Паттерны проектирования.

автор: **alexKudryavtsev-web**

Структурные паттерны.

Структурные шаблоны — шаблоны, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Фасад.

*Фасад* - это паттерн, который предоставляет простой (но урезанный) интерфейс к сложной системе объектов, библиотеке или Фреймворку.

К, примеру, есть посудомоечная машина и чтоб вместо указания стиля стирки, отжима, температуры и т.д. все это можно одной кнопкой, которая настроит систему по – умолчанию. В этой ситуация эта кнопка, является фасадом посудомоечной машины.

Такую же аналогию можно привести к компьютеру. Кнопка «Включение» является фасадом всех внутренних систем компьютера. Пускай, имеется простая симуляция работы машины:

class Power{

private boolean isActive = false;

public void on(){

System.out.println ("Компьютер включен");

isActive = true;

}

public void off(){

System.out.println("Компьютер выключен");

isActive = false;

}

public boolean isActive() {

return isActive;

}

}

class DVDRoom{

private boolean data = false;

public boolean hasData (){

return data;

}

public void load(){

data = true;

}

public void unload(){

data = false;

}

}

class HDD{

void copyDVD (DVDRoom dvd){

if(dvd.hasData())

System.out.println ("Копирование данных");

else

System.out.println ("Вставьте диск с данными");

}

}

Вот так может выглядеть его фасад:

class Computer{

private Power power = new Power();

private DVDRoom dvd = new DVDRoom();

private HDD hdd = new HDD();

public void copy(){

power.on ();

dvd.load ();

hdd.copyDVD (dvd);

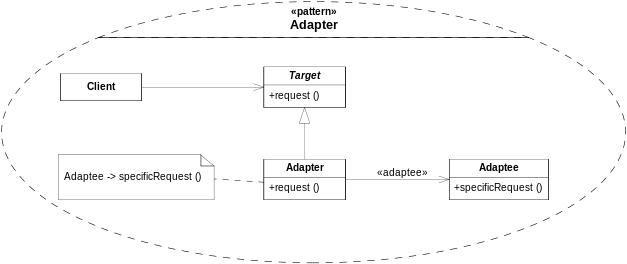
}

}

Паттерн Адаптер.

Адаптер — это паттерн, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс. Он выступает прослойкой между двумя объектами, превращая вызовы одного в вызовы понятные другому.

UML адаптера:



Различают два способа создать адаптер. Первый это создание путем наследования Adaptee и второй способ это с использованием композиции.

Пример создания адаптеры при помощи расширения:

//Target

interface VectorGraphicsInterface{

void drawLine ();

void drawSquare ();

}

//Adaptee

class RasterGraphics{

void drawRasterLine (){

System.out.println("Рисуем линию");

}

void drawRasterSquare (){

System.out.println ("Рисуем квадрат");

}

}

//Adapter

class Adapter extends RasterGraphics

implements VectorGraphicsInterface{

@Override

public void drawLine () {

drawRasterLine ();

}

@Override

public void drawSquare () {

drawRasterSquare ();

}

}

Пример использования:

Adapter adapter = new Adapter ();

adapter.drawSquare ();

adapter.drawLine ();

Реализации адаптера при помощи композиции:

//Adapter

class Adapter implements VectorGraphicsInterface{

RasterGraphics graphics;

Adapter (RasterGraphics graphics){

this.graphics = graphics;

}

@Override

public void drawLine() {

graphics.drawRasterLine ();

}

@Override

public void drawSquare() {

graphics.drawRasterSquare ();

}

}

Мост.

Мост это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо, друг от друга.

Есть класс Figure, который имеет подклассы Circle и Square. Нужно расширить иерархию фигур по цвету, то есть иметь Красные и Синие фигуры. Но чтобы всё это объединить, придётся создать 4 комбинации подклассов, вроде BlueCirlce и RedCircle.

Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.

Чтобы решить эту проблему можно разделить эту иерархию на два иерархия. Первая это *абстракция*, которая содержит класс Figure и его подклассы. Вторая это *реализация*, которая содержит интерфейс ColorFigure и его реализации Blue и White. При этом Figure содержит ссылку на ColorFigure.

Как это выглядит в коде:

//Абстракция

abstract class Figure{

protected ColorFigure color;

private int x, y, radius;

protected Figure(ColorFigure color, int x, int y, int radius){

this.color = color;

setRadius (radius);

setX (x);

setY (y);

setY (y);

}

public int getRadius() { return radius; }

public int getX() { return x; }

public int getY() { return y; }

public void setRadius(int radius) { this.radius = radius; }

public void setX(int x) { this.x = x; }

public void setY(int y) { this.y = y; }

abstract void draw();

}

class Square extends Figure{

protected Square(ColorFigure color, int x, int y, int radius) {

super (color, x, y, radius);

}

@Override

void draw() {

System.out.println ("Куб {");

System.out.println (getX () + ":" + getY ());

System.out.println (getRadius () +"\n}");

color.setColor ();

}

}

class Circle extends Figure{

protected Circle(ColorFigure color, int x, int y, int radius) {

super(color, x, y, radius);

}

@Override

void draw() {

System.out.println ("Шар {\n"+

getX ()+":"+getY ()+

"\n"+getRadius ());

color.setColor ();

}

}

//Реализация

interface ColorFigure{

void setColor();

}

class Blue implements ColorFigure{

@Override

public void setColor () {

System.out.println("Установлен синий цвет.");

}

}

Использование:

Figure [] figures = {

new Square(new Blue(), 5, 25, 30),

new Circle(new White(), 10, 10, 5)

};

for (Figure f: figures) {

f.draw ();

System.out.println ("\n");

}

class White implements ColorFigure{

@Override

public void setColor() {

System.out.println("Установлен белый цвет.");

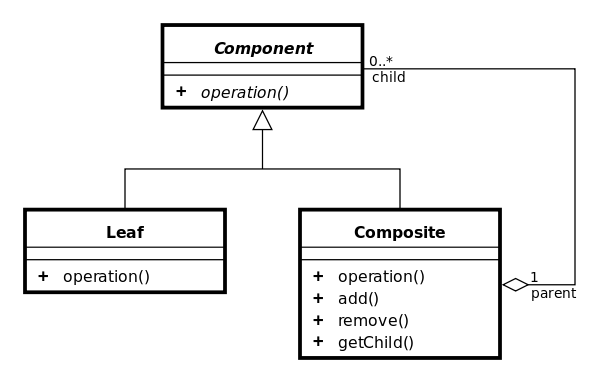
}

}

Компоновщик.

