

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ

по учебной практике

Тема: **Шаблоны проектирования в языке Java**

Студент гр. 2307_____ **Стукен В.А.**

Санкт-Петербург

2024

Оглавление

Поведенческий шаблон проектирования Observer (наблюдатель)	3
Назначение	3
Область применения	3
Порождающий шаблон проектирования Factory (фабрика).....	9
Назначение	9
Область применения	9
Структурный шаблон проектирования Decorator (декоратор).....	14
Назначение	14
Область применения	14
Вывод	20

Поведенческий шаблон проектирования Observer (наблюдатель)

Реализует у класса механизм, который позволяет объекту этого класса получать оповещения об изменении состояния других объектов и тем самым наблюдать за ними.

Классы, на события которых другие классы подписываются, называются субъектами (Subjects), а подписывающиеся классы называются наблюдателями.

Назначение

Определяет зависимость типа один ко многим между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Область применения

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует как минимум один объект, рассылающий сообщения;
- имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;
- позволяет избежать сильного зацепления взаимодействующих классов.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

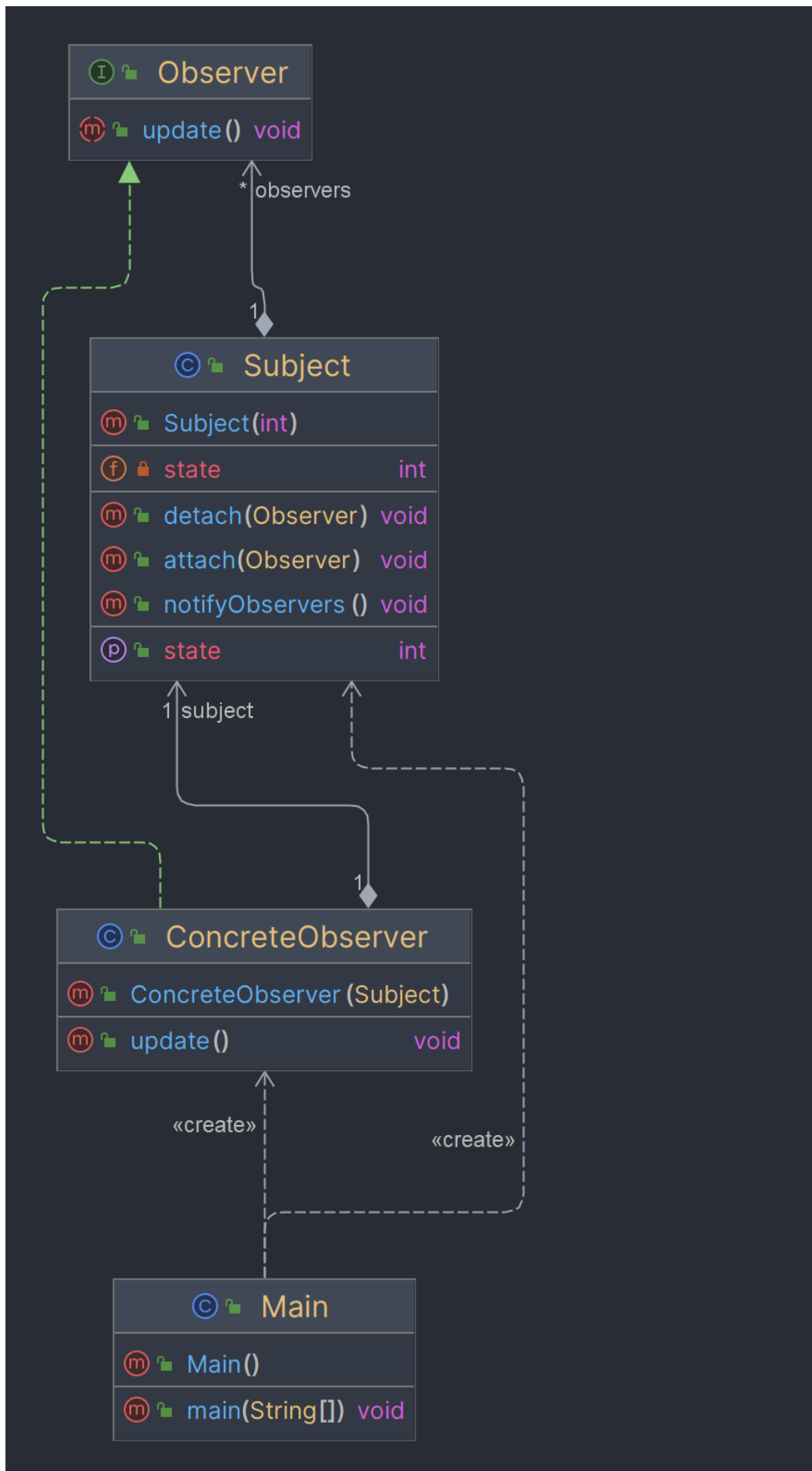
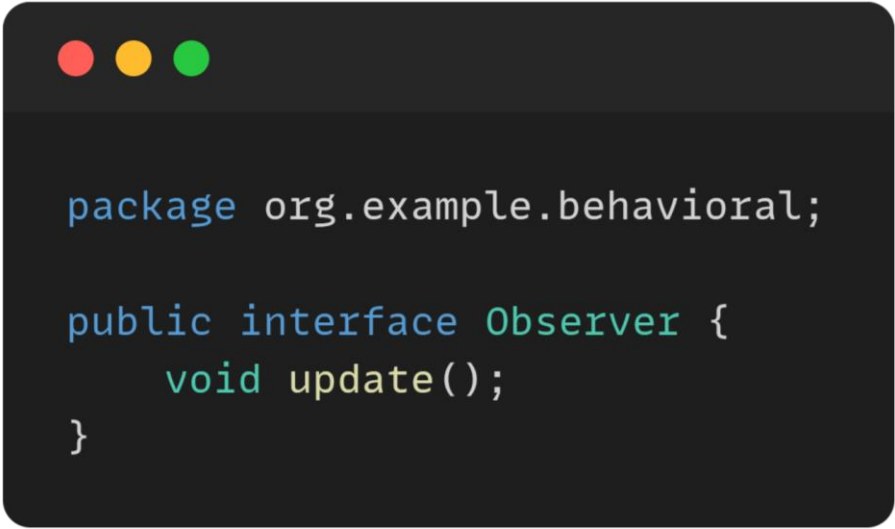


