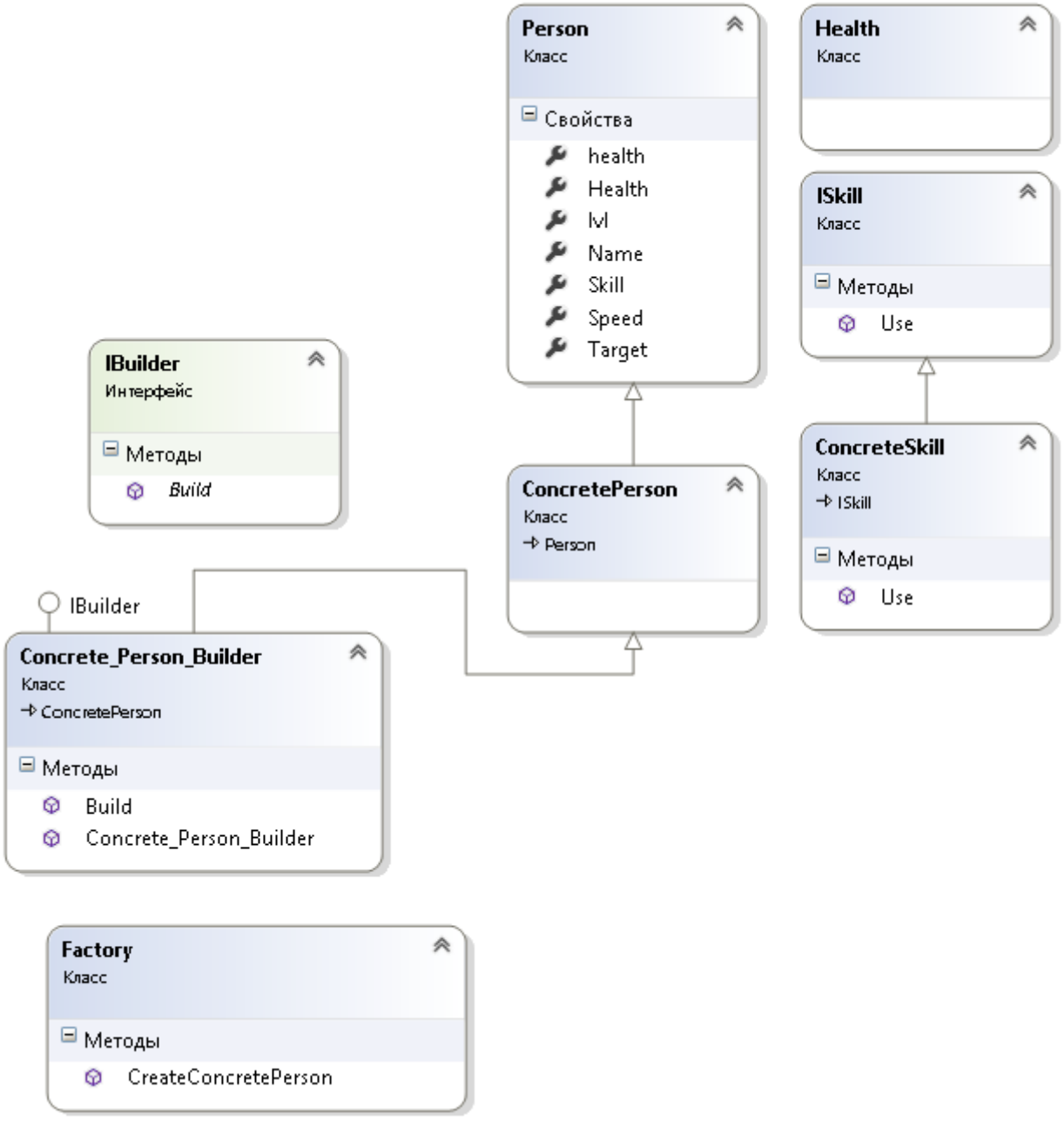
Разбор шаблонов проектирования на примере простой консольной игры.

Используемые шаблоны:

- Стратегия – поведенческий.
- Строитель – порождающий.
- Фабрика – порождающий.
- Одиночка – порождающий.

Структура приложения ввиде UML диаграм:



Persons:

```
public abstract class Person
{
    protected int speed;
    protected String name;
    protected int lvl;
    protected Person target;
    protected Health health;
    protected ISkill skill;
        ……
  Getter and setter methods;
}
```

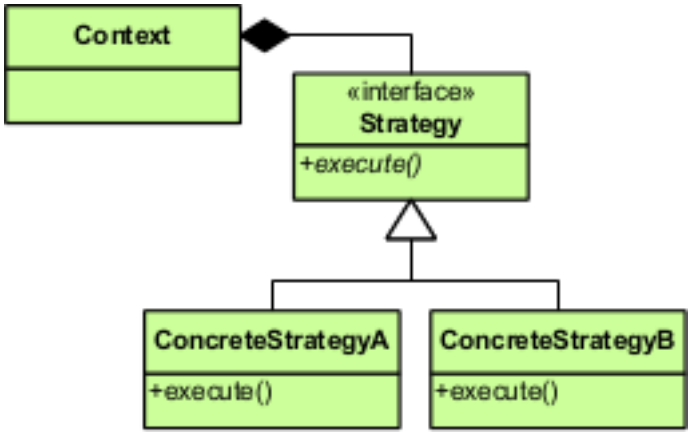
concrete_Person – наглядное абстрактное понятие кастомизации Person. В коде будут использованы: Sorcerer, warior, hunter.

```
public class concrete_Person extends Person  {
  protected int powerSkill = 10;
}
```

Components:

Каждый персонаж содержит переменную склочного типа ISkill которая в свою очередь содежит метод Use.

