Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

PAZ1C

Róbert Novotný
robert.novotny@upjs.sk

22. 10. 2009

Kontrakty ako spôsob návrhu

- **kontrakt** je dohoda medzi klientom a triedou
- „ak klient splní záväzky, trieda splní očakávania klienta"
- kontrakt hovorí **„ČO"** trieda vykoná
  - implementácia triedy hovorí "AKO" sa to dosiahne
  - to však klienta nezaujíma
  - trieda je čierna skrinka

Interfejs = vyjadrenie kontraktu

- V Jave možno vyjadriť kontrakt pomocou metód interfejsu
- **Interfejs** = zoznam metód = zoznam schopností triedy
- ak trieda implementuje interfejs, hovorí, že dokáže poskytnúť danú schopnosť
- v Jave: kľúčové slovo **interface**

Príklad: dátová štruktúra Zásobník

- zásobník (***stack***) je zoznam, kde položky možno len vkladať na vrchol a odoberať z vrchola
  - **LIFO** – *last in-first out* (posledný dnu-prvý von)
- čo má poskytovať zásobník?
  - **push()** – vlož na vrchol zásobníka
  - **pop()** – vyber z vrchola
  - **isEmpty()** – je zásobník prázdny?
  - **size()** – počet prvkov v zásobníku

Interface k zásobníku


```
public interface Stack {
  void push(Object o);
  Object pop();
  boolean isEmpty();
  int size();
}
```

- interface udáva len hlavičky metód
- žiadne kučeravé zátvorky
- filozoficky zodpovedá triede
  - všetky metódy sú verejné (netreba písať public, ale robí sa to)

Interface k zásobníku


```
public interface Stack {
  void push(Object o);
  Object pop();
  boolean isEmpty();
  int size();
}
```

```
Stack zásobník = new Stack();
```

- nemožno vytvoriť inštanciu!
- veď zásobník nemá kód v metódach
- teda nie je jasné, **AKO** sa majú metódy vykonať
- to povie až trieda, ktorá bude tento interfejs **implementovať**

Implementovanie interfejsu

PAZ1C

```
public class ListBasedStack implements Stack {
  private ArrayList data = new ArrayList();
  public void push(Object o) { data.add(o); }
  public Object pop() {
    if(data.isEmpty()) {
      return null;
    }
   return data.get(data.size() – 1);
  }
  public boolean isEmpty() { return data.isEmpty(); }
  public int size() { return data.size(); };
}
```

Použitie v kóde

- naša trieda **ListBasedStack** **implementuje** interfejs **Stack**
- hovorí „**ako**" sa metódy budú správať
- použitie u klienta:

```
Stack zásobník = new ListBasedStack();
```

- dátový typ vľavo = **ČO** = aký kontrakt chceme používať
- dátový typ vpravo: = **AKO** = ktorá trieda nám ho poskytne

Zásada pre používanie interfejsov

Programujte vzhľadom k interfejsom, nie k ich implementáciám

Program to an interface, not to an implementation!

—*Gang of Four (Gamma, Helm, Johnson, Vlissides),* **Design Patterns**

Programujte vzhľadom k rozhraniam!

- vždy, keď je to možné, používajte pre premenné interfejsy, nie konkrétne implementácie
  - v parametroch metód
  - v návratových hodnotách
  - v lokálnych premenných na ľavej strane
- umožníte tým dodržať zásadu čiernej skrinky

Interfejsy a implementácie


```
public class SprávcaŠtudentov {
  private List<String> študenti = new ArrayList<String>();
  public List<String> getŠtudenti() {
    return študenti;
  }
}
```

- **interfejsy** v type inštančnej premennej
  - klienta zaujíma, že dostane zoznam študentov, nemusí ho zaujímať, ako je tento zoznam implementovaný (tu: **ArrayList** = zoznam nad poľom)
- **implementácie** pri konštruovaní

Akú to má výhodu?

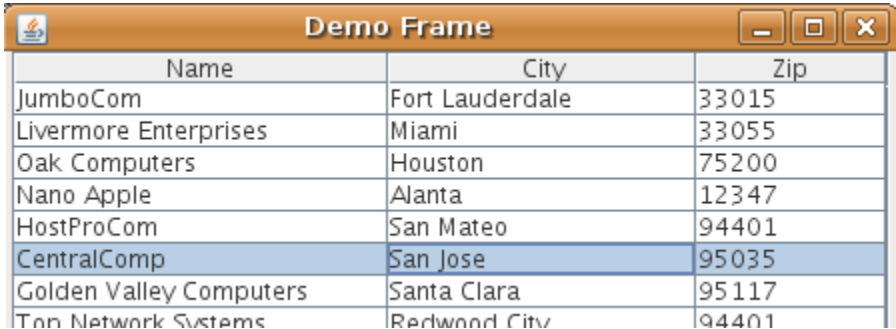
- ak všade používame implementácie a rozhodneme sa nahradiť ich, máme problém
  - všade používajme **ArrayListy**
  - lenže zrazu sa ukáže, že ich chceme nahradiť **CopyOnWriteArrayListom**, ktorý podporuje súčasný prístup z viacerých vlákien
- zbesile nahrádzanie v kóde
- **strašné** narušenie kompatibility
  - kód, ktorý používa naše triedy sa musí prepísať
  - „pokazia" sa oddedené triedy