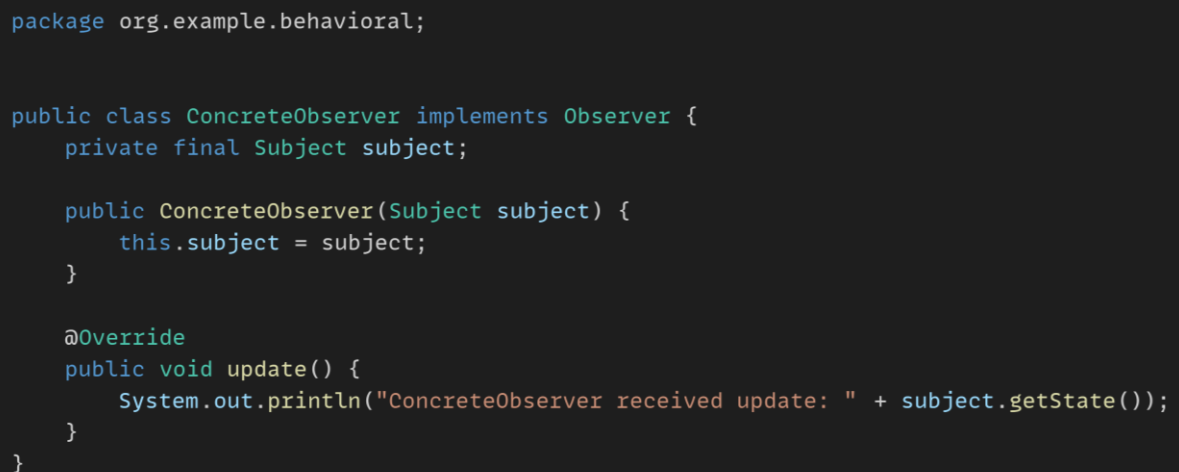
Рисунок 1 Диаграмма классов



```
package org.example.behavioral;

public interface Observer {
    void update();
}
```

Рисунок 2 Интерфейс Observer



```
package org.example.behavioral;

public class ConcreteObserver implements Observer {
    private final Subject subject;

    public ConcreteObserver(Subject subject) {
        this.subject = subject;
    }

    @Override
    public void update() {
        System.out.println("ConcreteObserver received update: " + subject.getState());
    }
}
```

Рисунок 3 класс ConcreteObserver, реализующий интерфейс Observer

```
package org.example.behavioral;

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

public class Subject {
    private final List<Observer> observers = new ArrayList<>();
    private int state;

    public Subject(int state) {
        this.state = state;
    }

    public int getState() {
        return state;
    }

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

    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();
        }
    }
}
```

