**Лабораторная работа № 4.**

**Шаблоны в языке Java.**

**Паттерны проектирования** - это готовые к использованию решения часто возникающих в программировании задач. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее. Паттерны проектирования, подходящий под задачу, реализуется в каждом конкретном случае. Следует, помнить, что такой паттерн, будучи примененным неправильно или к неподходящей задаче, может принести немало проблем. Тем не менее, правильно примененный паттерн поможет решить задачу легко и просто.

**Типы паттернов**:

* *порождающие*
* *структурные*
* *поведенческие*

Порождающие паттерны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

Структурные паттерны определяют отношения между классами и объектами, позволяя им работать совместно.

Поведенческие паттерны используются для того, чтобы упростить взаимодействие между сущностями.

*Порождающие*:

* **Singleton** (Одиночка) – ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту.
* **Factory** (Фабрика) – используется, когда у нас есть суперкласс с несколькими подклассами и на основе ввода, нам нужно вернуть один из подкласса.
* **Abstract Factory** (Абстрактная фабрика) – используем супер фабрику для создания фабрики, затем используем созданную фабрику для создания объектов.
* **Builder** (Строитель) – используется для создания сложного объекта с использованием простых объектов. Постепенно он создает больший объект от малого и простого объекта.
* **Prototype** (Прототип) – помогает создать дублированный объект с лучшей производительностью, вместо нового создается возвращаемый клон существующего объекта.

*Структурные*:

* **Adapter** (Адаптер) ­– это конвертер между двумя несовместимыми объектами. Используя паттерн адаптера, мы можем объединить два несовместимых интерфейса.
* **Composite** (Компоновщик) – использует один класс для представления древовидной структуры.
* **Proxy** (Заместитель) – представляет функциональность другого класса.
* **Flyweight** (Легковес) – вместо создания большого количества похожих объектов, объекты используются повторно.
* **Facade** (Фасад) – беспечивает простой интерфейс для клиента, и клиент использует интерфейс для взаимодействия с системой.
* **Bridge** (Мост) – делает конкретные классы независимыми от классов реализации интерфейса.
* **Decorator** (Декоратор) – добавляет новые функциональные возможности существующего объекта без привязки его структуры.

*Поведенческие*:

* **Template Method** (Шаблонный метод) – определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
* **Mediator** (Посредник) – предоставляет класс посредника, который обрабатывает все коммуникации между различными классами.
* **Chain of Responsibility** (Цепочка обязанностей) – позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.
* **Observer** (Наблюдатель) – позволяет одним обьектам следить и реагировать на события, происходящие в других объектах.
* **Strategy** (Стратегия) – алгоритм стратегии может быть изменен во время выполнения программы.
* **Command** (Команда) – интерфейс команды объявляет метод для выполнения определенного действия.
* **State** (Состояние) – объект может изменять свое поведение в зависимости от его состояния.
* **Visitor** (Посетитель) – используется для упрощения операций над группировками связанных объектов.
* **Interpreter** (Интерпретатор) – определяет грамматику простого языка для проблемной области.
* **Iterator** (Итератор) – последовательно осуществляет доступ к элементам объекта коллекции, не зная его основного представления.
* **Memento** (Хранитель) – используется для хранения состояния объекта, позже это состояние можно восстановить.

**Отношения между классами**

– агрегация (aggregation) — описывает связь «часть»–«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».

 – композиция (composition) — подвид агрегации, в которой «части» не могут существовать отдельно от «целого».

– зависимость (dependency) — изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.

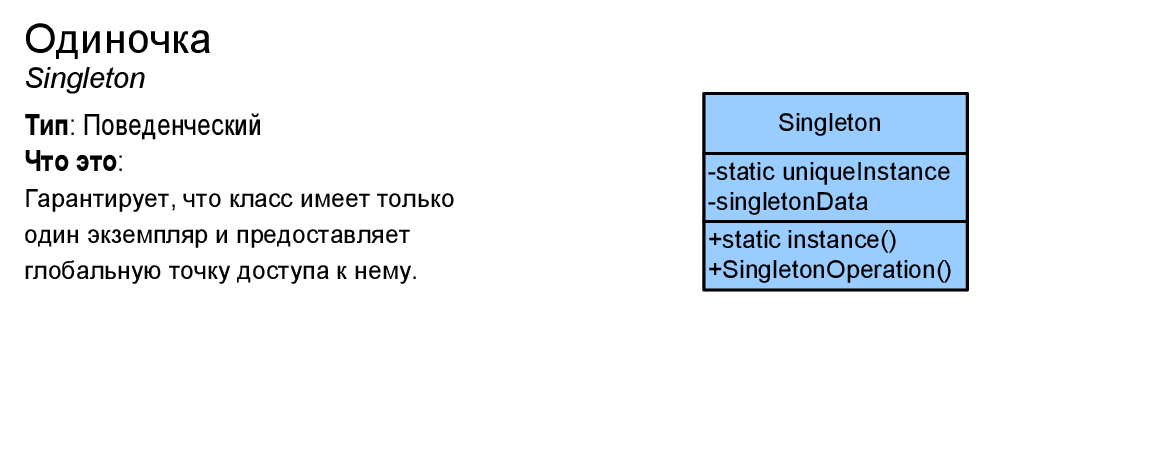
 – обобщение (generalization) — отношение наследования или реализации интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

**Порождающие**

**Singleton (Одиночка)**

*Описание:*

Ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту. Конструктор класса приватный. Метод getInstance() создает только один экземпляр класса.



**Реализация:**

class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

public void setUp() {

System.out.println("setUp");

}

}

public class SingletonTest {//тест

public static void main(String[] args){

Singleton singelton = Singleton.getInstance();

singelton.setUp();

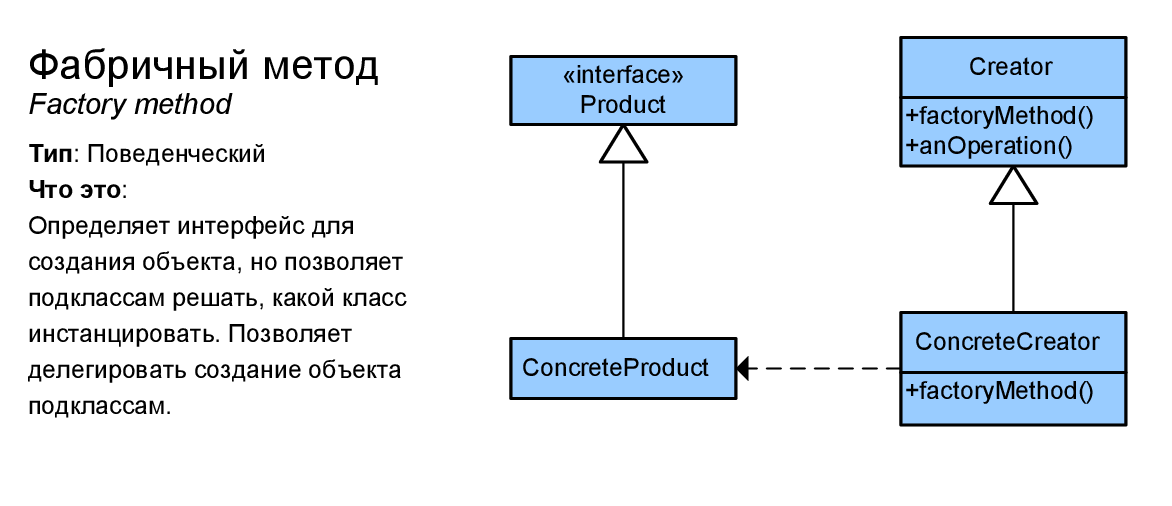
}

}

**Factory (Фабрика)**

*Описание:*

Используется, когда у нас есть суперкласс с несколькими подклассами и на основе ввода, нам нужно вернуть один из подкласса. Класс не знает какого типа объект он должен создать. Объекты создаются в зависимости от входящих данных.



