**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: |  | Проверил: |
| студент группы РТ5-31Б:  Коровин К.С. |  | преподаватель каф. ИУ5  Гапанюк Ю.Е. |
|  |  |  |
| Подпись и дата: |  | 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.](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) 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

1. Адаптер метод с тестом BDD

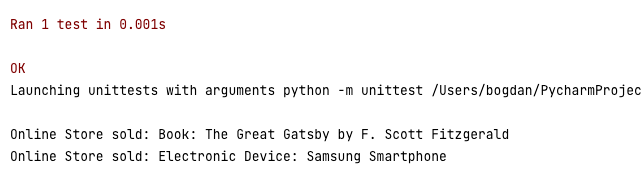
from abc import ABC, abstractmethod  
import unittest  
from unittest.mock import MagicMock  
  
# Тест для Адаптера  
class TestAdapter(unittest.TestCase):  
 def test\_payment\_adapter(self):  
 # Создаем Mock-объект для стороннего сервиса  
 third\_party\_service\_mock = MagicMock()  
 third\_party\_service\_mock.process\_payment.return\_value = "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

1. Наблюдатель с тестом создание Mock-объектов

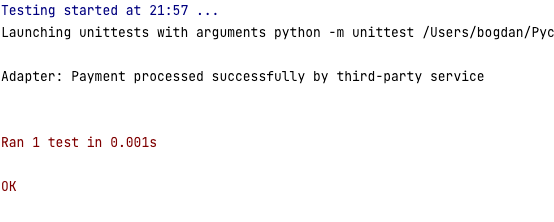
from abc import ABC, abstractmethod  
import unittest  
from unittest.mock import Mock  
  
# Тест для Наблюдателя  
class TestObserver(unittest.TestCase):  
 def test\_shopping\_cart\_observers(self):  
 cart = ShoppingCart()  
  
 # Создаем Mock-объекты для наблюдателей  
 observer1 = Mock()  
 observer2 = 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



1. Adapter.py



1. Observer.py

