Стратегия

// Определение интерфейса стратегии

```
public interface ShippingStrategy {
   double calculateShippingCost(double weight);
}
```

// Конкретные стратегии

```
// Стратегия для доставки почтой
public class PostalServiceShipping implements ShippingStrategy {
   @Override
   public double calculateShippingCost(double weight) {
      return weight * 1.2; // Условный коэффициент стоимости за вес
   }
}

// Стратегия для курьерской доставки
public class CourierServiceShipping implements ShippingStrategy {
   @Override
   public double calculateShippingCost(double weight) {
      return weight * 2.5; // Условный коэффициент стоимости за вес
   }
}
// Стратегия для экспресс-доставки
public class ExpressShipping implements ShippingStrategy {
   @Override
   public double calculateShippingCost(double weight) {
      return weight * 5.0; // Условный коэффициент стоимости за вес
   }
}
```

//Контекст, использующий стратегию

```
// Класс контекста, который использует различные стратегии
public class ShippingContext {
    private ShippingStrategy strategy;
    // Метод для установки стратегии
    public void setStrategy(ShippingStrategy strategy) {
        this.strategy = strategy;
    }
    // Метод для расчета стоимости доставки
    public double calculateCost(double weight) {
        return strategy.calculateShippingCost(weight);
    }
}
```

// Пример использования

```
public class StrategyExample {
    public static void main(String[] args) {
        ShippingContext context = new ShippingContext();
        // Рассчитаем стоимость для почтовой доставки
        context.setStrategy(new PostalServiceShipping());
        double postalCost = context.calculateCost(10.0);
        System.out.println("Postal Service Cost: " + postalCost);
        // Рассчитаем стоимость для курьерской доставки
        context.setStrategy(new CourierServiceShipping());
        double courierCost = context.calculateCost(10.0);
        System.out.println("Courier Service Cost: " + courierCost);
        // Рассчитаем стоимость для экспресс-доставки
        context.setStrategy(new ExpressShipping());
        double expressCost = context.calculateCost(10.0);
        System.out.println("Express Shipping Cost: " + expressCost);
    }
}
```

Шаблонный метод

// Абстрактный класс с шаблонным методом

```
public abstract class Beverage {
    // Шаблонный метод, определяющий шаги приготовления
    public final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    // Шаги, которые реализованы в базовом классе
    private void boilWater() {
        System.out.println("Boiling water");
    }
    private void pourInCup() {
        System.out.println("Pouring into cup");
    }
    // Абстрактные методы, которые будут реализованы в подклассах
    protected abstract void brew();
    protected abstract void addCondiments();
}
```

// Конкретные классы для чая и кофе

```
// Реализация для приготовления чая
public class Tea extends Beverage {
    @Override
    protected void brew() {
        System.out.println("Steeping the tea");
    }
    @Override
```

```
    protected void addCondiments() {
        System.out.println("Adding lemon");
    }
}
// Реализация для приготовления кофе
public class Coffee extends Beverage {
    @Override
    protected void brew() {
        System.out.println("Dripping coffee through filter");
    }
    @Override
    protected void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}
```

//Пример использования

```
public class TemplateMethodExample {
    public static void main(String[] args) {
        // Приготовление чая
        Beverage tea = new Tea();
        System.out.println("Making tea...");
        tea.prepareRecipe();

        System.out.println();

        // Приготовление кофе
        Beverage coffee = new Coffee();
        System.out.println("Making coffee...");
        coffee.prepareRecipe();
    }}
```

Интерфейс команды

```
// Интерфейс команды
public interface Command {
    void execute();
    void undo();
}
// Класс для управления освещением
public class Light {
    private String location;
    public Light(String location) {
        this.location = location;
    }
    public void on() {
        System.out.println(location + " light is ON");
    }
    public void off() {
        System.out.println(location + " light is OFF");
    }
}
// Команды для включения и выключения света
// Команда для включения света
public class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.on();
    }
```

```
    @Override
    public void undo() {
        light.off();
    }
}
// Команда для выключения света
public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.off();
    }
    @Override
    public void undo() {
        light.on();
    }
}
```