Рисунок 4 класс Subject

```
package org.example.behavioral;

public class Main {
    public static void main(String[] args) {
        // Create a ConcreteSubject with initial state = 3
        Subject stockMarket = new Subject(3);

        // Create some ConcreteObservers
        ConcreteObserver investor1 = new ConcreteObserver(stockMarket);
        ConcreteObserver investor2 = new ConcreteObserver(stockMarket);

        // Register the ConcreteObservers with the ConcreteSubject
        stockMarket.attach(investor1);
        stockMarket.attach(investor2);

        // Notify the ConcreteObservers of a state change
        stockMarket.notifyObservers();

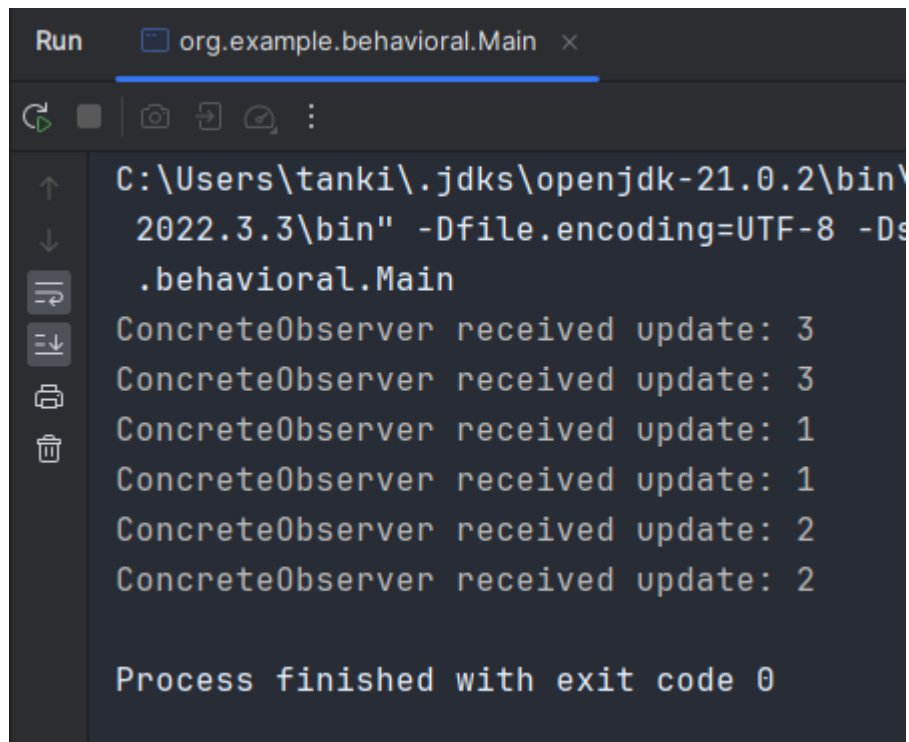
        // Set state = 1
        stockMarket.setState(1);

        // Remove an observer
        stockMarket.detach(investor1);

        // Set state = 2
        stockMarket.setState(2);

        // Notify the ConcreteObservers of another state change
        stockMarket.notifyObservers();
    }
}
```

Рисунок 5 класс Main, показывающий использование шаблона

The image shows a screenshot of an IDE's Run console. At the top, the title bar says "Run" and the active tab is "org.example.behavioral.Main". Below the title bar is a toolbar with icons for running, debugging, and other actions. The main area of the console displays the following text:

```
C:\Users\tanki\.jdk\openjdk-21.0.2\bin\java.exe -Dfile.encoding=UTF-8 -Djava.class.path=.\behavioral.Main
ConcreteObserver received update: 3
ConcreteObserver received update: 3
ConcreteObserver received update: 1
ConcreteObserver received update: 1
ConcreteObserver received update: 2
ConcreteObserver received update: 2

Process finished with exit code 0
```

Рисунок 6 результат работы программы

Порождающий шаблон проектирования Factory (фабрика)

Фабрика — это шаблон проектирования, который помогает решить проблему создания различных объектов в зависимости от некоторых условий.

Назначение

Основная цель шаблона "Фабрика" заключается в том, чтобы отделить процесс создания объектов от их использования. Это позволяет:

1. Избежать жесткой привязки к конкретным классам: Создание объектов осуществляется через интерфейс или абстрактный класс, что позволяет использовать различные реализации без изменения кода клиента.
2. Упростить добавление новых классов: Новые типы объектов могут быть добавлены без модификации существующего кода.
3. Контроль над созданием объектов: Легче управлять созданием объектов, например, внедрять логирование, кэширование или другие аспекты.

Область применения

Шаблон "Фабрика" полезен в следующих ситуациях:

1. Когда заранее неизвестны типы и зависимости объектов, с которыми будет работать программа: Это позволяет создавать объекты по мере необходимости, обеспечивая гибкость и расширяемость.
2. Когда система должна быть независимой от процесса создания, композиции и представления продуктов: Это важно для соблюдения принципа открытости/закрытости (Open/Closed Principle).
3. Для управления жизненным циклом объектов: Если создание объектов включает сложную логику (например, настройки, подключение к базам данных и т. д.), "Фабрика" упрощает это.
4. Когда требуется инкапсуляция логики создания объектов: Например, если нужно скрыть детали создания объекта от клиента.

