МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра «Вычислительные системы и технологии»

**Технологии разработки программных систем**

Отчет

по лабораторной работе №2

тема:

«Разработка По с использованием паттернов проектирования»

Вариант №8

«Интернет-магазин электронной техники»

ПРОВЕРИЛ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сапожников В.О

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Рыжов А.С.

23-ИВТ-5

Нижний Новгород 2025г

Описание перечня DAO классов и наборов операций

**BaseDAO** — это класс, который предоставляет базовые CRUD-операции (Create, Read, Update, Delete) для всех DAO-классов. Он служит родительским классом для всех специализированных DAO, уменьшая дублирование кода. BaseDAO реализует стандартные методы работы с сущностями базы данных:

- Создание (create) — добавление новой записи.

- Чтение (get\_all, get\_by\_id) — получение записей.

- Обновление (update) — изменение существующей записи.

- Удаление (delete) — удаление записи по ID. Каждый дочерний DAO-класс наследует эти методы и может расширять их.

|  |  |
| --- | --- |
| DAO | методы |
| BrandDAO | def find\_by\_country(self, country: str): - поиск брендов по странам |
| CategoryDAO | find\_by\_category(self, category\_name: str) -> Category | None: - поиск по категории  def get\_create\_category(self, category\_name: str) -> Category: - получение категории, а если её не существует - создание |
| CustomerDAO | get\_info\_by\_email(self, email) -> Customer | None: - получение информации о пользователе (имя, фамилия, телефон, почта, дата регисстрации) по почте |
| OrderCartDAO | get\_order\_cart\_by\_id(self, id: int): - получение корзины по id  get\_cart\_items(self, order\_id: int) -> List[OrderCart]: - получить все товары заказа |
| OrderDAO | get\_products\_by\_category(self, category\_name: str) -> list[Product]: - получение продуктов по называнию категории  get\_all\_orders(self) -> list[Order]: - получение всех заказов  add\_product\_to\_order(self, order\_id: int, product\_id: int, quantity: int, price: float) -> None: - добавляет товар в заказ |
| PaymentMethodDAO | show\_payment(self) -> list[PaymentMethod]: - показать доступные методы оплаты |
| ProductDAO | create\_product(……) – создание товара  def search\_by\_name(self, name: str) -> list[Product]: - поиск продуктов по названию  create\_product\_category(self, product\_name: str, category\_name, price, \*\*kwargs) -> Product: - создает продукт с категорией  get\_all\_products(self) -> list[Product]: - возвращает все товары |
| StatusDAO | get\_orders\_by\_status(self, status: str) -> list[Order]: - получить все заказы по статусу |
|  |  |

**Листинг кода DAO класса (OrderDAO)**

from sqlalchemy.orm import Session

from models.category import Category

from models.product import Product

from .base\_dao import BaseDAO

from models.order import Order

class OrderDAO(BaseDAO):

    def \_\_init\_\_(self, session: Session):

        super().\_\_init\_\_(session, Order)

def get\_products\_by\_category(self, category\_name: str) -> list[Product]:

        # получение продуктов по категории

        return (self.session.query(Product)

                .join(Category)

                .filter(Category.name.ilike(f'%{category\_name}%'))

                .all()

                )

def get\_all\_orders(self) -> list[Order]:

        # получение всех заказов

        return self.session.query(Order).all()

def add\_product\_to\_order(self, order\_id: int, product\_id: int, quantity: int, price: float) -> None:

        # добавление продукта в заказ

        """

        Добавляет товар в заказ

            order\_id: ID заказа

            product\_id: ID товара

            quantity: Количество

            price: Цена на момент заказа

        """

        from models.order\_cart import OrderCart

        # Проверяем существование заказа и товара

        order = self.session.get(Order, order\_id)

        if not order:

            raise ValueError(f"Заказ с ID {order\_id} не найден")

        product = self.session.get(Product, product\_id)

        if not product:

            raise ValueError(f"Товар с ID {product\_id} не найден")

        # Проверяем доступное количество

        if product.quantity < quantity:

            raise ValueError(f"Недостаточно товара {product.name} на складе")

        # Создаем связь между заказом и товаром

        order\_item = OrderCart(

            order\_id=order\_id,

            product\_id=product\_id,

            quantity=quantity,

            price=price  # Сохраняем цену на момент заказа

        )

        self.session.add(order\_item)

        self.session.commit()

        # Уменьшаем количество товара на складе

        product.quantity -= quantity

        self.session.add(product)

        self.session.commit()