// Класс пульта управления (Invoker)

```
// Класс пульта, который управляет командами
public class RemoteControl {
    private Command[] onCommands;
    private Command[] offCommands;
    private Command undoCommand;
    public RemoteControl() {
        onCommands = new Command[2];  // Для простоты 2 слота для команд
        offCommands = new Command[2];
```

```
        Command noCommand = new NoCommand();
        for (int i = 0; i < 2; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }
    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }
    public void onButtonWasPressed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }
    public void offButtonWasPressed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }
    public void undoButtonWasPressed() {
        undoCommand.undo();
    }
}
// Команда "ничего не делать", используется как заглушка
public class NoCommand implements Command {
    @Override
    public void execute() {
    }
    @Override
    public void undo() {
```

```
    }
}
```

//Пример использования

```
public class CommandPatternExample {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        // Создаем объект света
        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        // Создаем команды для света
        LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);
        // Назначаем команды в пульт
        remote.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remote.setCommand(1, kitchenLightOn, kitchenLightOff);
        // Управляем светом
        System.out.println("Testing Living Room Light:");
        remote.onButtonWasPressed(0);
        remote.offButtonWasPressed(0);
        remote.undoButtonWasPressed();
        System.out.println("\nTesting Kitchen Light:");
        remote.onButtonWasPressed(1);
        remote.offButtonWasPressed(1);
        remote.undoButtonWasPressed();
    }
}
```

Фабричный метод

```
// Абстрактный класс пиццы
public abstract class Pizza {
    public abstract void prepare();
    public void bake() {
        System.out.println("Baking the pizza...");
    }
    public void cut() {
        System.out.println("Cutting the pizza...");
    }
    public void box() {
        System.out.println("Boxing the pizza...");
    }
}
//Конкретные классы пицц
public class CheesePizza extends Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing cheese pizza...");
    }
}
public class PepperoniPizza extends Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing pepperoni pizza...");
    }
}
public class VeggiePizza extends Pizza {
    @Override
    public void prepare() {
```

```
        System.out.println("Preparing veggie pizza...");
    }
}
```

//Абстрактная фабрика пицц (фабричный метод)

```
public abstract class PizzaStore {
    // Фабричный метод, который создается в подклассах
    protected abstract Pizza createPizza(String type);
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        if (pizza != null) {
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
        } else {
            System.out.println("Sorry, we don't have that kind of pizza.");
        }
        return pizza;
    }
}
```

// Конкретные фабрики пицц

```
public class NewYorkPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        } else if (type.equals("veggie")) {
            return new VeggiePizza();
```

```
        } else {
            return null;
        }
    }
}
public class ChicagoPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza(); // В реальной ситуации могут быть другие реализации
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza(); // Chicago версия пиццы
        } else if (type.equals("veggie")) {
            return new VeggiePizza();
        } else {
            return null;
        }
    }
}
```

//Пример использования

```
public class FactoryMethodExample {
    public static void main(String[] args) {
        // Создаем фабрику пицц для Нью-Йорка
        PizzaStore newYorkStore = new NewYorkPizzaStore();
        // Заказываем пиццу с сыром в Нью-Йорке
        System.out.println("Order from New York Pizza Store:");
        newYorkStore.orderPizza("cheese");
        System.out.println();
        // Создаем фабрику пицц для Чикаго
        PizzaStore chicagoStore = new ChicagoPizzaStore();
```

```
        // Заказываем пиццу с пепперони в Чикаго
        System.out.println("Order from Chicago Pizza Store:");
        chicagoStore.orderPizza("pepperoni");
    }
}
```

Адаптер

//Интерфейсы для медиа-плееров

```
public interface MediaPlayer {
    void play(String mediaType, String fileName);
}

// Интерфейс для усовершенствованного медиа-плеера (например, для видео)
public interface AdvancedMediaPlayer {
    void playVideo(String fileName);
    void playAudio(String fileName);
}
```

