

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.02 «Информационные системы и технологии»

БАКАЛАВРСКАЯ РАБОТА

на тему

**Система автоматизированного тестирования
приложения с текстовым псевдографическим
интерфейсом пользователя**

Студент _____ Костин Денис Александрович
(подпись, дата) (ФИО полностью)

Руководитель _____ Иванчуков А.Г.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой _____ Козлов А. Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
А.Ю. Козлов «
20 » декабря 2024 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Костин Денис Александрович гр. 21ВИ1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Система автоматизированного тестирования приложения с
текстовым псевдографическим интерфейсом пользователя

Тема утверждена приказом ПГУ № _____ от « _____ » _____ 20__ г.

3. Руководитель работы доцент кафедры «МОиПЭВМ» кандидат технических наук
Иванчуков А.Г.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Разработать систему, обеспечивающую автоматизированное тестирование
приложения с псевдографическим интерфейсом пользователя.

2) Среда выполнения тестирования: ОС Linux Debian 12

Среда выполнения развертывания тестирования: ОС Windows 10

Языковые средства разработки: Python, Bash, PowerShell

Инструментальные средства разработки: JetBrains PyCharm, Visual Studio Code

Технологии разработки: PyTest, Allure, PyAutoGui, pytesteract, Selenium,

PageObject, AAA, VmWare

3) Основные функции системы тестирования:

а) разворачивание тестового окружения;

б) запуск тестирования на тестовом окружении;

в) прохождение тестов графического интерфейса;

г) прохождение тестов web - интерфейса;

д) фиксация результата тестов и создание отчета.

5. Перечень подлежащих разработке вопросов

1) Анализ предметной области и постановка задачи

2) Проектирование

3) Реализация

4) Контроль качества программного обеспечения

5) Оформление пояснительной записки

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области постановка задачи	10%	08.04.2025– 14.04.2025	
2	Проектирование	20%	15.04.2025– 25.04.2025	
3	Реализация	30%	26.04.2025– 12.05.2025	
4	Контроль качества программного обеспечения	20%	13.05.2025– 22.05.2025	
5	Оформление пояснительной записки	20%	23.05.2025– 03.06.2025	

Дата выдачи задания "20" декабря 2025 г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить

Декан факультета _____
(подпись, дата)

Реферат

Пояснительная записка содержит 85 листов, 25 рисунков, 1 таблицу, 24 использованных источников, 24 листинга, 4 приложения.

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ, СИСТЕМА ТЕСТИРОВАНИЯ, PYTHON, PYTEST, PYAUTOGUI, ALLURE, POWERSHELL, VMWARE

Объектом разработки является система автоматизированного тестирования приложения с текстовым псевдографическим интерфейсом пользователя

Цель работы – разработка системы, позволяющей автоматизировать тестирование приложения с текстовым псевдографическим пользовательским интерфейсом, а также создающее подробные отчеты о работоспособности тестируемого приложения.

Разработка автоматизации тестирования велась на языке программирования Python с использованием фреймворка тестирования PyTest и фреймворка формирования отчета Allure.

Разработка автоматизации развертывания и управления тестовым стендом велась с использованием языка программирования PowerShell и утилиты управления виртуальными машинами vmrun.

Система предназначена для работы на устройствах под управление ОС Windows 10. Автоматизированное выполнение тестирования предназначено для работы на устройствах под управлением ОС Linux Debian 12.

					ПГУ 09.03.02 – 08BP211.18 ПЗ			
Изм.	Лист	№ докум.	Подп.	Дата				
Разраб.		Костин Д.А.			Разработка информационной системы идентификации пользователей с применением искусственного интеллекта. Пояснительная записка	Лит.	Лист	Листов
Пров.		Иванчуков А.Г.					4	85
Реценз.						Группа 21ВИ1		
Н. контр.		Такташкин Д.В.						
Утверд.		Козлов А.Ю.						

Содержание

Введение.....	6
1 Анализ предметной области	7
1.1 Анализ способов взаимодействия пользователя с ПО	9
1.2 Особенности тестового стенда.....	11
2 Постановка задачи	12
3 Выбор инструментов разработки	13
3.1 Выбор языка программирования	13
3.2 Выбор фреймворка тестирования	15
3.3 Выбор фреймворка формирования отчета.....	20
3.4 Выбор инструментов управления приложением.....	25
3.4.1 Взаимодействие с экраном локального администрирования	25
3.4.2 Взаимодействие с интерфейсом удаленного администрирования ...	29
4 Проектирование	33
4.1 Проектирование системы тестирования	33
4.2 Структура проекта автоматизированных тестов.....	36
4.3 Проектирование автоматизированного разворачивания стенда тестирования.....	39
5 Реализация проекта автотестирования	41
5.1 Базовый функционал	41
5.2 Паттерн PageObject.....	49
5.3 Паттерн тестирования AAA	48
6 Тестирование приложения.....	52
Заключение	56
Список использованных источников	57
Приложение А	59
Приложение Б.....	61
Приложение В.....	70
Приложение Г	76

Введение

Современное программное обеспечение (далее – «ПО») обладает обширным функционалом, доступным пользователю. Так же актуальный процесс разработки ПО требует быстрых релизов новых опций и функций. Нельзя забывать о надежности работы текущего функционала и добавляемых функций. Проверкой каждой новой итерации разработки ПО занимаются инженеры по тестированию. Но зачастую ручное тестирование может становиться «узким горлышком» в процессе разработки ПО, это может быть вызвано:

- нехваткой времени на полную проверку функционала;
- пропуском ошибок в ПО из-за человеческого фактора;
- неполным пониманием каждого отдельного инженера по тестированию всего функционала ПО.

Автоматизированное тестирование призвано для решения этих проблем разработки ПО[1]. Оно позволяет разгрузить команду тестирования, взяв на себя такие задачи, как:

- рутинные проверки базового функционала ПО;
- развертывание стенда для отладки новой итерации ПО.

Тем самым позволяя тестировщикам сконцентрироваться на тестирование более сложных бизнес – сценариев ПО. Авто-тестирование является неотъемлемой частью современного процесса разработки ПО.

Целью данной выпускной работы будет разработать систему автоматизированного тестирования для приложения с псевдографическим интерфейсом, которая позволит выполнять проверку функционала новых итераций разработки приложения (далее – «сборок»), а так – же формировать подробный и понятный отчет о пройденных тестовых сценариях и о возможно возникших ошибках.

1 Анализ предметной области

Автоматизированное тестирование – это метод тестирования программного обеспечения, который предполагает использование инструментов и фреймворков автоматизации для выполнения одного и того же набора тест-кейсов снова и снова[1]. Ключевое различие между ручным и автоматизированным тестированием заключается в том, что ручное тестирование полностью зависит от человека, выполняющего набор действий. В то время как автоматизированные тесты могут быть написаны один раз и выполняться многократно практически без участия человека[2].

Существует два основных направления автоматизированного тестирования: тестирование на уровне кодовой базы и тестирование пользовательского интерфейса (оно же GUI – тестирование).

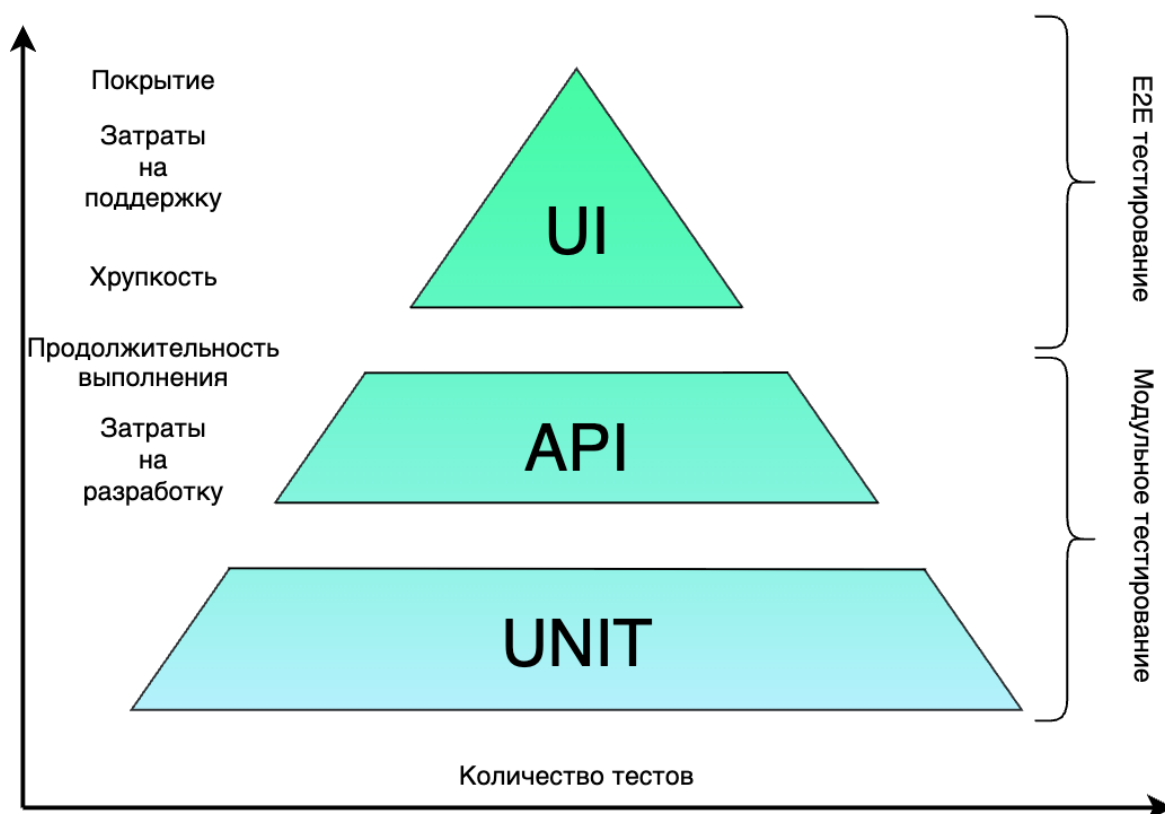


Рисунок 1 – Схема пирамиды тестирования

К первому типу относится модульное тестирование, так же называемое блочным или модульным тестированием. Этот подход позволяет проверить корректность работы отдельных модулей исходного кода программного обеспечения, а так-же их взаимодействие между собой. Внедрение этого подхода увеличивает скорость обнаружения ошибок, возникающих при добавлении новых модулей и редактировании уже написанного функционала. Подобные тесты в основном пишут сами разработчики ПО, так как они наиболее полно знакомы с самой кодовой базой. Данный подход не может полностью отловить все ошибки, возникающие при разработке из-за наличия таких ограничений как: изолированность среды тестирования и необходимость использования объектов – заглушек.

В дополнение к модульному тестированию используют автоматизированное GUI – тестирование, так – же называемое приемочным или end – to – end (E2E) тестированием. Особенности этого подхода является то, что приложение тестируют тем же способом, как и пользователи будут взаимодействовать с ним, а так – же то, что тестирование происходит методом «черного ящика», то есть без доступа к исходному коду. Это позволяет выделить отдельную команду ручных тестировщиков и автоматизаторов тестирования, которым нет необходимости разбираться в нюансах работы кодовой базы. В данной работе будет рассматриваться разработка системы автоматизации тестирования, основанная на подходе симуляции поведения пользователя через графический интерфейс приложения.

Объектом разработки является автоматизация процесса тестирования информационной системы электронного документооборота. В данном случае разрабатываемым продуктом является информационная система с псевдографическим пользовательским интерфейсом.

При проведении анализа предметной области тестирования ПО важно выделить:

- способы взаимодействия пользователя с ПО;
- доступный функционал тестируемого ПО;

– особенности тестового стенда.

В данном случае пользователи взаимодействуют с ПО через псевдографический экран сервера. Для удобства тестирования был разработан эмулятор этого экрана, доступный на тестовой сборке ПО. Так – же есть web–интерфейс администрирования, доступ к которому можно получить благодаря удаленному подключению через прокси туннель.

Целевая задача разрабатываемой системы – автоматизировать процесс тестирования программных средств электронной подписи.

Стоит отдельно выделить тестовую среду. В ее состав входят виртуальные машины с предустановленной операционной системой (Debian 12), программное обеспечение целевой информационной системы, а также средства автоматизации развертывания стенда.

1.1 Анализ способов взаимодействия пользователя с ПО

Известны два способа взаимодействия с целевой информационной системой. Первый способ – напрямую с физическим сервером через сенсорный псевдографический экран, представляющий собой окно 320 на 280 пикселей. Этот способ можно называть «локальным администрированием».

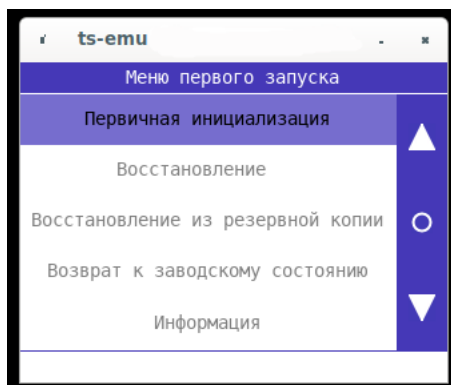


Рисунок 2 – Скриншот псевдографического экрана локального администрирования

Дополнительным способом взаимодействия является web–приложение удаленного администрирования в браузере. Это «дистанционное администрирование».

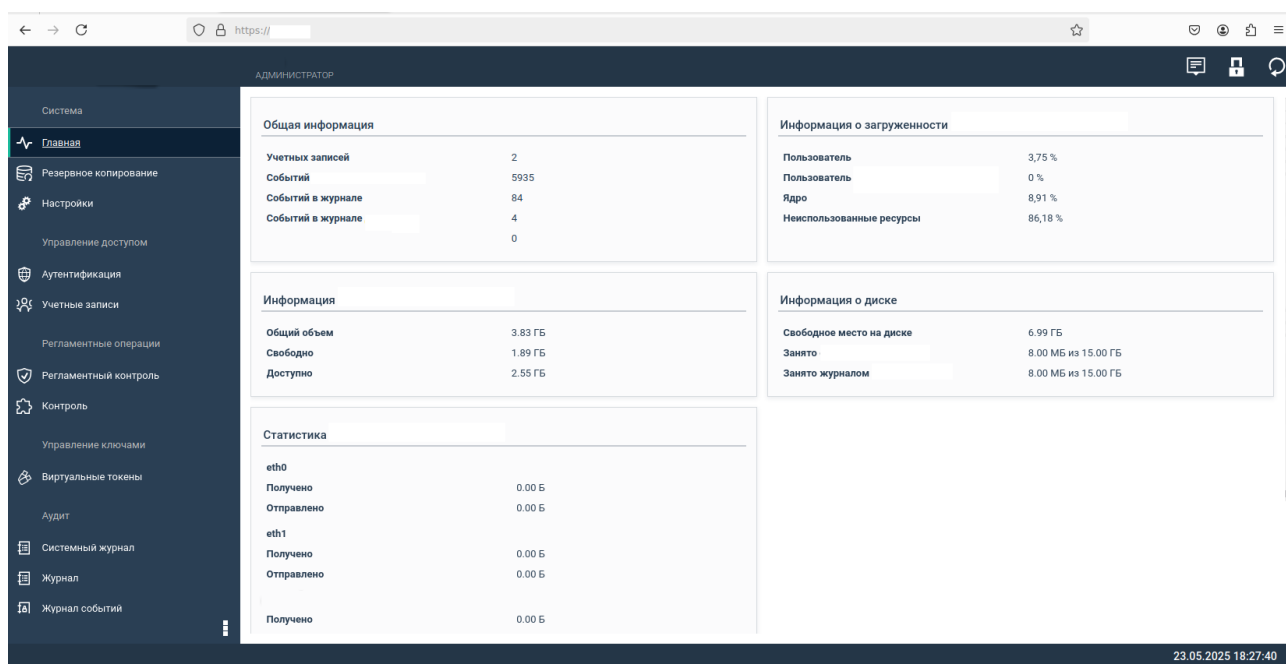


Рисунок 3 – Браузерный веб клиент дистанционного администрирования

Так как тестирование ПО происходит в виртуальной среде под управлением Debian 12, используется эмулятор локального экрана сервера, тестирующий может взаимодействовать с ним исключительно нажатиями курсором мыши в окне этого эмулятора.

При тестировании дистанционного администрирования, тестирующему доступны все стандартные сценарии взаимодействия с web – приложением, а именно:

- нажатия курсором мыши по интерактивным элементам;
- ввод данных с клавиатуры;
- прокручивание web – страницы колесом мыши и ползунком на web – странице;
- обновление страницы;
- открытие новых вкладок в браузера;
- переключение между вкладками браузера.

1.2 Особенности тестового стенда

Каждая новая итерация разработки ПО собирается средствами автоматической сборки на TeamCity и поставляется в виде дистрибутива со всеми службами разрабатываемого ПО в готовом состоянии. Этот образ дистрибутива устанавливается тестировщиком на виртуальную машину (или на железный сервер) для последующего тестирования.

Так как разрабатываемое ПО предоставляет возможность дистанционного администрирования, тестировщику требуется дополнительная виртуальная машина с полноценным Linux, графическим окружением и web – браузером, для проверок этого функционала.

