

---

# Factory Method

---



# Úvod - problém

- Mějme obchod s auty:

```
public class OrderCars {
    public Car orderCar(String model) {
        Car car;

        if(model.equals("Mark IV")) {
            car = new Mercury(model);
        } else if(model.equals("Corvette")) {
            car = new Chevrolet(model);
        } else if(model.equals("Fusion")) {
            car = new Ford(model);
        } else if(model.equals("Enzo")) {
            car = new Ferrari(model);
        } else if(model.equals("Fabia")) {
            car = new Skoda(model);
        }

        car.buildCar();
        car.testCar();
        car.shipCar();
    }
}
```

Při přidání nového modelu je nutné upravit

Kód, který se nebude často měnit

- Problém – míchání relativně stálého kódu s kódem, který se bude měnit při každém přidání nové třídy
- Problém – míchání úrovní abstrakce – auto X Fabia



# Řešení předchozího problému

```
public abstract class CarFactory {
    abstract Car createCar(String model);

    public Car orderCar(String model) {
        Car car = createCar(model);
        car.buildCar();
        car.testCar();
        car.shipCar();
        return car;
    }
}
```

Vytvoření objektu –  
využívá polymorfismu

Business logika

```
public class FordFactory extends CarFactory {
    Car createCar(String model) {
        if(model.equals("Fusion")) {
            return new Ford(model);
        } else if(model.equals("Mark IV"))
            return new Mercury(model);
        } else {
            return null;
        }
    }
}
```

- Vytvoření „frameworku“ pro objednávání aut
- Při přidání nového auta => pouze změna `FordFactory.createCar`



# Factory Method

## ■ Známý jako

- Tovární metoda
- public Constructor

## ■ Účel

- Definuje rozhraní pro vytvoření objektu
- Potomci sami rozhodují kterou třídu instanciovat
- Umožňuje odložit instanciaci z předka na potomka

## ■ Kategorie

- Class Creational (Tvořivé návrhové vzory)

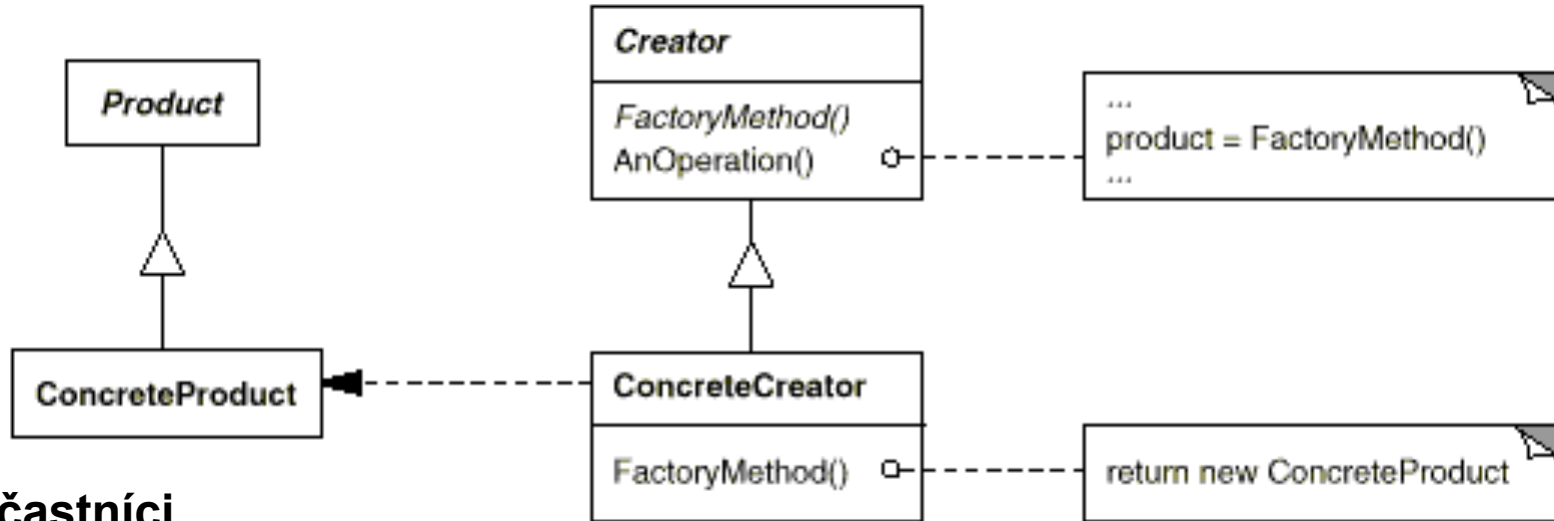
## ■ Využití

- Vytváření frameworků
  - Třída nezná objekty, které má vytvářet
  - Potomci specifikují vytvářené objekty
    - záleží na programátovi, kolik konkrétních potomků vytvoří
- Oddělení kódu, který se mění „často“, od „neměnného“



