**MINISTERUL EDUCAŢIEI AL REPUBLICII MOLDOVA**

**UNIVERSITATEA TEHNICĂ A MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICǍ ȘI MICROELECTRONICǍ**

**DEPARTAMENTUL INGINERIA SOFTWARE ȘI AUTOMATICĂ**

# *RAPORT*

Lucrarea de laborator nr. 1

**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 порождающие 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.

Порождающие шаблоны проектирования (Creational Design Patterns) — это шаблоны, которые сосредоточены на процессе создания объектов. Их основной целью является абстрагирование процесса инстанцирования объектов, чтобы сделать систему более независимой от конкретных классов, которые необходимо создавать. Это позволяет повысить гибкость и повторное использование кода, а также упростить управление объектами.

Порождающие шаблоны проектирования обычно применяются для отделения процесса создания объектов от их использования. Основная цель такого подхода — обеспечить возможность добавления новых производных типов без необходимости изменения кода, который использует базовые классы.

В разработке программного обеспечения порождающие шаблоны описывают механизмы создания объектов, адаптированные к конкретным ситуациям. Стандартные методы создания объектов могут привести к проектным проблемам или усложнить работу с кодом. Порождающие шаблоны решают эту проблему, контролируя процесс создания объектов и обеспечивая более гибкие и управляемые способы их инстанцирования.

***Singleton***

Синглтон (или одиночка) — это порождающий шаблон проектирования, который гарантирует, что у класса будет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Этот шаблон полезен в ситуациях, когда необходимо иметь ровно один объект для управления доступом к ресурсам или для координации действий в системе.

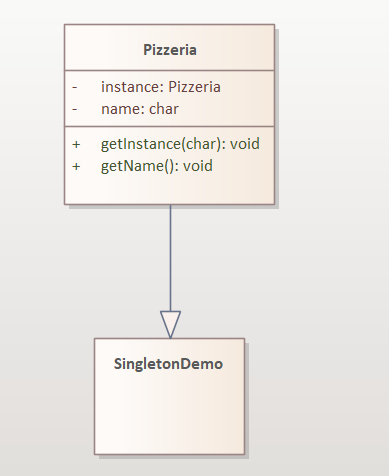


Рисунок 1 – Диаграмма реализации паттерна синглтон

**Pizzeria**: Этот класс представляет собой синглтон-реализацию пиццерии. У него есть статический метод **getInstance**, который возвращает единственный экземпляр класса **Pizzeria**. У класса также есть приватное поле **name**, содержащее название пиццерии, и публичный метод **getName**, возвращающий это название.

**SingletonDemo**: Это класс с методом **main**, который демонстрирует использование синглтона. В данном случае он вызывает метод **getInstance** класса **Pizzeria** дважды и выводит названия пиццерий, а также проверяет, что оба объекта, полученные с помощью **getInstance**, являются одним и тем же объектом.

Диаграмма классов показывает связь между классами **Pizzeria** и **SingletonDemo**, а также показывает, что класс **Pizzeria** реализует паттерн синглтона.

public static class Pizzeria {

        private static Pizzeria instance;

        private String name;

        private Pizzeria(String name) {

            this.name = name;

        }

        public static Pizzeria getInstance(String name) {

            if (instance == null) {

                instance = new Pizzeria(name);

            }

            return instance;

        }

        public String getName() {

            return name;

        }

    }

    public static class SingletonDemo {

        public static void main(String[] args) {

            Pizzeria pizzeria1 = Pizzeria.getInstance("Pizza Palace");

            Pizzeria pizzeria2 = Pizzeria.getInstance("Pizza Kingdom");

            System.out.println(pizzeria1.getName()); // Output: Pizza Palace

            System.out.println(pizzeria2.getName()); // Output: Pizza Palace

            System.out.println(pizzeria1 == pizzeria2); // Output: true

        }

    }

**Класс Pizzeria**

**1. Поле instance** это статическое поле, которое хранит единственный экземпляр класса **Pizzeria**.

Статическое поле принадлежит классу, а не конкретному экземпляру.

**2. Конструктор** объявлен как **private**, что предотвращает создание экземпляров этого класса извне. Конструктор принимает имя пиццерии и сохраняет его в поле **name**.

**3. Метод getInstance** это статический метод, который возвращает единственный экземпляр класса. Если экземпляр еще не создан (**instance == null**), он создает новый экземпляр с указанным именем. Если экземпляр уже существует, метод просто возвращает этот экземпляр. Этот метод гарантирует, что экземпляр класса будет создан только один раз, даже если он вызывается многократно.

