Seminář Java
X

Radek Kočí

Fakulta informačních technologií VUT

Duben 2011

Obsah

- Znovupoužitelnost
- Návrhové vzory
- Zásady programování

Návrhové vzory (design patterns)

Návrhové vzory

- základní sada řešení důležitých a stále se opakujících návrhů (výsledky skutečného používání)
- usnadňují znovupoužitelnost
- umožňují efektivní návrh (výběr vhodných alternativ, dokumentace, . . .)

Návrhový vzor

- nazývá, zabstraktňuje a identifikuje klíčové aspekty běžné návrhové struktury
- popisuje komunikující objekty a třídy upravené k řešení obecného návrhového problému
- vzor je šablona pro řešení, nikoli implemeptace problému!
  ”Každý vzor popisuje problém, který se neustále vyskytuje, a jádro řešení daného problému. Umožňuje toto řešení používat mnohokrát, aniž bychom to dělali dvakrát úplně stejným způsobem.”

Některé vzory si konkurují, některé vzory mohou používat pro svou implementaci jiné vzory

Návrhový vzor

Prvky návrhového vzoru

- název
  - krátký popis (identifikace) návrhového problému
- problém
  - popis, kdy se má vzor používat (vysvětlení problému, podmínky pro smysluplé použití vzoru, . . .)
- řešení
  - popis prvků návrhu, vztahů, povinností a spolupráce
  - nepopisuje konkrétní návrh, obsahuje abstraktní popis problému a obecné uspořádání prvků pro jeho řešení
- důsledky
  - výsledky a kompromisy (vliv na rozšiřitelnost, přenositelnost, . . .)
  - důležité pro hodnocení návrhových alternativ – náklady a výhody použití vzoru

Typy vzorů

Vzory se mohou týkat

- tříd
  - zabývají se vztahy mezi třídami a podtřídami (vztah je fixován)
- objektů
  - zabývaní se vztahy mezi objekty, jsou dynamičtější

Základní rozdělení vzorů

- tvořivý
  - zabývá se procesem tvorby objektů
- strukturální
  - zabývá se skladbou tříd či objektů
- chování
  - zabývá se způsoby vzájemné interakce mezi objekty či třídy
  - zabývá se způsoby rozdělení povinností mezi objekty či třídy

Jedináček (Singleton)

Účel

- třída může mít pouze jednu instanci
- tvořivý vzor – objekty
- Pzn.: možné řešení – statické metody $\Rightarrow$ nevýhody

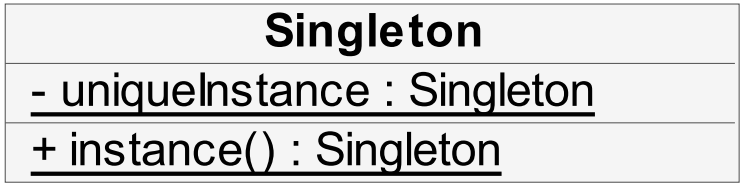
Motivace

- nutnost mít pouze jednu instanci (např. tiskové fronty)
- při pokusu o vytvoření nové instance se vrátí již existující

Důsledky

- řízený přístup k jediné instanci
- zdokonalování operací (dědičnost)
- usnadňuje změnu v návrhu (variabilní počet instancí)
- ...

Jedináček (Singleton)




```
public class Singleton {
    protected Singleton inst;

    private Singleton() {}

    // Tovarni metoda (Factory method)
    public static Singleton instance() {
        if (inst == null)
            inst = new Singleton();
        return inst;
    }
}
```

Abstraktní továrna (Abstract Factory)

Účel

- vytváření příbuzných nebo závislých objektů bez specifikace konkrétní třídy
- tvořivý vzor – objekty

