OMO

4 - Creational design patterns

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

Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, *david.kadlecek@dnai.ai*

Co jsou to design patterny

- **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 patter

"GANG OF FOUR" NÁVRHOVÉ VZORY

Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides

- 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, ovladačů k zařízení, jednoho logování 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
}
```

Singleton - lze upravit

Tato verze funguje i v prostředí, kde běží paralelně více threadů a blokování synchronním zámkem je minimalizováno. Použit tzv. ***Double checked locking*** pattern.

```
class ClassSingleton {
      private static volatile ClassSingleton INSTANCE;
      private static final Object object = new Object();

      private ClassSingleton() {
      }

      public static ClassSingleton getInstance(){
            if (INSTANCE != null) {
                return INSTANCE;
            }

            synchronized (object) {
                  if (INSTANCE == null) {
                        INSTANCE = new ClassSingleton();
                  }
                  return INSTANCE;
            }
      }
      // getters and setters
}
```

Pozn. The volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory. So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in an int or a boolean variable then you can declare them as volatile variable.

*If we do not make the **INSTANCE** variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of **INSTANCE** as not null and they will believe its still null.*

Simple Factory

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

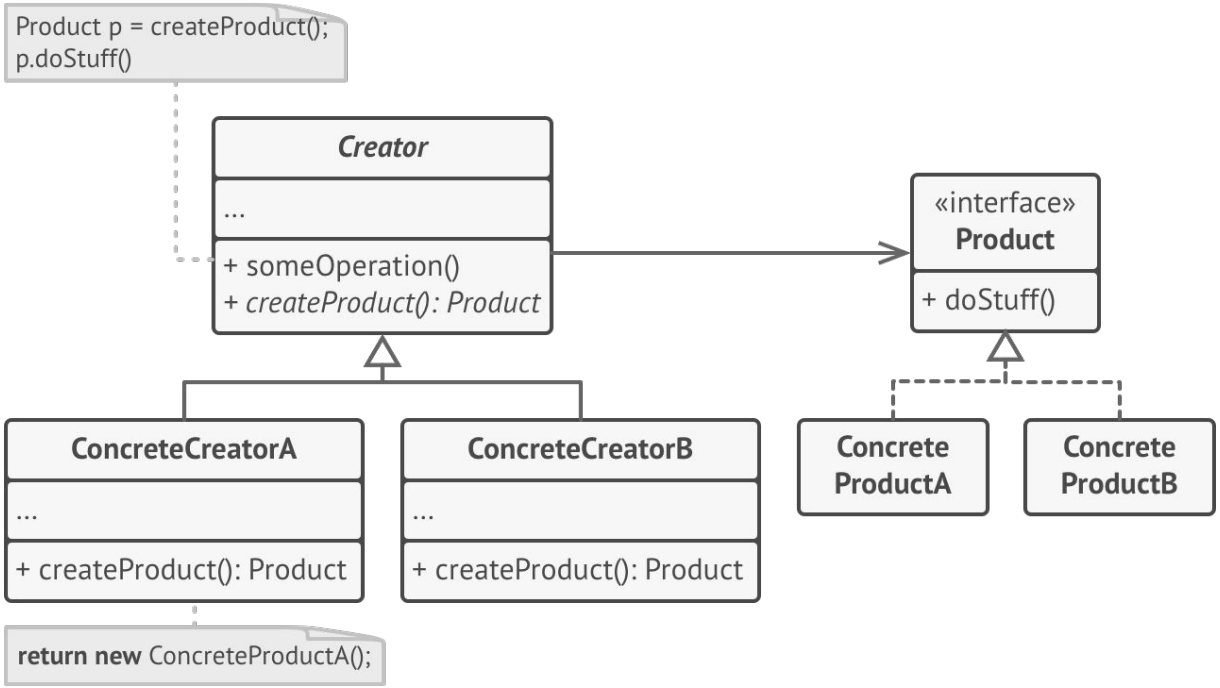
```
class PlantFactory {
     public Apple makeApple(int size, Boolean hasGoodTaste) {
          // Code for creating an Apple here.
     }
     public Orange makeOrange(int size) {
          // Code for creating an orange here.
     }
 }

...
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 můžu poskytout 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 konstruktoru objektu se volá faktory metoda, která je implementována/nebo redefinována v potomkovi.



