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

ОТЧЕТ

по учебно-технологической практике

Тема: Шаблоны проектирования (паттерны)

Студент гр. 9308 _____ Дементьев Д.П.

Преподаватель _____ Разумовский Г.В.

Санкт-Петербург
2021

Оглавление

Цель работы

Целью прохождения учебно-технологической практики является ознакомление с различными типами шаблонов проектирования(паттернов): порождающими, структурными и поведенческими; получение практических навыков в реализации некоторых паттернов на языке программирования Java.

Порождающие паттерны

Строитель (Builder)

Назначение: Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Когда использовать: Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации обычно спрятан внутри монструозного конструктора с десятком параметров, либо ещё хуже — распылён по всему клиентскому коду.

Пример: В качестве примера рассмотрим построение различных домов (класс House) с пошаговым возведением стен, дверей, крыши, окон, гаража и бассейна. Шаблон проектирования Строитель позволит нам задавать(добавлять) различные проекты домов лишь расширением имеющегося класса Director (некий «прораб» на стройке, задающий последовательность построения). Различные строители же будут реализовываться через общий интерфейс HouseBuilder, в приведённом примере будет лишь одна реализация данного интерфейса StandartHouseBuilder – построение обычных жилых домов. При желании можно реализовать и другие постройки, представляющие из себя различные здания. Диаграммы классов и последовательности представлены на Рис.1 и Рис.2 соответственно.

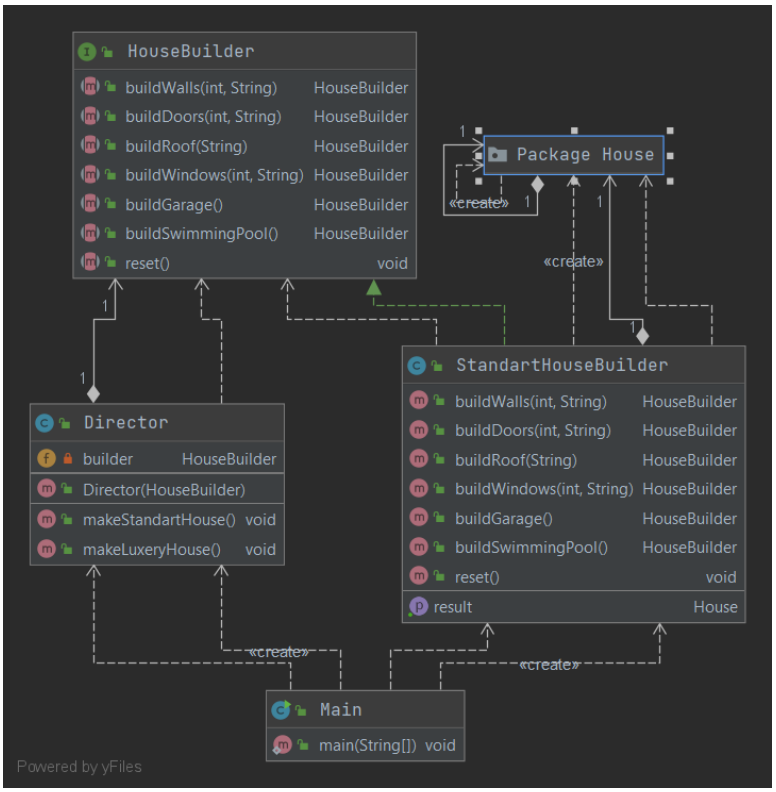


Рис.1. Диаграмма классов Builder

В диаграмме последовательности рассматривается лишь метод построения стандартного дома, т.к. для премиального дома все действия аналогичны.

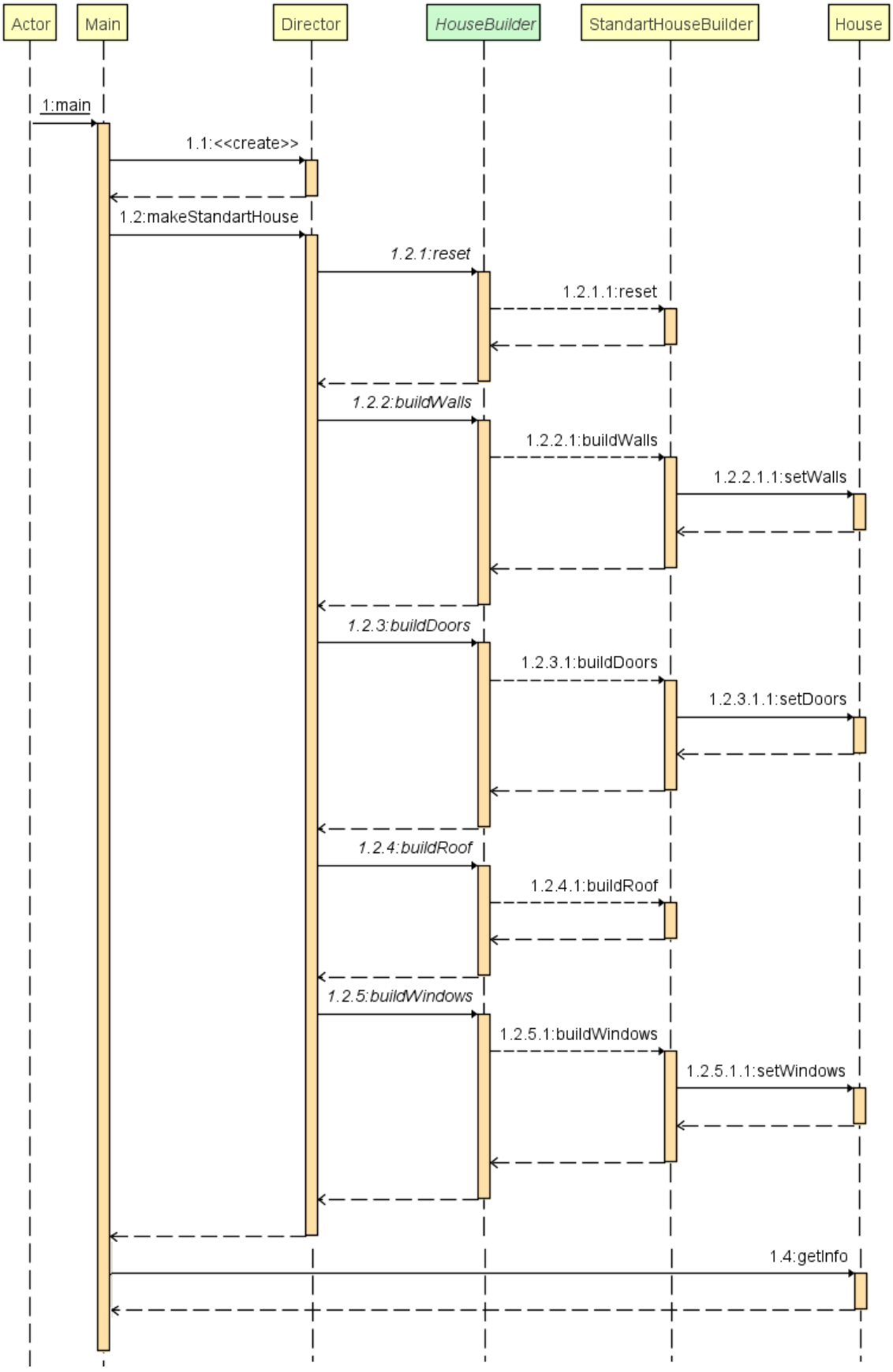


Рис.2. Диаграмма последовательности Builder

Код примера:

House.java (сущность жилого дома)

```
package House;

public class House {
    private Walls walls;
    private Doors doors;
    private Roof roof;
    private Windows windows;
    private boolean garage = false;
    private boolean swimmingPool = false;

    public void setWalls(int count, String material) {
        walls = new Walls(count, material);
    }
    public void setDoors(int count, String material) {
        doors = new Doors(count, material);
    }
    public void setWindows(int count, String material) {
        windows = new Windows(count, material);
    }
    public void setRoof(String material) {
        roof = new Roof(material);
    }
    public void addGarage() {
        garage = true;
    }
    public void addSwimmingPool() {
        swimmingPool = true;
    }
    /** Получение информации о доме **/
    public void getInfo() {
        System.out.println("House info:");
        System.out.println("  " + walls.getCount() + " walls made of " +
walls.getMaterial());
        System.out.println("  " + doors.getCount() + " doors made of " +
doors.getMaterial());
        System.out.println("  " + "roof made of " + roof.getMaterial());
        System.out.println("  " + windows.getCount() + " windows made of " +
windows.getMaterial());
        if (garage) System.out.println("  " + "with garage");
        if (swimmingPool) System.out.println("  " + "with swimming pool");
    }
}
```

HouseBuilder.java

```
/** Абстрактный класс Строителя Домов **/
public interface HouseBuilder {
    HouseBuilder buildWalls(int count, String material);
    HouseBuilder buildDoors(int count, String material);
    HouseBuilder buildRoof(String material);
    HouseBuilder buildWindows(int count, String material);
    HouseBuilder buildGarage();
    HouseBuilder buildSwimmingPool();
    void reset();
}
```

StandartHouseBuilder.java

```
import House.*;
/** Класс-строитель домов со стандартной планировкой **/
public class StandartHouseBuilder implements HouseBuilder {
    /** Поле для хранения строящегося дома **/
    private House house;

    @Override
    public HouseBuilder buildWalls(int count, String material) {
        if (house != null) {
            house.setWalls(count, material);
        }
        return this;
    }
    @Override
    public HouseBuilder buildDoors(int count, String material) {
        if (house != null) {
            house.setDoors(count, material);
        }
        return this;
    }
    @Override
    public HouseBuilder buildRoof(String material) {
        if (house != null) {
            house.setRoof(material);
        }
        return this;
    }
    @Override
    public HouseBuilder buildWindows(int count, String material) {
        if (house != null) {
            house.setWindows(count, material);
        }
        return this;
    }
    @Override
    public HouseBuilder buildGarage() {
        if (house != null) {
            house.addGarage();
        }
        return this;
    }
    @Override
    public HouseBuilder buildSwimmingPool() {
        if (house != null) {
            house.addSwimmingPool();
        }
        return this;
    }
    /** Строительство нового дома **/
    public void reset() {
        house = new House();
    }
    /** Получение построенного дома **/
    public House getResult() {
        return house;
    }
}
```

Director.java

```
public class Director {
    /** Поле, хранящее исполняющего проект строителя **/
    private HouseBuilder builder;
    /** создание "прораба" для выдачи указаний к постройке различных проектов
домов **/
    public Director(HouseBuilder _builder) {
        builder = _builder;
    }
    /** Стандартный деревянный дом **/
    public void makeStandartHouse() {
        builder.reset();
        builder.buildWalls(4, "Wood").buildDoors(1,
"Wood").buildRoof("Wood").buildWindows(2, "Glass");
    }
    /** Дом с премиальной отделкой, гаражом и бассейном **/
    public void makeLuxeryHouse() {
        builder.reset();
        builder.buildWalls(4, "Stone, Iron and Wood").buildDoors(3,
"Iron").buildRoof("Tile").buildWindows(4, "Premium
Glass").buildGarage().buildSwimmingPool();
    }
}
```

Результат работы:

Построение стандартного жилого дома

```
// создаём строителя стандартных планировок
StandartHouseBuilder builder = new StandartHouseBuilder();
// менеджер реализации различных шаблонов построения
Director director = new Director(builder);

// реализовать стандартный шаблон дома
director.makeStandartHouse();
builder.getResult().getInfo();
```

Полученный результат

```
House info:
  4 walls made of Wood
  1 doors made of Wood
  roof made of Wood
  2 windows made of Glass
```

Построение премиального жилого дома

```
// создаём строителя стандартных планировок
StandartHouseBuilder builder = new StandartHouseBuilder();
// менеджер реализации различных шаблонов построения
Director director = new Director(builder);

// реализовать премиальный шаблон дома
director.makeLuxeryHouse();
builder.getResult().getInfo();
```

Полученный результат

```
House info:
  4 walls made of Stone, Iron and Wood
  3 doors made of Iron
  roof made of Tile
  4 windows made of Premium Glass
  with garage
  with swimming pool
```

Таким образом, мы можем реализовывать и добавлять различные проекты домов лишь расширяя класс Director, задающий последовательные инструкции для реализации.

Фабричный метод (FactoryMethod)

Назначение: это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Когда использовать: когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код

Пример: в приведённом примере рассмотрим небольшую службу доставки грузов: на Грузовиках и на Баржах. В независимости от выбранного транспорта требуется следующий функционал: погрузка посылки на транспорт и доставка всего содержимого.
Диаграммы классов и последовательности представлены на Рис.3 и Рис.4 соответственно.

