**MINISTERUL EDUCAŢIEI AL REPUBLICII MOLDOVA**

**UNIVERSITATEA TEHNICĂ A MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICǍ ȘI MICROELECTRONICǍ**

**DEPARTAMENTUL INGINERIA SOFTWARE ȘI AUTOMATICĂ**

# *RAPORT*

Lucrarea de laborator nr. 2

**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 три Структурные Design Patterns

**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.

Структурные паттерны - это часть шаблонов проектирования в объектно-ориентированном программировании, которые используются для построения более гибких и расширяемых систем. Они фокусируются на том, как классы и объекты образуют более крупные структуры. Вот некоторые из наиболее распространенных структурных паттернов:

1. **Адаптер (Adapter)**: Преобразует интерфейс одного класса в интерфейс другого класса, который ожидают клиенты. Это позволяет объектам с несовместимыми интерфейсами работать вместе.
2. **Мост (Bridge)**: Разделяет абстракцию от ее реализации так, чтобы они могли изменяться независимо друг от друга. Это позволяет варьировать как абстракцию, так и реализацию отдельно.
3. **Компоновщик (Composite)**: Позволяет сгруппировать объекты в древовидные структуры для представления иерархий часть-целое. Клиенты могут обращаться к отдельным объектам или к целым иерархиям объектов одинаковым образом.
4. **Декоратор (Decorator)**: Позволяет добавлять новое поведение или функциональность объектам, не изменяя их код. Это достигается путем оборачивания объектов в классы-декораторы.
5. **Фасад (Facade)**: Предоставляет унифицированный интерфейс к группе интерфейсов в подсистеме. Фасад упрощает работу клиентов с системой, скрывая сложность подсистемы.
6. **Легковес (Flyweight)**: Эффективно разделяет множество мелких объектов между собой, экономя память, когда существует большое количество однородных объектов.
7. **Заместитель (Proxy)**: Представляет суррогат или заполнитель для другого объекта, чтобы контролировать доступ к нему. Это позволяет реализовать дополнительную логику при доступе к объекту.

Эти паттерны помогают улучшить модульность, повысить переиспользуемость кода и облегчить изменения в программном обеспечении. Каждый из них решает конкретные проблемы в проектировании ПО, и их выбор зависит от контекста задачи и требований проекта.

***Adapter***

Паттерн **Адаптер (Adapter)** используется для преобразования интерфейса одного класса в интерфейс другого, который ожидается клиентом. Этот паттерн позволяет объектам с несовместимыми интерфейсами работать вместе. Адаптер — мощный инструмент для интеграции различных систем и компонентов, позволяющий создать более гибкую и расширяемую архитектуру.

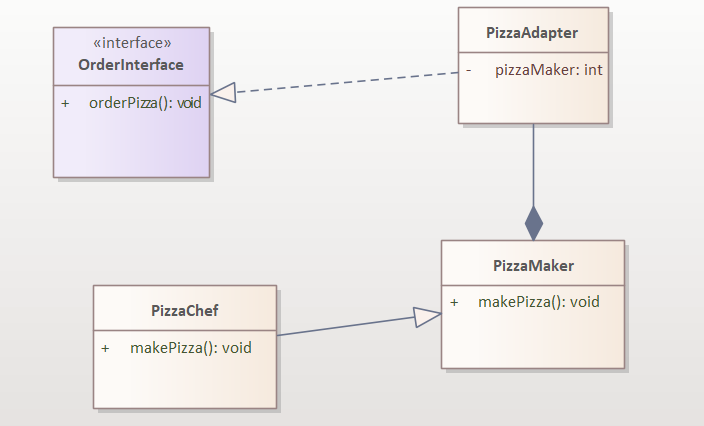


Рисунок 1 – Диаграмма реализации паттерна адаптер

**OrderInterface**: Интерфейс, предоставляющий метод **orderPizza()**.

**PizzaAdapter**: Класс-адаптер, который реализует интерфейс **OrderInterface** и использует объект **PizzaMaker** для выполнения заказа пиццы.