Motivace

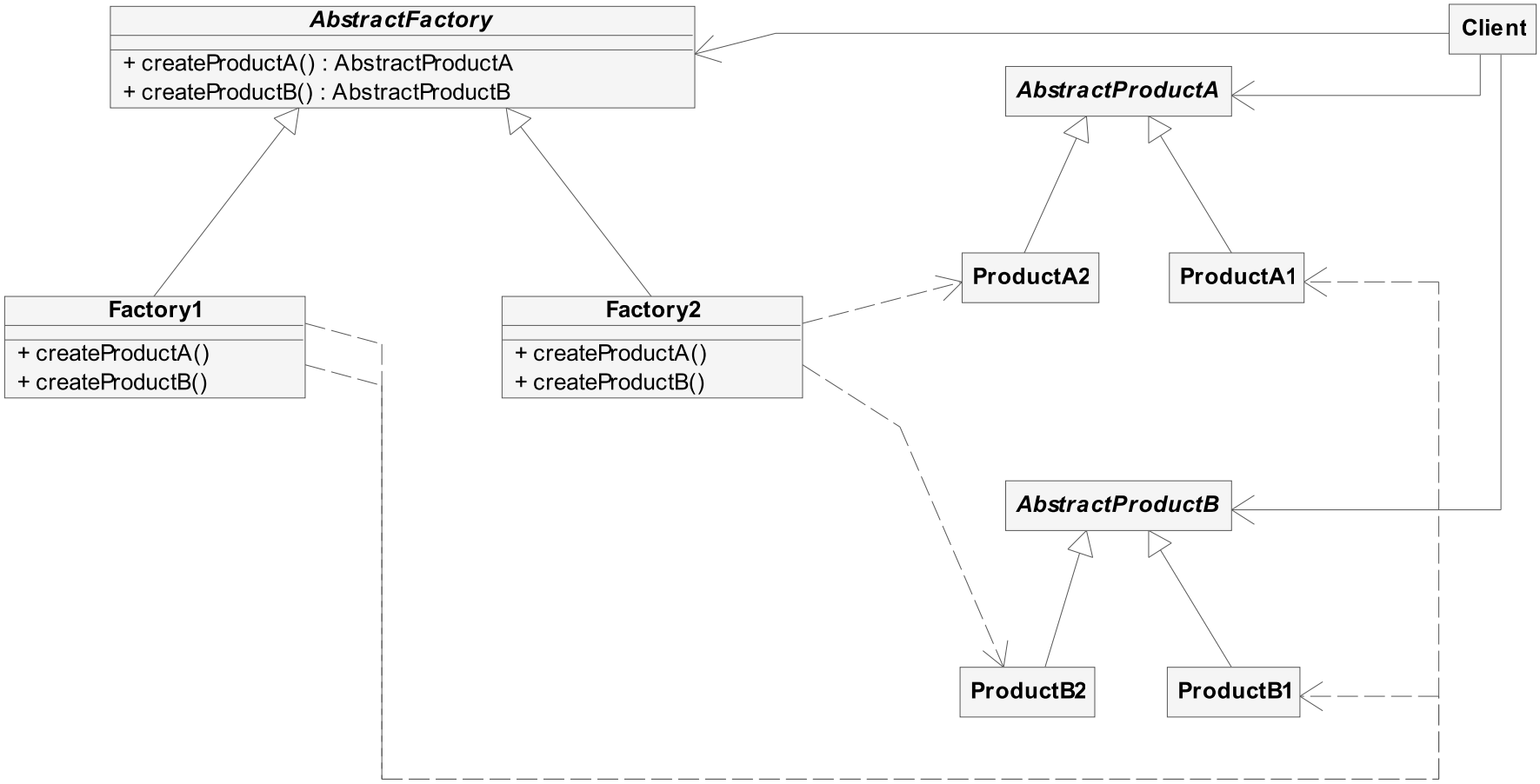
- např. změna vzhledu sady grafických nástrojů

Důsledky

- izoluje konkrétní třídy – klient pracuje pouze s rozhraním
- usnadňuje výměnu produktových řad (např. změna vzhledu, . . .)
- podpora zcela nových produktových řad je obtížnější
- . . .

