**Министерство образования Республики Беларусь**

**Учреждение образования**

**«Брестский Государственный технический университет»**

Лабораторная работа №7

По дисциплине «Технология проектирования программного обеспечения»

Тема: «Работа с конвейерами CI/CD на примере платформы GitHub Actions»

**Выполнил:**

студент 3 курса

группы АС-576

Пачко С.А.

**Проверил:**

старший преподаватель

кафедры ИИТ

Костомаров В.В.

Брест, 2025

Цель работы: приобрести базовые знания по настройке конвейеров сборки и тестирования CI/CD с помощью платформы GitHub Actions; применить   
на практике паттерны проектирования.

Задание 1.

Выбрать паттерн проектирования и записать свою фамилию напротив нужного паттерна в таблицу на вкладке со своей группой:

https://docs.google.com/spreadsheets/d/1a1pA2BjjiVv\_Dy4xw0\_m36l51scOLthZ3CtgvKNlSwo/edit?usp=sharing

Задание 2.

Создать персональный аккаунт GitHub (если его еще нет, сайт https://github.com/).

Создать git-репозиторий и коммиты по ходу написания программы   
и тестов.

Задание 3.

Настроить конвейер сборки и тестирования с помощью платформы GitHub Actions. Использовать событие push для сборки проекта и запуска тестов PyTest.

Задание 4.

Написать программу на любом языке программирования, используя выбранный паттерн, для прикладного примера из реального мира.

Переименовать классы паттерна в соответствии с прикладной областью выбранного примера.

Задание 5.

Написать минимум два теста программы для инструмента PyTest.

Задание 6.

Проверить работу GitHub Actions на вкладке Actions как для корректных тестов, так и для падающих тестов.

Ход работы:

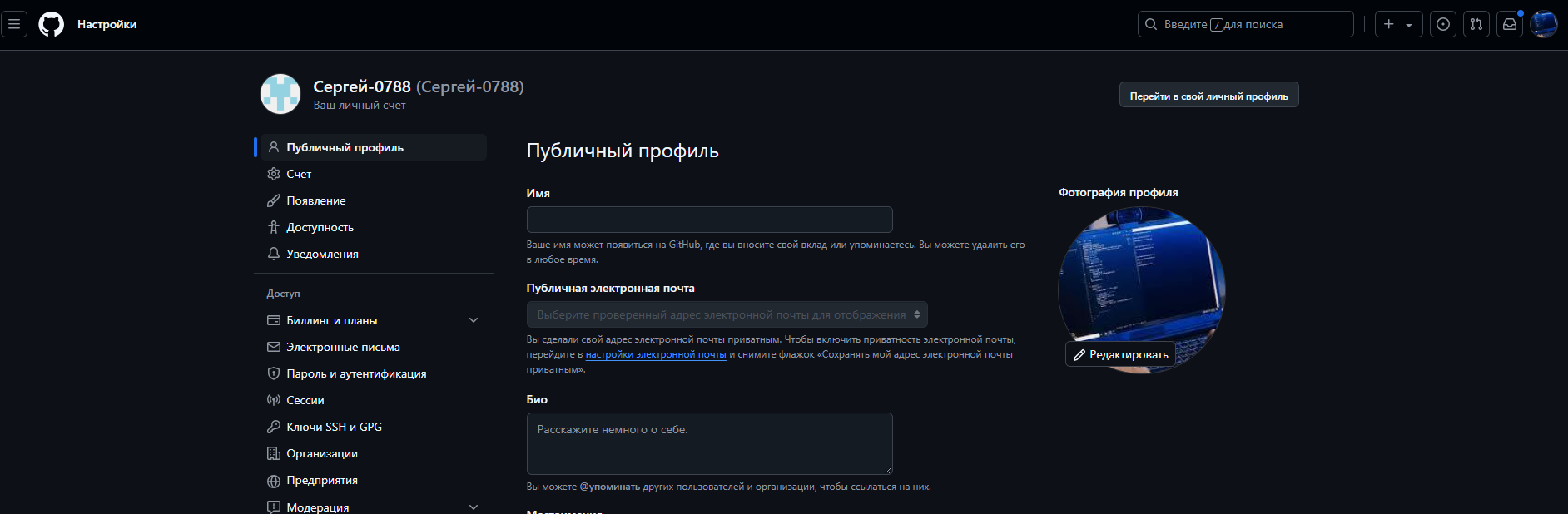
**Задание 1.**

Выбрали паттерн проектирования Builder и записали свою фамилию напротив нужного паттерна в таблицу на вкладке со своей группой.