**PizzaMaker**: Класс, отвечающий за приготовление пиццы. Включает метод **makePizza()**.

**PizzaChef**: Подкласс **PizzaMaker**, который переопределяет метод **makePizza()** для создания конкретной пиццы.

Диаграмма классов показывает связи между интерфейсом **OrderInterface**, классом **PizzaAdapter** и классами **PizzaMaker** и **PizzaChef**, а также отношения наследования между классами **PizzaMaker** и **PizzaChef**.

interface OrderInterface {

    String orderPizza();

}

class PizzaMaker {

    String makePizza() {

        return "";

    }

}

class PizzaChef extends PizzaMaker {

    @Override

    String makePizza() {

        return "Pizza is ready!";

    }

}

class PizzaAdapter implements OrderInterface {

    private final PizzaMaker pizzaMaker;

    public PizzaAdapter(PizzaMaker pizzaMaker) {

        this.pizzaMaker = pizzaMaker;

    }

    @Override

    public String orderPizza() {

        return pizzaMaker.makePizza();

    }

}

**1. OrderInterface** — это интерфейс, который ожидается клиентом. Он содержит метод **orderPizza()**, который клиент будет использовать для заказа пиццы.

**2. PizzaMaker** — это существующий класс с методом **makePizza()**, который нужно адаптировать к интерфейсу клиента.

**3. PizzaChef** — это класс, который наследует **PizzaMaker** и реализует метод **makePizza()**, возвращая строку "Pizza is ready!".

**4. PizzaAdapter** — это класс адаптера, который реализует интерфейс **OrderInterface** и содержит ссылку на экземпляр **PizzaMaker**. Он адаптирует метод **makePizza()** к методу **orderPizza()**.

**Объяснение работы паттерна Адаптер:**

1. **OrderInterface** — это интерфейс, который ожидается клиентом. Клиент ожидает метод **orderPizza()** для заказа пиццы.
2. **PizzaMaker** — это существующий класс с методом **makePizza()**, который имеет несовместимый с клиентом интерфейс.
3. **PizzaChef** — это конкретная реализация класса **PizzaMaker**, который действительно готовит пиццу и возвращает соответствующее сообщение.
4. **PizzaAdapter** — это адаптер, который реализует интерфейс **OrderInterface** и содержит ссылку на **PizzaMaker**. Он преобразует вызов **orderPizza()** в вызов **makePizza()**.

Клиент создает экземпляр **PizzaChef**, затем оборачивает его в **PizzaAdapter**, чтобы использовать метод **orderPizza()**.

***Результат***

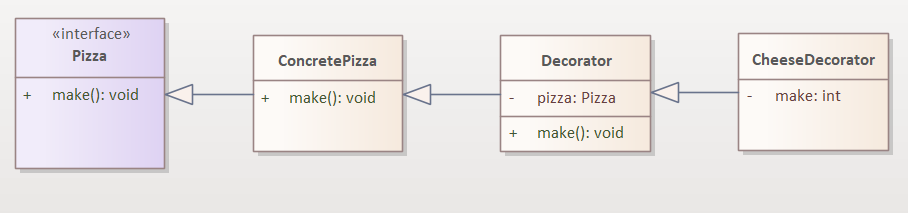


***Decorator***

Паттерн **Декоратор (Decorator)** используется для динамического добавления нового поведения или функциональности объектам. Это делается без изменения их кода, что позволяет соблюдать принцип открытости/закрытости (Open/Closed Principle) — принцип, согласно которому классы должны быть открыты для расширения, но закрыты для модификации.

Паттерн Декоратор предоставляет альтернативу наследованию для расширения функциональности объектов. Вместо создания подклассов для добавления нового поведения, вы оборачиваете объекты в декораторы, которые предоставляют дополнительные возможности.

Паттерн Декоратор часто используется, когда нужно добавлять функциональность к объекту, но не к классу. Это полезно, когда необходимо избежать создания большого количества подклассов для каждой возможной комбинации поведения.