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как, будто это единичный объект.

Иными словами данный паттерн позволяет реализовывать дерево. UML компоновщика:



Пример использования данного паттерна:

interface Sharpe{

void draw ();

}

class Circle implements Sharpe{

@Override

public void draw() {

System.out.println ("Круг");

}

}

class Square implements Sharpe{

@Override

public void draw() {

System.out.println ("Квадрат");

}

}

class Triangle implements Sharpe{

@Override

public void draw() {

System.out.println ("Треугольник");

}

}

class Composite implements Sharpe{

private ArrayList<Sharpe> list = new ArrayList<>();

@Override

public void draw() {

for (Sharpe s: list) {

s.draw();

}

}

public void add(Sharpe element){

list.add (element);

}

public void remove (Sharpe element){

list.remove(element);

}

}

Использование:

Composite mainComposite = new Composite ();

Composite composite = new Composite ();

Triangle triangle = new Triangle ();

composite.add (triangle);

Circle circle = new Circle ();

composite.add (circle);

Square square = new Square ();

composite.add (square);

mainComposite.add (composite);

Triangle triangle1 = new Triangle ();

mainComposite.add (triangle1);

mainComposite.draw ();

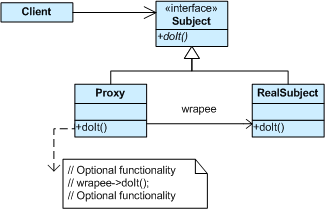
Паттерн Proxy.

Заместитель — это паттерн, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

Обычно его используют в следующих случаях:

* Удаленное замещение – локальный представитель вместо объекта в другом пространстве.
* Виртуальное замещение – создание емкого объекта при необходимости.
* Защищенное замещение – контроль доступа к указанному объекту.
* Умная ссылка - замена обычного указателя.

UML паттерна:



Ниже приведенная программа демонстрирует применение данного паттерна:

interface Project{

void run();

}

class RealProject implements Project{

private String url;

RealProject (String url){

this.url = url;

load ();

}

@Override

public void run() {

System.out.println("Выполнение проекта");

}

public void load () {

System.out.println("Загрузка проекта из "+url);

}

}

class ProxyProject implements Project{

private String url;

private RealProject project;

ProxyProject (String url){

this.url = url;

}

@Override

public void run() {

if (project == null)

project = new RealProject (url);

project.run ();

}

}

Применение:

Project project =

new ProxyProject (

"https://github.com/");

project.run();

Паттерн Flyweight.

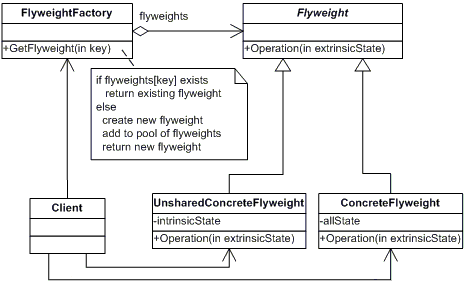
Паттерн Flyweight— структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

Flyweight используется для уменьшения затрат при работе с большим количеством мелких объектов. При проектировании приспособленца необходимо разделить его свойства на внешние и внутренние.

Внутренние свойства всегда неизменны, тогда как внешние могут отличаться в зависимости от места и контекста применения и должны быть вынесены за пределы Flyweight.

Flyweight дополняет Фабричный метод таким образом, что при обращении клиента Фабричный метод для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту. Если такого объекта нет, то фабрика создаст новый объект.

UML диаграмма:



Реализация паттерна:

//Flyweight

interface Developer{

void writeCode ();

}

class JavaDeveloper implements Developer{

@Override

public void writeCode() {

System.out.println ("Java программист пишет код.");

}

}

class CppDeveloper implements Developer{

@Override

public void writeCode() {

System.out.println("С++ программист пишет код.");

}

}

//FlyweightFactory

class DeveloperFactory{

private static final Map<String, Developer> map =

new HashMap<>();

public Developer getDeveloperBySpecialty (String specialty){

Developer d = map.get (specialty);

if(d == null){

switch (specialty){

case "java":

d = new JavaDeveloper();

System.out.println("Устройство Java программиста");

break;

case "c++":

d = new CppDeveloper ();

System.out.println("Устройство C++ программиста.);

}

map.put (specialty, d);

}

return d;

}

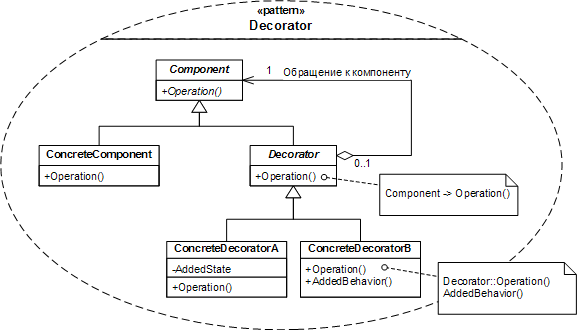
}

Декоратор.

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.

Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

UML диаграмма паттерна:



Создание данного паттерна:

interface PrintInterface{

void print();

}

class SimplePrint implements PrintInterface{

private String text;

public SimplePrint(String text){

this.text = text;

}

@Override

public void print() {

System.out.print(text);

}

}

abstract class Decorator implements PrintInterface{

protected PrintInterface obj;

Decorator(PrintInterface printer){

obj = printer;

}

}

class RightBracketDecorator extends Decorator{

RightBracketDecorator (PrintInterface printText) {

super(printText);

}

@Override

public void print() {

System.out.print ("[");

obj.print ();

}

}

class LeftBracketDecorator extends Decorator{

LeftBracketDecorator (PrintInterface printer) {

super (printer);

}

@Override

public void print() {

obj.print ();

System.out.print ("]");

}

}

class BracesDecorator extends Decorator{

BracesDecorator (PrintInterface printer){

super (printer);

}

@Override

public void print() {

System.out.print ("{");

obj.print ();

System.out.print ("}");

}

}

Использование:

PrintInterface printer =

new RightBracketDecorator(

new BracesDecorator(

new SimplePrint("Hello")));

printer.print ();

Порождающие шаблоны.

