Содержание

Паттерны

1. Стратегия
2. Шаблонный метод
3. Команда
4. Простая фабрика
5. Абстрактная фабрика
6. Адаптер
7. Декоратор
8. Наблюдатель
9. Синглтон
10. Итератор
11. Фасад

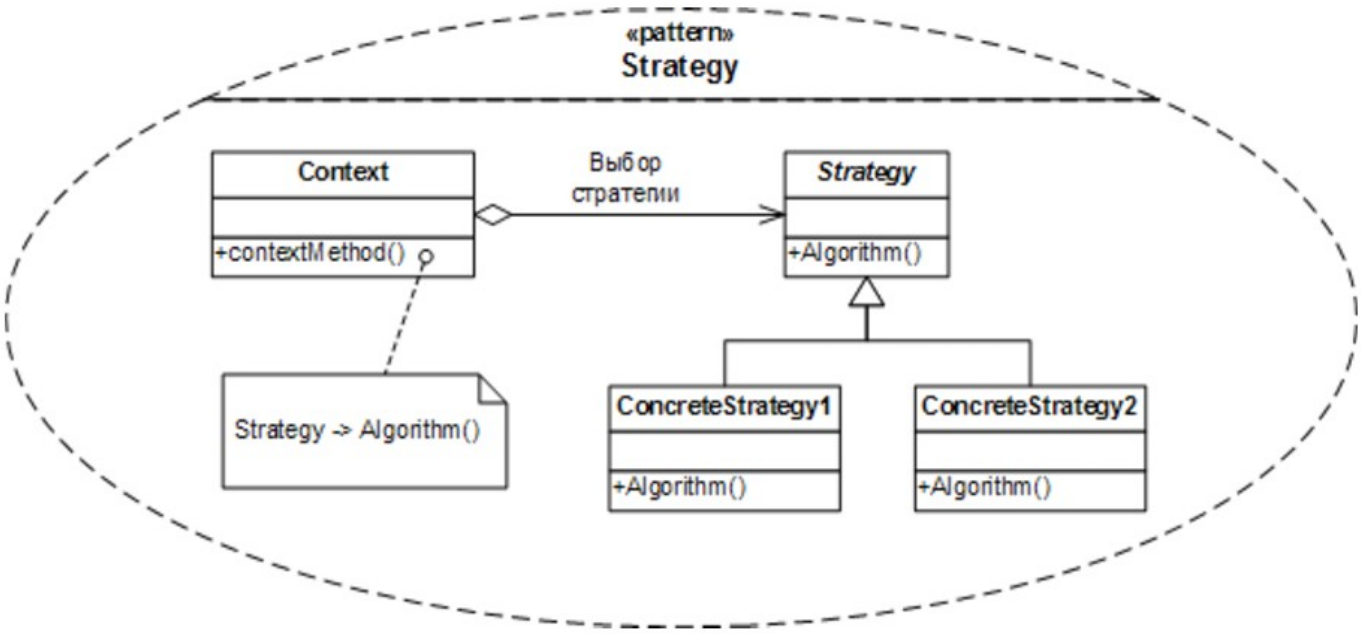
Гибкое проектирование

SOLID

1. Принцип единственной ответственности (Single-Responsibility Principle – SRP).
2. Принцип открытости/закрытости (Open/Closed Principle – OCP).
3. Принцип подстановки Лисков (Liskov Substitution Principle – LSP).
4. Принцип инверсии зависимости (Dependency-Inversion Principle – DIP).
5. Принцип разделения интерфейсов (Interface Segregation Principle – ISP).

Паттерны

Стратегия



Паттерн Стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. Он позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.

Паттерн стратегия — это способ организации кода, который позволяет выбирать из нескольких альтернативных способов выполнения определенного действия во время выполнения программы. Этот паттерн очень полезен, когда у вас есть различные способы решения одной и той же задачи и вы хотите, чтобы ваш код был гибким и легко изменяемым.

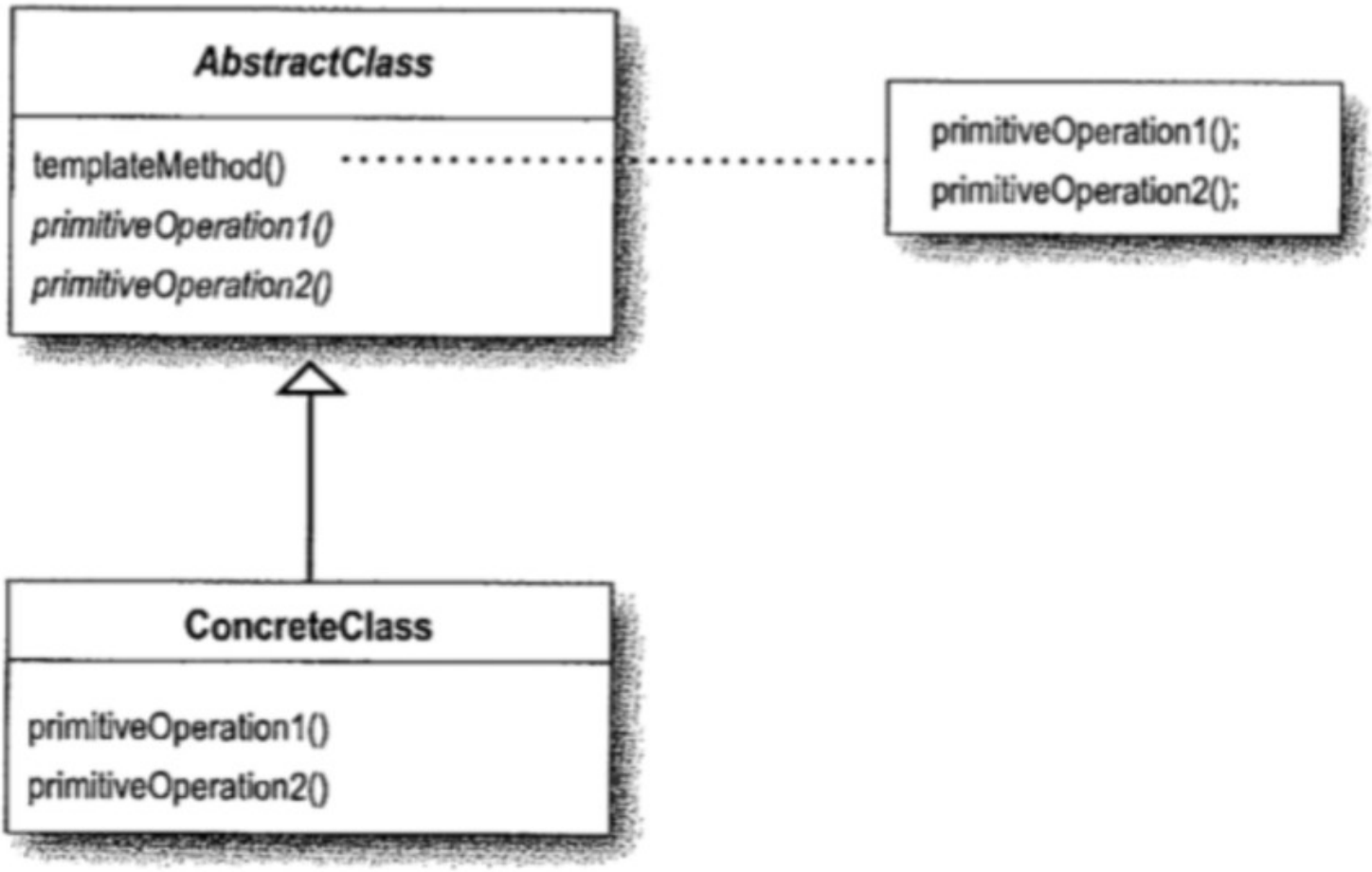
Пример проблемы: Представим, что у нас есть программа для расчета стоимости доставки товаров. В зависимости от выбранного способа доставки (наземная, воздушная, морская), расчет стоимости будет разным.

```java
// Интерфейс стратегии
interface DeliveryStrategy {
  double calculate(double weight);
}

// Конкретная стратегия для доставки наземным путем
class GroundDeliveryStrategy implements DeliveryStrategy {
  public double calculate(double weight) {
    // В данном случае просто возвращаем фиксированную стоимость доставки
    return weight * 0.5;
  }
}

// Конкретная стратегия для доставки воздушным путем
class AirDeliveryStrategy implements DeliveryStrategy {
  public double calculate(double weight) {
    // В данном случае просто возвращаем стоимость доставки, зависящую от веса и скорости доставки
    return weight * 1.5;
  }
}

// Конкретная стратегия для морской доставки
class SeaDeliveryStrategy implements DeliveryStrategy {
  public double calculate(double weight) {
    // В данном случае просто возвращаем стоимость доставки, зависящую от веса и плотности груза
    return weight * 0.3;
  }
}
// Контекст, который использует выбранную стратегию доставки
class DeliveryContext {
  private DeliveryStrategy strategy;

  public DeliveryContext(DeliveryStrategy strategy) {
    this.strategy = strategy;
```

```
    }

    public double executeStrategy(double weight) {
        return strategy.calculate(weight);
    }
}

// Пример использования
public class Main {
    public static void main(String[] args) {
        // Создаем контекст с выбранной стратегией доставки
        DeliveryContext context;

        // Выбираем стратегию доставки
        String deliveryType = "air"; // Можно изменить на "ground" или "sea"
        switch (deliveryType) {
            case "ground":
                context = new DeliveryContext(new GroundDeliveryStrategy());
                break;
            case "air":
                context = new DeliveryContext(new AirDeliveryStrategy());
                break;
            case "sea":
                context = new DeliveryContext(new SeaDeliveryStrategy());
                break;
            default:
                throw new IllegalArgumentException("Unknown delivery type");
        }

        // Выполняем расчет стоимости доставки
        double weight = 10; // Вес товара
        double cost = context.executeStrategy(weight);

        // Выводим результат
        System.out.println("Стоимость доставки: " + cost);
    }
}
```

