Modularita a Zapouzdření

OMO, LS 2013/2014

Modularita

- Umožňuje znovupoužívání kódu.

- Lze ji vnímat v různých rovinách abstrakce (metody, třídy, balíčky).

- Pokud je navržený program modulární, pak je obvykle dobře spravovatelný.

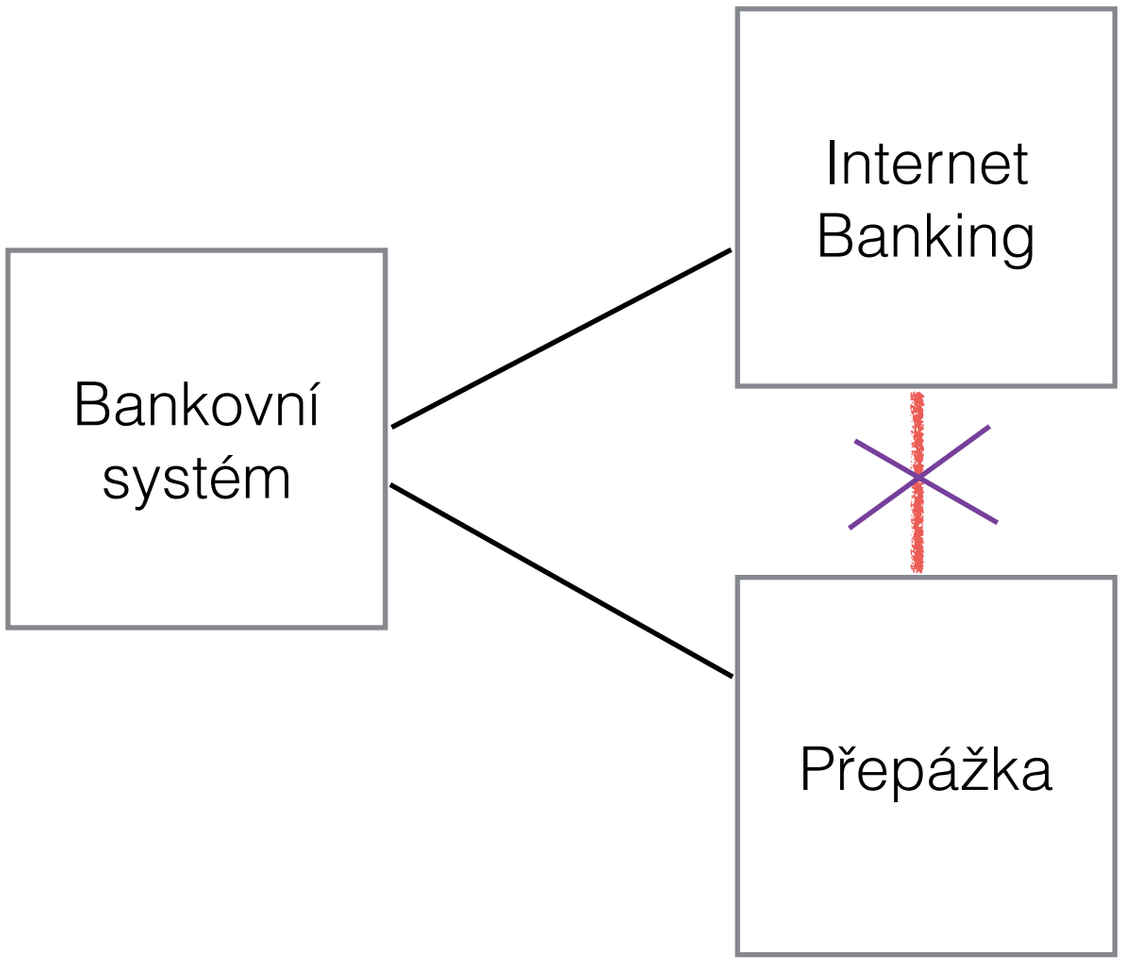
Modularita

- Umožňuje zaměnitelnost jednotlivých “modulů”.
- Výhodou je, že můžeme mít jednu architekturu, která pouhou výměnou jednotlivých modulů může plnit jinou funkci.
- Zpřehledňuje a snižuje množství kódu.
- Jednotlivé moduly by měly být na sobě co nejméně závislé (závislost pouze pomocí rozhraní).

Law of Demeter

- Pravidlo, které nám říká následující:
  - Každá jednotka by měla mít jen omezené znalosti o dalších jednotkách: pouze jednotky úzce spjaté s touto jednotkou.
  - Každá jednotka by měla mluvit pouze se svými přáteli a nemluvit s cizími jednotkami.
  - Mluvit jen s bezprostředními přáteli.
- Minimalizuje provázanost jednotlivých modulů.