**Логика методов:**

* **- def get\_products\_by\_category(self, category\_name: str) -> list[Product]: -** получение продуктов по категории
* **def get\_all\_orders(self) -> list[Order]: -** получение всех заказов
* **def add\_product\_to\_order(self, order\_id: int, product\_id: int, quantity: int, price: float) -> None: -** добавить конкретный товар к конкретному заказу

**Описание реализации паттерна «State»**

1. **Базовый абстрактный класс OrderState, который определяет базовый интерфейс для всех состояний.**
2. from abc import ABC, abstractmethod
3. class OrderState(ABC):
4. @property
5. @abstractmethod
6. def name(self) -> str:
7. pass
9. @abstractmethod
10. def next(self, order: 'Order', session: 'Session'):
11. pass
13. @abstractmethod
14. def prev(self, order: 'Order', session: 'Session'):
15. pass

**2. Конкретные состояния:**

**from patterns.state.order\_state import OrderState**

**from models.status import Status**

**from sqlalchemy.orm import Session**

**from models.order import Order**

**class ProvingState(OrderState):**

**def next(self, order: Order, session: Session) -> None:**

**status = session.query(Status).filter\_by(status="paid").first()**

**if not status:**

**raise ValueError("Статус 'paid' не найден")**

**order.status\_id = status.status\_id**

**from .concrete\_states import PaidState**

**order.\_state = PaidState()**

**session.commit()**

**def prev(self, order: Order, session: Session) -> None:**

**raise ValueError("Нельзя отменить заказ в статусе 'prooving'")**

**def name(self) -> str:**

**return "prooving"**

**class PaidState(OrderState):**

**def next(self, order: Order, session: Session) -> None:**

**status = session.query(Status).filter\_by(status="shipped").first()**

**if not status:**

**raise ValueError("Статус 'shipped' не найден")**

**order.status\_id = status.status\_id**

**from .concrete\_states import ShippedState**

**order.\_state = ShippedState()**

**session.commit()**

**def prev(self, order: Order, session: Session) -> None:**

**status = session.query(Status).filter\_by(status="prooving").first()**

**if not status:**

**raise ValueError("Статус 'prooving' не найден")**

**order.status\_id = status.status\_id**

**from .concrete\_states import ProvingState**

**order.\_state = ProvingState()**

**session.commit()**

**def name(self) -> str:**

**return "paid"**

**class ShippedState(OrderState):**

**def next(self, order: Order, session: Session) -> None:**

**raise ValueError("Заказ уже доставлен")**

**def prev(self, order: Order, session: Session) -> None:**

**status = session.query(Status).filter\_by(status="paid").first()**

**if not status:**

**raise ValueError("Статус 'paid' не найден")**

**order.status\_id = status.status\_id**

**from .concrete\_states import PaidState**

**order.\_state = PaidState()**

**session.commit()**

**def name(self) -> str:**

**return "shipped"**

**ProvingState** – может перевести заказ в следующий статус – paid

**PaidState –** может перевести заказ в следующий (shipped) и предыдущий(prooving)

ShippedState – может перевести заказ только в предыдущее состояние

**Описание реализации паттерна «Strategy»**

это поведенческий паттерн проектирования, который определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

1. **Класс-контекст.**
2. from .strategy import PaymentStrategy
3. class PaymentType:
4. def \_\_init\_\_(self, strategy: PaymentStrategy) -> None:
5. self.strategy = strategy
6. def set\_strategy(self, strategy: PaymentStrategy) -> None:
7. self.strategy = strategy
8. def pay(self, amount: float) -> str:
9. return self.strategy.pay(amount)

Использует стратегии через композицию, а не наследование. Также он можнт динамически менять стратегию.

2. конкретные стратегии

from abc import ABC, abstractmethod

# абстрактная стратегия

class PaymentStrategy(ABC):

    @abstractmethod

    def pay(self, amount: float) -> str:

        pass

# конкретная стратегия: оплата картой

class CardPayment(PaymentStrategy):

    def pay(self, amount: float) -> str:

        return f'Оплата по карте на сумму: {amount} руб.'

# конкретная стратегия: оплата в рассрочку

class InstallmentPlanPayment(PaymentStrategy):

    def pay(self, amount: float) -> str:

        return f'Оплата по рассрочке на сумму: {amount} руб.'

class CashByDelivery(PaymentStrategy):

    def pay(self, amount: float) -> str:

        return f'Оплата наличными при получении на сумму: {amount} руб.'