В этом примере мы создали различные стратегии доставки (наземной, воздушной и морской) и контекст, который использует выбранную стратегию для выполнения расчета стоимости доставки. В зависимости от выбранной стратегии в контексте, результат будет отличаться.

Шаблонный метод



Шаблонный метод
определяет основные шаги алгоритма
и позволяет субклассам
предоставить реализацию
одного или нескольких шагов

Шаблонный метод — это поведенческий паттерн проектирования, который определяет тело алгоритма в родительском классе, но позволяет подклассам переопределять определенные части алгоритма без изменения его структуры.

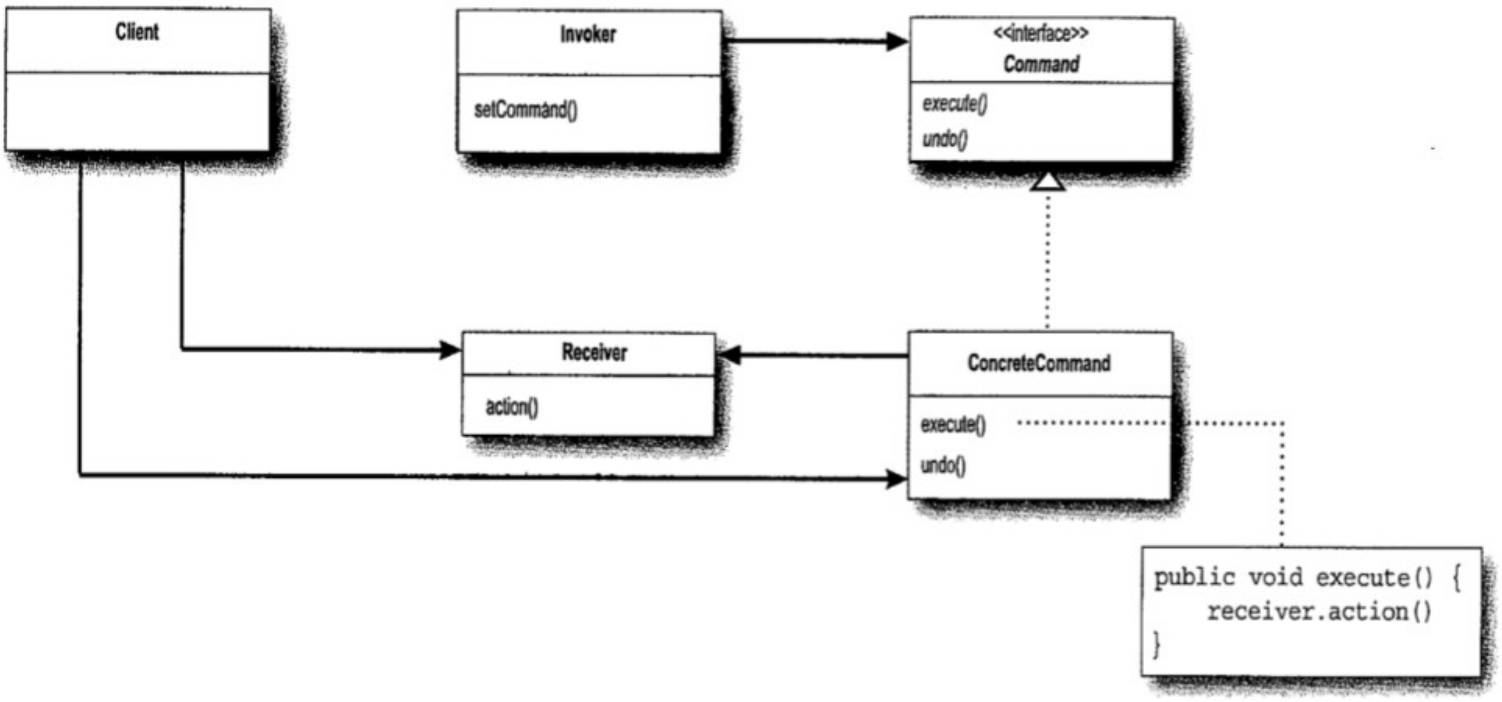
```java
// Абстрактный класс, определяющий скелет алгоритма
abstract class AlgorithmTemplate {
  // Шаги алгоритма, которые могут быть переопределены в подклассах
  protected abstract void step1();
  protected abstract void step2();
  protected abstract void step3();

  // Шаблонный метод, который определяет порядок выполнения шагов алгоритма
  public final void execute() {
    step1();
    step2();
    step3();
  }
}

// Конкретная реализация алгоритма
class ConcreteAlgorithm extends AlgorithmTemplate {
  protected void step1() {
    System.out.println("Выполнение шага 1");
  }

  protected void step2() {
    System.out.println("Выполнение шага 2");
  }

  protected void step3() {
    System.out.println("Выполнение шага 3");
  }
}

// Пример использования
public class Main {
  public static void main(String[] args) {
    // Создаем объект конкретного алгоритма
    AlgorithmTemplate algorithm = new ConcreteAlgorithm();
```

```
        // Выполняем алгоритм
        algorithm.execute();
    }
}
```

В этом примере у нас есть абстрактный класс `AlgorithmTemplate`, который определяет шаблон алгоритма с помощью метода `execute()`, а также абстрактные методы `step1()`, `step2()` и `step3()`, которые представляют собой шаги алгоритма. Конкретные реализации алгоритма, такие как `ConcreteAlgorithm`, переопределяют эти методы для выполнения конкретных действий на каждом шаге. В результате получается гибкий и легко расширяемый код.

Команда

Паттерн Команда инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.



Шаблон команда (Command) — это поведенческий паттерн, который превращает запросы или операции в отдельные объекты. Это позволяет отделить отправителя запроса от получателя, что дает возможность легко настраивать, ставить в очередь и отменять операции, а также поддерживать операции с отложенным выполнением.

```java
// Интерфейс команды
interface Command {
  void execute();
}

// Конкретная команда для включения света
class LightOnCommand implements Command {
  private Light light;

  public LightOnCommand(Light light) {
    this.light = light;
  }

  public void execute() {
    light.turnOn();
  }
}

// Конкретная команда для выключения света
class LightOffCommand implements Command {
  private Light light;

  public LightOffCommand(Light light) {
    this.light = light;
  }

  public void execute() {
    light.turnOff();
  }
}

// Класс-отправитель команды
class RemoteControl {
  private Command command;

  public void setCommand(Command command) {
    this.command = command;
  }

  public void pressButton() {
    command.execute();
```

