Московский государственный технический университет им. Н.Э. Баумана

Факультет Радиотехнический
Кафедра РТ5

Курс «Программирование на основе классов и шаблонов»

Отчет по лабораторной работе №5-6
«Шаблоны проектирования и модульное тестирование в Python»

Выполнил:

студент группы РТ5-31Б:
Коровин К.С.

Подпись и дата:

Проверил:

преподаватель каф. ИУ5
Гапанюк Ю.Е.

Подпись и дата:

Москва, 2023

Описание задания

1. Необходимо для произвольной предметной области реализовать от одного до трех шаблонов проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать следующий каталог. Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
2. В модульных тестах необходимо применить следующие технологии:
    - TDD - фреймворк.
    - BDD - фреймворк.
    - Создание Mock-объектов.

Текст программы

1. Фабричный метод с тестом TTD

```
from abc import ABC, abstractmethod
import unittest

# Тест для Фабричного метода
class TestFactoryMethod(unittest.TestCase):
    def test_book_creation(self):
        creator = BookStore()
        product = creator.create_product()
        self.assertIsInstance(product, Book)
        self.assertEqual(product.display_info(), "Book: The Great
Gatsby by F. Scott Fitzgerald")


# Продукт - Товар в интернет-магазине
class Product(ABC):
    @abstractmethod
    def display_info(self):
        pass

# Конкретный продукт - Книга
class Book(Product):
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def display_info(self):
        return f"Book: {self.title} by {self.author}"

# Конкретный продукт - Электронное устройство
class ElectronicDevice(Product):
    def __init__(self, name, brand):
        self.name = name
        self.brand = brand

    def display_info(self):
        return f"Electronic Device: {self.brand} {self.name}"

# Создатель - Интернет-магазин
class OnlineStore(ABC):
    @abstractmethod
    def create_product(self):
        pass

    def sell_product(self):
```

```
        product = self.create_product()
        result = f"Online Store sold: {product.display_info()}"
        return result

# Конкретный создатель - Интернет-магазин книг
class BookStore(OnlineStore):
    def create_product(self):
        return Book("The Great Gatsby", "F. Scott Fitzgerald")

# Конкретный создатель - Интернет-магазин электронных устройств
class ElectronicsStore(OnlineStore):
    def create_product(self):
        return ElectronicDevice("Smartphone", "Samsung")

# Использование
book_store = BookStore()
book_result = book_store.sell_product()
print(book_result)

electronics_store = ElectronicsStore()
electronics_result = electronics_store.sell_product()
print(electronics_result)

unittest
```

2. Адаптер метод с тестом BDD

```
from abc import ABC, abstractmethod
import unittest
from unittest.mock import MagicMock

# Тест для Адаптера
class TestAdapter(unittest.TestCase):
    def test_payment_adapter(self):
        # Создаем Mock-объект для стороннего сервиса
        third_party_service_mock = MagicMock()
        third_party_service_mock.process_payment.return_value = "Mocked
payment processed successfully"

        # Используем Mock-объект в адаптере
        adapter = PaymentAdapter(third_party_service_mock)

        # Проверяем, что метод адаптера вызывает метод Mock-объекта
        result = adapter.process_payment()
        self.assertEqual(result, "Adapter: Mocked payment processed
successfully")
        third_party_service_mock.process_payment.assert_called_once()


# Существующий класс стороннего сервиса для обработки платежей
class ThirdPartyPaymentService:
    def process_payment(self):
        return "Payment processed successfully by third-party service"

# Целевой интерфейс нашей системы платежей
class PaymentSystem(ABC):
    @abstractmethod
    def process_payment(self):
        pass

# Адаптер для интеграции стороннего сервиса с нашей системой
class PaymentAdapter(PaymentSystem):
    def __init__(self, third_party_service):
        self.third_party_service = third_party_service
```

```
    def process_payment(self):
        return f"Adapter: {self.third_party_service.process_payment()}"

# Использование
third_party_service = ThirdPartyPaymentService()
payment_adapter = PaymentAdapter(third_party_service)

result = payment_adapter.process_payment()
print(result)

unittest
```

3. Наблюдатель с тестом создание Mock-объектов

```
from abc import ABC, abstractmethod
import unittest
from unittest.mock import Mock

# Тест для Наблюдателя
class TestObserver(unittest.TestCase):
    def test_shopping_cart_observers(self):
        cart = ShoppingCart()

        # Создаем Mock-объекты для наблюдателей
        observer1 = Mock()
        observer2 = Mock()

        # Добавляем Mock-объекты в корзину покупок в качестве
наблюдателей
        cart.add_observer(observer1)
        cart.add_observer(observer2)

        # Добавляем товар в корзину и проверяем, что уведомления
отправлены каждому наблюдателю
        cart.add_item("Shoes")
        observer1.update.assert_called_once_with("Item added: Shoes")
        observer2.update.assert_called_once_with("Item added: Shoes")

        # Удаляем товар и проверяем, что уведомления отправлены каждому
наблюдателю
        cart.remove_item("Shoes")
        observer1.update.assert_called_with("Item removed: Shoes")
        observer2.update.assert_called_with("Item removed: Shoes")
# Наблюдатель
class Observer(ABC):
    @abstractmethod
    def update(self, message):
        pass

# Субъект - Корзина покупок
class ShoppingCart:
    def __init__(self):
        self._observers = []
        self._items = []

    def add_observer(self, observer):
        self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, message):
        for observer in self._observers:
```

```
                observer.update(message)

    def add_item(self, item):
        self._items.append(item)
        self.notify_observers(f"Item added: {item}")

    def remove_item(self, item):
        if item in self._items:
            self._items.remove(item)
            self.notify_observers(f"Item removed: {item}")

    def display_items(self):
        return self._items

# Конкретный наблюдатель - Клиент магазина
class Customer(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received message: {message}")

# Использование
cart = ShoppingCart()

customer1 = Customer("Alice")
customer2 = Customer("Bob")

cart.add_observer(customer1)
cart.add_observer(customer2)

cart.add_item("Laptop")
cart.add_item("Headphones")

cart.remove_item("Laptop")

unittest
```

Экранные формы с примерами выполнения программы

1. Factory Method.py

```
Ran 1 test in 0.001s

OK
Launching unittests with arguments python -m unittest /Users/bogdan/PycharmProjec

Online Store sold: Book: The Great Gatsby by F. Scott Fitzgerald
Online Store sold: Electronic Device: Samsung Smartphone
```

2. Adapter.py

```
Testing started at 21:57 ...
Launching unittests with arguments python -m unittest /Users/bogdan/Pyc

Adapter: Payment processed successfully by third-party service


Ran 1 test in 0.001s

OK
```

3. Observer.py

```
Testing started at 21:57 ...
Launching unittests with arguments python -m unittest /Users/bogdan/



Ran 1 test in 0.001s

OK
Alice received message: Item added: Laptop
Bob received message: Item added: Laptop
Alice received message: Item added: Headphones
Bob received message: Item added: Headphones
Alice received message: Item removed: Laptop
Bob received message: Item removed: Laptop
```