

Seminář Java

Návrhové vzory, Zásady programování

Radek Kočí

Fakulta informačních technologií VUT

Duben 2010

- Znovupoužitelnost
- Návrhové vzory
- Zásady programování

Dědičnost

- implementace třídy pomocí jiné (již existující)
- znovupoužitelnost bílé skříňky
- výhody a nevýhody
 - přímočaré použití, jednodušší úprava metod
 - statické
 - těsná vazba s nadřazenou třídou (problémy s modifikací)

Skládání

- nová funkce = poskládání již existujících objektů
- znovupoužitelnost černé skříňky
- výhody a nevýhody
 - dynamické
 - objekty se používají přes rozhraní
 - objekty lze za běhu zaměňovat (stejně typy)
 - menší, jednodušší a přehlednější návrh

Skládání – problém SELF

```
class A
{
    foo() { self.m(); }
    m() { print("Object A doing the job"); }
}
```

```
class B
{
    A a;
    foo() { a.foo(); }
    m() { print("Object B doing the job"); }
}
```

```
B b;
```

```
b.foo() => Object A doing the job.
```

Skládání – problém SELF

```
class A
{
    foo() { self.m(); }
    m() { print("Object A doing the job"); }
}
```

```
class B
{
    A a;
    foo() { a.foo(); }
    m() { print("Object B doing the job"); }
}
```

```
B b;
```

```
b.foo() => Object B doing the job.
```

Parametrizované typy (generické programování)

- *templates* v C++ (viz Standard Template Library – STL)
- *generics* v Java 5
- definují parametrizované typy
- má význam u staticky typovaných jazyků

```
interface Sink<T> { flush(T t); }  
<T> T writeAll(Collection<T> coll, Sink<? super T>  
{  
    T last;  
    for(T t : coll) {  
        last = t;  
        snk.flush(last);  
    }  
    return last;  
}
```

```
Sink<Object> s;  
Collection<String> cs;  
String str = writeAll(cs, s); // T je String
```


Objektově orientovaný návrh a programování

- *znovupoužitelnost?*
 - zajištění znovupoužitelnosti \Rightarrow obecný návrh
 - zajištění aplikovatelnosti na řešený problém \Rightarrow specifický návrh
 - *spor*
- ... *přesto*
 - proč nevyužít řešení, které již fungovalo
 - taková řešení jsou výsledkem mnoha pokusů a používání
 - \Rightarrow vzory pro řešení stejných typů problémů

Návrhové vzory

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

Návrhový vzor

- nazývá, abstrahuje a identifikuje klíčové aspekty běžné návrhové struktury
- popisuje komunikující objekty a třídy upravené k řešení obecného návrhového problému
- vzor je šablona pro řešení, nikoli implementace 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

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

Vzory se mohou týkat

- tříd
 - zabývají se vztahy mezi třídami a podtřídami (vztah je fixován)
- objektů
 - zabývání 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 třídami
 - zabývá se způsoby rozdělení povinností mezi objekty či třídami

Tvořivý

- Factory method
- Abstract Factory
- Singleton
- Prototype
- Builder

Strukturální

- Adapter (třída)
- Adapter (objekt)
- Decorator
- Facade
- Bridge
- Flyweight
- Composite
- Proxy

Chování

- Interpreter
- Iterator
- Visitor
- Memento
- Observer
- Mediator
- Command
- Chain of Responsibility
- State
- Strategy

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)

Singleton
<u>- uniqueInstance : Singleton</u>
<u>+ instance() : Singleton</u>

```
public class Singleton {
    protected Singleton inst;

    private Singleton() {}

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

Účel

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

Motivace

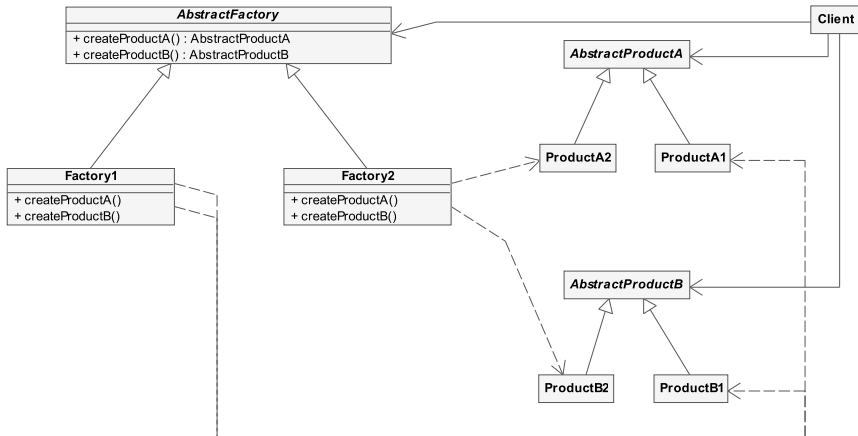
- např. změna vzhledu sady grafických nástrojů

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);
```

Účel

- zapouzdření požadavků nebo operací
- vzor chování

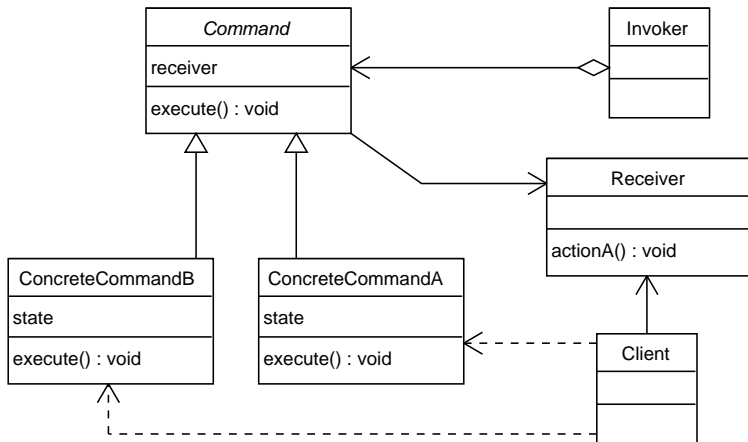
Motivace

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

Důsledky

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

Struktura



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

Překrývání metod třídy `Object`

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

Kompozice a dědičnost (znovupoužitelnost kódu)

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

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 vytvoreni objektu se stejnym stavem  
Calendar c = Calendar.getInstance();  
c.set(...);
```

Vytváření malých objektů

- malá funkcionalita konstruktorů
- rychlé (moderní implementace JVM)

Vytváření nových objektů

- může zlepšovat jednoduchost nebo sílu programu

Kdy lze opakovaně použít objekty

- objekty jsou neměnné
- klesá výkon

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

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);
}
```

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

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

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

```
/**
 * Vrací BigInteger, jehož hodnota je (this mod m).
 * @param m modulo, které musí být kladné.
 * @return this mod m.
 * @throws ArithmeticException pokud m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("m <= 0.");
    ...
}
```

- 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]));
}
```


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

Řešení

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

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 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

Zdroje

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