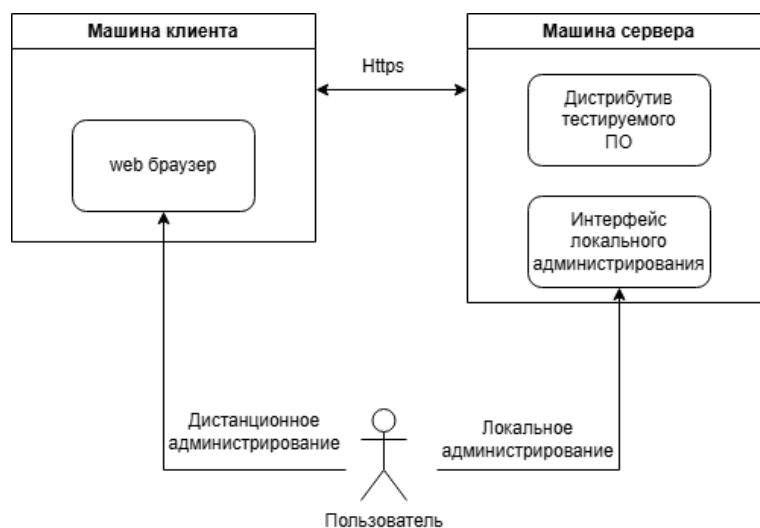


Рисунок 4 – Схема взаимодействия пользователя или тестировщика с разрабатываемым ПО

Как видно из схемы на рисунке 2, тестировщику требуется как минимум две виртуальные машины, для того что – бы полноценно проверить работоспособность тестируемого ПО. Одна из этих виртуальных машин будет использоваться в качестве сервера тестируемого программного обеспечения, а вторая – в качестве клиента дистанционного администрирования через web – приложение.

2 Постановка задачи

Исходя из проведенного анализа предметной области результатом данной работы должна стать автоматизированная система тестирования, удовлетворяющая следующим требованиям:

- автоматизированное развертывание стенда виртуальных машин;
- установка на тестовый стенд нового дистрибутива, разрабатываемого ПО;
- автоматизированное выполнение соответствующих для каждого режима работы тестов, взаимодейству через псевдографический экран;
- автоматизированное выполнение тестов на web – клиенте;
- генерация понятного для пользователя отчета о прохождении тестов и возможно возникших ошибках.

Согласно требованиям к система автоматизированного тестирования, составим диаграмму вариантов использования (рисунок 5).

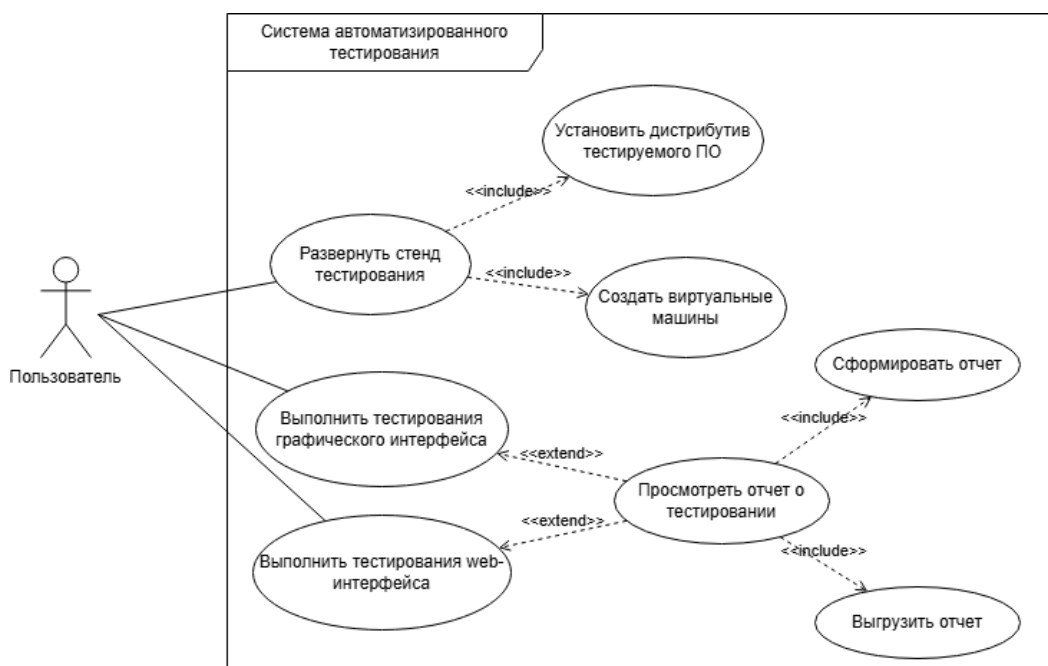


Рисунок 5 – Диаграмма вариантов использования

Выделенные на диаграмме варианты использования позволят облегчить процесс реализации системы тестирования, а также лучше понять, какие основные функции она должна автоматизировано выполнять.

3 Выбор инструментов разработки

Важным шагом при создании любого информационного продукта является выбор используемых инструментов разработки. Выбранный «стек» должен позволять закрыть все установленные требования к результату, а так – же должен быть понятен в использовании и относительно лёгок в освоении. Так как предполагается подключать к разработке авто-тестов членов команды тестирования, требование к простоте и скорости освоения стека технологий стоит особо выделить.

3.1 Выбор языка программирования

В первую очередь стоит определиться с выбором языка программирования, так как от этого во много будет зависеть дальнейший выбор доступных инструментов автоматизации, фреймворков тестирования и способов генерации отчета. В современной практике для автоматизации тестирования применяют множество языков программирования, отдельно можно выделить Java и Python.

Java – это высокоуровневый, объектно-ориентированный язык программирования, известный своей кроссплатформенностью благодаря JVM (Java Virtual Machine). В сфере QA-автоматизации Java является одним из ключевых языков благодаря мощным фреймворкам, таким как Selenium WebDriver (для тестирования веб-приложений), Appium (мобильное тестирование), JUnit и TestNG (для модульного и интеграционного тестирования)[3]. Его строгая типизация, богатая экосистема библиотек и поддержка многопоточности делают Java частым выбором для создания стабильных и масштабируемых автотестов, особенно в крупных проектах с высокой нагрузкой. Несмотря на популярность Java в автоматизированном тестировании, у языка есть несколько существенных минусов:

- сложность и многословность синтаксиса;
- долгая разработка и поддержка;

- сложность настройки инфраструктуры;
- менее удобен для DevOps – интеграции;
- сложность и многословность синтаксиса.

Java требует больше кода для простых операций по сравнению с другими языками. Жёсткая типизация, необходимость объявлять классы и методы, а также обработка исключений усложняют написание тестов и делают код более громоздким[6].

Из-за статической типизации и необходимости компиляции написание и отладка тестов занимают больше времени. Даже небольшие изменения требуют перекомпиляции, что замедляет процесс разработки, особенно в сравнении с интерпретируемыми языками.

Для работы Java – тестов часто требуется Maven или Gradle, управление зависимостями и JVM, что добавляет накладных расходов по сравнению с более лёгкими решениями (например, pytest в Python).

Хотя Java-тесты можно запускать в инструментах CI/CD (например, Jenkins или GitLab CI), их настройка сложнее, чем для скриптовых языков. Также JVM потребляет больше ресурсов, что может быть критично в контейнеризированных средах.

Python – один из самых популярных языков для автоматизированного тестирования благодаря простоте синтаксиса, богатой экосистеме библиотек и высокой скорости разработки. Главные преимущества Python – лаконичный код, динамическая типизация и поддержка разных парадигм, таких как ООП, и функциональное программирование, что ускоряет написание тестов[3]. Однако у языка есть и недостатки:

- медленное выполнение;
- проблемы с многопоточностью;
- динамическая типизация может приводить к ошибкам;
- менее строгая структура, из-за чего в больших проектах код сложнее поддерживать.

Таблица 1 – Сравнение языков программирования Java и Python

Критерий	Java	Python
Скорость разработки	Медленнее (многословный код, необходимость компиляции)	Быстрее (лаконичный синтаксис)
Производительность	Быстрее (JVM, многопоточность)	Медленнее (интерпретируемый ЯП)
Интеграция с CI/CD	Труднее (требуется Maven или Gradle)	Легче (достаточно установки python и зависимостей pip)
Наличие фреймворков автоматизации тестирования	Меньше (Selenium, Appium, Junit и TestNG)	Значительно больше (Selenium, Playwright, PyTest, Appium, Request)

Исходя из сравнения Python и Java, и беря во внимание требования к скорости разработки проекта авто-тестирования, желательной простоте кода и вовлечения тестировщиков в написания авто-тестов, Python становится предпочтительным выбором.

3.2 Выбор фреймворка тестирования

Python предлагает множество мощных фреймворков для автоматизированного тестирования, каждый из которых подходит для разных задач.

Unittest – это встроенный модуль Python, вдохновленный JUnit. Он поддерживает создание тест-кейсов, фикстур, проверок (assertions) и тестовых наборов (test suites)[8]. Хотя он надежен и не требует дополнительных зависимостей, его синтаксис более многословен по сравнению с современными альтернативами. Так же Unittest обладает довольно богатым функционалом для

настройки тестов, а именно различные встроенные декораторы, функции автоматического запуска тестов и функции проверки ожидаемого результата. Unittest – довольно старый фреймворк, поэтому в нем нету встроенной поддержки параллельного запуска тестов с использованием многопоточности.

Выделим преимущества и недостатки фреймворка.

В преимущества Unittest можно отметить:

- не требует отдельной установки и сразу готов к использованию;
- легко интегрируется в разные среды разработки;
- обладает расширяемой архитектурой;
- богатый встроенный функционал управления тестами.

Из недостатков можно выделить:

- код для теста и код для приложения нужно размещать в одном файле;
- громоздкие конструкции запуска тестов;
- сложность конфигурации окружения;
- нет поддержки динамических тестов.

Приведем пример написания авто – теста с использованием фреймворка Unittest для проверки функции (листинг 1).

Листинг 1 – Пример кода теста с использованием фреймворка Unittest

```
import unittest

# Определяем функцию add, которую будем тестировать
def add(x, y):
    return x + y

# Создаём дочерний класс TestAddition, который наследует от
unittest.TestCase
class TestAddition(unittest.TestCase):
    # Методы тестирования определяются с префиксом "test_"
    # В данном тесте мы проверяем, что функция add корректно
    # складывает два числа
    def test_addition(self):
        self.assertEqual(add(2, 3), 5)

if __name__ == "__main__":
    unittest.main()
```

Unittest – достаточно надёжный инструмент для тестирования приложений на Python. Он подходит для многих задач, например, для базового модульного тестирования. Особенно удобно, что не нужно ничего устанавливать и настраивать – всё нужное для работы в Python в нём уже есть.

Но если нужен более простой, понятный и лаконичный инструмент, лучше выбрать другой фреймворк[10]. Особенно для новичков, которые пока не знакомы с языком Python и изучают его только, чтобы писать автоматические тесты.

Фреймворк Behave тоже достаточно популярен для тестирования с помощью Python и при этом сильно отличается от остальных. Для написания тестов используется специальный формат – Gherkin[10]. Этот формат позволяет создавать тесты на английском языке, которые затем преобразуются в автоматические тесты уже на языке Python. Такой подход называется «Behaviour Driven Development». Чтобы обеспечить автоматическое преобразование сценариев в выполняемые тесты, в фреймворк Behave использует несколько компонентов. Во – первых используются Feature – файлы, в них описывают тесты на английском языке по специальным правилам. Их может писать специалист или инженер, знакомый с форматом Gherkin. Для их преобразования в исполняемый код требуются Step Definitions. Это функции на Python, в которых описано, как выполнять шаги из Feature-файлов. Их могут реализовывать или разрабатывать тестировщики, знакомые с Python.

В преимущества фреймворка Behave можно отнести:

- простой и понятный синтаксис Feature-файлов;
- поддержка подхода DRY (Don't Repeat Yourself);
- поддержка параллельного выполнения тестов.

Так же нужно отметить и недостатки:

- сложность написания функций Step Definitions;
- отсутствие встроенной поддержки API-тестирования или выполнения HTTP-запросов из сценариев;
- необходимость понимания BDD-методологии;

– скудная документация.

Приведем пример Feature файла теста с использованием фреймворка Behave (листринг 2).

Листринг 2 – Пример кода теста Feature файла фреймворка Behave

```
Feature: Login
  In order to access my account
  As a registered user
  I want to be able to login

  Scenario: Successful login with correct credentials
    Given I am on the login page
    When I enter my username and password
    And click the login button
    Then I should be redirected to the homepage
```

Видно, что тест пишется в стиле естественного языка. Для выполнения этого теста требуется написание кода с определением этих Feature (листринг 3).

Листринг 3 – Пример кода Step Definitions для теста из листринга 2

```
from behave import given, when, then

# Используется декоратор для замены текста на функцию
@given('I am on the login page')
def step_impl(context):
    context.browser.get('http://localhost:8000/login/')

@when('I enter my username and password')
def step_impl(context):

    # Выполнение действия на странице внутри кода функции шага
    context.browser.find_element_by_id('id_username').send_key('myuser
name')

    context.browser.find_element_by_id('id_password').send_keys('mypas
sword')

@when('click the login button')
def step_impl(context):
    context.browser.find_element_by_id('login_button').click()

@then('I should be redirected to the homepage')
def step_impl(context):
    assert context.browser.current_url ==
'http://localhost:8000/home/'
```

Фреймворк Behave стоит использовать, если в компании уже используется подход BDD и язык Grekhin[10], в противном случае внедрять новый подход может быть не целесообразно. Так же, Behave не подойдет для создания MVP (minimum valuable product – минимально жизнеспособный продукт), так как требует изначально довольно сложного подхода.

PyTest – самый популярный фреймворк для автоматизирования тестирования на Python[7]. Это open-source решение – ПО с открытым исходным кодом. Оно развивается за счет комьюнити и поддерживает практически все необходимые тестировщику функции: модульное и функциональное тестирование, а также тестирование API. PyTest стал стандартом де-факто для тестирования в Python благодаря своей простоте и расширяемости[11].

Его ключевыми особенностями можно назвать:

- минималистичный синтаксис тестовых функций;
- использование фикстур (@pytest.fixture) для управление данными и состоянием тестов;
- параметризация тестов;
- удобный и минималистичная функция проверки;
- богатый набор плагинов и интеграций;
- подробные отчеты об ошибках.

Минусами PyTest можно назвать:

- необходимость отдельной установки и настройки;
- несовместимость с версиями Python ниже 3.7.

Приведем пример использования PyTest для написания теста проверки функции (листинг 4).

Листинг 4 – Пример кода теста с использованием фреймворка PyTest:

```
import pytest

# Функция сложения двух чисел, которую будем тестировать
def add_function(x, y):
    return x + y

# Функция теста PyTest
```

```
# через @pytest.mark.parametrize заданы несколько наборов значений
@pytest.mark.parametrize("x, y, result", [(2, 3, 5), (2, 2, 5)])
def test_add_function(x, y, result):
```

```
# Вызов assert для проверки результат работы функции
    assert add_function(x, y) == result
```

Так же PyTest предоставляет довольно подробный отчет о выполненных тестах в виде текстового вывода в консоль. В выводе этого отчета можно быстро определить, на каком этапе выполнения произошла ошибка.

```
example_pytest.py::test_add_function[2-3-5] PASSED [ 50%]
example_pytest.py::test_add_function[2-2-5] FAILED [100%]

===== FAILURES =====
----- test_add_function[2-2-5] -----

x = 2, y = 2, result = 5

    @pytest.mark.parametrize("x, y, result", [(2, 3, 5), (2, 2, 5)])
    def test_add_function(x, y, result):
        # Вызов assert, для проверки работы нашей функции
        assert add_function(x, y) == result
>
E       assert 4 == 5
E       + where 4 = add_function(2, 2)

example_pytest.py:12: AssertionError
===== short test summary info =====
FAILED example_pytest.py::test_add_function[2-2-5] - assert 4 == 5
===== 1 failed, 1 passed in 0.01s =====
```

Рисунок 6 – Пример отчета о прохождении теста от PyTest

PyTest – практически универсальный фреймворк, который подходит для тестирования большинства приложений и сайтов. Он позволяет писать даже сложные тесты и минимальными усилиями и поддерживает большинство методологий тестирования. Поэтому в нашем случае он станет подходящим выбором в качестве фреймворка тестирования для автоматизации тестирования разрабатываемого ПО.

3.3 Выбор фреймворка формирования отчета

Выдвигая требования к результату разработки проекта авто – тестирования ПО, мы отдельно отметили важность формирования подробного и понятного отчета, о прохождении авто-тестов. Подобный отчет необходим для того, что – бы сразу понять, какие функции разрабатываемого приложения работают

штатно, а в каких возникли ошибки. Так-же сформированный отчет можно использовать в качестве «заключения» о работоспособности данной конкретной итерации разработки ПО.

Выбранный фреймворк тестирования PyTest сам предоставляет довольно подробный текстовый отчет о прохождении тестов (рисунок 6), о времени, затраченном на каждый тест и о местах возникновения ошибок[11]. Но, так как отчет от PyTest представляет собой текстовый вывод в консоль, он не достаточно наглядный. Так –же нет возможности кастомизировать вывод отчета, разделить его на шаги или другие логические блоки. В этом случае стоит рассмотреть существующие популярные инструменты для формирования отчета в Python.

