**MINISTERUL EDUCAŢIEI AL REPUBLICII MOLDOVA**

**UNIVERSITATEA TEHNICĂ A MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICǍ ȘI MICROELECTRONICǍ**

**DEPARTAMENTUL INGINERIA SOFTWARE ȘI AUTOMATICĂ**

# RAPORT

Lucrarea de laborator nr. 3

**Disciplina :** Tehnici și mecanisme de proiectare software

***A efectuat:*** st.gr. TI–202 F/r

Medelean Iulia

***A verificat:*** asist.univ.

Poștaru Andrei

***Chișinău 2024***

***Цель работы***

Реализовать 3 поведенческих шаблона на ваш выбор.

***Ход работы***

Шаблоны проектирования — это методические рекомендации по решению часто возникающих проблем в разработке программного обеспечения. Они представляют собой проверенные временем подходы к решению типовых задач, с которыми разработчики сталкиваются на регулярной основе. Важно понимать, что шаблоны проектирования — это не готовые классы, пакеты или библиотеки, которые можно просто подключить к приложению и ожидать автоматического решения всех проблем. Вместо этого, они описывают, как следует подходить к проектированию системы, чтобы наиболее эффективно справиться с определёнными вызовами.

Шаблоны проектирования можно рассматривать как руководство, которое помогает разработчикам принимать обоснованные решения при проектировании архитектуры программных систем. Эти шаблоны обобщают лучшие практики и предлагают стандартные решения, что позволяет улучшить читаемость и поддержку кода, а также упростить процесс разработки.

Шаблон проектирования, или паттерн, в разработке программного обеспечения — это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Такие шаблоны позволяют разработчикам использовать проверенные решения, избегая распространённых ошибок и снижая сложность разработки.

Примеры шаблонов проектирования включают в себя:

1. Порождающие шаблоны (например, Singleton, Factory Method, Abstract Factory) — они помогают управлять процессом создания объектов.
2. Структурные шаблоны (например, Adapter, Composite, Proxy) — они определяют способы создания отношений между объектами, чтобы упростить структуру системы.
3. Поведенческие шаблоны (например, Observer, Strategy, Command) — они описывают взаимодействие между объектами, помогая распределить обязанности и ответственность.

Использование шаблонов проектирования способствует созданию более гибких и расширяемых систем, что особенно важно в условиях быстро меняющихся требований и технологий.

Поведенческие паттерны проектирования (Behavioral Design Patterns) представляют собой решения типовых задач, связанных с взаимодействием объектов и распределением обязанностей между ними. Эти паттерны помогают организовать обмен данными между объектами и управлять их взаимодействием таким образом, чтобы система оставалась гибкой и легко расширяемой. Ниже приведены основные поведенческие паттерны с их подробным описанием:

* **Chain of Responsibility (Цепочка обязанностей)**

Этот паттерн позволяет передавать запрос последовательно по цепочке потенциальных обработчиков, пока один из них не обработает запрос. Он уменьшает зависимость отправителя запроса от его получателя, давая нескольким объектам шанс обработать запрос.

* **Command (Команда)**

Паттерн команд инкапсулирует запрос как объект, позволяя таким образом параметризовать клиентские объекты с различными запросами, ставить запросы в очередь или журналировать их. Это способствует разделению ответственности между инициатором и исполнителем запроса.

* **Interpreter (Интерпретатор)**

Паттерн интерпретатор определяет грамматику для языка и интерпретирует предложения этого языка. Этот паттерн полезен для разработки систем, работающих с формальными языками.

* **Iterator (Итератор)**

Паттерн итератор предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

* **Mediator (Посредник)**

Паттерн посредник определяет объект, который инкапсулирует способ взаимодействия множества объектов, снижая количество связей между ними. Он позволяет центрально управлять взаимодействиями между объектами.

* **Memento (Хранитель)**

Паттерн хранитель позволяет сохранять и восстанавливать прошлое состояние объекта без нарушения инкапсуляции. Он полезен для реализации операций отмены и повтора.

* **Observer (Наблюдатель)**

Паттерн наблюдатель определяет зависимость типа "один ко многим" между объектами так, что при изменении состояния одного объекта все зависимые от него оповещаются и обновляются автоматически.

* **State (Состояние)**

Паттерн состояние позволяет объекту изменять свое поведение при изменении его внутреннего состояния, создавая впечатление, что изменился сам объект.

* **Strategy (Стратегия)**

Паттерн стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Он позволяет алгоритмам изменяться независимо от клиентов, которые их используют.

* **Template Method (Шаблонный метод)**

Паттерн шаблонный метод определяет основу алгоритма и позволяет подклассам переопределять определенные шаги алгоритма, не изменяя его структуры.