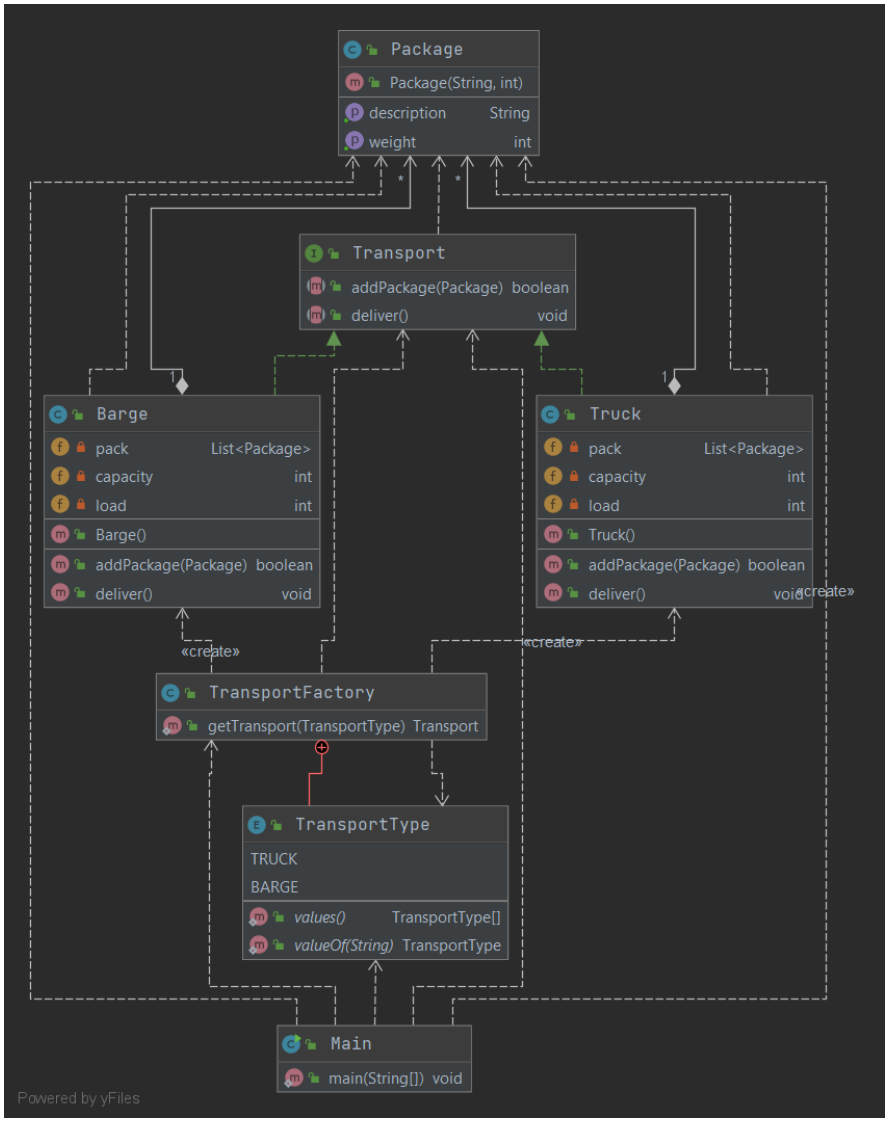


Рис.3. Диаграмма классов FactoryMethod

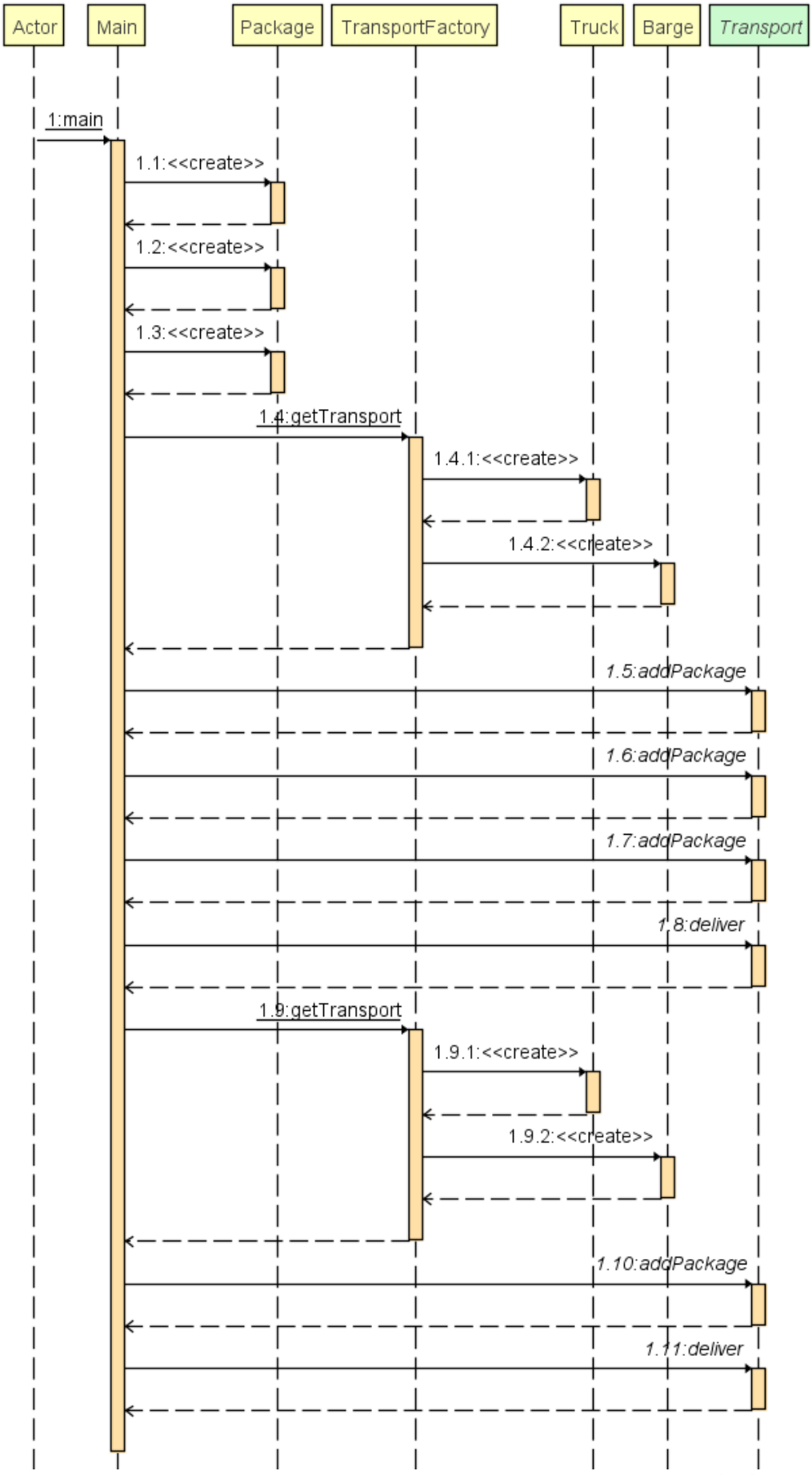


Рис.4. Диаграмма последовательности FactoryMethod

Код примера:

Package.java

```
package com.company;

public class Package {
    private String description;
    private int weight;

    public Package(String _description, int _weight) {
        description = _description;
        weight = _weight;
    }

    public String getDescription() {
        return description;
    }
    public int getWeight() {
        return weight;
    }
}
```

Transport.java

```
package com.company;

public interface Transport {
    /** Добавление посылки к перевозке **/
    boolean addPackage(Package pack);
    /** Доставка всего содержимого **/
    void deliver();
}
```

Truck.java

```
package com.company;

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

public class Truck implements Transport {
    /** перевозимый груз (посылки) **/
    private List<Package> pack;
    /** грузоподъемность в тоннах **/
    private int capacity;
    /** текущая загруженность **/
    private int load;

    public Truck() {
        pack = new ArrayList<>();
        capacity = 2000;
        load = 0;
    }

    @Override
    public boolean addPackage(Package pkg) {
        if (load + pkg.getWeight() > capacity) {
            return false;
        }
        else {
            pack.add(pkg);
            load += pkg.getWeight();
            return true;
        }
```

```
    }

    @Override
    public void deliver() {
        System.out.println("Truck successfully delivered:");
        for (Package p: pack) {
            System.out.println("  "+p.getDescription());
        }
    }
}
```

Barge.java

```
package com.company;

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

public class Barge implements Transport {
    /** перевозимый груз (посылки) **/
    private List<Package> pack;
    /** грузоподъемность в тоннах **/
    private int capacity;
    /** текущая загруженность **/
    private int load;

    public Barge() {
        pack = new ArrayList<>();
        capacity = 10000;
        load = 0;
    }

    @Override
    public boolean addPackage(Package pkg) {
        if (load + pkg.getWeight() > capacity) {
            return false;
        }
        else {
            pack.add(pkg);
            load += pkg.getWeight();
            return true;
        }
    }

    @Override
    public void deliver() {
        System.out.println("Barge successfully delivered:");
        for (Package p: pack) {
            System.out.println("  "+p.getDescription());
        }
    }
}
```

TransportFactory.java

```
package com.company;

public class TransportFactory
{
    /** Перечисление типов транспорта **/
    public enum TransportType
    {
        TRUCK, // Грузовик
        BARGE  // Баржа
```

```
    }

    /** Получение транспорта указанного типа **/
    public static Transport getTransport(TransportType TT)
    {
        Transport transport = null;
        switch(TT)
        {
            case TRUCK:
            {
                transport = new Truck();
                break;
            }
            case BARGE:
            {
                transport = new Barge();
                break;
            }
            default:
            {
                break;
            }
        }
        return transport;
    }
}
```

Результат работы:

Доставим следующие грузы:

```
Package coal = new Package("Coal", 1030),
        logs = new Package("Logs", 800),
        metals = new Package("Metals", 9999);
```

Попробуем загрузить все в новый Грузовик и произвести доставку:

```
Transport transport =
TransportFactory.getTransport(TransportFactory.TransportType.TRUCK);
transport.addPackage(coal);
transport.addPackage(logs);
transport.addPackage(metals);
transport.deliver();
```

Получим результат:

```
Truck successfully delivered:
  Coal
  Logs
```

Так как металлы слишком тяжелы для Грузовика, придется доставлять их при помощи Баржи:

```
transport = TransportFactory.getTransport(TransportFactory.TransportType.BARGE);
transport.addPackage(metals);
transport.deliver();
```

Результат второй доставки:

```
Barge successfully delivered:
  Metals
```

Таким образом, мы реализовали расширяемую службу доставки, т.к. мы можем без труда добавлять новый транспорт для доставок или же изменять специфику старых видов транспорта.

Прототип (Prototype)

Назначение: это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Когда использовать: когда ваш код не должен зависеть от классов копируемых объектов.

Пример: рассмотрим пример с геометрическими фигурами Прямоугольник и Круг, наследуемыми от общего родителя Фигура. Наша задача: реализовать функционал копирования объекта любой Фигуры и производных от нее (Прямоугольник и Круг)
Диаграммы классов и последовательности представлены на Рис.5 и Рис.6 соответственно.