Factory Method

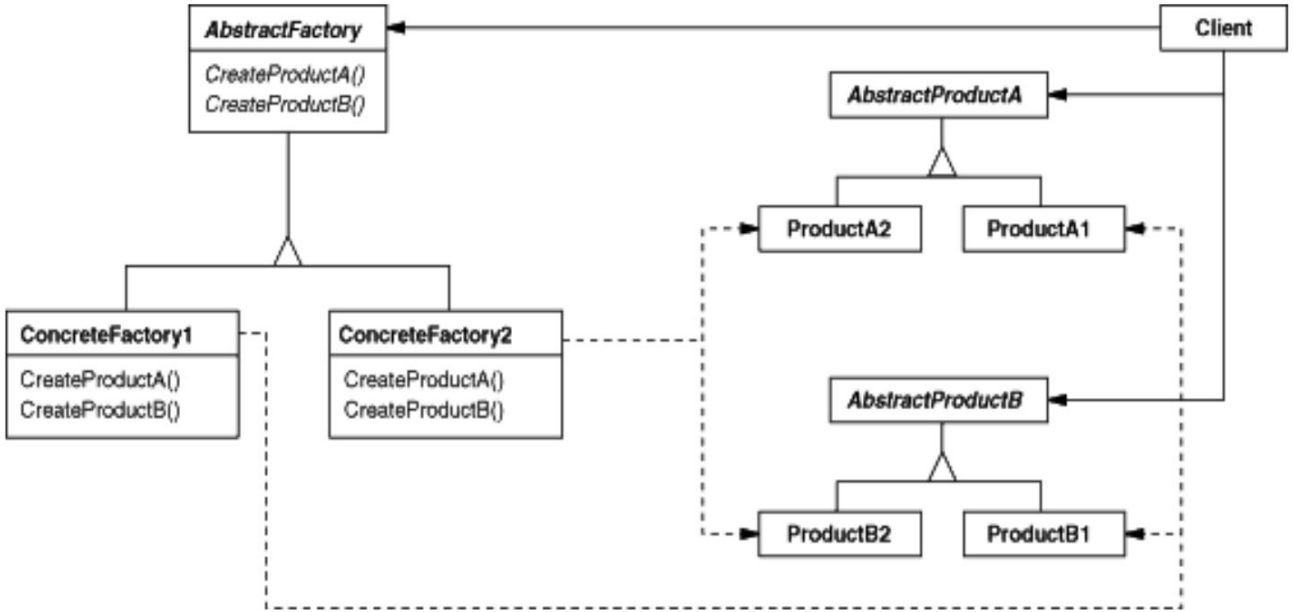
Druh objektu i jeho počáteční vlastnosti jsou dané přijatými parametry, případně i stavem objektu, který tovární metodu poskytuje.

Používá se tam, kde je vytvářený objekt nějakým způsobem odvozen 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 celé rodiny 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 oddděděné 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:

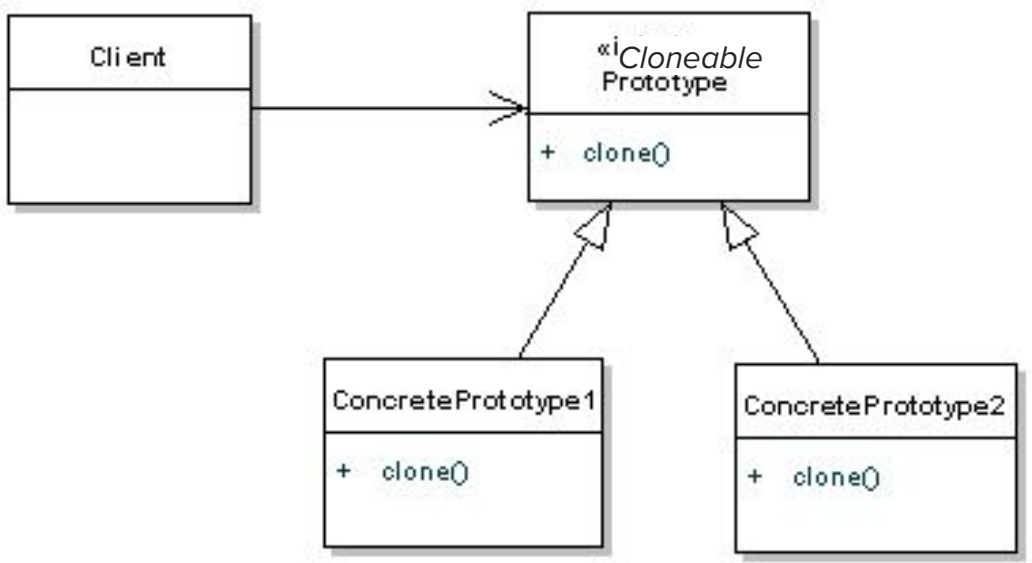
```
PlantFactory appleFactory = new AppleFactory();
PlantFactory orangeFactory = new OrangeFactory();
Plant apple = appleFactory.makePlant();
...
```