**Реализация:**

class Factory {

public OS getCurrentOS(String inputos) {

OS os = null;

if (inputos.equals("windows")) {

os = new windowsOS();

} else if (inputos.equals("linux")) {

os = new linuxOS();

} else if (inputos.equals("mac")) {

os = new macOS();

}

return os;

}

}

interface OS {

void getOS();

}

class windowsOS implements OS {

public void getOS () {

System.out.println("применить для виндовс");

}

}

class linuxOS implements OS {

public void getOS () {

System.out.println("применить для линукс");

}

}

class macOS implements OS {

public void getOS () {

System.out.println("применить для мак");

}

}

public class FactoryTest {//тест

public static void main(String[] args){

String win = "linux";

Factory factory = new Factory();

OS os = factory.getCurrentOS(win);

os.getOS();

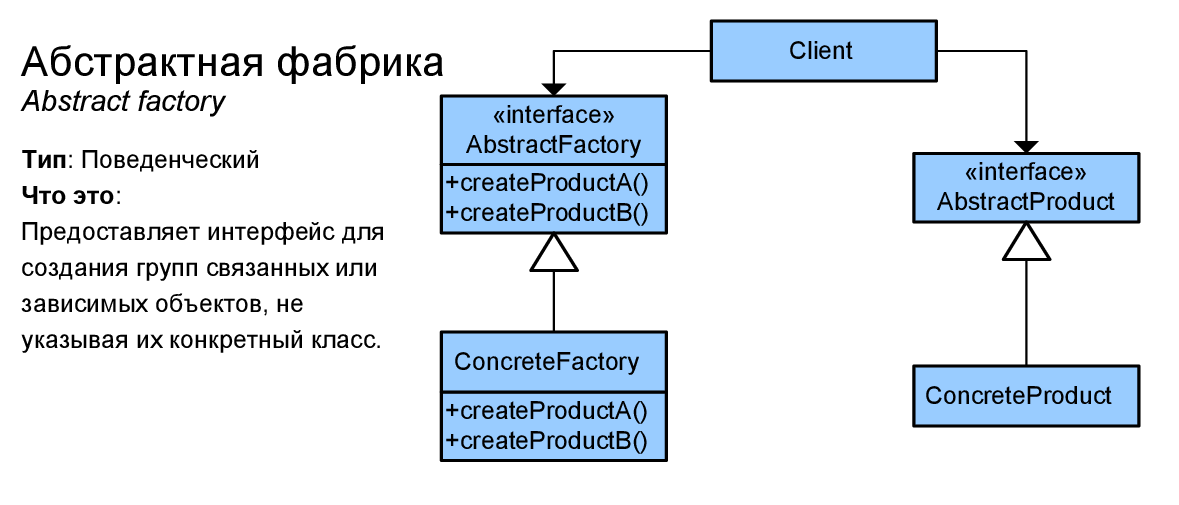
}

}

**Abstract Factory (Абстрактная фабрика)**

*Описание:*

Позволяет выбрать конкретную реализацию фабрики из семейства возможных фабрик. Создает семейство связанных объектов. Легко расширять.



**Реализация:**

interface Lada {

long getLadaPrice();

}

interface Ferrari {

long getFerrariPrice();

}

interface Porshe {

long getPorshePrice();

}

interface InteAbsFactory {

Lada getLada();

Ferrari getFerrari();

Porshe getPorshe();

}

class UaLadaImpl implements Lada {// первая

public long getLadaPrice() {

return 1000;

}

}

class UaFerrariImpl implements Ferrari {

public long getFerrariPrice() {

return 3000;

}

}

class UaPorsheImpl implements Porshe {

public long getPorshePrice() {

return 2000;

}

}

class UaCarPriceAbsFactory implements InteAbsFactory {

public Lada getLada() {

return new UaLadaImpl();

}

public Ferrari getFerrari() {

return new UaFerrariImpl();

}

public Porshe getPorshe() {

return new UaPorsheImpl();

}

}// первая

class RuLadaImpl implements Lada {// вторая

public long getLadaPrice() {

return 10000;

}

}

class RuFerrariImpl implements Ferrari {

public long getFerrariPrice() {

return 30000;

}

}

class RuPorsheImpl implements Porshe {

public long getPorshePrice() {

return 20000;

}

}

class RuCarPriceAbsFactory implements InteAbsFactory {

public Lada getLada() {

return new RuLadaImpl();

}

public Ferrari getFerrari() {

return new RuFerrariImpl();

}

public Porshe getPorshe() {

return new RuPorsheImpl();

}

}// вторая

public class AbstractFactoryTest {//тест

public static void main(String[] args) {

String country = "UA";

InteAbsFactory ifactory = null;

if(country.equals("UA")) {

ifactory = new UaCarPriceAbsFactory();

} else if(country.equals("RU")) {

ifactory = new RuCarPriceAbsFactory();

}

Lada lada = ifactory.getLada();

System.out.println(lada.getLadaPrice());

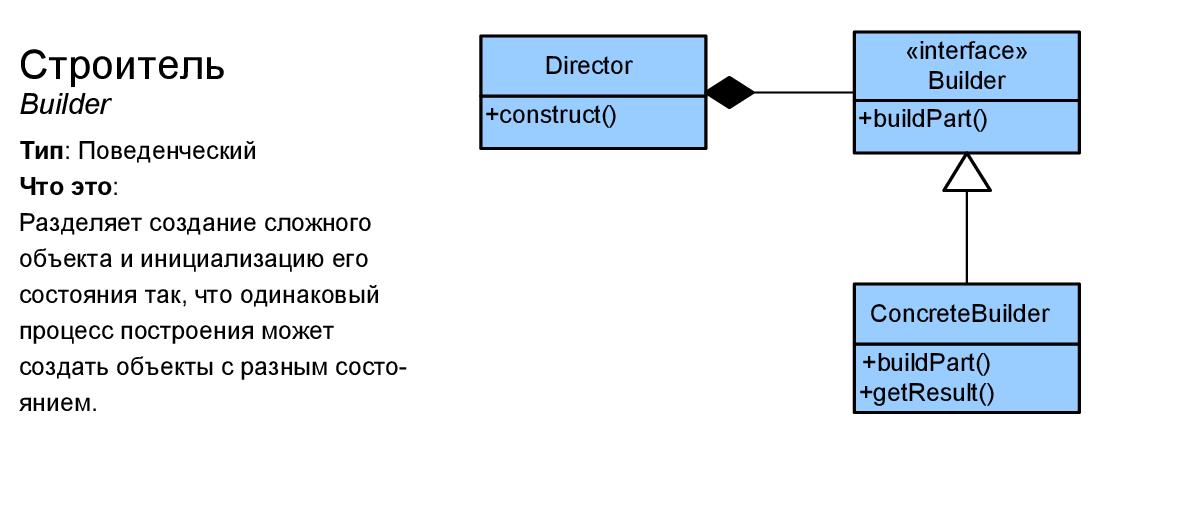
}

}

**Builder (Строитель)**

*Описание:*

Используется для создания сложного объекта с использованием простых объектов. Постепенно он создает больший объект от малого и простого объекта. Позволяет изменять внутреннее представление конечного продукта.



**Реализация:**

class Car {

public void buildBase() {

print("Делаем корпус");

}

public void buildWheels() {

print("Ставим колесо");

}

public void buildEngine(Engine engine) {

print("Ставим движок: " + engine.getEngineType());

}

private void print(String msg){

System.out.println(msg);

}

}

interface Engine {

String getEngineType();

}

class OneEngine implements Engine {

public String getEngineType() {

return "Первый двигатель";

}

}

class TwoEngine implements Engine {

public String getEngineType() {

return "Второй двигатель";

}

}

abstract class Builder {

protected Car car;

public abstract Car buildCar();

}

class OneBuilderImpl extends Builder {

public OneBuilderImpl(){

car = new Car();

}

public Car buildCar() {

car.buildBase();

car.buildWheels();

Engine engine = new OneEngine();

car.buildEngine(engine);

return car;

}

}

class TwoBuilderImpl extends Builder {

public TwoBuilderImpl(){

car = new Car();

}

public Car buildCar() {

car.buildBase();

car.buildWheels();

Engine engine = new OneEngine();

car.buildEngine(engine);

car.buildWheels();

engine = new TwoEngine();

car.buildEngine(engine);

return car;

}

}

class Build {

private Builder builder;

public Build(int i){

if(i == 1) {

builder = new OneBuilderImpl();

} else if(i == 2) {

builder = new TwoBuilderImpl();

}

}

public Car buildCar(){

return builder.buildCar();

}

}

public class BuilderTest {//тест

public static void main(String[] args) {

Build build = new Build(1);

build.buildCar();

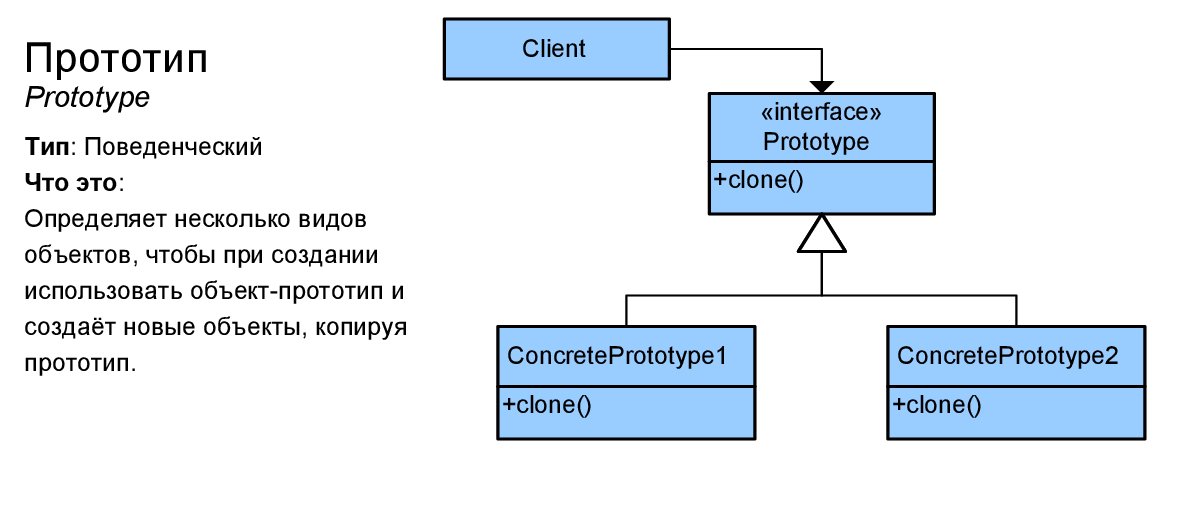
}

}

**Prototype (Прототип)**

*Описание:*

Помогает создать дублированный объект с лучшей производительностью, вместо нового создается возвращаемый клон существующего объекта. Клонирует существующий объект.