PaymentStrategy – абстрактный класс, интерфейс стратегии.

CardPayment, CashByDelivery, InstallmentPlanPayment – конкретные стратегии

**Описание реализации паттерна «Observer»**

это поведенческий паттерн проектирования, который используется, когда один объект должен автоматически уведомлять другой объект о любых изменениях его состояния. Этот паттерн реализует принцип "один ко многим" и позволяет объекту (называемому наблюдаемый или subject) уведомлять подписанные на него объекты (наблюдатели) об изменениях, не раскрывая своей внутренней структуры.

1. **Базовый абстрактный класс**
2. from abc import ABC, abstractmethod
3. from typing import List
4. class Observer(ABC):
5. @abstractmethod
6. def update(self, order\_id: int, status: str):
7. pass

**2. субъект**

class Subject(ABC):

    def \_\_init\_\_(self):

        self.\_observers: List[Observer] = []

    def attach(self, observer: Observer) -> None:

        # подписать наблюдателя

        if observer not in self.\_observers:

            self.\_observers.append(observer)

    def detach(self, observer: Observer):

        # отписать наблюдателя

        self.\_observers.remove(observer)

    def notify(self, order\_id: int, status: str) -> None:

        # уведомить других наблюдателей

        for observer in self.\_observers:

            observer.update(order\_id, status)

**3. конкретные наблюдатели:**

# конкретный наблюдатель - email

class EmailNotification(Observer):

    def update(self, order\_id, status) -> None:

        print(f'email: заказ {order\_id} получил статус "{status}"')

# конкретный наблюдатель - номер телефона

class SMSNotification(Observer):

    def update(self, order\_id, status) -> None:

        print(f'SMS: заказ {order\_id} получил статус "{status}"')

**Описание реализации паттерна «Facade»**

from datetime import datetime

from dao.orders\_dao import OrderDAO

from dao.products\_dao import ProductDAO

from dao.customers\_dao import CustomerDAO

from patterns.observer.observer import EmailNotification

from patterns.strategy.strategy import CardPayment

from patterns.strategy.strategy import CashByDelivery

from patterns.strategy.strategy import InstallmentPlanPayment

class OrderFacade:

    def \_\_init\_\_(self, session):

        self.session = session

        self.order\_dao = OrderDAO(session)

        self.product\_dao = ProductDAO(session)

        self.customer\_dao = CustomerDAO(session)

        self.payment\_dao = CardPayment()

        self.notifier = EmailNotification()

        # strategy

        self.\_payment\_strategies = {

            'card': CardPayment(),

            'cash': CashByDelivery(),

            'installpayment': InstallmentPlanPayment()

        }

        self.\_current\_strategy = self.\_payment\_strategies['card']  # по умолчанию

    def place\_order(self, customer\_id: int, product\_ids: list[int], address: str) -> bool:

        """Оформление заказа в один клик.

        Шаги:

        1. Проверить наличие товаров

        2. Создать заказ в БД

        3. Выполнить оплату

        4. Отправить уведомление"""

        try:

            products = []

            total\_amount = 0.0

            # 1

            for product\_id in product\_ids:

                product = self.product\_dao.get\_by\_id(product\_id)

                if product.quantity < 1:

                     raise ValueError(f"Товар {product\_id} закончился")

                products.append(product)

                total\_amount += float(product.price)

            # 2 создание заказа

            order\_data = {

                'customer\_id': customer\_id,

                'order\_date': datetime.today(),

                'total\_amount': total\_amount,

                'address': 'Красносельская д. 3',

                'status\_id': 1,  # Статус "Новый"

                'payment\_method\_id': 1

            }

            order = self.order\_dao.create(\*\*order\_data)

            # 3 Добавление товара в заказ

            for product in products:

                self.order\_dao.add\_product\_to\_order(

                    order\_id=order.order\_id,

                    product\_id=product.product\_id,

                    quantity=1,

                    price=product.price

                )

            return True

        except Exception as e:

            self.session.rollback()

            print(f"[Ошибка] {str(e)}")

            return False

    def add\_product\_by\_name(self, product\_name: str, quantity: int):

        """Добавление товара по названию"""

        product = self.product\_dao.get\_by\_name(product\_name)

        if not product:

            raise ValueError(f"Товар '{product\_name}' не найден")

        return self.add\_to\_cart(product.product\_id, quantity)  # Используем ID внутри

    def set\_payment\_method(self, method: str):

        self.\_current\_strategy = self.\_payment\_strategies.get(method)

        if not self.\_current\_strategy:

            raise ValueError(f'Неизвестный метод оплаты: {method}')

