**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:

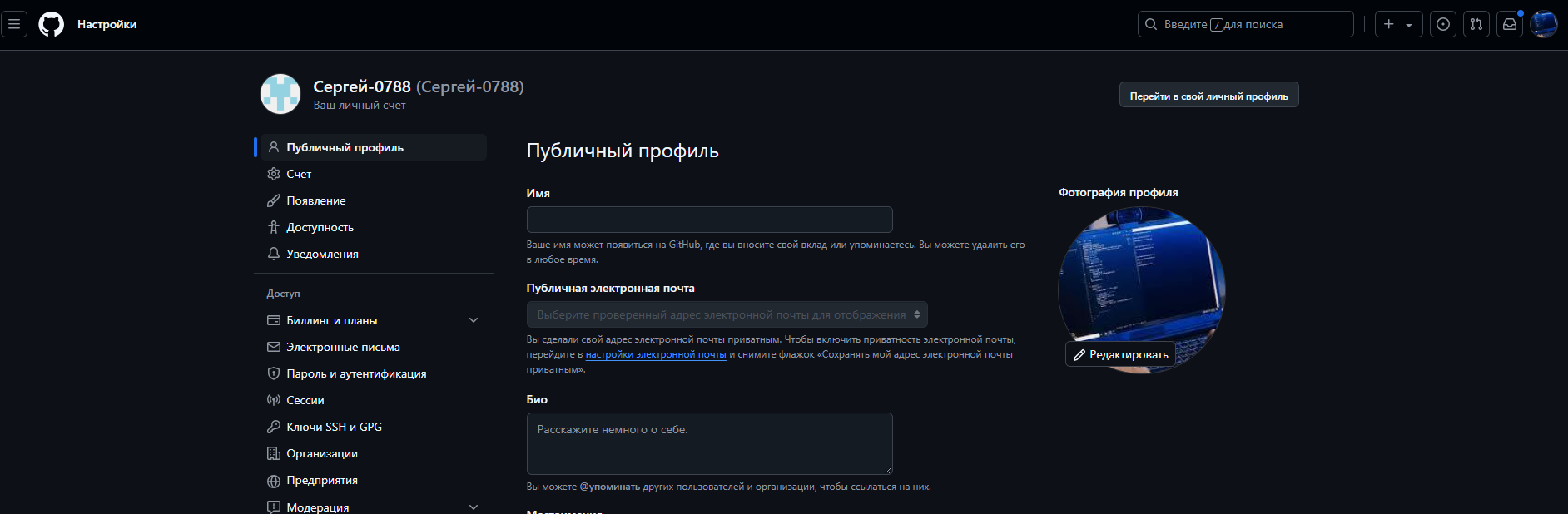
**Задание 1.**

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

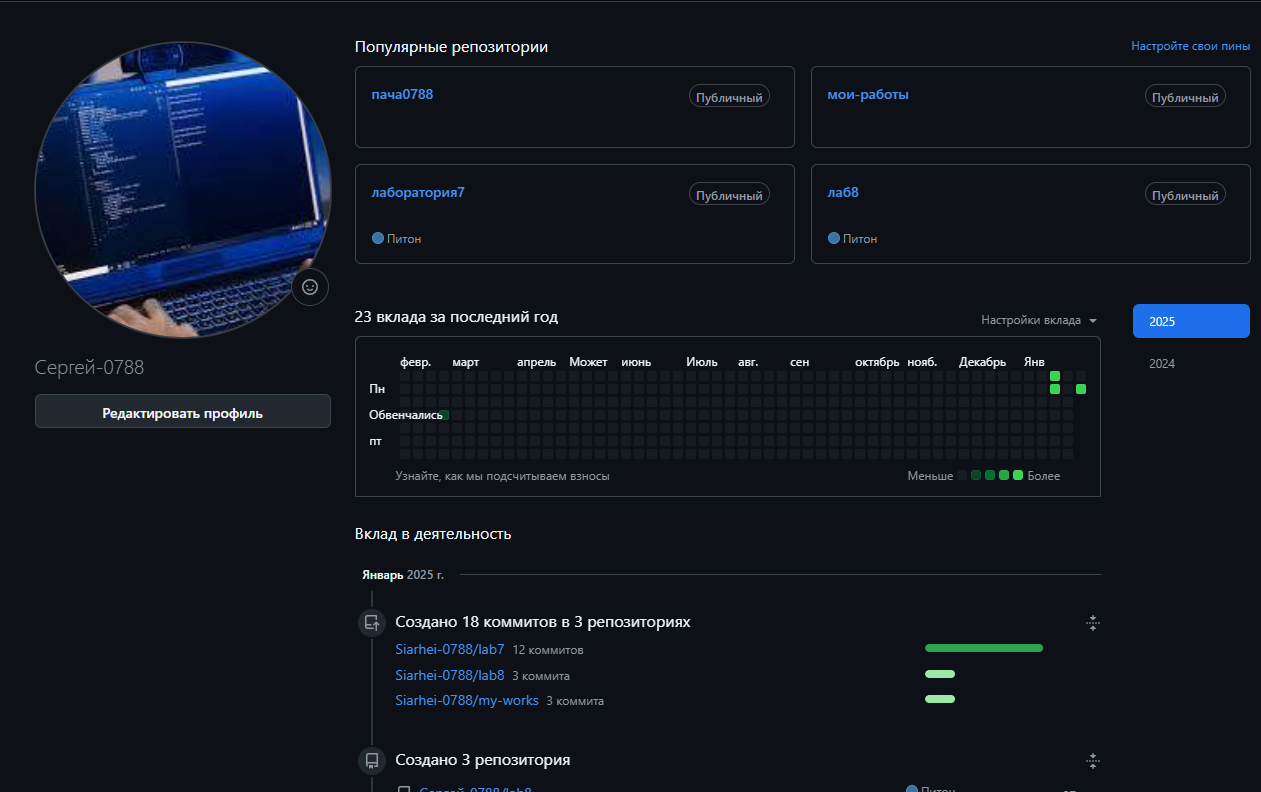
Builder – это шаблон проектирования, позволяющий создавать сложные объекты шаг за шагом. Шаблон позволяет создавать различные типы   
и представления объекта, используя тот же код построения.

**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** **pytest** # Импортируем библиотеку pytest для тестирования

**from** **selenium** **import** webdriver # Импортируем webdriver для управления браузером

**from** **selenium.webdriver.common.by** **import** By # Импортируем методы для поиска элементов на странице

# Определяем фикстуру для инициализации и завершения работы браузера

**@pytest.fixture**(scope="class")

**def** **browser**():

url = "https://belarusbank.by/" # URL страницы, к которой хотим обратиться

browser = webdriver.Chrome() # Инициализируем драйвер Chrome

browser.get(url) # Открываем указанный URL

**yield** browser # Передаем управление браузером в тесты

browser.quit() # Закрываем браузер после завершения тестов