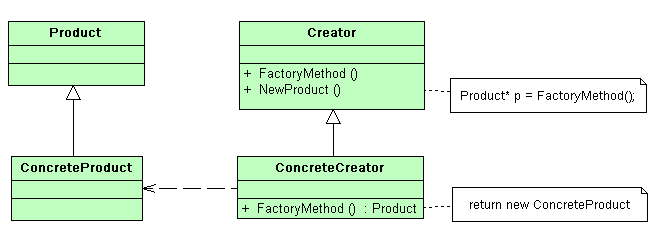
Порождающие шаблоны— шаблоны, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

Фабричный метод.

Фабричный метод — это паттерн, который решает проблему создания различных продуктов, без указания конкретных классов продуктов.

Он всегда используется, когда требуется гибкость при создании продуктов. Вообще он позволяет не использовать прямо new Подклассы могут переопределить этот метод, чтобы изменять тип создаваемых продуктов.

UML диаграмма паттерна:



Пример реализации паттерна Фабричный метод:

public class Main {

public static void main(String[] args) {

DeveloperFactory factory = createDeveloperBySpeciality (TypeDeveloper.C\_Sharp);

Developer developer = factory.createDeveloper ();

developer.writeCode ();

}

public static DeveloperFactory createDeveloperBySpeciality(TypeDeveloper type){

if(type == TypeDeveloper.JAVA)

return new JavaDeveloperFactory();

else if(type == TypeDeveloper.C\_Sharp)

return new CSharpDeveloperFactory();

else

return null;

}

}

enum TypeDeveloper{

JAVA, C\_Sharp

}

interface Developer{

void writeCode();

}

class JavaDeveloper implements Developer{

@Override

public void writeCode() {

System.out.println ("Пишу на Java");

}

}

class CSharpDeveloper implements Developer{

@Override

public void writeCode() {

System.out.println ("Пишу на C#");

}

}

interface DeveloperFactory{

Developer createDeveloper ();

}

class JavaDeveloperFactory implements DeveloperFactory{

@Override

public Developer createDeveloper() {

return new JavaDeveloper();

}

}

class CSharpDeveloperFactory implements DeveloperFactory{

@Override

public Developer createDeveloper() {

return new CSharpDeveloper();

}

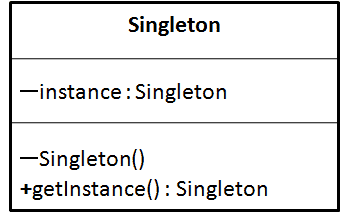
}

Паттерн Одиночка.

*Одиночка* это паттерн, который гарантирует существование только одного объекта определённого класса, а также позволяет достучаться до этого объекта из любого места программы.

Его плюсом считается то, что он удобен при работе, но с другой стороны нарушается модульность программы.

Его UML выглядит так:



Также много камней встречается, когда несколько потоков используют одиночку, что сулит ошибкам. Можно сделать саму экземпляр volatile, а внутри getInstance () реализовать синхронизированный блок.

Вот так может выглядеть выше изложенная одиночка, на примере создания логгера:

class ProgramLogger{

private static volatile ProgramLogger logger;

private static String logFile = "";

public static ProgramLogger getLogger() {

if(logger == null) {

synchronized (ProgramLogger.class){

if (logger == null)

logger = new ProgramLogger();

}

}

return logger;

}

public void addInfoLog(String logInfo){

logFile += logInfo;

}

public static String show(){

return logFile;

}

private ProgramLogger(){ }

}

Пример использования:

ProgramLogger.getLogger ().addInfoLog ("Первый лог\n");

ProgramLogger.getLogger ().addInfoLog ("Второй лог\n");

ProgramLogger.getLogger ().addInfoLog ("Третий лог\n");

System.out.println ("Лог: \n"+ ProgramLogger.show ());

System.out.println (ProgramLogger.getLogger ().toString ());

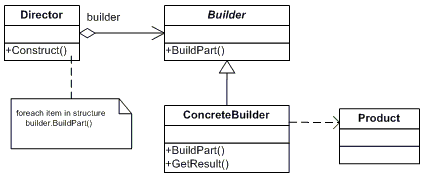
System.out.println (ProgramLogger.getLogger ().toString ());

System.out.println (ProgramLogger.getLogger ().toString ());

Шаблон Строитель

Строитель – это порождающий шаблон проектирования предоставляет способ создания составного объекта. Он отделяет конструирование сложного объекта от его представления так, что в результате одного и того же процесса конструирования могут получаться разные представления.

UML диаграмма паттерна:



Ниже представлен пример использования паттерна Строитель:

enum Transmission{

AUTO, MANUAL

}

class Car{

private String mark;

private int maxSpeed;

private Transmission transmission;

public void setMark(String mark) {

this.mark = mark;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

public void setTransmission(Transmission transmission) {

this.transmission = transmission;

}

@Override

public String toString() {

return "Car{"+mark+

", "+maxSpeed+", "+

transmission+"}";

}

}

abstract class CarBuilder{

Car car;

void createCar(){

car = new Car();

}

abstract void buildMaxSpeed();

abstract void buildMark();

abstract void buildTransmission();

public Car getCar() {

return car;

}

}

class TeslaBuilder extends CarBuilder{

@Override

void buildMark() {

car.setMark ("Тесла");

}

@Override

void buildTransmission() {

car.setTransmission (Transmission.AUTO);

}

@Override

void buildMaxSpeed() {

car.setMaxSpeed (300);

}

}

class BMVBuilder extends CarBuilder{

@Override

void buildMark() {

car.setMark ("BMV");

}

@Override

void buildTransmission() {

car.setTransmission (Transmission.MANUAL);

}

@Override

void buildMaxSpeed() {

car.setMaxSpeed (280);

}

}

class Director{

private CarBuilder builder;

public void setBuilder(CarBuilder builder) {

this.builder = builder;

}

Car buildCar (){

builder.createCar ();

builder.buildMark ();

builder.buildMaxSpeed ();

builder.buildTransmission ();

return builder.getCar();

}

}

Пример использования:

Director director = new Director ();

director.setBuilder (new BMVBuilder ());

Car car = director.buildCar ();

System.out.println(car);

Упрощенный строитель.