Reálny príklad (aj majster tesár...)

- trieda **JTable**: ovládací prvok pre zobrazenie tabuľkových dát

```
public JTable(Vector rowData, Vector columnNames)…
```

- java.util.**Vector** – trieda pre zoznamy (sťa ArrayList)
- uvedenie implementačnej triedy robí z tohto konštruktora **nepoužiteľnú** (= zbytočnú) vec

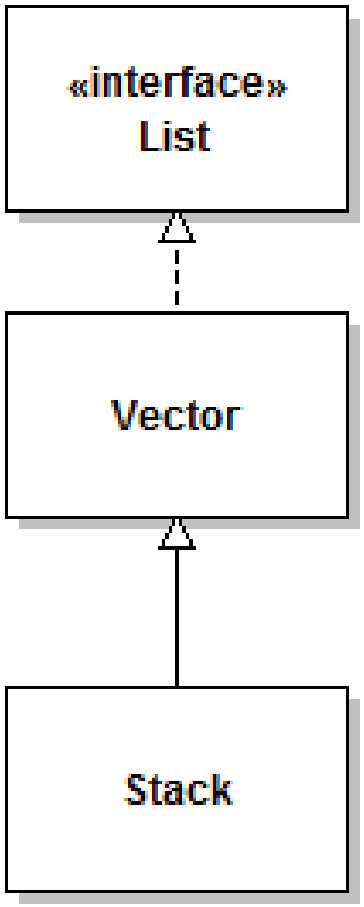


Kdepak udělali soudruzi zo Sunu chybu?

PAZ1C

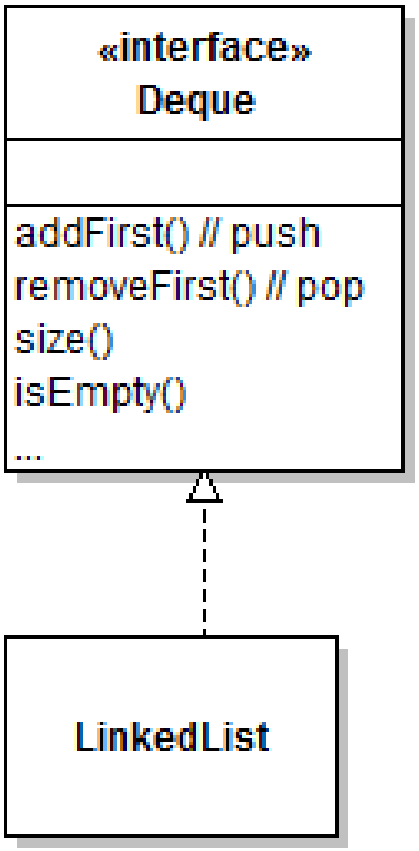
- dávno predávno (Java 1.0) existovala jediná trieda pre zoznamy: **java.util.Vector**
- Java 1.2 (1998): Joshua Bloch prekopal triedy pre kolekcie
- kopa interfejsov (**List**, **Set**, **Map**)...
- **Vector** sa stal zastaralým (namiesto neho **ArrayList**)
  - ešteže ArrayList implements List, Vector implements List
- žiaľ, konštruktor v **JTable** je nepoužiteľný, lebo očakáva implementáciu a nie interfejs
- ospravedlnenie: vtedy sa mnoho vecí nevedelo predvídať

Iný príklad utnutého majstra



- **Stack extends Vector**
- narúša to pravidlo „je"
  - „zásobník nie je vektorom založeným na poli"
- narúša to Liskovovej substitučný princíp
  - zásobník je zoznam, do ktorého možno vkladať len na koniec

Oprava chýb minulosti



- v JDK 6 – **Deque**
- interfejs pre „double-ended queue"
  - rad s dvoma koncami
- existuje viacero implementácií
  - **LinkedList**
  - **ArrayDeque**

