OMO

4 - Creational design patterns A

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

Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, *david.kadlecek@cz.ibm.com*

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.

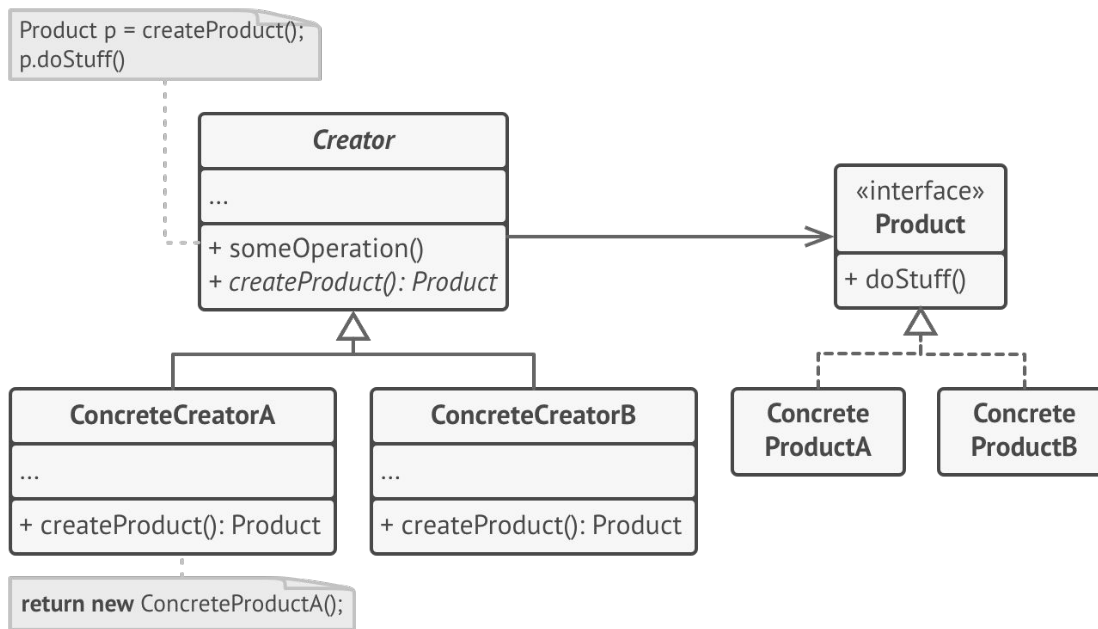
```
class PlantFactory {
     public Apple makeApple(int size, Boolean hasGoodTaste) {
          // Code for creating an Apple here.
     }
     public Orange makeOrange(int size) {
          // Code for creating an orange here.
     }
}

...
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 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 odvozený od aktuální instance třídy, která Factory Method poskytuje. Tyto metody se používají hlavně u immutable tříd.

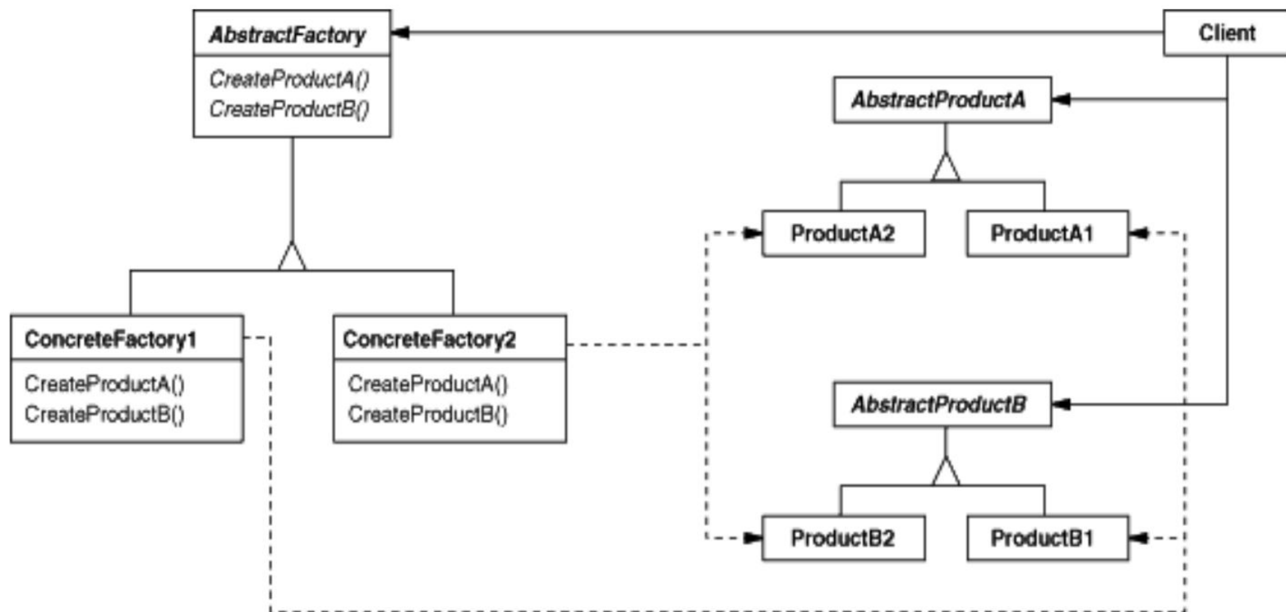
```
abstract class FruitCreator {
	protected abstract Fruit makeFruit(int size);
	public void pickFruit() {
		private final Fruit f = makeFruit();
	}
}