Příklad: Modularita



Law of Demeter

- Z hlediska OOP můžeme toto pravidlo aplikovat pomocí následujících 5 pravidel pro posílání zpráv metodou M v objektu O
  - O sobě (this)
  - parametrům metody M
  - objektům, které byly v rámci metody M vytvořeny/instanciovány
  - komponentám, které jsou součástí O
  - globálním proměnným přístupným objektem O v rozsahu metody M

Zapouzdření

- Nám umožňuje řídit přístupová práva k metodám, třídám a objektům.

- Je založeno na předpokladu, že správné fungování tříd je založeno na řadě invariantů (vlastnostech, které platí po celou dobu existence objektu).

  - Programátor, který třídu používá nemůže tyto invarianty všechny znát.

  - Z tohoto důvodu je dobré programátorovi zamezit v přístupu k členským proměnným a metodám, které by mohly nesprávným pužitím tyto invarianty porušit.

Zapouzdření

- Zapouzdřením se snažíme zabránit problému zpřístupnění implementace (Representation Exposure).

Příklad : CharSet - Specifikace

// Overview: A CharSet is a finite mutable set of Characters

// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: $this_{post} = this_{pre} \cup \{c\}$
public void insert (Character c);

// modifies: this
// effects: $this_{post} = this_{pre} - \{c\}$
public void delete (Character c);

// returns: ($c \in this$)
public boolean member (Character c);

// returns: cardinality of this
public int size ( );

Implementace

```
class CharSet {
  private List<Character> elts
     = new ArrayList<Character>();
  public void insert(Character c)    {
     elts.add(c);
  }
  public void delete(Character c)    {
     elts.remove(c);
  }
  public boolean member(Cha
     return elts.contains(c)
  }
  public int size() {
     return elts.size();
  }
}
```

```
CharSet s = new CharSet();
Character a
     = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a))
     // print "wrong";
else
     // print "right";
```

Kde je chyba?

- Může být špatně metoda **delete**
  - Měla by odstranit všechny výskyty znaku.
- Může být špatně metoda **insert**
  - Neměla by vložit znak pokud již v Listu je.
- Měli bychom se to dozvědět z **invariantu reprezentace** datového typu.

Representation Invariant

- Specifikuje stav ve kterém platí, že je datová struktura dobře utvořená.
- Zachycuje informaci, která musí platit napříč všemi metodami.
- Můžeme ho specifikovat např. takto:

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  ...
```

Nebo formálněji:

$\forall$ indices i of elts . elts.elementAt(i) $\neq$ null
$\forall$ indices i, j of elts .
    i $\neq$ j $\Rightarrow \neg$ elts.elementAt(i).equals(elts.elementAt(j))

Nyní můžeme nalézt chybu

```
// Rep invariant:
// elts has no nulls and no duplicates

public void insert(Character c) {
   elts.add(c);
}

public void delete(Character c) {
   elts.remove(c);
}
```

Další ukázka invariantu

```
class Account {
    private int balance;
    // history of all transactions
    private List<Transaction> transactions;
    …
}
```

// real-world constraints:

balance ≥ 0

balance = Σ_i transactions.get(i).amount

// implementation-related constraints:

transactions ≠ null

no nulls in transactions

CharSet pokračování

- Uvažujme přidání následující metody do třídy CharSet

```
// returns: a List containing the members of this
public List<Character> getElts();
```

- S následující implementací

```
// Rep invariant: elts has no nulls and no duplicates
public List<Character> getElts()  { return elts; }
```

- Dodržuje implementace getElts() rep invariant?

Representation Exposure

- Uvažujme následující kód

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Zpřístupnění implementace je tedy externí přístup k členské proměnné.

- Zpřístupnění implementace je téměř vždy problém (může dojít k porušení invariantu)! Snažíme se mu vyhnout.

Způsoby jak zabránit Representation Exposure

1. Využít immutabilitu

```
Character choose() {
  return elts.elementAt(0);
}
```

Character is immutable.

2. Vytvořit kopii

```
List<Character> getElts() {
  return new ArrayList<Character>(elts);
  // or: return (ArrayList<Character>) elts.clone();
}
```

Mutating a copy doesn't affect the original.

3. Vytvořit immutable kopii

```
List<Character> getElts() {
  return Collections.unmodifiableList<Character>(elts);
}
```

Client cannot mutate

Literatura

- http://courses.cs.washington.edu/courses/cse331/11wi/lectures/lect05-af-ri.pdf