* **Visitor (Посетитель)**

Паттерн посетитель позволяет добавлять новые операции к объектам, не изменяя их классы.

Он отделяет алгоритм от структуры объекта.

Каждый из этих паттернов решает специфические проблемы, связанные с поведением объектов и их взаимодействием, и помогает сделать систему более гибкой и легко изменяемой.

***Observer***

Паттерн проектирования "Наблюдатель" (Observer) относится к категории поведенческих паттернов. Этот паттерн позволяет одному объекту (наблюдателю) следить за состоянием другого объекта (субъекта) и автоматически получать уведомления об изменениях его состояния. Это обеспечивает слабую связанность между субъектом и наблюдателями и позволяет легко добавлять или удалять наблюдателей.

Паттерн "Наблюдатель" определяет зависимость "один ко многим" между объектами. Субъект ведет список наблюдателей и уведомляет их о любых изменениях своего состояния, обычно вызывая метод **update()** у каждого наблюдателя.

**Участники паттерна**

1. **Субъект (Subject)**: Объект, за состоянием которого следят. Он содержит методы для добавления, удаления и уведомления наблюдателей.
2. **Наблюдатель (Observer)**: Интерфейс или абстрактный класс, который определяет метод **update()**, вызываемый субъектом для уведомления наблюдателей об изменениях.
3. **Конкретный Субъект (Concrete Subject)**: Реализация субъекта. Хранит состояние и уведомляет наблюдателей об изменениях.
4. **Конкретный Наблюдатель (Concrete Observer)**: Реализация наблюдателя. Содержит логику, выполняемую в ответ на уведомление от субъекта.

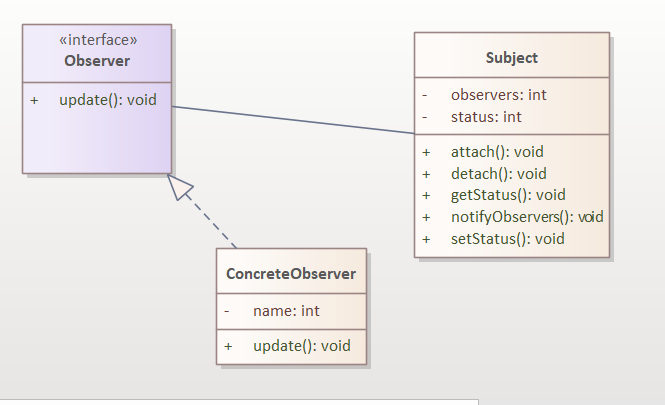


Рисунок 1 – Диаграмма реализации паттерна Наблюдателя

**Observer**: Интерфейс, определяющий метод **update()**, который используется для оповещения наблюдателей об изменении состояния.

**Subject**: Класс, представляющий наблюдаемый объект. Он содержит список наблюдателей, методы для добавления/удаления наблюдателей и метод для оповещения наблюдателей о изменениях.

**ConcreteObserver**: Конкретный наблюдатель, который реализует интерфейс **Observer**. В данном случае, это класс представляет собой клиента, который отслеживает статус заказа.

interface Observer {

    void update(String status);

}

class Subject {

    private List<Observer> observers = new ArrayList<>();

    private String status;

    public void attach(Observer observer) {

        observers.add(observer);

    }

    public void detach(Observer observer) {

        observers.remove(observer);

    }

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(status);

        }

    }

    public void setStatus(String status) {

        this.status = status;

        notifyObservers();

    }

    public String getStatus() {

        return status;

    }

}

class ConcreteObserver implements Observer {

    private String name;

    public ConcreteObserver(String name) {

        this.name = name;

    }

    @Override

    public void update(String status) {

        System.out.println("Customer " + name + ": Order status updated to: " + status);

    }

}

**Observer** - это интерфейс, который должен реализовывать любой класс, желающий получать уведомления о изменениях состояния объекта. В данном случае метод **update** будет вызываться у наблюдателей при изменении статуса заказа.

**ConcreteObserver** - это конкретная реализация интерфейса **Observer**. В данном случае это наблюдатель (например, клиент пиццерии), который получает обновления о статусе заказа. Класс хранит имя наблюдателя и выводит сообщение о новом статусе заказа при вызове метода **update**.