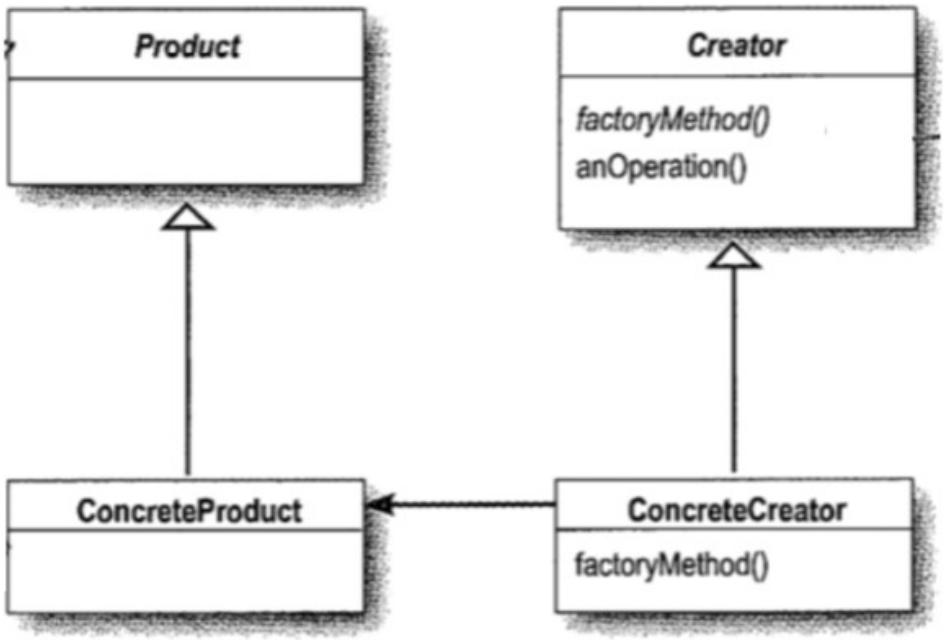
```
    }
}

// Класс, представляющий устройство - свет
class Light {
    public void turnOn() {
        System.out.println("Свет включен");
    }

    public void turnOff() {
        System.out.println("Свет выключен");
    }
}

// Пример использования
public class Main {
    public static void main(String[] args) {
        // Создаем устройство (свет)
        Light light = new Light();

        // Создаем команды для включения и выключения света
        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);

        // Создаем пульт управления и назначаем ему команды
        RemoteControl remoteControl = new RemoteControl();
        remoteControl.setCommand(lightOn); // Назначаем кнопке команду включения света

        // Нажимаем кнопку на пульте (включение света)
        remoteControl.pressButton();

        // Меняем команду на выключение света
        remoteControl.setCommand(lightOff); // Назначаем кнопке команду выключения света

        // Нажимаем кнопку на пульте (выключение света)
        remoteControl.pressButton();
    }
}
```

В этом примере мы создали интерфейс `Command`, который определяет метод `execute()` для выполнения операции. Затем мы создали конкретные команды для включения и выключения света (`LightOnCommand` и `LightOffCommand`). Класс `RemoteControl` представляет собой отправителя команды, который может назначать команду и вызывать ее выполнение. Таким образом, мы можем легко добавлять новые команды и изменять поведение приложения, не изменяя классы отправителя и получателя.

Фабричный метод

Простая фабрика

Паттерн Фабричный Метод определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом, Фабричный Метод делегирует операцию создания экземпляра субклассам.



Простая фабрика — это порождающий паттерн проектирования, который используется для создания объектов без необходимости раскрывать логику создания объекта для клиента. Основная идея заключается в том, чтобы вынести создание объектов в отдельную фабрику, чтобы клиентский код не зависел от конкретных классов объектов.

```java
// Интерфейс продукта
interface Product {
  void operation();
}

// Конкретные продукты
class ConcreteProductA implements Product {
  public void operation() {
    System.out.println("Операция в конкретном продукте A");
  }
}

class ConcreteProductB implements Product {
  public void operation() {
    System.out.println("Операция в конкретном продукте B");
  }
}

// Простая фабрика
class SimpleFactory {
  public Product createProduct(String type) {
    if (type.equals("A")) {
      return new ConcreteProductA();
    } else if (type.equals("B")) {
      return new ConcreteProductB();
    } else {
      throw new IllegalArgumentException("Неизвестный тип продукта");
    }
  }
}

// Пример использования простой фабрики
public class Main {
  public static void main(String[] args) {
    SimpleFactory factory = new SimpleFactory();

    Product productA = factory.createProduct("A");
    productA.operation();

    Product productB = factory.createProduct("B");
    productB.operation();
```

```
    }
}
```

В этом примере у нас есть интерфейс `Product`, который представляет общий интерфейс для всех продуктов. Простая фабрика `SimpleFactory` отвечает за создание конкретных продуктов (`ConcreteProductA` и `ConcreteProductB`) на основе переданного типа.

Абстрактная фабрика

Абстрактная фабрика — это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов. Этот паттерн расширяет простую фабрику, позволяя создавать не только отдельные объекты, но и целые семейства объектов.

```java
// Интерфейсы продуктов
interface AbstractProductA {
  void operationA();
}

interface AbstractProductB {
  void operationB();
}

// Конкретные продукты A
class ConcreteProductA1 implements AbstractProductA {
  public void operationA() {
    System.out.println("Операция в конкретном продукте A1");
  }
}

class ConcreteProductA2 implements AbstractProductA {
  public void operationA() {
    System.out.println("Операция в конкретном продукте A2");
  }
}

// Конкретные продукты B
class ConcreteProductB1 implements AbstractProductB {
  public void operationB() {
    System.out.println("Операция в конкретном продукте B1");
  }
}

class ConcreteProductB2 implements AbstractProductB {
  public void operationB() {
    System.out.println("Операция в конкретном продукте B2");
  }
}

// Абстрактная фабрика
interface AbstractFactory {
  AbstractProductA createProductA();
  AbstractProductB createProductB();
}
```

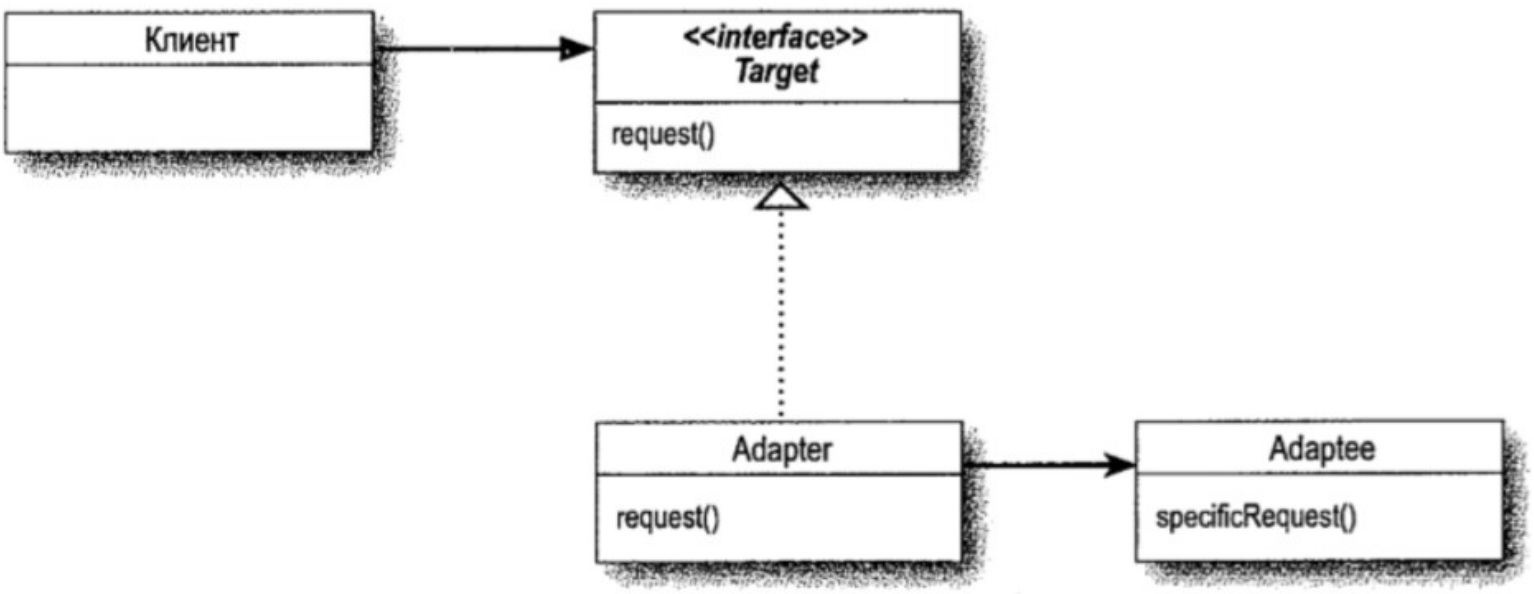
```
// Конкретные фабрики
class ConcreteFactory1 implements AbstractFactory {
  public AbstractProductA createProductA() {
    return new ConcreteProductA1();
  }

  public AbstractProductB createProductB() {
    return new ConcreteProductB1();
  }
}

class ConcreteFactory2 implements AbstractFactory {
  public AbstractProductA createProductA() {
    return new ConcreteProductA2();
  }

  public AbstractProductB createProductB() {
    return new ConcreteProductB2();
  }
}

// Пример использования абстрактной фабрики
public class Main {
  public static void main(String[] args) {
    AbstractFactory factory1 = new ConcreteFactory1();
    AbstractProductA productA1 = factory1.createProductA();
    AbstractProductB productB1 = factory1.createProductB();

    productA1.operationA();
    productB1.operationB();

    AbstractFactory factory2 = new ConcreteFactory2();
    AbstractProductA productA2 = factory2.createProductA();
    AbstractProductB productB2 = factory2.createProductB();

    productA2.operationA();
    productB2.operationB();
  }
}
```