Кроме вышеописанного строителя существует второй более упрощенный аналог строителя. Его особенность в том, что в нем есть два класс: Продукт и Строитель. В таком случае нет специализированных строителей. Вот так может, выглядит так:

class CarBuilder{

private String mark = "No mark";

private int maxSpeed = -1;

private Transmission transmission =

Transmission.MANUAL;

CarBuilder buildMaxSpeed (int maxSpeed){

this.maxSpeed = maxSpeed;

return this;

}

CarBuilder buildMark (String mark){

this.mark = mark;

return this;

}

CarBuilder buildTransmission (Transmission transmission) {

this.transmission = transmission;

return this;

}

Car build (){

Car c = new Car ();

c.setMaxSpeed (maxSpeed);

c.setTransmission (transmission);

c.setMark (mark);

return c;

}

}

Использование строителя:

Car tesla = new CarBuilder ()

.buildMark ("Tesla")

.buildMaxSpeed (280)

.buildTransmission (Transmission.AUTO)

.build ();

System.out.println (tesla);

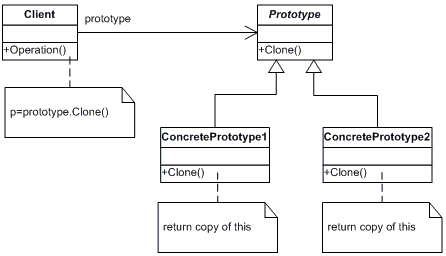
Прототип.

Прототип это паттерн, который позволяет копировать объекты любой сложности без привязки к их конкретным классам.

Все классы— Прототипы имеют общий интерфейс. Поэтому можно копировать объекты, не обращая внимания на их конкретные типы и всегда быть, уверены, что получите точную копию.

Клонирование совершается самим объектом прототипам, что позволяет ему скопировать значения всех полей, даже приватных.

UML диаграмма паттерна:



Ниже представлен пример использования паттерна Прототип:

interface Copyable<T>{

T copy ();

}

class Human implements Copyable<Human>{

private String name;

private int old;

public Human(String name, int old){

this.name = name;

this.old = old;

}

@Override

public Human copy() {

return new Human(name, old);

}

@Override

public String toString() {

return "Human{"+name+

", "+old+"}";

}

}

class HumanFactory{

private Human human;

HumanFactory(Human prototype){

human = prototype;

}

public void setPrototype(Human human) {

this.human = human;

}

Human makeCopy () {

return human.copy();

}

}

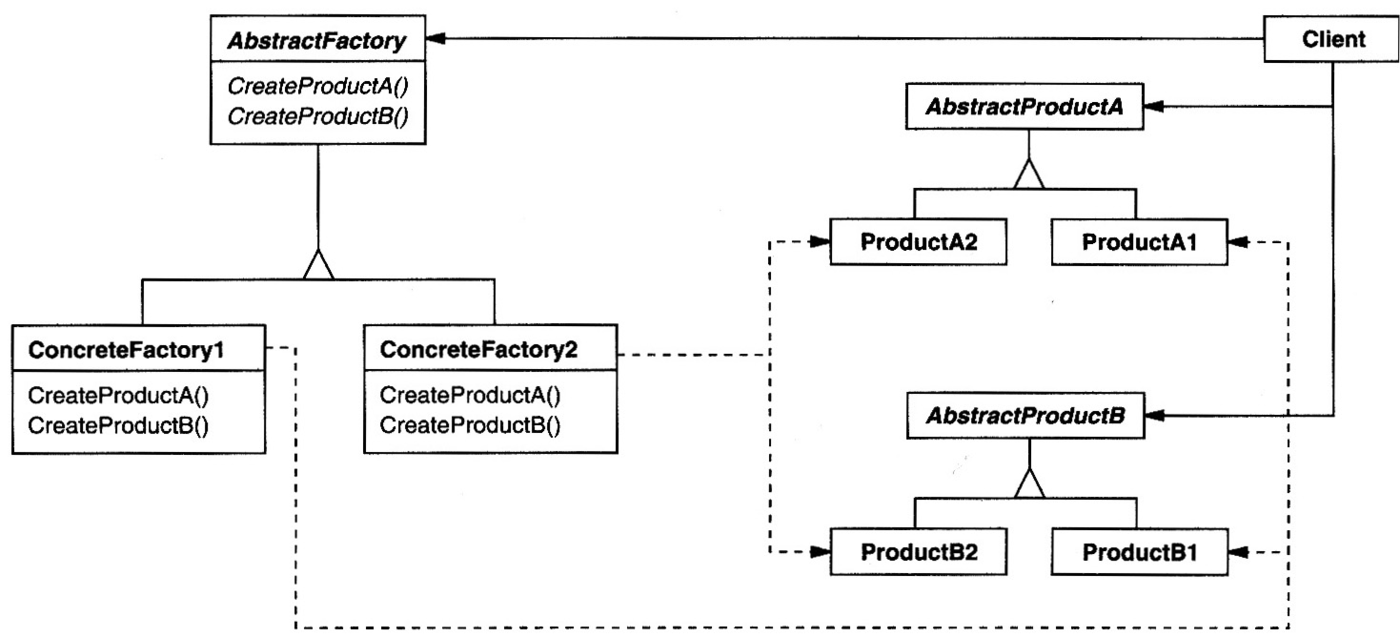
Абстрактная фабрика.