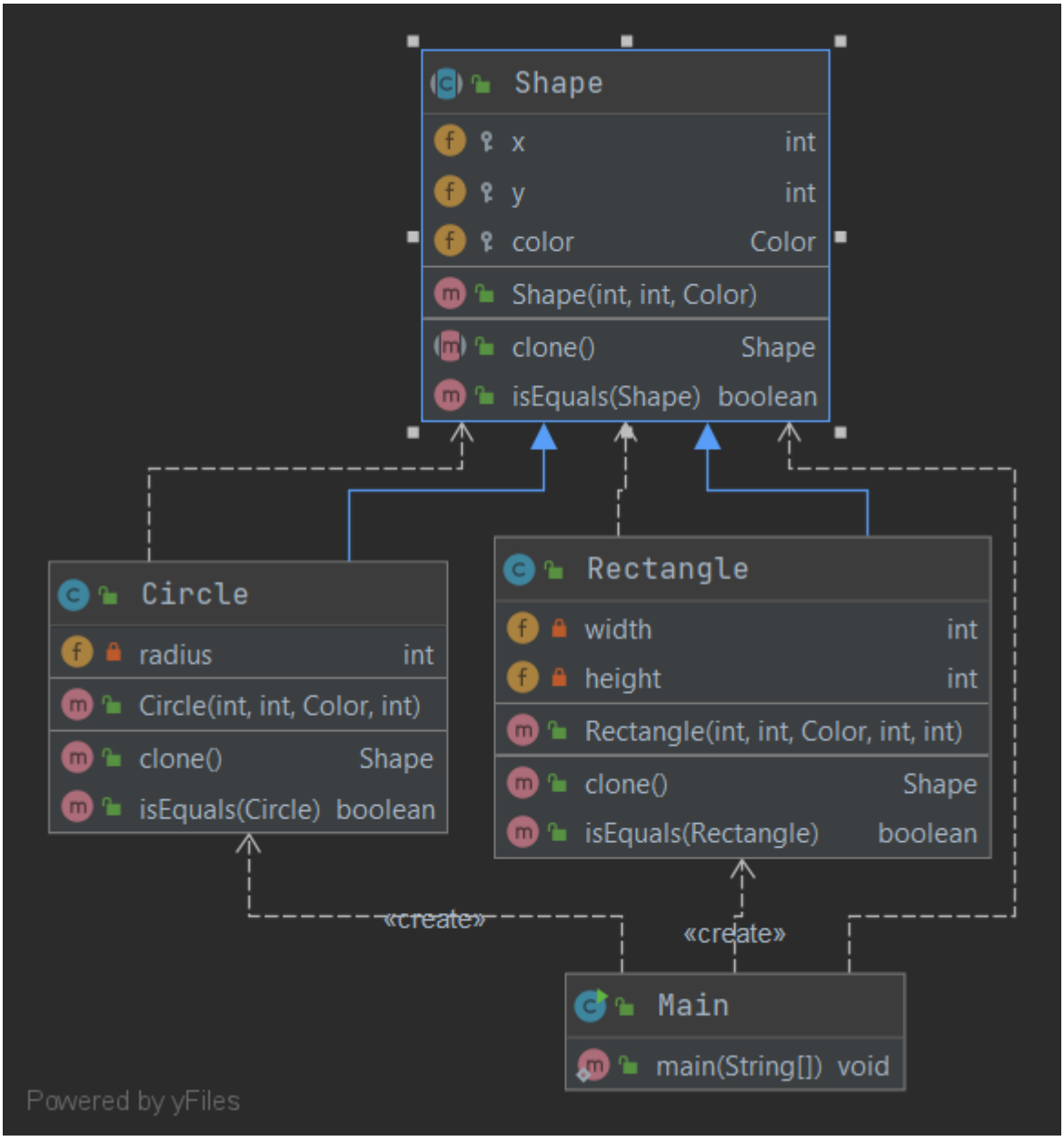


Рис.5. Диаграмма классов Prototype

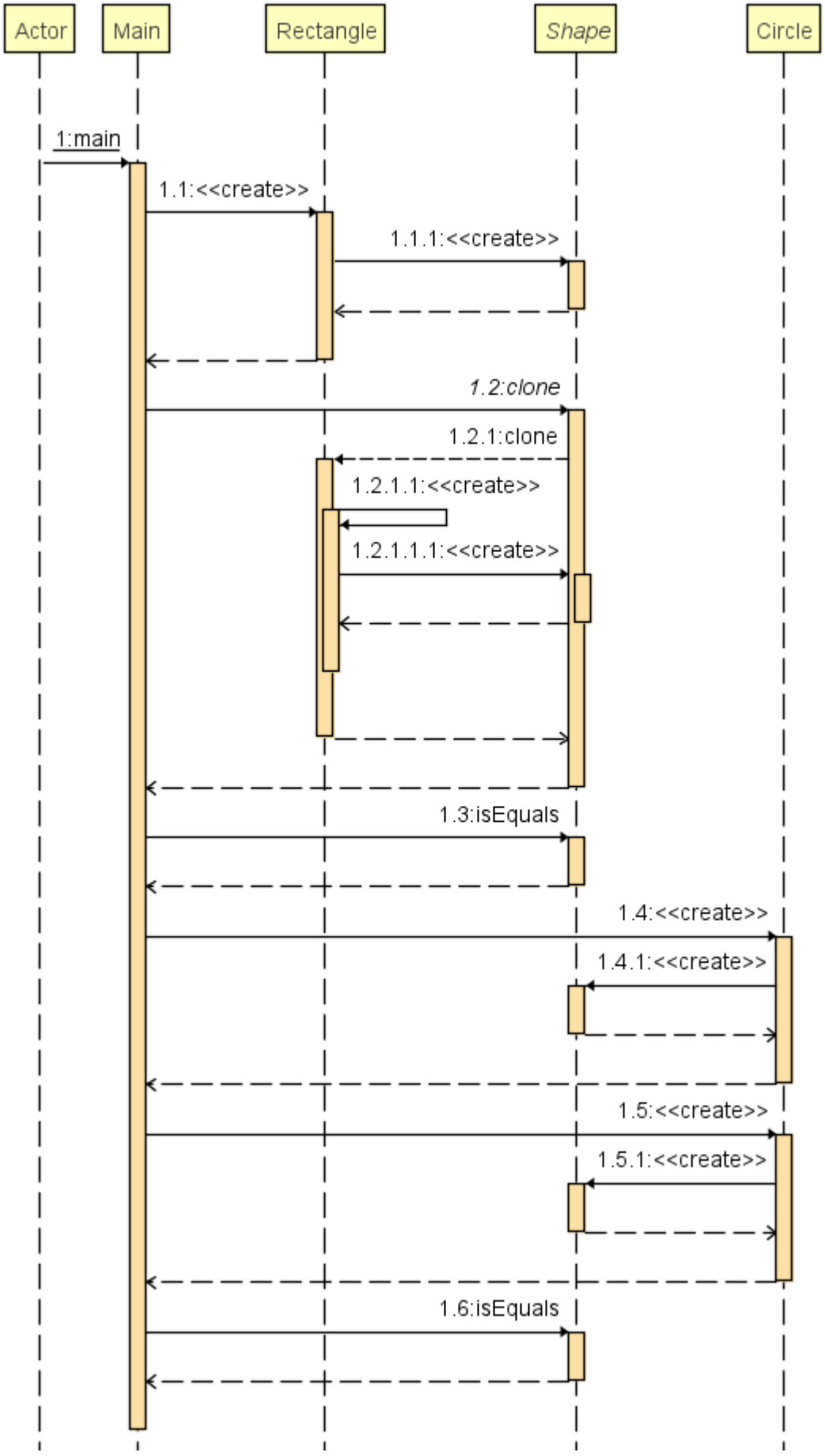


Рис.6. Диаграмма последовательности Prototype

Код примера:

Shape.java

```
package com.company;

import java.awt.*;

public abstract class Shape {
    /** Координаты центра фигуры **/
    protected int x, y;
    /** Цвет фигуры **/
    protected Color color;

    public Shape(int _x, int _y, Color _color) {
        x = _x;
        y = _y;
        color = _color;
    }

    public abstract Shape clone();
    public boolean isEquals(Shape other) {
        return x==other.x && y==other.y && color==other.color;
    }
}
```

Rectangle.java

```
package com.company;

import java.awt.*;

public class Rectangle extends Shape {
    /** Ширина **/
    private int width;
    /** Высота **/
    private int height;

    public Rectangle(int x, int y, Color color, int w, int h) {
        super(x, y, color);
        width = w;
        height = h;
    }

    @Override
    public Shape clone() {
        return new Rectangle(this.x, this.y, this.color, this.width,
this.height);
    }

    public boolean isEquals(Rectangle other) {
        return x==other.x && y==other.y && color==other.color &&
width==other.width && height==other.height;
    }
}
```

Circle.java

```
package com.company;

import java.awt.*;

public class Circle extends Shape {
    /** Радиус **/
    private int radius;

    public Circle(int x, int y, Color color, int r) {
        super(x, y, color);
        radius = r;
    }

    @Override
    public Shape clone() {
        return new Circle(this.x, this.y, this.color, this.radius);
    }

    public boolean isEquals(Circle other) {
        return x==other.x && y==other.y && color==other.color &&
radius==other.radius;
    }
}
```

Результат работы:

В демонстрации работы мы проверим, равны ли между собой оригинальный объект и его копия:

```
Shape rec_orig = new Rectangle(5, 10, Color.GREEN, 5, 5);
Shape rec_copy = rec_orig.clone();
System.out.println("Rectangle and his copy are " + (rec_orig.isEquals(rec_copy)
? "equals" : "not equals"));
```

Результат такой проверки говорит об их равенстве (что закономерно):

```
Rectangle and his copy are equals
```

Проверим корректность метода проверки на равенство:

```
Shape circle_1 = new Circle(1,1, Color.BLACK, 6),
       circle_2 = new Circle(1,1, Color.RED, 7);
System.out.println("Circle_1 and Circle_2 are " + (circle_1.isEquals(circle_2) ?
"equals" : "not equals"));
```

Видим, что для двух фигур с разными параметрами равенство не соблюдается, что доказывает корректность сравнения:

```
Circle_1 and Circle_2 are not equals
```

Таким образом, используя шаблон проектирования Прототип пользователь имеет возможность производить копирование геометрических фигур, не задумываясь о конкретных характеристиках фигуры.

Структурные паттерны

Мост (Bridge)

Назначение: отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо. При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию.

Когда использовать: когда вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, может работать с разными системами баз данных).

Пример: в этом примере Мост разделяет монолитный код приборов (Device) и пультов на две части: приборы (Device - выступают реализацией) и пульты управления ними (Remote - выступают абстракцией).
В качестве приборов рассматриваются Телевизор и Радио, в качестве пультов – двухкнопочный (позволяет изменять звук на +- 10% и переключать канал) и трёхкнопочный с функцией беззвучного режима.
Диаграммы классов и последовательности представлены на Рис.7 и Рис.8 соответственно.

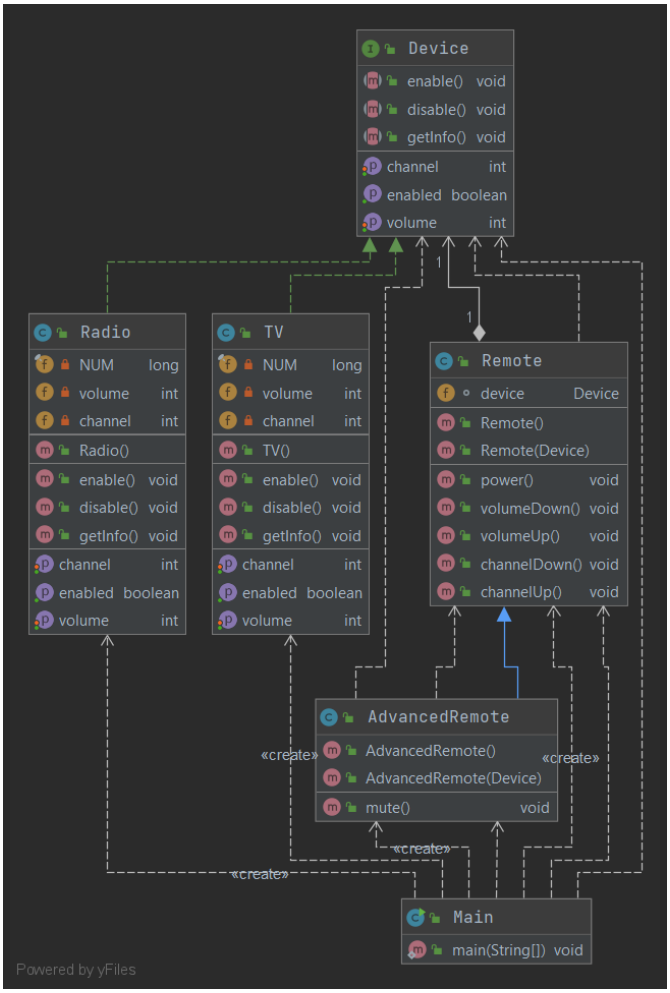


Рис.7. Диаграмма классов Bridge

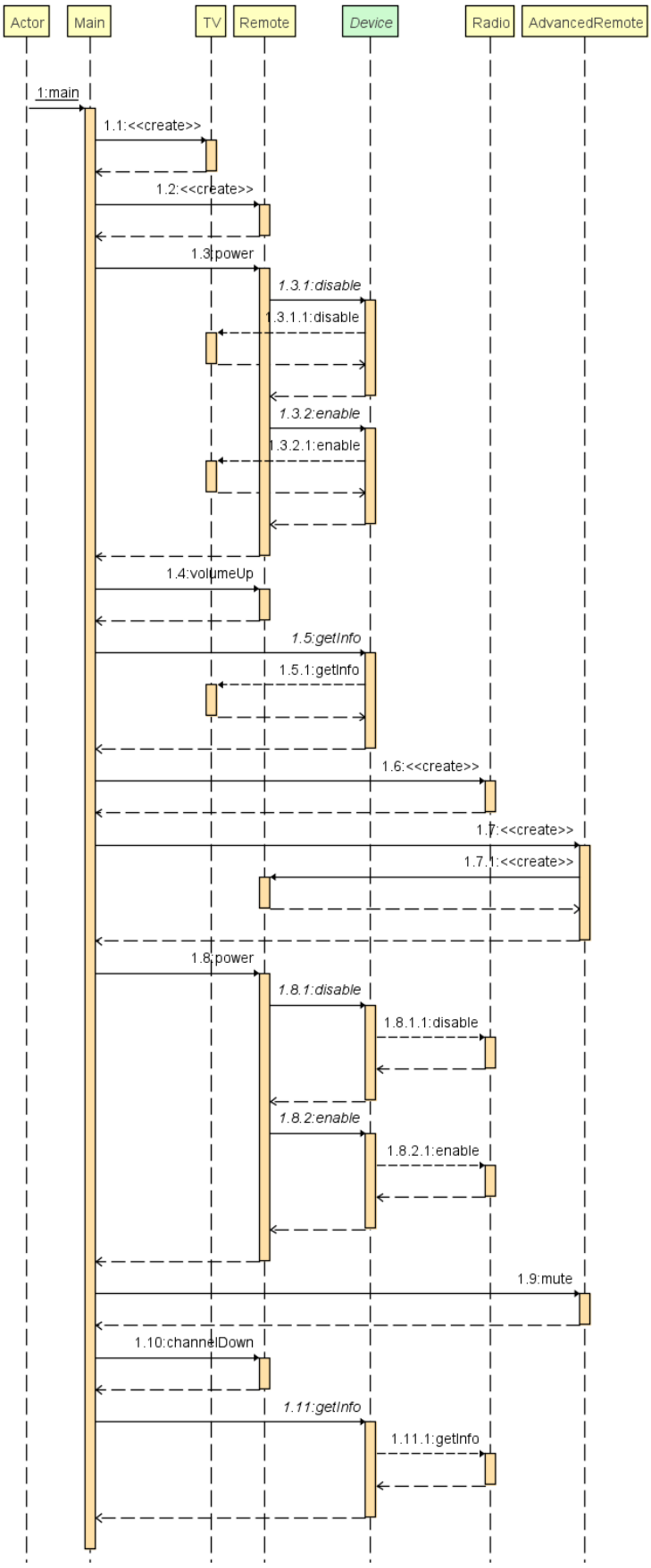


Рис.8. Диаграмма последовательности Bridge

Код примера:

Device.java

```
package com.company;

public interface Device {
    boolean isEnabled();
    void enable();
    void disable();
    int getVolume();
    void setVolume(int percent);
    int getChannel();
    void setChannel(int channel);
    void getInfo();
}
```

TV.java

```
package com.company;

import java.util.Date;

public class TV implements Device {
    /** Уникальный номер нового устройства (дата создания) **/
    private final long NUM;
    /** Включено/выключено **/
    private boolean enabled;
    /** Громкость в процентах **/
    private int volume;
    /** Номер канала **/
    private int channel;

    public TV() {
        Date date = new Date();
        NUM = date.getTime();
        enabled = false;
        volume = 25;
        channel = 1;
    }


    @Override
    public boolean isEnabled() {
        return enabled;
    }

    @Override
    public void enable() {
        enabled = true;
    }

    @Override
    public void disable() {
        enabled = false;
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int percent) {
        if (percent >= 0 && percent <= 100)
```

```
            volume = percent;
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int _channel) {
        if (channel > 0)
            channel = _channel;
    }

    public void getInfo() {
        System.out.println("TV №" + NUM + " info:");
        if (isEnabled()) System.out.println("  Is enabled");
        else System.out.println("  Is disabled");
        System.out.println("  Volume: " + volume + "%");
        System.out.println("  Channel: " + channel);
    }
}
```

Radio.java

```
package com.company;

import java.util.Date;

public class Radio implements Device {
    /** Уникальный номер нового устройства (дата создания) **/
    private final long NUM;
    /** Включено/выключено **/
    private boolean enabled;
    /** Громкость в процентах **/
    private int volume;
    /** Номер канала **/
    private int channel;

    public Radio() {
        Date date = new Date();
        NUM = date.getTime();
        enabled = false;
        volume = 25;
        channel = 1;
    }


    @Override
    public boolean isEnabled() {
        return enabled;
    }

    @Override
    public void enable() {
        enabled = true;
    }

    @Override
    public void disable() {
        enabled = false;
    }
```

```
    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int percent) {
        if (percent < 0) percent = 0;
        if (percent > 100) percent = 100;
        volume = percent;
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int _channel) {
        if (channel > 0)
            channel = _channel;
    }

    public void getInfo() {
        System.out.println("Radio №" + NUM + " info:");
        if (isEnabled()) System.out.println("  Is enabled");
        else System.out.println("  Is disabled");
        System.out.println("  Volume: " + volume + "%");
        System.out.println("  Channel: " + channel);
    }
}
```

Remote.java

```
package com.company;

public class Remote {
    Device device;

    public Remote() {
        device = null;
    }
    public Remote(Device _device) {
        device = _device;
    }

    public void power() {
        if (device != null)
            if (device.isEnabled())
                device.disable();
            else
                device.enable();
    }
    public void volumeDown() {
        if (device != null)
            if (device.isEnabled())
                device.setVolume(device.getVolume() - 10);
    }
    public void volumeUp() {
        if (device != null)
            if (device.isEnabled())
                device.setVolume(device.getVolume() + 10);
    }
```

```
    public void channelDown() {
        if (device != null)
            if (device.isEnabled())
                if (device.getChannel() == 1)
                    device.setChannel(99);
                else device.setChannel(device.getChannel() - 1);
    }
    public void channelUp() {
        if (device != null)
            if (device.isEnabled())
                if (device.getChannel() == 99)
                    device.setChannel(1);
                else device.setChannel(device.getChannel() + 1);
    }
}
```

AdvancedRemote.java

```
package com.company;

public class AdvancedRemote extends Remote {

    public AdvancedRemote() {
        super();
    }
    public AdvancedRemote(Device _device) {
        super(_device);
    }

    public void mute() {
        if (device.isEnabled()) device.setVolume(0);
    }
}
```

Результат работы:

Рассмотрим прибор Телевизор и двухкнопочный пульт управления к нему:

```
Device tv = new TV();
Remote remote = new Remote(tv);
remote.power();
remote.volumeUp();
```

При помощи пульта мы включаем телевизор и прибавляем звук, проверяем:

```
tv.getInfo();
```

По-умолчанию приборы выключены, их громкость составляет 25% и выбран первый канал. Видно, что пульт успешно выполнил свой функционал:

```
TV №1625510011249 info:
  Is enabled
  Volume: 35%
  Channel: 1
```

Теперь рассмотрим прибор Радио и возьмём трехнопочный пульт управления к нему:

```
Device radio = new Radio();
AdvancedRemote remote2 = new AdvancedRemote(radio);
remote2.power();
remote2.mute();
remote2.channelDown();
```

Аналогично проверяем состояние радио после проделанных действий:

```
radio.getInfo();
```

Получаем результат:

```
Radio №1625510011358 info:
  Is enabled
  Volume: 0%
  Channel: 99
```

Радио переведено в беззвучный режим и из-за отсутствия 1-1=0 канала выбран 99-ый.