**4. Метод getName** Возвращает имя пиццерии.

**Класс SingletonDemo**

**1. Метод main** демонстрируется использование Singleton паттерна. **Pizzeria.getInstance("Pizza Palace")** создает (если еще не создан) или возвращает единственный экземпляр **Pizzeria** с именем "Pizza Palace". **Pizzeria.getInstance("Pizza Kingdom")** пытается создать новый экземпляр с именем "Pizza Kingdom", но вместо этого возвращает уже созданный экземпляр "Pizza Palace", так как экземпляр уже существует.

Вывод на экран:

* **pizzeria1.getName()** и **pizzeria2.getName()** оба возвращают "Pizza Palace", что подтверждает наличие единственного экземпляра.
* **pizzeria1 == pizzeria2** возвращает **true**, что подтверждает, что обе переменные указывают на один и тот же объект.

Singleton паттерн гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру. В этом примере класс **Pizzeria** реализует этот паттерн, обеспечивая создание и доступ к единственному экземпляру пиццерии.

***Результат***



***Factory Method***

Фабричный метод (Factory Method) - это порождающий шаблон проектирования, который определяет интерфейс для создания объектов некоторого класса, но оставляет подклассам решение о том, какой класс инстанцировать. То есть, он делегирует ответственность за создание экземпляров подклассам.

Ключевые понятия:

* Интерфейс Фабрики (Creator): Это абстрактный класс или интерфейс, который объявляет метод фабрики для создания объектов. Этот метод может быть абстрактным, что заставляет подклассы предоставить свою собственную реализацию.
* Конкретные Фабрики (Concrete Creators): Это подклассы интерфейса фабрики, которые реализуют метод создания объектов. Каждая конкретная фабрика создает экземпляры определенного класса или его подклассов.
* Продукты (Products): Это абстрактные классы или интерфейсы объектов, которые создаются фабричным методом. Конкретные продукты предоставляют конкретную реализацию этих абстрактных классов.
* Конкретные Продукты (Concrete Products): Это классы, представляющие конкретные реализации продуктов. Каждая конкретная фабрика создает экземпляры определенного конкретного продукта.

Преимущество фабрики в том, что знание о том, как создать объект, находится в одном месте — внутри фабрики. Если схема объекта поменяется, то изменить код нам нужно будет только в одном месте — в фабрике.

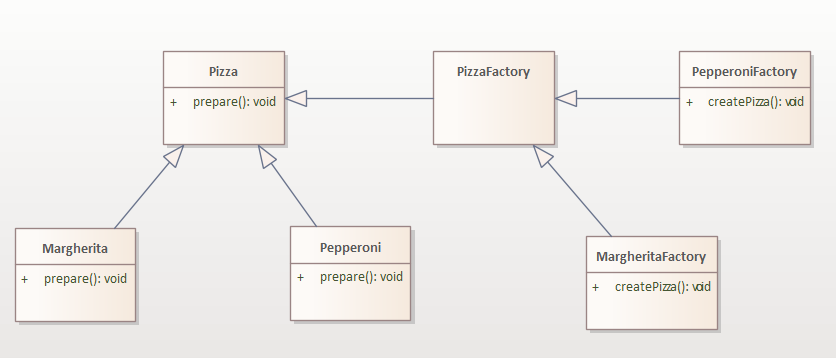


Рисунок 2 – Диаграмма реализации паттерна Фабричный метод

**Pizza**: Абстрактный класс, представляющий пиццу. Он имеет абстрактный метод **prepare()**, который должен быть реализован в конкретных подклассах.

**Margherita** и **Pepperoni**: Конкретные классы пицц, наследующие от абстрактного класса **Pizza** и реализующие метод **prepare()**.

**PizzaFactory**: Абстрактный класс, представляющий фабрику пицц. Он имеет абстрактный метод **createPizza()**, который должен быть реализован в конкретных подклассах.

**MargheritaFactory** и **PepperoniFactory**: Конкретные фабрики пицц, наследующие от абстрактного класса **PizzaFactory** и реализующие метод **createPizza()**, чтобы создать соответствующий объект пиццы.

