-приложения могут успешно конкурировать с их полноценными версиями по функциональности, при этом сохраняя все преимущества веб-платформы: кроссплатформенность, легкую доступность и простоту обновлений.

Архитектура веб-приложений существенно влияет на подходы к их тестированию. В современной разработке преобладают три основные модели.

1.1.1 Монолитная архитектура

Монолитная архитектура представляет собой традиционный подход к разработке веб-приложений, где все компоненты системы (интерфейс пользователя, бизнес-логика и доступ к данным) объединены в единое целое [1]. Основные характеристики:

* все части приложения разрабатываются, тестируются и развертываются как единое целое;
* простота начальной разработки и отладки;
* единая кодовая база упрощает процесс разработки на ранних этапах;
* проблемы с масштабированием при росте проекта;
* затрудненное внедрение новых технологий в устоявшуюся кодовую базу;

Типичные примеры: традиционные PHP-приложения (например, WordPress), Ruby on Rails приложения.

1.1.2 Архитектура Single Page Application

Архитектура SPA (Single Page Application) представляет собой современный подход к разработке веб-приложений, где загрузка страницы происходит один раз, а все последующие обновления контента выполняются динамически через API [2]. Основные особенности:

* клиентская часть полностью отделена от серверной;
* сервер выступает в роли поставщика данных через API (обычно REST или GraphQL);
* навигация между разделами приложения происходит без перезагрузки страницы;
* богатая интерактивность и плавность работы;
* первоначальная загрузка может занимать больше времени;

Примеры приложений: Gmail, Trello, Facebook.

1.1.3 Микросервисная архитектура

Микросервисная архитектура представляет собой подход, при котором приложение разбивается на множество небольших, слабо связанных между собой сервисов [3]. Ключевые характеристики:

* каждый сервис отвечает за определенную бизнес-функцию;
* cервисы общаются между собой через четко определенные API;
* возможность независимого масштабирования отдельных компонентов;
* разные сервисы могут использовать различные технологии;
* существенно усложняет процесс разработки и тестирования;

Примеры: Netflix, Uber, Amazon.

Тестирование веб-приложений требует комплексного подхода и включает несколько ключевых аспектов.

1.2 Виды тестирования web-приложений

1.2.1 Функциональное тестирование

Функциональное тестирование веб-приложений представляет собой систематическую проверку соответствия разрабатываемого продукта заявленным требованиям и спецификациям. Этот вид тестирования фокусируется на проверке бизнес-функций приложения и их соответствии ожиданиям конечных пользователей. Основная цель - убедиться, что все функции работают так, как было задумано разработчиками [4]. Данный вид тестирования проходит в несколько этапов:

1. Модульное тестирование представляет собой фундаментальный уровень проверки качества приложения, фокусирующийся на изолированном тестировании отдельных компонентов системы. Основная цель этого подхода заключается в верификации корректности работы минимальных логических единиц кода - функций, методов или классов [5]. Особое внимание уделяется проверке обработки граничных условий и исключительных ситуаций, что позволяет выявить потенциальные проблемы на самых ранних этапах разработки. Для обеспечения чистоты экспериментов широко применяются mock-объекты, которые заменяют реальные зависимости тестируемого модуля. Характерной чертой модульного тестирования является крайне высокая степень автоматизации, что позволяет легко интегрировать его в процесс непрерывной интеграции.
2. Интеграционное тестирование, которое переводит фокус с отдельных компонентов на их взаимодействие. Этот уровень тестирования специально разработан для выявления проблем, возникающих при объединении модулей в единую систему. Ключевыми аспектами являются проверка API-интерфейсов, анализ корректности межмодульных взаимодействий и тестирование интеграции с внешними сервисами. Особое значение имеет исследование потоков данных между различными компонентами системы, что позволяет обнаружить несоответствия в форматах данных или протоколах обмена [6]. Важной задачей интеграционного тестирования становится выявление проблем совместимости, которые часто остаются незамеченными при изолированной проверке модулей.
3. Системное тестирование представляет собой комплексную проверку всего приложения как единого целого. На этом уровне осуществляется тестирование сквозных бизнес-процессов, охватывающих всю систему от начала до конца. Основная цель - убедиться, что все компоненты, успешно прошедшие модульное и интеграционное тестирование, корректно работают в комплексе [7]. Тестирование проводится в условиях, максимально приближенных к промышленной эксплуатации, что позволяет оценить реальное поведение системы под нагрузкой. Важным аспектом является проверка соответствия системным требованиям, включая производительность, надежность и безопасность работы приложения в целом.
4. Приемочное тестирование, проводится непосредственно перед выпуском продукта в эксплуатацию. Этот уровень проверки отличается тем, что выполняется с непосредственным участием заказчика или представителей конечных пользователей. Основная задача - подтвердить, что разработанная система в полной мере соответствует всем бизнес-требованиям и ожиданиям заинтересованных сторон [7]. В ходе приемочного тестирования проводится комплексная оценка готовности решения к промышленной эксплуатации, включая анализ соответствия реального поведения системы заявленным в контракте условиям. Результаты этого тестирования становятся основанием для принятия решения о выпуске продукта или необходимости доработок.

1.2.2 Нефункциональное тестирования

Нефункциональное тестирование является критически важной частью процесса обеспечения качества веб-приложений. В отличие от функционального тестирования, которое проверяет "что делает система", нефункциональное тестирование отвечает на вопрос "как система это делает" [8]. Основные направления нефункционального тестирования:

* Тестирование производительности включает несколько ключевых аспектов. Нагрузочное тестирование позволяет определить предельные возможности системы при различных уровнях нагрузки. Оно помогает выявить "узкие места" в архитектуре приложения. Стресс-тестирование проверяет поведение системы в экстремальных условиях, превышающих нормальные рабочие нагрузки. Это важно для оценки устойчивости системы в пиковые периоды [9].
* Тестирование стабильности (так называемое "тестирование на выносливость") проводится в течение длительного периода времени для выявления таких проблем как утечки памяти или постепенная деградация производительности. Бенчмаркинг представляет собой сравнительный анализ производительности системы с конкурентными решениями или предыдущими версиями продукта [9].
* Тестирование безопасности начинается с проверки на наличие уязвимостей из списка OWASP Top 10 [10], который включает наиболее распространенные и опасные уязвимости веб-приложений. Особое внимание уделяется тестированию механизмов аутентификации и авторизации, включая проверку стойкости паролей, работу сессий и систему прав доступа.
* Тестирование удобства использования (UX/UI) включает юзабилити-тестирование с привлечением реальных пользователей, что позволяет выявить проблемы в интерфейсе, незаметные разработчикам. Проверка соответствия гайдлайнам платформ обеспечивает единообразие интерфейса и привычное для пользователей поведение элементов [9].
* Кроссплатформенное тестирование гарантирует корректную работу приложения в различных средах исполнения. Кросс-браузерное тестирование охватывает проверку работы в основных браузерах, включая Chrome, Firefox, Safari и Edge, а также их различные версии. Особое внимание уделяется мобильным браузерам и особенностям рендеринга страниц в них [11].