# Определяем класс тестов

**class** **Test**:

# Определяем тест, который должен потенциально завалиться (xfail) и относится к smoke тестам

**@pytest.mark.xfail**

**@pytest.mark.smoke**

**def** **test\_check\_name**(self, browser):

info\_block = browser.find\_element(By.CLASS\_NAME, "page-footer-bottom\_\_col") # Находим элемент по классу

bb = "Беларусбанк" # Ожидаемое название организации

# Проверяем, что ожидаемое название есть в тексте найденного элемента

**assert** (

bb **in** info\_block.text

), f"Название организации - {bb}, полученный - {info\_block.text}"

# Определяем тест, который будет пропущен (skip) и относится к регрессионным тестам

**@pytest.mark.skip**

**@pytest.mark.regression**

**def** **test\_check\_phone**(self, browser):

# Находим ссылку на раздел "О компании" и переходим по ней

about = browser.find\_element(

By.XPATH, "/html/body/nav[1]/div/div/div/div[2]/a"

).get\_attribute("href")

browser.get(about)

# Находим все элементы списка контактов

about\_block = browser.find\_elements(By.CLASS\_NAME, "dot-line-list\_\_item")

phone = about\_block[-**2**].text # Извлекаем текст предпоследнего элемента

chek\_number = "+375 (17) 218-84-31" # Ожидаемый номер телефона

# Проверяем, что ожидаемый номер телефона находится в тексте найденного элемента

**assert** (

chek\_number **in** phone

), f"Номер мобильного телефона для частных клиентов - {chek\_number}, полученный - {phone}"