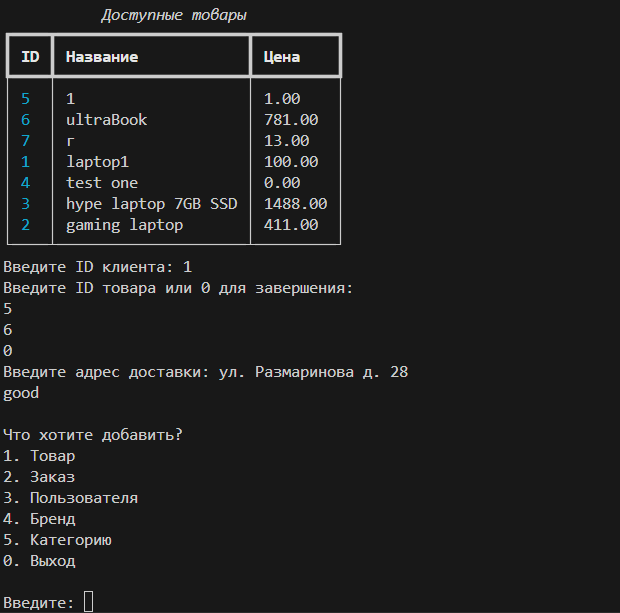
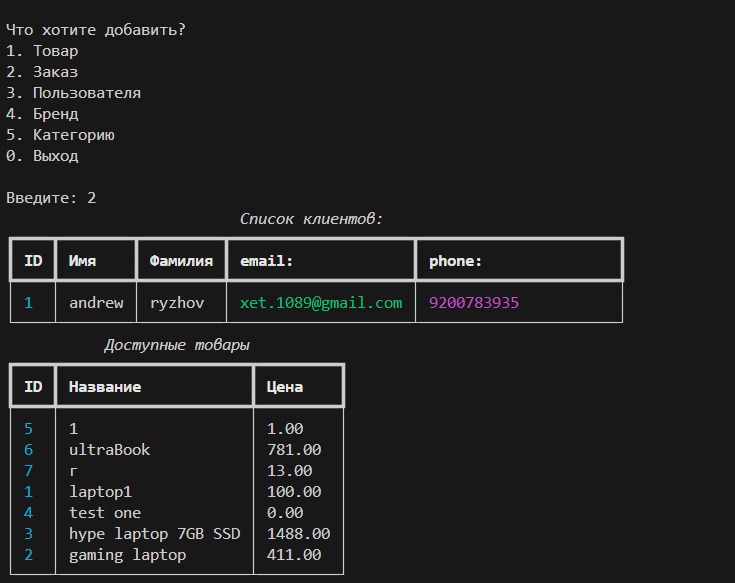
**Фасад** был использован для реализации создания заказа. По одной кнопке и выбору клиента и товаров, создается заказ с базовыми значениями

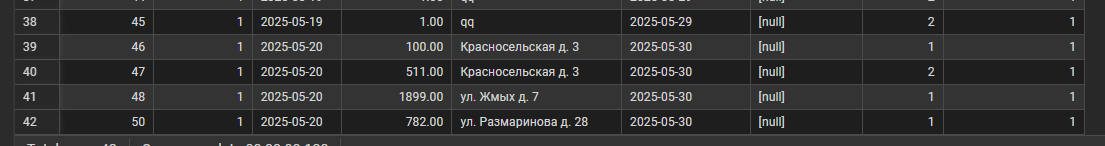
**Примеры использования паттернов**

**Façade:**

При выборе соответствующей кнопки меню можно создать заказ. В самом начале пользователю предоставляется таблица со всеми клиентами (и их ID), а также таблица со всеми доступными товарами.