**Реализация:**

interface Copyable {

Copyable copy();

}

class ComplicatedObject implements Copyable {

private Type type;

public enum Type {

ONE, TWO

}

public ComplicatedObject copy() {

ComplicatedObject complicatedobject = new ComplicatedObject();

return complicatedobject;

}

public void setType(Type type) {

this.type = type;

}

}

public class PrototypeTest {//тест

public static void main(String[] args) {

ComplicatedObject prototype = new ComplicatedObject();

ComplicatedObject clone = prototype.copy();

clone.setType(ComplicatedObject.Type.ONE);

}

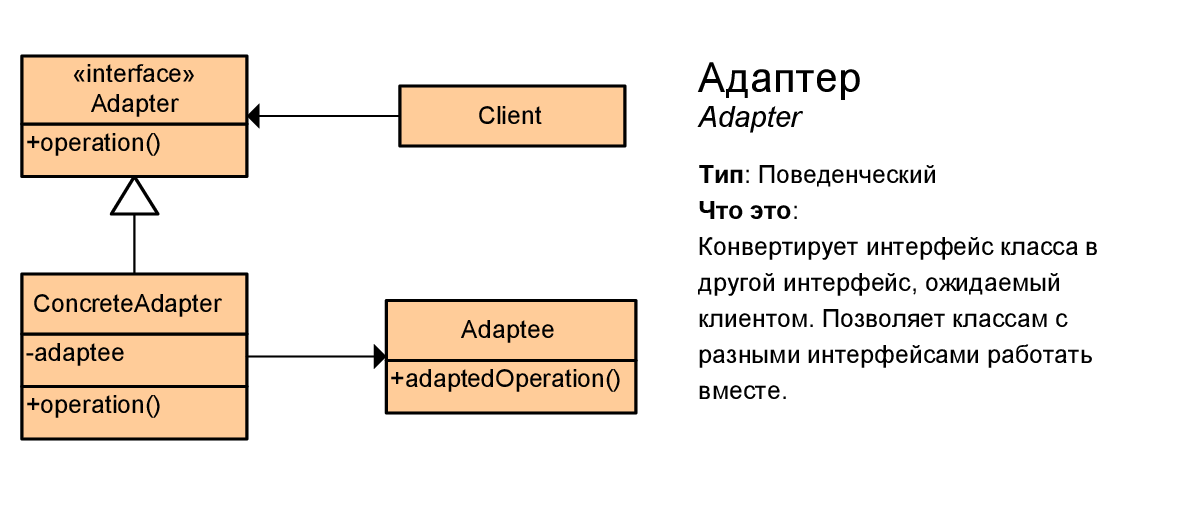
}

**Структурные**

**Adapter (Адаптер)**

*Описание:*

Используя паттерн, мы можем объединить два несовместимых объекта. Конвертер между двумя несовместимыми объектами.



**Реализация:**

class PBank {

private int balance;

public PBank() { balance = 100; }

public void getBalance() {

System.out.println("PBank balance = " + balance);

}

}

class ABank {

private int balance;

public ABank() { balance = 200; }

public void getBalance() {

System.out.println("ABank balance = " + balance);

}

}

class PBankAdapter extends PBank {

private ABank abank;

public PBankAdapter(ABank abank) {

this.abank = abank;

}

public void getBalance() {

abank.getBalance();

}

}

public class AdapterTest {//тест

public static void main(String[] args) {

PBank pbank = new PBank();

pbank.getBalance();

PBankAdapter abank = new PBankAdapter(new ABank());

abank.getBalance();

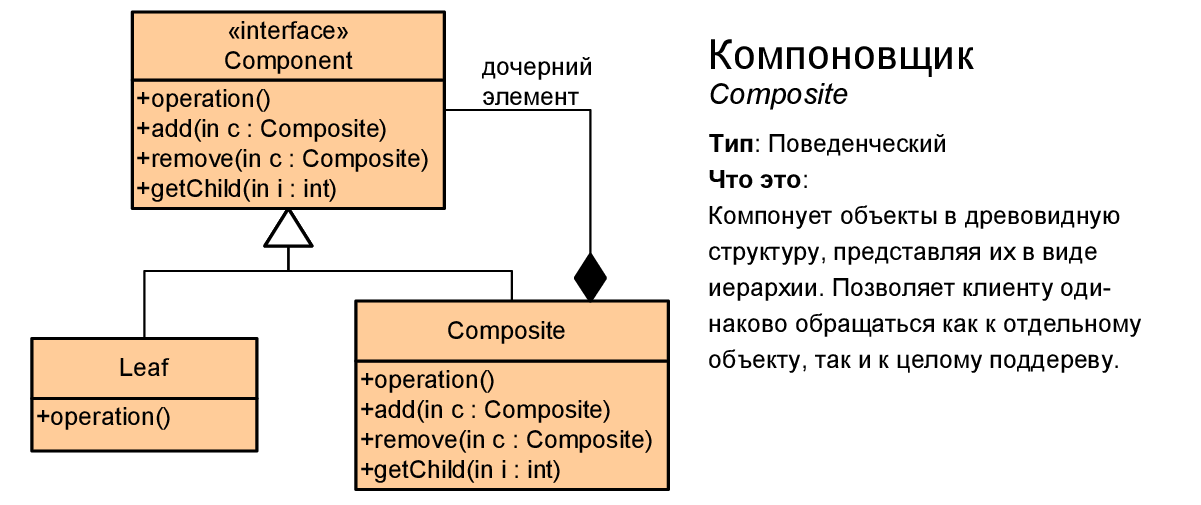
}

}

**Composite (Компоновщик)**

*Описание:*

Группирует несколько объектов в древовидную структуру используя один класс. Позволяет работать с несколькими классами через один объект.



**Реализация:**

import java.util.ArrayList;

import java.util.List;

interface Car {

void draw(String color);

}

class SportCar implements Car {

public void draw(String color) {

System.out.println("SportCar color: " + color);

}

}

class UnknownCar implements Car {

public void draw(String color) {

System.out.println("UnknownCar color: " + color);

}

}

class Drawing implements Car {

private List<Car> cars = new ArrayList<Car>();

public void draw(String color) {

for(Car car : cars) {

car.draw(color);

}

}

public void add(Car s){

this.cars.add(s);

}

public void clear(){

System.out.println();

this.cars.clear();

}

}

public class CompositeTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar();

Car unknownCar = new UnknownCar();

Drawing drawing = new Drawing();

drawing.add(sportCar);

drawing.add(unknownCar);

drawing.draw("green");

drawing.clear();

drawing.add(sportCar);

drawing.add(unknownCar);

drawing.draw("white");

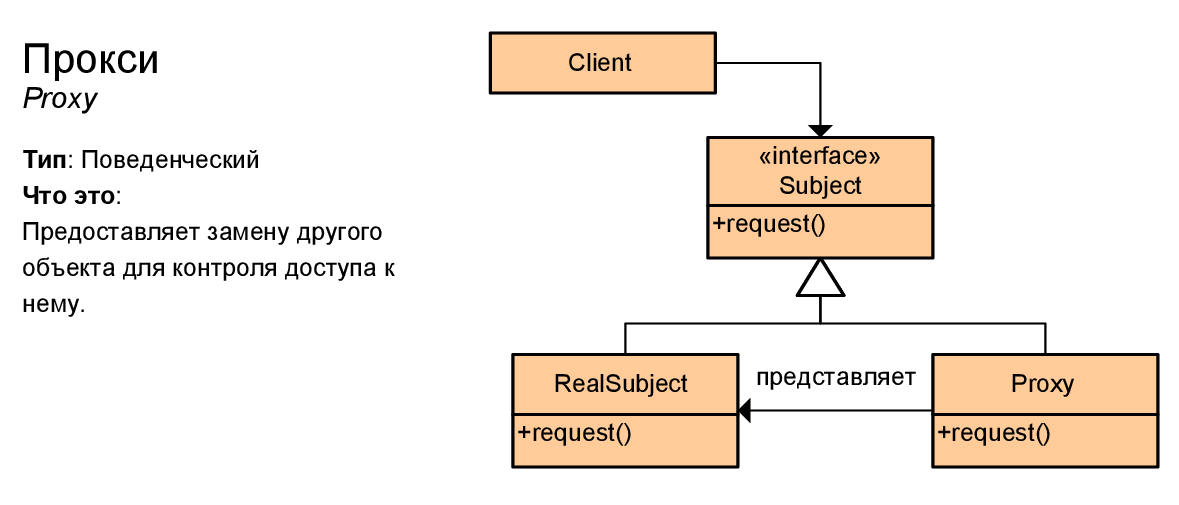
}

}

**Proxy (Заместитель)**

*Описание:*

Представляет объекты, которые могут контролировать другие объекты перехватывая их вызовы. Можно перехватить вызов оригинального объекта.