Таким образом, пульты работают с приборами через общий интерфейс. Это даёт возможность связать пульты с различными приборами. Сами пульты можно развивать независимо от приборов. Для этого достаточно создать новый подкласс абстракции. Вы можете создать простой как простой пульт с двумя кнопками, так и более сложный пульт с тач-интерфейсом.

Декоратор (Decorator)

Назначение: это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Когда использовать: когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует; когда нельзя расширить обязанности объекта с помощью наследования.

Пример: в примере рассмотрим основной объект - обычный автомобиль SimpleCar с интерфейсом, общим для всех машин Car. Чтобы сделать из простого автомобиля скоростной спортивный автомобиль у нас есть класс-декоратор SportCarDecorator, который в конструкторе принимает класс SimpleCar и добавляет скорости обычному автомобилю. Аналогично для грузового автомобиля создан класс-декоратор TruckDecorator.
Диаграммы классов и последовательности представлены на Рис.9 и Рис.10 соответственно.

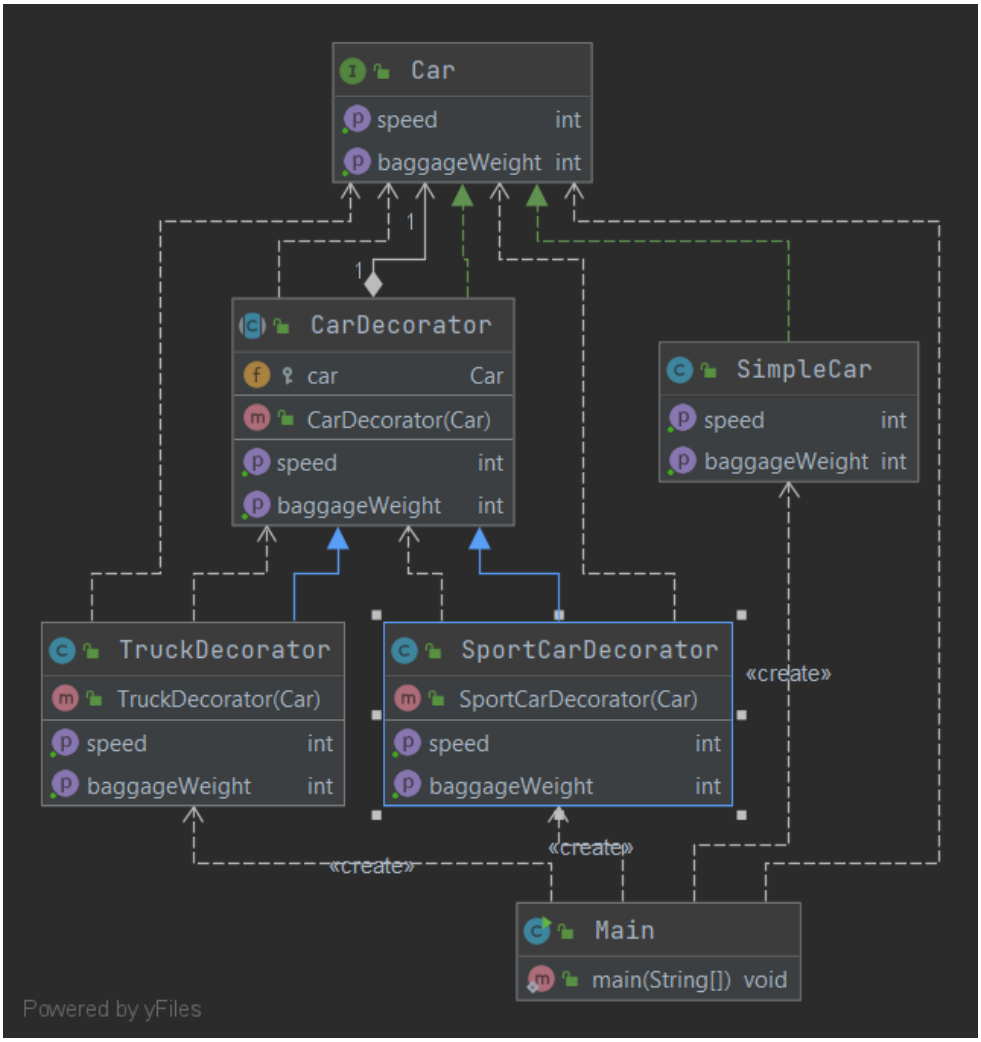


Рис.9. Диаграмма классов Decorator

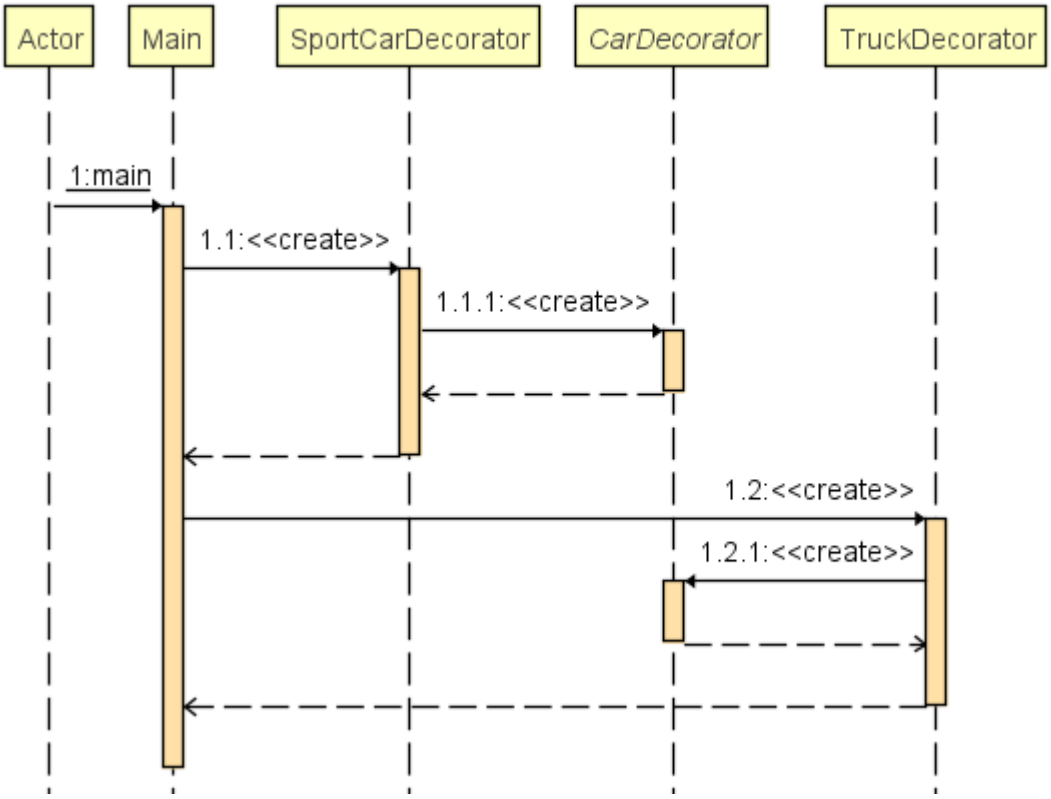


Рис.10. Диаграмма последовательности Decorator

Код примера:

Car.java

```
package com.company;
/** Интерфейс машины */
public interface Car {
    int getSpeed();
    int getBaggageWeight();
}
```

SimpleCar.java

```
package com.company;
/**
 * Класс стандартной машины (базовой)
 * Конкретный компонент
 */
public class SimpleCar implements Car {
    private int speed = 60;
    private int baggageWeight = 100;

    @Override
    public int getSpeed() {
        return this.speed;
    }

    @Override
    public int getBaggageWeight() {
        return this.baggageWeight;
    }
}
```

CarDecorator.java

```
package com.company;

/** Абстрактный класс декоратора машины */
public abstract class CarDecorator implements Car {
    protected Car car;
    public CarDecorator(Car car) {
        this.car = car;
    }
    public abstract int getSpeed();
    public abstract int getBaggageWeight();
}
```

SportCarDecorator.java

```
package com.company;
/** Класс-декоратор спортивной машины */
public class SportCarDecorator extends CarDecorator {
    public SportCarDecorator(Car car) {
        super(car);
    }

    @Override
    public int getSpeed() {
        return this.car.getSpeed() + 50;
    }
    @Override
    public int getBaggageWeight() {
        int newBaggageWeight = this.car.getBaggageWeight() - 30;
        return newBaggageWeight > 0 ? newBaggageWeight : 0;
    }
}
```

TruckDecorator.java

```
package com.company;
/** Класс-декоратор грузовой машины */
public class TruckDecorator extends CarDecorator {
    public TruckDecorator(Car car) {
        super(car);
    }

    @Override
    public int getSpeed() {
        return this.car.getSpeed();
    }

    @Override
    public int getBaggageWeight() {
        return this.car.getBaggageWeight() + 1000;
    }
}
```

Результат работы:
Для демонстрации работы создадим обычный автомобиль, затем улучшим его до спортивного автомобиля, а затем сделаем уже спортивное авто ещё и грузовым:

```
Car simpleCar = new SimpleCar();
Car sportCar = new SportCarDecorator(simpleCar);
Car sportTruck = new TruckDecorator(sportCar);
```

Получаем:

```
simpleCar: 60 speed and 100 weight (default)
sportCar: 110 speed and 70 weight (+50 speed, -30 weight)
sportTruck: 110 speed and 1070 weight (+1000 weight)
```

Можем видеть, что уровень вложенности «улучшений» неограничен.

Таким образом, мы реализовали шаблон проектирования Декоратор, который позволяет нам из обычного автомобиля сделать Спортивный или(и) Грузовой. Создание различных производных классов-декораторов позволят «улучшить» обычный автомобиль неограниченное количество раз.

Легковес (Flyweight)

Назначение: это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативной память за счёт экономного разделения общего состояния объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Когда использовать: когда не хватает оперативной памяти для поддержки всех нужных объектов.

- в приложении используется большое число объектов;
- из-за этого высоки расходы оперативной памяти;
- большую часть состояния объектов можно вынести за пределы их классов;
- многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено

Пример: В этом примере мы создадим и нарисуем лес (1.000.000 деревьев)! Каждому дереву соответствует свой объект, имеющий некоторое состояние (координаты, текстура и прочее). Такая программа хоть и работает, но потребляет слишком много памяти. Много деревьев имеют одинаковые свойства (название, текстуру, цвет). Потому мы можем применить паттерн Легковес и «закешировать» эти свойства в отдельных объектах TreeType. Теперь вместо хранения этих данных в миллионах объектов деревьев Tree, мы будем ссылаться на один из нескольких объектов-легковесов. Клиенту даже необязательно знать обо всём этом. Фабрика легковесов TreeType сама позаботится о создании нового типа дерева, если будет запрошено дерево с какими-то уникальными параметрами.
Диаграммы классов и последовательности представлены на Рис.11 и Рис.12 соответственно.