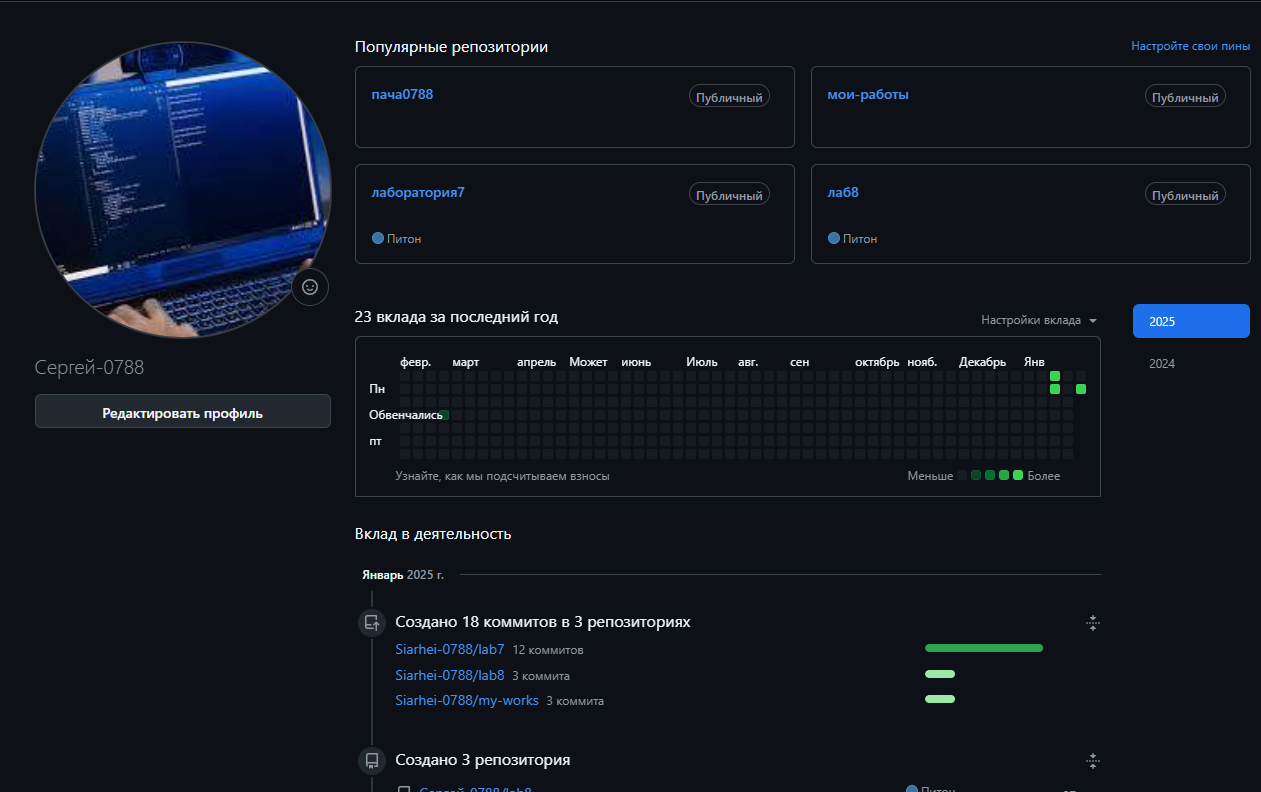
Builder – это шаблон проектирования, позволяющий создавать сложные объекты шаг за шагом. Шаблон позволяет создавать различные типы   
и представления объекта, используя тот же код построения.

**Задание 2.**

Создали персональный аккаунт GitHub (Siarhei-0788).