Pytest –html – это плагин для PyTest, который генерирует HTML-отчеты о выполнении тестов[20]. Он прост в использовании и не требует сложной настройки, что делает его удобным для быстрого анализа результатов тестирования. Для формирования отчета после прохождения тестов достаточно добавить «--html=report.html» в команду для запуска сценария тестов PyTest.

```
> pytest -v --html=report.html --capture=tee-sys example_pytest.py
```

Рисунок 7 – Пример команды запуска теста с генерацией html отчета

Так же pytest –html позволяет добавлять в отчет вывод от команды «print()» и стриншоты от Selenium или Playwright. Сформируем отчет с использованием pytest –html и откроем в браузере (рисунок 8).

report.html

Report generated on 25-May-2025 at 21:12:32 by [pytest-html](#) v4.1.1

Environment *(hide details)*

Python	3.13.2
Platform	macOS-14.6.1-arm64-arm-64bit-Mach-O
Packages	<ul style="list-style-type: none">• pytest: 8.3.5• pluggy: 1.6.0
Plugins	<ul style="list-style-type: none">• html: 4.1.1• metadata: 3.1.1

Summary

2 tests took 1 ms.

(Un)check the boxes to filter the results.

☒ 1 Failed, ☒ 1 Passed, ☒ 0 Skipped, ☒ 0 Expected failures, ☒ 0 Unexpected passes, ☒ 0 Errors, ☒ 0 Reruns

Result 	Test
Failed	example_pytest.py::test_add_function[2-2-5]
<pre>x = 2, y = 2, result = 5 @pytest.mark.parametrize("x, y, result", [(2, 3, 5), (2, 2, 5)]) def test_add_function(x, y, result): # Вызов assert, для проверки работы нашей функции assert add_function(x, y) == result > assert 4 == 5 E + where 4 = add_function(2, 2) example_pytest.py:12: AssertionError</pre>	
Passed	example_pytest.py::test_add_function[2-3-5]

Рисунок 8 – Пример отчета от pytest –html

Выделим плюсы и минусы этого инструмента формирования отчета. К плюсам можно отнести:

- простота установки и настройки за счет полноценной интеграции с PyTest;
- удобный формат отчета в виде html файла;
- возможность добавлять дополнительные данные в отчет.

Из минусов стоит отметить:

- отсутствие истории запуска тестов;
- ограниченная аналитика в отчете;
- минимальная кастомизация структуры отчета.

Наиболее популярным инструментом генерации отчетов авто-тестов является Allure-report[15]. Это мощный фреймворк для создания интерактивных и детализированных отчетов о выполнении тестов в формате html документа. Он поддерживает множество языков (Java, Python, JavaScript и др.) и фреймворков (Pytest, TestNG, JUnit, Cucumber и др.)(21).

Allure-report имеет плагин для интеграции с PyTest[22]. Он позволяет кастомизировать отчет используя различные декораторы в коде теста на Python. Приведем пример использования декораторов плагина Pytest-Allure для кастомизации отчета (листинг 5).

Листинг 5 – Пример кода теста с использованием декораторов Allure для форматирования отчета

```
import pytest
import allure

def add_function(x, y):
    with allure.step("Первый шаг теста"):
        print("Это первый шаг теста")
    with allure.step(f"Это второй шаг теста, возвращаем сумму {x}
+ {y}"):
        return x + y

@allure.epic("Это верхняя группа заголовков")
@allure.story("Это второй подзаголовок группы")
@allure.title("Это заголовок теста")
@allure.testcase("Здесь может быть информация о тест-кейсе")
@pytest.mark.parametrize("x, y, result", [(2, 3, 5), (2, 2, 5)])
def test_add_function(x, y, result):
    with allure.step("Тут могло быть описание подготовки теста"):
        assert add_function(x, y) == result
```

После выполнения теста с использованными декораторами, Allure автоматически сформирует отчет в один файл в формате html. Его можно открыть и изучить в браузере (рисунок 9).

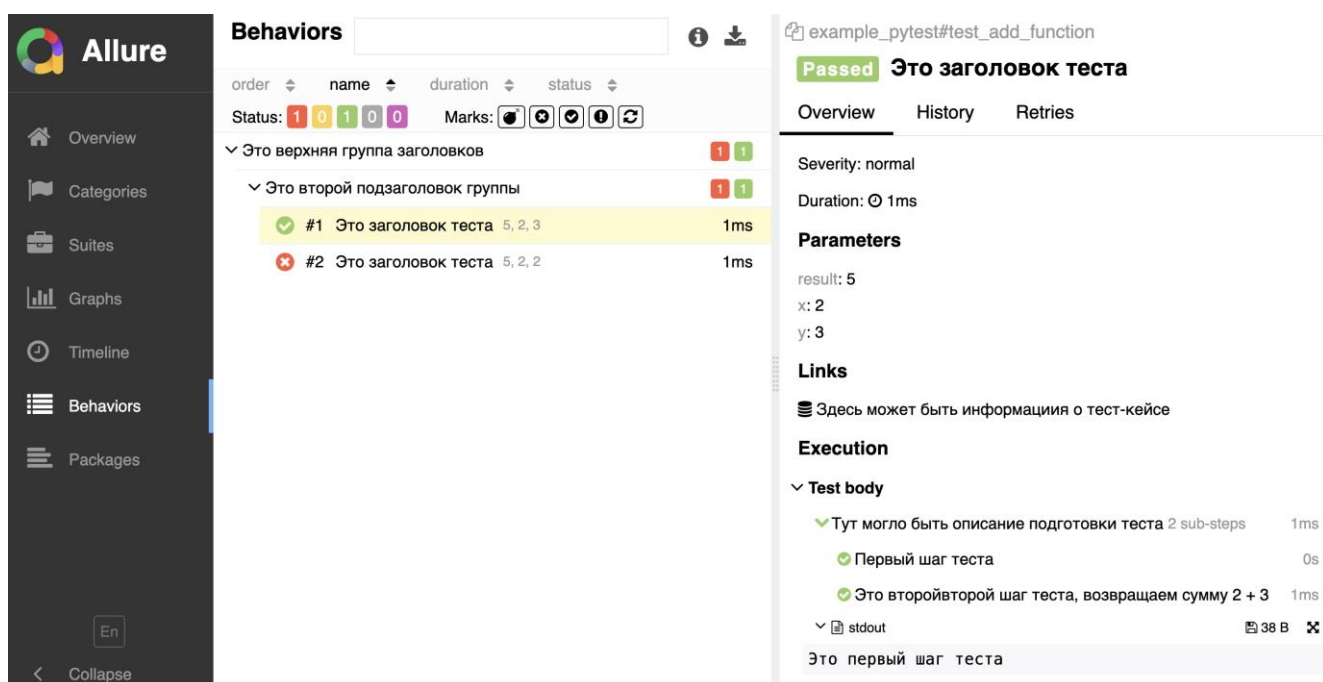


Рисунок 9 – Пример отчета о прохождении теста с форматированием Allure

Как видно из скриншота примера отчета, Allure предоставляет очень наглядный и подробный отчет о прохождении тестов. Так же стоит отметить удобный и современный интерфейс страницы отчета.

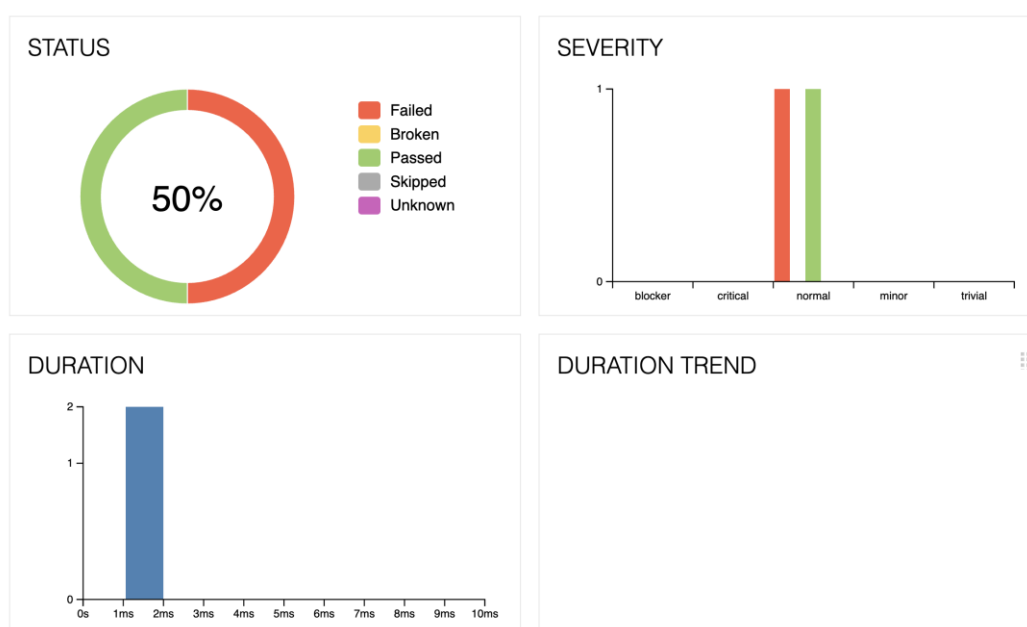


Рисунок 10 – Пример аналитики прохождения тестов от Allure

Подобная собираемая статистика от Allure может пригодится для анализа и выявления недостаточно быстро работающих функций приложения. Так же

сразу становится понятно, какой процент покрытого авто –тестами функционала отрабатывает верно.

Для того, что –бы определиться с выбором фреймворка формирования отчета, сформулируем плюсы и минусы Allure –report. К плюсам можно отнести:

- возможность разбивки тестов на группы и шаги;
- возможность добавлять в отчет дополнительные данные;
- понятный интерактивный интерфейс отчета;
- анализ статистики прохождения тестов;
- удобная интеграция с CI/CD.

Так же стоит отметить и недостатки:

- требуется настройка Allure CLI для интеграции с CI/CD;
- требуется установка allure – server и его зависимостей;
- может быть избыточным для небольших проектов.

Allure – это мощный инструмент для генерации отчетов в Python-тестировании. Он требует больше настроек, чем pytest – html, но дает гораздо больше возможностей для анализа результатов, что делает его предпочтительным выбором в качестве инструмента формирования отчета.

3.4 Выбор инструментов управления приложением

Разрабатываемая система автоматического тестирования будет тестировать приложение с точки зрения подхода приемочного тестирования, а значит – симулировать возможные действия пользователя. Исходя из этого, требуется верно подобрать инструменты и библиотеки, так же беря во внимание специфичность интерфейсов тестируемого ПО.

3.4.1 Взаимодействие с экраном локального администрирования

Как упоминалось ранее, пользователи могут взаимодействовать с разрабатываемым ПО через нажатия на специальный псевдографический экран, а для удобства обеспечения процесса тестирования был разработан эмулятор данного экрана, запускаемый в графическом окружении X11 для ОС Linux[12].

Значит, для симуляции действий пользователя с экраном, требуются инструменты, которые:

- позволяют отправлять «нажатия» на интерактивные элементы эмулятора экрана;
- проверяют результат выполненных действий.

Xlib – это низкоуровневая библиотека для работы с X Window System (основной графической системой в Linux/Unix). Python-xlib – это Python – обёртка вокруг Xlib, предоставляющая удобный API для управления окнами, ввода и мониторинга событий. Данная библиотека предоставляет обширные возможности управления приложениями, основные из них:

- получение информации об активных окнах;
- управление окнами приложений;
- эмуляция ввода мыши и клавиатуры через XTest;
- работа с буфером обмена.

Приведем пример кода с использованием библиотеки Xlib для автоматизации взаимодействия с графическим приложением Gedit (листинг 6).

Листинг 6 – Автоматизация работы с окном Gedit через инструменты Xlib

```
from Xlib import X, display, Xutil

d = display.Display()
root = d.screen().root

# Запустить графическое приложение gedit (через subprocess)
import subprocess
subprocess.Popen(["gedit"])

# Найти окно приложения Gedit по названию
win = None
for window in root.query_tree().children:
    name = window.get_wm_name()
    if name and "gedit" in name.lower():
        win = window
        break

if win:
    # Фокусировка окна приложения
    win.set_input_focus(X.RevertToParent, X.CurrentTime)
    d.sync()
```

```
xtest = d.xtest()
# Симуляция ввода с клавиатуры
xtest.fake_input(d, X.KeyPress, 38)
xtest.fake_input(d, X.KeyRelease, 38)
d.sync()
```

Библиотека Xlib предоставляет полный контроль для управления окнами приложений и симуляции взаимодействия с ними мышью и клавиатурой, но в ней отсутствует критический компонент – а именно возможность каким-либо образом указать с каким конкретным интерактивным элементом приложения нужно провзаимодействовать. Единственный способ – это указать точные координаты на экране (например X :1200, Y:550) и, переместив туда курсор мыши, симулировать нажатие. Данный способ кажется не надежным и трудоемким, пользуясь им придется «в ручную» прописывать координаты всех интерактивных элементов тестируемого приложения (кнопок, полей ввода и тому подобных). Так же данная библиотека не предоставляет возможностей каким –либо образом подтвердить правильность выполнения теста. Сформулируем плюсы и минусы использования этой библиотеки. К плюсам относятся:

- полный контроль над окном тестируемого приложения;
- высокая скорость выполнения за счет низкоуровневого доступа.

Из минусов можно выделить:

- сложный API;
- требуются знания X11;
- отсутствуют возможности проверки результата работы.

Таким образом возможностей библиотеки Xlib недостаточно для автоматизации взаимодействия и проверки графического экрана. Ее можно использовать для симуляции мыши, но для подтверждения результата прохождения теста нужно найти дополнительные инструменты, например, библиотека «компьютерного зрения» OpenCV.

OpenCV (Open Source Computer Vision Library) – это мощная библиотека компьютерного зрения с открытым исходным кодом, разработанная Intel. Она

обладает обширным функционалом работы с изображением и видео, но нас интересует ее возможности распознавания изображений. Ее можно использовать для валидирования результата прохождения теста и для поиска интерактивных элементов (кнопок, полей ввода и т. д.).

Так же существует популярное средство автоматизации работы с GUI – PyAutoGui, которое в своей основе использует функционал библиотек Xlib, и при этом имеет средства распознавания изображений на экране.

PyAutoGUI – это библиотека Python, предназначенная для автоматизации управления мышью, клавиатурой и экрана, а также для работы с графическим интерфейсом (GUI)[13]. Она позволяет записывать и воспроизводить движения мыши, имитировать нажатие клавиш, делать снимки экрана и распознавать элементы на экране. PyAutoGUI имеет интуитивно понятный интерфейс и кроссплатформенную совместимость, что упрощает создание автоматизированных скриптов.

Пример кода с использованием PyAutoGui для автоматизации поиска и нажатия на изображение (листинг 7).

Листинг 7 – Использование PyAutoGui для нажатие на изображение на экране

```
import pyautogui

# Поиск изображения на экране, похожего на изображение из файла
target = pyautogui.locateCenterOnScreen("image_to_click.png",
confidence=0.9)
# Перемещение курсора мыши к координатам изображения
pyautogui.moveTo(target, duration=1)
# Симуляция нажатия ЛКМ мыши
pyautogui.click()
# Симуляция нажатия ПКМ мыши
pyautogui.click(button="right")
# Симуляция двойного нажатия ЛКМ мыши
pyautogui.doubleClick()
```

Можно выделить плюсы использования библиотеки PyAutoGui для автоматизации тестирования:

- простота использования;
- не требуется доступ к коду приложения;

- подробная документация.

Из минусов стоит назвать:

- относительно медленная скорость работы;
- нет возможности взаимодействовать с «невидимыми» элементами;
- необходимость поддерживать скриншоты интерактивных элементов;
- относительная ненадежность распознавания изображений на экране;
- нет возможности работы с текстом.

Не смотря на большое количество ограничений, PyAutoGui подходит для автоматизации тестирования графического приложения[12]. Ненадежность распознавания элементов можно будет скомпенсировать, вычлняя координаты экрана эмулятора и отталкиваясь от них, искать интерактивный элемент. А для распознавания текста и его проверки можно использовать специализированную библиотеку компьютерного зрения, например, Tesseract OCR. Она имеет поддержку русского и английского языков и обертку для Python – pytesseract.

Таким образом для автоматизации тестирования графического экрана целесообразно использовать PyAutoGui для выполнения нажатий мыши по интерактивным элементам и pytesseract для вычленения текста и его дальнейшей проверки.

3.4.2 Взаимодействие с интерфейсом удаленного администрирования

Так как для разрабатываемого приложения создан web–клиент для дистанционного администрирования запускаемый в браузере, следует автоматизировать тестирование взаимодействия с этим web–клиентом. Для этого нужно определиться с инструментами автоматизирования тестирования. Эти инструменты должны позволять симулировать действия пользователя на web – страницах клиента, такие как нажатия мышью по интерактивным элементам, прокручивание страницы, управление вкладками браузера (открытие и закрытие, переход по вкладкам), ввод текста в поля ввода, выбор и загрузка файлов в соответствующие поля. Так же инструменты тестирования web –приложения

должны позволять получать текст из элементов web –страницы для проверки правильности выполнения теста.

Автоматизация тестирования web – приложений является частой задачей для разработчиков авто–тестов[1], поэтому для этой задачи создано множество фреймворков и библиотек. Так же существуют устоявшиеся подходы написания авто-тестов web интерфейса. Для того, чтобы определиться с выбором фрейморка автоматизации работы с web–страницами, рассмотрим популярные аналоги для Python.

Одним из наиболее популярных и современных фреймворков автоматизации работы с web–браузером является Playwright[15]. Playwright – это современный фреймворк для автоматизации тестирования веб-приложений, разработанный Microsoft[17]. Он поддерживает Chromium, Firefox и Safari, что позволяет тестировать приложения во всех основных браузерах с единой кодовой базой. Playwright предоставляет высокоуровневый API для автоматизации взаимодействия с web–страницами, включая навигацию, заполнение форм, клики и проверки[16]. Приведем пример кода авто-теста формы авторизации на web–странице с использованием Playwright (листинг 8).