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

```java
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);
    }
}
```

```java
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:

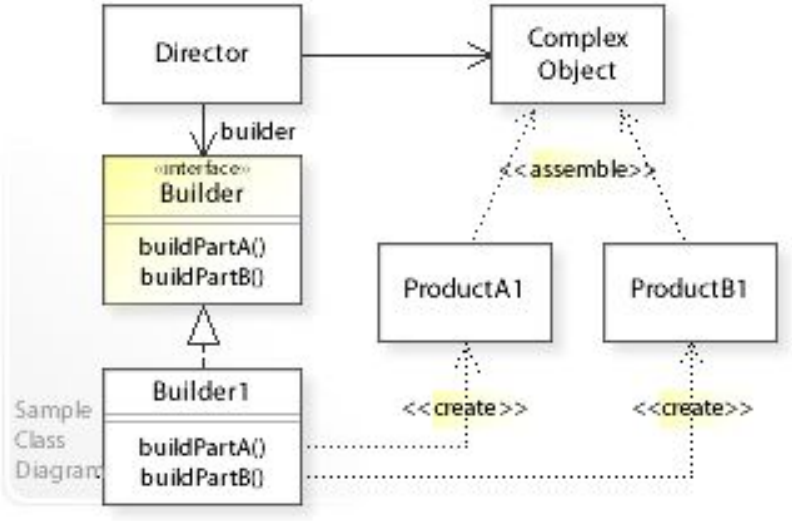
```java
...
Employee employee = new Employee("Karel Novak", new Address("Karlova 12"));
Employee employeeCopy = (Employee) employee.clone();
```

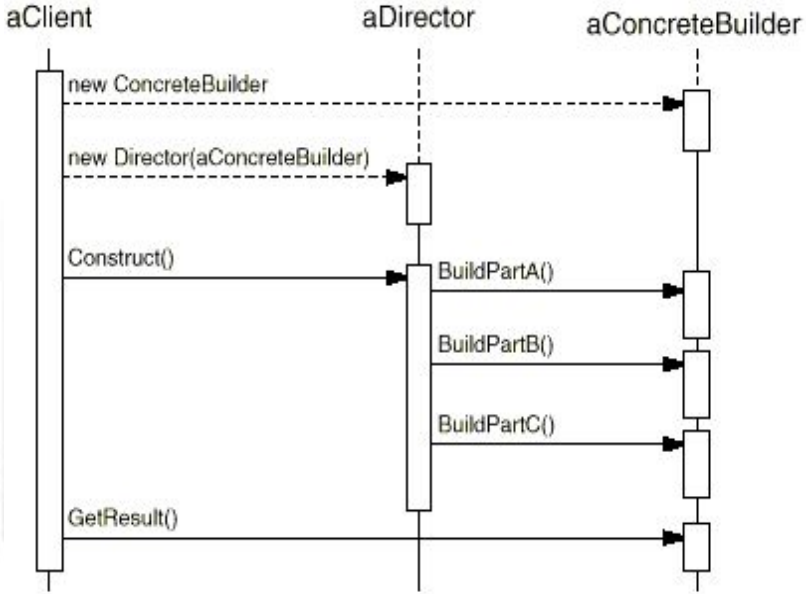
Builder