Рисунок 2 – Диаграмма реализации паттерна декоратор

**Pizza**: Интерфейс, который определяет метод **make()**.  
**ConcretePizza**: Конкретная реализация интерфейса **Pizza**.  
**Decorator**: Абстрактный класс-декоратор, который реализует интерфейс **Pizza** и содержит ссылку на объект типа **Pizza**.  
**CheeseDecorator** и **MushroomDecorator**: Конкретные декораторы, которые расширяют функциональность базового пиццы, добавляя сыр и грибы соответственно. Каждый из них также содержит ссылку на объект типа **Pizza**.

Диаграмма классов показывает структуру и отношения между интерфейсом **Pizza**, конкретной реализацией **ConcretePizza** и абстрактными классами-декораторами **Decorator**, а также конкретными декораторами **CheeseDecorator** и **MushroomDecorator**, которые декорируют базовую пиццу.

interface Pizza {

    String make();

}

class ConcretePizza implements Pizza {

    @Override

    public String make() {

        return "Plain Pizza";

    }

}

abstract class Decorator implements Pizza {

    protected final Pizza pizza;

    public Decorator(Pizza pizza) {

        this.pizza = pizza;

    }

    @Override

    public String make() {

        return pizza.make();

    }

}

class CheeseDecorator extends Decorator {

    public CheeseDecorator(Pizza pizza) {

        super(pizza);

    }

    @Override

    public String make() {

        return pizza.make() + " with cheese";

    }

}

class MushroomDecorator extends Decorator {

    public MushroomDecorator(Pizza pizza) {

        super(pizza);

    }

    @Override

    public String make() {

        return pizza.make() + " with mushrooms";

    }

}

***Результат***



**Pizza** — это интерфейс или базовый класс, определяющий метод **make()**, который возвращает описание пиццы.

**ConcretePizza** — это конкретная реализация интерфейса **Pizza**, представляющая обычную пиццу без добавок.

**Decorator** — это абстрактный класс, реализующий интерфейс **Pizza** и содержащий ссылку на объект типа **Pizza**. Этот класс используется в качестве базового для конкретных декораторов.

**CheeseDecorator** — это конкретный декоратор, добавляющий сыр к базовой пицце.

**MushroomDecorator** — это конкретный декоратор, добавляющий грибы к базовой пицце.

**Объяснение работы паттерна Декоратор:**

1. **Pizza** — это интерфейс, который определяет метод **make()**. Все пиццы и их декораторы должны реализовывать этот интерфейс.
2. **ConcretePizza** — это конкретная реализация интерфейса **Pizza**, представляющая обычную пиццу без добавок.
3. **Decorator** — это абстрактный класс, реализующий интерфейс **Pizza** и содержащий ссылку на объект типа **Pizza**. Он делегирует выполнение метода **make()** объекту **Pizza**, который он декорирует.
4. **CheeseDecorator** — это конкретный декоратор, добавляющий сыр к базовой пицце. Он расширяет класс **Decorator** и переопределяет метод **make()**, добавляя функциональность.
5. **MushroomDecorator** — это конкретный декоратор, добавляющий грибы к базовой пицце. Он также расширяет класс **Decorator** и переопределяет метод **make()**, добавляя функциональность.

***Facade***

Паттерн **Фасад (Facade)** предназначен для предоставления унифицированного интерфейса к сложной подсистеме. Фасад определяет высокоуровневый интерфейс, который упрощает взаимодействие с подсистемой, скрывая её сложность. Это позволяет клиентам работать с подсистемой через простой интерфейс, не вдаваясь в детали её внутренней реализации.

Паттерн Фасад помогает уменьшить сложность работы с подсистемой, предоставляя упрощённый интерфейс, за которым скрываются сложные или многочисленные компоненты подсистемы. Это позволяет клиентам взаимодействовать с подсистемой через единый объект.

**Компоненты паттерна Фасад**

1. **Фасад (Facade)**: Предоставляет простой интерфейс для взаимодействия с подсистемой. Он делегирует запросы клиентов соответствующим объектам подсистемы.
2. **Подсистема (Subsystem)**: Набор классов, которые выполняют основную работу. Подсистема не знает о существовании фасада и работает независимо.

