

# OMO

## 4 - Creational design patterns

- Singleton
- Simple Factory
- Factory Method
- Abstract Factory
- Prototype
- Builder
- IoC

---

Ing. David Kadleček, PhD.

[kadlecd@fel.cvut.cz](mailto:kadlecd@fel.cvut.cz), [david.kadlecek@cz.ibm.com](mailto:david.kadlecek@cz.ibm.com)

# Co jsou to design patterns

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern

## “GANG OF FOUR” NÁVRHOVÉ VZORY

ERICH GAMMA, RICHARD HELM, RALPH JOHNSON A JOHN VLISIDES

- CREATIONAL – VZORY PRO VYTVÁŘENÍ INSTANCÍ
- STRUCTURAL – VZORY PRO ORGANIZOVÁNÍ OBJEKTŮ DO STRUKTUR
- BEHAVIORAL – VZORY PRO PROVÁDĚNÍ PROGRAMOVÉ LOGIKY

# Creational design patterns

= “chytřejší konstruktory”, které:

- skrývají komplexitu vytváření objektů
- předepisují typy objektů/metod, které musí potomci třídy, vytvářet/implementovat
- zaručují, že se vytvoří pouze jedna instance
- vytváří přesné kopie objektů (včetně všech atributů a navázaných tříd)
- “Zapůjčují” již vytvořené objekty, místo neustálého vytváření nových

A jsou to:

- **Singleton**
- **Simple Factory**
- **Abstract Factory**
- **Factory Method**
- **Prototype**
- **Builder**
- **IoC**

# Singleton

Singleton se používá jestliže chceme, aby od jedné třídy existovala maximálně jedna implementace.

Používá se např. Pro načítání konfigurace, drivery pro ovládání zařízení, logging atd.

Obecně se doporučuje vyhnout se jeho používání, protože se jedná o “**synchronizovanou globální proměnnou**”, která je jednak mutable a také je zdrojem blokování threadů při větší zátěži

*Pozn. bez synchronized nefunguje v multithreaded aplikaci*

```
class ClassSingleton {
    private static ClassSingleton INSTANCE;
    private String info = "Initial info class";

    private ClassSingleton() {
    }

    public synchronized static ClassSingleton getInstance(){
        if (INSTANCE == null) {
            INSTANCE = new ClassSingleton();
        }

        return INSTANCE;
    }

    // getters and setters
}
```

# Simple Factory

Vytváření objektů konkrétního typu je zapouzdřené do metod ve specializované třídě. Uživatel třídy je odstíněn od komplexity vytváření objektů.

```
class PlantFactory {
    public Apple makeApple(int size, Boolean hasGoodTaste) {
        // Code for creating an Apple here.
    }
    public Orange makeOrange(int size) {
        // Code for creating an orange here.
    }
}

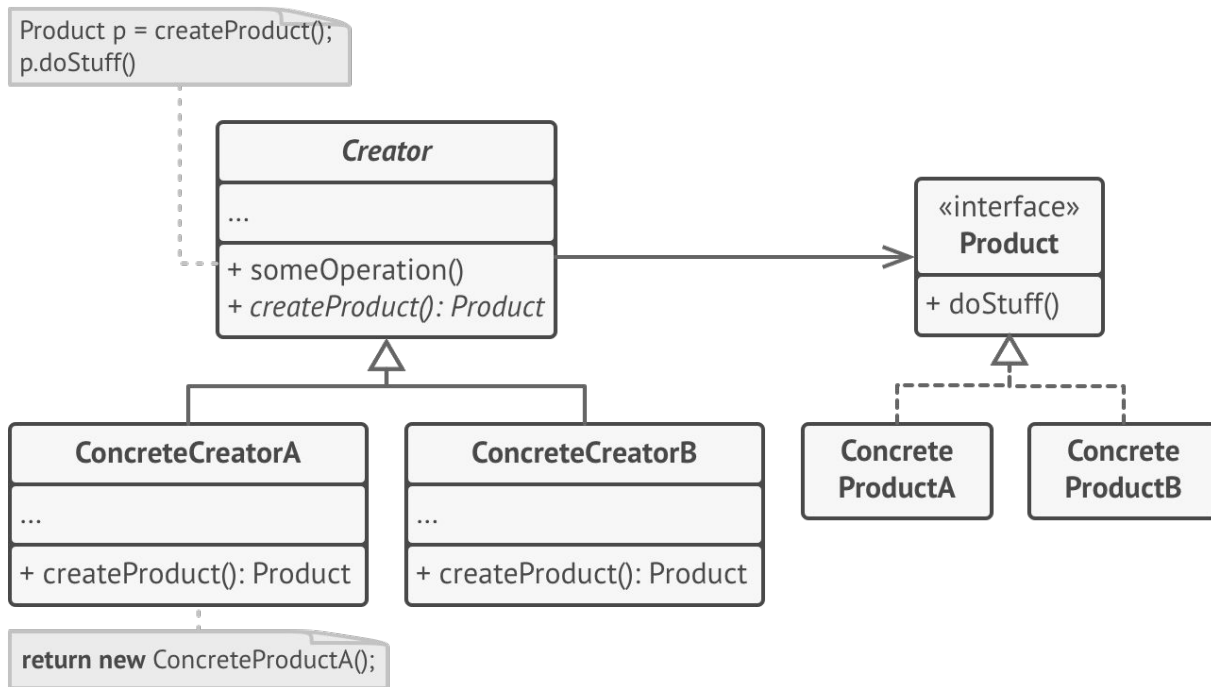
...
PlantFactory plantFactory = new PlantFactory()
Apple bigTastyApple = plantFactory.makeApple(10, true)
Apple smallUglyApple = plantFactory.makeApple(2, false)
Orange orange = plantFactory.makeOrange(5)
```