Здесь мы имеем интерфейсы `AbstractProductA` и `AbstractProductB`, представляющие продукты семейства А и Б соответственно. `AbstractFactory` определяет методы для создания продуктов семейств А и Б. Конкретные фабрики (`ConcreteFactory1` и `ConcreteFactory2`) реализуют абстрактную фабрику для создания конкретных продуктов.

Основное различие между простой фабрикой и абстрактной фабрикой заключается в том, что простая фабрика создает отдельные объекты, а абстрактная фабрика создает семейства взаимосвязанных объектов.

Адаптер

Паттерн Адаптер преобразует интерфейс класса к другому интерфейсу, на который рассчитан клиент. Адаптер обеспечивает совместную работу классов, невозможную в обычных условиях из-за несовместимости интерфейсов.



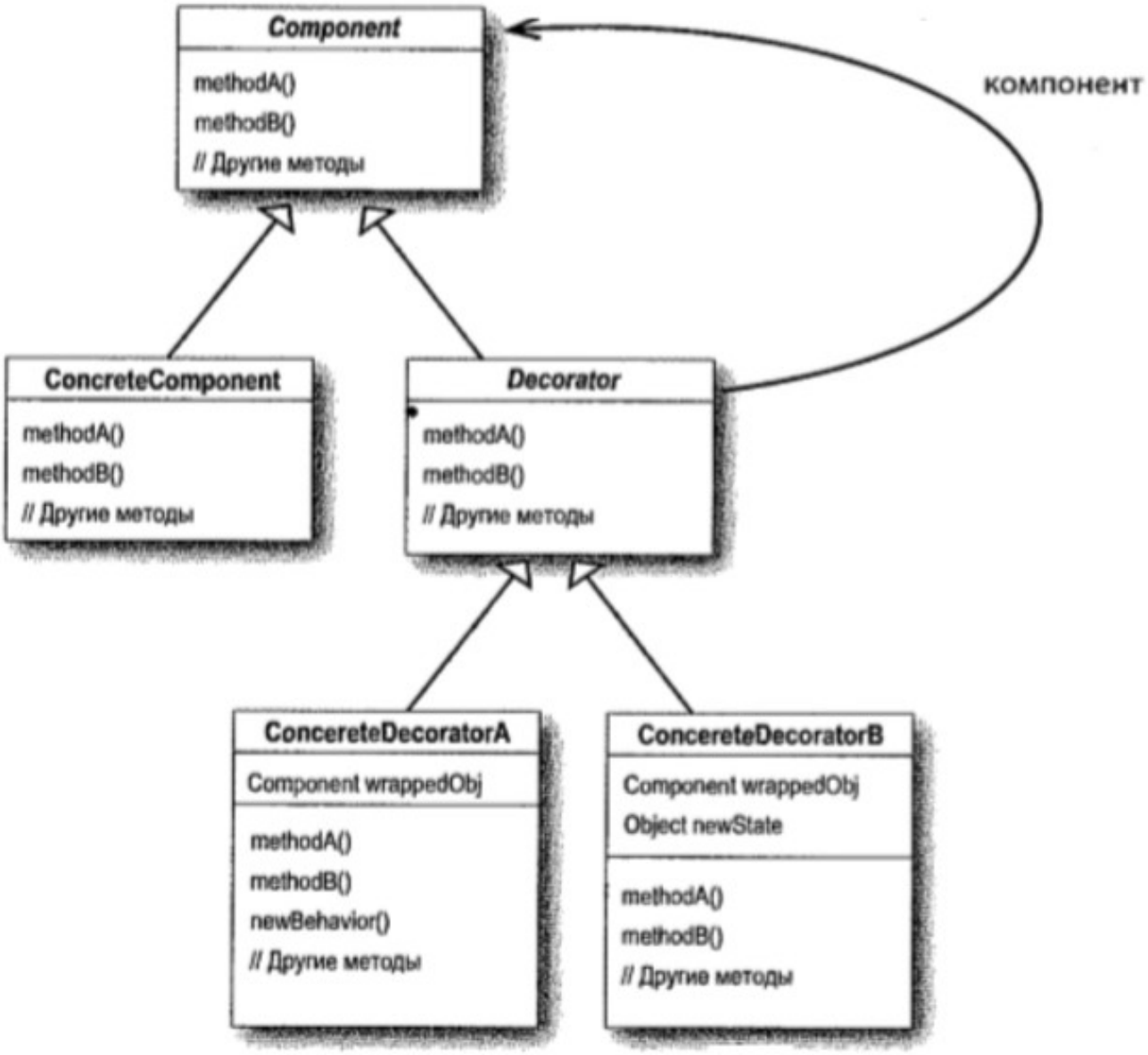
Паттерн адаптер (Adapter pattern) - это структурный шаблон проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он оборачивает один интерфейс в другой, что позволяет объектам совместно работать, не изменяя своих оригинальных интерфейсов.

```java
// Интерфейс, который ожидает клиентский код
interface Target {
  void request();
}

// Класс, который мы хотим адаптировать к интерфейсу Target
class Adaptee {
  public void specificRequest() {
    System.out.println("Adaptee: specificRequest()");
  }
}

// Адаптер, преобразующий интерфейс Adaptee в интерфейс Target
class Adapter implements Target {
  private Adaptee adaptee;

  public Adapter(Adaptee adaptee) {
    this.adaptee = adaptee;
  }

  @Override
  public void request() {
    adaptee.specificRequest();
  }
}

// Пример использования
public class Main {
  public static void main(String[] args) {
    // Создаем экземпляр объекта Adaptee
    Adaptee adaptee = new Adaptee();
    // Создаем адаптер и передаем ему объект Adaptee
    Target adapter = new Adapter(adaptee);
    // Вызываем метод request() через адаптер
    adapter.request();
  }
}
```

В данном примере `Adaptee` представляет класс с каким-то специфичным методом `specificRequest()`, а `Target` - интерфейс, который мы ожидаем использовать. `Adapter` оборачивает `Adaptee` и реализует интерфейс `Target`, перенаправляя вызовы метода `request()` к методу `specificRequest()`. Таким образом, клиентский код может использовать объект `Adapter`, не зная о существовании `Adaptee`.

Декоратор

Паттерн Декоратор динамически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности.



- Декораторы имеют тот же супертип, что и декорируемые объекты.
- Объект можно «завернуть» в один или несколько декораторов.
- Так как декоратор относится к тому же супертипу, что и декорируемый объект, мы можем передать декорированный объект вместо исходного.
- Декоратор добавляет свое поведения до и (или) после делегирования операций декорируемому объекту, выполняющему остальную работу.
- Объект может быть декорирован в любой момент времени, так что мы можем декорировать объекты динамически и с произвольным количеством декораторов.

Паттерн декоратор (Decorator pattern) - это структурный шаблон проектирования, который позволяет добавлять новое поведение или функциональность объекту динамически, не изменяя его основного класса. Он используется для расширения функциональности объектов без необходимости создания подклассов.