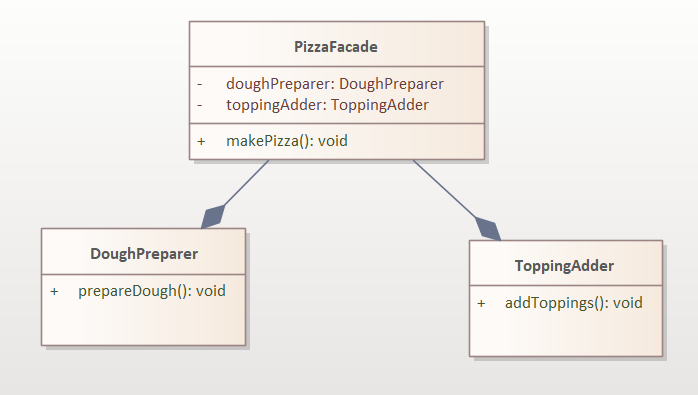


Рисунок 3 – Диаграмма реализации паттерна фасад

**DoughPreparer** и **ToppingAdder**: Классы, представляющие функциональность подготовки теста и добавления начинки соответственно. Они содержат методы **prepareDough()** и **addToppings()**.

**PizzaFacade**: Класс-фасад, который скрывает сложность подсистемы и предоставляет унифицированный интерфейс для взаимодействия с этой подсистемой. Он содержит ссылки на объекты **DoughPreparer** и **ToppingAdder** и предоставляет метод **makePizza()**, который вызывает методы подготовки теста и добавления начинки для создания пиццы.

class DoughPreparer {

    String prepareDough() {

        return "Dough is ready!";

    }

}

class ToppingAdder {

    String addToppings() {

        return "Toppings are added!";

    }

}

class PizzaFacade {

    private final DoughPreparer doughPreparer;

    private final ToppingAdder toppingAdder;

    public PizzaFacade() {

        this.doughPreparer = new DoughPreparer();

        this.toppingAdder = new ToppingAdder();

    }

    String makePizza() {

        String dough = doughPreparer.prepareDough();

        String toppings = toppingAdder.addToppings();

        return dough + " " + toppings;

    }

}

***Результат***

******

**DoughPreparer** — это класс, ответственный за подготовку теста для пиццы.

**ToppingAdder** — это класс, ответственный за добавление топпингов на пиццу.

**PizzaFacade** — это фасад, который предоставляет простой интерфейс для клиента, скрывая сложность создания пиццы, взаимодействуя с **DoughPreparer** и **ToppingAdder**.

**Объяснение работы паттерна Фасад:**

1. **DoughPreparer** — это класс, который отвечает за подготовку теста. Метод **prepareDough()** возвращает строку "Dough is ready!".
2. **ToppingAdder** — это класс, который отвечает за добавление топпингов. Метод **addToppings()** возвращает строку "Toppings are added!".
3. **PizzaFacade** — это класс фасада, который предоставляет простой интерфейс для создания пиццы. Он создает экземпляры **DoughPreparer** и **ToppingAdder** и использует их методы внутри метода **makePizza()**, возвращая объединенную строку "Dough is ready! Toppings are added!".

***Заключение***

Структурные паттерны проектирования играют ключевую роль в создании гибких и расширяемых систем, предоставляя проверенные решения для организации классов и объектов. Паттерн **Фасад** упрощает взаимодействие с подсистемами, скрывая их сложность за унифицированным интерфейсом. **Декоратор** позволяет динамически добавлять функциональность объектам, не изменяя их исходного кода, что повышает гибкость системы. **Адаптер** устраняет несовместимость интерфейсов, позволяя взаимодействовать объектам с различными интерфейсами. **Компоновщик** упрощает работу с древовидными структурами объектов, позволяя клиентам единообразно управлять как отдельными объектами, так и их группами. **Мост** разделяет абстракции и их реализации, позволяя им изменяться независимо друг от друга. **Легковес** оптимизирует использование памяти при работе с большим количеством однотипных объектов, разделяя их общие состояния. **Заместитель** контролирует доступ к объектам, добавляя дополнительную логику при их использовании. Эти паттерны способствуют модульности, переиспользованию кода и повышению гибкости архитектуры, делая системы более понятными и легкими в сопровождении.