**Q:** *Jaká je výhoda proti jednoduchému konstruktoru?*

**A:** *Factory může poskytnout dalším vývojářům jako API na vytváření ovoce. Factory vymezuje: (i) všechny typy ovoce, co chci vyrábět, (ii) přes jaké metody umožňuji jejich vytváření. Dále pak do Factory můžu zapracovat aspekty jako např. Kolik ovoce jsem vyrobil, centrální logování výroby atd.*

# Factory Method

**Factory Method** se používá za účelem vytvoření objektu pokud nechceme specifikovat konkrétní třídu objektu, který je vytvářen. Místo konstrukturu objektu se volá factory metoda, která je implementována/nebo předefinována v potomkovi.



# Factory Method

Druh objektu i jeho počáteční vlastnosti jsou dané přijatými parametry, případně i stavem objektu, který tovární metodu poskytuje.

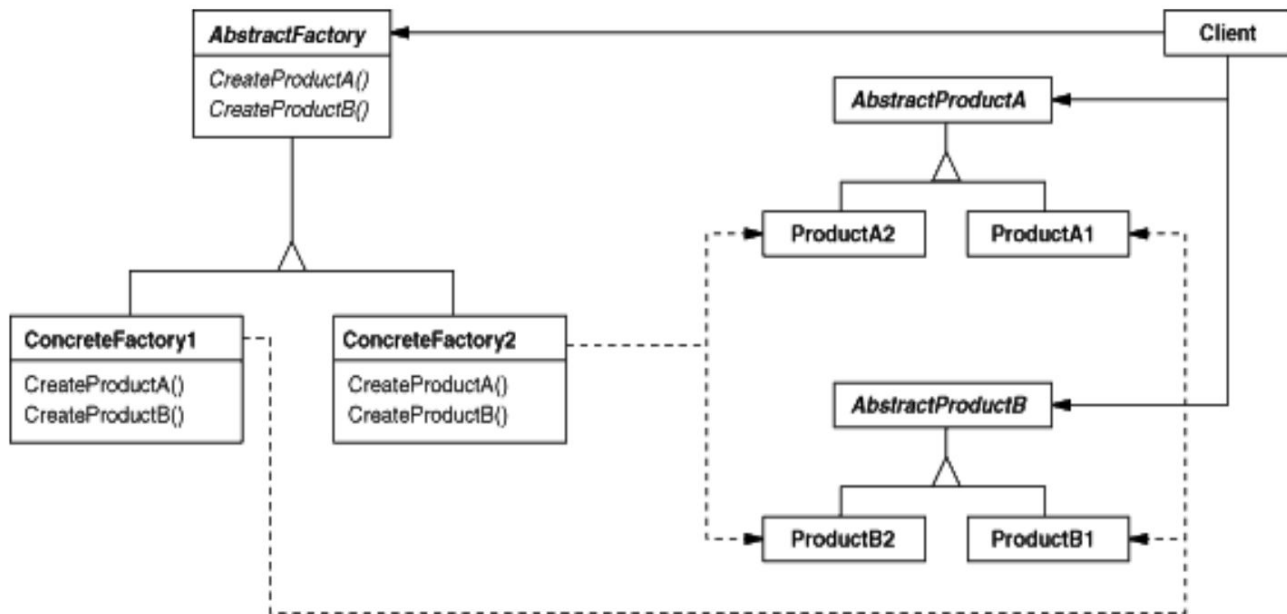
Používá se tam, kde je vytvářený objekt nějakým způsobem odvozený od aktuální instance třídy, která Factory Method poskytuje. Tyto metody se používají hlavně u immutable tříd.

```
abstract class FruitCreator {
    protected abstract Fruit makeFruit(int size);
    public void pickFruit() {
        private final Fruit f = makeFruit();
    }
}

class OrangeCreator extends FruitCreator {
    @Override protected Fruit makeFruit(int size){
        if (size > 5)
            return new BigOrange();
        else
            return new SmallOrange();
    }
}
...
OrangeCreator fruitCreator = new OrangeCreator();
Fruit orange = fruitCreator.makeFruit(4) //returns
SmallOrange instance
```