**Subject** - это класс, который отслеживает список наблюдателей и уведомляет их об изменениях своего состояния. Ключевые методы:

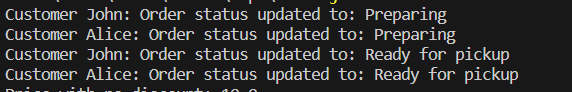
* **attach(Observer observer)** - добавляет наблюдателя в список.
* **detach(Observer observer)** - удаляет наблюдателя из списка.
* **notifyObservers()** - уведомляет всех наблюдателей о изменении состояния.
* **setStatus(String status)** - устанавливает новый статус и уведомляет наблюдателей.
* **getStatus()** - возвращает текущий статус.

**Описание:**

1. Создается объект **Subject**, представляющий заказ.
2. Создаются два наблюдателя (**ConcreteObserver**), представляющих клиентов Джона и Алису.
3. Оба клиента (наблюдателя) добавляются к заказу через метод **attach**.
4. Когда статус заказа меняется (**setStatus**), все наблюдатели уведомляются о новом статусе через метод **notifyObservers**.

Этот механизм позволяет автоматически уведомлять всех подписанных клиентов о статусе их заказа, обеспечивая гибкость и расширяемость системы.

***Rezultatul***



***Strategy***

Паттерн проектирования "Стратегия" (Strategy) относится к категории поведенческих паттернов. Этот паттерн позволяет определять семейство алгоритмов, инкапсулировать каждый из них и делать их взаимозаменяемыми. Это позволяет менять алгоритмы независимо от клиентов, которые их используют.

Паттерн "Стратегия" предоставляет способ определения семейства алгоритмов, которые могут быть выбраны и использованы в зависимости от ситуации. Вместо того чтобы реализовывать алгоритм прямо в классе, вы создаете набор классов, которые инкапсулируют разные вариации алгоритма.

**Участники паттерна**

1. **Контекст (Context)**: Класс, использующий стратегию. Он содержит ссылку на объект стратегии и методы, которые делегируют выполнение алгоритма объекту стратегии.
2. **Стратегия (Strategy)**: Общий интерфейс для всех алгоритмов. Он определяет метод, который будет вызывать контекст.
3. **Конкретные стратегии (Concrete Strategies)**: Реализации интерфейса стратегии. Каждая конкретная стратегия реализует алгоритм.

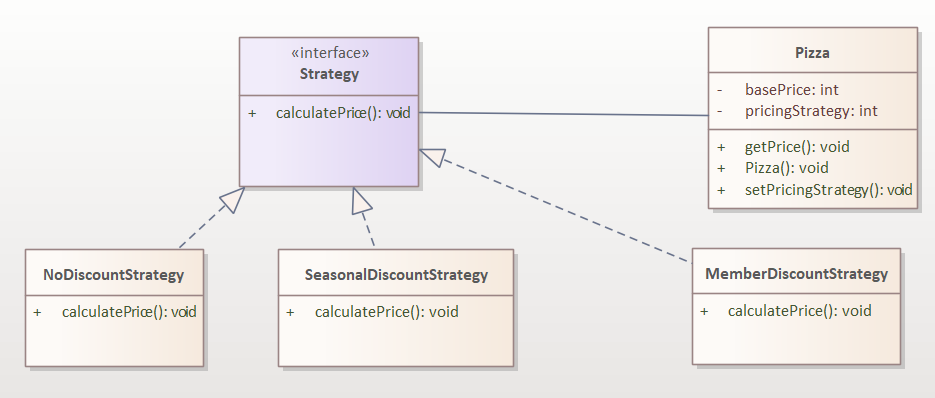


Рисунок 2 – Диаграмма реализации паттерна Стратегии

**Strategy**: Интерфейс, определяющий метод **calculatePrice**, который используется для расчета цены пиццы.

**NoDiscountStrategy**, **SeasonalDiscountStrategy**, **MemberDiscountStrategy**: Классы, реализующие интерфейс **Strategy** и предоставляющие различные стратегии ценообразования для пиццы.

**Pizza**: Класс, представляющий пиццу. Он содержит базовую цену и ссылку на объект стратегии ценообразования. Метод **getPrice** использует текущую стратегию для расчета конечной цены пиццы.

interface Strategy {

    double calculatePrice(double basePrice);

}

class NoDiscountStrategy implements Strategy {

    @Override

    public double calculatePrice(double basePrice) {

        return basePrice;

    }

}

class SeasonalDiscountStrategy implements Strategy {

    @Override

    public double calculatePrice(double basePrice) {

        return basePrice \* 0.9;

    }

}

class MemberDiscountStrategy implements Strategy {

    @Override

    public double calculatePrice(double basePrice) {

        return basePrice \* 0.8;

    }

}

class Pizza {

    private double basePrice;

    private Strategy pricingStrategy;

    public Pizza(double basePrice) {

        this.basePrice = basePrice;

    }

    public void setPricingStrategy(Strategy pricingStrategy) {

        this.pricingStrategy = pricingStrategy;

    }

    public double getPrice() {

        return pricingStrategy.calculatePrice(basePrice);

    }

}

Паттерн Стратегия позволяет определить семейство алгоритмов, инкапсулировать их и сделать их взаимозаменяемыми. Паттерн позволяет алгоритмам изменяться независимо от клиентов, которые ими пользуются. В контексте пиццерии этот паттерн может использоваться для расчета стоимости пиццы с различными стратегиями скидок.