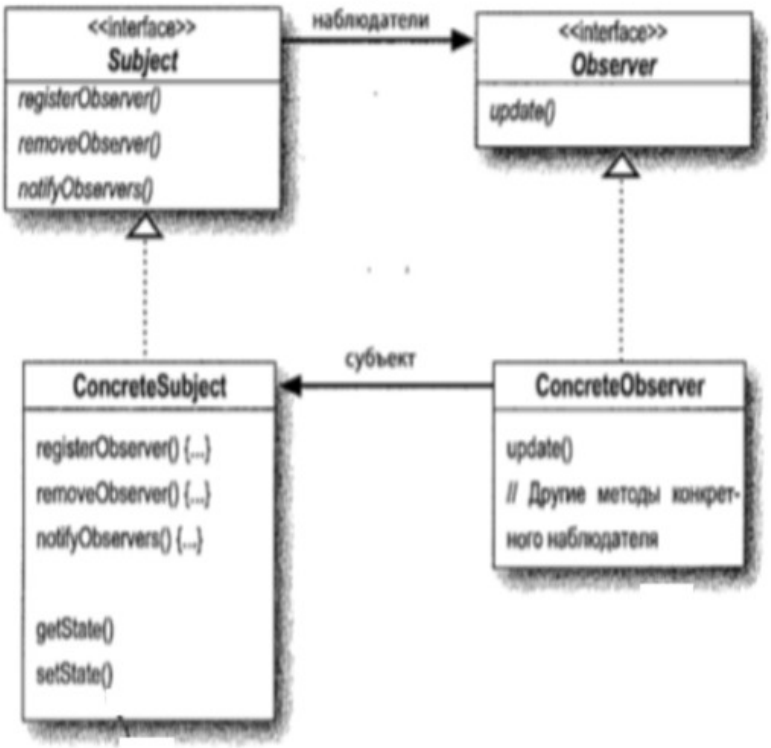
```java
// Интерфейс, представляющий компонент
interface Component {
  void operation();
}

// Конкретный компонент
class ConcreteComponent implements Component {
  @Override
  public void operation() {
    System.out.println("ConcreteComponent: operation()");
  }
}

// Базовый класс декоратора
abstract class Decorator implements Component {
  protected Component component;

  public Decorator(Component component) {
    this.component = component;
  }

  @Override
  public void operation() {
    component.operation();
  }
}

// Конкретный декоратор добавляет новую функциональность
class ConcreteDecorator extends Decorator {
  public ConcreteDecorator(Component component) {
    super(component);
  }

  @Override
  public void operation() {
    super.operation();
    addedBehavior();
  }

  private void addedBehavior() {
    System.out.println("ConcreteDecorator: addedBehavior()");
```

```
    }
}

// Пример использования
public class Main {
    public static void main(String[] args) {
        // Создаем экземпляр конкретного компонента
        Component component = new ConcreteComponent();
        // Оборачиваем компонент в декоратор
        Component decoratedComponent = new ConcreteDecorator(component);
        // Вызываем метод operation() через декорированный компонент
        decoratedComponent.operation();
    }
}
```

В данном примере `Component` представляет основной интерфейс, а `ConcreteComponent` - конкретный компонент, который реализует этот интерфейс. `Decorator` - абстрактный класс декоратора, который также реализует интерфейс `Component` и имеет ссылку на компонент. `ConcreteDecorator` - конкретный декоратор, который добавляет новую функциональность к объекту через композицию.

Таким образом, при вызове метода `operation()` через декорированный компонент, сначала будет вызван метод базового компонента, а затем добавленное поведение из декоратора.

Наблюдатель

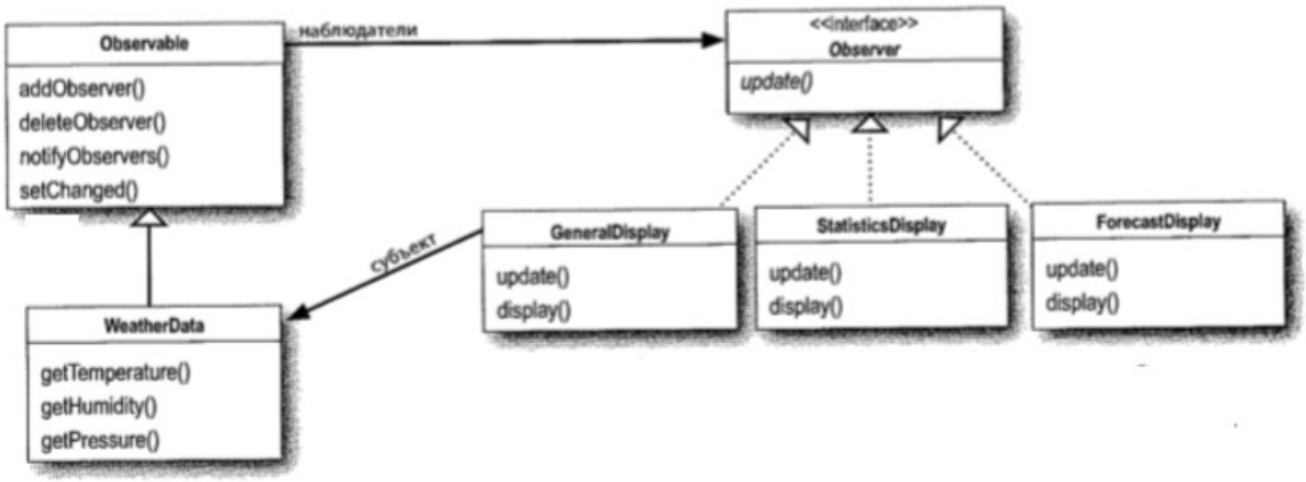


Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

Принцип проектирования

Стремитесь к слабой связанности взаимодействующих объектов.

Встроенная реализация в Java



Наблюдатель — это поведенческий паттерн проектирования, который определяет зависимость "один-ко-многим" между объектами таким образом, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически. Этот паттерн позволяет создать слабую связь между объектами, что способствует гибкой архитектуре приложения.

```java
import java.util.ArrayList;
import java.util.List;

// Интерфейс наблюдателя
interface Observer {
  void update(String message);
}

// Конкретный наблюдатель
class ConcreteObserver implements Observer {
  private String name;

  public ConcreteObserver(String name) {
    this.name = name;
  }

  public void update(String message) {
    System.out.println(name + " получил сообщение: " + message);
  }
}

// Интерфейс наблюдаемого объекта
interface Subject {
  void addObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers(String message);
}

// Конкретный наблюдаемый объект
class ConcreteSubject implements Subject {
  private List<Observer> observers = new ArrayList<>();

  public void addObserver(Observer observer) {
    observers.add(observer);
  }

  public void removeObserver(Observer observer) {
    observers.remove(observer);
  }

  public void notifyObservers(String message) {
```

```
    for (Observer observer : observers) {
      observer.update(message);
    }
  }
}

// Пример использования наблюдателя
public class Main {
  public static void main(String[] args) {
    ConcreteSubject subject = new ConcreteSubject();

    Observer observer1 = new ConcreteObserver("Наблюдатель 1");
    Observer observer2 = new ConcreteObserver("Наблюдатель 2");

    subject.addObserver(observer1);
    subject.addObserver(observer2);

    subject.notifyObservers("Новое сообщение!");
  }
}
```

В этом примере у нас есть интерфейс `Observer`, который представляет наблюдателя, способного получать уведомления о изменениях. `ConcreteObserver` — конкретная реализация наблюдателя.

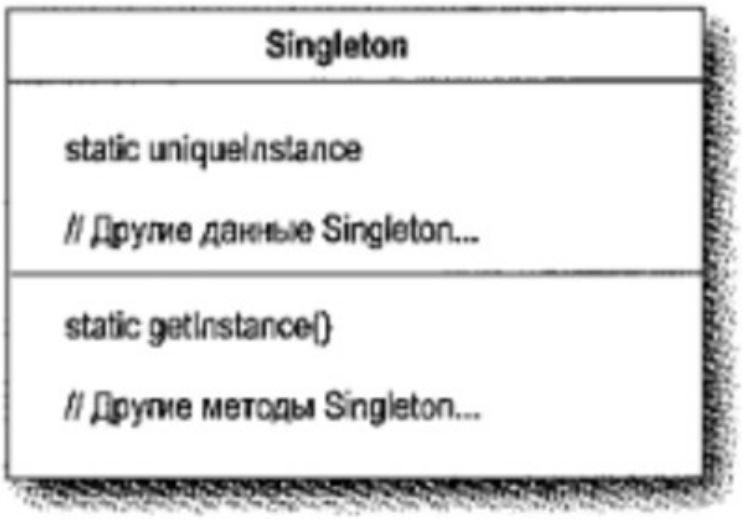
Интерфейс `Subject` определяет методы для добавления, удаления и уведомления наблюдателей. `ConcreteSubject` — конкретный наблюдаемый объект.