# Factory Method - struktura

## ■ Struktura

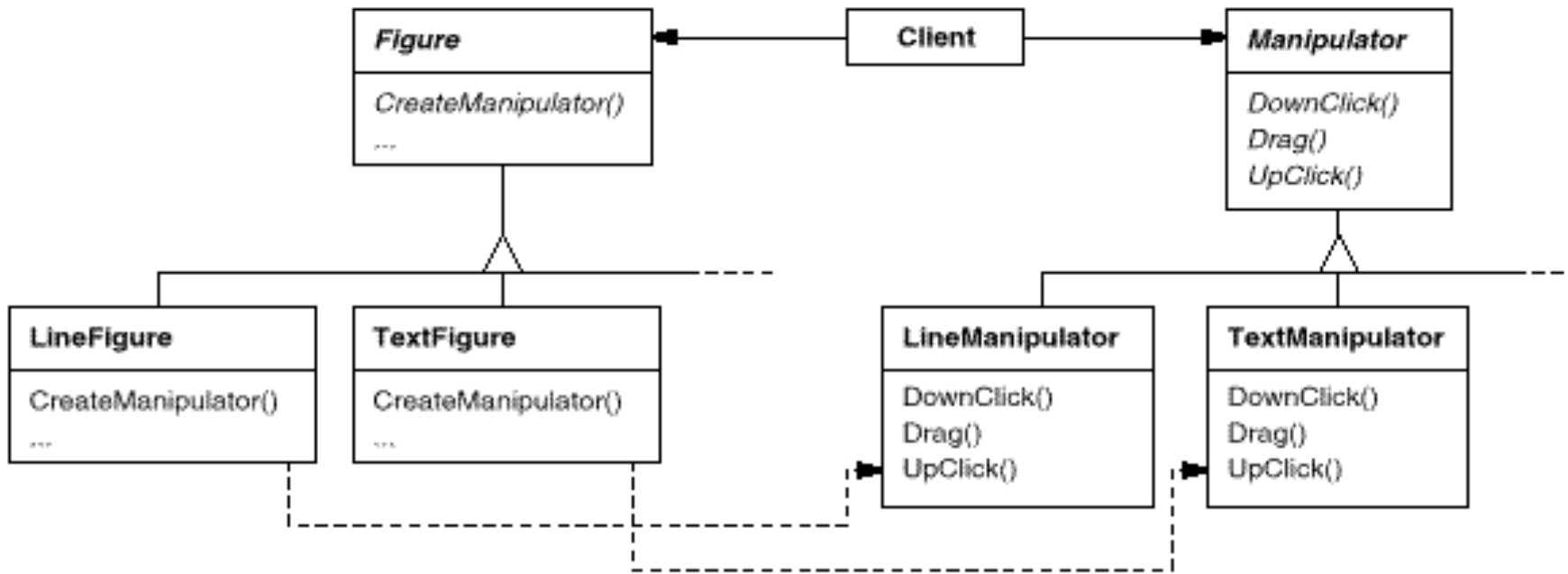


## ■ Účastníci

- Product (*Car*)
  - definuje rozhraní objektů vytvářených tovární metodou
- ConcreteProduct (*Fabia*)
  - implementuje rozhraní Productu
- Creator (*CarFactory*)
  - deklaruje tovární metodu vracející objekt typu Product
  - může definovat defaultní implementaci vracející defaultní objekt ConcreteProduct
- ConcreteCreator (*FordFactory*)
  - implementuje tovární metodu vracející instanci ConcreteProductu