//Конкретные классы плееров

```
// Класс, воспроизводящий только аудиофайлы
public class AudioPlayer implements MediaPlayer {
    @Override
    public void play(String mediaType, String fileName) {
        if (mediaType.equalsIgnoreCase("audio")) {
            System.out.println("Playing audio file. Name: " + fileName);
        } else {
            System.out.println("Invalid media type. Audio player supports only audio files.");
        }
    }
}
```

```
// Класс, воспроизводящий видео и аудио (более продвинутый плеер)
public class VideoPlayer implements AdvancedMediaPlayer {
   @Override
   public void playVideo(String fileName) {
      System.out.println("Playing video file. Name: " + fileName);
   }
   @Override
   public void playAudio(String fileName) {
      System.out.println("Playing audio file. Name: " + fileName);
   }
}
```

//Адаптер для воспроизведения видео через интерфейс `MediaPlayer`

```
public class MediaAdapter implements MediaPlayer {
   private AdvancedMediaPlayer advancedMediaPlayer;

   public MediaAdapter(String mediaType) {
      if (mediaType.equalsIgnoreCase("video")) {
         advancedMediaPlayer = new VideoPlayer();
      }
   }

   @Override
   public void play(String mediaType, String fileName) {
      if (mediaType.equalsIgnoreCase("video")) {
         advancedMediaPlayer.playVideo(fileName);
      } else {
         System.out.println("Unsupported media type for adapter.");
      }
   }
}
```

//Модифицированный аудиоплеер с поддержкой видео через адаптер

```
// Аудио плеер, который использует адаптер для воспроизведения видео
public class AudioPlayerWithAdapter implements MediaPlayer {
   private MediaAdapter mediaAdapter;
   @Override
   public void play(String mediaType, String fileName) {
      if (mediaType.equalsIgnoreCase("audio")) {
         System.out.println("Playing audio file. Name: " + fileName);
      } else if (mediaType.equalsIgnoreCase("video")) {
         mediaAdapter = new MediaAdapter(mediaType);
         mediaAdapter.play(mediaType, fileName);
      } else {
         System.out.println("Invalid media type. Audio player supports only audio and video
files.");
      }
   }
}
```

//Пример использования

```
public class AdapterPatternExample {
   public static void main(String[] args) {
      MediaPlayer player = new AudioPlayerWithAdapter();

      // Воспроизводим аудио
      player.play("audio", "song.mp3");

      // Воспроизводим видео через адаптер
      player.play("video", "movie.mp4");
      // Попытка воспроизвести неподдерживаемый тип
      player.play("text", "document.txt");
   }
}
```

Декоратор

//Интерфейс компонента (Базовый кофе)

```
// Интерфейс для кофе
public interface Coffee {
   String getDescription();
   double getCost();
}
```

//Конкретные реализации кофе

```
// Обычный черный кофе
public class SimpleCoffee implements Coffee {
   @Override
   public String getDescription() {
      return "Simple coffee";
   }
   @Override
   public double getCost() {
      return 5.0; // Цена за простой кофе
   }
}
```

//Абстрактный декоратор

```
// Абстрактный класс декоратора, который реализует интерфейс Coffee
public abstract class CoffeeDecorator implements Coffee {
   protected Coffee decoratedCoffee;
   public CoffeeDecorator(Coffee coffee) {
      this.decoratedCoffee = coffee;
   }
   @Override
   public String getDescription() {
      return decoratedCoffee.getDescription();
```

```
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }}
```

//Конкретные декораторы

```
// Декоратор для добавления молока
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with milk";
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 1.5; // Стоимость добавки молока
    }
}
// Декоратор для добавления шоколада
public class ChocolateDecorator extends CoffeeDecorator {

    public ChocolateDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with chocolate";
    }
```

```
    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 2.0; // Стоимость добавки шоколада
    }
}
```