# Abstract Factory

**Abstract Factory** je abstraktním rozšířením *Simple Factory*, když chceme vytvářet celou rodinu objektů, ale v různých variantách. Z *Factory Method* se zase přebírá to, že není třeba přesně specifikovat produkt, který vytváříme, ale pouze předepisujeme jaké typy objektů se mají vytvářet.



Příkladem je grupa Volkswagen, která sdílí komponenty a postupy přes své dceřiné společnosti. Existuje *GroupAbstractFactory*, která předepisuje co se má vyrábět v přes jaké rozhraní. Od ní jsou oddělené *AudiFactory*, *SkodaFactory*, *VolkswagenFactory* ...



## Vytváření produktů:

```
interface PlantFactory {
    Plant makePlant();
    Picker makePicker();
}

public class AppleFactory implements PlantFactory {
    Plant makePlant() {
        return new Apple();
    }
    Picker makePicker() {
        return new ApplePicker();
    }
}

public class OrangeFactory implements PlantFactory {
    Plant makePlant() {
        return new Orange();
    }
    Picker makePicker() {
        return new OrangePicker();
    }
}
```

## Produkty:

```
abstract class Plant {
    ...
}

class Apple extends Plant {
    ...
}

class Orange extends Plant {
    ...
}

abstract class Picker {
    ...
}

class ApplePicker extends Picker {
    ...
}
```

## Client:

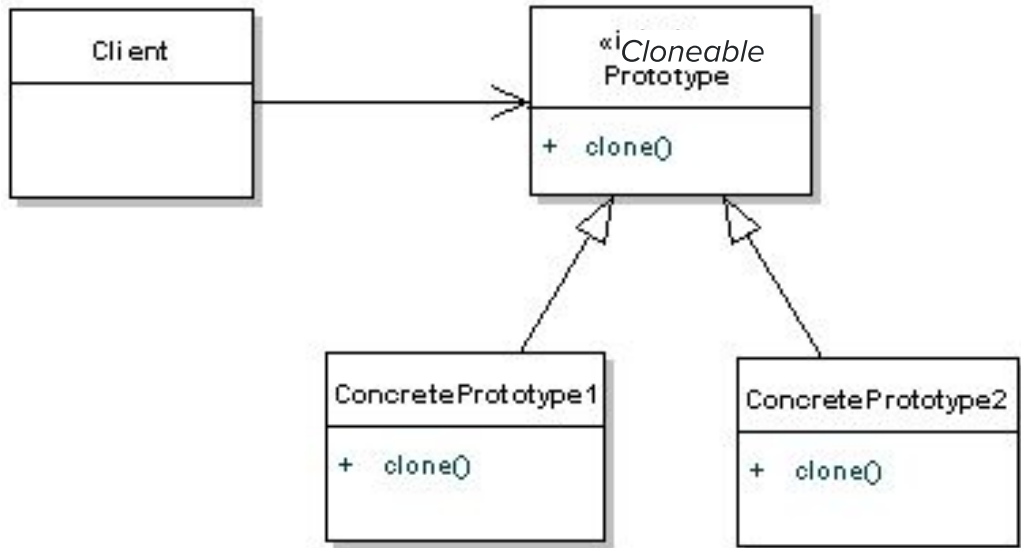
```
PlantFactory appleFactory = new AppleFactory();
PlantFactory orangeFactory = new OrangeFactory();
Plant apple = appleFactory.makePlant();
...
```