1.2.3 Регрессионное тестирование

Регрессионное тестирование служит критически важным механизмом обеспечения стабильности веб-приложения при внесении изменений в код. Его основная задача заключается в своевременном выявлении "регрессий" — ситуаций, когда новые правки нарушают ранее работающий функционал [12]. Особую актуальность этот вид тестирования приобретает в agile-средах с частыми релизами, где риск непреднамеренного нарушения существующей функциональности особенно высок. Современные подходы к регрессионному тестированию предполагают формирование "защитного пояса" из автоматизированных тестов, охватывающих ключевые сценарии использования приложения. Оптимальная стратегия включает как полные прогоны тестов перед крупными релизами, так и селективное тестирование затронутых функциональных областей при небольших изменениях. Эффективная организация регрессионного тестирования требует тщательного анализа рисков, грамотного приоритизации тест-кейсов и интеграции в процесс непрерывной поставки (CI/CD), что в совокупности позволяет значительно снизить вероятность появления критических дефектов.

1.3 Обзор существующих решений для тестирования

1.3.1 Решения для тестирования пользовательских интерфейсов

В последние годы тестирование пользовательских интерфейсов (UI) веб-приложений стало важным направлением в обеспечении качества разработок. Современные технологии требуют сложных подходов к проверке функциональности, совместимости и устойчивости приложений к высоким нагрузкам и динамическим изменениям. Поэтому индустрия активно использует автоматизацию тестирования с помощью специальных инструментов, каждый из которых обладает своими характеристиками, преимуществами и ограничениями. На текущий момент к ключевым управляемым разработчиками и тестировщиками технологиям относят такие как Selenium WebDriver, Cypress, Playwright и TestCafe.

Selenium WebDriver можно рассматривать как универсальный и фундаментальный инструмент для автоматизации тестирования UI. Это мощный open-source API, который позволяет осуществлять программное управление браузерами, полностью имитируя действия реального пользователя. Selenium изначально проектировался для поддержания кроссбраузерного подхода, поэтому он совместим с различными операционными системами и поддерживает широкий спектр языков программирования, таких как Java, Python, C#, JavaScript и Ruby. Основой инфраструктуры Selenium является архитектура клиент-сервер. API взаимодействует с драйверами конкретных браузеров посредством стандартизированного JSON Wire Protocol, что обеспечивает стабильную работу даже на сложных проектах с многослойной системой проверки [12].

Главным преимуществом Selenium является его гибкость: он позволяет тестировать приложения разного масштаба и структуры, начиная с простых формовых интерфейсов и заканчивая многостраничными веб-приложениями. Однако, несмотря на свою универсальность, этот инструмент требует от команды глубокого понимания работы браузерного окружения и навыков управления ожиданиями асинхронных операций. Также нужно учитывать, что Selenium не предоставляет встроенных решений для автоматического ожидания загрузки контента или стабильности элементов, что может стать вызовом для новичков.

Cypress считается более современной альтернативой Selenium благодаря своей уникальной архитектуре и подходу к выполнению тестов. Этот инструмент разработан с учетом разработчиков, что делает его особенно полезным для тех, кто ищет мощные отладочные возможности. Cypress работает непосредственно в браузерной среде, что позволяет ему перехватывать и модифицировать команды до их выполнения [13]. Такой подход обеспечивает детальный контроль над процессом тестирования, что делает сценарии предсказуемыми и удобными для анализа.

Одной из выдающихся особенностей Cypress является его интеграция с функцией "time-travel". Она позволяет разработчикам буквально «перематывать» время выполнения действий теста и изучать изменения состояния веб-приложения пошагово. Это существенно упрощает диагностику ошибок и исправление багов. Также выделяется специальная интегрированная среда автоматического управления динамическими действиями браузера, которая делает Cypress особенно полезным для тестирования продуктов, созданных на современных JavaScript-фреймворках, таких как React, Vue или Angular.

Тем не менее, Cypress не свободен от ограничений. Например, его работа ограничивается браузерами на базе Chromium, такими как Chrome и Edge, что создает проблемы с тестированием кроссбраузерной совместимости. Несмотря на это, инструмент идеально подходит для создания и выполнения стабильных тестов в современных проектах.

Разработанный в Microsoft, Playwright сочетает в себе элементы традиционных решений вроде Selenium и современных подходов, характерных для Cypress. Инструмент позволяет работать с несколькими браузерными движками, такими как Chromium, WebKit и Firefox, что делает его особенно удобным в случае тестирования приложений, ориентированных на использование в различных операционных системах и браузерах. Инструмент поддерживает сложные сценарии тестирования и предоставляет уникальную возможность работы с множественными браузерными контекстами, что полезно для эмуляции действий нескольких пользователей или работы с изолированными сессиями.

Важной особенностью Playwright является функция Codegen — автоматической записи действий пользователя в формате тестового сценария. Это заметно ускоряет процесс создания тестов, что особенно полезно для внедрения автоматизации в быстро развивающихся проектах. Также Playwright предоставляет разработчикам детализированные инструменты для работы с API, позволяя прослушивать или изменять сетевые запросы, замещать данные для тестов или эмулировать нестабильные сценарии серверного отклика.

Однако Playwright может иметь ограниченное применение для устаревших проектов, где требуется поддержка таких браузеров, как Internet Explorer [14]. Несмотря на это, его использование особенно оправдано в условиях, где необходимо обеспечить высокую достоверность результатов и стабильность кроссбраузерного тестирования.

Особенностью TestCafe является архитектура, избавленная от внешних драйверов и зависимостей от WebDriver. Решение функционирует полностью независимо, что сокращает время и энергозатраты на настройку тестовой среды. Это также делает его удобным для использования командами с небольшим опытом автоматизации тестирования. Подход TestCafe включает встроенный прокси-сервер, который автоматически перехватывает и изменяет запросы и ответы между браузером и тестируемым приложением. Таким образом, инструмент показывает стабильные результаты даже в сложных динамических условиях.