Zásobník v čiernej skrinke spojového zoznamu


```
public boolean testujZatvorky(String vyraz) {
  Stack<Character> zasobnik = new Stack<Character>();
  for (int i = 0; i < vyraz.length(); i++) {
    if(vyraz.charAt(i) == '(') zasobnik.push('(');
    if(vyraz.charAt(i) == '[') zasobnik.push('[');
    try {
      if((vyraz.charAt(i) == ')')
        && (zasobnik.pop() != '(')) return false;
      if((vyraz.charAt(i) == ']')
        && (zasobnik.pop() != '[')) return false;
    } catch (EmptyStackException e) {
      return false;
    }
  }
  return zasobnik.empty();
}
```

Zásobník v čiernej skrinke spojového zoznamu

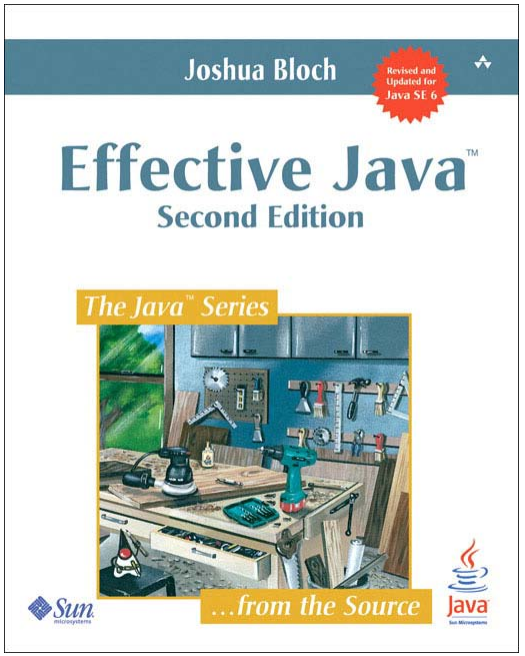

```
public boolean testujZatvorky(String vyraz) {
  Map<Character, Character> mapaZnakov
    = new HashMap<Character, Character>();
  mapaZnakov.put(']', '[');
  mapaZnakov.put(')', '(');
  Deque<Character> zasobnik = new LinkedList<Character>();
  for (char znak : vyraz.toCharArray()) {
    if(mapaZnakov.values().contains(znak)) {
      zasobnik.push(znak);
    } else {
      Character z1 = mapaZnakov.get(znak);
      Character z2 = zasobnik.poll();

      if(z1 == null || z2 == null || !z1.equals(z)) {
        return false;
      }
    }
  }
  return zasobnik.isEmpty();
}
```

Voláme metódy na interfejse!

Kolekcie a ich tragédia

- Joshua Bloch – *Effective Java*, 2. vydanie (2008)
  - krátka, ale geniálna kniha o zákutiach Javy
- príklad:
  - urobme množinu, ktorá si bude evidovať počet pridaných prvkov
  - „do tejto množiny bolo pridaných X prvkov"



Uprednostňujte kompozíciu pred dedičnosťou

- vieme, že dedičnosť narúša zapúzdrenie
- navyše dedičnosť **vyžaduje** znalosti o správaní rodičovskej triedy
  - hneď dáme príklad

Návrh kódu a logika uvažovania

- oddedíme z **HashSet**
- prekryjeme metódu **add()**
  - pripočítame jednotku
  - zavoláme rodičovskú metódu, ktorá pridá prvok
- musíme prekryť aj **addAll()** (pridanie kolekcie do množiny)
  - pripočítame toľko, koľko je prvkov v množine
  - zavoláme rodičovskú metódu, ktorá pridá prvky

Návrh kódu a logika uvažovania


```
public class InstrumentedHashSet<E> extends HashSet<E> {

  private int početPridaní = 0;

  public boolean add(E e) {
    početPridaní++;
    return super.add(e);
  }

  public boolean addAll(Collection<? extends E> c) {
    početPridaní += c.size();
    return super.addAll(c);
  }
}
```

Použitie triedy


```
Set<String> s = new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Baldrick", "Edmund", "Queenie"));
```

- lenže ak zistíme počet pridaných prvkov, zistíme, že máme výsledok **6**
- **prečo?** nik nevie
- pozrieme do zdrojákov!
  - ešteže ich Sun zverejňuje...
- vinník: metóda **addAll()** v **java.util.AbstractCollection**

Návrh kódu a logika uvažovania


```
public boolean addAll(Collection<? extends E> c) {
  boolean modified = false;
  Iterator<? extends E> e = c.iterator();
  while (e.hasNext()) {
    if (add(e.next())) modified = true;
  }
  return modified;
}
```

Metóda **addAll()** volá metódu **add()**!
Hm!