Abstraktní továrna (Abstract Factory)

Struktura



Abstraktní továrna (Abstract Factory)

```
// abstract product
public interface Wall { ... }

// abstract factory
public abstract class MazeFactory {
    public abstract Wall makeWall();
}

public class MazeGame {
    public Maze createMaze(MazeFactory factory) {
        Wall wall = factory.makeWall();
        ...
    }
}
```

Abstraktní továrna (Abstract Factory)

```
// product
public class StdWall implements Wall { ... }

// factory
public class StdMazeFactory extends MazeFactory {
    public Wall makeWall() {
        return new StdWall();
    }
}

MazeGame game = new MazeGame();
MazeFactory factory = new StdMazeFactory();
game.createMaze(factory);
```

Abstraktní továrna (Abstract Factory)

```
// product
public class SpecialWall implements Wall { ... }

// factory
public class SpecMazeFactory extends MazeFactory {
    public Wall makeWall() {
        return new SpecialWall();
    }
}

MazeFactory specFactory = new SpecMazeFactory();
game.createMaze(specFactory);
```

Command

Účel

- zapouzdření požadavků nebo operací
- vzor chování

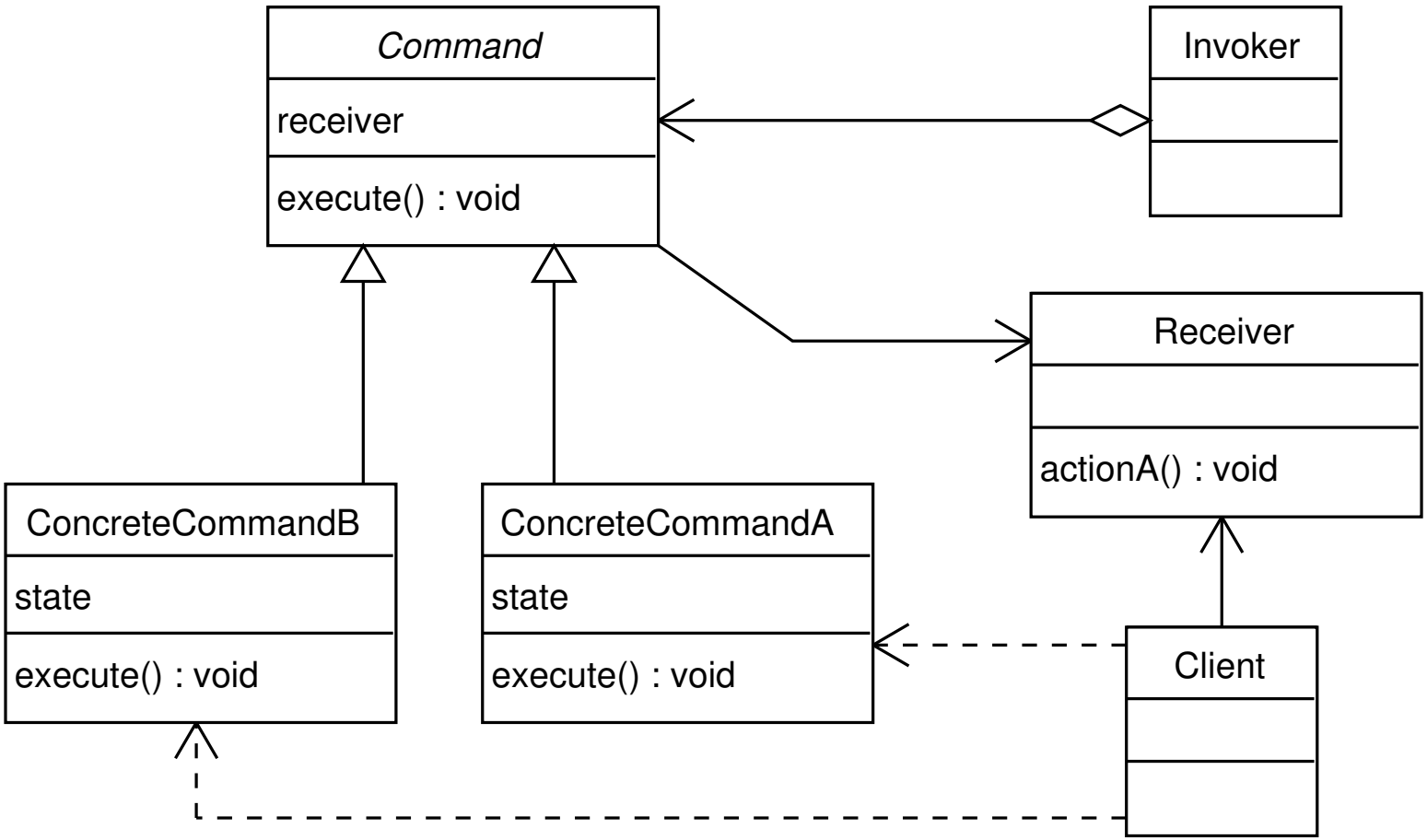
Motivace

- zaslání požadavku na obecné úrovni, aniž známe konkrétní protokol
- podpora *undo* operací

Důsledky

- reprezentuje jeden provedený příkaz
- umožňuje uchovávat předchozí stav klienta
- ...

Struktura


Command
receiver
execute() : void
Invoker
Receiver
actionA() : void
ConcreteCommandB
state
execute() : void
ConcreteCommandA
state
execute() : void
Client

Command

Návrhové vzory

Zdroje

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Návrh programů pomocí vzorů
  - popis 23 základních vzorů
- `http://objekty.vse.cz/Objekty/Vzory`
- `http://en.wikipedia.org`

Zásady programování

Co už známe