Абстрактная фабрика — это порождающий паттерн проектирования, который решает проблему создания целых семейств, связанных продуктов, без указания конкретных классов продуктов.

Абстрактная фабрика задаёт интерфейс создания всех доступных типов продуктов, а каждая конкретная реализация фабрики порождает продукты одной из вариаций.

Клиентский код вызывает методы фабрики для получения продуктов, вместо самостоятельного создания с помощью оператора new. При этом фабрика сама следит за тем, чтобы создать продукт нужной вариации.

UML диаграмма паттерна:



Чтобы понять данную UML – диаграмму следует привести пример применения. Методы интерфейса DevelopmentTeamFactory (AbstractFactory) должны возвращать программистов (getProgrammer ()) и управляющего (getManager()).

Для программистов пускай будет интерфейс Programmer и у него будет метод writeCode () и для управляющего ProjectManager с методом control ().

Также у них будут реализации под создание (JavaDeveloper) и управление банков (BankingPM). Еще должен быть BankingDevelopmentTeam (ConcreteFactory1).

Такое же и надо провернуть с командой, но уже для wed – программистов. Класс Client вызывает методы фабрики для получения продуктов, вместо самостоятельного создания с помощью оператора new. При этом фабрика сама следит за тем, чтобы создать продукт нужной вариации.

Все вышесказанное в коде может выглядеть так. Ради краткости показан только код для создания группы для банка.

**Интерфейс Programmer**:

public interface Programmer {

void writeCode ();

}

**Интерфейс ProjectManager:**

public interface ProjectManager {

void control ();

}

**Интерфейс DevelopmentTeamFactory**:

public interface DevelopmentTeamFactory {

Programmer getProgrammer ();

ProjectManager getManager ();

}

**Класс JavaDeveloper:**

public class JavaDeveloper implements Programmer {

@Override

public void writeCode() {

System.out.println ("Java программист пишет код.");

}

}

**Класс BankingPM:**

public class BankingPM implements ProjectManager{

@Override

public void control() {

System.out.println ("Глава управляет за командой");

}

}

**Класс BankingDevelopmentTeam:**

public class BankingDevelopmentTeam

implements DevelopmentTeamFactory {

@Override

public Programmer getProgrammer() {

return new JavaDeveloper();

}

@Override

public ProjectManager getManager() {

return new BankingPM();

}

}

**Клиент:**

public class Client {

public static void main(String[] args) {

BankingDevelopmentTeam team = new BankingDevelopmentTeam ();

ProjectManager manager = team.getManager ();

Programmer programmer = team.getProgrammer ();

programmer.writeCode ();

manager.control ();

}

}

Поведенческие шаблоны.

Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Паттерн Iterator.

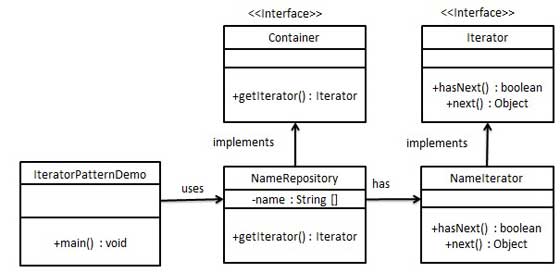
Iterator - это шаблон проектирования, который позволяет пройтись по всем элементам некоторого составного объекта.

Одним из важных условий при реализации паттерна является то, что итератор должен гарантировать не раскрытие внутреннего устройства объекта.

Например, такие элементы как дерево, связанный список, хэш-таблица и массив могут быть пролистаны (и модифицированы) с помощью объекта Итератор.

Он реализуется по средствам двух интерфейс. Первый Iterator, который определяет два методы hasNext () и next (), а также второй Iterable, метод которого должен возвращать реализацию Iterator.

Вот так выглядит UML изображение паттерна:



Также во многих принято в интерфейсе Iterator добавлять метод reset(), который позволяет перезапустить итератор. Обычно метод next (), если нечего возвращать генерирует исключение NoSuchFieldException.

Простой пример без использования Iterator стандартного API:

interface Iterator{

Object next() throws NoSuchFieldException;

boolean hasNext();

void reset();

}

interface Iterable{

Iterator getIterator ();

}

class NameRepository implements Iterable{

private List<String> list = Arrays.asList(

"Фасад",

"Одиночка",

"Фабричный метод",

"Стратегия");

@Override

public Iterator getIterator() {

return new NameRepositoryIterable();

}

private class NameRepositoryIterable implements Iterator{

int index = 0;

@Override

public boolean hasNext() {

if(index<list.size())

return true;

return false;

}

@Override

public Object next() throws NoSuchFieldException {

if(this.hasNext()) {

return list.get(index++);

}

else

throw new NoSuchFieldException();

}

@Override

public void reset() {

index = 0;

}

}

}

Здесь стоит отметить реализацию итератор. Сам класс NameRepository реализует Iterable, но у него есть закрытый внутренний класс, который и реализует Iterator.

Пример использования:

NameRepository repository = new NameRepository ();

Iterator iterator = repository.getIterator ();

while (iterator.hasNext())

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

iterator.reset ();

while (iterator.hasNext())

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

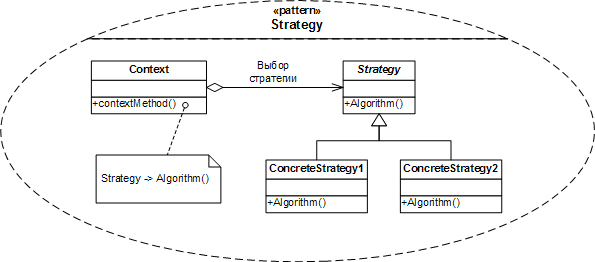
Паттерн Стратегия.

К примеру, есть у человек есть 3 вариантов доехать из Москвы до Питера. Это езжать на машине, на велосипеде и тележке из супермаркета. И он выбирается, к примеру, последний путь, тогда человек едет до Питера на тележке. Это и есть паттерн стратегия.

*Состояние* – это шаблон, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса.

Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

UML паттерна:



Из изложенного сверху описания можно понять, что для его реализации нужно иметь интерфейс ChoiceWays , у которого есть метод move(). Еще есть пять реализаций с конкретными реализациями, как добраться до Питера.

Также должен быть класс User, у которого будет поле ChoiceWays и его сеттер, а также moveToPiter(), который будет вызвать метод move(). Выше сказанные слова в коде может выглядеть так:

class User {

private ChoiceWays choice;

public void setChoice(ChoiceWays choice) {

this.choice = choice;

}

public void moveToPiter(){

choice.move ();

}

}

interface ChoiceWays{

void move();

}

class MoveCar implements ChoiceWays{

@Override

public void move() {

System.out.println ("Едим на машине");

}

}

class RideBicycle implements ChoiceWays{

@Override

public void move() {

System.out.println ("Катимся на велосипеде");

}

}

class MovingOnCart implements ChoiceWays{

@Override

public void move() {

System.out.println ("На тележке в Питер");

}

}

Пример использования:

User user = new User ();

user.setChoice (new RideBicycle ());

user.moveToPiter ();

user.setChoice (new MoveCar ());

user.moveToPiter ();

user.setChoice (new MovingOnCart ());

user.moveToPiter ();

Паттерн состояние.

Пускай у человека есть 3 вариантов доехать из Москвы до Питера. Это езжать на машине, на велосипеде и тележке из супермаркета. И он решает то, что сначала поедет на велосипеде, потом пересядет на тележку и после доедет на машине. Это и есть паттерн состояние.

*Состояние* это шаблон, который используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

Вся левая часть UML диаграммы Состояния аналогична диаграмме Стратегии. Но в отличие от Стратегии в Состоянии контекст может сам менять значение поле, содержащее ссылку интрефейс.

Пример состояние на примере изменения класса User:

class User{

private ChoiceWays choice;

public void setChoice(ChoiceWays choice) {

this.choice = choice;

}

public void moveToPiter(){

choice.move();

}

public void nextTransport(){

if (choice instanceof MoveCar)

setChoice (new RideBicycle());

else if(choice instanceof MovingOnCart)

setChoice (new MoveCar());

else if(choice instanceof RideBicycle)

setChoice(new MoveCar());

}

}

Пример использования состояния:

User user = new User();

user.setChoice (new MoveCar ());

for (int i = 0; i < 3; i++) {

user.nextTransport();

user.moveToPiter();

}

Шаблонный метод.

Шаблонный метод - шаблон, определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

К примеру, есть класс A с метод method(), который выводит числа 1, 2, 4. Также есть класс B с аналогичным методом method(), который выводит уже 1, 2 и 3. По существу реализации двух методов скопированы.

Для решения данной проблемы можно создать абстрактный класс C с final методом templateMethod() и абстрактным методом differ ().

Первый выводит 1 и 2, а далее вызывает differ (), который содержит разницу между реализацией. Вот так может это выглядеть в коде:

abstract class C {

final void templateMethod(){

System.out.print ("1");

System.out.print ("2");

differ ();

}

abstract void differ();

}

class A extends C{

@Override

void differ() {

System.out.print ("4");

}

}

class B extends C{

@Override

void differ() {

System.out.print ("3");

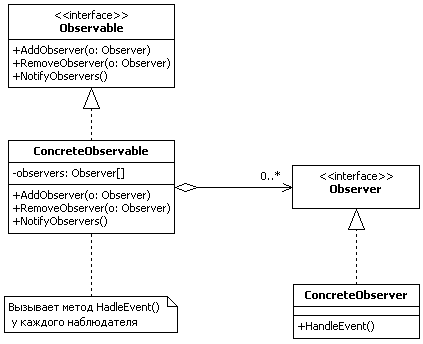
}

}

Наблюдатель.

Наблюдатель (англ. Observer) — шаблон, который позволяет объекту этого класса получать оповещения об изменении состояния других объектов, и тем самым наблюдать за ними.

UML диаграмма паттерна:



*Здесь*:

* Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей.
* Observer — интерфейс, с помощью которого наблюдатель получает оповещение.
* ConcreteObservable — конкретный класс, который реализует интерфейс Observable.
* ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

Паттерн Наблюдатель следует использовать в следующих случаях:

* Существует, как минимум, один объект, рассылающий сообщения.
* Есть не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения.
* Нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Ниже представлен пример использования Наблюдателя в создании простой версии газеты:

interface Observer{

void handleEvent(ArrayList<String> news);

}

class User implements Observer{

private String name;

public User(String name){

this.name = name;

}

@Override

public void handleEvent(ArrayList<String> news) {

System.out.println(name +", сейчас в мире: ");

news.forEach(System.out::println);

System.out.print ("\n");

}

}

interface Observable{

void addObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObserver ();

}

class Newspaper implements Observable{

private ArrayList<Observer> subscribers = new ArrayList<>();

private ArrayList<String> news = new ArrayList<>();

public void addNews (String news){

this.news.add (news);

}

public void addAllNews(String... news){

this.news.addAll (Arrays.asList (news));

}

@Override

public void addObserver(Observer observer) {

subscribers.add (observer);

}

@Override

public void removeObserver(Observer observer) {

subscribers.remove (observer);

}

@Override

public void notifyObserver() {

for (Observer observer : subscribers)

observer.handleEvent (news);

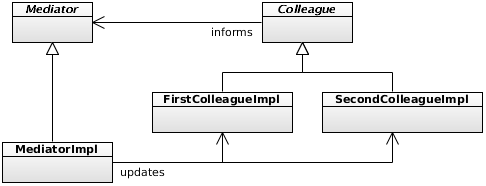
}

}

Посредник.

Посредник — паттерн, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность и избавляя, объекты от необходимости явно ссылаться друг на друга.

UML – диаграмма паттерна:



Чтобы понять данный паттерн нужно рассмотреть обычный чат. Сам интерфейс чата Chat это *Mediator*. У него должна быть конкретная реализация, к примеру, класс SimpleChat ().

Интерфейс Colleague представляет пользователей. Админа *FirstColleagueImpl* и Обычного пользователя *SecondColleagueImpl*.