TestCafe особенно хорошо работает с инновационными SPA (single-page application) благодаря встроенной системе ожидания загрузки динамического контента. Это позволяет тестам стабильно взаимодействовать с интерфейсами даже при интенсивных фоновых операциях. Также важно отметить, что TestCafe поддерживает браузеры без необходимости настройки дополнительных плагинов или внешних расширений.

Среди возможных ограничений TestCafe можно выделить сложности с кастомизацией тестов для особых нужд, что делает инструмент менее предпочтительным для сложных архитектур или высоконагруженных приложений [15]. Тем не менее, он остается идеальным выбором для быстрого выполнения тестов и проверки функциональности небольших продукта.

Основные особенности для каждого фреймворка приведены в таблице ниже.

Таблица 1 – сравнение фреймворков

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Фреймворк | Языки программирования | Браузе-ры | Преимущества | Недостатки |
| Selenium | Java, Python, C#, JavaScript | Все | Гибкость, активное сообщество | Сложность освоения, хрупкость тестов |
| Cypress | JavaScript, TypeScript | Chrome, Edge | Простота использования встроенная отладка | Не поддерживает Firefox и Safari |
| Playwright | JavaScript, Python, Java, C# | Chrome, Firefox, WebKit | Стабильность, открытый API | Молодой проект |
| TestCafe | JavaScript, TypeScript | Все | Простая настройка, автоматические ожидания | Ограниченная кастомизация сложных тестов |

Правильный выбор инструмента для автоматизации тестирования UI зависит от множества факторов. На первых этапах следует учитывать конкретный стэк технологий, использованный в проекте, так как это напрямую повлияет на совместимость с выбранным инструментом. Например, если задействуются фреймворки нового поколения вроде React или Vue, более предпочтительным может стать Cypress или TestCafe.

Другой важный аспект — требования к кроссбраузерной поддержке, которые определяют использование Selenium или Playwright, если приложения должны быть совместимы с различными платформами. Масштаб и размер проекта также играют значимую роль, фиксируя внимание на производительности инструмента, его скорости и стабильности. Например, для крупномасштабных приложений с интенсивным пользовательским взаимодействием Playwright станет наиболее подходящим выбором, в то время как Cypress идеально подходит для тесной отладки и тестирования отдельного функционала.

Поскольку ни один инструмент не способен универсально удовлетворить все потребности, современные команды тестировщиков всё чаще комбинируют решения. Например, для одновременного охвата различных браузеров можно использовать Playwright. В то же время Cypress остается лучшим выбором для тестирования интерактивных интерфейсов и анализа сложных анимаций. Такой подход позволяет эффективно распределять ресурсы и достигать оптимальной гибкости процессов тестирования.

1.3.2 Решения для модульного тестирования

Функциональное тестирование обеспечивает проверку работы всей системы или её отдельных частей на уровне взаимодействия с пользователем или других системных модулей. Оно гарантирует, что каждый компонент выполняет свои основные функции в соответствии с техническими и пользовательскими требованиями. Однако, чтобы добиться высокого качества программного обеспечения, важно не только тестировать функциональность на уровне общего взаимодействия, но и проводить более точечные проверки. Здесь на помощь приходит модульное тестирование, главной задачей которого является проверка работы отдельных модулей или функций программы изолированно от всей системы.

В языках программирования, таких как C++, модульное тестирование играет особую роль, так как оно позволяет выявлять ошибки в базовой логике работы на ранних этапах разработки. Учитывая сложность C++ и его применение в высокопроизводительных или критически важных системах, тестирование отдельных компонентов, таких как классы, функции или небольшие модули, становится ключевым шагом в создании надежного кода

Модульное тестирование на C++ требует использования специализированных инструментов, которые упрощают процесс написания, выполнения и анализа результатов тестов. Среди наиболее популярных фреймворков для модульного тестирования в C++ выделяют Google Test (GTest), Catch2, Boost.Test и некоторые другие. Каждый из них имеет свои особенности, преимущества и подходы к организации тестов, поэтому выбор инструмента зависит от требований проекта, навыков команды и целевого функционала. Ниже подробно рассмотрены ключевые фреймворки.

Google Test – это один из самых мощных и распространённых фреймворков для модульного тестирования в C++. Разработанный и поддерживаемый Google, он стал стандартным выбором для большинства проектов, благодаря своей функциональности, гибкости и активному сообществу. Ниже представлены ключевые особенности данного фреймворка.

* Мощный API для создания тестов: GTest предоставляет понятный синтаксис для организации тестов и их запуска. Для написания тестов используется макрос TEST(TestSuiteName, TestName), который упрощает организацию набора тестов [16].
* Поддержка фикстур (fixtures): GTest позволяет задавать общее состояние для группы тестов с помощью стандартных классов фикстур (SetUp и TearDown). Это особенно полезно для тестирования кода, зависящего от одинакового начального состояния [16].
* Параметризация тестов: Вы можете писать тесты с параметрами, чтобы проверять одну и ту же функцию с различными входными данными. Это упрощает написание проверок для функций с множеством вариантов использования [16].
* Отчёты и интеграция: GTest поддерживает генерацию отчетов в формате XML, что удобно для анализа в системах непрерывной интеграции (CI). Кроме того, GTest легко интегрируется с популярными сборочными инструментами, такими как CMake [16].
* Экосистема: Google Test широко используется в реальных проектах, имеет обширную документацию и множество примеров.

Преимущества:

* простота организации тестов и полезные утилиты для проверки условий;
* активное развитие проекта и официальная поддержка со стороны Google;
* подробные отчёты об ошибках при выполнении тестов;

Недостатки:

* cтандартная библиотека GTest довольно объемна, что может усложнить использование в проектах с ограниченными ресурсами;
* начальная настройка библиотеки может быть сложной для новичков;

Catch2 – это современный, интуитивно понятный фреймворк для модульного тестирования, который активно используется разработчиками C++ благодаря простоте написания тестов и лаконичному дизайну. Он был создан как альтернатива более сложным инструментам, таким как GTest, и идеально подходит для небольших и средних проектов. Далее рассмотрены основные особенности этого фреймворка [17].