Листинг 8 – Авто–тест с использованием библиотеки Playwright

```
from playwright.sync_api import sync_playwright

def test_login():
    with sync_playwright() as p:
        # Запуск браузера
        browser = p.chromium.launch()
        # Открытие новой страницы
        page = browser.new_page()
        # Переход по ссылке
        page.goto("https://example.com/login")
        # Заполнение формы логина
        page.fill("#userName", "testuser")
        page.fill("#password", "Password123!")
        page.click("#login")
        # Проверка успешного входа
        assert page.is_visible("text=Profile")
        # Закрытие браузера
        browser.close()
```

Выделим плюсы использования фреймворка Playwright для авто-тестирования:

- кросс-браузерное тестирование с поддержкой большинства современных браузеров;
- встроенное ожидание элементов;
- генерация скриншотов и видео прогонов тестов;
- поддержка REST запросов для генерации тестовых данных и тестирования API;
- высокоуровневое API работы с web-страницами.

Из минусов можно отметить:

- отсутствует поддержка некоторых браузеров;
- мало готовых решений и устоявшихся практик.

Selenium WebDriver – это самый популярный фреймворк для автоматизации тестирования веб-приложений с открытым исходным кодом[15]. Он позволяет имитировать действия пользователя в браузере через API WebDriver. Поддерживает все основные браузеры такие как Chrome, Firefox, Safari, Edge, Opera[18]. Приведем пример кода тестирования web-страницы авторизации с использованием фреймворка Selenium (листинг 9).

Листинг 9 – Авто-тест с использованием библиотеки Selenium

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

def test_login_selenium():
    # Открытие браузера
    driver = webdriver.Firefox()
    # Переход по ссылке
    driver.get("https://example.com/login")
    # Явное ожидание элемента
    username = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "username"))
    )
    # Ввод строки в поле
    username.send_keys("testuser")
    driver.find_element(By.ID,
        "password").send_keys("password123")
    # Нажатие мышью на элемент
```

```

driver.find_element(By.XPATH,
"//button[@type='submit']").click()
# Проверка успешного входа
assert "Dashboard" in driver.title

```

Выделим преимущества фреймворка Selenium:

- поддержка большинства браузеров, включая устаревшие;
- большое сообщество и множество устоявшихся решений и подходов;
- более низкоуровневое API, по сравнению с Playwright, позволяющее выполнять более сложные действия и точные проверки[19];

- возможность создавать скриншоты web–страницы для отчета.

Также имеются явные минусы:

- требуются явные ожидания элементов;
- работает медленнее чем Playwright;
- требуется настройка окружения;
- требуется установка драйверов браузера.

Исходя из сопоставления плюсов и минусов и беря во внимания наличие опыта работы с фреймворком Selenium, эта библиотека станет подходящим выбором для автоматизации тестирования web–клиента разрабатываемого ПО.

4 Проектирование

На этапе проектирования системы автоматизированного тестирования приложения с псевдографическим экраном пользователя рассмотрим архитектуру основных используемых компонентов системы.

4.1 Проектирование системы тестирования

Проектирование системы автоматизированного тестирования осуществлялось с учетом особенностей тестируемого ПО и ограничений, накладываемых выбранными инструментами разработки.

Диаграмма компонентов системы приведена на рисунке 11.

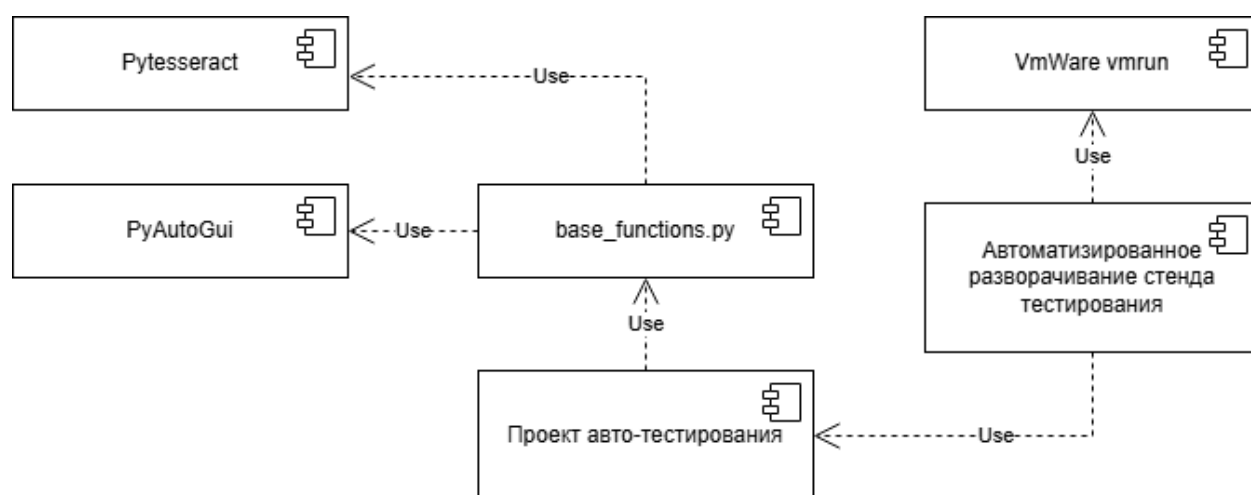


Рисунок 11 – Диаграмма компонентов системы тестирования

Рассмотрим подробнее каждый компонент системы, используемый для автоматизации тестирования.

PyAutoGui является одним из ключевых компонентов, позволяющим автоматизировать тестирование эмулятора экрана графического приложения. В практике автоматизации тестирования в большинстве случаев автоматизируется взаимодействие с web – приложением. Для этого существует множество готовых инструментов, таких как Selenium или Playwright. В нашем же случае требуется автоматизировать тестирование графического приложения под управлением ОС Linux. Библиотека PyAutoGui, за счет распознавания изображений с

применением компьютерного зрения (так – же называемого OCR), автоматизировать взаимодействие с элементами эмулятора экрана.

Основными использованными функциями библиотеки PyAutoGui являются:

- «locateCenterOnScreen()» – это функция поиска центра координат изображения на экране;
- «click()» – это функция симуляции нажатия правой кнопки мыши по переданным координатам на экране.

Pytesseract является вспомогательным компонентом, используемым для распознавания текста с изображений. Это необходимо для более точных проверок и более сложных сценариев тестирования. Так – же многие функции тестируемого приложения было бы не возможно проверить без использования распознавания текста, например, выбор элемента из текстового списка.

Основная используемая для тестирования функция библиотеки Pytesseract – это «imageToSting()». Данная функция принимает на вход файл изображения и, с использованием рекуррентной нейронной сети, натренированной на шрифтах русского и английского языков, возвращает распознанный текст.

Отдельно использовать эти компоненты не удобно, поэтому был создан «base_functions.py». Это набор функций, использующих возможности библиотек PyAutoGui и Pytesseract для реализации базовых задач при автоматизации тестирования, таких как:

- обнаружение и вычисление региона окна экрана эмулятора;
- проверка отображения элемента на экране эмулятора;
- нажатие на интерактивный элемент экрана эмулятора;
- переход по страницам экрана эмулятора.

Рассмотрим отдельно архитектуру компонента base_functions.py (рисунок 12).

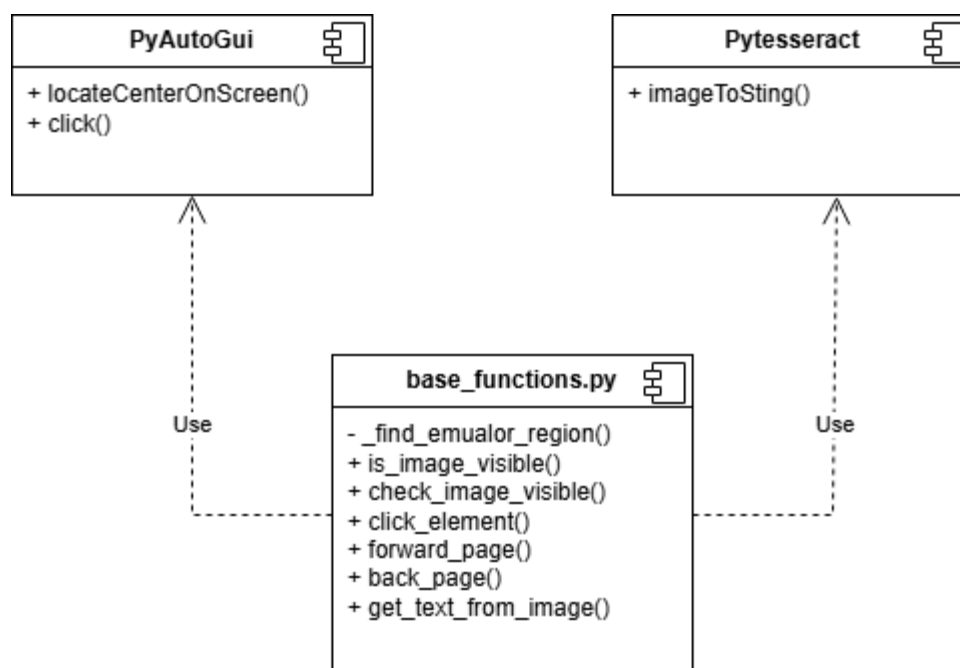


Рисунок 12 – Диаграмма классов базового функционала

Данный компонент в себя включает следующие функции:

- функция «`_find_emulator_region()`», назначение – поиск региона экрана эмулятора и вычисление его координат;
- функция «`is_image_visible()`», назначение – поиск элемента по его изображению на экране эмулятор и возврат True или False, в зависимости от результата;
- функция «`check_visible()`», назначение – проверка отображения элемента на экране эмулятора по его скриншоту, оборачивает функцию `is_image_visible()` с функцией проверки `assert()`;
- функция «`click_element()`», назначение – нажатие на интерактивный элемент на экране эмулятора по его скриншоту;
- функция «`forward_page()`», назначение – переход на следующую страницу экрана эмулятора путем нажатия на стрелку вправо в нижней части экрана;
- функция «`back_page()`», назначение – переход на предыдущую страницу экрана эмулятора путем нажатия на стрелку влево в нижней части экрана;

– функция `get_text_from_image()`, назначение – получение строки текста со скриншота определенного региона эмулятора экрана.

Таким образом компонент «`base_functions.py`» содержит в себе весь функционал, необходимый для автоматизации взаимодействия и проверки графического эмулятора экрана.

4.2 Структура проекта автоматизированных тестов

При проектировании структуры автоматизированных тестов был использован популярный паттерн PageObject.

Суть этого подхода заключается в разделении бизнес – логики взаимодействия с тестируемым приложением и логики самих тестов. Согласно этому паттерну, функционал тестируемого приложения разделяют на соответствующие страницы и создают классы этих страниц. В классах страниц приложения описывается логика взаимодействия и проверки элементов этих страниц. В самих же авто–тестах остается только логика последовательного выполнения действий, необходимых для проверки тестируемого функционала.

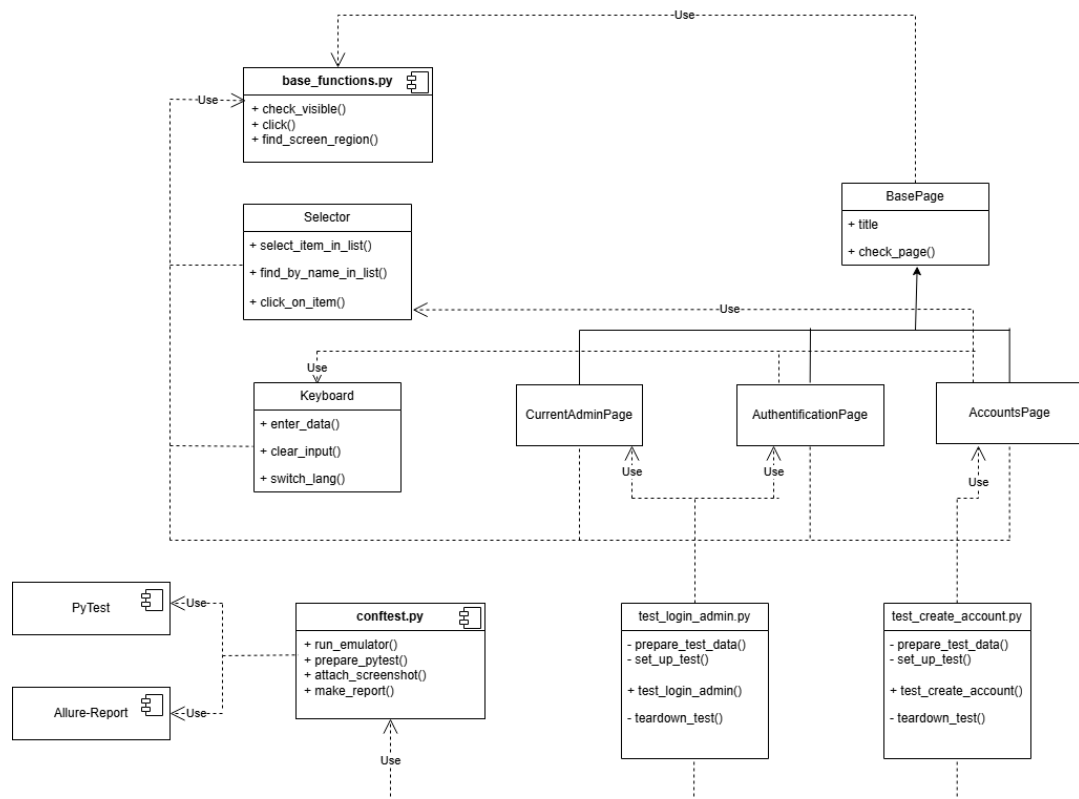


Рисунок 13 – Диаграмма классов проекта авто–тестов

На рисунке 13 приведена часть системы автоматизированного тестирования, отвечающая за описание страниц и авто-тестов. Рассмотрим подробнее изображенные классы.

Класс «BasePage» является базовым классом страницы, он содержит универсальные для всех страниц тестируемого приложения поля и методы, а именно заголовок страницы и метод проверки отображения страницы, который может быть переопределен и дополнен в классах наследниках.

Класс «CurrentAdminPage» – это один из множества классов страниц тестируемого приложения. Он наследуется от базового класса страницы и включает в себе логику проверки элементов страницы текущего администратора.

Класс «AuthenticationPage» включает в себе логику взаимодействия и проверки элементов страницы аутентификации.

Класс «AccountsPage» описывает логику взаимодействия и проверки элементов страницы учетных записей.

Класс «Keyboard» не является классом страницы, он содержит в себе логику взаимодействия с экранной клавиатурой экрана эмулятора тестируемого ПО. В его методах описаны такие действия, как ввод текста, переключения языка раскладки клавиатуры и очистка поля ввода текста. Функционал данного класса используется в методах классов страниц. Например, в классе «AuthenticationPage» метод ввода класса клавиатуры используется для заполнения полей логина и пароля учетной записи администратора. Основные методы класса «Keyboard»:

- метод «enter_data» отвечает за ввод текста путем нажатия курсором мыши по символам экранной клавиатуры;
- метод «clear_input» выполняет очистку поля ввода путем нажатия на кнопку «delete» экранной клавиатуры;
- метод «switch_lang» выполняет переключение раскладки языка экранной клавиатуры нажатием на символ смены языка.

Класс «Selector» так –же является классом элемента, используемого при проверки многих страницы. Он описывает логику взаимодействия с элементом текстового списка. Элементы в текстовом списке находятся с помощью распознавания текста из компонента «base_functions». Методы данного класса используются, например, в классе «AccountsPage» страницы учетных записей для выбора нужной учетной записи в общем списке. Основными методами этого класса являются:

- метод «select_item_in_list» выполняет поиск элемента в списке по его названию и открывает этот элемент нажатием курсора мыши;
- метод «find_by_name_in_list» выполняет проверку наличия элемента в списке;
- метод «click_on_item» выполняет нажатие мышью на элемент в списке.

Все эти классы в своих методах используют функционал компонента «base_functions.py», описанного выше.

Согласно паттерну PageObject, логика самих авто–тестов вынесена в отдельные файлы, название которых должно начинаться с «test_». Для подготовки тестового окружения и запуска экрана эмулятора используется файл «conftest.py». В нем описаны функции, выполняющие настройку инструментария PyTest перед каждой сессии тестов:

- функция «run_emulator» запускает эмулятор экрана и пробрасывает его, как графическое приложения, по ssh;
- функция «prepare_pytest» подготавливает переменные окружения и настраивает конфигурацию PyTest;
- функция «attach_screenshot» снимает скриншот экрана эмулятора и добавляет его в отчет от Allure при неудачно завершенном тесте;
- функция «make_report» формирует отчет о пройденных тестах.

В файлах тестов используется обращения к методам классов страниц для выполнения действий и проверок на них.

4.3 Проектирование автоматизированного разворачивания стенда тестирования

Важно частью системы автоматизированного тестирования является компонент автоматического развертывания и подготовки стенда тестирования.

Стенд тестирования – это подготовленная среда, позволяющая полноценно проверить весь доступный функционал тестируемого ПО. В нашем случае это две виртуальные машины под управлением средства виртуализации VmWare и машина – хост под управлением ОС Windows 10.

Для управления виртуальными машинами использует консольная утилита vmrun. Она позволяет выполнять такие действия, как клонирование, запуск и выполнение команды на виртуальной машине.

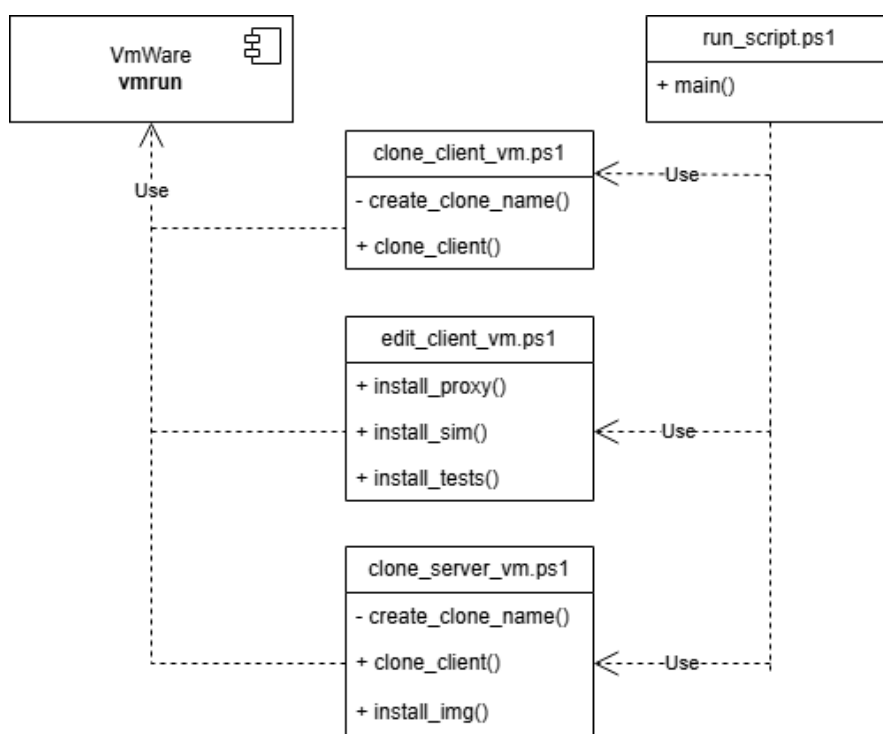


Рисунок 14 – Диаграмма классов автоматического развертывания стенда

Рассмотрим подробнее структуру разворачивания стенда тестирования (рисунок 14). Авто –разворачивание состоит из нескольких файлов скриптов на языке программирования PowerShell.

Файл «clone_client_vm.ps1» отвечает за клонирование виртуальной машины клиента и содержит функции:

- функция «create_clone_name» выполняет генерацию имени машины клона;
- функция «clone_client» выполняет клонирование и запуск машины клиента.

В файле «edit_client_vm.ps1» собраны функции настройки машины клиента и установки на нее необходимых зависимостей:

- функция «install_proxu» устанавливает прокси-сервер и выполняет его настройку;
- функция «install_sim» устанавливает симулятор виртуальных ключей и выполняет его настройку;
- функция «install_tests» копирует на машину клиента проекты авто-тестов графического интерфейса и web-интерфейса.

В файле «clone_server_vm.ps1» содержатся функции клонирования и настройки виртуальной машины сервера:

- функция «create_clone_name» генерирует уникальное имя машины сервера;
- функция «clone_client» клонирует и запускает машину сервера;
- функция «install_img» устанавливает на машину сервера установочный образ тестируемого ПО.

Файл «run_script.ps1» содержит функцию «main», которая является точкой входа выполнения автоматического разворачивания стенда тестирования. В ней последовательно вызываются все функции, отвечающие за клонирование, настройку и запуск виртуальных машин. Все взаимодействие с виртуальными машинами осуществляется путем выполнения запросов к консольной утилите управления виртуальными машинами «vmrun».

5 Реализация проекта автотестирования

Определившись с инструментами и фреймворками разработки можно преступать к реализации проекта автоматизации тестирования. В первую очередь стоит автоматизировать тестирование локального администрирования, так как преступить к тестированию web-клиента дистанционного администрирования можно только имея настроенный на этапе первичной инициализации сервер.

5.1 Базовый функционал

Так как для автоматизации тестирования локального администрирования, которое представляет из себя графическое приложение на ОС Linux, мы будем использовать библиотеки, не заточенные именно под авто-тестирование, базовый функционал нужно описать самостоятельно. Перечислим минимальный набор функций, который позволит автоматизировать тестирование:

- поиск элемента на графическом экране;
- взаимодействие с элементом (клик мыши);
- проверка наличия или отсутствия элемента на экране;
- ожидание появления или исчезновения элемента;
- получение текста из элемента для дальнейшей проверки.

В выбранной для автоматизации библиотеке PyAutoGui есть метод `click(x, y)`, который принимает на вход координаты (X;Y) точки на экране симулирует нажатие правой кнопкой мыши по этим координатам. Координаты интерактивного элемента, который мы хотим нажать в ходе авто-теста, можно получить с помощью другой функции PyAutoGui – `locateCenterOnScreen(image_file, time_to_search, region_to_search, confidence)`. Данная функция принимает в параметрах путь до файла изображения, которое нужно найти на экране, время поиска и регион экрана, на котором будет выполняться поиск (по умолчанию – весь экран). Найдя изображение,

достаточно схожее (параметр confidence) с картинкой из параметров, функция вернет координаты (X:Y) центра этого изображения.

Для увеличения точности поиска элементов напишем функцию, которая будет вычислять регион экрана эмулятора на экране и передавать его координаты в дальнейшие шаги поиска элементов.

Изобразим цепочку действий для взаимодействия с элементом на диаграмме (рисунок 11).



Рисунок 15 – Диаграмма последовательности взаимодействий с элементом

Для поиска самого окна эмулятора можно отталкиваться от заголовка и, заранее зная размеры окна эмулятора, вычислять его координаты. Найти заголовок можно по скриншоту заголовка окна, так – как он не изменяется.

Подобные скриншоты универсальных элементов нужно хранить в общей папке в корне проекта.

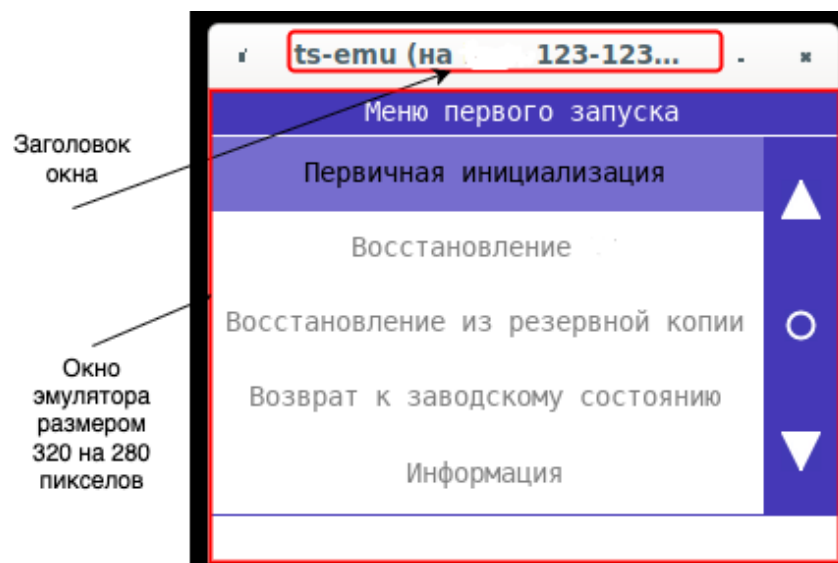


Рисунок 16 – Окно эмулятора виртуального экрана

Напишем функцию, которая будет находить и вычислять регион окна эмулятора на экране, отталкиваясь от скриншота заголовка этого окна (листинг 10).

Листинг 10 – Функция поиска окна эмулятора