**Strategy** - это интерфейс, который определяет метод **calculatePrice**. Этот метод принимает базовую цену и возвращает цену после применения определенной стратегии (например, без скидки, сезонная скидка, скидка для членов клуба и т.д.).

**NoDiscountStrategy** - это конкретная реализация интерфейса **Strategy**. Данная стратегия не применяет никаких скидок и возвращает базовую цену.

**SeasonalDiscountStrategy** - это конкретная реализация интерфейса **Strategy**. Данная стратегия применяет сезонную скидку (10%) и возвращает цену с учетом этой скидки.

**MemberDiscountStrategy** - это конкретная реализация интерфейса **Strategy**. Данная стратегия применяет скидку для членов клуба (20%) и возвращает цену с учетом этой скидки.

**Pizza** - это класс, представляющий пиццу. В нем хранятся базовая цена и текущая стратегия ценообразования. Ключевые методы:

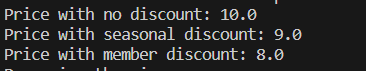
* **setPricingStrategy(Strategy pricingStrategy)** - устанавливает стратегию ценообразования.
* **getPrice()** - возвращает цену пиццы с учетом установленной стратегии.

**Описание:**

1. Создается объект **Pizza** с базовой ценой 10.0.
2. Устанавливается стратегия **NoDiscountStrategy**, и выводится цена без скидки.
3. Устанавливается стратегия **SeasonalDiscountStrategy**, и выводится цена с сезонной скидкой.
4. Устанавливается стратегия **MemberDiscountStrategy**, и выводится цена со скидкой для членов клуба.

Этот механизм позволяет легко менять стратегию ценообразования для пиццы, делая систему более гибкой и расширяемой.

***Rezultatul***



***Command***

Паттерн проектирования "Команда" (Command) относится к категории поведенческих паттернов. Этот паттерн инкапсулирует запрос в виде объекта, позволяя параметризовать клиентские объекты с различными запросами, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Паттерн "Команда" преобразует запросы на выполнение определенных действий в объекты, что позволяет передавать эти запросы как аргументы, хранить их в структурах данных, а также управлять их выполнением.

**Участники паттерна**

1. **Команда (Command)**: Интерфейс, определяющий метод **execute()**, который вызывает выполнение определенного действия.
2. **Конкретная команда (Concrete Command)**: Реализация интерфейса команды. Она содержит ссылку на объект-получатель и реализует метод **execute()**, вызывая соответствующий метод получателя.
3. **Получатель (Receiver)**: Объект, который знает, как выполнять действия, связанные с командой. Реализует конкретную логику, связанную с выполнением команды.
4. **Отправитель (Invoker)**: Отправляет запрос команде для выполнения определенного действия. Он не знает о деталях реализации команды, а только вызывает метод **execute()**.

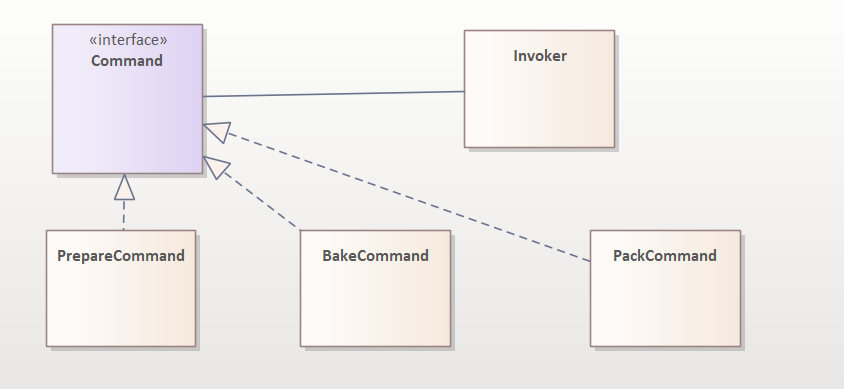


Рисунок 3 – Диаграмма реализации паттерна команда

**Command**: Интерфейс, определяющий метод **execute()**, который используется для выполнения команды.

**PrepareCommand**, **BakeCommand**, **PackCommand**: Классы, реализующие интерфейс **Command** и предоставляющие конкретные команды для подготовки, выпекания и упаковки пиццы соответственно.