- započíta sa to dvakrát – raz v prekrytej metóde **addAll()**, ktorá zavolá našu prekrytú metódu **add()**

Čuduj sa svete, dokumentácia!

PAZ1C

addAll

```
public boolean addAll(Collection c)
```

Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)

This implementation iterates over the specified collection, and adds each object returned by the iterator to this collection, in turn.

Note that this implementation will throw an `UnsupportedOperationException` unless `add` is overridden (assuming the specified collection is non-empty).

Specified by:

`addAll` in interface `Collection`

Parameters:

`c` - collection whose elements are to be added to this col

Returns:

`true` if this collection changed as a result of the call.

Throws:

`UnsupportedOperationException` - if this collection
`addAll` method.

`NullPointerException` - if the specified collection is null.

See Also:

`add(Object)`

Z dokumentácie možno odvodiť toto správanie.

Čo ak dokumentácia nie je?

Návrh kódu a logika uvažovania


```
public boolean addAll(Collection<? extends E> c) {
  boolean modified = false;
  Iterator<? extends E> e = c.iterator();
  while (e.hasNext()) {
    if (add(e.next())) modified = true;
  }
  return modified;
}
```

Metóda **addAll()** volá metódu **add()**!
Hm!

- započíta sa to dvakrát – raz v prekrytej metóde **addAll()**, ktorá zavolá našu prekrytú metódu **add()**

Dedičnosť je bezpečná, ak...

- dedíte od tried v rovnakom balíčku
- máte dosah na implementáciu rodičovských tried
- dedíte od tried, ktoré boli explicitne navrhnuté na odddedenie a majú o tom dokumentáciu
- implementujete interfejs alebo tvoríte interfejs, ktorý dedí od iného interfejsu

Dediť od náhodných tried krížom cez balíčky môže viesť k nečakaným problémom!

Riešenie problému

- zrušiť pripočítavanie v metóde addAll
  - lebo sme si prečítali dokumentáciu
  - lenže my sa spoliehame na implementačný detail, ktorý sa môže zmeniť, a potom máme problém
- iné riešenie: prekryť si metódu addAll po svojom
  - prelez kolekciu, pridaj prvok cez (prekrytú) metódu add()
  - lenže časom môžeme zistiť, že kopírujeme kód z metódy rodičovskej triedy
- rozumné riešenie je použiť **delegáciu** (tretí spôsob skladania tried)

Delegácia = hybrid dedičnosti a kompozície

- trieda implementuje **interfejs**
- zároveň má inštančnú premennú konkrétnej triedy, na ktorú **deleguje** volania
- táto delegovaná trieda plní rolu rodičovskej triedy
- tie metódy delegovanej triedy, ktoré chce zmeniť (kvázi prekryť) upraví
- namiesto **super** používame delegovanú triedu

Nástrel delegácie


```
public class InstrumentedSet<E> implements Set<E> {
  private final Set<E> s;

  public InstrumentedSet(Set<E> s) { this.s = s; }

  public void clear() { s.clear(); }

  public boolean contains(Object o) {
    return s.contains(o);
  }
  public boolean isEmpty() {
    return s.isEmpty();
  }
  // .. ďalšie metódy
}
```

objekt, na ktorý budeme delegovať volania

v metóde delegujeme volanie na príslušný objekt

Nástrel delegácie - pokračovanie


```
public class InstrumentedSet<E> implements Set<E> {
  // private final Set<E> s;
  // pokračovanie z predošlého snímku

  private int početPridaní = 0;

  public boolean add(E e) {
    početPridaní++;
    return s.add(e);
  }

  public boolean addAll(Collection<? extends E> c) {
    početPridaní += c.size();
    return s.addAll(c);
  }
}
```

V metóde delegujeme volanie na príslušný objekt
Je to analógia **super.**

Použitie triedy


```
Set<String> set
   = new InstrumentedSet<String>(new HashSet<String>());
```

- množina bude delegovať volania na objekt typu **HashSet**
- množina **set** sa správa v metódach **add()** a **addAll()** podľa nášho želania, ostatné metódy sú delegované na **HashSet** v parametri konštruktora

Výhody a nevýhody delegácie

- Simulujeme dedičnosť
- Nemusíme sa spoliehať na neznáme správanie rodičovskej triedy
- Môžeme dodať funkcionalitu pre počítanie pridaných prvkov do ľubovoľnej triedy, ktorá implementuje **Set**
- Nevýhody:
  - treba vygenerovať kód pre delegovanie
    - rozumné IDE to spraví za nás
  - kód vyzerá na pohľad nepríjemne, ale je len zjavný