В `Main` мы создаем наблюдаемый объект `ConcreteSubject` и два наблюдателя (`ConcreteObserver`), добавляем их к наблюдаемому объекту и уведомляем их об изменении состояния объекта с помощью метода `notifyObservers`.

Таким образом, при изменении состояния наблюдаемого объекта все его наблюдатели автоматически уведомляются и обновляются, что позволяет поддерживать согласованность между объектами в системе.

Синглтон

Паттерн Одиночка гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.



Паттерн Singleton (Одиночка) - это порождающий шаблон проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.

```java
// Класс Singleton с ленивой инициализацией (Lazy Initialization)
public class Singleton {
  // Приватное статическое поле для хранения единственного экземпляра
  private static Singleton instance;

  // Приватный конструктор, чтобы предотвратить создание экземпляров извне
  private Singleton() {}

  // Публичный статический метод для получения единственного экземпляра
  public static Singleton getInstance() {
    // Если экземпляр еще не создан, создаем его
    if (instance == null) {
      instance = new Singleton();
    }
    // Возвращаем существующий экземпляр
    return instance;
  }

  // Другие методы и поля класса...
}

// Пример использования
public class Main {
  public static void main(String[] args) {
    // Получаем экземпляр Singleton
    Singleton singleton1 = Singleton.getInstance();
    // Еще раз получаем экземпляр Singleton
    Singleton singleton2 = Singleton.getInstance();

    // Проверяем, что оба экземпляра ссылаются на один и тот же объект
    if (singleton1 == singleton2) {
      System.out.println("singleton1 и singleton2 - это один и тот же объект
Singleton.");
    } else {
      System.out.println("singleton1 и singleton2 - разные объекты Singleton.");
    }
  }
}
```

В данном примере класс `Singleton` имеет приватное статическое поле `instance`, которое хранит единственный экземпляр класса. Конструктор класса `Singleton` приватный, чтобы предотвратить создание экземпляров извне. Метод `getInstance()` является статическим и возвращает этот единственный экземпляр, создавая его при необходимости.

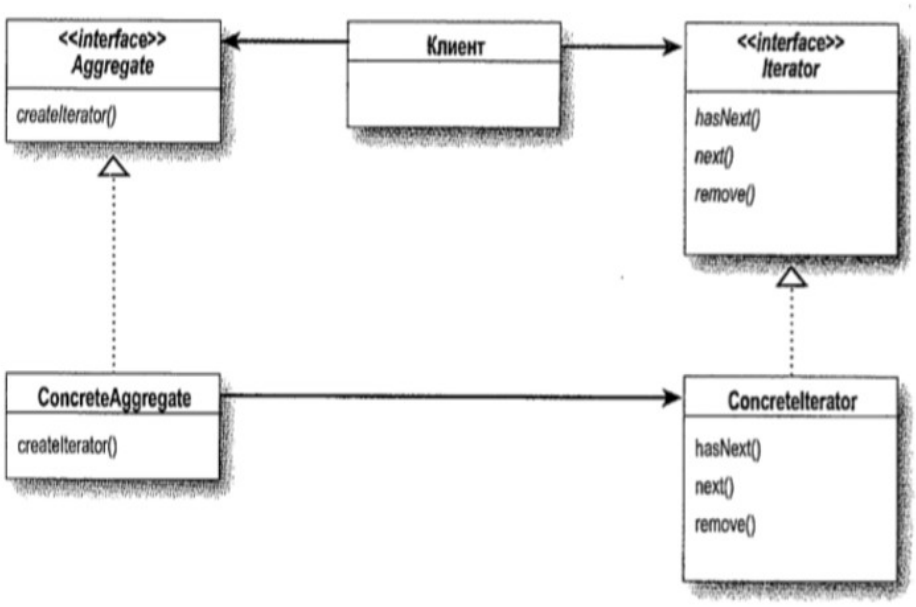
Таким образом, при вызове метода `getInstance()` будет возвращаться один и тот же экземпляр класса `Singleton` во всех точках программы.

Итератор

Паттерн Итератор предоставляет механизм последовательного перебора элементов коллекции без раскрытия ее внутреннего представления.

Принцип проектирования

Класс должен иметь только одну причину для изменения.



Итератор — это поведенческий паттерн проектирования, который предоставляет механизм последовательного доступа к элементам коллекции без раскрытия ее внутреннего представления. Он позволяет перебирать элементы коллекции без знания о ее внутренней структуре, что упрощает работу с коллекциями различных типов.

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

// Интерфейс итератора
interface MyIterator {
  boolean hasNext();
  Object next();
}

// Конкретный итератор
class ConcreteIterator implements MyIterator {
  private List<Object> elements;
  private int position = 0;

  public ConcreteIterator(List<Object> elements) {
    this.elements = elements;
  }

  public boolean hasNext() {
    return position < elements.size();
  }

  public Object next() {
    if (this.hasNext()) {
      return elements.get(position++);
    } else {
      return null;
    }
  }
}

// Интерфейс коллекции
interface MyCollection {
  MyIterator createIterator();
}

// Конкретная коллекция
class ConcreteCollection implements MyCollection {
  private List<Object> elements = new ArrayList<>();
```

```
    public void addElement(Object element) {
        elements.add(element);
    }

    public MyIterator createIterator() {
        return new ConcreteIterator(elements);
    }
}

// Пример использования итератора
public class Main {
    public static void main(String[] args) {
        ConcreteCollection collection = new ConcreteCollection();
        collection.addElement("Элемент 1");
        collection.addElement("Элемент 2");
        collection.addElement("Элемент 3");

        MyIterator iterator = collection.createIterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

В этом примере у нас есть интерфейс `MyIterator`, который определяет методы для перебора элементов коллекции. `ConcreteIterator` — конкретная реализация итератора.

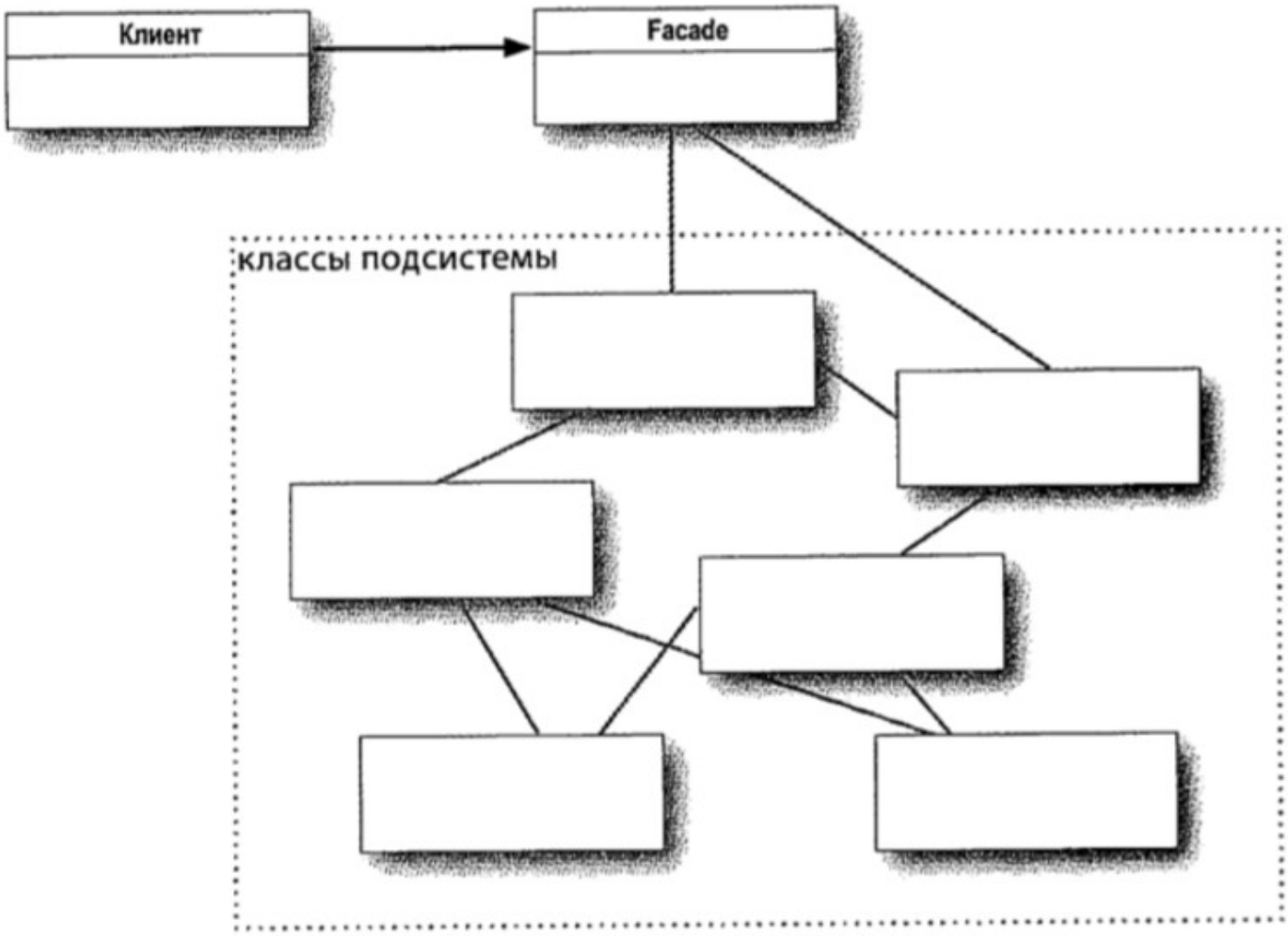
Интерфейс `MyCollection` определяет метод для создания итератора. `ConcreteCollection` — конкретная реализация коллекции.