* Единственный заголовочный файл: Catch2 распространяется как header-only библиотека, что устраняет необходимость в сложной установке и минимизирует зависимость от внешних источников.
* Простой синтаксис для тестов: Тестовые кейсы создаются с помощью макроса TEST\_CASE, где можно легко описать сценарий тестирования. Такой подход снижает порог вхождения.
* Встроенные matchers: Catch2 предоставляет инструменты для проверки условий и результатов выполнения кода, упрощая процесс написания проверок.
* Поддержка BDD: Catch2 поддерживает Behavior-Driven Development (BDD), что позволяет писать тесты, ориентированные на качественное описание поведения системы.
* Кроссплатформенность: Catch2 работает на всех основных операционных системах, включая Windows, Linux и macOS.

Преимущества:

* минимальная настройка: библиотека не требует сложного создания проекта – достаточно подключить заголовок;
* удобочитаемость тестов: благодаря интуитивно понятному синтаксису, написание тестов становится простым и быстрым;
* подходит для небольших проектов, где важна скорость настройки;

Недостатки:

* может быть менее производительным на крупных тестовых наборах по сравнению с Google Test;
* поддержка параметризированных тестов менее удобна и функциональна, чем в GTest;

Boost.Test является одним из зрелых и богатых функционалом инструментов для модульного тестирования. Это часть большой Boost-библиотеки, которая включает множество модулей, полезных для разработки на C++. Ниже показаны основные особенности Boost Test:

* Интеграция с библиотекой Boost: Boost.Test идеально подходит для крупных проектов, которые уже используют Boost. Фреймворк тесно интегрирован с другими модулями этой экосистемы и обеспечивает мощную совместимость [18].
* Гибкие настройки тестов: Boost.Test предоставляет несколько уровней тестирования – от простых проверок до продвинутой конфигурации тестовых наборов для сложных приложений [18].
* Генерация отчётов: Утилита Boost.Test поддерживает производительные механизмы логирования и предоставляет подробные отчеты [18].
* Управление исключениями: Boost.Test включает мощные средства для проверки работы кода, использующего исключения, поэтому идеально подходит для тестирования сценариев, в которых важно выявить ошибки обработки исключений [18].

Преимущества:

* глубокая интеграция с частью экосистемы Boost;
* поддержка сложного тестирования, включая многопоточность;
* поддержка старых версий стандарта C++ (до C++98);

Недостатки:

* высокая сложность настройки и использования;
* устаревший синтаксис, что делает его менее интуитивным по сравнению с Catch2;
* библиотека достаточно «тяжёлая», что может увеличивать размер исполняемого файла;

Другие инструменты для тестирования C++

Doctest: Этот инструмент вдохновлён концепцией Catch2. Его ключевыми преимуществами являются высокая производительность и минималистичный синтаксис. Doctest также отлично подходит для быстрого прототипирования [19].

CppUnit: это аналог JUnit для C++. Инструмент хорош для старых проектов, однако менее активно поддерживается [20].

Фреймворки для модульного тестирования на C++ имеют свои особенности и сферы применения. Google Test выделяется своей мощью, универсальностью и практически промышленным стандартом. Catch2 привлекает лёгкостью использования и лаконичностью, что делает его отличным выбором для небольших проектов. Boost.Test подойдёт для опытных разработчиков, работающих в крупных экосистемах Boost. Выбор конкретного инструмента зависит от задач проекта, требований к тестированию, а также квалификации команды.

2. Современные платформы для автоматизированного тестирования

Автоматизированное тестирование превратилось в ключевой элемент процесса разработки программного обеспечения, позволяя повысить качество, ускорить циклы релизов и снизить затраты на поддержку продукта. С развитием технологий и ростом требований к масштабируемости и гибкости тестовых процессов появились специальные платформы, которые значительно облегчают создание, выполнение и поддержку автотестов.

Сегодня на рынке присутствует множество решений, которые предлагают широкий спектр возможностей — от простых инструментов записи и воспроизведения действий до комплексных систем с поддержкой искусственного интеллекта и интеграцией в DevOps-экосистемы. Выбор подходящей платформы часто зависит от специфики проекта, используемых технологий и структуры команды.

Далее будут рассмотрены основные категории данных платформ.

2.1 Коммерческие платформы

2.1.1 Tricentis Tosca

Tricentis Tosca — это одна из ведущих коммерческих платформ для автоматизированного тестирования, широко используемая в крупных и средних организациях. Главной особенностью Tosca является технология model-based testing (MBT), которая позволяет создавать цифровую модель приложения. Вместо написания традиционных скриптов тестировщики конструируют абстрактные модели бизнес-процессов и пользовательских сценариев, а сама платформа на их основе автоматически генерирует тесты.

Это предоставляет значительные преимущества:

* Быстрая генерация тестовых сценариев для широкого спектра функциональных требований.
* Повышенная устойчивость тестов к изменениям интерфейса, поскольку модели отражают логику работы приложения, а не детали реализации.
* Инструменты для визуального моделирования и управления тестовыми процессами, что облегчает работу нетехнических специалистов.
* Поддержка end-to-end тестирования, охватывающего пользовательские интерфейсы, API, базу данных и интеграции.
* Встроенная функциональность управления тестовыми данными и тест-кейсами, что облегчает поддержание и анализ качества тестового покрытия.

Tosca также интегрируется с популярными системами управления проектами и CI/CD, что позволяет организовать автоматизацию тестирования в рамках полного цикла разработки и доставки программного продукта. Одним из ключевых преимуществ является возможность масштабировать тестовые сценарии для крупных проектов с большим количеством функциональных модулей.

2.1.2 Micro Focus UFT One

Micro Focus UFT One (Unified Functional Testing) — платформа, изначально разработанная для профессионалов. Она предназначена для автоматизации тестирования сложных корпоративных приложений, включая веб, десктоп, мобильные и API-интерфейсы.

Основные возможности UFT One включают:

* Мощный механизм распознавания объектов, способный точно идентифицировать элементы интерфейса даже в динамичных и сложных приложениях.
* Поддержка разнообразных технологий и платформ, включая HTML5, Java, .NET, SAP, Oracle, а также тестирование баз данных.
* Встроенные инструменты для автоматизации тестирования API, баз данных и пользовательских интерфейсов в единой интегрированной среде.
* Гибкий редактор скриптов на VBScript, который позволяет создавать кастомные и детализированные тестовые сценарии с возможностью расширения и интеграции с другими инструментами.
* Интеграция с инструментами управления тестированием, такими как Micro Focus ALM (Application Lifecycle Management), что повышает эффективность организации процессов обеспечения качества.
* Поддержка автоматической регрессионной проверки, позволяющая быстро выявлять и локализовать ошибки после изменений в коде приложения.