Вот так это может выглядеть в коде:

interface Chat{

void sendMessage(String message, User user);

}

interface User{

void sendMessage(String message);

void getMessage(String message);

}

class SimpleUser implements User{

private Chat chat;

private String name;

SimpleUser (Chat chat, String name){

this.chat = chat;

this.name = name;

}

@Override

public void sendMessage(String message) {

chat.sendMessage (message, this);

}

@Override

public void getMessage(String message) {

System.out.println (this.name+" получил: "+message);

}

}

class Admin implements User{

private Chat chat;

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Admin(Chat chat, String name){

this.chat = chat;

this.name = name;

}

@Override

public void sendMessage(String message) {

chat.sendMessage (message, this);

}

@Override

public void getMessage (String message) {

System.out.println (this.name+" получил: "+message);

}

}

class SimpleChat implements Chat{

private Admin admin;

private List<User> listUsers = new ArrayList<>();

public void setAdmin(Admin admin) {

this.admin = admin;

}

public void addUserToChat(User user){

listUsers.add (user);

}

@Override

public void sendMessage(String message, User user) {

for (User u: listUsers) {

if(!u.equals(user)){

u.getMessage (message);

}

}

admin.getMessage (message);

}

}

Использование:

SimpleChat chat = new SimpleChat ();

SimpleUser user1 = new SimpleUser (chat, "User-1");

chat.addUserToChat (user1);

SimpleUser user2 = new SimpleUser (chat, "User-2");

chat.addUserToChat (user2);

Admin admin = new Admin (chat, "Петух");

chat.setAdmin (admin);

user1.sendMessage ("Hello, World ");

Команда.

*Команда* — шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Команда используется для инкапсуляции всей информации, необходимой для выполнения действия или вызова события в более позднее время.

Эта информация включает в себя имя метода, объект, который является владельцем метода и значения параметров метода.

Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода.

UML диаграмма паттерна состоит из четырех «терминов»: команды (command), приёмник команд (receiver), вызывающий команды (invoker) и клиент (client).

UML – диаграмма паттерна:



Пример создания паттерна:

//Invoker

class Computer{

public void start(){

System.out.println ("Start");

}

public void stop(){

System.out.println ("Stop");

}

public void reset(){

System.out.println ("Reset");

}

}

//Command

interface Command{

void execute();

}

//ConcreteCommand1

class StartCommand implements Command{

private Computer comp;

StartCommand (Computer comp){

this.comp = comp;

}

@Override

public void execute() {

comp.start ();

}

}

//ConcreteCommand2

class StopCommand implements Command{

private Computer comp;

StopCommand (Computer comp){

this.comp = comp;

}

@Override

public void execute() {

comp.stop ();

}

}

//ConcreteCommand3

class ResetCommand implements Command{

private Computer comp;

public ResetCommand(Computer comp) {

this.comp = comp;

}

@Override

public void execute() {

comp.reset ();

}

}

//Receiver

class User{

private Command start, stop, reset;

User (Command start, Command stop, Command reset){

this.start = start;

this.reset = reset;

this.stop = stop;

}

public void startComp(){

start.execute ();

}

public void stopComp(){

stop.execute ();

}

public void resetComp (){

reset.execute ();

}

}

Использование:

Computer comp = new Computer ();

User user = new User (

new StartCommand(comp),

new StopCommand(comp),

new ResetCommand(comp));

user.startComp ();

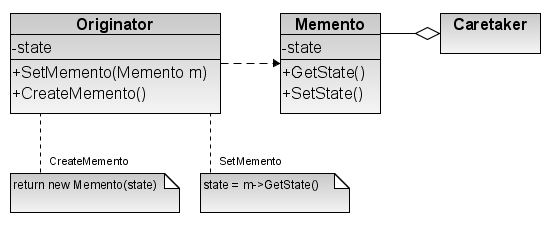
user.stopComp ();

user.resetComp ();

Хранитель.

Хранитель — шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в это состояние. К примерам можно отнести сохранение в играх и редакторах.

UML диаграмма паттерна:



Originator хранит какое – нибудь состояние. Memento сохраняет *state*, а Caretaker оперирует хранителем.

Пример паттерна:

class Game{

private String level;

private int second;

public void set(String level, int second){

this.level = level;

this.second = second;

}

public void load(Save memento){

this.level = memento.getLevel ();

this.second = memento.getSecond ();

}

public Save save(){

return new Save(level, second);

}

@Override

public String toString() {

return level+", "+second;

}

}

class Save{

private final String level;

private final int second;

Save(String level, int second){

this.level = level;

this.second = second;

}

public int getSecond () {

return second;

}

public String getLevel () {

return level;

}

}

class File{

private Save gameSave;

public Save getSave() {

return gameSave;

}

public void setSave(Save gameSave) {

this.gameSave = gameSave;

}

}

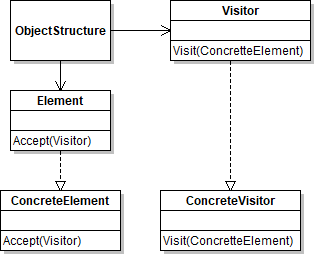
Использование:

File save = new File();  
Game game = new Game();  
  
game.set("LVL 1", 50);  
System.*out*.println(game);  
  
save.setSave (game.save ());  
game.set("LVL 2", 120);  
System.*out*.println(game);  
  
game.load (save.getSave ());  
System.*out*.println(game);

Посетитель.

Посетитель — шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов. При изменении visitor нет необходимости изменять обслуживаемые классы.

UML – диаграмма посетителя:



К примеру, есть машина. Она состоит из разных элементов, реализующий интерфейс ElementCare. В этом интерфейсе есть метод accept (), который позволяет посетителю залезть в машину.

Для посетителей есть интерфейс Visitor. У него есть метод accept (), который перегружается под каждый элемент. Конкретные реализации интерфейса определят свои действия для каждого элемента.