```
def find_emulator_region_window(min_region_search_time=10):
    """
    Находим регион по заголовку окна ts-emu
    :param min_region_search_time: максимальное время окна эмулятора
    на экране
    :return: Кортеж [int(x), int(y), int(width), int(height)]
    где x и y - координата левого верхнего угла;
    width - ширина области; height - высота области
    """

    os.makedirs(ROOT + "/log/screens/", exist_ok=True)

    try:
        # Ищем центр изображения заголовка окна
        ts_emu = pyautogui.locateCenterOnScreen(
            ROOT + "/screens/ts_emu.png",
            minSearchTime=min_region_search_time
        )
        # Вычисляем координаты региона экранчика
        region = (int(ts_emu.x - 73), int(ts_emu.y - 20), 330, 285)
        # Скриншот для логирования
        screenshot = pyautogui.screenshot(
```

```

ROOT + "/log/screens/log_region.png", region=region
)
screenshot.save(ROOT + "/log/screens/test_region_tw.png")

# Возвращаем координаты региона экранчика
return region

# Если не найдено изображение заголовка, пытаемся найти
изображение свернутого окна
except pyautogui.ImageNotFoundException:
    try:
        ts_emu = pyautogui.locateCenterOnScreen(
            ROOT + "/screens/ts_emu_unfocused.png",
            minSearchTime=min_region_search_time,
        )
        region = (int(ts_emu.x - 73), int(ts_emu.y - 20), 330, 285)
        screenshot = pyautogui.screenshot(
            ROOT + "/log/screens/log_region.png", region=region
        )
        screenshot.save(ROOT + "/log/screens/test_region_tw.png")
        return region

# В случае, когда не найдено и свернутое окно, возвращаем
None
except pyautogui.ImageNotFoundException as ex:
    print(f"Не найдено окно эмулятора ts-emu!\nОшибка: {ex}\n")
    return None

```

Следующим шагом в цепочке взаимодействия с элементом является поиск координат самого элемента по его скриншоту внутри найденного окна эмулятора. Напишем функцию, выполняющую это действие (листинг 11).

Листинг 11 – Функция поиска элемента внутри региона экрана эмулятора

```

def is_image_visible_on_screen(
    file: string,
    confidence=0.8,
    min_search_time=1,
    min_region_search_time=10,
    in_emu_window=True,
    log_func=print,
):
    """
    Ищет изображение на экране и возвращает координаты центра

    :param file - путь до файла изображения для поиска
    :param confidence - процент соответствия с изображением из
    файла
    :param min_search_time - время ожидания элемента
    :param min_region_search_time - время ожидание окна эмулятора
    :param in_emu_window - флаг поиска внутри окна эмулятора
    """

```

```

try:
    if in_emu_window:

        # Поиск координат центра элемента внутри окна
эмулятора
        target = pyautogui.locateOnScreen(
            image=file,
            confidence=confidence,
            minSearchTime=min_search_time,

region=get_emulator_window_region(min_region_search_time),
        )
    else:

        # Поиск координат центра элемента вне окна эмулятора
        target = pyautogui.locateOnScreen(
            image=file, confidence=confidence,
minSearchTime=min_search_time
        )

        # Возвращаем True и координаты элемента
        return True, target

except pyautogui.ImageNotFoundException as ex:
    log_func(f"Не удалось найти {file} ошибка: {ex}")

    # Элемент не найден, возвращаем False
    return False, None

```

В конце необходимо написать функцию нажатия на найденный элемент (листинг 12). Так же в этой функции будет заключена проверка присутствия искомого элемента на экране, что – бы не происходили нажатия по не существующим элементам.

Листинг 12 – Функция нажатия на элемент

```

def click_element(
    file: string,
    confidence=0.8,
    min_search_time=1,
    repeat=1,
    offset_x: int = None,
    offset_y: int = None,
    min_region_search_time=10,
    log_func=print,
):
    """
    Кликает по элементу file
    либо кликает по координатам с отступом (offset_x; offset_y)
    от центра элемента file

```

```

:param file - путь до файла изображения для поиска
:param confidence - процент соответствия с изображением из
файла
:param min_search_time - время ожидания элемента
:param repeat - количество нажатий
:param offset_x - отступ по X
:param offset_y - отступ по Y
:param min_region_search_time - время ожидание окна эмулятора
"""

# Проверяем, что элемент отображает на экране
if check_element_visible(file, confidence, min_search_time,
log_func=log_func):
    target = pyautogui.locateCenterOnScreen(
        file,
        confidence=confidence,
        minSearchTime=min_search_time,

region=get_emulator_window_region(min_region_search_time),
    )

    # Получаем координаты центра элемента и по необходимости
    вычисляем отступ
    x, y = target.x, target.y
    if offset_x is not None:
        x = x + offset_x
    if offset_y is not None:
        y = y + offset_y

    # Нажимаем по полученным координатам
    pyautogui.click(x=x, y=y, clicks=repeat)

```

Данного набора функций достаточно для автоматизации таких действий пользователя как: нажатие на элементы, переход по страницам (по нажатию на кнопки – стрелки), ввод текста с экранной клавиатуры (рисунок 13).

.,?_	abc	def
ghi	jkl	mno
pqrs	tuv	wxyz
РУС	—	123

Рисунок 17 – Экранная клавиатура эмулятора

Валидировать результат прохождения теста можно проверкой наличия или отсутствия какого – то ключевого элемента на экране, например, заголовка страницы (рисунок 14), на которую должен был перейти тест.



Рисунок 18 – Заголовок страницы настроек даты и времени

Но авто-тесты так же должны проверять корректность ввода информации. Для этого в проекте выбрана библиотека pytesseract[14]. В этой библиотеки есть функционал распознавания текста на изображении, функция pytesseract.image_to_string() получает на вход путь до файла изображения и возвращает распознанный текст в виде строки. Опишем функцию, которая будет получать текст с нужного места на экране эмулятора (листинг 13).

Листинг 13 – Функция получения текста с экрана эмулятора

```
def parse_text_on_image(
    image_to_search: string,
    offset_x: int,
    offset_y: int,
    width: int,
    height: int,
    confidence=0.7,
    min_search_time=1,
    save_screen_name="log_screen.png",
    lang="rus+eng",
    min_region_search_time=10,
    config="--psm 7",
    log_func=print,
):
    """
    Функция делает скриншот области и возвращает распознанный
    текст
    :param image_to_search - картинка, от центра которой будем
    делать отступ к полю текста
    :param offset_x - отступ по X
    :param offset_y - отступ по Y
    :param width - ширина поля
    :param height - высота поля
    :param lang - языки для распознавания текста
    :param config - конфигурация для tesseract (по умолчанию
    воспринимает текст как одну строку)
```

```

: return - найденный текста или None
"""

# Ищем изображение от координат которого будем отталкиваться
if is_image_visible(
    image_to_search, confidence, min_search_time,
    log_func=log_func
):
    target = pyautogui.locateCenterOnScreen(
        image=image_to_search,
        confidence=confidence,

region=get_emulator_window_region(min_region_search_time),
    )

    # Делаем скриншот области где будем распознавать текст
    screenshot = pyautogui.screenshot(
        region=(int(target.x - offset_x), int(target.y -
offset_y), width, height)
    )
    os.makedirs(ROOT + "/log/screens/", exist_ok=True)
    screenshot.save(ROOT + "/log/screens/" + save_screen_name)

    # Возвращаем распознанный Tesseract'ом текст
    text_found = pytesseract.image_to_string(screenshot,
lang=lang, config=config)
    return text_found.strip("\n")

# Если изображение не найдено - возвращаем None
return None

```

Полученный таким образом текст не всегда будет полностью соответствовать оригиналу, могут добавиться «артефакты» – лишние символы, либо, особенно если текст содержит разные языки, некоторые буквы могут быть распознаны неверно. Для валидации не совсем корректного текста воспользуемся утилитой библиотеки `rapidfuzz`, которая сравнивает строки и возвращает процент их схожести. В авто-тестах будем сравнивать ожидаемый текст с полученным от распознавания `tesseract`, и, если процент схожести достаточно велик, считать результат удовлетворительным. Напишем функцию проверки схожести строк с использованием библиотеки `rapidfuzz` (листинг 14).

Листинг 14 – Функция проверки схожести строк

```

@allure.step("Сравнение на схожесть найденной строки с ожидаемой")
def compare_strings(s1, s2, min_similarity=85.0, log_func=print):
    """

```

```

:param s1 - строка для сравнения
:param s2 - строка для сравнения
:param min_similarity - минимальный процент схожести строк
"""
# Сравнением строки
similarity = fuzz.ratio(s1, s2,
processor=utils.default_process)
log_func(f"Схожесть строки '{s1}' и '{s2}' = {similarity}")