Особенность UFT One — возможность работы с корпоративными системами со сложной бизнес-логикой, где требуется глубинное тестирование разных слоев программного продукта. Платформа широко применяется в банковской сфере, страховании, телекоммуникациях и других отраслях с высокими требованиями к качеству ПО.

2.2 Облачные платформы для автоматизированного тестирования

В отличие от традиционных коммерческих платформ, которые обычно требуют установки и настройки на локальной инфраструктуре компании, облачные решения предоставляют тестовые инструменты как сервис — полностью в облаке. Такой подход значительно упрощает запуск и масштабирование автоматизации тестирования, снижает потребность в самостоятельном администрировании и поддержке среды.

Облачные платформы позволяют пользователям создавать, запускать и анализировать тесты через веб-интерфейс, а управление инфраструктурой и обновлениями берёт на себя провайдер сервиса. Благодаря этому снижены затраты на начальное развёртывание и эксплуатацию, а также сокращено время выхода проекта на рынок.

Ниже рассмотрим два ярких представителя современных облачных решений — Mabl и Testim, которые активно используют технологии машинного обучения и искусственного интеллекта для повышения эффективности автотестирования.

2.2.1 Mabl

Mabl — это облачная платформа, предоставляющая инструмент для создания и поддержки автоматизированных тестов с помощью машинного обучения. Одна из ключевых особенностей Mabl — способность автоматически адаптировать тесты при изменениях пользовательского интерфейса. Это позволяет существенно снизить трудоемкость обслуживания тестовой базы и ускоряет процесс обновления сценариев.

Платформа объединяет функциональное, регрессионное и тестирование производительности в едином цикле, обеспечивая всесторонний подход к обеспечению качества. Mabl легко интегрируется с популярными инструментами DevOps, такими как Jenkins, Bamboo, GitHub Actions, что делает возможной их бесшовную интеграцию в CI/CD.

Кроме того, Mabl предлагает удобный интерфейс с возможностью записи действий пользователя, что подходит даже для инженеров с минимальным опытом программирования. Аналитические возможности платформы включают детальные отчёты с рекомендациями по устранению выявленных дефектов и мониторингом стабильности тестов.

2.2.2 Testim

Testim также представляет собой облачное решение, активно использующее искусственный интеллект для оптимизации процесса автоматизации тестирования. Главной особенностью Testim является интеллектуальная стабилизация тестов: система автоматически подстраивает локаторы элементов пользовательского интерфейса при изменениях в структуре страниц, что предотвращает часто встречающиеся поломки тестов.

Для Agile-команд Testim предлагает быстрый старт через запись действий пользователей, позволяя быстро создавать первоначальные тесты. По мере повышения квалификации инженеры могут расширять и совершенствовать тесты, используя встроенный редактор кода. Такое комбинирование упрощает переход от базовой автоматизации к более продвинутому уровню.

Кроме того, Testim поддерживает выполнение регрессионного и функционального тестирования, что обеспечивает гибкость в покрытии различных аспектов качества ПО. Система интегрируется с основными системами управления проектами и CI/CD платформами, обеспечивая автоматическое выполнение тестов с анализом результатов и уведомлениями.

2.3 Решения с открытым исходным кодом

В корпоративной сфере всё большую популярность приобретают решения с открытым исходным кодом, которые сочетают гибкость кастомизации с экономической эффективностью. Такие инструменты позволяют компаниям создавать масштабируемые и настраиваемые тестовые окружения без необходимости больших затрат на лицензии, при этом обеспечивая профессиональный уровень автоматизации. Далее рассмотрим два хорошо зарекомендовавших себя представителя — Apache JMeter и Galen Framework.

2.3.1 Apache JMeter

Apache JMeter — это многофункциональный инструмент, первоначально разработанный для нагрузочного тестирования, но со временем получивший богатые возможности для функционального и API-тестирования. Его популярность в корпоративной среде объясняется расширяемостью и масштабируемостью.

Ключевые возможности JMeter включают:

* Нагрузочное тестирование: моделирование большого количества виртуальных пользователей для проверки производительности и устойчивости приложений.
* Функциональное тестирование: проверка бизнес-логики через сценарии для веб-приложений, API, баз данных и других сервисов.
* Тестирование API: поддержка HTTP, REST, SOAP и других протоколов с возможностью создания сложных сценариев запросов и обработки ответов.
* Широкий набор плагинов: расширяют функционал и позволяют интегрировать JMeter практически с любым инструментом DevOps и управления тестами.
* Гибкие средства создания отчетности и анализа: визуализация результатов помогает быстро выявлять узкие места и сбои.

JMeter запускается на Java, что гарантирует совместимость с разнообразными платформами. Благодаря открытой архитектуре корпоративные команды могут адаптировать инструмент под свои задачи и создавать собственные расширения.

2.3.2 Galen Framework

Galen Framework — специализированный инструмент для тестирования адаптивного и кросс-браузерного пользовательского интерфейса с акцентом на качество верстки. Он особенно полезен для проектов с жёсткими требованиями к внешнему виду интерфейса и точному соответствию дизайн-макетам.

Основные особенности Galen:

* Проверка расположения элементов и визуальных стилей: позволяет определить отклонения от заданных правил и макетов на различных разрешениях экранов.
* Использование простого языка описания спецификаций (.spec): тестировщики описывают расположение, размеры и другие характеристики элементов, включая условия адаптивности.
* Поддержка разнообразных браузеров и платформ: позволяет выполнять тесты на реальных и виртуальных устройствах, интегрируется с Selenium для реализации автоматических сценариев.
* Возможность интеграции с CI/CD: обеспечивает автоматическую проверку верстки при каждом коммите или сборке.
* Отчетность с визуальными доказательствами: включают скриншоты, показывающие найденные несоответствия и ошибки верстки.

Galen отлично подходит для QA-команд, которые уделяют повышенное внимание визуальному качеству и пользовательскому опыту, обеспечивая уверенность в корректном отображении интерфейсов на разных устройствах.

Несмотря на очевидные преимущества и функциональную насыщенность существующих коммерческих, облачных и open-source платформ для автоматизированного тестирования, в некоторых проектах и организациях возникает необходимость создания собственной, кастомизированной системы. Это решение основывается на желании максимально адаптировать тестовую инфраструктуру под специфические требования бизнеса, технические особенности проекта и процессы разработки.

