12. Паттерны проектирования, порождающие паттерны

Теоретическое введение

Паттерны проектирования — это повторяющиеся решения типичных задач проектирования программного обеспечения. Они помогают сделать код более гибким, поддерживаемым и повторно используемым. Порождающие паттерны проектирования играют важную роль в управлении процессом создания объектов, предоставляя гибкие способы инкапсуляции создания объектов и обеспечения их корректного использования.

Порождающие паттерны (Creational Patterns) — это группа паттернов проектирования, которые фокусируются на процессе создания объектов. Они помогают абстрагировать или скрыть сложность создания объектов, обеспечивая гибкость и независимость системы от конкретных классов объектов.

Основные задачи порождающих паттернов:

1. Инкапсуляция создания объектов: отделение логики создания объекта от его использования.
2. Управление сложностью: упрощение создания сложных объектов.
3. Поддержка гибкости: предоставление возможности изменения способа создания объектов без модификации кода, использующего эти объекты.

Основные порождающие паттерны

1. Singleton (Одиночка)

Цель: гарантирует, что у класса будет только один экземпляр, и предоставляет глобальную точку доступа к нему.

Применение: управление ресурсами (например, пул соединений с базой данных), централизованное управление конфигурациями.

Пример реализации приведен на Листинге 12.1.

Листинг 12.1 – Пример реализации паттерна Singleton

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

2. Factory Method (Фабричный метод)

Цель: определяет общий интерфейс для создания объектов в подклассах, оставляя решение о том, какой класс инстанцировать, за ними.

Применение: создание объектов с похожими характеристиками, но разными типами (например, в зависимости от окружения или параметров).

Пример реализации приведен на Листинге 12.2.

Листинг 12.2 – Пример реализации паттерна Factory Method

```
abstract class Creator {
    public abstract Product createProduct();
}

class ConcreteCreator extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProduct();
    }
}
```

3. Abstract Factory (Абстрактная фабрика)

Цель: предоставляет интерфейс для создания семейств взаимосвязанных объектов, не указывая их конкретные классы.

Применение: создание групп объектов, которые работают вместе (например, UI-компоненты для разных операционных систем).

Пример реализации приведен на Листинге 12.3.

Листинг 12.3 – Пример реализации паттерна Abstract Fabric

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
    public Checkbox createCheckbox() {
        return new WinCheckbox();
    }
}
```

4. Builder (Строитель)

Цель: разделяет процесс построения сложного объекта от его представления, позволяя создавать разные представления одного объекта.

Применение: создание сложных объектов с множеством параметров или шагов (например, объект конфигурации или сложный UI).

Пример реализации приведен на Листинге 12.4.

Листинг 12.4 – Пример реализации паттерна Builder

```
class Product {
    private String partA;
    private String partB;

    public void setPartA(String partA) { this.partA = partA; }
    public void setPartB(String partB) { this.partB = partB; }
}

class Builder {
    private Product product = new Product();

    public Builder buildPartA(String partA) {
        product.setPartA(partA);
        return this;
    }

    public Builder buildPartB(String partB) {
        product.setPartB(partB);
        return this;
    }

    public Product build() {
        return product;
    }
}
```

5. Prototype (Прототип)

Цель: позволяет создавать копии объектов, избегая затрат на создание нового объекта "с нуля".

Применение: клонирование объектов, если их создание сложное или ресурсоемкое.

Пример реализации приведен на Листинге 12.5.

Листинг 12.5 – Пример реализации паттерна Prototype

```
interface Prototype extends Cloneable {
    Prototype clone();
}

class ConcretePrototype implements Prototype {
    private String state;

    public void setState(String state) { this.state = state; }

    @Override
    public Prototype clone() {
        try {
            return (Prototype) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Практическое задание

В данной практической работе представлено 5 вариантов заданий. Выбор варианта осуществляется в соответствии с порядковым номером студента в списке. Если порядковый номер студента равен 1, выполняется задание варианта №1; если порядковый номер равен 5 — выполняется вариант №5. В случае, если порядковый номер превышает количество вариантов (например, 6 или более), задание выбирается циклически, начиная с варианта №1.

Вариант №1

Singleton – Задание

Реализовать систему управления настройками приложения. Необходимо создать класс AppSettings, который будет использоваться для хранения и управления настройками (например, тема, язык, путь к файлам).

Гарантировать, что класс AppSettings будет иметь только один экземпляр.

Реализовать методы для установки и получения настроек.

Продемонстрировать работу Singleton, используя несколько потоков.

Пример функционала:

Метод getInstance() для получения объекта.

Метод setSetting(String key, String value) для изменения настроек.

Метод getSetting(String key) для получения значения настроек

Вариант №2

Factory Method – Задание

Разработать систему доставки еды. Необходимо реализовать абстрактный класс DeliveryService с методом createOrder(). Каждый подкласс будет представлять конкретный способ доставки (например, PizzaDelivery, GroceryDelivery).

Реализовать абстрактный метод createOrder().

Создать два подкласса для разных типов доставки.

Продемонстрировать работу фабрики, создавая заказы через соответствующие сервисы.

Пример функционала:

Класс DeliveryService с методом createOrder().

Классы PizzaDelivery и GroceryDelivery, которые создают заказы соответствующих типов.

Метод deliver() для вывода информации о доставке.

Вариант №3

Abstract Factory – Задание

Разработать систему для генерации UI-компонентов для разных платформ (Windows и MacOS). Создать интерфейс GUIFactory с методами createButton() и createCheckbox().

Реализовать конкретные фабрики для каждой платформы.

Реализовать классы для кнопок и чекбоксов с платформенно-зависимыми реализациями.

Написать клиентский код, который использует фабрики для создания UI-компонентов.

Пример функционала:

Класс WinGUIFactory создает Windows-элементы.

Класс MacGUIFactory создает MacOS-элементы.

Вызов метода draw() на созданных элементах должен отображать, какой платформе они принадлежат.

Вариант №4

Builder – Задание

Разработать систему для создания заказа в ресторане. Использовать паттерн Builder для построения объекта Order, который включает следующие параметры:

Основное блюдо.

Гарнир.

Напиток.

Десерт.

Создать класс Order с соответствующими полями.

Реализовать класс OrderBuilder для пошагового создания заказа.

Продемонстрировать сборку заказов с разными конфигурациями.

Пример функционала:

Метод setMainDish(String mainDish) для выбора основного блюда.

Метод setDrink(String drink) для выбора напитка.

Метод build() возвращает готовый заказ.

Вариант №5

Prototype – Задание

Реализовать систему для управления 2D-объектами (например, фигурами: кругами, прямоугольниками). Использовать паттерн Prototype для клонирования объектов.

Создать интерфейс Shape с методом clone().

Реализовать классы Circle и Rectangle, которые поддерживают клонирование.

Продемонстрировать клонирование объектов, изменение их свойств и сохранение оригинальных объектов неизменными.

Пример функционала:

Поля для хранения параметров (радиус для круга, ширина и высота для прямоугольника).

Метод clone() возвращает копию объекта.

В клиентском коде создается оригинальная фигура, затем клонируется и модифицируется копия.