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

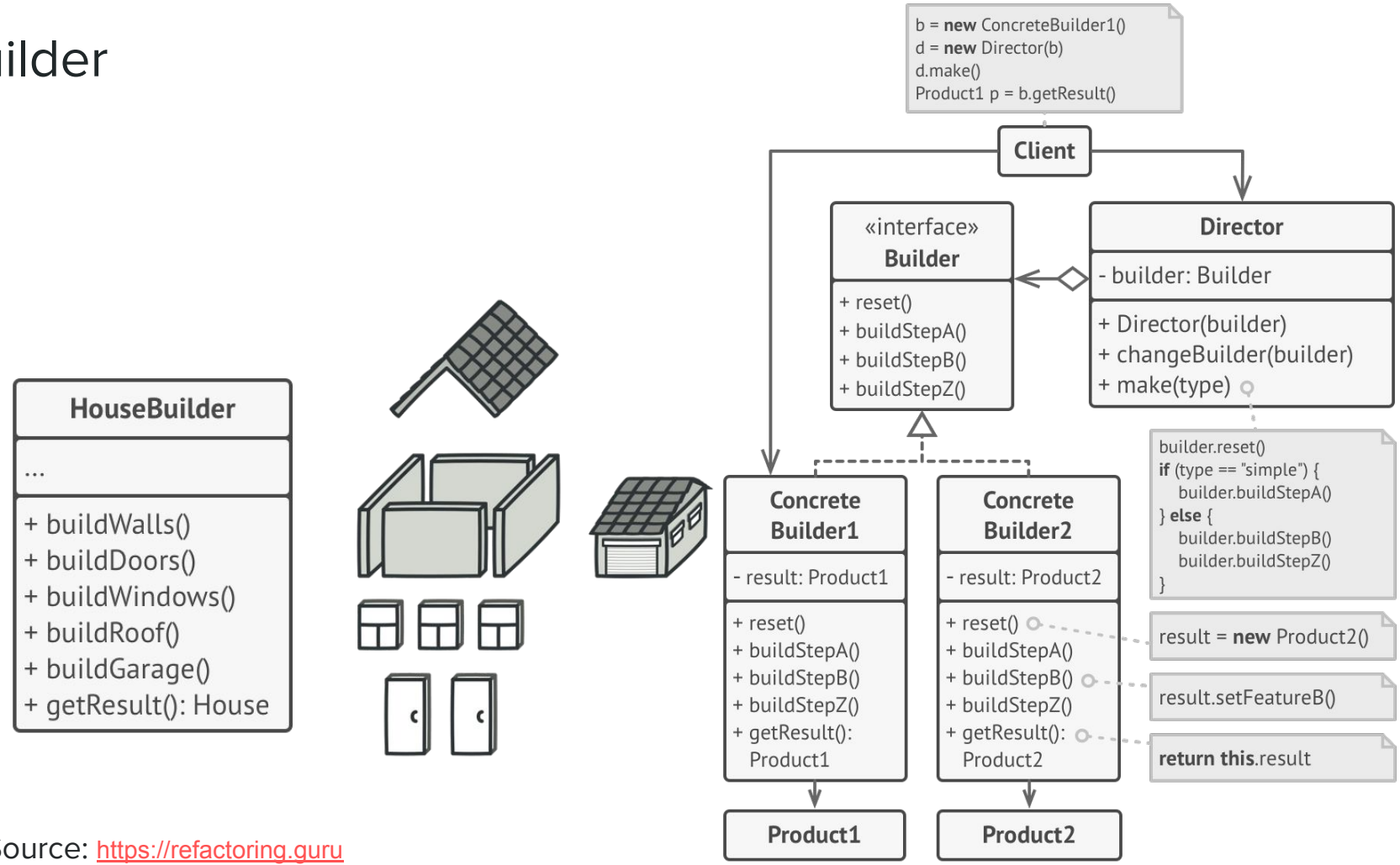
Výhody - možnost prohazování interních reprezentací, mohu ovládat proces konstrukce (aplikací různých kroků v různém pořadí sestavovat různé objekty) - narozdíl od Factory, kde zadám parametry finálního objektu.

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





Builder



Source: https://refactoring.guru

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 rozhraním