ISkill – Стратегия реализуя которую мы задам поведение конкретного обьекта.




```
public interface ISkill {
    public  void Use(Person target);
    public void setPower(int power);
    public int getPower();
}
```

Реализации ISkill на примере класса IceArrow

```
public class IceArrow implements ISkill {
    int power = 0;
    public IceArrow(int power ) {
        this.power = power;
    }
    @Override
    public void setPower(int power) {
        power = power;
```

```
    }
    @Override
    public int getPower() {
        return power;
    }
    @Override
    public void Use(Person target) {
        System.out.println("Целе " + target.getName()+" нанесено " +
this.power + " урона ледяной стрелой...");
    }
}
```

Например для огненного шара мы изменим **урона ледяной стрелой** на **урона огненным шаром** , а так же можем изменить функцию расчет урона итд.

Так же каждый персонаж содержит переменную склочного типа Health отвечающую за уровень его жизни.

```
public class Health
{
    private int HelghtPoints;
    public void setHelghtPoints(int helghtPoints) {
        HelghtPoints = helghtPoints;
    }
    public int getHelghtPoints() {
        return HelghtPoints;
    }
    public Health(int helghtPoints) {
        HelghtPoints = helghtPoints;
    }
}
```

Что бы скрыть процес построения обьектов наследуемых от класса Person можно использовать шаблон строитель.

Шаблон состоит из трех компонентов: Director, AbstractBuilder, Builder.
Директор это не кий заказник построения обьекта, это может бать как код-клиент так и специализированый клас – как в нашем случае (будете показано ниже).
AbstractBuilder в нашем случае интерфейс IBuilder.

Реализация-листинг

```
public interface IBuilder
{
    public  Person Build();
}
```

1) Он может напрямую возвращать concrete_person
2) Скрываем реализацию обьекта внутри билдера.
3) Можем реализовать внутри билдера необходимые нам алгоритмы, не затрагивая при этом на клиентский код.

```
public class IceSorcererBuilder extends Sorcerer implements IBuilder
{
    @Override
    public void setLvl(int lvl) {
        super.setLvl(lvl);
    }
    @Override
    public void setName(String name) {
        super.setName(name);
    }

    @Override
    public void setSpeed(int speed) {
        super.setSpeed(speed);
    }

    @Override
    public void setHealth(Health health) {
        super.setHealth(health);
    }

    public IceSorcererBuilder(String name, int lvl, int speed ) {
        this.setHealth(new Health(5*lvl));
        this.setSpeed(speed);
        this.setLvl(lvl);
        this.setName(name);
        this.skill = new IceArrow(powerSkill * lvl);
    }

    public Person Build()
    {
        this.setName(name + " Ice Sorcerer");
        return  this;
    }
}
```

Шаблоны фабрика и Одиночка.

В данном примере кода реализованы сразу 2 шаблона:
диночка, который позволяет создавать обьект только 1 раз, таким образом реализовав концепцию один ко многим.

Фабрика – содержит методы создание для всех типов Person и реализует создание обьекта от пользователя оставля ему необходимый інтерфейс.

Итог по фабрике – создаем только один раз, скрываем реализацию обьекта от пользователя. При этом используем строитель для самой реализации

```
public class OrkFactory   {

    private static volatile OrkFactory _instance = null;

    private OrkFactory() {}

    public static synchronized OrkFactory getInstance() {
        if (_instance == null)
            synchronized (OrkFactory.class) {
                if (_instance == null)
                    _instance = new OrkFactory();
            }
        return _instance;
    }
    public Hunter CreateOrkHunter(int lvl) {
        return (Hunter) new HunterBuilder("Ork", lvl, 4).Build();
    }

    public Warior CreateOrkLanceKinght(int lvl) {
        return (Warior) new LanceKnightBuilder("Ork", lvl, 2);
    }

    public Warior CreateOrkKinght(int lvl) {
        return (Warior) new KnightBuilder("Ork", lvl, 2);
    }

    public Sorcerer CreateOrkFierySorcerer(int lvl) {
        return (Sorcerer) new IceSorcererBuilder("Ork", lvl, 2);
    }

    public Sorcerer CreateOrkIceSorcerer(int lvl) {
        return (Sorcerer) new FierySorcererBuilder("Ork", lvl, 2);
    }
}
```

Пример реализации всего выше написанного в Main

```
public class Main {

    public static void main(String[] args) {

         List<Person> orks = new ArrayList();
        OrkFactory orkFactory = (OrkFactory) OrkFactory.
                getInstance();
        orks.add(orkFactory.CreateOrkFierySorcerer(1));
        orks.add(orkFactory.CreateOrkHunter(1));
        orks.add(orkFactory.CreateOrkHunter(1));
        orks.add(orkFactory.CreateOrkIceSorcerer(1));
        orks.add(orkFactory.CreateOrkKinght(1));
        orks.add(orkFactory.CreateOrkKinght(1));
        orks.add(orkFactory.CreateOrkKinght(1));
        orks.add(orkFactory.CreateOrkKinght(1));
        orks.add(orkFactory.CreateOrkKinght(1));
        orks.add(orkFactory.CreateOrkLanceKinght(1));
        orks.add(orkFactory.CreateOrkLanceKinght(1));


        Person player = new IceSorcererBuilder("Kelj",10,2).Build();

        for (Person person: orks
             )
        {
         person.getSkill().Use(player);
        }
    }
}
```