**Реализация:**

interface Image {

void display();

}

class RealImage implements Image {

private String file;

public RealImage(String file){

this.file = file;

load(file);

}

private void load(String file){

System.out.println("Загрузка " + file);

}

public void display() {

System.out.println("Просмотр " + file);

}

}

class ProxyImage implements Image {

private String file;

private RealImage image;

public ProxyImage(String file){

this.file = file;

}

public void display() {

if(image == null){

image = new RealImage(file);

}

image.display();

}

}

public class ProxyTest {//тест

public static void main(String[] args) {

Image image = new ProxyImage("test.jpg");

image.display();

image.display();

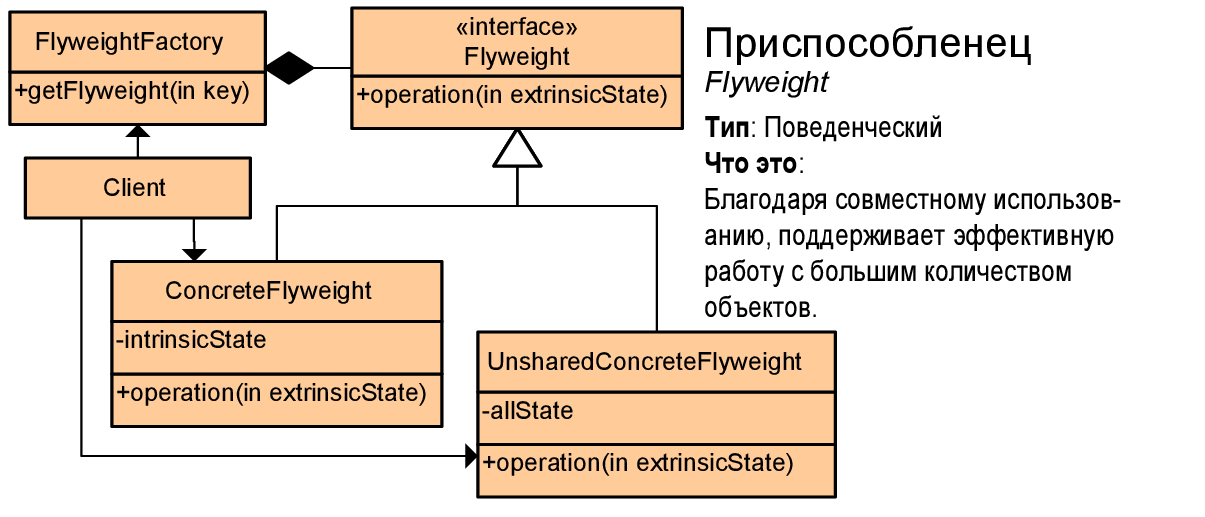
}

}

**Flyweight (Легковес)**

*Описание:*

Вместо создания большого количества похожих объектов, объекты используются повторно. Экономит память.



**Реализация:**

class Flyweight {

private int row;

public Flyweight(int row) {

this.row = row;

System.out.println("ctor: " + this.row);

}

void report(int col) {

System.out.print(" " + row + col);

}

}

class Factory {

private Flyweight[] pool;

public Factory(int maxRows) {

pool = new Flyweight[maxRows];

}

public Flyweight getFlyweight(int row) {

if (pool[row] == null) {

pool[row] = new Flyweight(row);

}

return pool[row];

}

}

public class FlyweightTest {//тест

public static void main(String[] args) {

int rows = 5;

Factory theFactory = new Factory(rows);

for (int i = 0; i < rows; i++) {

for (int j = 0; j < rows; j++) {

theFactory.getFlyweight(i).report(j);

}

System.out.println();

}

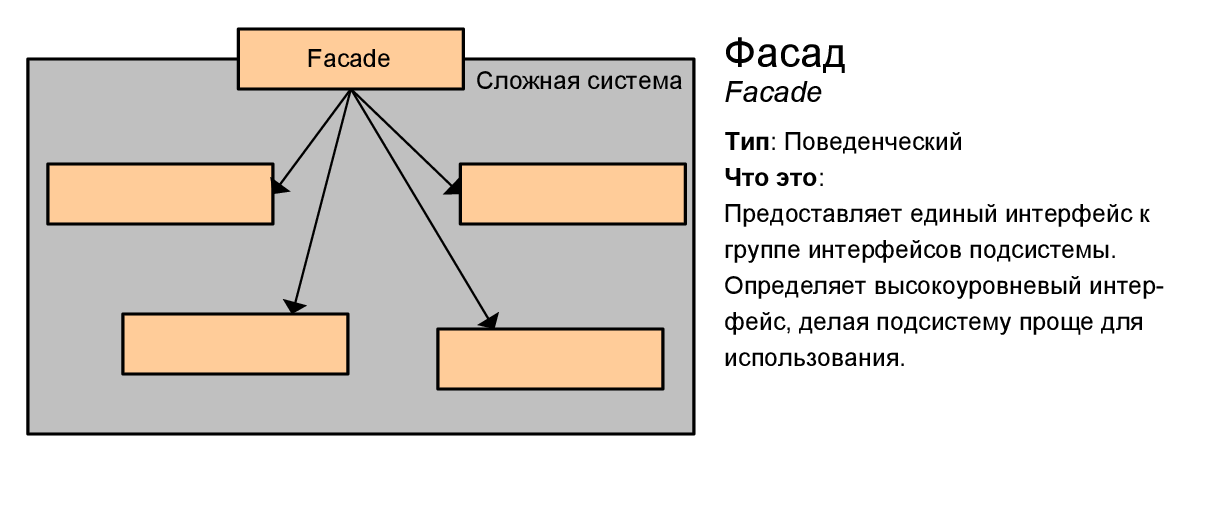
}

}

**Facade (Фасад)**

*Описание:*

Скрывает сложную систему классов приводя все вызовы к одному объекту. Помещает вызов нескольких сложных объектов в один объект.



**Реализация:**

interface Car {

void start();

void stop();

}

class Key implements Car {

public void start() {

System.out.println("Вставить ключи");

}

public void stop() {

System.out.println("Вытянуть ключи");

}

}

class Engine implements Car {

public void start() {

System.out.println("Запустить двигатель");

}

public void stop() {

System.out.println("Остановить двигатель");

}

}

class Facade {

private Key key;

private Engine engine;

public Facade() {

key = new Key();

engine = new Engine();

}

public void startCar() {

key.start();

engine.start();

}

public void stoptCar() {

key.stop();

engine.stop();

}

}

public class FacadeTest {//тест

public static void main(String[] args) {

Facade facade = new Facade();

facade.startCar();

System.out.println();

facade.stoptCar();

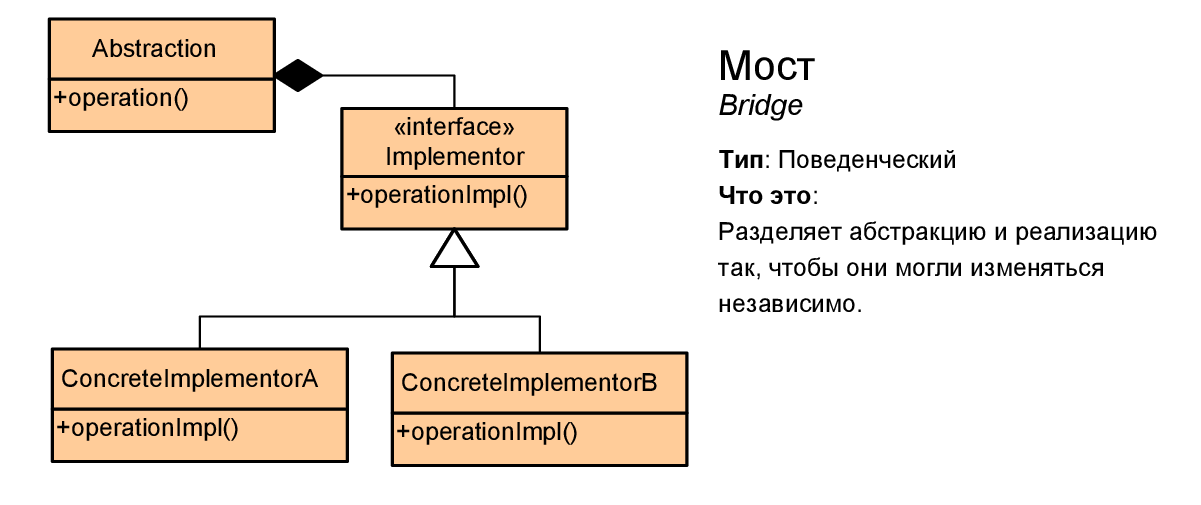
}

}

**Bridge (Мост)**

*Описание:*

Разделяет реализацию и абстракцию, дает возможность изменять их свободно друг от друга. Делает конкретные классы независимыми от классов реализации интерфейса.