# Определяем параметризованный тест для проверки на разных языках

**@pytest.mark.parametrize**("language", ["ru", "en"])

**def** **test\_check\_adr**(self, browser, language):

**if** language == "ru":

# Проверка на русском языке

info\_block = browser.find\_element(By.CLASS\_NAME, "page-footer-bottom\_\_col")

addr\_assert = "г.Минск, пр.Дзержинского, 18" # Ожидаемый адрес на русском языке

# Проверяем, что ожидаемый адрес находится в тексте найденного элемента

**assert** (

addr\_assert **in** info\_block.text

), f"Адресс организации - {addr\_assert}, полученный - {info\_block.text}"

**elif** language == "en":

# Проверка на английском языке

browser.get("https://belarusbank.by/en") # Переходим на английскую версию сайта

info\_block = browser.find\_element(By.CLASS\_NAME, "page-footer-bottom\_\_col")

addr\_assert = "220089, Minsk, Dzerzhinski ave, 18, Belarus" # Ожидаемый адрес на английском языке

# Проверяем, что ожидаемый адрес находится в тексте найденного элемента

**assert** (

addr\_assert **in** info\_block.text

), f"Адресс организации - {addr\_assert}, полученный - {info\_block.text}"

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:

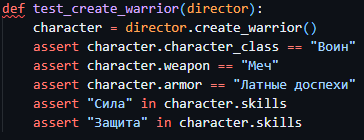
тест на проверку реализации интерфейса ICharacterBuilder, который проверяет, что объект builder является экземпляром класса, реализующего интерфейс ICharacterBuilder;



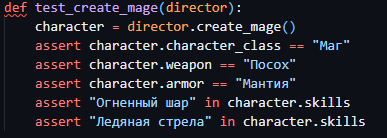
тест на проверку реализации интерфейса ICharacterDirector, который проверяет, что объект director является экземпляром класса, реализующего интерфейс ICharacterDirector;



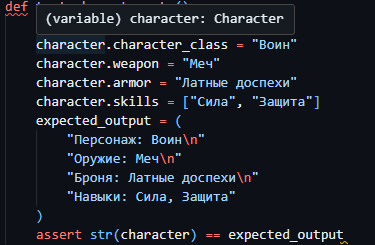
тест на создание персонажа типа "Воин", который проверяет, что метод create\_warrior корректно создает персонажа с классом "Воин", оружием "Меч", броней "Латные доспехи" и навыками "Сила" и "Защита";



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.



Вывод:

В данной лабораторной работе мы успешно выполнили задачи по тестированию веб-страницы с использованием библиотеки pytest и Selenium. В ходе работы выполнено:

1. инициализация драйвера Selenium: Мы использовали библиотеку Selenium для автоматизации браузера. Драйвер Google Chrome был инициализирован для доступа к сайту Беларусбанк;

2. создание фикстуры для браузера: Мы создали фикстуру browser, которая открывает сайт и предоставляет управление браузером для выполнения тестов. После завершения тестов браузер автоматически закрывался;

3. тестирование элементов на странице: Были созданы тесты для проверки различных элементов на веб-странице:

проверка наличия названия "Беларусбанк" в нижнем колонтитуле.

проверка правильности номера телефона для частных клиентов.

проверка адреса на русском и английском языках.

4. использовали метки @pytest.mark.xfail, @pytest.mark.smoke, @pytest.mark.skip и @pytest.mark.regression для управления тестами:

xfail для тестов, которые ожидаемо могут не пройти.

smoke для проверки базовой функциональности.

skip для пропущенных тестов.

regression для проверки существующего функционала.

Параметризация тестов позволила запускать тесты с разными значениями параметров, такими как язык сайта.

4. запустили тесты с использованием pytest, и тесты проверили различные элементы на веб-странице. Результаты тестов показывают, соответствуют   
ли данные элементы ожидаемым значениям.

В конечном итоге, лабораторная работа продемонстрировала возможность использования библиотек pytest и Selenium для тестирования веб-страниц.   
Мы научились автоматизировать браузерные действия, искать и проверять элементы на странице, а также сохранять результаты тестов.