Создали git-репозиторий «lab8» и коммиты по ходу написания программы   
и тестов.



**Задание 3.**

Настроили конвейер сборки и тестирования с помощью платформы GitHub Actions. Использовали событие push для сборки проекта и запуска тестов PyTest

**Задание 4.**

Программа на python, использующая паттерн Builder.

Код программы:

**import** **curses**

# Класс Character, который мы будем создавать

**class** **Character**:

**def** **\_\_init\_\_**(self):

self.character\_class = None

self.weapon = None

self.armor = None

self.skills = []

**def** **\_\_str\_\_**(self):

skills\_str = ", ".join(self.skills) **if** self.skills **else** "нет навыков"

**return** (f"Персонаж: {self.character\_class}**\n**"

f"Оружие: {self.weapon}**\n**"

f"Броня: {self.armor}**\n**"

f"Навыки: {skills\_str}")

# Интерфейс Builder (Строитель)

**class** **CharacterBuilder**:

**def** **\_\_init\_\_**(self):

self.character = Character()

**def** **set\_class**(self, character\_class):

self.character.character\_class = character\_class

**return** self

**def** **set\_weapon**(self, weapon):

self.character.weapon = weapon

**return** self

**def** **set\_armor**(self, armor):

self.character.armor = armor

**return** self

**def** **add\_skill**(self, skill):

self.character.skills.append(skill)

**return** self

**def** **build**(self):

**return** self.character

# Класс Director (Директор), который управляет процессом создания

**class** **CharacterDirector**:

**def** **\_\_init\_\_**(self, builder):

self.builder = builder

**def** **create\_warrior**(self):

**return** self.builder.set\_class("Воин") \

.set\_weapon("Меч") \

.set\_armor("Латные доспехи") \

.add\_skill("Сила") \

.add\_skill("Защита") \

.build()

**def** **create\_mage**(self):

**return** self.builder.set\_class("Маг") \

.set\_weapon("Посох") \

.set\_armor("Мантия") \

.add\_skill("Огненный шар") \

.add\_skill("Ледяная стрела") \

.build()

# Функция для отображения персонажа

**def** **display\_character**(stdscr, character):

stdscr.clear() # Очищаем экран

stdscr.addstr("Созданный персонаж:**\n**", curses.color\_pair(**1**))

stdscr.addstr(str(character) + "**\n\n**", curses.color\_pair(**1**))

stdscr.addstr("Нажмите любую клавишу для продолжения...", curses.color\_pair(**1**))

stdscr.getch() # Ожидаем нажатия любой клавиши

# Функция для отображения текстового интерфейса

**def** **draw\_menu**(stdscr):

curses.curs\_set(**0**) # Скрываем курсор

stdscr.clear() # Очищаем экран

stdscr.refresh() # Обновляем экран

# Цвета

curses.start\_color()

curses.init\_pair(**1**, curses.COLOR\_CYAN, curses.COLOR\_BLACK)

curses.init\_pair(**2**, curses.COLOR\_YELLOW, curses.COLOR\_BLACK)

builder = CharacterBuilder()

director = CharacterDirector(builder)

current\_row = **0**

menu\_items = ["Создать Воина", "Создать Мага", "Выйти"]

**while** True:

stdscr.clear()

stdscr.addstr("Добро пожаловать в генератор персонажей RPG!**\n\n**", curses.color\_pair(**1**))

# Отображение элементов меню

**for** idx, item **in** enumerate(menu\_items):

**if** idx == current\_row:

stdscr.addstr(f"> {item}**\n**", curses.color\_pair(**2**))

**else**:

stdscr.addstr(f" {item}**\n**", curses.color\_pair(**1**))

key = stdscr.getch() # Получаем нажатие клавиши

# Обработка нажатия клавиш

**if** key == curses.KEY\_UP **and** current\_row > **0**:

current\_row -= **1**

**elif** key == curses.KEY\_DOWN **and** current\_row < len(menu\_items) - **1**:

current\_row += **1**

**elif** key == curses.KEY\_ENTER **or** key **in** [**10**, **13**]: # Enter

**if** current\_row == **0**:

character = director.create\_warrior()

display\_character(stdscr, character)

**elif** current\_row == **1**:

character = director.create\_mage()

display\_character(stdscr, character)

**elif** current\_row == **2**:

**break**

# Запуск программы

**def** **main**():

curses.wrapper(draw\_menu) # Обертываем функцию draw\_menu в curses.wrapper для корректной работы с curses

**print**("Спасибо за использование генератора персонажей!")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

В коде паттерн Builder применен для создания сложного объекта *Character* пошагово. Паттерн позволяет отделить процесс создания объекта   
от его представления, что делает код более гибким и поддерживаемым.