В `Main` мы создаем коллекцию `ConcreteCollection`, добавляем в нее элементы и получаем итератор для перебора этих элементов. Затем мы перебираем элементы с помощью итератора и выводим их на экран.

Таким образом, паттерн "Итератор" позволяет перебирать элементы коллекции без знания о ее внутренней структуре, что делает работу с коллекциями более удобной и гибкой.

Фасад

Паттерн Фасад предоставляет унифицированный интерфейс к группе интерфейсов подсистемы. Фасад определяет высокоуровневый интерфейс, упрощающий работу с подсистемой.



Шаблон Фасад (Facade) - это структурный шаблон проектирования, который предоставляет унифицированный интерфейс к ряду интерфейсов в подсистеме. Фасад определяет высокоуровневый интерфейс, упрощающий работу с подсистемой, скрывая ее сложность.

```java
// Подсистема 1
class Subsystem1 {
  public void operation1() {
    System.out.println("Subsystem1: operation1()");
  }

  // Другие методы подсистемы...
}

// Подсистема 2
class Subsystem2 {
  public void operation2() {
    System.out.println("Subsystem2: operation2()");
  }

  // Другие методы подсистемы...
}

// Фасад, предоставляющий унифицированный интерфейс к подсистеме
class Facade {
  private Subsystem1 subsystem1;
  private Subsystem2 subsystem2;

  public Facade() {
    this.subsystem1 = new Subsystem1();
    this.subsystem2 = new Subsystem2();
  }

  // Методы, предоставляющие упрощенный интерфейс к подсистеме
  public void operation() {
    System.out.println("Facade: operation()");
    subsystem1.operation1();
    subsystem2.operation2();
  }

  // Другие методы фасада...
}

// Пример использования
public class Main {
  public static void main(String[] args) {
```

```
        // Используем фасад для работы с подсистемой
        Facade facade = new Facade();
        facade.operation();
    }
}
```

В данном примере классы `Subsystem1` и `Subsystem2` представляют подсистемы с различными операциями. Класс `Facade` предоставляет унифицированный интерфейс к этим подсистемам, скрывая их сложность. Метод `operation()` фасада предоставляет упрощенный интерфейс для выполнения операций в подсистемах.

Таким образом, при вызове метода `operation()` фасада будут выполнены соответствующие операции в подсистемах через унифицированный интерфейс.

Что такое гибкое проектирование

В технологии гибкой разработки дизайн программной системы документирован главным образом ее исходным кодом, что диаграммы, описывающие исходный код, – это дополнение к дизайну, а не сам дизайн.

Ароматы дизайна

Ароматы дизайна – запахи гниющей программы
О том, что программа начинает загнивать, можно узнать по появлению следующих ароматов:

- Жесткость
- Хрупкость
- Косность
- Вязкость
- Ненужная сложность
- Ненужные повторения
- Непрозрачность

Жесткость

Жесткость – это характеристика программы, затрудняющая внесение в нее изменений, даже самых простых. Дизайн жесткий, если единственное изменение вызывает целый каскад других изменений в зависимых модулях. Чем больше модулей приходится изменять, тем жестче дизайн.

Хрупкость

Хрупкость – это свойство программы повреждаться во многих местах при внесении единственного изменения. Зачастую новые проблемы возникают в частях, не имеющих концептуальной связи с той, что была изменена. Исправление одних проблем ведет к появлению новых.

Косность

Дизайн является косным, если он содержит части, которые могли бы оказаться полезны в других системах, но усилия и риски, сопряженные с попыткой отделить эти части от оригинальной системы, слишком велики. Печальная, но часто встречающаяся ситуация.

Вязкость

Вязкость проявляется в двух формах: вязкость программы и вязкость окружения.

Ненужная сложность

Дизайн попахивает ненужной сложностью, если содержит элементы, неиспользуемые в текущий момент. Готовясь к самым разным ситуациям, мы засоряем дизайн конструкциями, которые никогда не будут востребованы. Некоторые прогнозы оправдываются, большинство – нет. А между тем дизайн обременен всеми этими неиспользуемыми элементами. В результате программа становится сложной и малопонятной.

Ненужные повторения

Копирование и вставка полезны при редактировании текста, но могут оказывать разрушительное влияние при редактировании кода. Слишком часто системы включают десятки, а то и сотни повторяющихся фрагментов кода. Если в системе есть дублирующийся код, то задача ее изменения может потребовать значительных усилий. Ошибки, обнаруженные в повторяющемся блоке, должны быть исправлены во всех его копиях. Но, поскольку повторения немного отличаются друг от друга, то и исправления будут разными.

Непрозрачность

Непрозрачность – это трудность модуля для понимания. Код может быть ясным и выразительным или темным и запутанным. Код, эволюционирующий со временем, постепенно становится все более и более непрозрачным.

Итак, что такое гибкое проектирование?

Это процесс, а не разовое событие. Это постоянное применение определенных принципов, паттернов и методик для улучшения структуры, и понятности программы.

SOLID

Перечислим эти принципы:

- Принцип единственной ответственности (Single-Responsibility Principle – SRP).
- Принцип открытости/закрытости (Open/Closed Principle – OCP).
- Принцип подстановки Лисков (Liskov Substitution Principle – LSP).
- Принцип инверсии зависимости (Dependency-Inversion Principle – DIP).
- Принцип разделения интерфейсов (Interface Segregation Principle – ISP).

Принцип единственной ответственности (Single-Responsibility Principle – SRP)

Принцип единственной ответственности (SRP): Каждый класс должен иметь только одну причину для изменения. Это означает, что класс должен быть ответственен только за одну четко определенную часть функциональности программы.

Приведу сначала пример правильного соблюдения Принципа единственной ответственности (SRP), а затем примеры его нарушения.

Пример соблюдения SRP:

```java
// Пример класса, соблюдающего SRP
class Employee {
  private String name;
  private double salary;
  private String department;

  public Employee(String name, double salary, String department) {
    this.name = name;
    this.salary = salary;
    this.department = department;
  }

  // Методы, относящиеся только к управлению сотрудниками
  public void calculateSalary() {
    // Расчет зарплаты сотрудника
  }

  public void assignDepartment(String department) {
    // Назначение сотруднику нового департамента
  }
}
```

В этом примере класс `Employee` сосредоточен только на управлении сотрудниками и не содержит никакой логики, не относящейся к этой области. Он имеет только две ответственности: расчет зарплаты и назначение департамента сотруднику.

Примеры нарушения SRP:

1. Класс, объединяющий управление данными и их представлением:
```java
class Employee {
    private String name;
    private double salary;

    public void saveToDatabase() {
        // Сохранение информации о сотруднике в базу данных
    }

    public void displayData() {
        // Отображение информации о сотруднике на экране
    }
}
```
Этот класс нарушает SRP, так как он отвечает и за сохранение данных в базу данных, и за их отображение на экране. Лучше разделить эти функции на разные классы.

2. Класс, выполняющий слишком много операций:
```java
class Employee {
    private String name;
    private double salary;
    private String department;

    public void processSalary() {
        // Обработка зарплаты сотрудника
    }

    public void sendNotification() {
        // Отправка уведомления сотруднику
    }

    public void saveToFile() {
        // Сохранение информации о сотруднике в файл
    }
}
```

В этом примере класс `Employee` нарушает SRP, так как он выполняет сразу три разные операции: обработку зарплаты, отправку уведомления и сохранение данных в файл. Лучше разделить эти операции на отдельные классы.