**PizzaOrder**: Класс, представляющий заказ пиццы. Он содержит методы **prepare()**, **bake()** и **pack()**, которые выполняют соответствующие действия для заказа.

**Invoker**: Класс, который хранит команду и запускает её выполнение.

interface Command {

    void execute();

}

class PizzaOrder {

    public void prepare() {

        System.out.println("Preparing the pizza.");

    }

    public void bake() {

        System.out.println("Baking the pizza.");

    }

    public void pack() {

        System.out.println("Packing the pizza.");

    }

}

class PrepareCommand implements Command {

    private PizzaOrder pizzaOrder;

    public PrepareCommand(PizzaOrder pizzaOrder) {

        this.pizzaOrder = pizzaOrder;

    }

    @Override

    public void execute() {

        pizzaOrder.prepare();

    }

}

class BakeCommand implements Command {

    private PizzaOrder pizzaOrder;

    public BakeCommand(PizzaOrder pizzaOrder) {

        this.pizzaOrder = pizzaOrder;

    }

    @Override

    public void execute() {

        pizzaOrder.bake();

    }

}

class PackCommand implements Command {

    private PizzaOrder pizzaOrder;

    public PackCommand(PizzaOrder pizzaOrder) {

        this.pizzaOrder = pizzaOrder;

    }

    @Override

    public void execute() {

        pizzaOrder.pack();

    }

}

class Invoker {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void executeCommand() {

        command.execute();

    }

}

Паттерн Команда используется для инкапсуляции запроса как объекта, что позволяет параметризовать объекты с различными запросами. В пиццерии это может быть полезно для реализации системы заказа пиццы.

**Command** - это интерфейс, определяющий метод **execute()**, который запускает выполнение определенной операции.

**PizzaOrder** - это класс, представляющий процесс заказа пиццы. В данном случае, он содержит методы **prepare()**, **bake()** и **pack()**, которые соответствуют этапам подготовки, приготовления и упаковки пиццы соответственно.

**PrepareCommand** - это конкретная реализация интерфейса **Command**, которая инкапсулирует операцию подготовки пиццы. При вызове метода **execute()**, этот объект запускает выполнение метода **prepare()** у объекта **PizzaOrder**.

**BakeCommand** - это конкретная реализация интерфейса **Command**, которая инкапсулирует операцию приготовления пиццы. При вызове метода **execute()**, этот объект запускает выполнение метода **bake()** у объекта **PizzaOrder**.

**PackCommand** - это конкретная реализация интерфейса **Command**, которая инкапсулирует операцию упаковки пиццы. При вызове метода **execute()**, этот объект запускает выполнение метода **pack()** у объекта **PizzaOrder**.

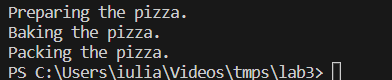
**Invoker** - это класс, который управляет выполнением команд. Он хранит объект команды и вызывает его метод **execute()** по запросу.

**Описание:**

1. Создаются объекты команд (**PrepareCommand**, **BakeCommand**, **PackCommand**), каждый из которых инкапсулирует определенную операцию заказа пиццы.
2. Создается объект **Invoker**, который будет вызывать команды.
3. Каждая команда добавляется в **Invoker** через метод **setCommand**.
4. Последовательно вызываются методы **executeCommand** у **Invoker**, что приводит к выполнению соответствующих операций заказа пиццы.

Этот механизм позволяет легко добавлять, удалять и изменять операции заказа пиццы, не затрагивая клиентский код, что делает систему более гибкой и расширяемой.

***Результат***



***Заключение***

Поведенческие шаблоны проектирования предоставляют механизмы для организации взаимодействия между объектами, управления потоком выполнения и обеспечения гибкости в реализации бизнес-логики. В контексте данного обсуждения мы рассмотрели три основных поведенческих шаблона: Наблюдатель (Observer), Стратегия (Strategy) и Шаблонный метод (Template Method).

Наблюдатель позволяет объектам подписываться и отписываться от оповещений о изменениях в других объектах, реагируя на эти изменения. Стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми, что полезно при выборе алгоритма из набора в зависимости от ситуации. Шаблонный метод определяет скелет алгоритма в суперклассе, позволяя подклассам переопределять определенные шаги алгоритма, обеспечивая гибкость в изменении его поведения.

Каждый из этих шаблонов предоставляет решение для определенных проблем в организации взаимодействия объектов и управлении выполнением кода, делая код более гибким, легко поддерживаемым и переиспользуемым.