# Prototype

**Prototype pattern** = kontrolované vytváření kopií objektů. Zavedeme interface *Cloneable*, ten předepisuje metodu *clone()*, kterou musí implementovat všechny odvozené třídy. Odvozené třídy v metodě *clone* vytvoří kopii sebe sama.

**Shallow copy** - *clone()* vytvoří novou instanci té samé třídy a nastaví všechny atributy na hodnoty atributů z instance na které voláme *clone()*.

**Deep copy** - *clone()* provede shallow copy a pak *clone()* i všech navázaných objektů. Naklonuje celý objektový graf. Hloubka kopie může být různá.



# Prototype

```
public interface Cloneable {
    Cloneable clone();
}
public class Address implements Cloneable {
    private String street;
    public Address(String street){
        this.street = street;
    }
    @Override
    public Cloneable clone() {
        return new Address(this.street);
    }
}
```

```
public class Employee implements Cloneable {
    private String name;
    private Address address;
    public Employee(String name, Address address){
        this.name = name;
        this.address = address;
    }
    @Override
    public Cloneable clone() {
        return new Employee(this.name, (Address)
address.clone());
    }
}
```

Client:

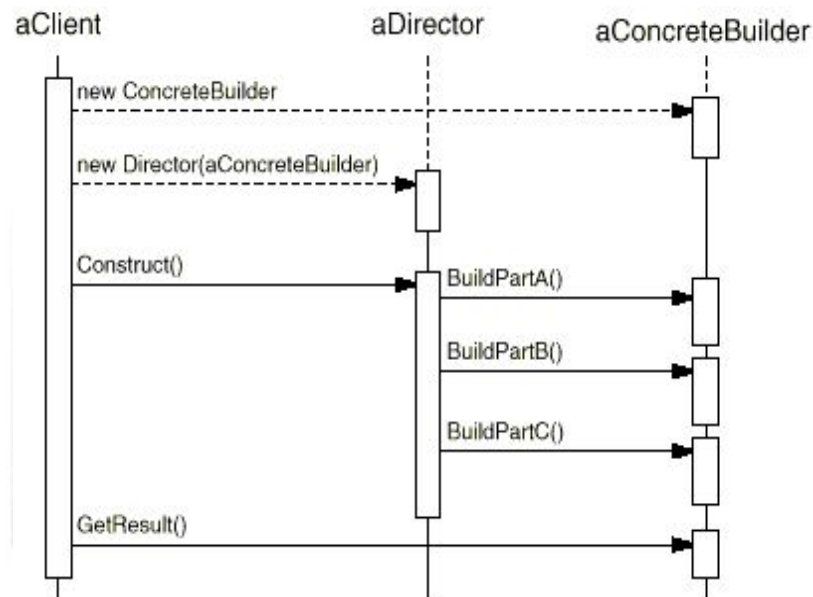
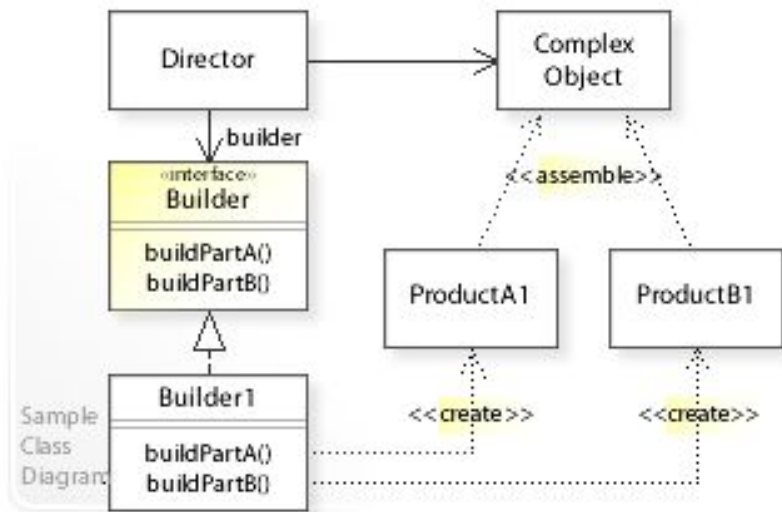
```
...
Employee employee = new Employee("Karel Novak", new Address("Karlova 12"));
Employee employeeCopy = (Employee) employee.clone();
```