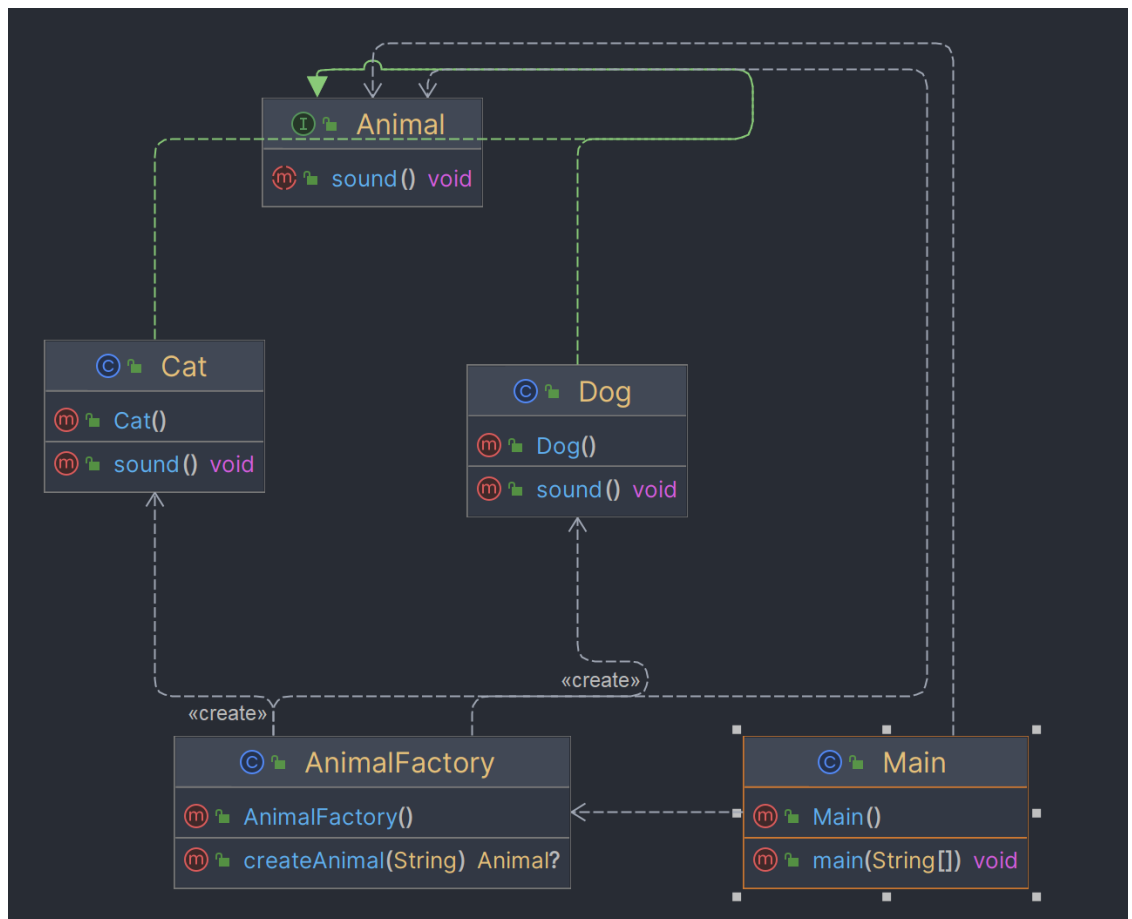
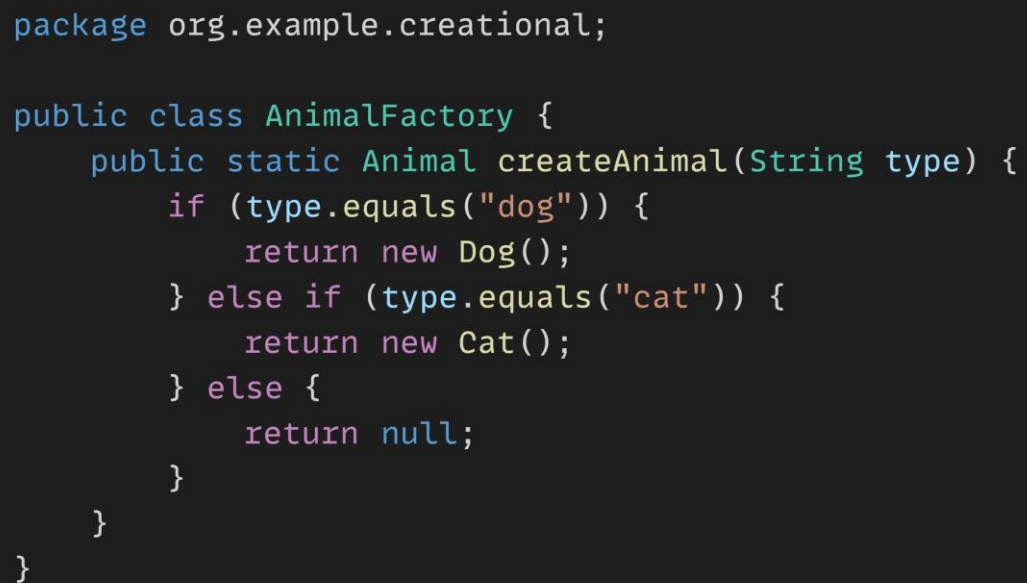