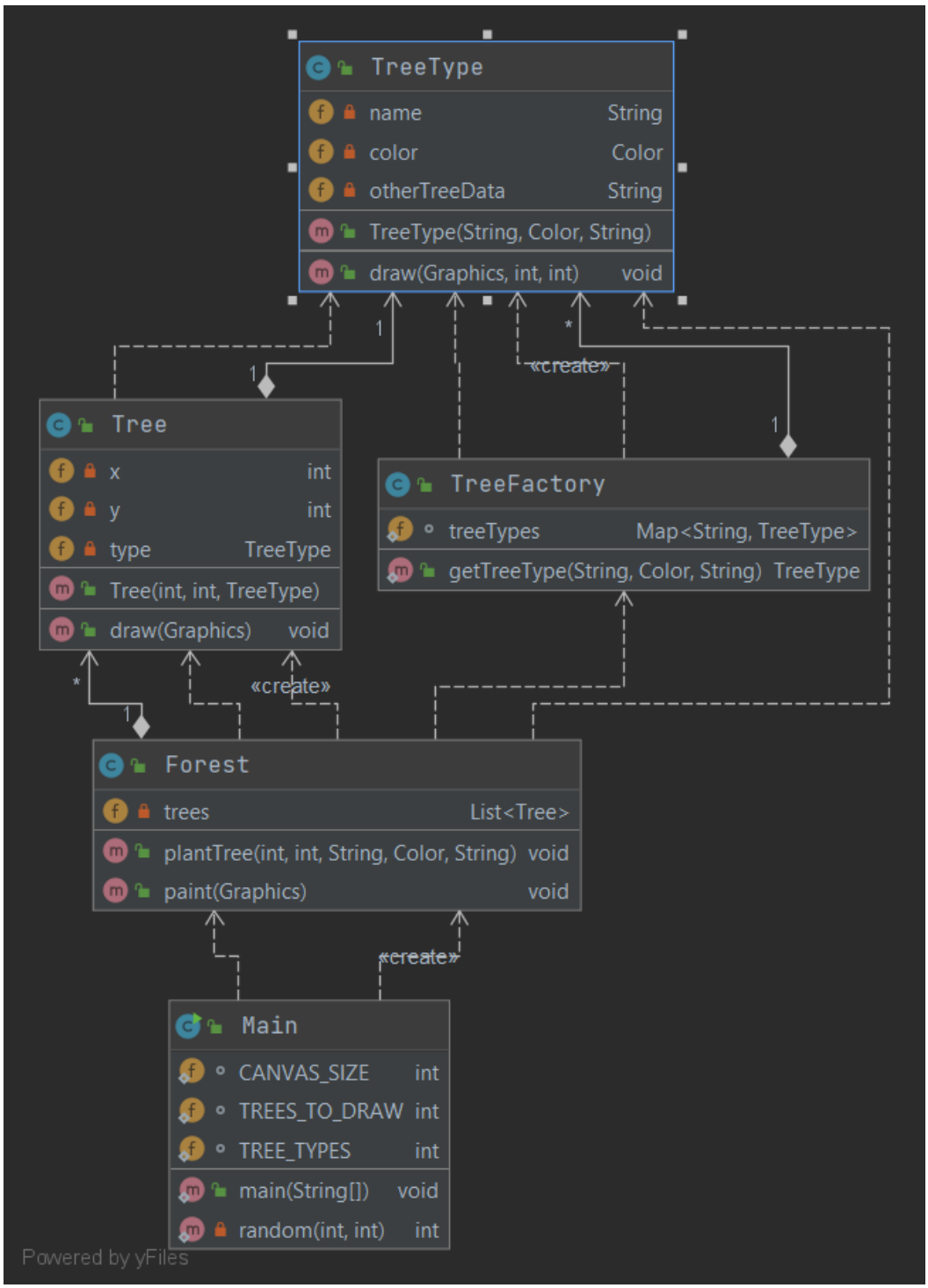


Рис.11. Диаграмма классов Flyweight

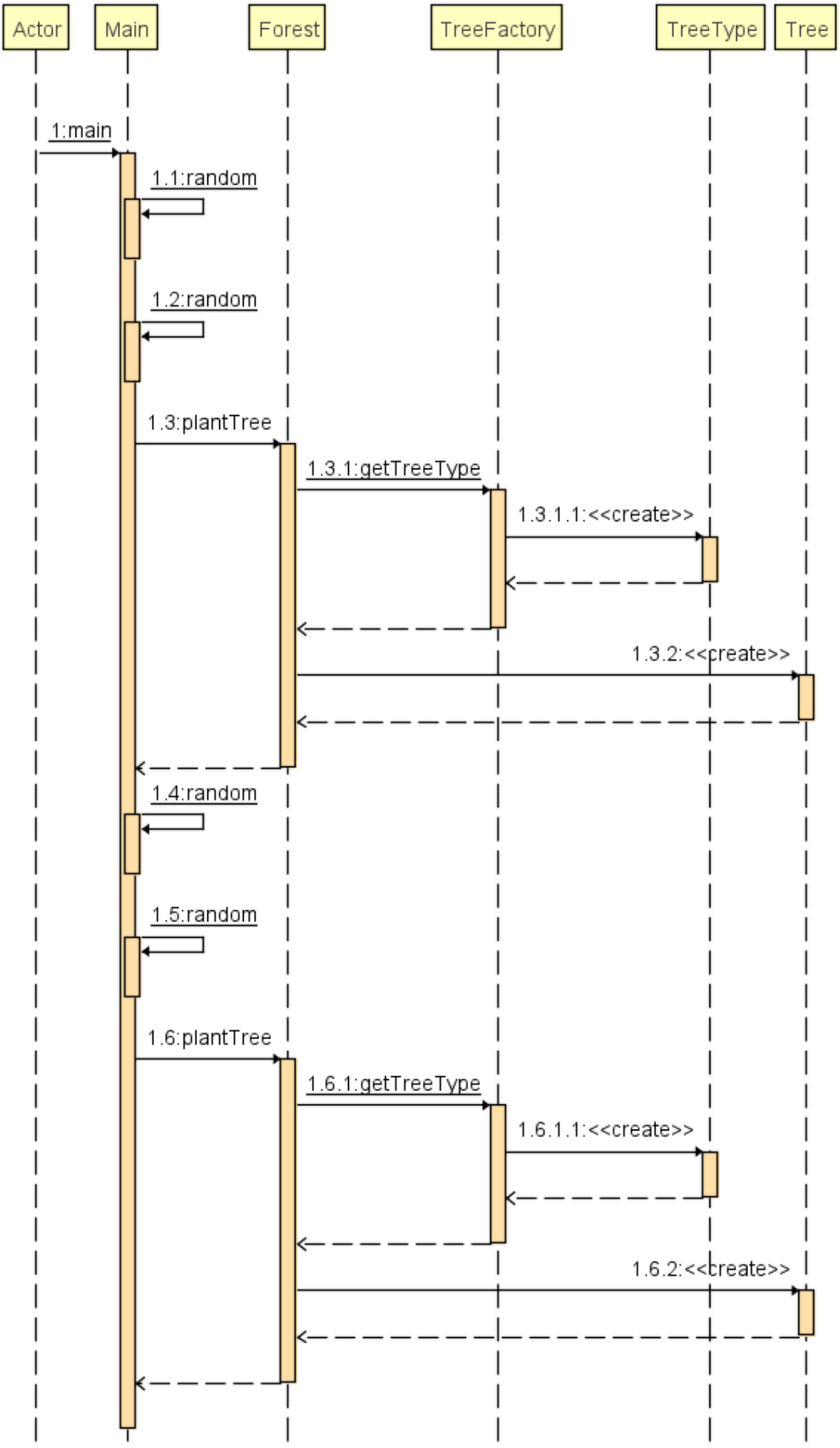


Рис.12. Диаграмма последовательности Flyweight

Код примера:

Tree.java

```
package com.company;

import java.awt.*;
/** Класс уникального дерева */
public class Tree {
    private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

TreeType.java

```
package com.company;

import java.awt.*;
/** Легковес, имеющий общее состояние нескольких деревьев */
public class TreeType {
    private String name;
    private Color color;
    private String otherTreeData;

    public TreeType(String name, Color color, String otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;
    }

    public void draw(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);
    }
}
```

TreeFactory.java

```
package com.company;

import java.awt.*;
import java.util.HashMap;
import java.util.Map;
/** Фабрика деревьев */
public class TreeFactory {
    static Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, Color color, String
otherTreeData) {
        TreeType result = treeTypes.get(name);
        if (result == null) {
            result = new TreeType(name, color, otherTreeData);
            treeTypes.put(name, result);
        }
        return result;
    }
}
```

Forest.java

```
package com.company;

import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;
/** GUI-лес, который рисует деревья */
public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();
    /** Высадить дерево */
    public void plantTree(int x, int y, String name, Color color, String
otherTreeData) {
        TreeType type = TreeFactory.getTreeType(name, color, otherTreeData);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }
    /** Отрисовка леса (Перегрузка метода библиотеки JFrame) */
    @Override
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }
    }
}
```

Main.java (рассматриваемый пример)

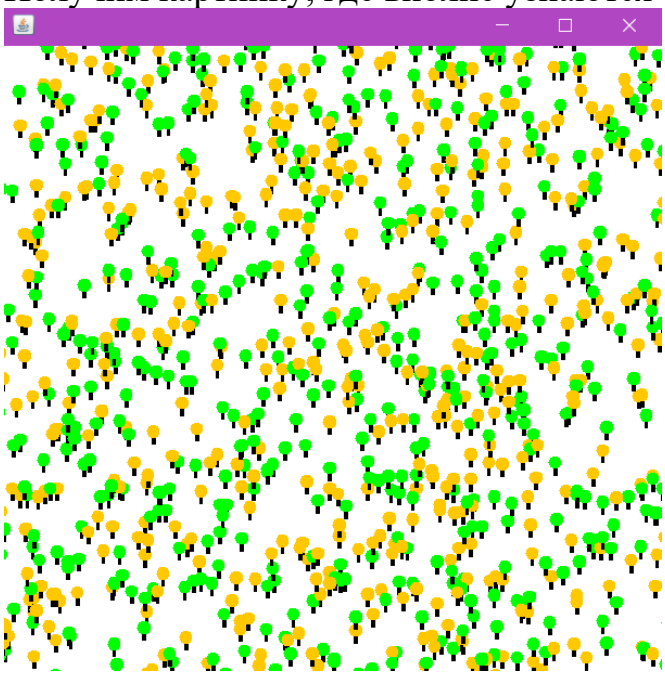
```
package com.company;

import java.awt.*;

public class Main {

    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 100000;
    static int TREE_TYPES = 2;
    ...
    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Summer Oak", Color.GREEN, "Oak texture stub");
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");
        }
        forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
        forest.setVisible(true);

        System.out.println(TREES_TO_DRAW + " trees drawn");
        System.out.println("---------------------");
        System.out.println("Memory usage:");
        System.out.println("Tree size (8 bytes) * " + TREES_TO_DRAW);
        System.out.println("+ TreeTypes size (~30 bytes) * " + TREE_TYPES + "");
        System.out.println("---------------------");
        System.out.println("Total: " + ((TREES_TO_DRAW * 8 + TREE_TYPES * 30) /
1024 / 1024) +
                "MB (instead of " + ((TREES_TO_DRAW * 38) / 1024 / 1024) +
"MB)");
    }

    private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));
    }

}
```

Результат работы:

Покажем корректность работы алгоритма создания и последующей отрисовки Леса из 1000 Деревьев двух типов (Летних и Осенних):