Одной из причин является необходимость полной прозрачности и контроля над всем жизненным циклом тестов — от их написания до исполнения и анализа результатов. Использование готовых платформ часто ограничивает возможности кастомизации и накладывает зависимости от внешних поставщиков, что в долгосрочной перспективе может влиять на гибкость и стоимость владения.

В данном проекте основой стали зрелые и проверенные инструменты с открытым исходным кодом, а именно Selenium для функционального тестирования пользовательских интерфейсов, Boost Test для написания модульных и интеграционных тестов на уровне кода, а также Gitea Actions как легковесный CI/CD-процессор для автоматизации запуска и управления тестовыми конвейерами. Этот набор инструментов позволяет построить платформу, которая сочетает в себе гибкость, масштабируемость и удобство интеграции в существующие DevOps-процессы.

Выбор данных компонентов продиктован их широким признанием, стабильностью и возможностью свободно модифицировать и расширять их функциональность под конкретные задачи проекта. Такой подход обеспечивает независимость от коммерческих решений и дает возможность с высокой точностью соответствовать внутренним стандартам качества и требованиям безопасности.

Таким образом, несмотря на наличие мощных и функциональных решений на рынке, создание собственной платформы позволяет максимально адаптировать инструменты автоматизации под уникальные потребности и особенности организации, обеспечивая тем самым более эффективный и управляемый процесс тестирования.

2 Ход работы

2.1 Написание сценариев тестирования

Первым этапом работы являлось создание сценариев для тестирования приложения. Для этого потребовалось изучить функциональные возможности самого приложения, его архитектуру и взаимодействие между компонентами. Также были проанализированы технические требования и ожидаемое поведение системы, что позволило сформировать корректные и релевантные тестовые случаи.

На основе проведённого анализа были разработаны сценарии тестирования, охватывающие как позитивные (проверка работы приложения в штатных условиях), так и негативные (обработка ошибок и нестандартные ситуации) сценарии использования. Каждый тестовый сценарий включал:

* описание тестируемой функции;
* шаги для воспроизведения;
* ожидаемый результат;
* критерии успешного прохождения теста.

Этот этап позволил систематизировать процесс тестирования и обеспечить максимальное покрытие требований, что в дальнейшем способствовало выявлению потенциальных уязвимостей и ошибок в работе приложения.

Пример тестового сценария приведен в приложении А.

2.2 Автоматизация тестовых сценариев

Для реализации данной задачи был выбран комплекс инструментов, включающий:

* pytest - как основной фреймворк для организации и выполнения тестов;
* Selenium WebDriver - для автоматизации взаимодействия с пользовательским интерфейсом тестируемого приложения.

Процедура автоматизации включала последовательное выполнение следующих работ:

1. Подготовка тестового окружения:
   * установка и конфигурация программного обеспечения;
   * настройка драйверов для взаимодействия с веб-браузером;
   * создание структуры проекта с выделением модулей для тестов, вспомогательных функций и конфигурационных файлов.

2. Разработка автоматизированных тестов:

* + трансформация ручных тестовых сценариев в программный код с использованием возможностей pytest;
  + реализация фикстур для управления жизненным циклом тестовых данных и состояний системы;
  + организация модульной структуры тестов в соответствии с принципами поддержки и повторного использования кода.

3. Интеграция Selenium WebDriver:

* + разработка скриптов для автоматизированного взаимодействия с элементами пользовательского интерфейса,
  + реализация механизмов явных и неявных ожиданий для обеспечения стабильности выполнения тестов,
  + внедрение системы проверок (assert) для верификации соответствия фактических результатов ожидаемым.

Пример реализации автотеста приведён в приложении Б.

Внедрение автоматизированного тестирования позволило достичь следующих результатов:

* сокращение временных затрат на проведение регрессионного тестирования;
* повышение точности проверок за счёт исключения человеческого фактора;
* обеспечение возможности интеграции в процессы непрерывной поставки.

2.3 Разработка модульных тестов

Следующим этапом работы стала разработка модульных тестов для проверки корректности работы отдельных компонентов приложения**.**  Для реализации данной задачи был применен комплекс инструментов, включающий:

* Boost.Test - как основной фреймворк для организации модульного тестирования на C++,
* FakeIt - библиотеку для создания mock-объектов и заглушек (stubs),
* дополнительные инструменты CMake для интеграции тестов в процесс сборки.

Работа по внедрению модульного тестирования включала следующие этапы:

1. Проектирование тестовой инфраструктуры.
2. Настройка тестового окружения.
3. Создание заглушек и mock-объектов.
4. Реализация тестовых случаев.

Пример тестового случая приведён в приложении В.

Реализация модульных тестов позволила достичь следующих результатов:

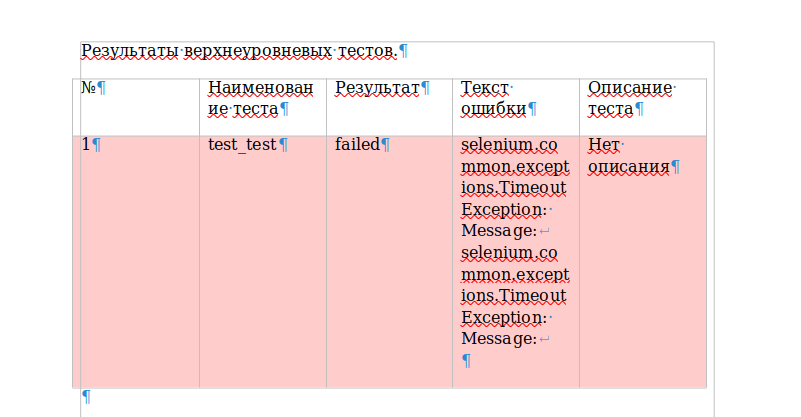
* повышение надежности отдельных компонентов приложения,
* раннее выявление ошибок на этапе разработки,
* упрощение рефакторинга за счет наличия тестовой базы.