# Builder

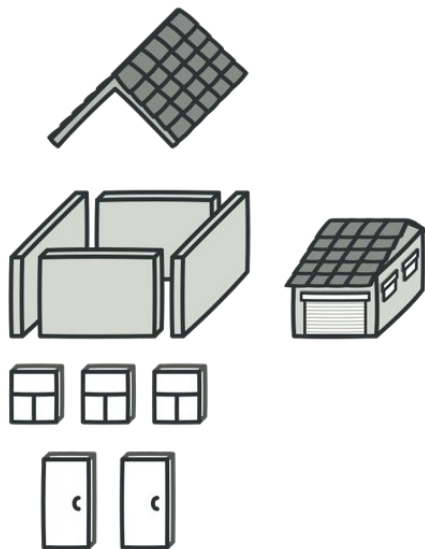
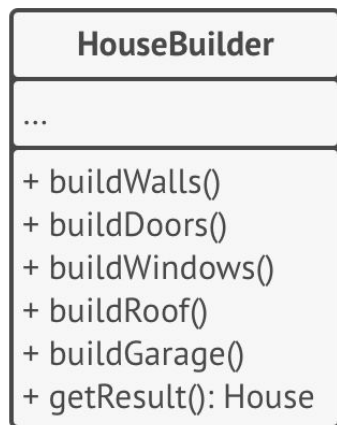
**Builder pattern** = oddělení konstrukce komplexního objektu od jeho reprezentace. Stejný konstrukční proces (sekvence příkazů) může vytvořit různé reprezentace

**Výhody** - možnost prohazování interních reprezentací, mohou ovládat proces konstrukce (aplikací různých kroků v různém pořadí sestavovat různé objekty) - narozdíl od Factory, kde zadám parametry finálního objektu.

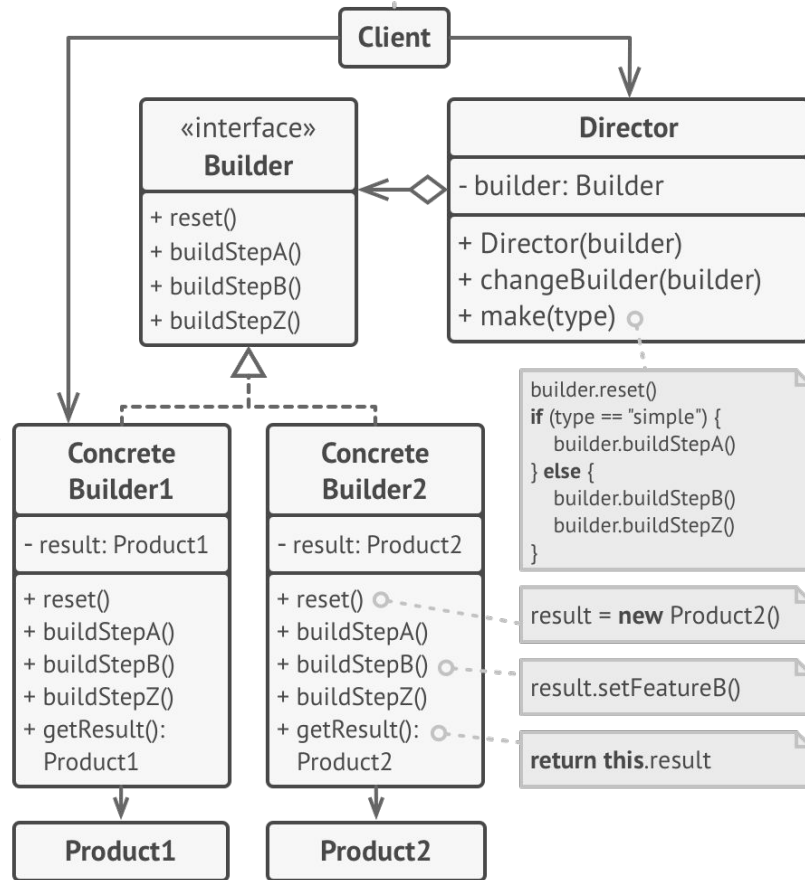
**Nevýhody** - třídy builderu jsou mutable, vyžaduje různé konkrétní buildery pro každý typ objektu