# Проверяем, что процент схожести не меньше ожидаемого
assert similarity > min_similarity, \
    "Строки {} и {} недостаточно схожи! Коэффициент схожести: {}".format(s1, s2, similarity)

```

Данного функционала достаточно для автоматизации разнообразных сценариев проверок эмулятора экрана локального администрирования.

5.2 Паттерн PageObject

При написании проекта авто – тестов следует уделить внимание организации кода для того, что – бы он был легко поддерживаемым и расширяемым. В современной практике автотестирования UI стандартом стал паттерн PageObject и его расширения PageComponent и PageFactory[15]. Цель применения этого паттерна в том, что – бы отделить логику работы программы автоматизации от логики теста, для того, что – бы в случае изменения UI разрабатываемого приложения, изменения нужно было бы вносить только в часть логики работы с UI но не в сам авто-тест.

Суть паттерна PageObject заключается в описании страниц приложения в виде классов, заключающих в себе элементы страницы в атрибутах и логику работы с ними в методах. Его используют для уменьшения дублирования кода. Без PageObject одни и те же действия с одним элементом могут дублироваться в разных тестах, что усложняет поддержку кода. Если изменится UI приложения или логика работы, изменения нужно будет вносить в одном месте – классе соответствующей страницы. Так – же этот подход способствует улучшению читаемости тестов. Вместо низкоуровневой логики в тестах используется понятная бизнес – логика обращения к классам страниц. Разделение логики теста и бизнес – логики улучшает гибкость и масштабируемость кода проекта

тестирования. Можно легко комбинировать методы классов разных страницы для создания более сложных сценариев тестирования. Так же единообразие паттерна PageObject позволяет разработчикам авто – тестов с легкостью переиспользовать классы, написанные другими разработчиками, и добавлять только еще не автоматизированный функционал.

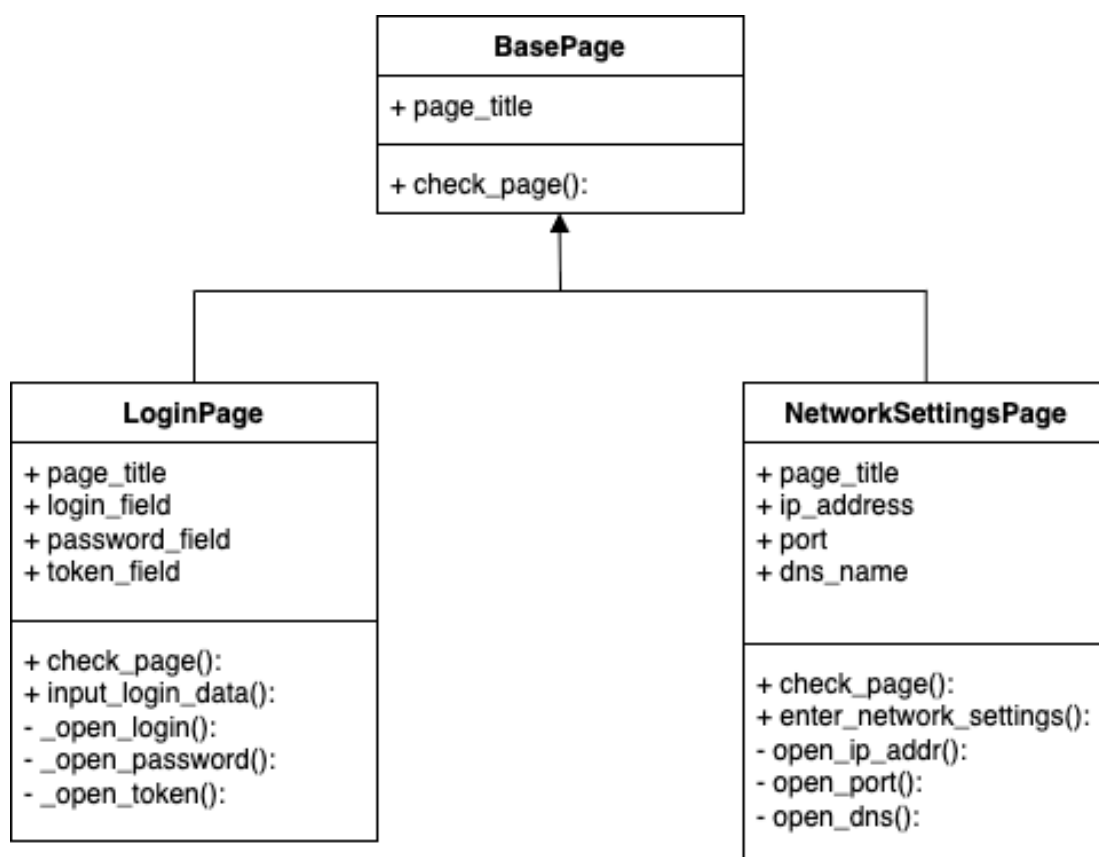


Рисунок 19 – UML-диаграмма классов страниц согласно паттерну PageObject

Из диаграммы классов страниц (рисунок 15) видно, что, согласно паттерну, создается базовый класс страницы BasePage. В нем задаются универсальные для всех страниц элементы и методы работы с ними. В нашем случае это заголовок страницы, так–как он присутствует на всех страницах локального администрирования, и метод проверки страницы, который проверяет отображение заголовка страницы и может быть переопределен в классах-наследниках с добавлением специфичных для страницы проверок. Класс базовой страницы должен быть небольшим (листинг 15).

Листинг 15 – Класс базовой страницы BasePage

```
class BasePage:
    """
    Базовый класс страниц
    при наследовании переопределяются все атрибуты
    """
    # Так как элементы страницы хранятся в виде скриншотов,
    # то в параметрах страницы указываются пути до файлов
    скриншотов
    PATH_TO_SCREENSHOT = ROOT + "/pages"
    page_title = f"{PATH_TO_SCREENSHOT}/override_me"

    # Имя страницы для отчета и проверок
    page_name = "Override me"

    @allure.step(f'Проверка отображения страницы "{page_name}"')
    def check_page(self, min_search_time=10):

        # Проверка отображения заголовка страницы, в наследниках
        можно
        # Добавлять проверки отображения других элементов страницы
        base_functions.check_visible(self.page_title,
        min_search_time)
```

Как видно, базовый класс не представляет какую – либо конкретную страницу, а лишь является заготовкой для наследников – классов реальных страниц. В классе страницы авторизации будет описана логика работы с элементами, специфичными для этой страницы, и проверки логики работы с ними (листинг 16).

Листинг 16 – Класс страницы авторизации

```
class LoginPage(BasePage):
    """
    Класс страницы Аутентификация
    """
    def __init__(self):

        # Скриншоты элементов страницы
        self.PATH_TO_SCREENSHOT = super().ROOT +
        "/pages/authentication/screens/"
        self.page_title = self.PATH_TO_SCREENSHOT + "title.png"
        self.login_title = self.PATH_TO_SCREENSHOT +
        "login_title.png"
        self.pin_title = self.PATH_TO_SCREENSHOT + "pin_title.png"
        self.select_key_title = self.PATH_TO_SCREENSHOT +
        "select_key_title.png"
        @allure.step('Проверка отображения страницы "Аутентификация"')
        def check_page(self, min_search_time=10):
```

```

        # Переопределенная проверка элементов страницы
        base_functions.check_visible(self.page_title,
min_search_time)
        base_functions.check_visible(self.login_title,
min_search_time)
        base_functions.check_visible(self.key_title,
min_search_time)
        base_functions.check_visible(self.pin_title,
min_search_time)

    # Методы, специфичные для страницы аутентификации

    @allure.step('Нажатие на "Имя учетной записи"')
    def open_input_login(self, log_func=print):
        base_functions.click(self.login_title, log_func=log_func)

    @allure.step('Нажатие на "ПИН"')
    def open_input_pin(self, log_func=print):
        base_functions.click(self.pin_title, log_func=log_func)

```

У экрана локального администрирования множество страниц. Каждый класс отвечает за хранение уникальной для этой страницы логики работы и проверок. Элементы страниц хранятся в виде полей класс с путями до файлов скриншотов этих элементов страниц.

5.3 Паттерн тестирования AAA

Важной задачей при написании надежных, легко читаемых и поддерживаемых тестов является их правильное структурирования. Существует множество подходов, но наиболее популярным является шаблон Arrange – Act – Assert (Подготовка – Выполнение – Проверка), так же называемый AAA[2].

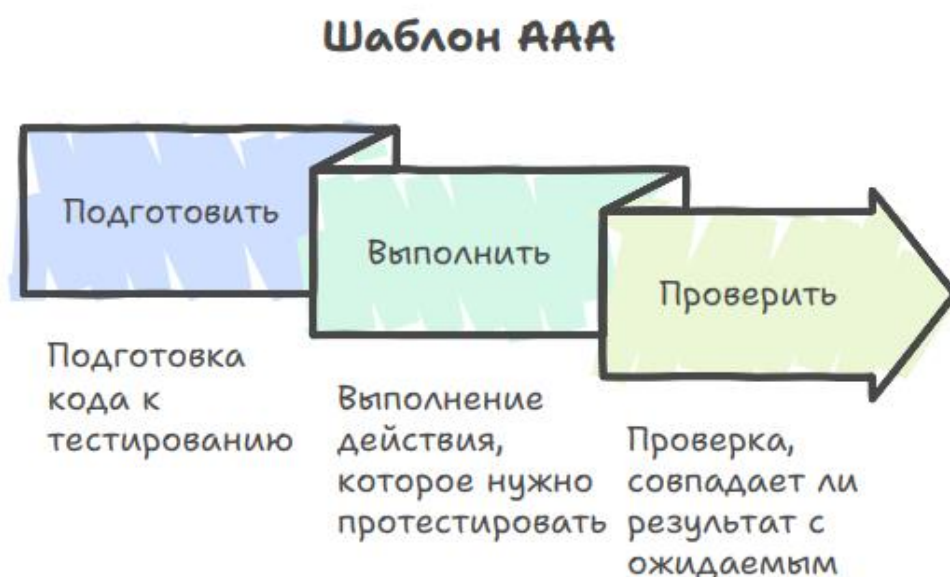


Рисунок 20 – Схема паттерна Arrange – Act – Assert

Суть этого подхода к организации тестов заключается в разделении тестов на три стандартных этапа. Первый этап Arrange заключается в подготовке настройке тестового окружения, переходе в то состояние, от куда должен начитать свое выполнение тест. Далее идет этап Act, то есть выполнение логики сценария тестирования. И последний этап Assert – проверка правильности выполненных в ходе сценария тестирования действий.

Фреймворк PyTest позволяет разбивать код авто-теста на эти этапы благодаря использованию фикстур – `@pytest.fixture`. Это специальные декораторы, предоставляемые библиотекой PyTest, которые указывают, какой код должен выполняться до или после прохождения теста[21].

Примером фикстуры, выполняющейся на этапе подготовки, будет функция запуска экрана эмулятора (листинг 17). Она автоматически при старте сессии тестов пробрасывает по ssh эмулятор экрана локального администрирования от виртуальной машины сервера на машину клиента, где выполняются тесты.

Листинг 17 – Фикстура запуска экрана эмулятора при старте сессии тестов

```
# Запуск экрана эмулятора
@pytest.fixture(autouse=True, scope="session")
def run_emu(request):

    # Получаем ip адрес VM сервера из конфига
    in_ssh = request.config.getoption("ssh")
    ssh_addr = request.config.getoption("ssh_addr")
    if in_ssh:
        # Открываем ssh и запускаем окно эмулятора
        com = f'ssh -X {ssh_addr} "cd
/usr/local/bin/dotnet/Administrator/; dotnet
./Infotecs.TtpHsm.Administrator.Local.dll;"'
        com = split(com)
        ssh_proc = subprocess.Popen(com,
stdout=subprocess.DEVNULL)
        sleep(5)
        yield
        # Закрываем процесс ssh подключения
        try:
            ssh_proc.kill()
            ssh_proc.wait(timeout=5)
        except subprocess.TimeoutExpired:
            ssh_proc.terminate()
            ssh_proc.wait()
        # Закрываем процесс окна эмулятора на машине сервера
        grep_dotnet = subprocess.run(
            [
                "ssh",
                ssh_addr,
                "ps aux | grep \"dotnet
./Infotecs.TtpHsm.Administrator.Local.dll\" | awk '{print $2}'",
            ],
            capture_output=True,
            universal_newlines=True,
        ).stdout
        for proc in grep_dotnet.strip().split("\n"):
            subprocess.run(
                ["ssh", ssh_addr, f"kill {proc}"],
                capture_output=True,
            )
    else:
        yield
```

Так –же, на этапе подготовки теста, может выполняться подготовка тестовых данных или выполняться рутинные действия, не имеющие прямого отношения к содержанию теста, например, авторизация. Приведем пример авто-теста, написанного с соблюдением шаблона AAA (листинг 18).

Листинг 18 – Авто-тест проверки авторизации с соблюдением паттерна AAA

```
@pytest.fixture
def _get_login_test_data():

    # Подготовка тестовых данных
    data = json.load(open(PATH +
"/common_test_data/admins_login_data.json"))
    return {"login": data["wrong_asrzi_admin_name"],
            "pin": data["wrong_asrzi_admin_pin"]}

@pytest.fixture
def _open_login_page():

    # Подготовка состояния теста (открытие страницы авторизации)

side_selector.scroll_to_menu_option(admin_menu.Options.AUTHENTICAT
ION)
    side_selector.click_ok()

@allure.epic("Локальное администрирование")
@allure.feature("Управление учетными записями")
@allure.title("Проверка ошибки авторизации с неверными данными
администратора")
def test_login_admin(_get_login_test_data, _open_login_page):
    (name, key, pin) = (
        _get_login_test_data["login"],
        _get_login_test_data["pin"],)
    # Выполнение последовательности действий теста
    with allure.step(f"Ввод неверного логина {name}"):
        authentication_page.open_input_login()
        keyboard.enter_data(name)
        bf.click_arrow_right()

    with allure.step(f"Ввод неверного ПИН кода {pin}"):
        authentication.open_input_pin()
        keyboard.enter_data(pin)
        bf.click_arrow_right()

    # Проверка результата выполненных действий
    with allure.step(f"Проверка отображения сообщени о неверных
данных входа"):
        authentication_page.check_error_auth_notification(min_search_time=
5)
```

Для наглядности приведем схему поэтапного выполнения теста.



Рисунок 21 – Схема поэтапного выполнения теста

Таким образом паттерн написания тестов Arrange – Act – Assert является незаменимым подходом для написания чистых и легко поддерживаемых тестов. А в сочетании с использованием возможностей PyTest, таких как фикстуры и параметризация, он позволяет создавать эффективные и масштабируемые тестовые сценарии.

6 Тестирование приложения

Запуск авто –тестов после развертывания тестового стенда происходит выполнением команды на клиентской машине. Команда старта теста выполняется через утилиту управления виртуальными машинами `vmrun` (листинг 23).

Листинг 23 – Команда запуска тестов первичной инициализации

```
# Запуск группы тестов первичной инициализации
write_log "Tests first_init started..."
& $vmrunPath -T ws -gu $vmLogin -gp $vmPass runScriptInGuest
$clientClonePath /bin/bash "source /home/user/run_test.sh
first_init /home/user"
write_log "Tests done!"
```

Тесты разделены на тематические группы, и каждая группа тестов запускается аналогичной командой. После прохождения всех групп тестов выполняется команда генерации отчета о результатах прохождения тестов `allure-report` и его выгрузка на машину пользователя (листинг 24).

Листинг 24 – Генерация и выгрузка отчета `allure-report`