**Реализация:**

interface Engine {

void setEngine();

}

abstract class Car {

protected Engine engine;

public Car(Engine engine){

this.engine = engine;

}

abstract public void setEngine();

}

class SportCar extends Car {

public SportCar(Engine engine) {

super(engine);

}

public void setEngine() {

System.out.print("SportCar engine: ");

engine.setEngine();

}

}

class UnknownCar extends Car {

public UnknownCar(Engine engine) {

super(engine);

}

public void setEngine() {

System.out.print("UnknownCar engine: ");

engine.setEngine();

}

}

class SportEngine implements Engine {

public void setEngine(){

System.out.println("sport");

}

}

class UnknownEngine implements Engine {

public void setEngine(){

System.out.println("unknown");

}

}

public class BridgeTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar(new SportEngine());

sportCar.setEngine();

System.out.println();

Car unknownCar = new UnknownCar(new UnknownEngine());

unknownCar.setEngine();

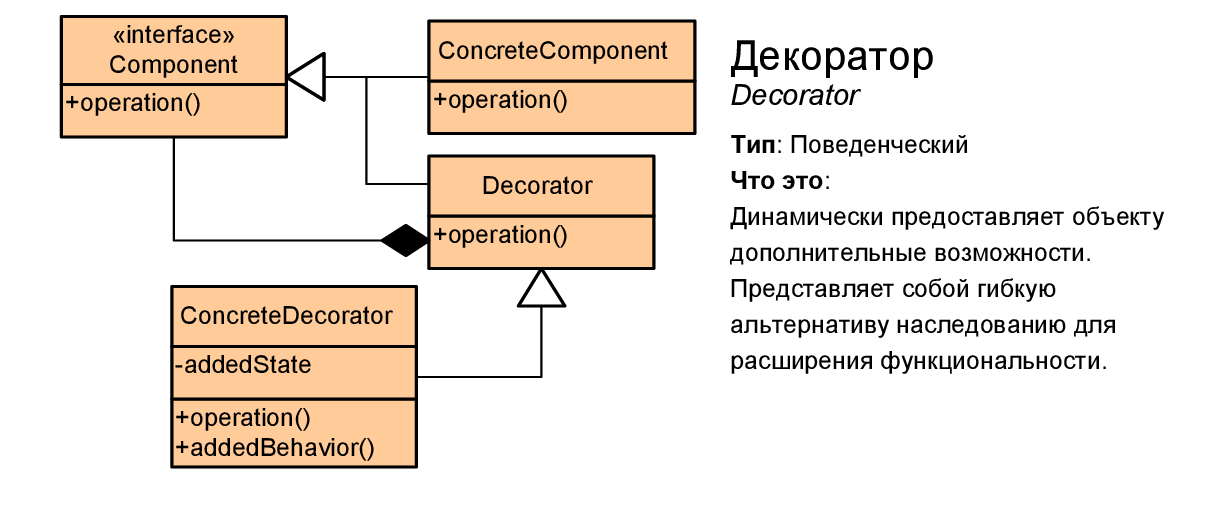
}

}

**Decorator (Декоратор)**

*Описание:*

Добавляет новые функциональные возможности существующего объекта без привязки его структуры.



Реализация:

interface Car {

void draw();

}

class SportCar implements Car {

public void draw() {

System.out.println("SportCar");

}

}

class UnknownCar implements Car {

public void draw() {

System.out.println("UnknownCar");

}

}

abstract class CarDecorator implements Car {

protected Car decorated;

public CarDecorator(Car decorated){

this.decorated = decorated;

}

public void draw(){

decorated.draw();

}

}

class BlueCarDecorator extends CarDecorator {

public BlueCarDecorator(Car decorated) {

super(decorated);

}

public void draw() {

decorated.draw();

setColor();

}

private void setColor(){

System.out.println("Color: red");

}

}

public class DecoratorTest {//тест

public static void main(String[] args) {

Car sportCar = new SportCar();

Car blueUnknownCar = new BlueCarDecorator(new UnknownCar());

sportCar.draw();

System.out.println();

blueUnknownCar.draw();

}

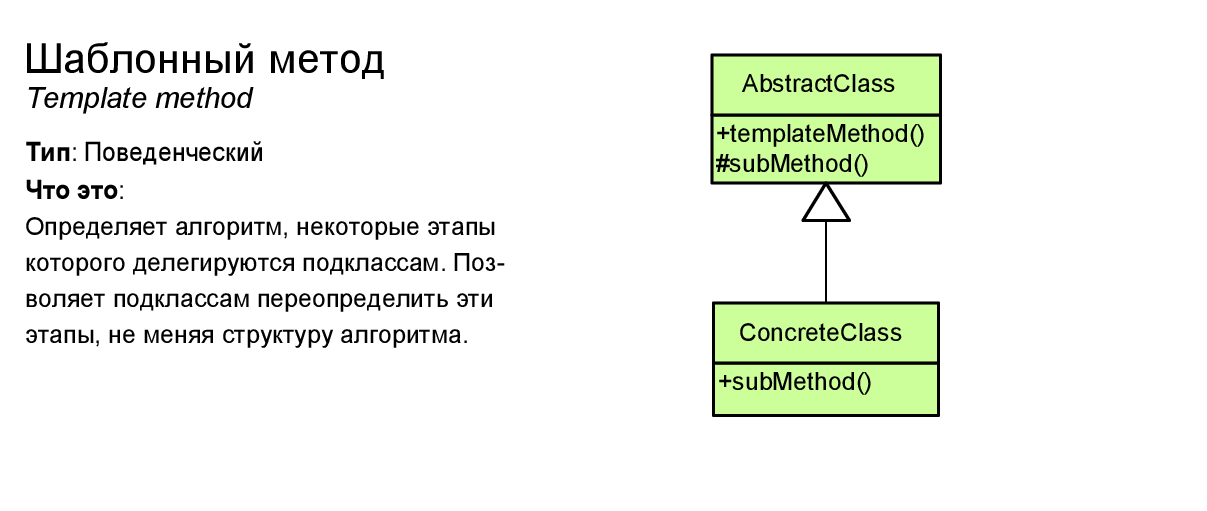
}

**Поведенческие**

**Template Method (Шаблонный метод)**

*Описание:*

Позволяет определить основу алгоритма и позволяющий подклассам переопределять определенные шаги алгоритма, не изменяя его структуру в целом.



**Реализация:**

abstract class Car {

abstract void startEngine();

abstract void stopEngine();

public final void start(){

startEngine();

stopEngine();

}

}

class OneCar extends Car {

public void startEngine() {

System.out.println("Start engine.");

}

public void stopEngine() {

System.out.println("Stop engine.");

}

}

class TwoCar extends Car {

public void startEngine() {

System.out.println("Start engine.");

}

public void stopEngine() {

System.out.println("Stop engine.");

}

}

public class TemplateTest {//тест

public static void main(String[] args) {

Car car1 = new OneCar();

car1.start();

System.out.println();

Car car2 = new TwoCar();

car2.start();

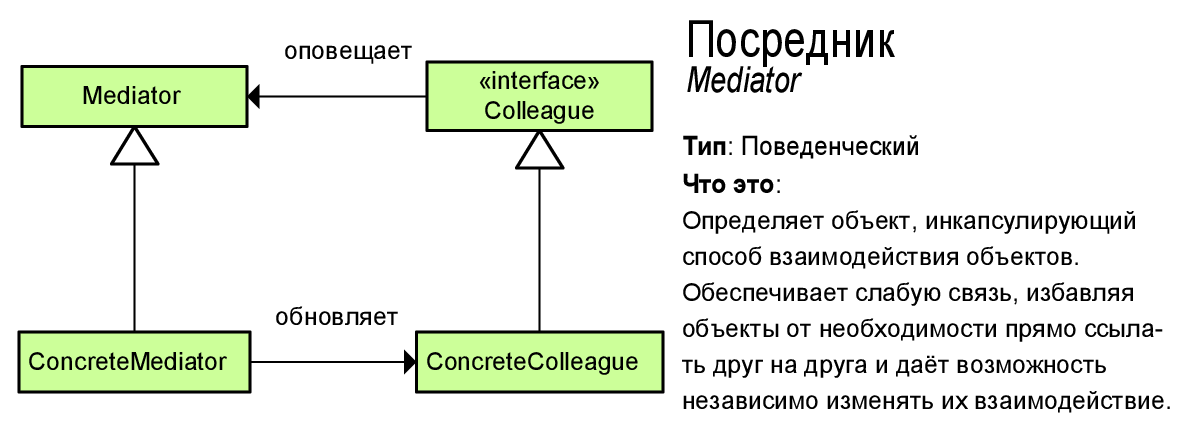
}

}

**Mediator (Посредник)**

*Описание:*

Предоставляет класс посредника, который обрабатывает все связи между различными классами.