# Builder



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```



# Builder - příklad s fluent interfacem

```
enum EColor {  
    RED, WHITE, BLACK  
}  
  
public class House {  
    private EColor color;  
    private List<Window> windows = new ArrayList<>();  
    private List<Wall> walls = new ArrayList<>();  
    private Door door;  
  
    public void setColor(EColor color) {  
        this.color = color;  
    }  
  
    public void addWindow(Window window) {  
        this.windows.add(window);  
    }  
  
    public void addWall(Wall wall) {  
        this.walls.add(wall);  
    }  
  
    public void setDoor(Door door) {  
        this.door = door;  
    }  
}
```

# Builder - příklad s fluent interfacem

```
public class HouseBuilder {
    private House house = new House();

    public HouseBuilder addWall(){
        house.addWall(new Wall());
        return this;
    }
    public HouseBuilder addWindow(){
        house.addWindow(new Window());
        return this;
    }
    public HouseBuilder buildDoor() {
        house.setDoor(new Door());
        return this;
    }
    public HouseBuilder paint(EColor
color){
        house.setColor(color);
        return this;
    }
    public House getResult(){
        return house;
    }
    ...
}
```

**fluent interface** - metody  
vrací *HouseBuilder*, což  
umožňuje, že po zavolání  
metody mohou hned v řetězci  
volat další metodu

# Builder - příklad s fluent interfacem

```
public static void main(String [] p){
    HouseBuilder builder = new HouseBuilder();
    //Add walls
    House yourHouse = builder.
        addWall().
        addWall().
        addWall().
        addWall().
        addWindow().
        addWindow().
        buildDoor().
        paint(IColor.RED).
        getResult();
}
```

## <<= Fluent interface

celou konstrukci provedu  
jedním zřetězeným voláním



# Dependency injection

Frameworks:

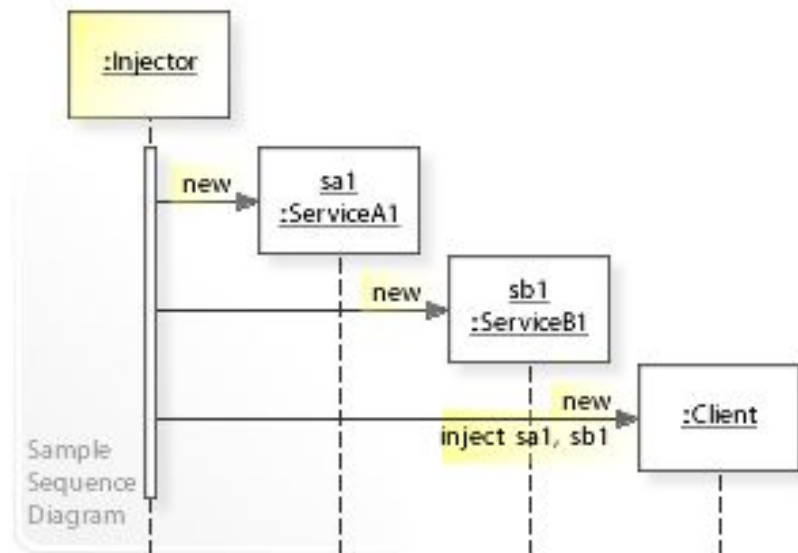
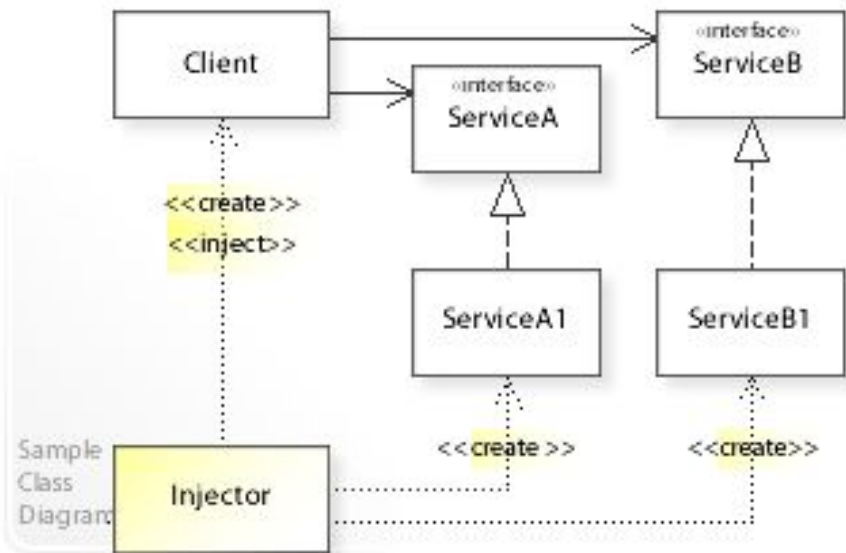
*Spring*