```
# Генерация и выгрузка на хост allure отчета из /home/user/allure-
report
write_log "Generating report..."
& $vmrunPath -T ws -gu $vmLogin -gp $vmPass runScriptInGuest
$clientClonePath /bin/bash "allure generate --single-file -c -o
/home/user/allure-report /home/user/allure-results/"
scp -i $keyPath -o StrictHostKeyChecking=no -r
${SSHADDRESS}:/home/$vmLogin/allure-report/index.html "$distrDir"
```

После формирования отчета, он выгружается путем копирования командой `scp` на машину пользователя. Это удобно для тестировщиков, так как, они могут выполнять тестирование на уже настроенном окружении, и получать подробный отчет о работоспособности системы.

Полученный таким образом файл в формате HTML можно открыть в браузере и детально изучить процесс прохождения тестов. Пример отчета на рисунке 20.

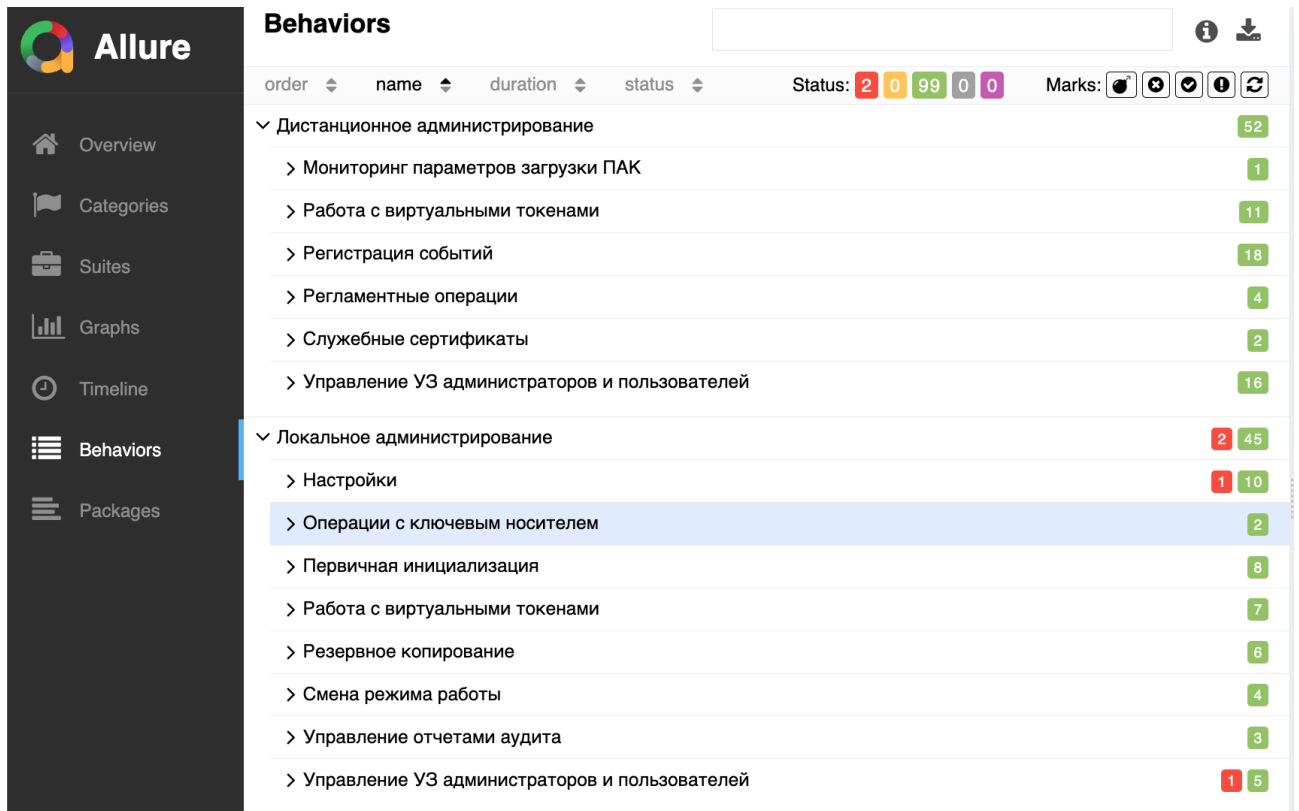


Рисунок 22 – Отчет allure-report о прохождении авто-тестов

Отчет разделен на тематические группы тестов, каждая из которых отвечает за определенную часть функционала тестируемого приложения. Это позволяет быстрее определить, какие функции работают штатно, а в каких наблюдаются проблемы. Так – же в описании шагов тестов приложена информация о тест – кейсах, соответствующих этим тестам. Можно легко находить и сверять актуальность тест – кейса и его автоматизации.

Так – же allure-report предоставляет статистику о прохождении тестов (рисунок 22). Из нее можно определить, какой процент функций приложения работает корректно, и сколько занимает время прохождения конкретного теста.

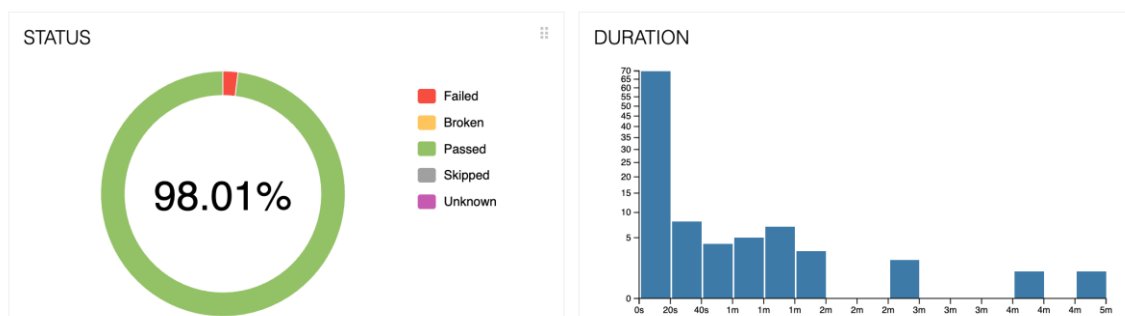


Рисунок 23 – Статистика прохождения тестов в отчете allure-report

Благодаря разбиению тестов на шаги можно подробно рассмотреть процесс прохождения конкретного теста и выяснить на каком этапе выполнения приложение могло отработать неверно, к тому же разбиение на шаги помогает воспроизводить ошибку при дальнейшей ручной проверке.

Execution		
▼ Set up		
✔ run_emu		5s 007ms
✔ attach_screenshot_on_failure		0s
➤ select_admin_mode 2 sub-steps		596ms
✔ get_admins_login_data		0s
➤ login_as_asrzi_admin 31 sub-steps		25s 068ms
➤ open_date_time_settings 16 sub-steps		4s 026ms
✔ get_date_time_setting_test_data		0s
✔ teardown		1ms
▼ Test body		
➤ Проверка отображения страницы "Дата и время" 4 sub-steps		268ms
➤ Проверка установки невалидной даты: 31/02/2024 в поле "Дата:" 7 sub-steps		10s 988ms
➤ Проверка установки допустимой даты: 25/05/2025 9 sub-steps	17s 063ms	
➤ Проверка установки значения: (+ 03:00) Asia/Baghdad в поле "Часовой пояс" 11 sub-steps	6s 372ms	
➤ Attach screenshot on test failure 1 attachment		37ms
AssertionError: Найденный текст в блоке 'Время': 122253 не соответствует ожидаемому: 13:32		
▼ Tear down		
➤ teardown::0 5 sub-steps		14s 710ms
➤ attach_screenshot_on_failure::0 1 sub-step, 1 attachment		42ms
✔ run_emu::0		578ms

Рисунок 24 – Список шагов авто–теста

Так–же, при падении теста, в отчет добавляется скриншот последнего состояния приложения (рисунок 24), это может быть полезно для более точного определения того, что в приложении отработало неверно, или это могла быть ошибка в самом тесте.

✓ Attach screenshot on test failure 1 attachment

37ms

Failed test screenshot

56 KiB



AssertionError: Найденный текст в блоке 'Время': 122253 не соответствует ожидаемому: 13:32

Рисунок 25 – Скриншот состояния сервера при ошибке в тесте

Таким образом сформированный после прохождения тестов отчет предоставляет подробную информацию о состоянии тестируемой версии ПО. Опираясь на этот отчет можно сделать вывод о работоспособности тестируемого приложения и выделить проблемный функционал. Тестировщики могут опираться на результаты выполненных авто –тестов для формирования сценариев дальнейших проверок версии разрабатываемого ПО.

Заключение

В ходе выполнения данной выпускной квалификационной работы была разработана система автоматизации тестирования приложения с псевдографическим интерфейсом пользователя. При разработке данной системы были применены актуальные практики авто-тестирования и использованы современные инструменты автоматизации тестирования, такие как язык программирования Python, библиотека тестирования PyTest и фреймворк формирования отчетов Allure-report.

В результате проведенной работы создана система автоматизированного тестирования проверяющее базовый функционал тестируемого программного обеспечения, автоматизирующее полных процесс от развертывания стенда тестирования до формирования подробного отчета о результате выполненных проверок.

В дальнейшем, за счет примененных практик создания легко поддерживаемых и расширяемых тестов, возможно создание более сложных сценариев тестирования или покрытие проверками нового функционала разрабатываемого ПО.

Список использованных источников

1. Теория тестирования от А до Я [Электронный ресурс]. – URL: <https://ilarionhalushka.github.io/testing-theory> (дата обращения 25.04.2025)
2. Виды тестирования ПО [Электронный ресурс]. – URL: http://wiki.rosalab.ru/ru/index.php/Виды_тестирования_ПО (Дата обращения: 26.04.2025).
3. Автоматизация тестирования: Java или Python? [Электронный ресурс]. – URL: <https://habr.com/ru/companies/otus/articles/492200/> (дата обращения 27.04.2025)
4. Документация языка программирования Python [Электронный ресурс]. – URL: <https://docs.python.org/3/> (дата обращения 29.04.2025)
5. Документация языка программирования Java [Электронный ресурс]. – URL: <https://docs.oracle.com/en/java/> (дата обращения 29.04.2025)
6. Java vs Python [Электронный ресурс]. – URL: <https://practicum.yandex.ru/blog/java-vs-python-chto-vybrat/> (дата обращения 29.04.2025)
7. Браян Оккен, Python testing with Pytest [Электронный ресурс]. – URL: <https://habr.com/ru/articles/426699/> (дата обращения 01.05.2025)
8. Документация фреймворка Unittest [Электронный ресурс]. – URL: <https://docs.python.org/3/library/unittest.html> (дата обращения 03.05.2025)
9. Документация фреймворка Behave [Электронный ресурс]. – URL: <https://behave.readthedocs.io/en/latest/> (дата обращения 03.05.2025)
10. Сравнение BDD фреймворков Python: плюсы и минусы [Электронный ресурс]. – URL: <https://habr.com/ru/companies/otus/articles/460046/> (дата обращения 03.05.2025)
11. Документация фреймворка Pytest [Электронный ресурс]. – URL: <https://docs.pytest.org/en/stable/> (дата обращения 03.05.2025)
12. Десктопные GUI-тесты на Python [Электронный ресурс]. – URL: <https://habr.com/ru/companies/yandex/articles/336476/> (дата обращения 08.05.2025)

13. Документация фреймворка PyAutoGui [Электронный ресурс]. – URL: <https://pyautogui-ru.readthedocs.io/ru/latest/> (дата обращения 08.05.2025)
14. Документация утилиты Tesseract–OCR [Электронный ресурс]. – URL: <https://tesseract-ocr.github.io/> (дата обращения 12.05.2025)
15. UI автотесты на Python с запуском на CI/CD и Allure отчетом. PageObject, PageComponent, PageFactory [Электронный ресурс]. – URL: <https://habr.com/ru/articles/896936/> (дата обращения 15.05.2025)
16. Документация фреймворка Playwright [Электронный ресурс]. – URL: <https://playwright.dev/python/docs/intro> (дата обращения 15.05.2025)
17. Playwright: Лучшая альтернатива Selenium. [Электронный ресурс]. – URL: <https://habr.com/ru/companies/amvera/articles/834846/> (дата обращения 15.05.2025)
18. Selenium для Python [Электронный ресурс]. – URL: <https://habr.com/ru/articles/248559/> (дата обращения 16.05.2025)
19. Документация фреймворка Selenium [Электронный ресурс]. – URL: <https://www.selenium.dev/documentation/> (дата обращения 16.05.2025)
20. Документация утилиты Pytest-html [Электронный ресурс]. – URL: <https://pytest-html.readthedocs.io/en/latest/> (дата обращения 20.05.2025)
21. Тесты на pytest с генерацией отчетов в Allure с использованием Docker и Gitlab Pages и частично selenium [Электронный ресурс]. – URL: <https://habr.com/ru/articles/513432/> (дата обращения 20.05.2025)
22. Документация фреймворка Allure-report [Электронный ресурс]. – URL: <https://allurereport.org/docs/pytest/> (дата обращения 20.05.2025)
23. Документация языка программирования PowerShell [Электронный ресурс]. – URL: <https://learn.microsoft.com/en-us/powershell/> (дата обращения 25.05.2025)
24. Документация средства виртуализации VmWare [Электронный ресурс]. – URL: <https://www.vmware.com/docs/vmware-product-guide-18-may> (дата обращения 15.05.2025)

Приложение А
ГЛОССАРИЙ
(справочное)

Фреймворк – программная платформа, определяющая структуру программной системы и облегчающая ее разработку.

Java – строго типизированный объектно-ориентированный язык программирования общего назначения.

Python – высокоуровневый язык программирования общего назначения.

E2E – вид тестирования, которые проверяет весь рабочий процесс приложения от начала до конца, эмитируя действия реального пользователя.

AAA – шаблон для структурирования автоматизированных тестов, в рамках которого тест разбивается на три этапа: подготовку (Arrange), выполнение действий (Act) и проверку (Assert).

PageObject – шаблон проектирования автоматизированных тестов, согласно которому, логика взаимодействия со страницами тестируемого приложения инкапсулируется в соответствующие классы страниц.

Тестовый стенд – это специально настроенная среда, используемая для тестирования программного обеспечения.

Тест – кейс – это последовательное структурированное описание действий, необходимых для проверки определенных функций программного обеспечения.

Авто – тест – это программный скрипт, который автоматически выполняет действия, необходимые для проверки функциональности программного обеспечения.

Приложение Б
ИСХОДНЫЙ КОД БАЗОВОГО ФУНКЦИОНАЛА
(рекомендуемое)

Листинг Б.1 – Исходный код базовых функций взаимодействия и проверок графического экрана эмулятора

```

import os
import string
import pyautogui
import pytesseract
import allure
import re
from rapidfuzz import fuzz, utils
from enum import Enum

ROOT = os.path.dirname(os.path.abspath(__file__))

pyautogui.PAUSE = 0.2
pyautogui.FAILSAFE = True

def get_emulator_window_region(min_region_search_time=10):
    """
    Находим регион по заголовку окна ts-emu
    :param min_region_search_time: максимальное время окна
    эмулятора на экране
    :return: Кортеж [int(x), int(y), int(width), int(height)]
    где x и y - координата левого верхнего угла;
    width - ширина области; height - высота области
    """

    os.makedirs(ROOT + "/log/screens/", exist_ok=True)

    try:
        # Ищем центр изображения заголовка окна
        ts_emu = pyautogui.locateCenterOnScreen(
            ROOT + "/screens/ts_emu.png",
            minSearchTime=min_region_search_time
        )
        # Вычисляем координаты региона экранчика
        region = (int(ts_emu.x - 73), int(ts_emu.y - 20), 330,
285)
        # Скриншот для логирования
        screenshot = pyautogui.screenshot(
            ROOT + "/log/screens/log_region.png", region=region
        )
        screenshot.save(ROOT + "/log/screens/test_region_tw.png")

        # Возвращаем координаты региона экранчика
        return region

    # Если не найдено изображение заголовка, пытаемся найти
    # изображение свернутого окна
    except pyautogui.ImageNotFoundException:
        try:

```

```

        ts_emu = pyautogui.locateCenterOnScreen(
            ROOT + "/screens/ts_emu_unfocused.png",
            minSearchTime=min_region_search_time,
        )
        region = (int(ts_emu.x - 73), int(ts_emu.y - 20), 330,
285)

        screenshot = pyautogui.screenshot(
            ROOT + "/log/screens/log_region.png",
region=region
        )
        screenshot.save(ROOT +
"/log/screens/test_region_tw.png")
        return region

    # В случае, когда не найдено и свернутое окно, возвращаем
None
    except pyautogui.ImageNotFoundException as ex:
        print(f"Не найдено окно эмулятора ts-emu!\nОшибка:
{ex}\n")
        return None

def is_image_visible(
    file: string,
    confidence=0.8,
    min_search_time=1,
    min_region_search_time=10,
    in_emu_window=True,
    log_func=print,
):
    """
    Ищет изображение на экране и возвращает координаты центра

    :param file - путь до файла изображения для поиска
    :param confidence - процент соответствия с изображением из
файла
    :param min_search_time - время ожидания элемента
    :param min_region_search_time - время ожидание окна эмулятора
    :param in_emu_window - флаг поиска внутри окна эмулятора
    """
    try:
        if in_emu_window:
            # Поиск координат центра элемента внутри окна
эмулятора
            target = pyautogui.locateOnScreen(
                image=file,
                confidence=confidence,
                minSearchTime=min_search_time,

region=get_emulator_window_region(min_region_search_time),
            )
        else:
            # Поиск координат центра элемента вне окна эмулятора

```

```

        target = pyautogui.locateOnScreen(
            image=file, confidence=confidence,
minSearchTime=min_search_time
        )
        # Возвращаем True и координаты элемента
        return True, target

    except pyautogui.ImageNotFoundException as ex:
        log_func(f"Не удалось найти {file} ошибка: {ex}")
        # Элемент не найден, возвращаем False
        return False, None

@allure.step("Проверка видимости на экране")
def check_visible(
    file: string,
    confidence=0.8,
    min_search_time=1,
    in_emu_window=True,
    log_func=print,
):
    is_visible = is_image_visible(
        file,
        confidence,
        min_search_time,
        in_emu_window=in_emu_window,
        log_func=log_func,
    )
    assert is_visible == True, log_func(
        "{} не найден на экране за {} секунд!".format(file,
min_search_time)
    )
    return is_visible

@allure.step("Проверка отсутствия элемента на экране")
def check_not_visible(
    file: string, confidence=0.8, min_search_time=1,
in_emu_window=True
):
    is_visible = is_image_visible(
        file, confidence, min_search_time,
in_emu_window=in_emu_window
    )
    assert is_visible == False, "{} найден на экране за {}
секунд!".format(
        file, min_search_time
    )
    return is_visible

def click(
    file: string,

```

```

confidence=0.8,
min_search_time=1,
repeat=1,
offset_x: int = None,
offset_y: int = None,
min_region_search_time=10,
log_func=print,
):
    """
    Кликает по элементу file
    либо кликает по координатам с отступом (offset_x; offset_y)
    от центра элемента file

    :param file - путь до файла изображения для поиска
    :param confidence - процент соответствия с изображением из
    файла
    :param min_search_time - время ожидания элемента
    :param repeat - количество нажатий
    :param offset_x - отступ по X
    :param offset_y - отступ по Y
    :param min_region_search_time - время ожидание окна эмулятора
    """

    # Проверяем, что элемент отображает на экране
    if check_visible(file, confidence, min_search_time,
log_func=log_func):
        target = pyautogui.locateCenterOnScreen(
            file,
            confidence=confidence,
            minSearchTime=min_search_time,

region=get_emulator_window_region(min_region_search_time),
        )

        # Получаем координаты центра элемента и по необходимости
        вычисляем отступ
        x, y = target.x, target.y
        if offset_x is not None:
            x = x + offset_x
        if offset_y is not None:
            y = y + offset_y

        # Нажимаем по полученным координатам
        pyautogui.click(x=x, y=y, clicks=repeat)

def click_one_in_many(file: string, item_in_list=0, repeat=1):
    list_images = list(pyautogui.locateAllOnScreen(file))
    pyautogui.click(list_images[item_in_list], clicks=repeat)

def click_arrow_left(min_time_search=10, repeat=1,
log_func=print):

```

```

for _ in range(repeat):
    click(
        ROOT + "/screens/arrow_left.png",
        confidence=0.9,
        min_search_time=min_time_search,
        log_func=log_func,
    )

def click_arrow_right(min_time_search=10, repeat=1,
confidence=0.9, log_func=print):
    for _ in range(repeat):
        click(
            ROOT + "/screens/arrow_right.png",
            confidence=confidence,
            min_search_time=min_time_search,
            log_func=log_func,
        )

def click_white_arrow_up(min_time_search=1, repeat=1):
    for _ in range(repeat):
        click(
            ROOT + "/screens/white_arrow_up.png",
            confidence=0.9,
            min_search_time=min_time_search,
            repeat=repeat,
        )

def click_white_arrow_down(min_time_search=1, repeat=1):
    for _ in range(repeat):
        click(
            ROOT + "/screens/white_arrow_down.png",
            confidence=0.9,
            min_search_time=min_time_search,
            repeat=repeat,
        )

class Patterns(Enum):
    NUMS = r"^[0-9]"
    CHAR = r"^[a-zA-Za-яA-Я]"
    NUMS_AND_CHARS = r"^[a-zA-Za-яA-Я0-9]"
    EMAIL = r"^[a-zA-Z0-9._%+-@]+"
    IP_ADDRESS = r"^[0-9.() ]"