4.1. Класс *Character* представляет собой объект, который мы хотим создать. Он содержит несколько атрибутов:

character\_class (класс персонажа, например, "Воин" или "Маг");

weapon (оружие);

armor (броня);

skills (список навыков).

4.2. Класс CharacterBuilder отвечает за пошаговое создание объекта Character. Он предоставляет методы для установки различных атрибутов персонажа:

set\_class – устанавливает класс персонажа;

set\_weapon – устанавливает оружие;

set\_armor – устанавливает броню;

add\_skill – добавляет навык.

Каждый метод возвращает self, что позволяет использовать цепочку вызовов (method chaining).

4.3. Класс CharacterDirector управляет процессом создания объекта Character. Он использует CharacterBuilder для создания конкретных типов персонажей (например, воина или мага). Директор инкапсулирует логику создания, что упрощает повторное использование кода.

4.4. Графический интерфейс (CharacterCreatorApp) использует CharacterDirector для создания персонажей. Когда пользователь нажимает кнопку "Создать Воина" или "Создать Мага", вызываются соответствующие методы, которые создают объект Character и отображают его в текстовом поле.

**Задание 5. Тест программ.**

Код теста:

**import** **pytest**

**from** **abc** **import** ABC, abstractmethod

# Определение интерфейсов

**class** **ICharacterBuilder**(ABC):

**@abstractmethod**

**def** **set\_class**(self, character\_class):

**pass**

**@abstractmethod**

**def** **set\_weapon**(self, weapon):

**pass**

**@abstractmethod**

**def** **set\_armor**(self, armor):

**pass**

**@abstractmethod**

**def** **add\_skill**(self, skill):

**pass**

**@abstractmethod**

**def** **build**(self):

**pass**

**class** **ICharacterDirector**(ABC):

**@abstractmethod**

**def** **create\_warrior**(self):

**pass**

**@abstractmethod**

**def** **create\_mage**(self):

**pass**

# Реализация классов

**class** **Character**:

**def** **\_\_init\_\_**(self):

self.character\_class = None

self.weapon = None

self.armor = None

self.skills = []

**def** **\_\_str\_\_**(self):

skills\_str = ", ".join(self.skills) **if** self.skills **else** "нет навыков"

**return** (f"Персонаж: {self.character\_class}**\n**"

f"Оружие: {self.weapon}**\n**"

f"Броня: {self.armor}**\n**"

f"Навыки: {skills\_str}")

**class** **CharacterBuilder**(ICharacterBuilder):

**def** **\_\_init\_\_**(self):

self.character = Character()

**def** **set\_class**(self, character\_class):

self.character.character\_class = character\_class

**return** self

**def** **set\_weapon**(self, weapon):

self.character.weapon = weapon

**return** self

**def** **set\_armor**(self, armor):

self.character.armor = armor

**return** self

**def** **add\_skill**(self, skill):

self.character.skills.append(skill)

**return** self

**def** **build**(self):

**return** self.character

**class** **CharacterDirector**(ICharacterDirector):

**def** **\_\_init\_\_**(self, builder):

self.builder = builder

**def** **create\_warrior**(self):

**return** self.builder.set\_class("Воин") \

.set\_weapon("Меч") \

.set\_armor("Латные доспехи") \

.add\_skill("Сила") \

.add\_skill("Защита") \

.build()

**def** **create\_mage**(self):

**return** self.builder.set\_class("Маг") \

.set\_weapon("Посох") \

.set\_armor("Мантия") \

.add\_skill("Огненный шар") \

.add\_skill("Ледяная стрела") \

.build()