Рисунок 7 диаграмма классов

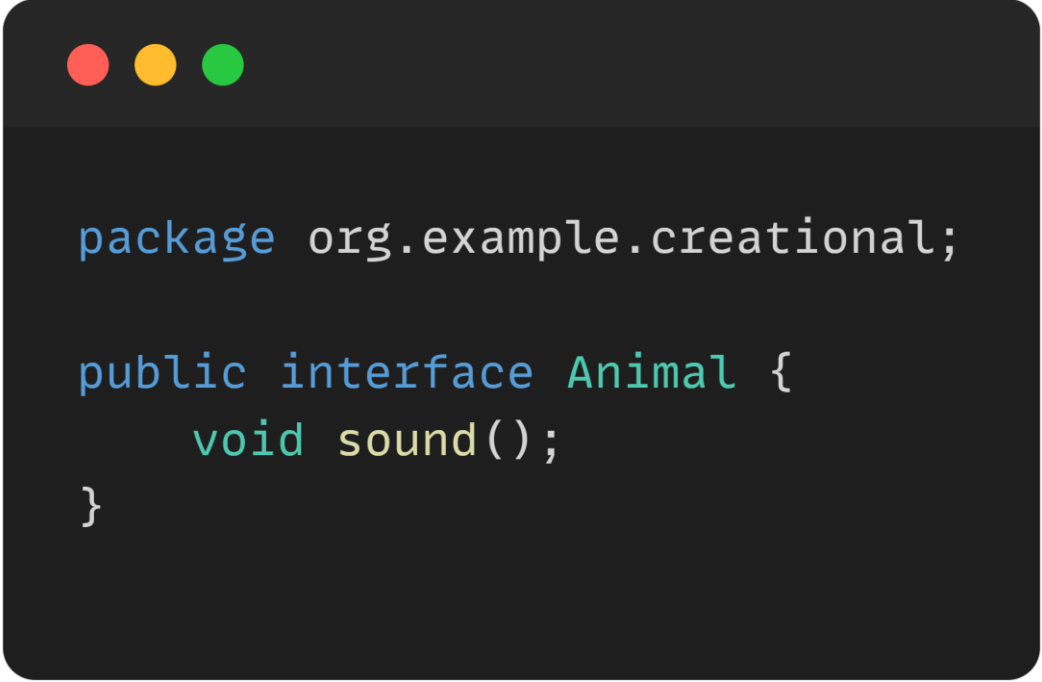


```
package org.example.creational;

public class AnimalFactory {
    public static Animal createAnimal(String type) {
        if (type.equals("dog")) {
            return new Dog();
        } else if (type.equals("cat")) {
            return new Cat();
        } else {
            return null;
        }
    }
}
```

codesnap.dev

Рисунок 8 класс AnimalFactory

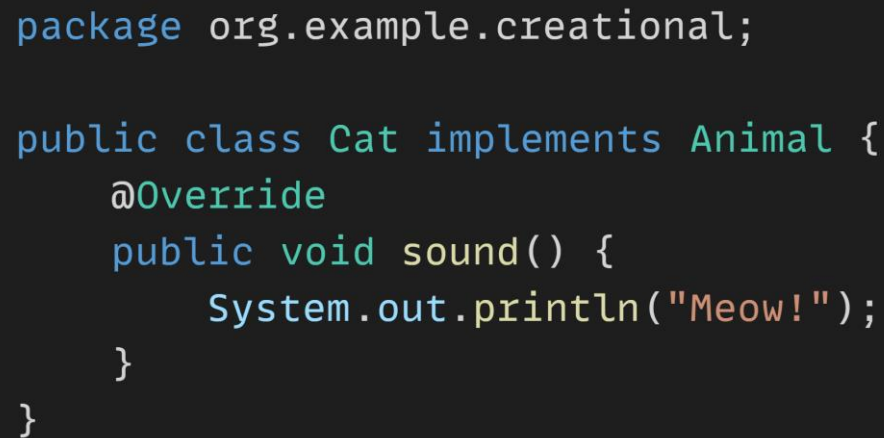


```
package org.example.creational;

public interface Animal {
    void sound();
}
```

codesnap.dev

Рисунок 9 базовый класс Animal

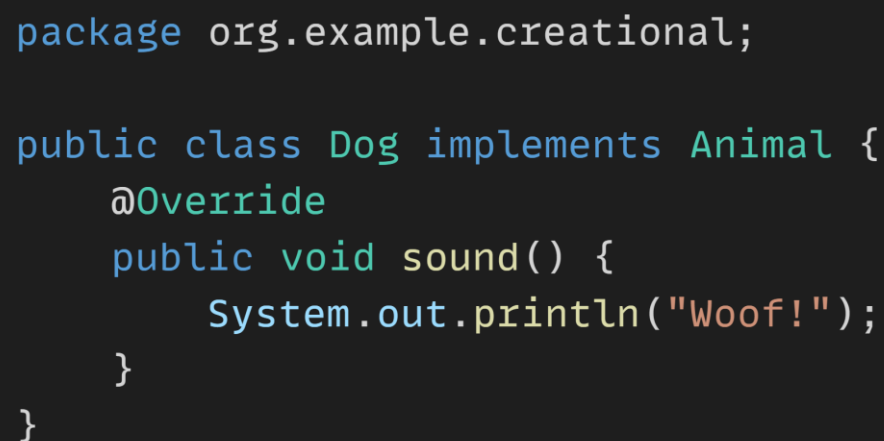


```
package org.example.creational;

public class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Meow!");
    }
}
```

codesnap.dev

Рисунок 10 класс Cat



```
package org.example.creational;

public class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Woof!");
    }
}
```