# Factory Method - paralelní hierarchie



## ■ Propojení paralelních hierarchií tříd

- třída deleguje některé akce na jinou třídu
- příklad: manipulovatelné grafické objekty (úsečka, obdélník, kruh, ...)
  - stav manipulace je nutné udržovat pouze během manipulace - ne u grafického objektu
  - manipulace různých objektů uchovávají jiný stav (koncový bod, rozteč řádků, ...)
  - objekt `Manipulator`, konkrétní grafické objekty mají vlastní manipulátory
  - paralelní hierarchie
  - `Figure` deklaruje `CreateManipulator`, potomci vytvářejí vlastní manipulátory
  - částečně paralelní hierarchie - dědění defaultního manipulátoru



# Factory Method - implementace

## ■ Implementace

- dvě hlavní varianty
  - Creator je abstraktní třída, nemá vlastní implementaci tovární metody
    - nutnost dědičnosti a vlastní implementace
  - Creator je konkrétní třída s defaultní implementací
    - flexibilita
    - pravidlo: Vytvářet objekty v separátní metodě => potomci mohou ovlivnit (polymorfismus), které třídy se budou instanciovat.
- parametrizované tovární metody
  - variace - více druhů produktů
  - tovární metoda dostane parametr identifikující druh objektu (ProdId)
  - všechny objekty sdílejí rozhraní Productu

```
class Creator {  
  
Product create (ProdId id) {  
    if(id==MINE)    return new MyProd;  
    if(id==YOURS)  return new YourProd;  
    return 0;  
}  
}
```

```
class MyCreator extend Creator {  
    Prod create (ProdId id) {  
        if(id==MINE)    return new MyProd2;  
        if(id==THEIRS)  return new TheirProd;  
        return super.create(id);  
    }  
}
```

Obsluha ostatních id se  
deleguje na předka



# Factory Method - implementace

## □ pozdní inicializace

- tovární metody jsou vždy virtuální, často čistě virtuální - pozor na volání z konstruktorů
- lazy initialization - přístup pouze přes přístupovou metodu (accessor)
- konstruktor Creatoru inicializuje ukazatel na 0
- v případě potřeby se produkt vytvoří 'na žádost'

```
class Creator {  
    private Product product;  
  
    public final Product getProduct() {  
        if (product == null) {  
            product = createProduct();  
        }  
        return product;  
    }  
  
    protected Product createProduct();  
}
```

accessor

inicializace  
na žádost





# Factory Method - implementace

- použití šablon C++
  - někdy zbytečně pracné vytvářet konkrétní potomky
  - šablona parametrizovaná Produktem
  - není zapotřebí vytvářet potomky Creatora

vrací konkrétního potomka  
abstraktního produktu

```
class Creator {
public:
    public Product* CreateProduct() = 0;
};

template <class T>
class StdCreator: public Creator {
public:
    public Product* CreateProduct();
};

template <class T>
Product* StdCreator<T>::CreateProduct() {
    return new T;
}

class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StdCreator<MyProduct> myCreator;
```



# Factory Method - implementace

## □ použití šablon Java

- někdy zbytečně pracné vytvářet konkrétní potomky
- šablona parametrizovaná Produktem
- není zapotřebí vytvářet potomky Creatora
- reflexe nejsou tak hezké

```
class Creator {
    public Product createProduct() {}
};

class StdCreator<T extends Product> extends Creator {
    private Class<T> clazz;

    public StdCreator(Class<T> clazz) {
        this.clazz = clazz
    }

    public Product createProduct() {
        try{
            return clazz.newInstance()
        } catch (Exception) {
            // do not ever do this!!!!
            // Unless u are a pervert!
        }
    }
};

class MyProduct extends Product {
    .
};

StdCreator<MyProduct> myCreator;
```



# Factory Method - bludiště

- CreateMaze má napevno použité třídy pro komponenty
- Factory Method umožní potomkům volbu komponent

```
class MazeGame {  
  
    public Maze createMaze();  
    // factory methods:  
    public Maze makeMaze() {  
        return new Maze90;  
    }  
    public Room makeRoom(int n) {  
        return new Room(n);  
    }  
    public Wall makeWall() {  
        return new Wall90;  
    }  
    public Door makeDoor(Room* r1, Room* r2) {  
        return new Door(r1, r2);  
    }  
};
```

- Tovární metody vrací příslušnou komponentu
- MazeGame obsahuje defaultní implementace