Диаграмма классов показывает отношения между абстракциями пиццы и их фабриками, а также между конкретными реализациями

public static abstract class Pizza {

        public abstract String prepare();

    }

    public static class Margherita extends Pizza {

        @Override

        public String prepare() {

            return "Preparing Margherita Pizza";

        }

    }

    public static class Pepperoni extends Pizza {

        @Override

        public String prepare() {

            return "Preparing Pepperoni Pizza";

        }

    }

    public static abstract class PizzaFactory {

        public abstract Pizza createPizza();

    }

    public static class MargheritaFactory extends PizzaFactory {

        @Override

        public Pizza createPizza() {

            return new Margherita();

        }

    }

    public static class PepperoniFactory extends PizzaFactory {

        @Override

        public Pizza createPizza() {

            return new Pepperoni();

        }

    }

    public static class FactoryMethodDemo {

        public static void main(String[] args) {

            PizzaFactory margheritaFactory = new MargheritaFactory();

            Pizza margherita = margheritaFactory.createPizza();

            System.out.println(margherita.prepare());

            PizzaFactory pepperoniFactory = new PepperoniFactory();

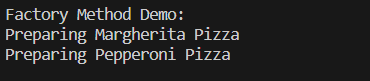
            Pizza pepperoni = pepperoniFactory.createPizza();

            System.out.println(pepperoni.prepare());

        }

    }

***Результат***



**Абстрактный класс Pizza**  - это класс, представляющий пиццу. Он содержит абстрактный метод **prepare()**, который должен быть реализован в конкретных подклассах.

**Конкретные классы Margherita и Pepperoni** - это конкретные классы, которые расширяют абстрактный класс **Pizza**.Они реализуют метод **prepare()**, возвращая строку, описывающую процесс приготовления конкретной пиццы.

**Абстрактный класс PizzaFactory** - это абстрактный класс, представляющий фабрику пицц.Он содержит абстрактный метод **createPizza()**, который должен быть реализован в конкретных подклассах.

**Конкретные фабрики MargheritaFactory и PepperoniFactory -** это конкретные фабрики, которые расширяют абстрактный класс **PizzaFactory**.Они реализуют метод **createPizza()**, создавая конкретные экземпляры пиццы (**Margherita** и **Pepperoni** соответственно).

**Класс FactoryMethodDemo -** это класс, демонстрирующий работу паттерна Фабричный метод.В методе **main** создаются фабрики пицц (**MargheritaFactory** и **PepperoniFactory**), затем с их помощью создаются соответствующие пиццы (**Margherita** и **Pepperoni**) и выводится сообщение о подготовке каждой из них.

***Builder***

Реализация шаблона Builder (Строитель) включает создание классов, которые отвечают за пошаговое построение сложных объектов. Этот шаблон полезен, когда создание объекта требует выполнения множества шагов или имеет различные вариации, и когда необходимо обеспечить, чтобы процесс конструирования был независим от конечного продукта, который создается.

**Основные компоненты:**

1. **Продукт (Product):** Это объект, который мы хотим создать с помощью шаблона Builder. Продукт может иметь сложную структуру и состояние, которые мы хотим построить.
2. **Строитель (Builder):** Это интерфейс или абстрактный класс, который определяет методы для пошагового построения продукта. Каждый метод соответствует определенному шагу конструирования.
3. **Конкретные строители (Concrete Builders):** Это классы, которые реализуют интерфейс или наследуют абстрактный класс строителя. Они реализуют методы для создания и настройки конкретных компонентов продукта.
4. **Директор (Director):** Это класс, который управляет процессом конструирования. Он получает экземпляр конкретного строителя и вызывает его методы для построения продукта.

Шаблон Builder позволяет создавать объекты с различными конфигурациями и структурами, обеспечивая гибкость и удобство в поддержке кода.

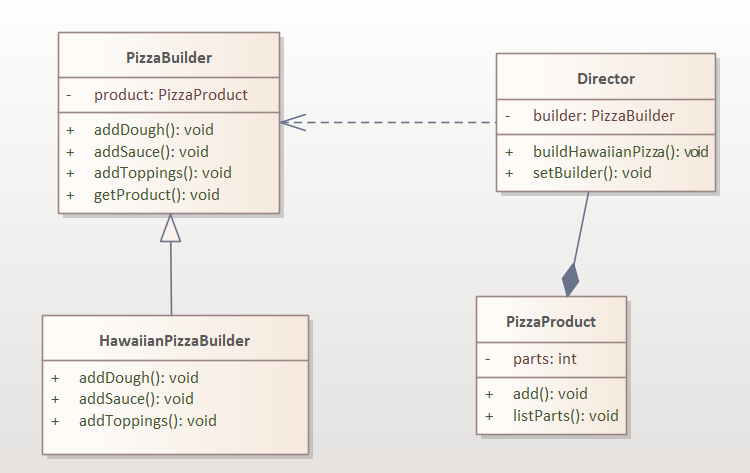


Рисунок 3 – Диаграмма реализации паттерна строитель

**PizzaBuilder**: Абстрактный класс, который представляет строителя пиццы. Он имеет методы для добавления теста, соуса и начинки, а также метод **getProduct()**, который возвращает готовый продукт.