# Фикстуры для тестов

**@pytest.fixture**

**def** **builder**():

**return** CharacterBuilder()

**@pytest.fixture**

**def** **director**(builder):

**return** CharacterDirector(builder)

# Тесты для проверки интерфейсов

**def** **test\_builder\_implements\_interface**(builder):

**assert** isinstance(builder, ICharacterBuilder)

**def** **test\_director\_implements\_interface**(director):

**assert** isinstance(director, ICharacterDirector)

# Тесты для проверки создания персонажей

**def** **test\_create\_warrior**(director):

character = director.create\_warrior()

**assert** character.character\_class == "Воин"

**assert** character.weapon == "Меч"

**assert** character.armor == "Латные доспехи"

**assert** "Сила" **in** character.skills

**assert** "Защита" **in** character.skills

**def** **test\_create\_mage**(director):

character = director.create\_mage()

**assert** character.character\_class == "Маг"

**assert** character.weapon == "Посох"

**assert** character.armor == "Мантия"

**assert** "Огненный шар" **in** character.skills

**assert** "Ледяная стрела" **in** character.skills

# Тест для проверки строкового представления персонажа

**def** **test\_character\_str**():

character = Character()

character.character\_class = "Воин"

character.weapon = "Меч"

character.armor = "Латные доспехи"

character.skills = ["Сила", "Защита"]

expected\_output = (

"Персонаж: Воин**\n**"

"Оружие: Меч**\n**"

"Броня: Латные доспехи**\n**"

"Навыки: Сила, Защита"

)

**assert** str(character) == expected\_output

В данном коде присутствуют следующие тесты:

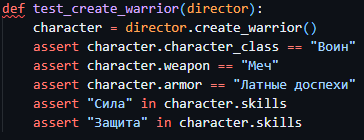
тест на проверку реализации интерфейса ICharacterBuilder, который проверяет, что объект builder является экземпляром класса, реализующего интерфейс ICharacterBuilder;



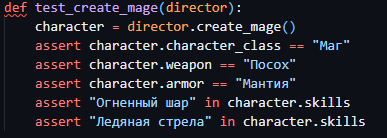
тест на проверку реализации интерфейса ICharacterDirector, который проверяет, что объект director является экземпляром класса, реализующего интерфейс ICharacterDirector;



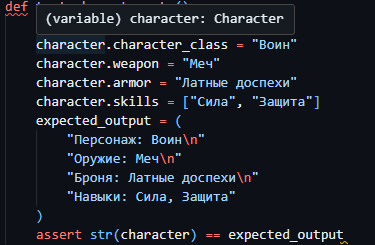
тест на создание персонажа типа "Воин", который проверяет, что метод create\_warrior корректно создает персонажа с классом "Воин", оружием "Меч", броней "Латные доспехи" и навыками "Сила" и "Защита";



тест на создание персонажа типа "Маг", который проверяет, что метод create\_mage корректно создает персонажа с классом "Маг", оружием "Посох", броней "Мантия" и навыками "Огненный шар" и "Ледяная стрела";

**

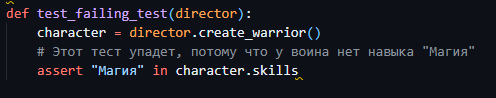
тест на проверку строкового представления персонажа, который проверяет, что метод \_\_str\_\_ класса Character корректно возвращает строковое представление персонажа.



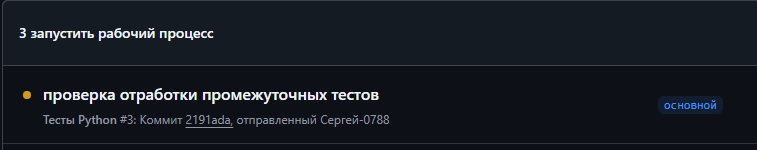
**Задание 6.**

Проверка работы GitHub Actions.