3. Класс, выполняющий дополнительные действия:
```java
class Employee {
    private String name;
    private double salary;

    public void processData() {
        // Обработка данных сотрудника
    }

    public void logActivity() {
        // Логирование действий сотрудника
    }
}
```
Этот класс нарушает SRP, так как помимо обработки данных сотрудника он также отвечает за логирование действий. Лучше вынести логирование в отдельный класс.

Принцип открытости/закрытости (Open/Closed Principle – OCP)

Принцип открытости/закрытости (OCP): Программные сущности, такие как классы, модули и функции, должны быть открыты для расширения, но закрыты для модификации. Это означает, что при добавлении новой функциональности программа должна расширяться, а не изменяться.

Пример соблюдения OCP:

```java
// Пример интерфейса и классов, соблюдающих OCP
interface Shape {
  double calculateArea();
}

class Circle implements Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  @Override
  public double calculateArea() {
    return Math.PI * radius * radius;
  }
}

class Rectangle implements Shape {
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  @Override
  public double calculateArea() {
    return width * height;
  }
}
```

```

В этом примере, добавление новых форм (например, треугольников) не потребует изменения существующего кода. Вместо этого, вы просто создаете новый класс, реализующий интерфейс `Shape`.

Примеры нарушения OCP:

1. Модификация существующего класса для добавления нового функционала:
```java
class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    // Нарушение OCP: добавление нового функционала изменяет существующий класс
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}
```
В этом примере, если требуется добавить новый функционал (например, расчет периметра), вы вынуждены изменить существующий класс `Circle`, нарушая принцип OCP.

2. Использование условных операторов для выбора различных вариантов функциональности:
```java
class Shape {
    private String type;
    private double value;

    public Shape(String type, double value) {
        this.type = type;
        this.value = value;
```
```

```
    }

    // Нарушение OCP: добавление новых типов требует изменения этого класса
    public double calculateArea() {
      if (type.equals("circle")) {
        return Math.PI * value * value;
      } else if (type.equals("rectangle")) {
        // Расчет площади прямоугольника
      }
      // Другие условия для других типов фигур...
    }
  }
```

В этом примере, чтобы добавить новые типы фигур, необходимо изменять метод `calculateArea()`, нарушая принцип OCP.

3. Наследование с переопределением методов:

```java
class Shape {
  // Базовый класс фигуры
}

class Circle extends Shape {
  // Реализация для круга
}

class Rectangle extends Shape {
  // Реализация для прямоугольника
}

// Нарушение OCP: для добавления новой фигуры нужно изменять базовый класс и все его потомки
class Triangle extends Shape {
  // Реализация для треугольника
}
```

В этом примере, добавление новых фигур требует изменения существующей иерархии классов, нарушая принцип OCP. Лучше использовать композицию вместо наследования.

Принцип подстановки Лисков (Liskov Substitution Principle – LSP).

Принцип подстановки Лисков (LSP): Объекты в программе должны быть заменяемыми своими наследниками без изменения правильности выполнения программы. Если класс A является подтипом класса B, то объекты класса B могут быть заменены объектами класса A без нарушения корректности программы.

Пример соблюдения LSP:

```java
// Пример иерархии классов, соблюдающих LSP
class Shape {
  protected double width;
  protected double height;

  public void setWidth(double width) {
    this.width = width;
  }

  public void setHeight(double height) {
    this.height = height;
  }

  public double calculateArea() {
    return width * height;
  }
}

class Rectangle extends Shape {
  // Реализация для прямоугольника
}

class Square extends Shape {
  // Реализация для квадрата
  @Override
  public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width); // Высота также устанавливается равной стороне квадрата
  }
```

```
    @Override
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height); // Ширина также устанавливается равной стороне квадрата
    }
}
```

В этом примере класс `Square` является подтипом класса `Rectangle`, и его методы `setWidth()` и `setHeight()` переопределены таким образом, чтобы они корректно работали для квадрата, сохраняя инвариант (ширина всегда равна высоте). Это соблюдение Принципа подстановки Лисков.

Пример нарушения LSP:

```java
// Пример нарушения LSP
class Bird {
    public void fly() {
        // Реализация полета
    }
}

class Ostrich extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Страус не может летать");
    }
}
```

В этом примере класс `Ostrich` наследует от класса `Bird`, но переопределяет метод `fly()`, выбрасывая исключение, потому что страус не может летать. Это нарушение Принципа подстановки Лисков, так как оно нарушает ожидаемое поведение подкласса (то есть, ожидается, что все подклассы `Bird` могут летать).

Принцип инверсии зависимости (Dependency-Inversion Principle – DIP)

Принцип инверсии зависимости (DIP): Зависимости в программе должны строиться на абстракциях, а не на конкретных реализациях. Это означает, что модули верхнего уровня не должны зависеть от модулей нижнего уровня, а оба типа модулей должны зависеть от абстракций.

Пример соблюдения DIP:

```java
// Пример соблюдения DIP с использованием интерфейсов
interface Switchable {
    void turnOn();

    void turnOff();
}

class Light implements Switchable {
    @Override
    public void turnOn() {
        // Логика включения света
    }

    @Override
    public void turnOff() {
        // Логика выключения света
    }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void toggle() {
        if (device != null) {
            if (isOn()) {
                device.turnOff();
            } else {
```

```
            device.turnOn();
        }
    }
  }

  private boolean isOn() {
    // Логика проверки состояния устройства
    return false;
  }
}
```

В этом примере класс `Switch` зависит от абстракции `Switchable` через интерфейс, а не от конкретной реализации. Это соблюдение Принципа инверсии зависимости.

Пример нарушения DIP:

```java
// Пример нарушения DIP
class Light {
  public void turnOn() {
    // Логика включения света
  }

  public void turnOff() {
    // Логика выключения света
  }
}

class Switch {
  private Light light;

  public Switch(Light light) {
    this.light = light;
  }

  public void toggle() {
    if (light != null) {
      if (isOn()) {
        light.turnOff();
      } else {
        light.turnOn();
```

```
        }
      }
    }

    private boolean isOn() {
      // Логика проверки состояния устройства
      return false;
    }
}
```

В этом примере класс `Switch` прямо зависит от конкретной реализации `Light`, что нарушает Принцип инверсии зависимости. Лучше было бы зависеть от абстракции `Switchable`, что позволило бы легче внедрять различные устройства, совместимые с интерфейсом `Switchable`.

Принцип разделения интерфейсов (Interface Segregation Principle – ISP).

Принцип разделения интерфейсов (ISP): Необходимо создавать маленькие и специфические интерфейсы, предназначенные для конкретных клиентов. Это означает, что клиенты не должны зависеть от методов, которые они не используют.

Пример соблюдения ISP:

```java
// Пример соблюдения ISP
interface Switchable {
  void turnOn();

  void turnOff();
}

interface Dimmable {
  void setBrightness(int brightness);
}

class Light implements Switchable, Dimmable {
  @Override
  public void turnOn() {
    // Логика включения света
  }

  @Override
  public void turnOff() {
    // Логика выключения света
  }

  @Override
  public void setBrightness(int brightness) {
    // Логика установки яркости света
  }
}
```

В этом примере интерфейсы `Switchable` и `Dimmable` разделены таким образом, что класс `Light` может реализовать только те методы, которые ему нужны. Это соблюдение Принципа разделения интерфейсов.

Пример нарушения ISP:

```java
// Пример нарушения ISP
interface Machine {
  void turnOn();

  void turnOff();

  void print();

  void scan();
}

class MultiFunctionPrinter implements Machine {
  @Override
  public void turnOn() {
    // Логика включения устройства
  }

  @Override
  public void turnOff() {
    // Логика выключения устройства
  }

  @Override
  public void print() {
    // Логика печати документа
  }

  @Override
  public void scan() {
    // Логика сканирования документа
  }
}
```

В этом примере класс `MultiFunctionPrinter` реализует интерфейс `Machine`, который содержит методы `print()` и `scan()`. Однако, не все устройства, реализующие `Machine`, могут поддерживать как печать, так и сканирование. Это нарушение Принципа разделения интерфейсов, так как устройства, которые не поддерживают печать или сканирование, будут вынуждены реализовывать эти методы, которые для них не имеют смысла. Лучше было бы разделить интерфейс `Machine` на более мелкие интерфейсы, например, `Printer` и `Scanner`, чтобы каждое устройство реализовывало только необходимый функционал.