```
static int CANVAS_SIZE = 500;
static int TREES_TO_DRAW = 1000;
static int TREE_TYPES = 2;
...
Forest forest = new Forest();
for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
    forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
            "Summer Oak", Color.GREEN, "Oak texture stub");
    forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
            "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");
}
forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
forest.setVisible(true);
```

Получим картинку, где вполне узнаются силуэты деревьев:



Посчитаем, какую оптимизацию по оперативной памяти мы получили, используя Легковес даже на таком небольшом примере.

Размеры хранимых полей каждого из Деревьев:

Две координаты: 8 байт

Цвет, Название и Описание: примерно 30 байт – эти данные для деревьев одного Типа (в нашем случае имеются Летние и Осенние деревья) одинаковы, именно это будет являться Легковесом

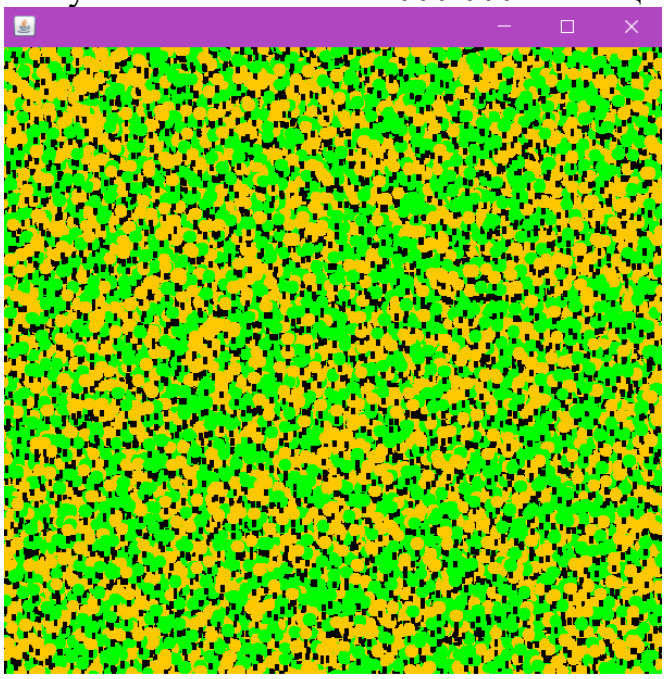
- Если бы каждое дерево хранило свои Координаты, Цвет, Название и Описание:
  ```
  (1000 * 38) / 1024 = 37 KB
  ```
- При использовании легковеса:
  ```
  (1000 * 8 + 2 * 30) / 1024 = 8 KB
  ```

Даже на таком небольшом количестве деревьев разница является кратной.

Для «красоты» приведу пример с 1.000.000 деревьев:

- Если бы каждое дерево хранило свои Координаты, Цвет, Название и Описание:
```
(1000000 * 38) / 1024 / 1024 = 36 MB
```
- При использовании легковеса:
```
(1000000 * 8 + 2 * 30) / 1024 / 1024 = 7 MB
```

Получившийся Лес из 1.000.000 саженцев:



Таким образом, очевидно, что использование Легковеса значительно улучшило оптимизацию содания и последующей отрисовке Леса.

Поведенческие паттерны

Цепочка команд (Chain of Command)

Назначение: это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Когда использовать: когда возможно разделить большое действие на последовательность запросов и обработок, а также количество этих запросов со временем может увеличиваться.

Пример: в приведённом примере мы рассмотрим работу банкомата по выдаче сумм, кратных 10. Цепочка команд будет представлять из себя: Проверка кратности -> Выдача банкнот по 50$ -> Выдача банкнот по 20$ -> Выдача банкнот по 10$. На каждом этапе в случае успешного прохождения проверки на кратность будет выдаваться максимально возможная сумма сначала из 50$ банкнот, затем из 20$ и в конце из 10$.
Диаграммы классов и последовательности представлены на Рис.13 и Рис.14 соответственно.

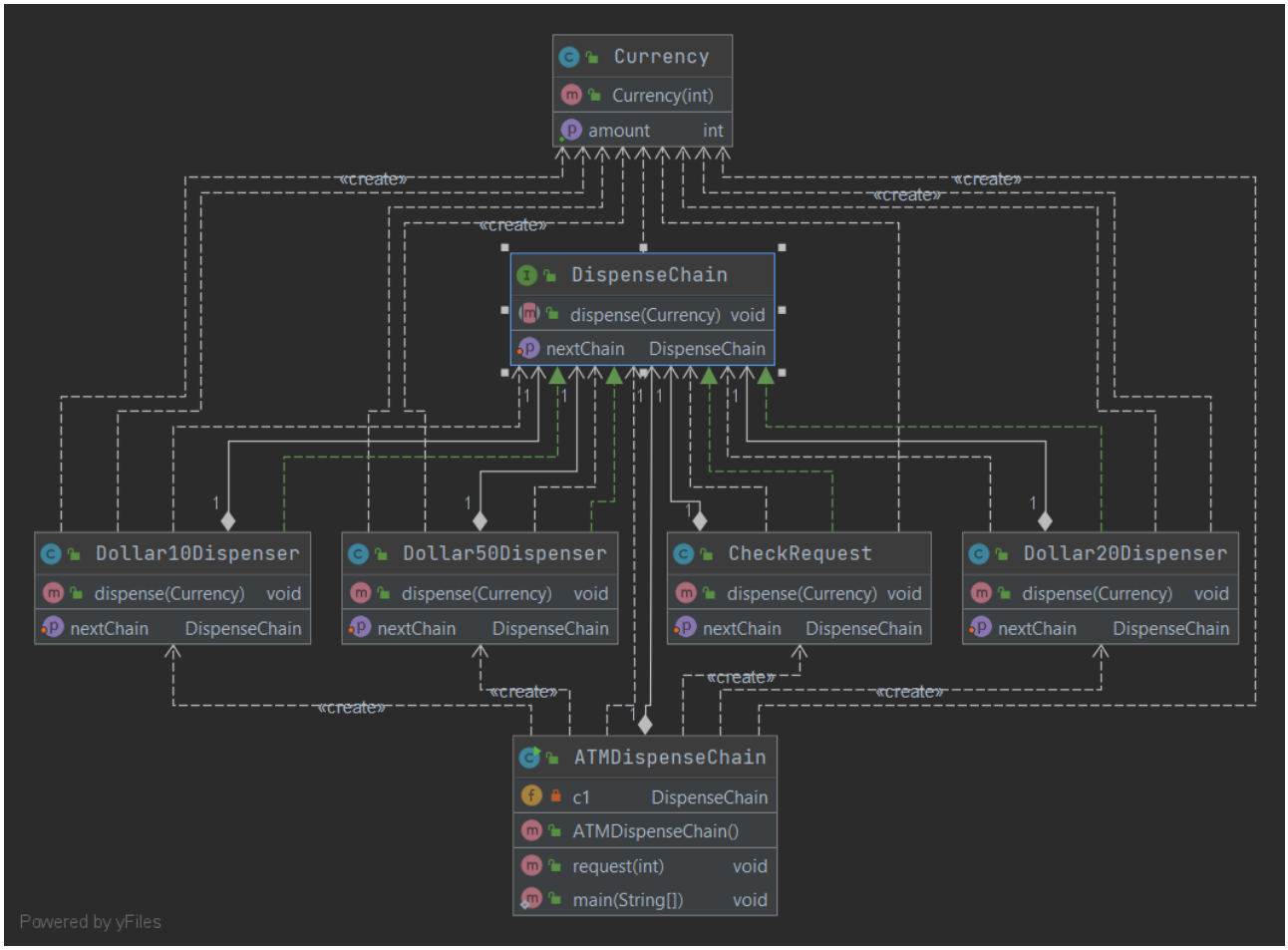


Рис.13. Диаграмма классов ChainOfCommand

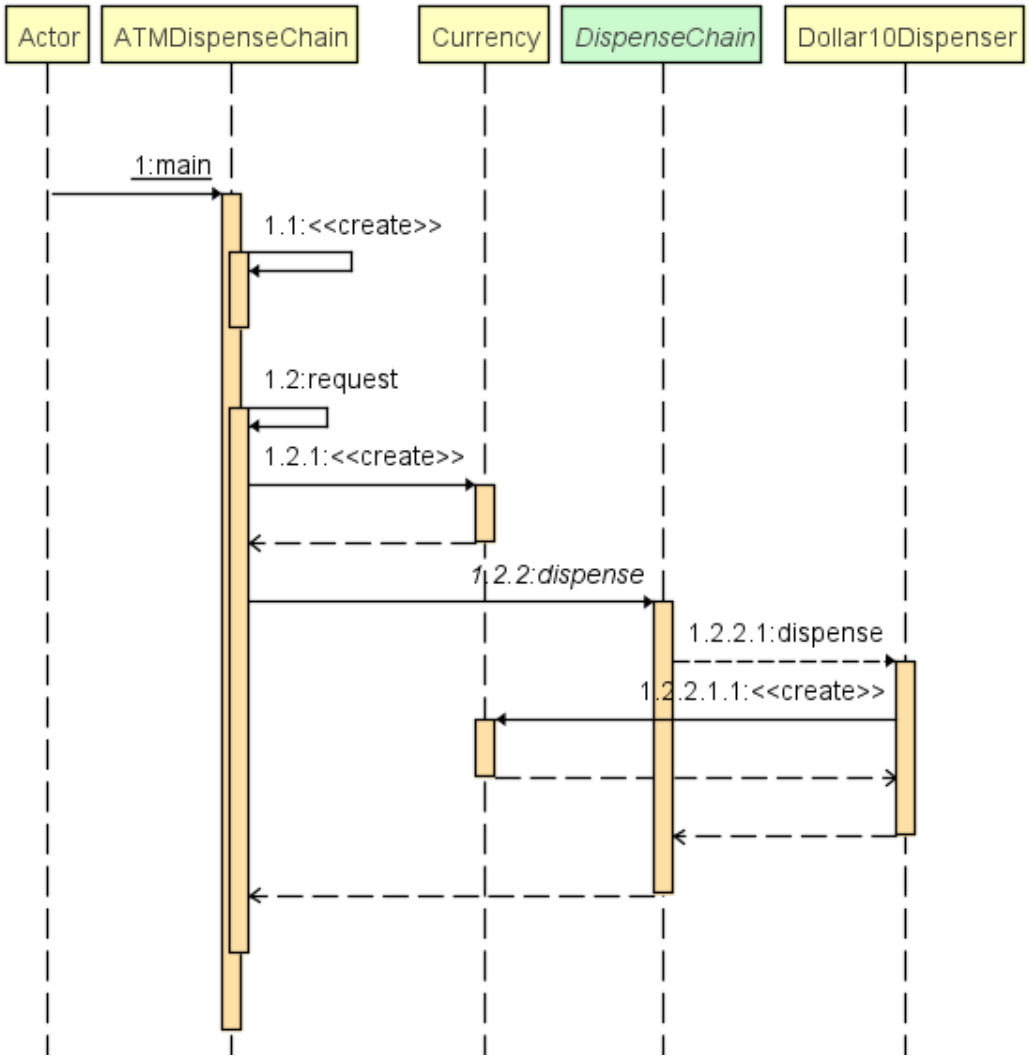


Рис.14. Диаграмма последовательности ChainOfCommand

Код примера:

Currency.java

```
package com.company;
public class Currency {
    private int amount;

    public Currency(int amt){
        this.amount=amt;
    }

    public int getAmount(){
        return this.amount;
    }
}
```

DispenseChain.java

```
package com.company;
public interface DispenseChain {

    void setNextChain(DispenseChain nextChain);

    void dispense(Currency cur);
}
```

CheckRequest.java

```
package com.company;

public class CheckRequest implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }
    @Override
    public void dispense(Currency cur) {
        if (cur.getAmount() % 10 != 0) {
            System.out.println("Error! Amount should be in multiple of 10s.");
        }
        else {
            this.chain.dispense(cur);
        }
    }
}
```

Dollar10Despenser.java

```
package com.company;

public class Dollar10Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 10){
            int num = cur.getAmount()/10;
            int remainder = cur.getAmount() % 10;
            System.out.println("Dispensing "+num+" 10$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }

}
```

Dollar20Despenser.java

```
package com.company;

public class Dollar20Dispenser implements DispenseChain{

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }
```

```
    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 20){
            int num = cur.getAmount()/20;
            int remainder = cur.getAmount() % 20;
            System.out.println("Dispensing "+num+" 20$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }

}
```

Dollar50Despenser.java

```
package com.company;

public class Dollar50Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 50){
            int num = cur.getAmount()/50;
            int remainder = cur.getAmount() % 50;
            System.out.println("Dispensing "+num+" 50$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }

}
```

ATMDespenserChain.java

```
package com.company;

import java.util.Scanner;

public class ATMDispenseChain {

    private DispenseChain c1;

    public ATMDispenseChain() {
        // инициализация цепочки
        this.c1 = new CheckRequest();
        DispenseChain c2 = new Dollar50Dispenser();
        DispenseChain c3 = new Dollar20Dispenser();
        DispenseChain c4 = new Dollar10Dispenser();

        // последовательность вызова
        c1.setNextChain(c2);
        c2.setNextChain(c3);
        c3.setNextChain(c4);
    }
```

```
    public void request(int amount) {
        this.c1.dispense(new Currency(amount));
    }

    public static void main(String[] args) {
        ATMDispenseChain atmDispenser = new ATMDispenseChain();
        while (true) {
            int amount = 0;
            System.out.println("Enter amount to dispense");
            Scanner input = new Scanner(System.in);
            amount = input.nextInt();
            // запрос выдачи денег
            atmDispenser.request(amount);
        }

    }

}
```

Результат работы:
Сперва проверим проверку кратности

```
Enter amount to dispense
123
Error! Amount should be in multiple of 10s.
```

Отлично, не пройдя проверку кратности, выполнение цепочки прерывается

Теперь проверим корректность выдачи банкнот всех видов

```
Enter amount to dispense
430
Dispensing 8 50$ note
Dispensing 1 20$ note
Dispensing 1 10$ note
```

Также выдача без использования всех банкнот

```
Enter amount to dispense
200
Dispensing 4 50$ note
```

И такой вариант

```
Enter amount to dispense
210
Dispensing 4 50$ note
Dispensing 1 10$ note
```

Таким образом, наш банкомат работает корректно. При необходимости количество проверок в любое время можно увеличить, например добавить проверку на наличие требуемых банкнот в банкомате. В этом и заключается преимущество шаблона Цепочка команд.

Команда (Command)

Назначение: это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Когда использовать: когда вы хотите параметризовать объекты выполняемым действием; когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети; когда вам нужна операция отмены.

Пример: в примере по реакции на действия пользователя, текстовый редактор создаёт объекты команд. Каждая команда выполняет некоторое действие, а затем попадают в стек истории. Теперь, чтобы выполнить отмену, мы берём последнюю команду из списка и выполняем обратное действие либо восстанавливаем состояние редактора, сохранённое в этой команде.
Диаграммы классов и последовательности представлены на Рис.15 и Рис.16 соответственно.

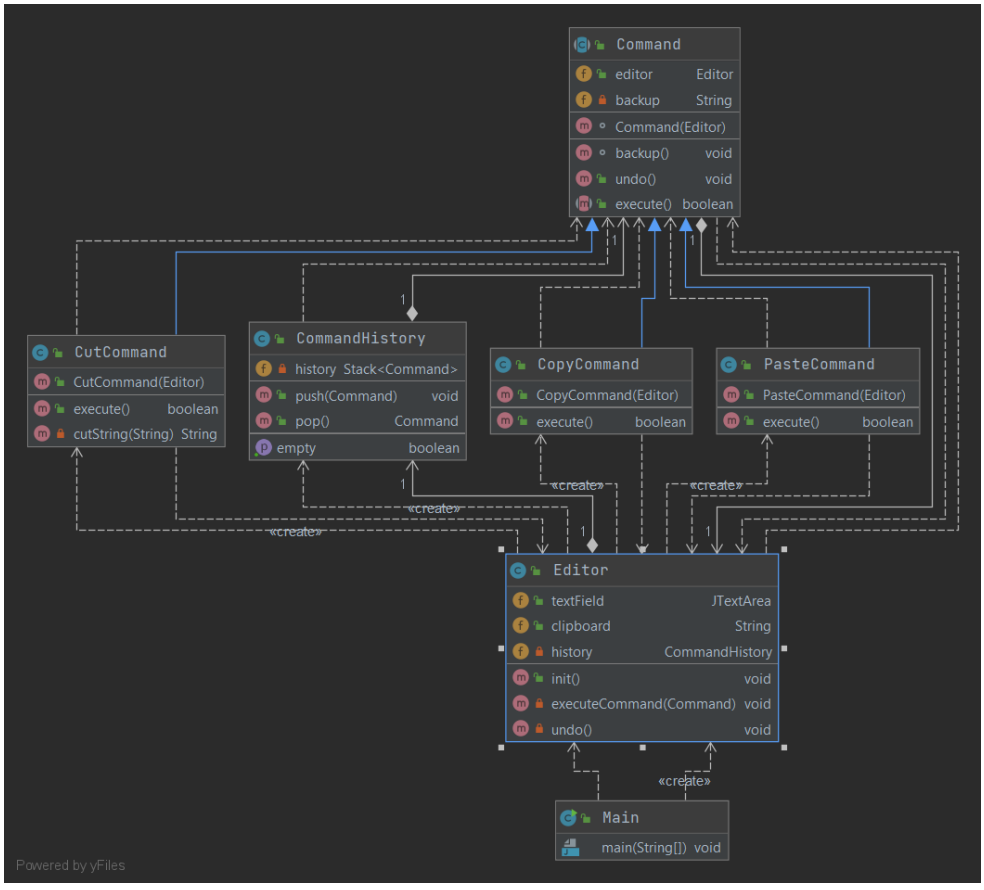


Рис.15. Диаграмма классов Command

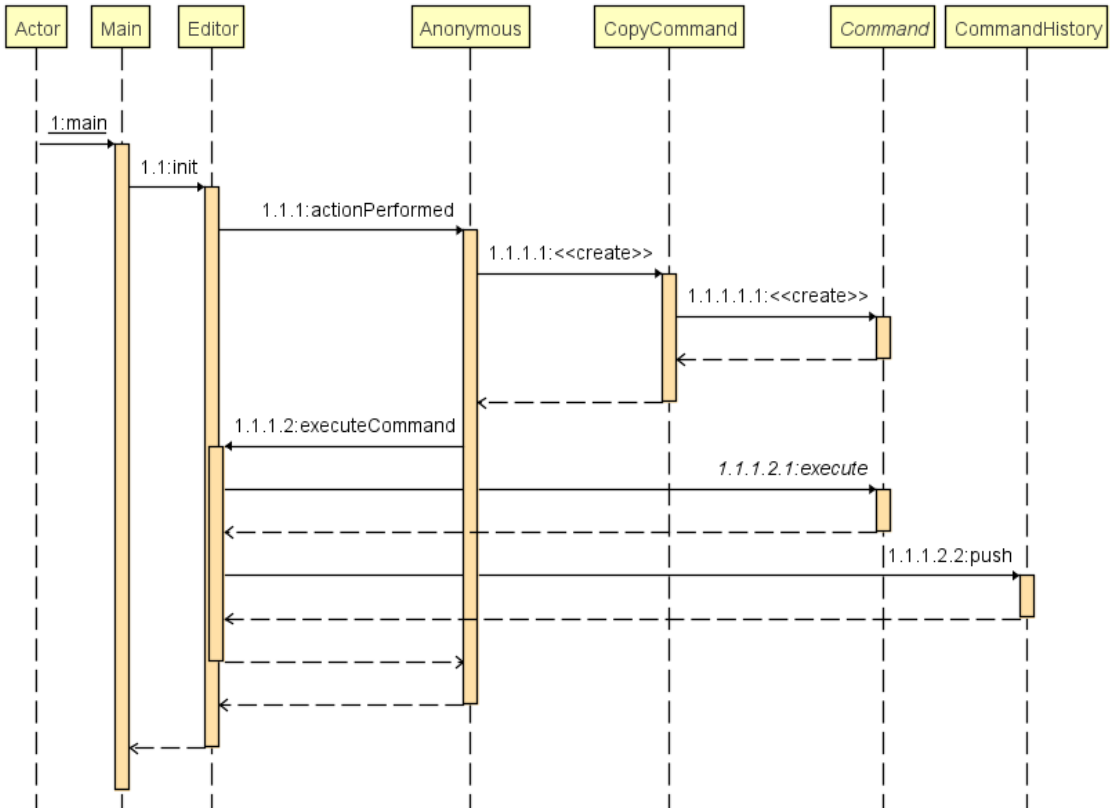


Рис.16. Диаграмма последовательности Command

Код примера:

Command.java