Překrývání metod třídy `Object`

- nefinální metody `equals`, `hashCode`, `toString`
- vždy když překryjete metodu `equals`, překryjte i metodu `hashCode`
- vždy překryjte metodu `toString`

Kompozice vs. dědičnost (znovupoužitelnost kódu)

- Dědičnost (třída je odvozena z jiné třídy)
  - narušuje zapouzdření (závislé na implementačních detailech)
- Kompozice (objekt je složen z jiných objektů)
  - metody jsou delegovány (nezávislé na implementačních detailech)
  - problém SELF

Duplicitní objekty

Duplicitní objekty

- objekty mající stejný stav
- většinou neměnné
- vytváření objektů je zbytečné

Opakované použití shodného objektu

- může být rychlejší (optimální využití paměti) a přehlednější
- většinou použitelné pro neměnné objekty

Duplicitní objekty

```
String s = new String("retez");
String s = "retez";
```

```
Map m = new HashMap();
Set s1 = m.keySet();
Set s2 = m.keySet();
```

```
// opakovane vytvareni objektu se stejnym stavem
Calendar c = Calendar.getInstance();
c.set(...);
```

Dupl.java

Duplicitní objekty

Vytváření malých objektů

- malá funkcionalita kontruktorů
- rychlé (moderní implementace JVM)

Vytváření nových objektů

- může zlepšovat jednoduchost nebo sílu programu

Kdy lze opakovaně použit objekty

- objekty jsou neměnné
- klesá výkon

Defenzivní kopie objektů

Defenzivní programování

- předpoklad, že klienti vaší třídy se pokusí zničit její invarianty
- neodkrývat interní prvky objektů
- před uložením provést defenzivní kopii
- před vrácením provést defenzivní kopii

`DefCopy.java`

Tovární metody místo konstruktorů

Získání instance třídy

- konstruktory
- statické tovární metody

Výhody továrních metod

- mají názvy
- nemusí vytvářet nový objekt při volání
- nemusí vracet instanci pouze volané třídy
- např. synchronizované kolekce, ...