//Пример использования

```
public class DecoratorPatternExample {
    public static void main(String[] args) {
        // Заказ обычного кофе
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " costs $" + coffee.getCost());
        // Заказ кофе с молоком
        Coffee milkCoffee = new MilkDecorator(coffee);
        System.out.println(milkCoffee.getDescription() + " costs $" + milkCoffee.getCost());
        // Заказ кофе с молоком и шоколадом
        Coffee chocolateMilkCoffee = new ChocolateDecorator(milkCoffee);
        System.out.println(chocolateMilkCoffee.getDescription() + " costs $" +
chocolateMilkCoffee.getCost());
    }
}
```

Наблюдатель

//Интерфейс наблюдателя

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

// Интерфейс субъекта

```
public interface Subject {
   void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();}
```

// Конкретная реализация субъекта (Новостной канал)

```
import java.util.ArrayList;
import java.util.List;
public class NewsChannel implements Subject {
   private List<Observer> observers; // Список наблюдателей
   private String latestNews;
   public NewsChannel() {
      observers = new ArrayList<>();
   }
   // Регистрация наблюдателя
   @Override
   public void registerObserver(Observer observer) {
      observers.add(observer);
   }
   // Удаление наблюдателя
   @Override
   public void removeObserver(Observer observer) {
      observers.remove(observer);
   }
   // Уведомление всех наблюдателей
   @Override
   public void notifyObservers() {
      for (Observer observer : observers) {
         observer.update(latestNews);
      }
   }
   // Установка новых новостей и уведомление наблюдателей
   public void setNews(String news) {
      this.latestNews = news;
      notifyObservers();    }}
```

// Конкретные наблюдатели (Подписчики)

```
// Наблюдатель - подписчик на новости
public class NewsSubscriber implements Observer {
    private String name;
    public NewsSubscriber(String name) {
        this.name = name;
    }
    // Реакция на обновление новостей
    @Override
    public void update(String news) {
        System.out.println(name + " received news update: " + news);
    }
}
```

// Пример использования

```
public class ObserverPatternExample {
    public static void main(String[] args) {
        // Создаем новостной канал (субъект)
        NewsChannel newsChannel = new NewsChannel();
        // Создаем подписчиков (наблюдатели)
        NewsSubscriber subscriber1 = new NewsSubscriber("Subscriber 1");
        NewsSubscriber subscriber2 = new NewsSubscriber("Subscriber 2");
        // Подписчики подписываются на канал
        newsChannel.registerObserver(subscriber1);
        newsChannel.registerObserver(subscriber2);
        // Публикация новости
        newsChannel.setNews("Breaking news: Java is awesome!");
        // Удаляем одного подписчика и публикуем еще одну новость
        newsChannel.removeObserver(subscriber1);
        newsChannel.setNews("More news: Observer pattern in action!");
    }}
```

Итератор

//Интерфейс итератора

```
// Интерфейс для итератора
public interface Iterator {
   boolean hasNext();  // Проверка, есть ли следующий элемент
   Object next();      // Возвращает следующий элемент
}
```

// Интерфейс коллекции (Iterable)

```
// Интерфейс для коллекции, которая может возвращать итератор
public interface IterableCollection {
   Iterator createIterator();
}
```

// Конкретная коллекция (список имен):

// Класс, представляющий коллекцию имен

```
public class NameCollection implements IterableCollection {
   private String[] names = {"John", "Jane", "Michael", "Emma"};
   @Override
   public Iterator createIterator() {
      return new NameIterator();
   }
   // Внутренний класс для итератора
   private class NameIterator implements Iterator {
      int index = 0;
      @Override
      public boolean hasNext() {
         return index < names.length;  // Если индекс меньше длины массива, есть следующий элемент
      }
```

```
    @Override
    public Object next() {
        if (this.hasNext()) {
            return names[index++];  // Возвращаем текущий элемент и увеличиваем индекс
        }
        return null;
    }
  }
}
```

// Пример использования

```
public class IteratorPatternExample {
  public static void main(String[] args) {
    // Создаем коллекцию имен
    NameCollection nameCollection = new NameCollection();
    // Получаем итератор для коллекции
    Iterator iterator = nameCollection.createIterator();
    // Перебираем элементы коллекции
    System.out.println("Iterating through names:");
    while (iterator.hasNext()) {
      String name = (String) iterator.next();
      System.out.println(name);
    }
  }
}
```