*google/guice*

*Dagger*

*Autofac*

**VKLÁDÁNÍ ZÁVISLOSTÍ (DEPENDENCY INJECTION (DI))** JE V OBJEKTOVĚ ORIENTOVANÉM PROGRAMOVÁNÍ TECHNIKA PRO VKLÁDÁNÍ ZÁVISLOSTÍ MEZI JEDNOTLIVÝMI KOMPONENTAMI PROGRAMU TAK, ABY JEDNA KOMPONENTA MOHLA POUŽÍVAT DRUHOU, ANIŽ BY NA NI MĚLA V DOBĚ SESTAVOVÁNÍ PROGRAMU REFERENCI.



# Dependency injection

```
@Configuration
@ComponentScan("my.package")
public class Config {

    @Bean(name = "oEng")
    public Engine engine() {
        return new Engine("v8", 5);
    }
    @Bean
    public Transmission transmission() {
        return new Transmission("sliding");
    }
}

@Component
public class Car {

    @Autowired
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

# Dependency injection

**@Bean** - tato anotace indikuje, že metoda vyprodukuje bean, spravovaný Spring kontainerem. Beany jsou deklarovány pomocí anotace **@Bean** annotation v konfigurační třídě

**@Component** je anotace na úrovni třídy. Při scanování **Spring Framework automaticky detekuje třídy s anotací @Component**. Instance této třídy mají stejné jméno jako třídy, ale v lower case prvním písmenem. Také můžeme explicitně specifikovat jaké jméno se má používat - *value* argument anotace.

Podle čeho rozlišujeme co správně injectovat, aby nevznikaly konflikty:

1. Jménem - jméno bean je stejné jako jméno property jiného beanu
2. Typem
3. Kvalifikátorem `@Qualifier("fooFormatter")`

TŘI RŮZNÉ VZORY, JAK LZE OBJEKTU PŘIDAT EXTERNÍ REFERENCI JINÉHO OBJEKTU.

- VKLÁDÁNÍ **ATRIBUTY** - EXTERNÍ MODUL, KTERÝ JE DO OBJEKTU PŘIDÁN, IMPLEMENTUJE ROZHRAŇÍ, JEŽ OBJEKT OČEKÁVÁ V DOBĚ SESTAVENÍ PROGRAMU.
- VKLÁDÁNÍ **SETTER METODOU** - OBJEKT MÁ SETTER METODU, POMOCÍ NIŽ LZE ZÁVISLOST INJEKTOVAT.
- VKLÁDÁNÍ **KONSTRUKTOREM** - ZÁVISLOST JE DO OBJEKTU INJEKTOVÁNA V PARAMETRU KONSTRUKTORU JIŽ PŘI JEHO ZRODU.

```
public class Customer {  
    @Autowired  
    private Person person;  
    private int type;  
}
```

```
public class Customer {  
    private Person person;  
    @Autowired  
    public void setPerson (Person person) {  
        this.person=person;  
    }  
}
```

```
@Component  
public class Customer {  
    private Person person;  
    @Autowired  
    public Customer (Person person) {  
        this.person=person;  
    }  
}
```

# Dependency injection

```
@Component
public class BeanB1 implements
BeanInterface {
    //
}
@Component
public class BeanB2 implements
BeanInterface {
    //
}
```

```
@Component
public class BeanA {
    @Autowired
    @Qualifier("beanB2")
    private IBean dependency;
    ...
}
```