**Реализация:**

class Mediator {

public static void sendMessage(User user, String msg){

System.out.println(user.getName() + ": " + msg);

}

}

class User {

private String name;

public User(String name){

this.name = name;

}

public String getName() {

return name;

}

public void sendMessage(String msg){

Mediator.sendMessage(this, msg);

}

}

public class MediatorTest {//тест

public static void main(String[] args) {

User user1 = new User("user1");

User user2 = new User("user2");

user1.sendMessage("message1");

user2.sendMessage("message2");

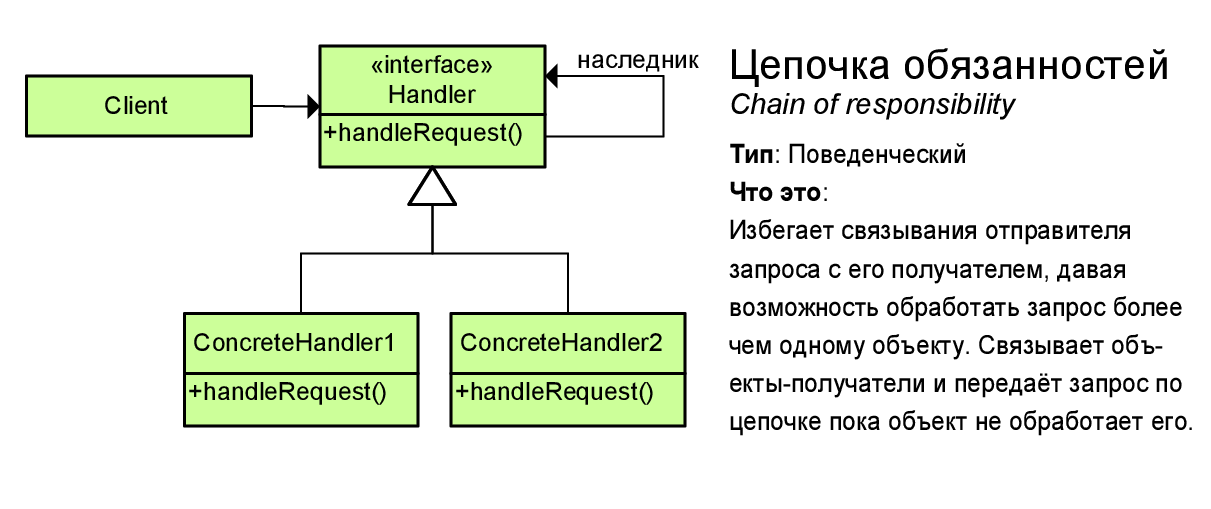
}

}

**Chain of Responsibility (Цепочка обязанностей)**

*Описание:*

Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.



**Реализация:**

interface Payment {

void setNext(Payment payment);

void pay();

}

class VisaPayment implements Payment {

private Payment payment;

public void setNext(Payment payment) {

this.payment = payment;

}

public void pay() {

System.out.println("Visa Payment");

}

}

class PayPalPayment implements Payment {

private Payment payment;

public void setNext(Payment payment) {

this.payment = payment;

}

public void pay() {

System.out.println("PayPal Payment");

}

}

public class ChainofResponsibilityTest {//тест

public static void main(String[] args) {

Payment visaPayment = new VisaPayment();

Payment payPalPayment = new PayPalPayment();

visaPayment.setNext(payPalPayment);

visaPayment.pay();

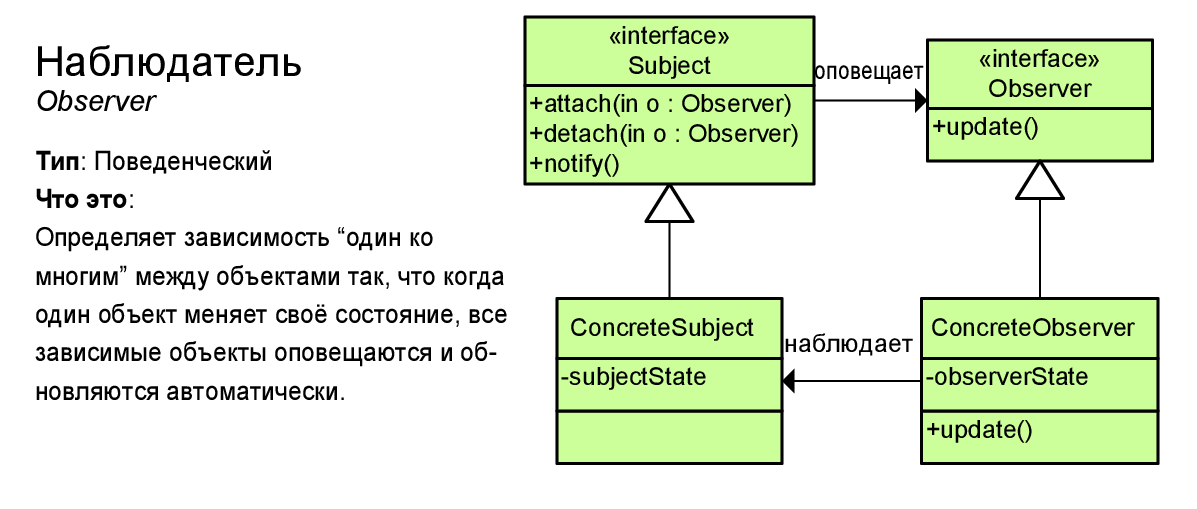
}

}

**Observer (Наблюдатель)**

*Описание:*

Позволяет одним объектам наблюдать за действиями что происходят в других объектах.



**Реализация:**

import java.util.ArrayList;

import java.util.List;

interface Observer {

void event(List<String> strings);

}

class University {

private List<Observer> observers = new ArrayList<Observer>();

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

public void addStudent(String name) {

students.add(name);

notifyObservers();

}

public void removeStudent(String name) {

students.remove(name);

notifyObservers();

}

public void addObserver(Observer observer){

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers(){

for (Observer observer : observers) {

observer.event(students);

}

}

}

class Director implements Observer {

public void event(List<String> strings) {

System.out.println("The list of students has changed: " + strings);

}

}

public class ObserverTest {//тест

public static void main(String[] args) {

University university = new University();

Director director = new Director();

university.addStudent("Vaska");

university.addObserver(director);

university.addStudent("Anna");

university.removeStudent("Vaska");

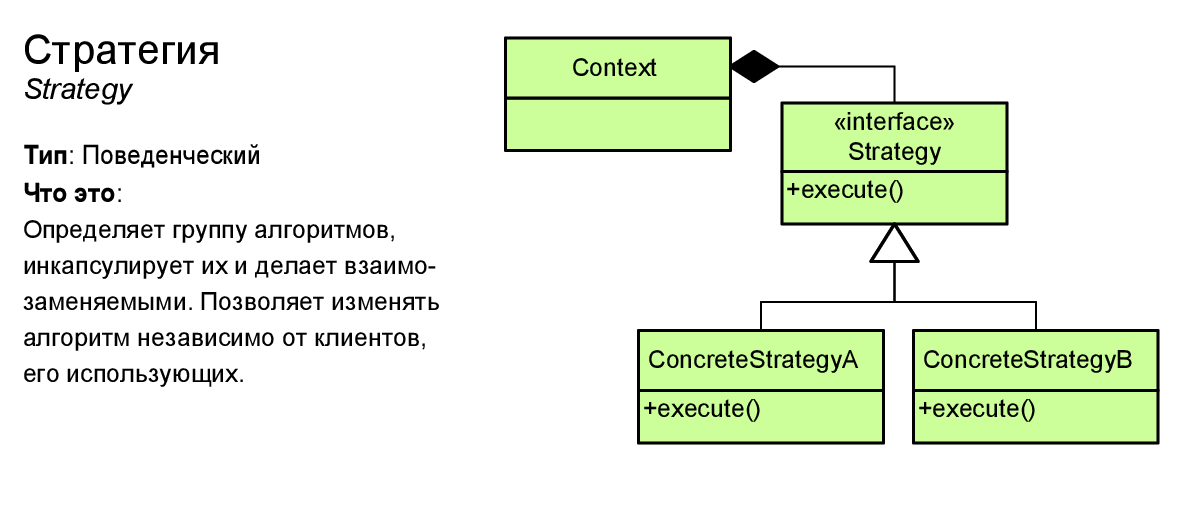
}

}

**Strategy (Стратегия)**

*Описание:*

Определяет ряд алгоритмов позволяя взаимодействовать между ними. Алгоритм стратегии может быть изменен во время выполнения программы.