Пример паттерна:

interface Visitor{

void visit(Test test);

void visit(ProjectClass projectClass);

}

class Junior implements Visitor{

@Override

public void visit(Test test) {

System.out.println("Джуниор написал не надежный тест");

}

@Override

public void visit(ProjectClass projectClass) {

System.out.println("Джуниор написал класс средне");

}

}

class Senior implements Visitor{

@Override

public void visit(ProjectClass projectClass) {

System.out.println("Синьор написал правильно класс");

}

@Override

public void visit(Test test) {

System.out.println("Синьор написал правильный тест");

}

}

interface ProjectElement{

void deWritten(Visitor visitor);

}

class Test implements ProjectElement{

@Override

public void deWritten(Visitor visitor) {

visitor.visit (this);

}

}

class ProjectClass implements ProjectElement{

@Override

public void deWritten(Visitor visitor) {

visitor.visit (this);

}

}

class Project implements ProjectElement{

private ProjectElement [] elements = {

new Test(),

new ProjectClass()

};

@Override

public void deWritten(Visitor visitor) {

for (ProjectElement e: elements) {

e.deWritten (visitor);

}

}

}

Использование:

Project project = new Project ();

project.deWritten (new Junior ());

System.out.println ("\n=============" +

"=============\n");

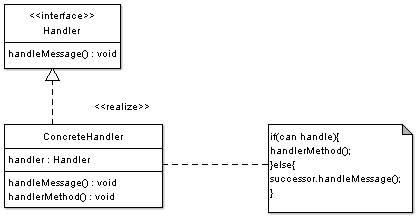
project.deWritten (new Senior ());

Цепочка обязанностей.

Цепочка обязанностей — шаблон проектирования, предназначенный для организации в системе уровней ответственности. Он избавляет от жёсткой привязки отправителя запроса к его получателю, позволяя выстраивать цепь из различных обработчиков динамически.

Она работает следующим образом. Есть некое сообщение или событие, которое должно обработать обработчик. Он может обработать его и отпустить, либо в противном случае передать его следующему обработчику. А тот же в свою очередь повторяет прошлый пункт.

UML диаграмма паттерна:



Чтоб понять эту диаграмма следует рассмотреть программу. Ее цель оповещать менеджера об известиях.

Есть абстрактный класс Notifier (*Handler*). Он определяет два поля. Первый это переменная priority, которая инициализируется в конструкторе и экземпляр Notifier – next, который служит следующим обработчиком и для него есть сеттер.

Его метод notifyManager () оповещает менеджера об событие. Ему передается *сообщение* и *level*. Если level <= priority, т.е. сообщение важно мы осуществится вызов метода write (*handleMessage* ()). Дальше если next не null то сообщение идет дальше по цепочке.

**Notifier:**

public abstract class Notifier {

private int priority;

private Notifier next;

Notifier(int priority){

this.priority = priority;

}

public void setNext(Notifier next) {

this.next = next;

}

public void notifyManager(String message, int level){

if(priority >= level)

write(message);

if(next != null)

next.notifyManager (message, level);

}

public abstract void write(String message);

}

Также есть у него есть 3 конкретные реализации. Первая это **SMSNotifier**:

public class SMSNotifier extends Notifier {

SMSNotifier (int priority) {

super (priority);

}

@Override

public void write(String message) {

System.out.println ("СМС сообщение: "+message);

}

}

Вторая это **EmailNotifier**:

public class EmailNotifier extends Notifier {

EmailNotifier (int priority) {

super (priority);

}

@Override

public void write(String message) {

System.out.println ("Email сообщение: "+message);

}

}

Третья это **ReportNotifier**:

public class ReportNotifier extends Notifier {

ReportNotifier (int priority){

super (priority);

}

@Override

public void write(String message) {

System.out.println ("Отчет: "+message);

}

}

Также есть класс, определяющий константы:

public class Priority {

public final static int

ROUTINE=1,

IMPORTANT = 2,

HOUSTON\_PROBLEM = 3;

}

Использование:

Notifier rn = new ReportNotifier (Priority.ROUTINE);

Notifier sms = new SMSNotifier (Priority.IMPORTANT);

Notifier emn = new EmailNotifier (Priority.HOUSTON\_PROBLEM);

rn.setNext (sms);

sms.setNext (emn);

rn.notifyManager ("Отчет Бла-Бла", Priority.ROUTINE);

sms.notifyManager ("…", Priority.IMPORTANT);

emn.notifyManager ("Тикай бочок потек", Priority.HOUSTON\_PROBLEM);

Паттерн Интерпретатор.

Интерпретатор — поведенческий шаблон проектирования, решающий часто встречающуюся, но подверженную изменениям, задачу.

Его преимуществом является то, что грамматику становится легко расширять и изменять, реализации классов, описывающих узлы абстрактного синтаксического дерева похожи (легко кодируются). Можно легко изменять способ вычисления выражений. Но при этом сопровождать обширную логику.

Пример создания интерпретатора способного считать числовые выражение:

interface Expression{

int interpret();

}

class Number implements Expression{

int number;

public Number(int number){

this.number = number;

}

@Override

public int interpret() {

return number;

}

}

class Plus implements Expression{

Expression left, right;

Plus(Expression left, Expression right){

this.left = left;

this.right = right;

}

@Override

public int interpret() {

return left.interpret() + right.interpret();

}

}

class Minus implements Expression{

Expression left, right;

Minus(Expression left, Expression right) {

this.left = left;

this.right = right;

}

@Override

public int interpret() {

return left.interpret () - right.interpret ();

}

}

class Context{

Expression evaluate (String s){

int pos = s.length() -1;

while (pos > 0){

if(Character.isDigit(s.charAt(pos)))

pos--;

else{

Expression left = evaluate (s.substring (0, pos));

Number right = new Number (

Integer.valueOf (

s.substring (pos + 1, s.length ())));

char operator = s.charAt(pos);

switch (operator){

case '+':

return new Plus(left, right);

case '-':

return new Minus(left, right);

default:

throw new IllegalArgumentException();

}

}

}

return new Number (Integer.valueOf (s));

}

}

Использование:

Context intr = new Context ();

Expression result = intr.evaluate ("2+2-2");

System.out.println (result.interpret ());