codesnap.dev

Рисунок 11 класс Dog



```
package org.example.creational;

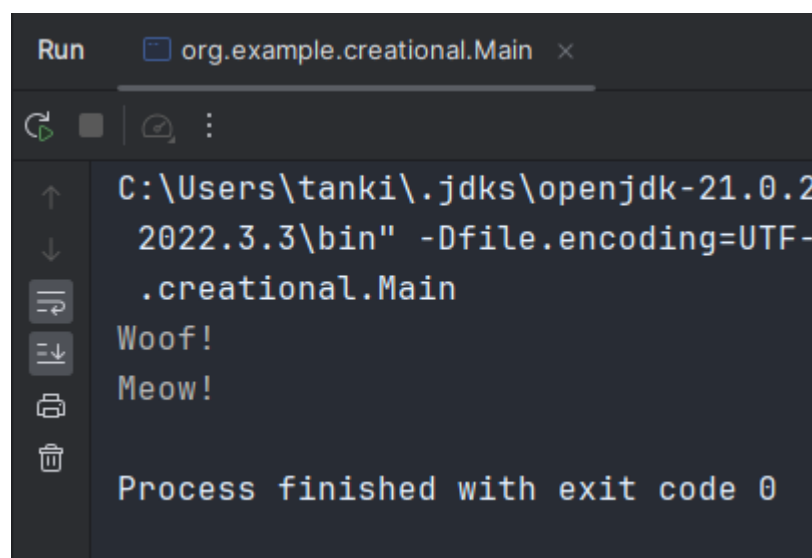
public class Main {
    public static void main(String[] args) {
        Animal dog = AnimalFactory.createAnimal("dog");
        Animal cat = AnimalFactory.createAnimal("cat");

        if (dog != null) {
            dog.sound(); // Output: Woof!
        } else {
            System.out.println("Unknown animal type");
        }

        if (cat != null) {
            cat.sound(); // Output: Meow!
        } else {
            System.out.println("Unknown animal type");
        }
    }
}
```

codesnap.dev

Рисунок 12 класс Main, иллюстрирующий использование шаблона



```
Run  org.example.creational.Main x
C:\Users\tanki\.jdk\openjdk-21.0.2
2022.3.3\bin" -Dfile.encoding=UTF-8
.org.example.creational.Main
Woof!
Meow!

Process finished with exit code 0
```

Рисунок 13 результат работы программы

Структурный шаблон проектирования Decorator (декоратор)

Шаблон проектирования "Декоратор" (Decorator) — это структурный паттерн, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в классы-обертки (декораторы). Декораторы предоставляют гибкую альтернативу наследованию для расширения функциональности.

Назначение

Основная цель шаблона "Декоратор" заключается в том, чтобы добавлять объектам новую функциональность без изменения их кода. Это позволяет:

1. Избежать монолитных классов: Вместо создания одного класса с многочисленными функциями, можно создать несколько маленьких классов-декораторов.
2. Расширять поведение объектов: Декораторы позволяют добавлять поведение объектам на лету, без необходимости создания подклассов.
3. Упрощать тестирование и повторное использование кода: Поведения, оформленные в виде декораторов, могут быть легко протестированы и использованы повторно.

Область применения

Шаблон "Декоратор" полезен в следующих ситуациях:

1. Когда нужно добавлять поведение объектам на лету: Например, добавление различных визуальных эффектов к пользовательскому интерфейсу.
2. Когда невозможно использовать наследование: Либо из-за ограничения языка, либо из-за необходимости комбинировать несколько поведений в различных сочетаниях.
3. Когда необходимо разделить обязанности между классами: Это помогает соблюдать принцип единственной ответственности, делая код более читабельным и поддерживаемым.