**Реализация:**

interface Strategy {

void download(String file);

}

class DownloadWindownsStrategy implements Strategy {

public void download(String file) {

System.out.println("windows download: " + file);

}

}

class DownloadLinuxStrategy implements Strategy {

public void download(String file) {

System.out.println("linux download: " + file);

}

}

class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public void download(String file){

strategy.download(file);

}

}

public class StrategyTest {//тест

public static void main(String[] args) {

Context context = new Context(new DownloadWindownsStrategy());

context.download("file.txt");

context = new Context(new DownloadLinuxStrategy());

context.download("file.txt");

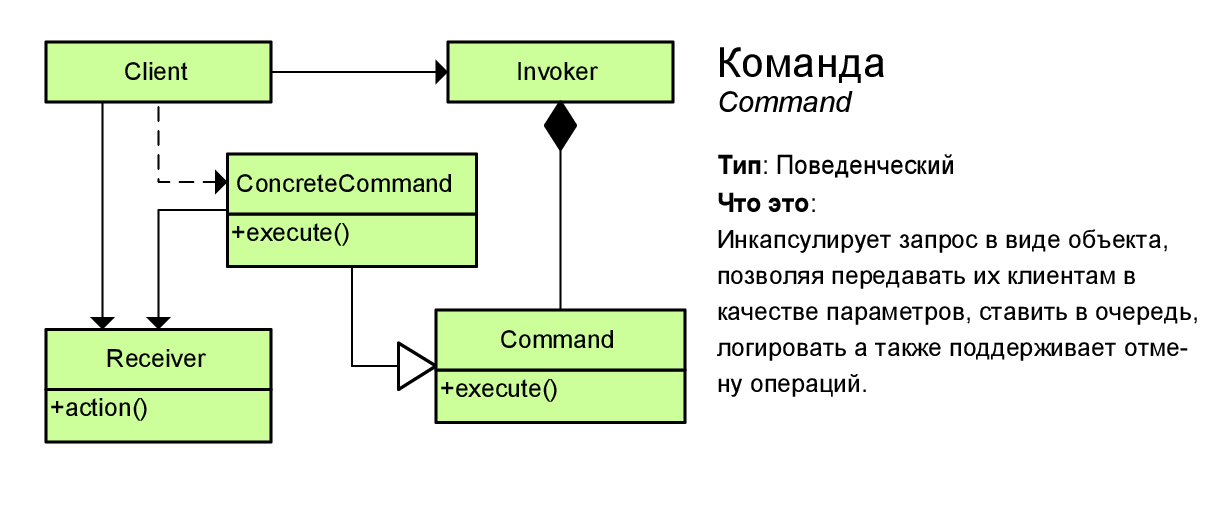
}

}

**Command (Команда)**

*Описание:*

Позволяет инкапсулировать различные операции в отдельные объекты.



**Реализация:**

interface Command {

void execute();

}

class Car {

public void startEngine() {

System.out.println("запустить двигатель");

}

public void stopEngine() {

System.out.println("остановить двигатель");

}

}

class StartCar implements Command {

Car car;

public StartCar(Car car) {

this.car = car;

}

public void execute() {

car.startEngine();

}

}

class StopCar implements Command {

Car car;

public StopCar(Car car) {

this.car = car;

}

public void execute() {

car.stopEngine();

}

}

class CarInvoker {

public Command command;

public CarInvoker(Command command){

this.command = command;

}

public void execute(){

this.command.execute();

}

}

public class CommandTest {//тест

public static void main(String[] args) {

Car car = new Car();

StartCar startCar = new StartCar(car);

StopCar stopCar = new StopCar(car);

CarInvoker carInvoker = new CarInvoker(startCar);

carInvoker.execute();

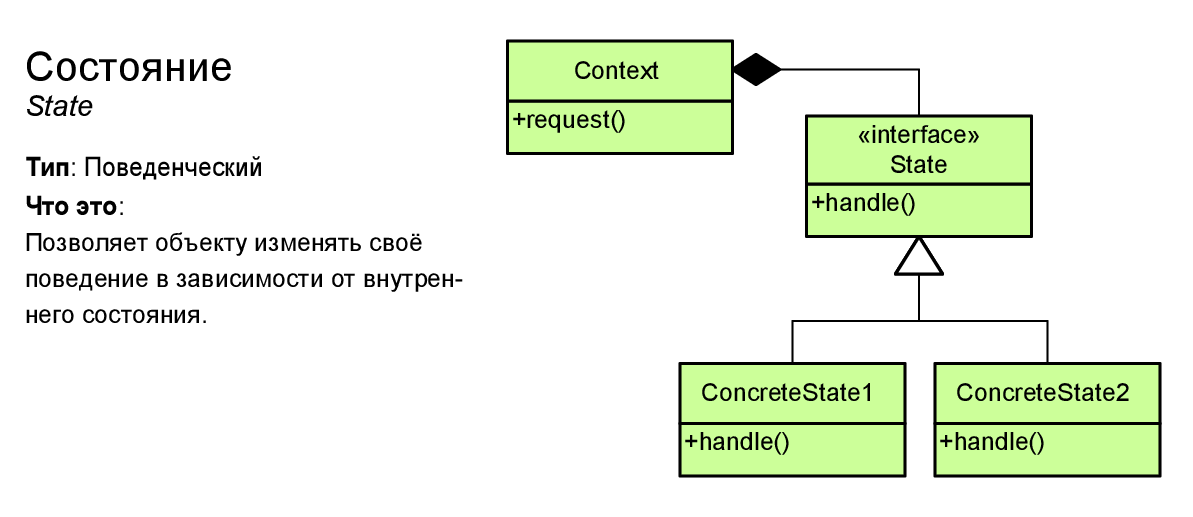
}

}

**State (Состояние)**

*Описание:*

Позволяет объекту изменять свое поведение в зависимости от его состояния.



**Реализация:**

interface State {

void doAction();

}

class StartPlay implements State {

public void doAction() {

System.out.println("start play");

}

}

class StopPlay implements State {

public void doAction() {

System.out.println("stop play");

}

}

class PlayContext implements State {

private State state;

public void setState(State state){

this.state = state;

}

public void doAction() {

this.state.doAction();

}

}

public class StateTest {//тест

public static void main(String[] args) {

PlayContext playContext = new PlayContext();

State startPlay = new StartPlay();

State stopPlay = new StopPlay();

playContext.setState(startPlay);

playContext.doAction();

playContext.setState(stopPlay);

playContext.doAction();

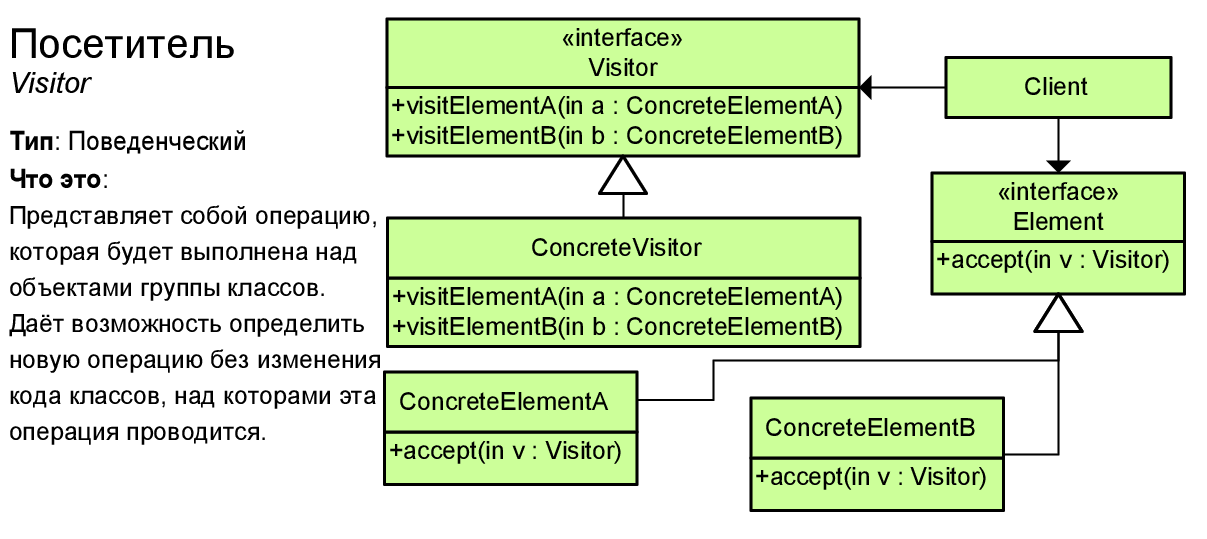
}

}

**Visitor (Посетитель)**

*Описание:*

Используется для упрощения операций над группировками связанных объектов.



**Реализация:**

interface Visitor {

void visit(SportCar sportCar);

void visit(Engine engine);

void visit(Whell whell);

}

interface Car {

void accept(Visitor visitor);

}

class Engine implements Car {

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

class Whell implements Car {

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

class SportCar implements Car {

Car[] cars;

public SportCar(){

cars = new Car[]{new Engine(), new Whell()};

}

public void accept(Visitor visitor) {

for (int i = 0; i < cars.length; i++) {

cars[i].accept(visitor);

}

visitor.visit(this);

}

}

class CarVisitor implements Visitor {

public void visit(SportCar computer) {

print("car");

}

public void visit(Engine engine) {

print("engine");

}

public void visit(Whell whell) {

print("whell");

}

private void print(String string) {

System.out.println(string);

}

}

public class VisitorTest {//тест

public static void main(String[] args) {

Car computer = new SportCar();

computer.accept(new CarVisitor());