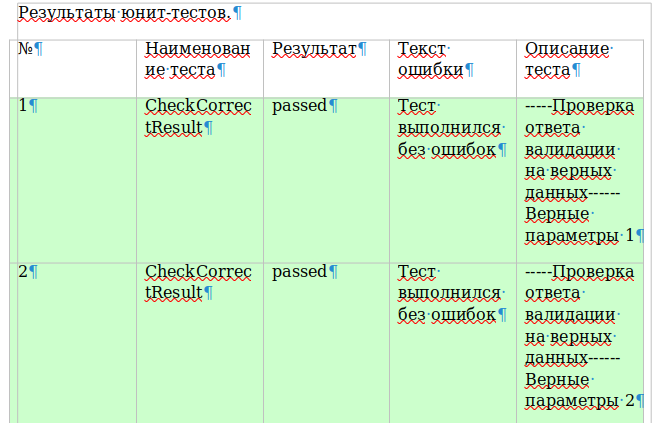
2.4 Разработка автоматической системы для запуска тестов и формирования отчетов

Следующим этапом работы стала разработка автоматизированной системы запуска тестов и формирования отчетов**.**  Данная система была реализована в виде специализированного скрипта на языке Python и обеспечивает последовательное выполнение следующих операций:

1. Запуск модульных тестов:
   * автоматический запуск тестовых наборов, реализованных на Boost.Test;
   * обработка кодов возврата для определения успешности выполнения.
2. Парсинг результатов модульного тестирования:
   * анализ сгенерированного XML-файла с использованием библиотеки xml.etree.ElementTree;
   * извлечение ключевых параметров.
3. Формирование отчета:
   * создание документа в формате DOCX с использованием библиотеки python-docx;
   * структурирование информации.
4. Запуск системных тестов и дополнение отчета:
   * автоматический запуск тестов Selenium через pytest;
   * захват и обработка выходных данных;
   * дополнение отчета результатами каждого теста.

Пример сформированного отчета представлен на рис.1 и 2.

Рисунок 1 – Пример отчета для верхнеуровневых тестов

Рисунок 2 – Пример отчета для юнит-тестов

СПИСОК ИСТОЧНИКОВ

1. Купер А. Об интерфейсе. Основы проектирования взаимодействия. – 4-е изд. – Санкт-Петербург : Символ-Плюс, 2019. – 688 с.
2. Scott E. Jr. SPA Design and Architecture: Understanding single-page web applications. – Simon and Schuster, 2015. – 256 p.
3. Мычко С. И. Микросервисная архитектура // Информационные технологии. – 2019. – С. 166–168.
4. Beizer B. Black-box testing: techniques for functional testing of software and systems. – John Wiley & Sons, Inc., 1995. – 320 p.
5. Hamill P. Unit test frameworks: tools for high-quality software development. – O'Reilly Media, Inc., 2004. – 232 p.
6. Orso A. Integration testing of object-oriented software // Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano. – 1998. – P. 1–119.
7. Di Lucca G. A., Fasolino A. R., Tramontana P. Testing web applications // International Conference on Software Maintenance, 2002. Proceedings. – IEEE, 2002. – P. 310–319.
8. Сеидова И. Э., Абдуллаев Э. М. Нефункциональное тестирование программного обеспечения: особенности и применение // In The World Of Science and Education. – 2025. – № 15 (март). – С. 60–63.
9. OWASP Foundation. OWASP Top Ten: Critical Web Application Security Risks. – 2021. – URL: https://owasp.org/www-project-top-ten/ (дата обращения: 10.04.2025).
10. Roy Choudhary S. Cross-platform testing and maintenance of web and mobile applications // Companion Proceedings of the 36th International Conference on Software Engineering. – 2014. – P. 642–645.
11. Котляров В. П. Основы тестирования программного обеспечения. – Москва, 2016. – 320 с.
12. BrowserStack. Selenium Testing Tool. – URL: https://www.browserstack.com/selenium (дата обращения: 20.04.2025).
13. Cypress: Fast, easy and reliable testing for anything that runs in a browser. – URL: https://www.cypress.io/ (дата обращения: 22.04.2025).
14. Microsoft. Playwright: Reliable end-to-end testing for modern web apps. – URL: https://learn.microsoft.com/en-us/microsoft-edge/playwright/ (дата обращения: 25.04.2025).
15. BrowserStack. TestCafe Framework Tutorial. – URL: https://www.browserstack.com/guide/testcafe-framework-tutorial (дата обращения: 26.04.2025).
16. Google. GoogleTest: C++ Testing Framework. – URL: https://google.github.io/googletest/ (дата обращения: 27.04.2025).
17. GitLab. Develop C++ Unit Testing with Catch2, JUnit, and GitLab CI. – 2024. – URL: https://about.gitlab.com/blog/2024/07/02/develop-c-unit-testing-with-catch2-junit-and-gitlab-ci/ (дата обращения: 28.04.2025).
18. Boost C++ Libraries. Boost.Test Documentation. – URL: https://www.boost.org/doc/libs/1\_82\_0/libs/test/doc/html/index.html (дата обращения: 28.04.2025).
19. JetBrains. Better Ways of Testing with doctest. – 2019. – URL: https://blog.jetbrains.com/rscpp/2019/07/10/better-ways-testing-with-doctest/ (дата обращения: 28.04.2025).
20. Freedesktop.org. CppUnit: C++ Unit Testing Framework. – URL: https://www.freedesktop.org/wiki/Software/cppunit/ (дата обращения: 28.04.2025).
21. Tested & Failed by Tricentis. – URL: https://testedfailed.tricentis.com/ (дата обращения: 28.04.2025).
22. Micro Focus. UFT One: Data Sheet [PDF]. – URL: https://www.microfocus.com/ru-ru/media/data-sheet/uft-one-ds-ru.pdf?utm\_source=OSPRU (дата обращения: 28.04.2025).
23. Mabl: Low-code test automation. – URL: https://www.mabl.com/ (дата обращения: 29.04.2025).
24. testIT. Integrations with Testim. – URL: https://docs.testit.software/user-guide/integrations/automation/testim.html (дата обращения: 29.04.2025).
25. The Apache Software Foundation. Apache JMeter. – URL: https://jmeter.apache.org/ (дата обращения: 30.04.2025).
26. Хабрахабр. Тестирование производительности: JMeter vs Gatling vs Tsung. – 2015. – URL: https://habr.com/ru/articles/272213/ (дата обращения: 30.04.2025).
27. GitLab. The DevSecOps Platform. – URL: https://about.gitlab.com/ (дата обращения: 30.04.2025).
28. Хабрахабр. Microsoft: Тестирование в облаке. – 2018. – URL: https://habr.com/ru/companies/microsoft/articles/423367/ (дата обращения: 01.05.2025).
29. Sauce Labs: Continuous Testing Cloud. – URL: https://saucelabs.com/ (дата обращения: 01.05.2025).
30. BrowserStack: Real Device Cloud for Testing. – URL: https://www.browserstack.com/ (дата обращения: 01.05.2025).