```
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 mohu hned v řetězci volat další metodu

Builder - příklad s fluent rozhraním

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

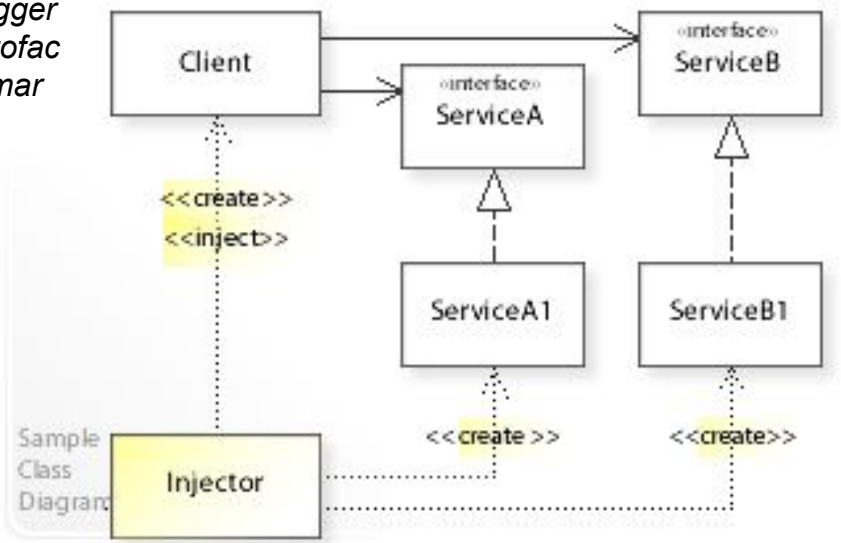
<<= Fluent interface

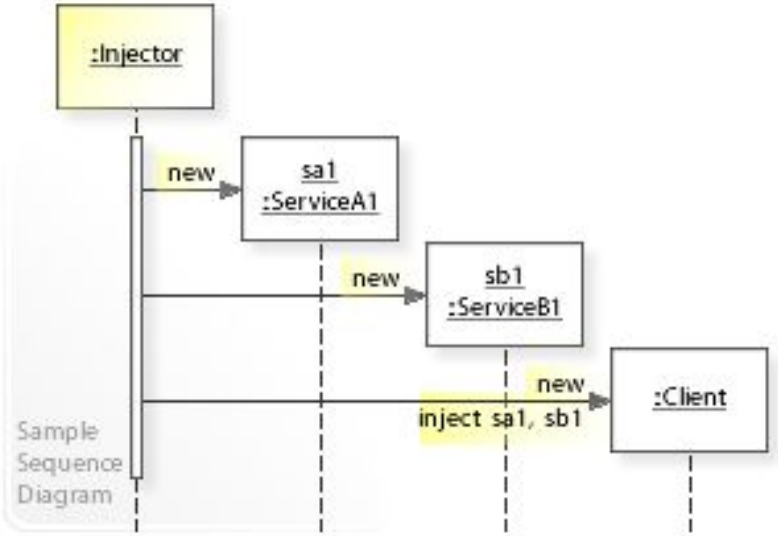
celou konstrukci provedu jedním zřetězeným voláním

Dependency injection

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.

Příklady frameworků implementujících DI:
Spring
google/guice
Dagger
Autofac
Lamar





Dependency injection

Zjednodušeně řečeno, namísto toho, abych při vytváření objektů musel volat konstruktory jednotlivých tříd *(new Xyz(...))* a do konstruktorů ručně vkládal reference na objekty, které do nich vstupují jako parametry (alternativně ručně volal setter metody a do nich vkládal reference na tyto objekty), tak to vše za mě udělá DI framework. DI framework se orientuje podle typů parametrů, které vstupují do konstruktorů nebo setter metod a použije existující instance odpovídajících typů.

DI framework si tedy oscanuje třídy ve zvolených adresářích, z nich udělá instance a ty propojí.

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

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

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

```
@Component
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

@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 třídal, 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 jmémo property jiného beanu
2. Typem
3. Kvalifikátorem *@Qualifier("fooFormatter")*

Dependency injection

```java
@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");
    }
}
```

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

```java
ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
Car car = context.getBean(Car.class);
```

Dependency injection

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

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