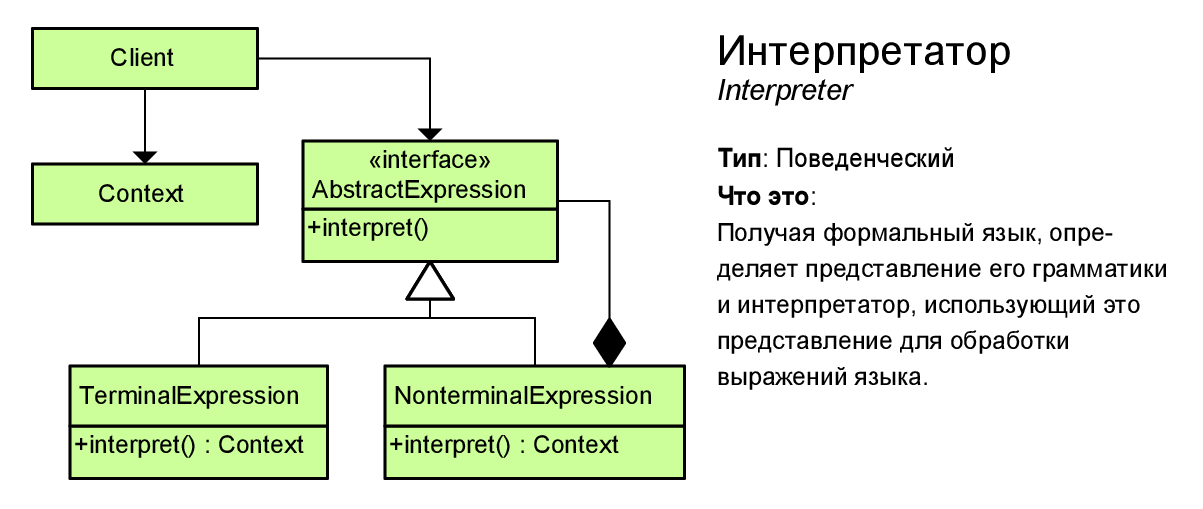
}

}

**Interpreter (Интерпретатор)**

*Описание:*

Позволяет определить грамматику простого языка для проблемной области.



**Реализация:**

interface Expression {

String interpret(Context context);

}

class Context {

public String getLowerCase(String s){

return s.toLowerCase();

}

public String getUpperCase(String s){

return s.toUpperCase();

}

}

class LowerExpression implements Expression {

private String s;

public LoverExpression(String s) {

this.s = s;

}

public String interpret(Context context) {

return context.getLoverCase(s);

}

}

class UpperExpression implements Expression {

private String s;

public UpperExpression(String s) {

this.s = s;

}

public String interpret(Context context) {

return context.getUpperCase(s);

}

}

public class InterpreterTest {//тест

public static void main(String[] args) {

String str = "TesT";

Context context = new Context();

Expression loverExpression = new LoverExpression(str);

str = loverExpression.interpret(context);

System.out.println(str);

Expression upperExpression = new UpperExpression(str);

str = upperExpression.interpret(context);

System.out.println(str);

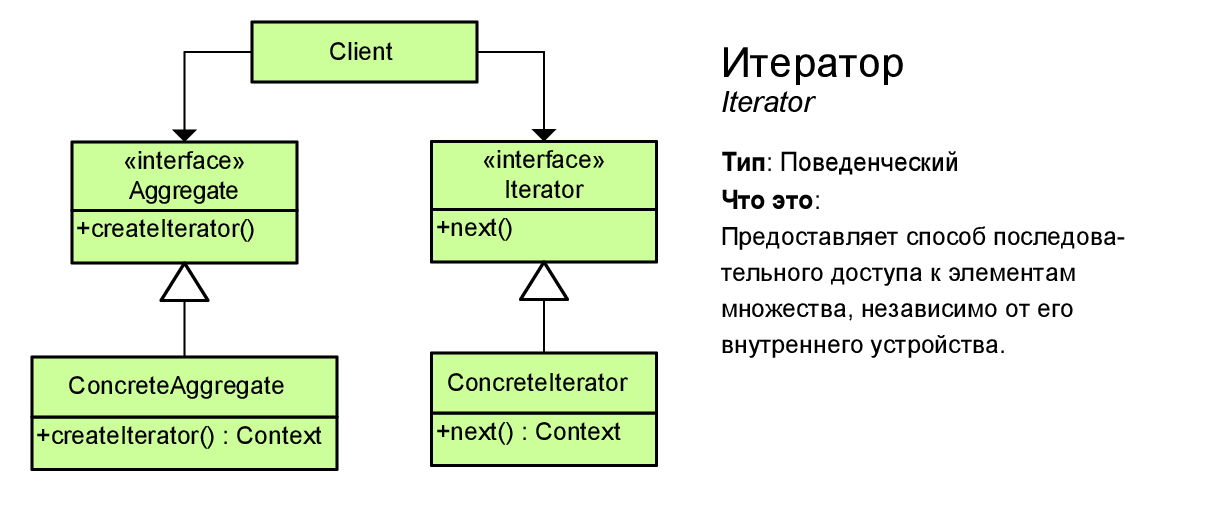
}

}

**Iterator (Итератор)**

*Описание:*

Последовательно осуществляет доступ к элементам объекта коллекции, не зная его основного представления.



**Реализация:**

interface Iterator {

boolean hasNext();

Object next();

}

class Numbers {

public int num[] = {1 , 2, 3};

public Iterator getIterator() {

return new NumbersIterator();

}

private class NumbersIterator implements Iterator {

int ind;

public boolean hasNext() {

if(ind < num.length) return true;

return false;

}

public Object next() {

if(this.hasNext()) return num[ind++];

return null;

}

}

}

public class IteratorTest {//тест

public static void main(String[] args) {

Numbers numbers = new Numbers();

Iterator iterator = numbers.getIterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

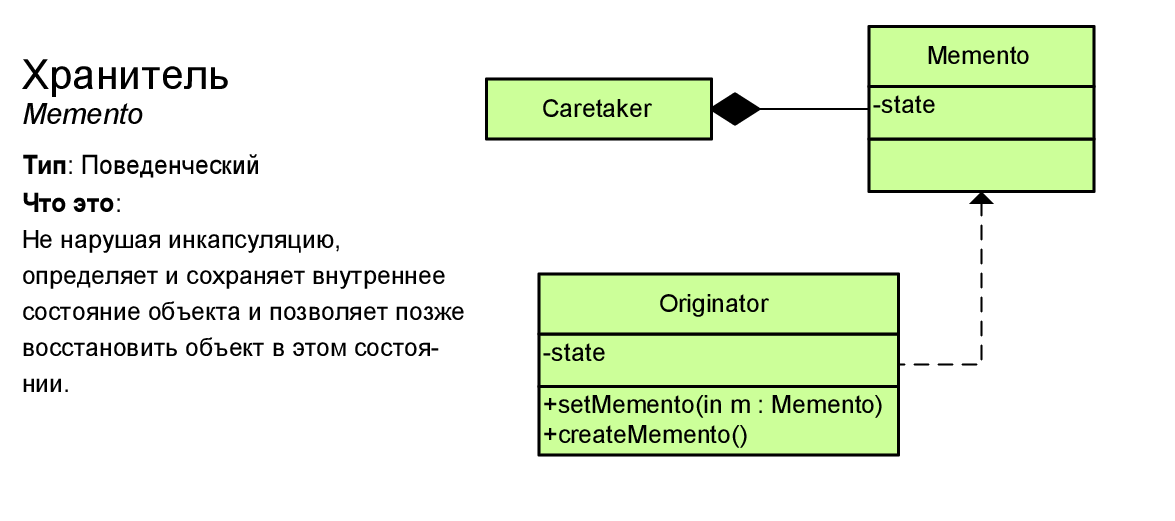
}

}

**Memento (Хранитель)**

*Описание:*

Позволяет сохранить текущее состояние объекта, позже это состояние можно восстановить. Не нарушает инкапсуляцию.



**Реализация:**

import java.util.ArrayList;

import java.util.List;

class Memento {

private String name;

private int age;

public Memento(String name, int age){

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

class User {

private String name;

private int age;

public User(String name, int age) {

this.name = name;

this.age = age;

System.out.println(String.format("create: name = %s, age = %s", name, age));

}

public Memento save(){

System.out.println(String.format("save: name = %s, age = %s", name, age));

return new Memento(name, age);

}

public void restore(Memento memento){

name = memento.getName();

age = memento.getAge();

System.out.println(String.format("restore: name = %s, age = %s", name, age));

}

}

**Варианты заданий**

|  |  |
| --- | --- |
| ***Вариант*** | ***Класс*** |
| 1 | Singleton, Adapter, Template Method |
| 2 | Factory, Composite, Mediator |
| 3 | Abstract Factory, Proxy, Chain of Responsibility |
| 4 | Builder, Flyweight, Observer |
| 5 | Prototype, Facade, Strategy |
| 6 | Singleton, Bridge, Command |
| 7 | Factory, Decorator, State |
| 8 | Abstract Factory, Adapter, Visitor |
| 9 | Builder, Composite, Interpreter |
| 10 | Prototype, Proxy, Iterator |
| 11 | Singleton, Flyweight, Memento |
| 12 | Factory, Facade, Template Method |
| 13 | Abstract Factory, Bridge, Mediator |
| 14 | Builder, Decorator, Chain of Responsibility |
| 15 | Prototype, Adapter, Observer |
| 16 | Singleton, Composite, Strategy |
| 17 | Factory, Proxy, Command |
| 18 | Abstract Factory, Flyweight, State |
| 19 | Builder, Facade, Visitor |
| 20 | Prototype, Bridge, Interpreter |
| 21 | Singleton, Decorator, Iterator |
| 22 | Factory, Adapter, Memento |