Фасад

//Система домашнего кинотеатра

```
// Проектор
public class Projector {
    public void turnOn() {
        System.out.println("Projector is turned on.");
    }
    public void turnOff() {
        System.out.println("Projector is turned off.");
    }
    public void setInput(String input) {
        System.out.println("Projector input set to " + input + ".");
    }
}
// Звуковая система
public class SoundSystem {
    public void turnOn() {
        System.out.println("Sound system is turned on.");
    }
    public void turnOff() {
        System.out.println("Sound system is turned off.");
    }
    public void setVolume(int level) {
        System.out.println("Sound volume set to " + level + ".");
    }
}
// DVD-плеер
public class DVDPlayer {
    public void turnOn() {
        System.out.println("DVD player is turned on.");
```

```
    }


    public void turnOff() {
        System.out.println("DVD player is turned off.");
    }
    public void play(String movie) {
        System.out.println("Playing movie: " + movie);
    }
}
```

// Класс Фасада

```
// Фасад для домашнего кинотеатра
public class HomeTheaterFacade {
    private Projector projector;
    private SoundSystem soundSystem;
    private DVDPlayer dvdPlayer;
    public HomeTheaterFacade(Projector projector, SoundSystem soundSystem, DVDPlayer
dvdPlayer) {
        this.projector = projector;
        this.soundSystem = soundSystem;
        this.dvdPlayer = dvdPlayer;
    }
    // Метод для упрощённого запуска системы
    public void watchMovie(String movie) {
        System.out.println("Setting up the home theater to watch a movie...");
        projector.turnOn();
        projector.setInput("DVD");
        soundSystem.turnOn();
        soundSystem.setVolume(10);
        dvdPlayer.turnOn();
        dvdPlayer.play(movie);
    }
```

```
    // Метод для завершения сеанса
    public void endMovie() {
        System.out.println("Shutting down the home theater...");
        dvdPlayer.turnOff();
        soundSystem.turnOff();
        projector.turnOff();
    }
}
```

//Пример использования

```
public class FacadePatternExample {
    public static void main(String[] args) {
        // Создаем компоненты системы
        Projector projector = new Projector();
        SoundSystem soundSystem = new SoundSystem();
        DVDPlayer dvdPlayer = new DVDPlayer();

        // Создаем фасад
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(projector, soundSystem,
dvdPlayer);

        // Используем фасад для управления системой
        homeTheater.watchMovie("Inception");
        System.out.println();
        homeTheater.endMovie();
    }
}
```

Синглтон

//Класс Синглтон

```
public class LoggerSingleton {
   // Статическая переменная для хранения единственного экземпляра
   private static LoggerSingleton instance;
   // Приватный конструктор, чтобы предотвратить создание экземпляра вне класса
   private LoggerSingleton() {
      // Приватный конструктор препятствует созданию объектов извне
   }
   // Публичный статический метод для получения единственного экземпляра
   public static LoggerSingleton getInstance() {
      if (instance == null) {
         instance = new LoggerSingleton();  // Создаем экземпляр при первом вызове
      }
      return instance;
   }
   // Метод для записи логов
   public void log(String message) {
      System.out.println("Log entry: " + message);
   }
}
```

// Пример использования Синглтона

```
public class SingletonPatternExample {
   public static void main(String[] args) {
      // Получаем единственный экземпляр LoggerSingleton
      LoggerSingleton logger1 = LoggerSingleton.getInstance();
      // Используем метод log() для записи логов
      logger1.log("This is the first log message.");


      // Получаем второй раз тот же экземпляр LoggerSingleton
```

```
        LoggerSingleton logger2 = LoggerSingleton.getInstance();
        logger2.log("This is the second log message.");
        // Проверяем, что оба объекта — один и тот же экземпляр
        if (logger1 == logger2) {
            System.out.println("Logger1 and Logger2 are the same instance.");
        } else {
            System.out.println("Logger1 and Logger2 are different instances.");
        }
    }
}
```