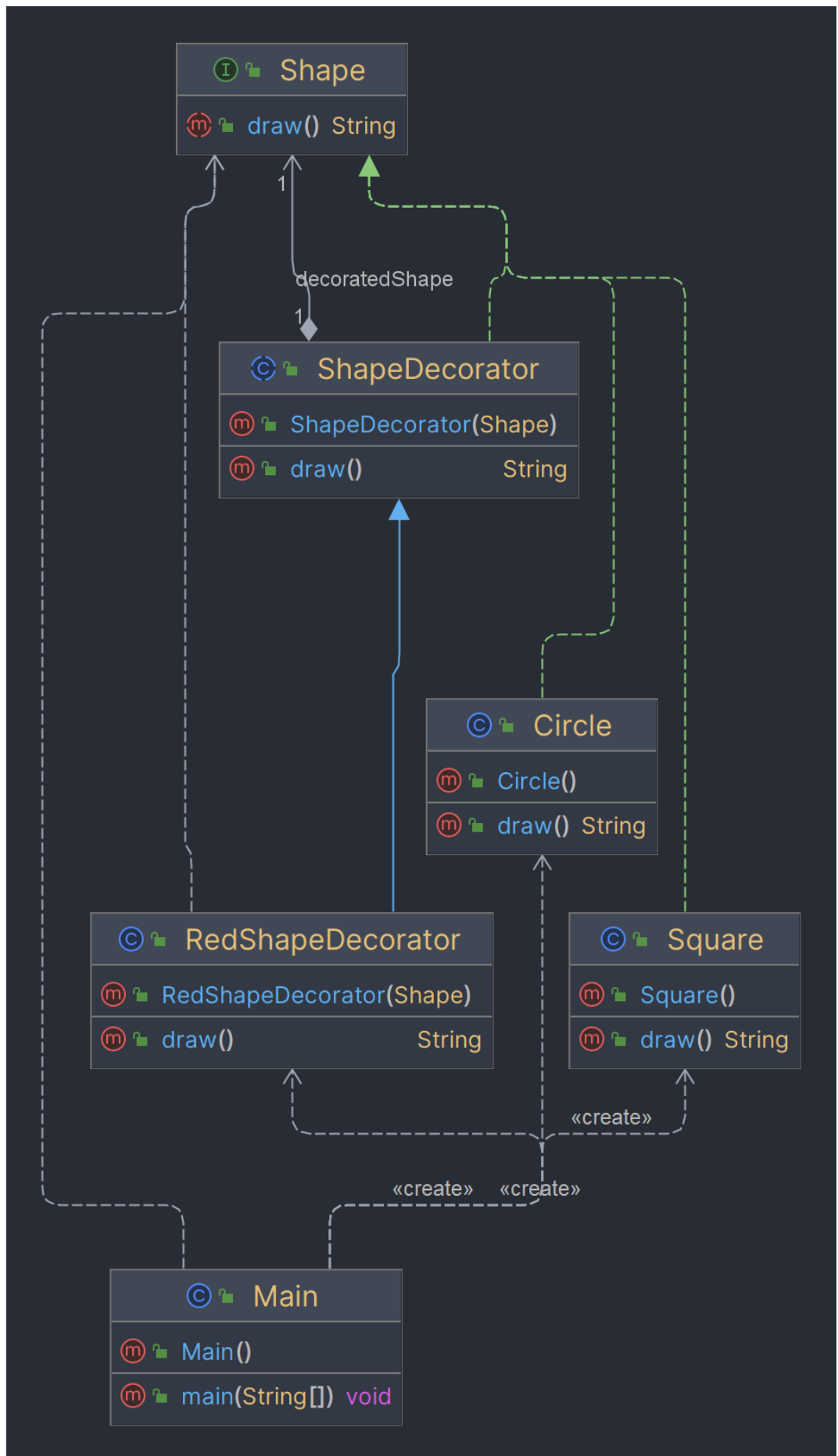


Рисунок 14 диаграмма классов

```
package org.example.structural;

public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    @Override
    public String draw() {
        return decoratedShape.draw();
    }
}
```

codesnap.dev

Рисунок 15 класс ShapeDecorator

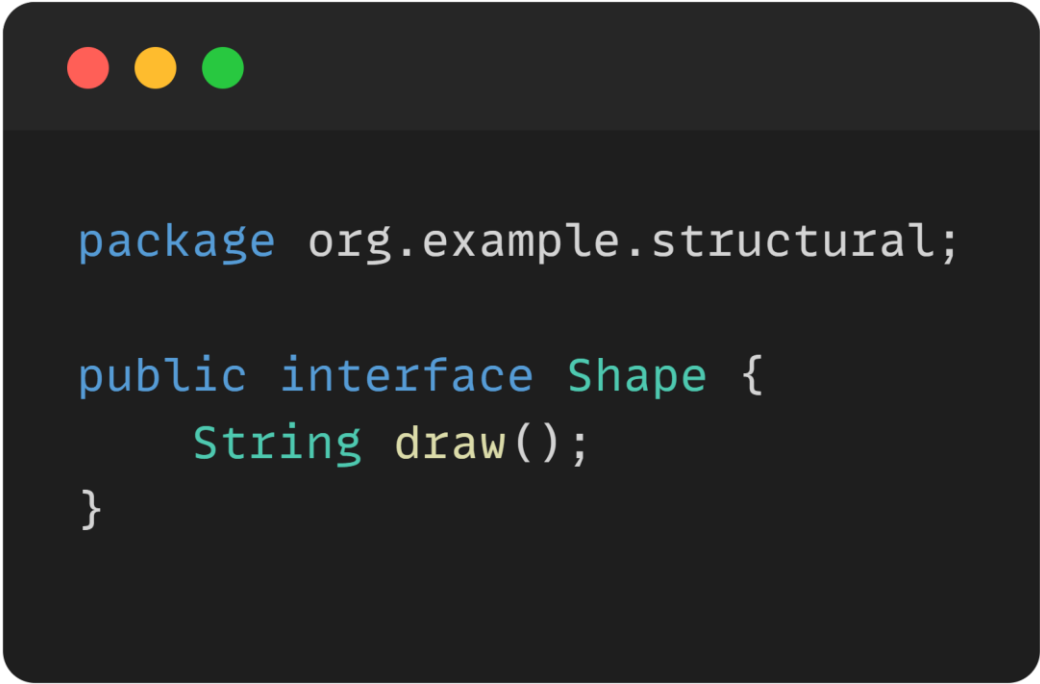
```
package org.example.structural;

public class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public String draw() {
        return "Drawing a red " + decoratedShape.draw();
    }
}
```