```
package com.company;
/** Абстрактная базовая комманда */
public abstract class Command {
    protected Editor editor;
    private String backup;

    Command(Editor editor) {
        this.editor = editor;
    }
    /** Создание резервной копии */
    void backup() {
        backup = editor.textField.getText();
    }
    /** Отмена команды */
    public void undo() {
        editor.textField.setText(backup);
    }
    /** Применение команды */
    public abstract boolean execute();
}
```

CopyCommand.java

```
package com.company;

/** Команда копирования */
public class CopyCommand extends Command {

    public CopyCommand(Editor editor) {
        super(editor);
    }
```

```
    @Override
    public boolean execute() {
        editor.clipboard = editor.textField.getSelectedText();
        return false;
    }
}
```

PasteCommand.java

```
package com.company;

/** Команда вставки */
public class PasteCommand extends Command {

    public PasteCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        if (editor.clipboard == null || editor.clipboard.isEmpty()) return
false;

        backup();
        editor.textField.insert(editor.clipboard,
editor.textField.getCaretPosition());
        return true;
    }
}
```

CutCommand.java

```
package com.company;

/** Команда вырезания */
public class CutCommand extends Command {

    public CutCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        if (editor.textField.getSelectedText().isEmpty()) return false;

        backup();
        String source = editor.textField.getText();
        editor.clipboard = editor.textField.getSelectedText();
        editor.textField.setText(cutString(source));
        return true;
    }

    private String cutString(String source) {
        String start = source.substring(0,
editor.textField.getSelectionStart());
        String end = source.substring(editor.textField.getSelectionEnd());
        return start + end;
    }
}
```

CommandHistory.java

```java
package com.company;

import java.util.Stack;
/** История команд */
public class CommandHistory {
    private Stack<Command> history = new Stack<>();

    public void push(Command c) {
        history.push(c);
    }

    public Command pop() {
        return history.pop();
    }

    public boolean isEmpty() { return history.isEmpty(); }
}
```

Editor.java

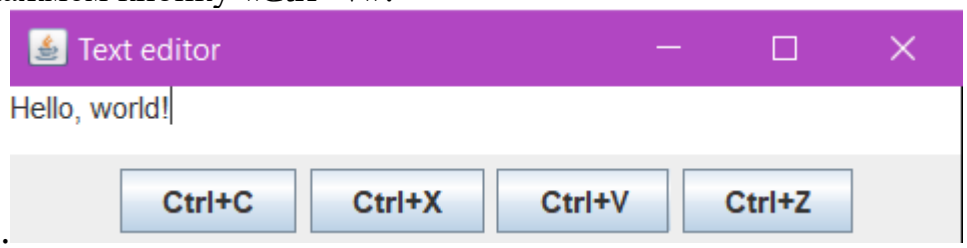
```java
package com.company;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
/** Оболочка текстового редактора */
public class Editor {
    public JTextArea textField;
    public String clipboard;
    private CommandHistory history = new CommandHistory();

    public void init() {
        JFrame frame = new JFrame("Text editor");
        JPanel content = new JPanel();
        frame.setContentPane(content);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
        textField = new JTextArea();
        textField.setLineWrap(true);
        content.add(textField);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.CENTER));
        JButton ctrlC = new JButton("Ctrl+C");
        JButton ctrlX = new JButton("Ctrl+X");
        JButton ctrlV = new JButton("Ctrl+V");
        JButton ctrlZ = new JButton("Ctrl+Z");
        Editor editor = this;
        ctrlC.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new CopyCommand(editor));
            }
        });
        ctrlX.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new CutCommand(editor));
            }
        });
        ctrlV.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new PasteCommand(editor));
            }
```

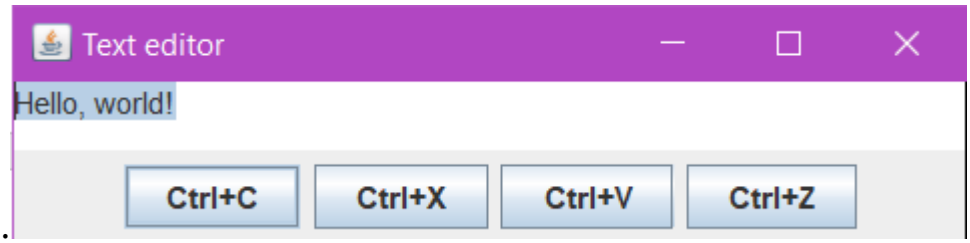
```
        });
        ctrlZ.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                undo();
            }
        });
        buttons.add(ctrlC);
        buttons.add(ctrlX);
        buttons.add(ctrlV);
        buttons.add(ctrlZ);
        content.add(buttons);
        frame.setSize(450, 200);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private void executeCommand(Command command) {
        if (command.execute()) {
            history.push(command);
        }
    }

    private void undo() {
        if (history.isEmpty()) return;

        Command command = history.pop();
        if (command != null) {
            command.undo();
        }
    }
}
```

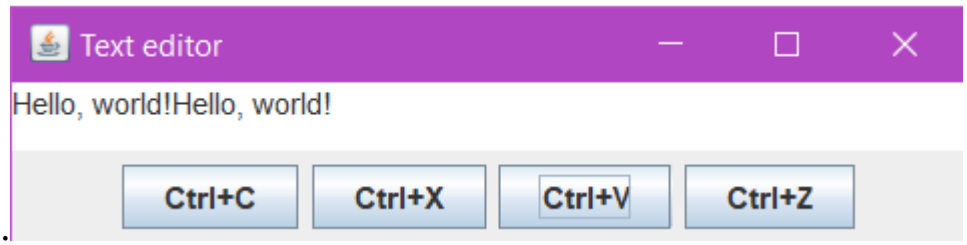
Main.java

```
package com.company;

public class Main {

    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.init();
    }
}
```

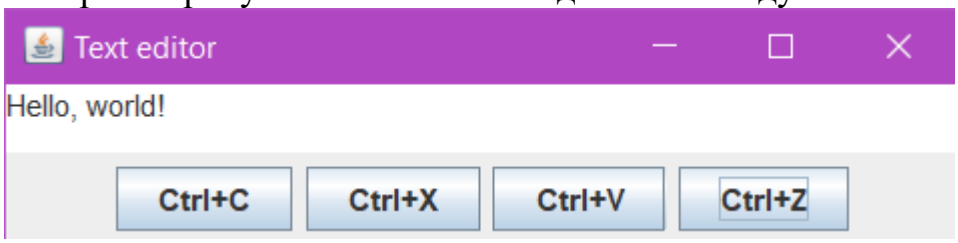
Результат работы:
Покажем работоспособность исполняемых команд. Последовательно выделим текст, нажмём кнопку «Ctrl+C», выберем место для вставки в тексте, нажмём кнопку «Ctrl+V»:

1. 

2. 

3. 

Теперь попробуем отменить последнюю команду нажатием кнопки «Ctrl+Z»:



Таким образом, реализация шаблона проектирования Команда позволила нам представлять каждое из возможных действий в виде отдельного объекта, которые могут выстраиваться в стек и в последствии быть отменены. При желании количество команд может неограничено расширяться за счёт общего для всех комманд абстрактного класса, а все другие возможные места вызова команд (например, горячие клавиши) будут также создавать аналогичные объекты команд, как и те, которые создаются нажатием на кнопки в GUI.

Стратегия (Strategy)

Назначение: это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно взаимозаменять прямо во время исполнения программы.

Когда использовать: Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта; Когда у вас есть множество похожих классов, отличающихся только некоторым поведением; Когда вы не хотите обнажать детали реализации алгоритмов для других классов; Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет вариацию алгоритма.

Пример: в зависимости от выбора пользователя в методе Main выбирается конкретная стратегия защиты базы NightElfBase с помощью ее метода setDefenceStrategy(), после осуществляется нападение на базу Орками/Людьми/Нежитью и проверяется, подходит ли выбранная стратегия для защиты.
Диаграммы классов и последовательности представлены на Рис.17.1, Рис.17.2, Рис.17.3 и Рис.18 соответственно.

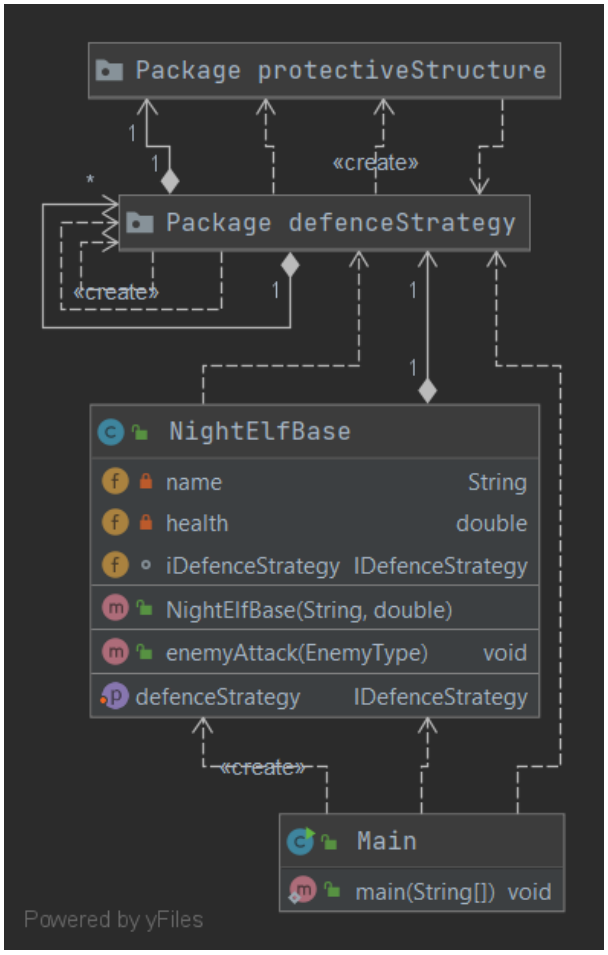


Рис.17.1. Диаграмма классов Strategy

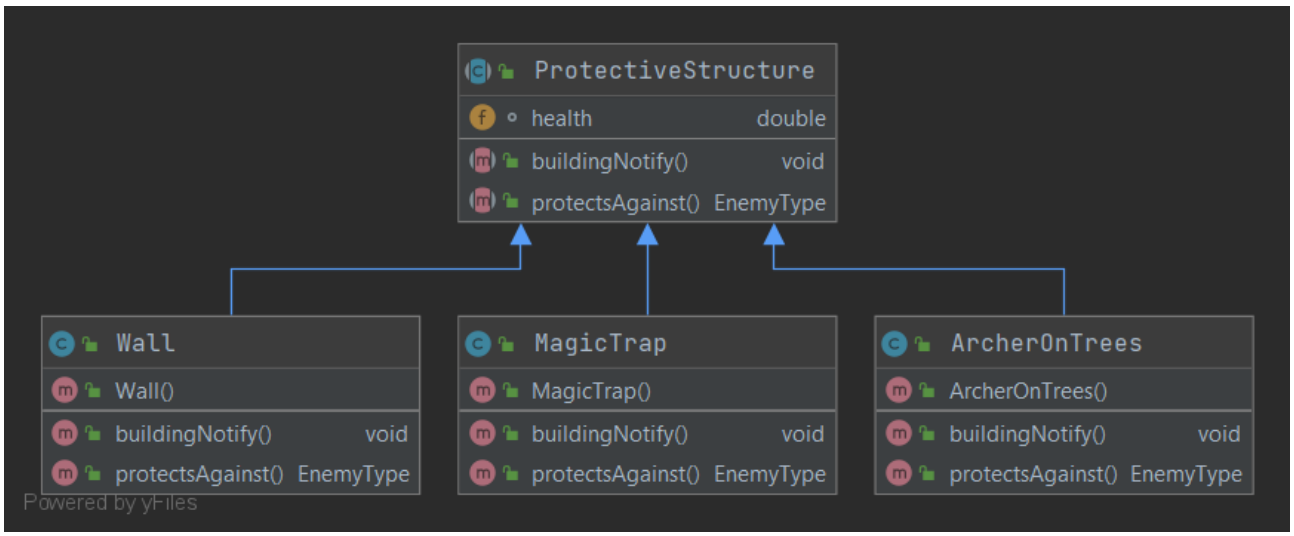


Рис.17.2. Диаграмма классов Strategy

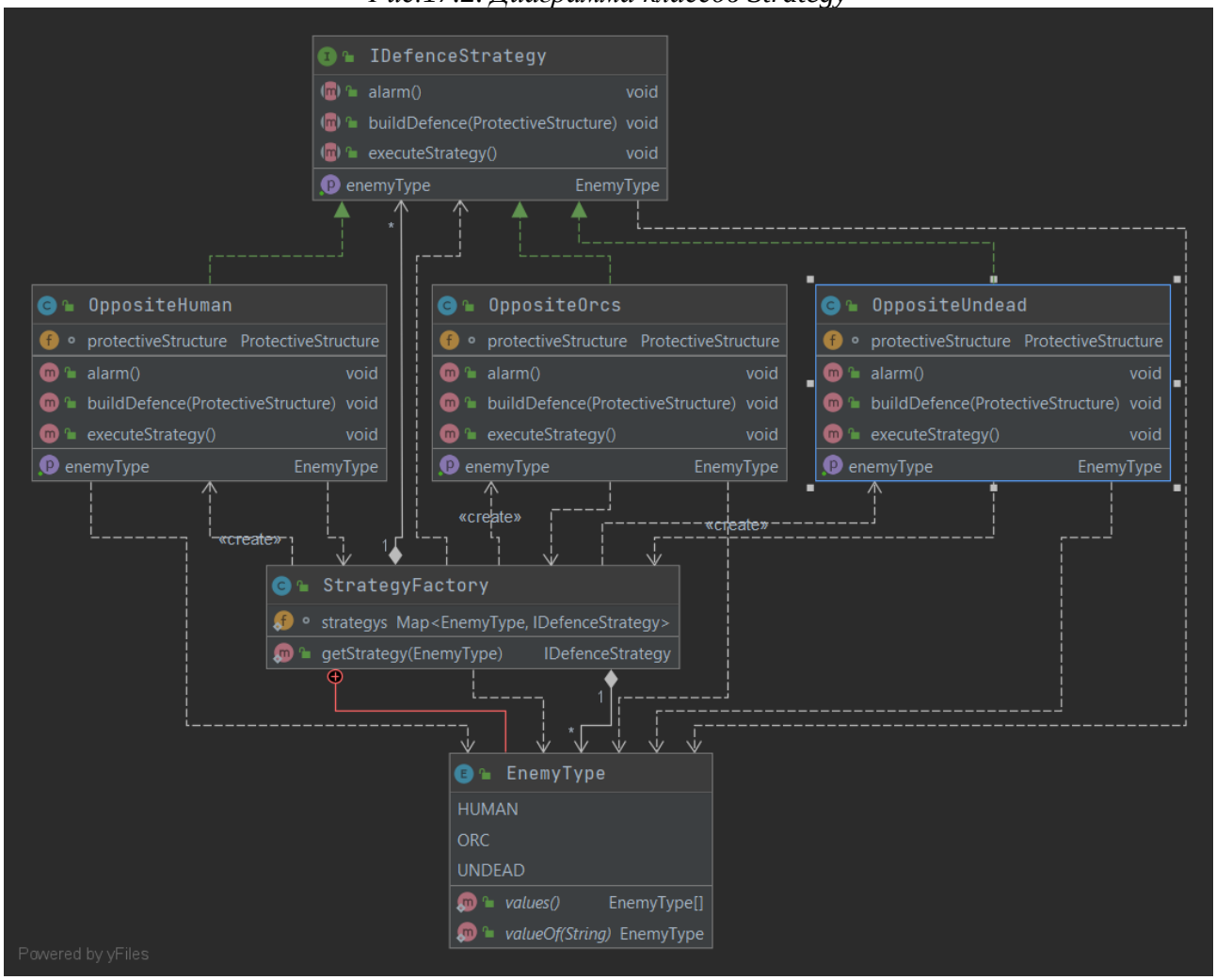


Рис.17.3. Диаграмма классов Strategy

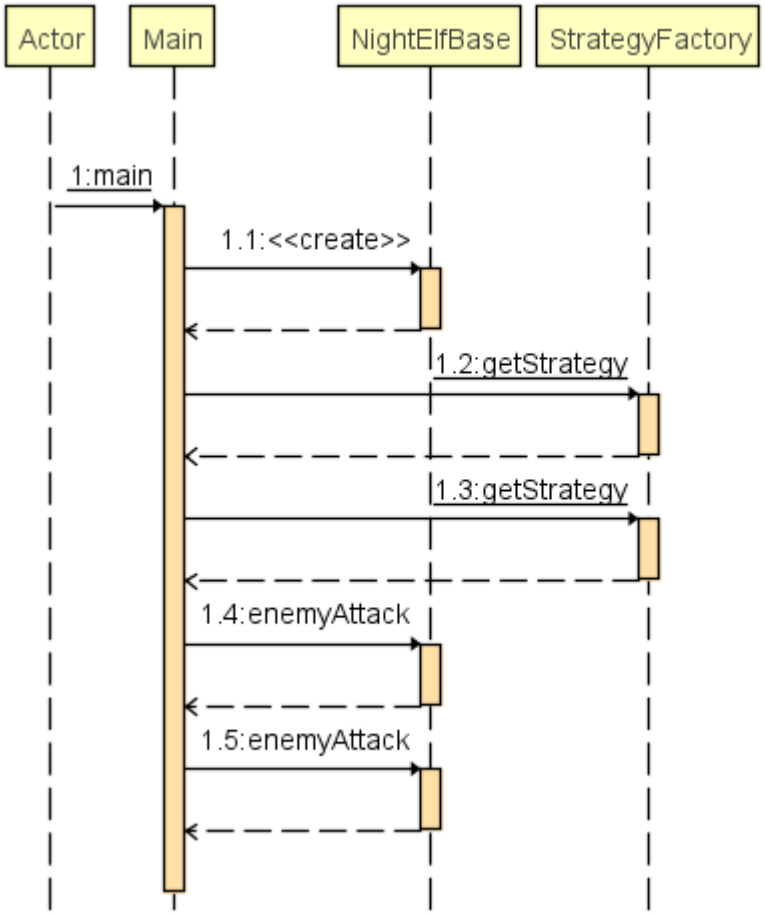


Рис.18. Диаграмма последовательности Strategy

Код примера:

IDefenceStrategy.java