# Factory Method - bludiště

```
class MazeGame {
Maze createMaze () {
    Maze aMaze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door theDoor = makeDoor(r1, r2);
    aMaze.addRoom(r1);
    aMaze.addRoom(r2);
    r1.setSide(North, makeWall());
    r1.setSide(East, theDoor);
    r1.setSide(South, makeWall());
    r1.setSide(West, makeWall());
    r2.sSetSide ....
    return aMaze;
}
```

```
class BombedMazeGame extends MazeGame {
    public BombedMazeGame();

    public Wall makeWall() const
        { return new BombedWall; }
    public Room makeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

potomci implementují  
specializace různých komponent

```
class EnchantedMazeGame extends MazeGame {
    public EnchantedMazeGame();

    public Room* MakeRoom(int n) {
        return new EnchantedRoom(n, castSpell());
    }
    public Door* MakeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }
    protected Spell castSpell() {};
};
```



# Factory Method - bludiště

```
class MazeGame {
Maze createMaze () {
    Maze aMaze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door theDoor = makeDoor(r1, r2);
    aMaze.addRoom(r1);
    aMaze.addRoom(r2);
    r1.setSide(North, makeWall());
    r1.setSide(East, theDoor);
    r1.setSide(South, makeWall());
    r1.setSide(West, makeWall());
    r2.sSetSide ....
    return aMaze;
}
```

```
class BombedMazeGame extends MazeGame {
    public BombedMazeGame();

    public Wall makeWall() const
        { return new BombedWall; }
    public Room makeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

potomci implementují  
specializace různých komponent

```
class EnchantedMazeGame extends MazeGame {
    public EnchantedMazeGame();

    public Room makeRoom(int n) {
        return new EnchantedRoom(n, castSpell());
    }
    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }
    protected Spell castSpell() {};
};
```



# Factory Method – příklady použití

## ■ Qt gui framework (C++)

```
public Qmenu* QMainWindow::createPopupMenu()
```

## ■ Java API – XML parsers, DateFormat třída

- DocumentBuilderFactory, DocumentBuilder
- SAXParserFactory, SAXParser

```
DateFormat df1 = DateFormat.getDateInstance(DateFormat.FULL, Locale.FRANCE);
```

Factory  
method

## ■ Zend webový aplikační framework (PHP)

- Třída Zend\_Db – connector k různým typům databází

```
// require_once 'Zend/Db/Adapter/Pdo/MySQL.php';  
// Automatically load class Zend_Db_Adapter_Pdo_Mysql  
// and create an instance of it.
```

```
$db = Zend_Db::factory('Pdo_Mysql', array(  
    'host'      => '127.0.0.1',  
    'username' => 'webuser',  
    'password' => 'xxxxxxxx',  
    'dbname'   => 'test'  
));
```

Factory  
method

Pdo\_Sqlite,  
Pdo\_Mssql, Oracle,  
DB2, ...



# Factory Method – příklady použití

```
public static function factory($adapter, $config = array()) {
    // obtaining adapter class name
    $adapterName = strtolower($adapterNamespace . '_' . $adapter);

    // Load the adapter class. This throws an exception
    // if the specified class cannot be loaded.
    @Zend_Loader::loadClass($adapterName);

    // Create an instance of the adapter class.
    // Pass the config to the adapter class constructor.
    $dbAdapter = new $adapterName($config);

    // Verify that the object created is
    // a descendent of the abstract adapter type.
    if (! $dbAdapter instanceof Zend_Db_Adapter_Abstract) {
        // throwing Zend_Db_Exception
        require_once 'Zend/Db/Exception.php';
        throw new Zend_Db_Exception("Adapter class '$adapterName' does not "
            . "extend Zend_Db_Adapter_Abstract");
    }

    return $dbAdapter;
}
```

Factory  
method

Concrete  
Product

test of Product  
base class



# Factory Method - související návrhové vzory

## ■ Shrnutí

- flexibilita za relativně malou cenu
- obvyklá základní metoda zvýšení flexibility
  - public constructor
- časté využití jinými vzory

## ■ Abstract Factory

- často implementovaná pomocí továrních metod

## ■ Template Method

- často volány tovární metody

## ■ Prototype

- nevyžaduje dědění Creatora
- ... avšak vyžaduje navíc inicializaci produktové třídy