**ПРИЛОЖЕНИЕ А**

**(справочное)**

**Пример тестового сценария**

Описание функции: режим замера сопротивления для соединителей.

1. Перейти во вкладку "Работа".
2. Перейти во вкладку "Режимы испытаний".
3. Перейти во вкладку с необходимым режимом.
4. Выбрать все элементы для проверки.
5. Нажать кнопку “Начать проверку”.
6. Подтвердить начало проверки.
7. Проверить, что кнопка "Начать проверку" заблокирована.
8. Проверить, что кнопка "выход" изменилась на кнопку "отмена".
9. Дождаться окончания проверки.
10. Проверить правильность сформированного протокола.
11. Проверить, что по всем элементам получен результат.
12. Выйти из режима.
13. Проверить, что во вкладке диагностика, есть запись об окончании проверки.

Ожидаемый результат: режим корректно отрабатывает, по всем выбранным модулям приходит ответ.

Критерии успешного прохождения теста:

* Во время проверки все необходимые кнопки блокируются
* По всем модулям был получен ответ
* После окончания проверки протокол правильно сформирован
* - Во вкладке “Диагностика” есть запись с результатом проверки.

**ПРИЛОЖЕНИЕ Б**

**(справочное)**

**Пример модульного теста**

BOOST\_AUTO\_TEST\_CASE(TestCheckResistPositiveAnswer){

BOOST\_TEST\_MESSAGE("Проверка режима");

Json::Value state;

state["Operable"] = 15;

state["Available"] = 0;

Mock<Request> state\_mock;

When(Method(state\_mock, get\_result)).Return(&state);

When(Method(state\_mock, run)).Return();

Mock<RequestStateFactory> state\_factory\_mock;

When(Method(state\_factory\_mock, createRequestState)).Return(dynamic\_cast<RequestStateTask\*>(&state\_mock.get()));

IRequestStateFactory\* state\_factory = &state\_factory\_mock.get();

RequestStateTask\* state\_ptr = dynamic\_cast<RequestStateTask\*>(&state\_mock.get());;

RequestStateTask\* state\_request = state\_ptr;

state\_ptr -> run();

Json::Value\* value = (\*state\_request).get\_result();

Mock<Request> resist\_mock;

When(Method(resist\_mock, get\_result)).Return(&resist);

**Продолжение ПРИЛОЖЕНИЯ Б**

Mock<IRequestResistFactory> resist\_factory\_mock;

std::unique\_ptr<RequestResistTask> unique\_ptr\_resist((RequestResistTask \*)&resist\_mock.get());

When(Method(resist\_factory\_mock, createRequestResist)).Return(std::move(unique\_ptr\_resist));

RequestResistFactory\* resist\_factory = (RequestResistFactory\* )&resist\_factory\_mock.get();

std::string path\_file = "../storage/mode.csv";

iteration = 0;

Json::Value json;

Json::Value\* json\_ptr = &json;

CsvParsTask parser = CsvParsTask(&iteration, &path\_file, &data, json\_ptr);

parser.run();

iteration = 0;

Json::Value input\_message;

input\_message["list"][0] = "0";

input\_message["list"][1] = "1";

input\_message["list"][2] = "2";

input\_message["list"][3] = "3";

std::cout << "\n 4 \n";

for (int i=0; i<4; i++){

ConnectorsChek chek\_object = ConnectorsChek(&input\_message, &iteration, &data, json\_ptr,

**Продолжение ПРИЛОЖЕНИЯ Б**

(RequestResistFactory\*)&resist\_factory\_mock.get(), (RequestStateFactory\*)state\_factory);

chek\_object.run();

std::cout << "1 \n";

std::cout << ((\*json\_ptr)["array"][i]["module"]["res\_task"].asString());

BOOST\_CHECK\_EQUAL((\*json\_ptr)["array"][i]["module"]["res\_task"].asString(), "positive");

iteration++;

}

}

**ПРИЛОЖЕНИЕ В**

**(справочное)**

**Пример системного теста**

def check\_mode(web\_driver\_mode, mode):

web\_driver\_mode.find\_element(By.ID, mode).click()

try:

WebDriverWait(web\_driver\_mode, 3).until(lambda d:d.find\_element(By.TAG\_NAME, "span").is\_displayed())

circuits\_background\_color = web\_driver\_mode.find\_element(By.TAG\_NAME, "span").value\_of\_css\_property("background-color")

except Exception:

web\_driver\_mode.execute\_script("arguments[0].click()", web\_driver\_mode.find\_element(By.XPATH, "//button[text()='выход']"))

time.sleep(1)

web\_driver\_mode.find\_element(By.ID, mode).click()

WebDriverWait(web\_driver\_mode, 3).until(lambda d:d.find\_element(By.TAG\_NAME, "span").is\_displayed())

circuits\_background\_color = web\_driver\_mode.find\_element(By.TAG\_NAME, "span").value\_of\_css\_property("background-color")

check\_boxes = web\_driver\_mode.find\_elements(By.TAG\_NAME, "label")

for i in range(len(check\_boxes) - 1):

check\_boxes[i].click()

**Продолжение ПРИЛОЖЕНИЯ В**

web\_driver\_mode.find\_element(By.XPATH, "//button[text()='Да']").click()

WebDriverWait(web\_driver\_mode, 100).until(lambda d:d.find\_element(By.XPATH, "//button[text()='Ok']").is\_displayed())

report = web\_driver\_mode.find\_element(By.TAG\_NAME, "p").text

result = report[report.find("Результат")+10:] == 'отрицательный'

web\_driver\_mode.find\_element(By.XPATH, "//button[text()='Ok']").click()

web\_driver\_mode.find\_element(By.XPATH, "//button[text()='Выход']").click()

circuits\_states = web\_driver\_mode.find\_elements(By.TAG\_NAME, "span")

is\_broken = any([state.value\_of\_css\_property("background-color") == 'rgb(250, 0, 33)' for state in circuits\_states])

assert result == is\_broken, "Неправильно сформирован протокол"

assert all([state.value\_of\_css\_property("background-color") != circuits\_background\_color for state in circuits\_states]), "По одному или нескольким модулям не было получено ответа"