```
package com.company.defenceStrategy;

import com.company.protectiveStructure.*;
/** Общий интерфейс для стратегий защиты */
public interface IDefenceStrategy {
    /** Оповещение */
    void alarm();
    /** Возвести защиту */
    void buildDefence(ProtectiveStructure protectiveStructure);
    /** Реализовать стратегию защиты */
    void executeStrategy();
    /** Проверка защиты от конкретного типа врага */
    public abstract StrategyFactory.EnemyType getEnemyType();
}
```

OppositeOrc.java

```
package com.company.defenceStrategy;

import com.company.protectiveStructure.*;

public class OppositeOrcs implements IDefenceStrategy{
    ProtectiveStructure protectiveStructure;

    @Override
    public void alarm() {
        System.out.println("Берегитесь орки идут в нападение!");
```

```
    }

    @Override
    public void buildDefence(ProtectiveStructure protectiveStructure) {
        this.protectiveStructure = protectiveStructure;
    }

    @Override
    public void executeStrategy() {
        alarm();
        System.out.println("Возводим защитные сооружения.");
        buildDefence(new Wall());
        System.out.println("Защищаемся от глухой обороны, стражники на стены!");
    }

    @Override
    public StrategyFactory.EnemyType getEnemyType() {
        return StrategyFactory.EnemyType.ORC;
    }
}
```

OppositeHuman.java

```
package com.company.defenceStrategy;

import com.company.protectiveStructure.*;

public class OppositeHuman implements IDefenceStrategy{
    ProtectiveStructure protectiveStructure;

    @Override
    public void alarm() {
        System.out.println("Люди возводят базу рядом с нами!");
    }

    @Override
    public void buildDefence(ProtectiveStructure protectiveStructure) {
        this.protectiveStructure = protectiveStructure;
    }

    @Override
    public void executeStrategy() {
        alarm();
        System.out.println("Организуем засаду!");
        buildDefence(new ArcherOnTrees());
        System.out.println("Устроим им партизанскую войну, друиды
перевоплощайтесь в зверей!");
    }

    @Override
    public StrategyFactory.EnemyType getEnemyType() {
        return StrategyFactory.EnemyType.HUMAN;
    }
}
```

OppositeUndead.java

```
package com.company.defenceStrategy;

import com.company.protectiveStructure.*;

public class OppositeUndead implements IDefenceStrategy{
    ProtectiveStructure protectiveStructure;
```

```
    @Override
    public void alarm() {
        System.out.println("Мерзкая нежить пачкает наши леса!");
    }

    @Override
    public void buildDefence(ProtectiveStructure protectiveStructure) {
        this.protectiveStructure = protectiveStructure;
    }

    @Override
    public void executeStrategy() {
        alarm();
        System.out.println("Установите для них ловушки, нежить лучше не
подпускать!");
        buildDefence(new MagicTrap());
        System.out.println("Просим помощи у леса, дендроиды пойдут на
передовую.");
    }

    @Override
    public StrategyFactory.EnemyType getEnemyType() {
        return StrategyFactory.EnemyType.UNDEAD;
    }
}
```

StrategyFactory.java

```
package com.company.defenceStrategy;

import java.util.HashMap;
import java.util.Map;
/** Фабрика стратегий защиты */
public class StrategyFactory {
    /** Хранение всех пар "Враг - Стратегия Защиты" в единственном виде */
    static Map<EnemyType, IDefenceStrategy> strategys = new HashMap<>();
    /** Перечисление типов нападающих **/
    public enum EnemyType
    {
        HUMAN, // Люди
        ORC, // Орки
        UNDEAD // Нежить
    }
    /** Получение транспорта указанного типа **/
    public static IDefenceStrategy getStrategy(EnemyType ET)
    {
        IDefenceStrategy result = strategys.get(ET);
        if (result == null) {
            switch(ET)
            {
                case HUMAN:
                {
                    result = new OppositeHuman();
                    break;
                }
                case ORC:
                {
                    result = new OppositeOrcs();
                    break;
                }
                case UNDEAD:
                {
                    result = new OppositeUndead();
                    break;
```

```
                }
                default:
                {
                    break;
                }
            }
            strategys.put(ET, result);
        }
        return result;
    }
}
```

ProtectiveStructure.java

```
package com.company.protectiveStructure;

import com.company.defenceStrategy.StrategyFactory;

/** Способ обороны */
public abstract class ProtectiveStructure {
    double health;
    public abstract void buildingNotify();
    /** Проверка защиты от конкретного типа врага */
    public abstract StrategyFactory.EnemyType protectsAgainst();
}
```

ArcherOnTrees.java

```
package com.company.protectiveStructure;

import com.company.defenceStrategy.StrategyFactory;

public class ArcherOnTrees extends ProtectiveStructure{

    public ArcherOnTrees() {
        this.health = 700;
        buildingNotify();
    }

    @Override
    public void buildingNotify() {
        System.out.println("Лучники снайперы заняли позиции!");
    }

    @Override
    public StrategyFactory.EnemyType protectsAgainst() {
        return StrategyFactory.EnemyType.HUMAN;
    }
}
```

Wall.java

```
package com.company.protectiveStructure;

import com.company.defenceStrategy.StrategyFactory;

public class Wall extends ProtectiveStructure{

    public Wall() {
        this.health = 1000;
        buildingNotify();
    }

    @Override
```

```
    public void buildingNotify() {
        System.out.println("Возведены стены, врагу не пройти!");
    }

    @Override
    public StrategyFactory.EnemyType protectsAgainst() {
        return StrategyFactory.EnemyType.ORC;
    }
}
```

MagicTrap.java

```
package com.company.protectiveStructure;

import com.company.defenceStrategy.StrategyFactory;

public class MagicTrap extends ProtectiveStructure{

    public MagicTrap() {
        this.health = 400;
        buildingNotify();
    }

    @Override
    public void buildingNotify() {
        System.out.println("В окрестностях установлены магические ловушки, врага
ждет сюрприз");
    }

    @Override
    public StrategyFactory.EnemyType protectsAgainst() {
        return StrategyFactory.EnemyType.UNDEAD;
    }
}
```

NightElfBase.java

```
package com.company;

import com.company.defenceStrategy.*;

import java.awt.*;

/** База, требующая защиты */
public class NightElfBase {
    private String name;
    private double health;
    IDefenceStrategy iDefenceStrategy;

    public NightElfBase(String name, double health) {
        this.name = name;
        this.health = health;
        System.out.println("Создана лесная база ночных эльфов " + name);
    }

    public void setDefenceStrategy(IDefenceStrategy strategy) {
        iDefenceStrategy = strategy;
        iDefenceStrategy.executeStrategy();
    }

    public void enemyAttack(StrategyFactory.EnemyType ET){
        System.out.println(ET + " атакуют " + this.name);
        if (this.health <= 0 || iDefenceStrategy == null ||
iDefenceStrategy.getEnemyType() != ET) {
```

```
            System.out.println("  Защита не выдержала! Стратегия была
провальна...");
            this.health = 0;
            System.out.println("  База стёрта с лица земли");
        }
        else {
            System.out.println("  Атака успешно отражена! Ваша стратегия
сработала!");
        }
    }

}
```

Результат работы:

Для проверки работоспособности сначала создадим базу эльфов, требующую защиты:

```
NightElfBase nightElfBase = new NightElfBase("Лесная Чаща", 3000);
```

Вывод в терминал:

```
Создана лесная база ночных эльфов Лесная Чаща
```

Также предположим, что в нашем распоряжении имеется две возможные стратегии защиты:

```
IDefenceStrategy strategyAgainstHuman =
StrategyFactory.getStrategy(StrategyFactory.EnemyType.HUMAN);
IDefenceStrategy strategyAgainstOrc =
StrategyFactory.getStrategy(StrategyFactory.EnemyType.ORC);
```

И наша разведка доложила о приближении агрессивно настроенных Людей, выберем стратегию против них:

```
nightElfBase.setDefenceStrategy(strategyAgainstHuman);
```

Вывод в терминал:

```
Люди возводят базу рядом с нами!
Организуем засаду!
Лучники снайперы заняли позиции!
Устроим им партизанскую войну, друиды перевоплощайтесь в зверей!
```

Момент истины! На нас напали Люди, разведка оказалась права:

```
nightElfBase.enemyAttack(StrategyFactory.EnemyType.HUMAN);
```

Вывод в терминал:

```
HUMAN атакуют Лесная Чаща
  Атака успешно отражена! Ваша стратегия сработала!
```

Проверим, что было бы, если разведка ошиблась, и на нас нападали бы Орки:

```
nightElfBase.enemyAttack(StrategyFactory.EnemyType.ORC);
```

Вывод в терминал:

```
ORC атакуют Лесная Чаща
  Защита не выдержала! Стратегия была провальна...
  База стёрта с лица земли
```

Таким образом, мы реализовали паттерн Стратегия и показали его успешную работоспособность.

Выводы

При выполнении учебно-технологической практики мы ознакомились с основными типами шаблонов проектирования(паттернами), а также реализовали некоторые из них на языке программирования Java.

Список использованных источников

1. Разумовский Г.В. Порождающие шаблоны проектирования / Лекция от 01.07.2021

2. Разумовский Г.В. Структурные шаблоны проектирования / Лекция от 02.07.2021

3. Разумовский Г.В. Поведенческие шаблоны проектирования / Лекция от 05.07.2021

4. Александр Швец. Погружение в паттерны проектирования, 2018. – 385 с.

5. JavaRush. URL: https://javarush.ru [дата последнего обращения 05.07]