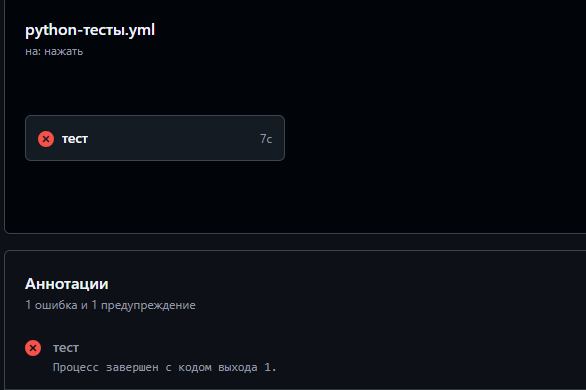
Чтобы проверить, как GitHub Actions обрабатывает падающие тесты, добавим заведомо неправильный тест:



Запушили изменения и перешли во вкладку Actions.



Видим, что тест завершился с ошибкой.



**Вывод:**

В рамках лабораторной работы была изучена и успешно реализована настройка конвейера непрерывной интеграции и доставки (CI/CD)   
с использованием платформы GitHub Actions. Основной целью было автоматизировать процесс тестирования кода и обеспечить его стабильность при каждом изменении в репозитории. В ходе работы были выполнены следующие шаги:

1. Изучение GitHub Actions

GitHub Actions – это мощный инструмент для автоматизации задач, таких как тестирование, сборка и развертывание. Он позволяет создавать workflows (рабочие процессы), которые выполняются при определенных событиях (например, push или pull request).

Были изучены основные компоненты GitHub Actions:

Workflows: YAML-файлы, описывающие последовательность задач.

Jobs: Отдельные задачи, которые выполняются в рамках workflow.

Steps: Шаги внутри job, такие как установка зависимостей или запуск тестов.

Actions: Готовые блоки для выполнения часто используемых задач (например, установка Python).

2. Настройка CI/CD для проекта

Для проекта был создан workflow, который автоматически запускает тесты при каждом пуше в ветку main или при создании pull request.

Конфигурация workflow была описана в файле .github/workflows/python-tests.yml. Основные шаги включали:

Клонирование репозитория.

Установку Python и зависимостей (например, pytest).

Запуск тестов с помощью команды pytest.

3. Тестирование кода

Были написаны модульные тесты для проверки корректности работы классов Character, CharacterBuilder и CharacterDirector.

Тесты проверяли:

Корректность создания персонажей (воина и мага).

Соответствие интерфейсам ICharacterBuilder и ICharacterDirector.

Строковое представление персонажа.

Для проверки работы CI/CD были добавлены как корректные тесты, так и заведомо падающие тесты. Это позволило убедиться, что GitHub Actions корректно обрабатывает как успешные, так и неудачные выполнения.

4. Анализ результатов

При успешном выполнении всех тестов GitHub Actions показывал статус Success и зеленую галочку ✅.

При наличии ошибок в тестах workflow завершался с статусом Failure,   
а в логах отображалась подробная информация о том, какой тест упал и почему.

Это позволило оперативно находить и исправлять ошибки в коде.

5. Преимущества использования GitHub Actions

Автоматизация: Тесты запускаются автоматически, что экономит время и снижает вероятность человеческой ошибки.

Прозрачность: Логи выполнения доступны в реальном времени, что упрощает отладку.

Интеграция: GitHub Actions легко интегрируется с другими инструментами, такими как линтеры, системы сборки и развертывания.

Гибкость: Возможность настраивать workflows под конкретные задачи проекта.

6. Рекомендации по улучшению

Добавить проверку покрытия кода тестами с использованием библиотеки pytest-cov.

Внедрить линтеры (например, flake8 или black) для проверки стиля кода.

Расширить тестирование на несколько версий Python, чтобы убедиться   
в совместимости кода.

Настроить автоматическое развертывание приложения после успешного прохождения тестов.

В конечном итоге работа с GitHub Actions позволила успешно автоматизировать процесс тестирования. Использование CI/CD значительно упрощает разработку, так как ошибки выявляются на ранних этапах, а процесс интеграции изменений становится более предсказуемым и надежным.

Таким образом, проделанная работа демонстрирует важность   
и преимущества внедрения CI/CD в современной разработке программного обеспечения.