Nevýhody

- těžko odlišitelné od jiných statických metod
- nutnost dodržování konvencí pojmenování

Tovární metody místo konstruktorů

```
public static final Boolean TRUE =
                          new Boolean(true);
public static final Boolean FALSE =
                         new Boolean(false);

public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

```
public static <T> Set<T> synchronizedSet(Set<T> s)
{
    return new SynchronizedSet<T>(s);
}
```

Rozhraní a abstraktní třídy

Definice typu, který umožňuje více implementací

- rozhraní
- abstraktní třída

Porovnání

- třídy lze snadno přizpůsobit tak, aby implementovaly rozhraní
- rozhraní může definovat smíšený typ nezávislý na dědičnosti tříd
- rozhraní umožňují flexibilní a (typově) bezpečná vylepšení funkčnosti
- rozvíjet abstraktní třídu je jednodušší než rozvíjet rozhraní

Rozhraní a abstraktní třídy

Doporučení

- pro definici typů používejte (pokud to jde) vždy rozhraní
- změna implementace rozhraní pak znamená pouze změnu názvu konstruktoru (nebo tovární metody) bez nutnosti přepisovat další kód

Neveřejné atributy

Objekt by měl mít pod kontrolou změnu svých atributů

- třída by měla jen výjimečně definovat veřejné atributy
- přístup přes veřejné metody
- zajistit, aby přímá modifikace atributu nebyla možná

```
public static final Type[] VALUES = { ... };
// => prvky pole se mohou měnit!
```

```
private static final Type[] privateVALUES = { ... }

public static final List VALUES = Collections.
    unmodifiableList(Arrays.asList(privateVALUES));
```

Kontrola platnosti parametrů

- vždy kontrolujte platnost parametrů metod
- podmínky vždy dokumentujte

```
/**
 * Vraci BigInteger, jehoz hodnota je (this mod m).
 * @param m modulo, ktere musi byt kladne.
 * @return this mod m.
 * @throws ArithmeticException pokud m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("m <= 0.");
    ...
}
```

Přetěžování s rozvahou

- volba překryté metody (dědičnost) závisí na běhovém typu objektu (vybere se vždy ta nejspecifičtější varianta)
- volba přetížené metody se provádí při kompilaci

Přetěžování s rozvahou

```
public String classify(Set s)
    { return "Mnozina"; }
public String classify(List l)
    { return "Seznam"; }
public String classify(Collection c)
    { return "Neznama kolekce"; }

Collection[] test = new Collection[] {
    new HashSet(),
    new ArrayList(),
    new HashMap().values()
};
for (int i = 0; i < test.length; i++) {
    System.out.println(classify(test[i]));
}
```

Zastaralé odkazy na objekty

Paměťové úniky

- neúmyslné zachování již nepoužívaných objektů
- vyšší aktivita GC, vyšší spotřeba paměti
- nejsou zřejmé (špatně se odhalují)

Zdroj

- správa vlastní paměti
- cache

`Stack.java`

Zastaralé odkazy na objekty

Řešení

- nastavit `null`
- opakované použití proměnné
- definovat proměnnou v nejmenším možném oboru platnosti

Zvýšení výkonu (správa paměti)

Eliminace správy paměti

- zmenšit počet dočasných proměnných (v cyklech apod.)
- používat metody, které nevytvářejí dočasné objekty nebo nevracejí kopii objektu

```
String s = "55";
int i = new Integer(s).intValue();
int i = Integer.parseInt(s);
```

- místo řetězení + používat `StringBuffer`

`Str.java`

Optimalizace

Optimalizace se řídí dvěma zásadami:
1. Nedělejte ji.
2. Zatím ji nedělejte (pro experty); dokud nemáte dokonale jasné řešení.

– M. A. Jackson

Pár poznámek k optimalizaci

- snažte se psát *dobré*, nikoliv *rychlé* programy
- dobré programy lze dobře optimalizovat, chyby ve špatně napsaných optimalizovaných programech se hledají těžko
- měřte výkonnost před a po optimalizaci!
  - mnohdy má ”optimalizace” velmi malý nebo i negativní dopad na výkonnost
- identifikace problémového místa
  - nespoléhat se na intuitivní pohled (tady musí být problém)
  - profilovací nástroje

Dokumentační komentáře

Zásady programování

Zdroje

- Joshua Bloch: Effective Java (2nd Edition)
- Martin Fowler: Refactoring: Improving the Design of Existing Code