codesnap.dev

Рисунок 16 класс RedShapeDecorator, расширяющий базовый класс ShapeDecorator

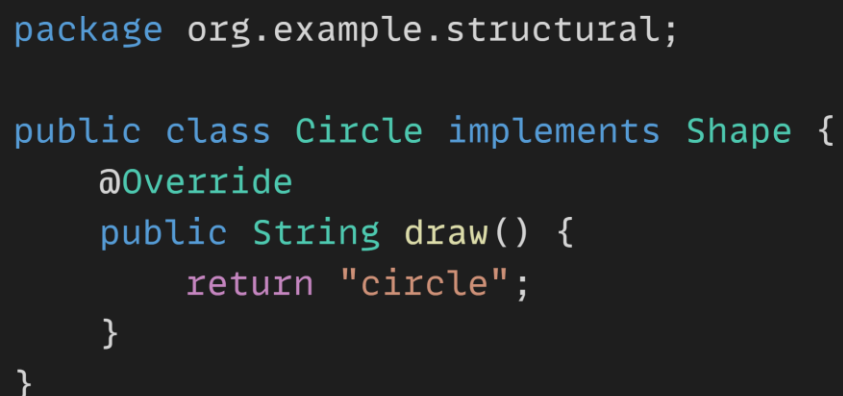


```
package org.example.structural;

public interface Shape {
    String draw();
}
```

codesnap.dev

Рисунок 17 базовый класс Shape



```
package org.example.structural;

public class Circle implements Shape {
    @Override
    public String draw() {
        return "circle";
    }
}
```

Рисунок 18 класс Circle

```
package org.example.structural;

public class Square implements Shape {
    @Override
    public String draw() {
        return "square";
    }
}
```

codesnap.dev

Рисунок 19 класс Square

```
package org.example.structural;

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        System.out.println("Drawing a circle: " + circle.draw());

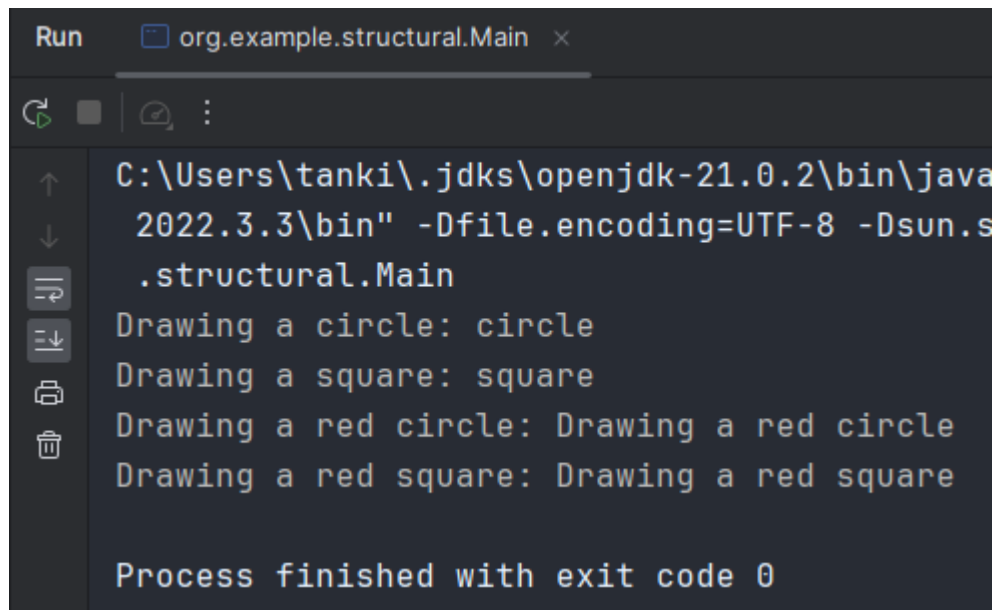
        Shape square = new Square();
        System.out.println("Drawing a square: " + square.draw());

        Shape redCircle = new RedShapeDecorator(new Circle());
        System.out.println("Drawing a red circle: " + redCircle.draw());

        Shape redSquare = new RedShapeDecorator(new Square());
        System.out.println("Drawing a red square: " + redSquare.draw());
    }
}
```

codesnap.dev

Рисунок 20 класс Main, иллюстрирующий использование шаблона



```
Run  org.example.structural.Main x
C:\Users\tanki\.jdk\openjdk-21.0.2\bin\java
  2022.3.3\bin" -Dfile.encoding=UTF-8 -Dsun.s
    .structural.Main
Drawing a circle: circle
Drawing a square: square
Drawing a red circle: Drawing a red circle
Drawing a red square: Drawing a red square

Process finished with exit code 0
```

Рисунок 21 результат работы программы

Вывод

В ходе практической работы познакомился с тремя различными шаблонами проектирования – Observer, Factory, Decorator. Изучил, для чего нужен каждый из них и когда следует использовать их.