# Распознавание текста с области экрана
def get_text_from_image(
    image_to_search: string,
    offset_x: int,
    offset_y: int,

```

```

width: int,
height: int,
confidence=0.7,
min_search_time=1,
save_screen_name="log_screen.png",
lang="rus+eng",
min_region_search_time=10,
config="--psm 7",
log_func=print,
):
    """
    Функция делает скриншот области и возвращает распознанный
    текст
    :param image_to_search - картинка, от центра которой будем
    делать отступ к полю текста
    :param offset_x - отступ по X
    :param offset_y - отступ по Y
    :param width - ширина поля
    :param height - высота поля
    :param lang - языки для распознавания текста
    :param config - конфигурация для tesseract (по умолчанию
    воспринимает текст как одну строку)

    :return - найденный текст или None
    """

    # Ищем изображение от координат которого будем отталкиваться
    if is_image_visible(
        image_to_search, confidence, min_search_time,
        log_func=log_func
    ):
        target = pyautogui.locateCenterOnScreen(
            image=image_to_search,
            confidence=confidence,

region=get_emulator_window_region(min_region_search_time),
        )

        # Делаем скриншот области где будем распознавать текст
        screenshot = pyautogui.screenshot(
            region=(int(target.x - offset_x), int(target.y -
offset_y), width, height)
        )
        os.makedirs(ROOT + "/log/screens/", exist_ok=True)
        screenshot.save(ROOT + "/log/screens/" + save_screen_name)

        # Возвращаем распознанный Tesseract'ом текст
        text_found = pytesseract.image_to_string(screenshot,
lang=lang, config=config)
        return text_found.strip("\n")

    # Если изображение не найдено - возвращаем None
    return None

```

```

# Возвращаем только текст, соответствующий паттерну из
Patterns(числа, строки, email и тд)
def get_only_pattern_from_image(
    pattern: Patterns,
    image_to_search: string,
    offset_x: int,
    offset_y: int,
    width: int,
    height: int,
    confidence=0.7,
    min_search_time=1,
    save_screen_name="log_screen.png",
    lang="rus+eng",
):
    text_found = get_text_from_image(
        image_to_search,
        offset_x,
        offset_y,
        width,
        height,
        confidence,
        min_search_time,
        save_screen_name,
        lang,
    )
    assert text_found is not None, "Текст на изображении {} не
найден!".format(
        image_to_search
    )
    return re.sub(pattern.value, "", text_found)L

@allure.step("Сравнение на схожесть найденной строки с ожидаемой")
def compare_strings(s1, s2, min_similarity=85.0, log_func=print):
    """
    :param s1 - строка для сравнения
    :param s2 - строка для сравнения
    :param min_similarity - минимальный процент схожести строк
    """
    # Сравнением строки
    similarity = fuzz.ratio(s1, s2,
processor=utils.default_process)
    log_func(f"Схожесть строки '{s1}' и '{s2}' = {similarity}")

    # Проверяем, что процент схожести не меньше ожидаемого
    assert similarity > min_similarity, \
        "Строки {} и {} недостаточно схожи! Коэффициент схожести:
{}".format(s1, s2, similarity)

@allure.step("Нажатие на кнопку - крестик закрытия окна ts-emu")

```

```
def click_close_emu_window():
    click(ROOT + "/screens/ts_emu.png", offset_x=230, offset_y=0)
    check_not_visible(ROOT + "/screens/ts_emu.png",
in_emu_window=False)
```

```
def white_arrow_right(min_time_search=1, log_func=print):
    click(
        ROOT + "/screens/white_arrow_right.png",
        confidence=0.9,
        min_search_time=min_time_search,
        log_func=log_func,
    )
```

```
def check_page(file: string, confidence=0.8, min_search_time=0.1,
log_func=print):
    is_visible = is_image_visible(file, confidence,
min_search_time, log_func=log_func)
    return is_visible
```

Приложение В
ИСХОДНЫЙ КОД АВТО-ТЕСТОВ
(рекомендуемое)

Листинг В.1 – Авто-тест настройки даты и времени на этапе первичной инициализации

```

@pytest.mark.first_init
@pytest.mark.order(4)
@allure.epic("Локальное администрирование")
@allure.feature("Первичная инициализация")
@allure.story("1063666 Выбор алгоритмов. Настройка даты и времени.")
@allure.title("Проверка на странице Дата и время")
@allure.testcase(
    "URL TO TESTS CASE"
)
def test_set_date_and_time(get_set_date_time_test_data):
    (
        timezone_check,
        timezone_input,
        timezone_valid,
        invalid_day,
        invalid_month,
        invalid_year,
    ) = (
        get_set_date_time_test_data["timezone_check"],
        get_set_date_time_test_data["timezone_input"],
        get_set_date_time_test_data["timezone_valid"],
        get_set_date_time_test_data["invalid_day"],
        get_set_date_time_test_data["invalid_month"],
        get_set_date_time_test_data["invalid_year"],
    )

    with allure.step('Проверка страницы "Дата и время"'):
        bf.click_arrow_right()
        date_and_time.check_page()

    with allure.step('Проверка доступных значений для поля "Часовой пояс"'):
        date_and_time.open_timezone_input()
        date_and_time.click_timezone_btn_more()

    with allure.step(
        f'Проверка установки в поле "Часовой пояс" значения: {timezone_input}'
    ):
        date_and_time.select_timezone(timezone_input)
        date_and_time.check_timezone_text(timezone_input)
        date_and_time.check_time_text(
            get_time_timezone(timezone_input), min_similarity=70.0
        )

    with allure.step(
        f'Проверка установки в поле "Часовой пояс" валидного значения: {timezone_valid}'
    ):

```

```

):
    date_and_time.open_timezone_input()
    date_and_time.click_timezone_btn_more()
    date_and_time.select_timezone(timezone_valid)
    date_and_time.check_time_text(
        get_time_timezone(timezone_valid), min_similarity=70.0
    )

    with allure.step(
        f'Проверка установки в поле "Дата" невалидной даты:
        {invalid_day}/{invalid_month}/{invalid_year}'
    ):
        date_and_time.open_date_input()
        date_and_time.input_date(invalid_day, invalid_month,
invalid_year)
        date_and_time.check_invalid_date_notification()
        bf.click_arrow_left()
        current_date_string = get_current_date_time()[0]
        date_and_time.check_date_text(current_date_string)
        bf.click_arrow_right()

```

Листинг В.2 – Авто-тест проверки аутентификации учетной записи после восстановления из резервной копии

```

@pytest.mark.authentication
@allure.epic("Локальное администрирование")
@allure.feature("Аутентификация ")
@allure.title("Авто-тест 1209191 [Л] Аутентификация после
восстановления из РК")
@allure.testcase(
    "URL TO TESTCASE"
)
def test_auth_after_reserve_copy(_setup_res_copy,
_get_auth_after_reserve_copy, request, get_admins_login_data,
teardown):
    ssh_addr = request.config.getoption("ssh_addr")
    copy_name = _get_auth_after_reserve_copy["res_copy_name"]

    with allure.step(f"Восстановление из резервной копии
{copy_name}"):

        side_selector.scroll_to_menu_option(restore_reserve_copy.Options.L
OCAL_VAULT)
        side_selector.click_ok()
        base_functions.click_arrow_right(repeat=3)
        list_selector.click_on_element_in_list(
            reserve_copy_list.select_reserve_copy_page_title,
copy_name
        )
        base_functions.click_arrow_right()
        sleep(10)

```

```

        reserve_copy_list.check_need_to_reboot_notification()
        base_functions.click_arrow_right()
        sleep(60)

    with allure.step("Проверка авторизации АСрЗИ после
восстановления"):
        run_emulator(ssh_addr)
        select_mode.click_admin_mode(min_search_time=30)
        login, key, pin = (
            get_admins_login_data["asrzi_login"],
            get_admins_login_data["asrzi_key"],
            get_admins_login_data["asrzi_pin"],
        )

    side_selector.scroll_to_menu_option(admin_menu.Options.AUTHENTICAT
ION)
        side_selector.click_ok()
        authentication.input_login_data(login=login, key=key,
pin=pin)
        base_functions.click_arrow_right()
        sleep(5)

    side_selector.scroll_to_menu_option(admin_menu.Options.INFO)
        side_selector.click_ok()

    side_selector.scroll_to_menu_option(information.Options.CURRENT_AD
MIN)
        side_selector.click_ok()
        assert
base_functions.compare_strings(current_admin.get_current_admin_tex
t(), "asrzi"), \
        "Имя текущего администратора не совпадает с ожидаемым"

```

Листинг В.3 – Авто-тест установки порта на странице настроек сетевых параметров

```

@pytest.mark.order(4)
@pytest.mark.settings
@allure.epic("Локальное администрирование")
@allure.feature("Настройки")
@allure.story("Авто-тест 1188941 Установка сетевых параметров")
@allure.title('Проверка установки значения в поле "Порт:"')
@allure.testcase(
    "URL TO TESTCASE"
)
def test_set_port(get_network_setting_test_data):
    port = get_network_setting_test_data["port"]
    with allure.step('Проверка установки пустой строки в поле
"Порт:"'):
        network_parameters.open_input_port()
        keyboard.click_delete(repeat=5)

```

```

        bf.click_arrow_right()
        network_parameters.check_invalid_port_notification()
        bf.click_arrow_left()
        assert not bf.compare_strings(
            network_parameters.get_port_text(), " "
        ), 'В поле "Порт:" установлена пустая строка!'

    with allure.step(f'Установка валидного значения: {port} в
поле "Порт:"'):
        network_parameters.open_input_port()
        keyboard.click_delete(repeat=5)
        keyboard.enter_data(port)
        bf.click_arrow_right()
        found_port = network_parameters.get_port_text()
        assert bf.compare_strings(
            found_port, port
        ), 'Найденное значение в поле "Порт:" : {} не
соответствует ожидаемому: {}'.format(
            found_port, port
        )

```

Листинг В.4 – Авто-тест удаления виртуального токена

```

@pytest.mark.order(7)
@pytest.mark.virt_tokens
@allure.epic("Локальное администрирование")
@allure.feature("Работа с виртуальными токенами")
@allure.title("Авто-тест 1179743 Удаление виртуального токена")
@allure.testcase(

    "https://tfs.infotecs.ru/tfs/infotecstcl1/PRG_CustomDev/_workitems/
edit/1179743"
)
def test_delete_virtual_token(
    select_admin_mode,
    login_as_asrzi_admin,
    open_virtual_tokens_page,
    get_virtual_token_test_data,
    delete_test_apstdts_acc,
):
    token_mark, delete_reason = (
        get_virtual_token_test_data["mark"],
        get_virtual_token_test_data["delete_reason"],
    )

    with allure.step(f"Удаление виртуального токена:
{token_mark}"):
        virtual_token.open_token_in_list_by_name(token_mark)
        side_selector.click_more_btn(min_search_time=5)
        virtual_token.open_token_operations()
        virtual_token.click_delete_token()

```

```
        with allure.step(f"Ввод причины удаления токена: {delete_reason}"):
            keyboard.enter_data(delete_reason)
            bf.click_arrow_right()

        virtual_token.check_token_deleted_notification()
        time.sleep(5)

    with allure.step(f"Проверка отсутствия в списке виртуального токена: {token_mark}"):
        assert not
        virtual_token.find_token_in_list_by_name(token_mark) [
            0
        ], "Удаленный токен с именем: {} найден на экране!".format(token_mark)
```

Приложение Г
ALLURE ОТЧЕТ О ПРОХОЖДЕНИИ ТЕСТИРОВАНИЯ ПО
(рекомендуемое)

Behaviors				
order	name	duration	status	
			Status:	2 0 99 0 0
Marks:				
<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>				
▼	Дистанционное администрирование		52	
>	Мониторинг параметров загрузки ПАК		1	
>	Работа с виртуальными токенами		11	
>	Регистрация событий		18	
>	Регламентные операции		4	
>	Служебные сертификаты		2	
>	Управление УЗ администраторов и пользователей		16	
▼	Локальное администрирование		2	45
>	Настройки		1	10
▼	Операции с ключевым носителем		2	
✓	#1 Авто-тест 1187454 [Л] Форматирование ключевого носителя	1m 00s		
✓	#2 Авто-тест 1309688 [Л] Форматирование собственного ключевого носителя	1s 819ms		
>	Первичная инициализация		8	
>	Работа с виртуальными токенами		7	
>	Резервное копирование		6	
>	Смена режима работы		4	
>	Управление отчетами аудита		3	
>	Управление УЗ администраторов и пользователей		1	5

Рисунок Г.1 – Группировка тестов по темам



DURATION

70

60

50

40

30

20

15

10

5

0

0s

20s

40s

1m

1m

1m

2m

2m

2m

3m

3m

3m

4m

4m

4m

5m

DURATION TREND

Рисунок Г.2 – Аналитика прохождения тестов в отчете

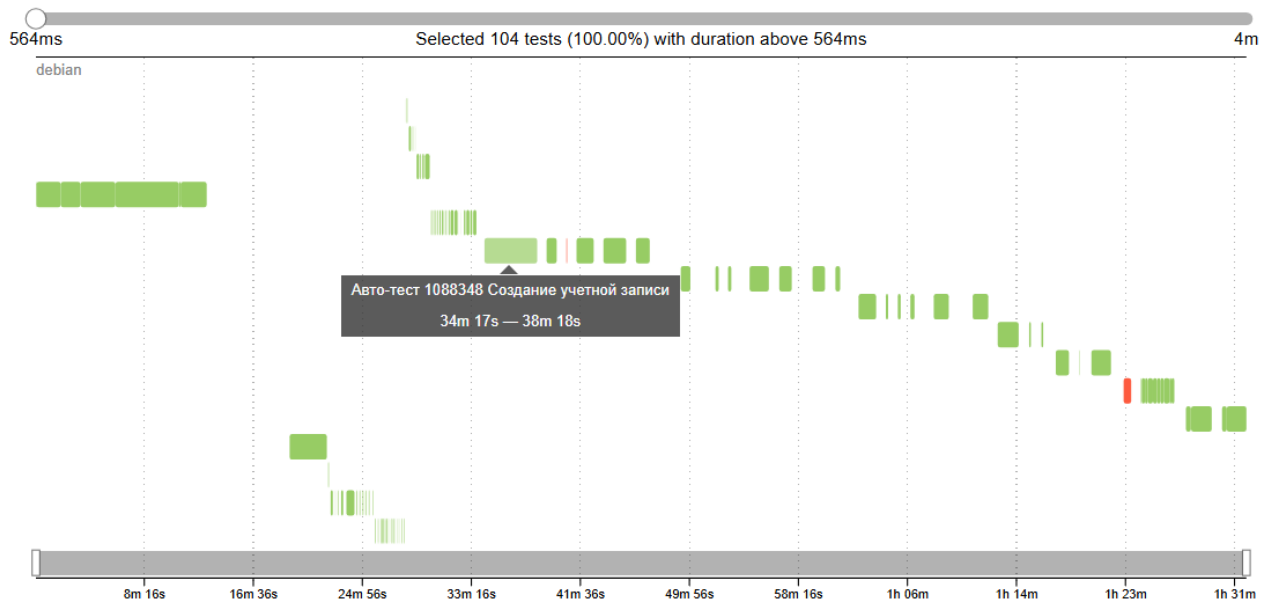


Рисунок Г.3 – Временной график прохождения авто-тестов

Failed

Авто-тест 1189561 Установка даты и времени

Overview

History

Retries

AssertionError: Найденный текст в блоке 'Время': 122253 не соответствует ожидаемому: 13:32

Tags: settings order(1)

Categories: Product defects

Severity: normal

Duration: 34s 692ms

Links

https://tfs.infotecs

Execution

> Set up

✓ Test body

> Проверка отображения страницы "Дата и время" 4 sub-steps268ms

> Проверка установки невалидной даты: 31/02/2024 в поле "Дата:" 7 sub-steps10s 988ms

> Проверка установки допустимой даты: 25/05/2025 9 sub-steps17s 063ms

✓ Проверка установки значения: (+ 03:00) Asia/Baghdad в поле "Часовой пояс" 11 sub-steps6s 372ms

> Нажатие на "Часовой пояс:" 1 sub-step527ms

> Нажатие на кнопку "..." 1 sub-step531ms

> Выбор часового пояса 1 parameter, 4 sub-steps2s 224ms

✓ Проверка, что текст в блоке "Время:" соответствует ожидаемому 2 parameters, 1 sub-step3s 061ms

time'13:32'

min_similarity85.0

> Сравнение на схожесть найденной строки с ожидаемой 4 parameters0s

AssertionError: Найденный текст в блоке 'Время': 122253 не соответствует ожидаемому: 13:32

Рисунок Г.4 – Описание шагов теста и аналитика ошибки

✓ Attach screenshot on test failure 1 attachment

37ms

Failed test screenshot

56 KiB



AssertionError: Найденный текст в блоке 'Время': 122253 не соответствует ожидаемому: 13:32

Рисунок Г.5 – Добавление скриншота ошибки в отчет