**HawaiianPizzaBuilder**: Конкретный строитель для гавайской пиццы. Реализует абстрактные методы **addDough()**, **addSauce()** и **addToppings()** для создания гавайской пиццы.

**PizzaProduct**: Класс, представляющий продукт (пиццу). Он содержит список частей пиццы и методы для их добавления и вывода.

**Director**: Класс, который управляет процессом построения пиццы. Он имеет ссылку на объект строителя и методы для установки строителя и создания пиццы.

**BuilderDemo**: Класс с методом **main**, который демонстрирует использование паттерна Строитель. Создает объект директора, устанавливает строителя для гавайской пиццы и строит пиццу.

Диаграмма классов показывает отношения между классами и их функциональность в рамках паттерна Строитель.

public static abstract class PizzaBuilder {

        protected PizzaProduct product = new PizzaProduct();

        public abstract void addDough();

        public abstract void addSauce();

        public abstract void addToppings();

        public PizzaProduct getProduct() {

            PizzaProduct result = product;

            product = new PizzaProduct();

            return result;

        }

    }

    public static class HawaiianPizzaBuilder extends PizzaBuilder {

        @Override

        public void addDough() {

            product.add("Dough: Thin Crust");

        }

        @Override

        public void addSauce() {

            product.add("Sauce: Tomato");

        }

        @Override

        public void addToppings() {

            product.add("Toppings: Ham, Pineapple");

        }

    }

    public static class PizzaProduct {

        private List<String> parts = new ArrayList<>();

        public void add(String part) {

            parts.add(part);

        }

        public void listParts() {

            System.out.println("Pizza parts: " + String.join(", ", parts));

        }

    }

    public static class Director {

        private PizzaBuilder builder;

        public void setBuilder(PizzaBuilder builder) {

            this.builder = builder;

        }

        public void buildHawaiianPizza() {

            builder.addDough();

            builder.addSauce();

            builder.addToppings();

        }

    }

    public static class BuilderDemo {

        public static void main(String[] args) {

            Director director = new Director();

            PizzaBuilder builder = new HawaiianPizzaBuilder();

            director.setBuilder(builder);

            System.out.println("Hawaiian Pizza: ");

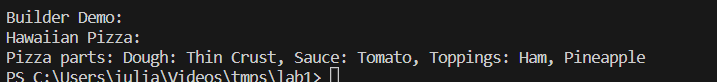
            director.buildHawaiianPizza();

            builder.getProduct().listParts();

        }

    }

***Результат***



**Абстрактный класс PizzaBuilder -** это абстрактный класс, представляющий строителя пиццы.Он определяет методы для добавления теста (**addDough**), соуса (**addSauce**) и начинки (**addToppings**). Также он содержит метод **getProduct()**, который возвращает готовый продукт (объект **PizzaProduct**).

**Конкретный строитель HawaiianPizzaBuilder -**это конкретный класс-строитель, который расширяет абстрактный класс **PizzaBuilder**.В этом классе реализуются методы добавления теста, соуса и начинки для гавайской пиццы.

**Класс PizzaProduct** - это класс, представляющий готовый продукт - пиццу.Он содержит список частей пиццы и метод для добавления новых частей (**add**) и вывода всех частей на экран (**listParts**).

**Класс Director** - это класс, который управляет процессом построения объекта.Он содержит ссылку на объект строителя и метод **buildHawaiianPizza()**, который вызывает методы строителя для построения гавайской пиццы.

**Класс BuilderDemo -** это класс, демонстрирующий работу паттерна Строитель.В методе **main** создается экземпляр **Director** и объект **HawaiianPizzaBuilder**, затем строитель устанавливается в директор и вызывается метод **buildHawaiianPizza()**, чтобы построить гавайскую пиццу. Затем вызывается метод **listParts()** для вывода всех частей пиццы на экран.

Паттерн Строитель используется для создания сложных объектов путем пошаговой сборки их различных частей. В этом примере абстрактный класс PizzaBuilder определяет шаги построения пиццы, а конкретный строитель HawaiianPizzaBuilder реализует эти шаги для гавайской пиццы. Класс Director управляет процессом сборки, а класс BuilderDemo демонстрирует его использование.

***Заключение***

Порождающие паттерны представляют собой набор шаблонов проектирования, которые облегчают процесс создания объектов. Они позволяют отделить создание объектов от их использования, обеспечивая гибкость и управляемость в процессе разработки программного обеспечения. Эти паттерны решают проблемы, связанные с созданием объектов в различных ситуациях, предоставляя различные методики и стратегии для их создания. Порождающие паттерны также способствуют повышению расширяемости, управляемости и поддерживаемости кода, что делает их важными инструментами в арсенале любого разработчика.