class OrangeCreator extends FruitCreator {
	@Override protected Fruit makeFruit(int size){
		if (size > 5)
			return new BigOrange();
		else
			return new SmallOrange();

	}
}
...
FruitCreator fruitCreator = new FruitCreator();
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 odděděné 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 {
      ...
}

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

Client:

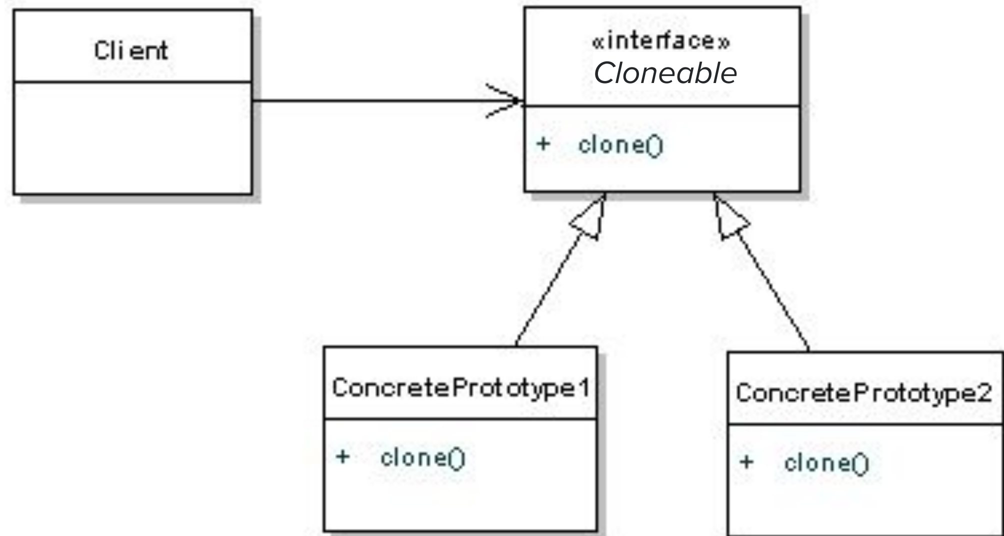
```
PlantFactory appleFactory = new AppleFactory();
PlantFactory orangeFactory = new OrangeFactory();
Plant apple = appleFactory.makePlant();
...
```

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();
```