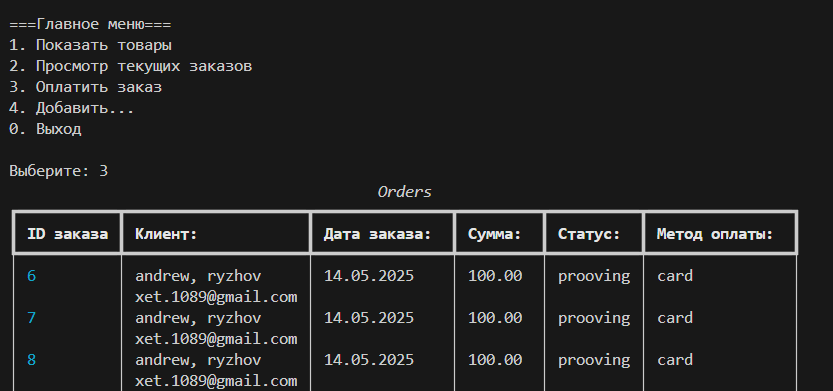
Пользователь вводит ID пользователя ,ID товаров, адрес доставки.



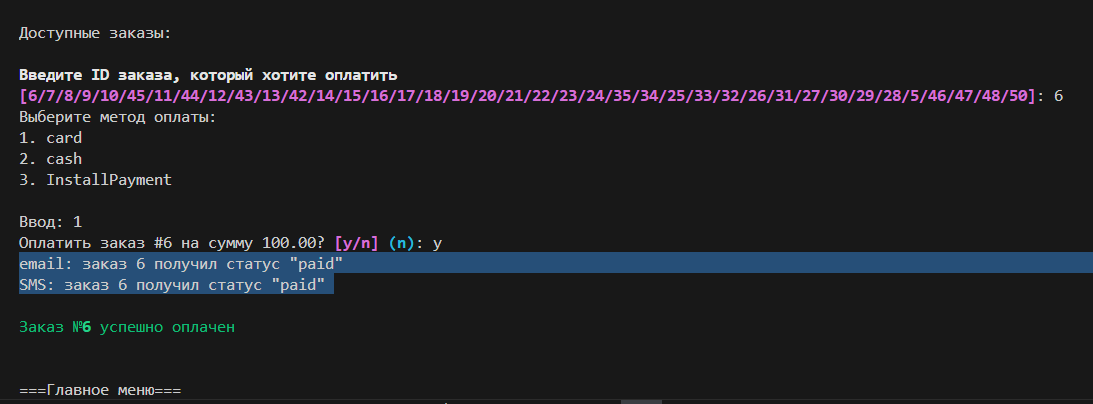


**State**

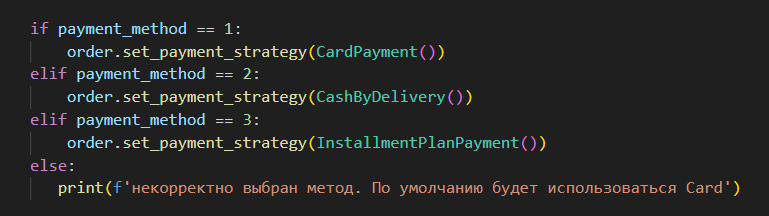
При нажатии кнопки «Оплатить заказ» предлагается таблица со всеми заказами, после чего пользователь выбирает ID нужного ему заказа



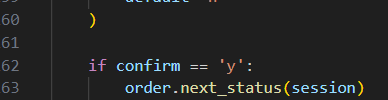
Strategy



После выбора нужного пользователю заказа, его спрашивают о методе оплаты. Выбор метода реализуется через стратегию:

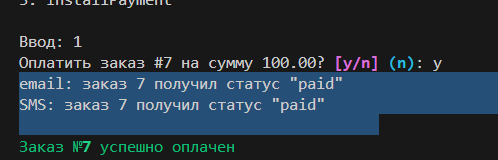


Смена статуса происходит с помощью паттерна State. Метод next\_status()



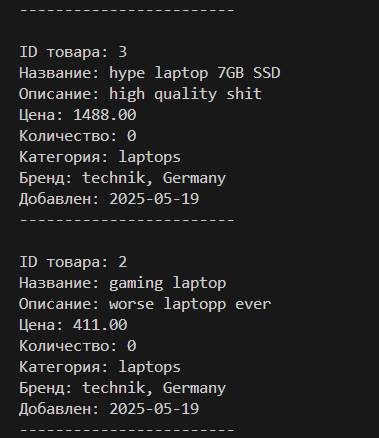
Observer

Во время оплаты создаются наблюдатели, на них подписывается объект – конкретный заказ order. Во время оплаты вызывается метод update()



На скриншоте выделенный текст – это работа наблюдателя.

При выборе кнопки «показать товары» выполняется метод ProductDAO.get\_all\_products().



При выборе кнопки «показать заказы» выполняется метод OrderDAO. get\_all\_orders().

