Prednáška 4: Aplikácia objektovo-orientovaných mechanizmov

Objektovo-orientované programovanie 2012/13

Valentino Vranić

Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

13. marec 2013

Obsah prednášky

Rôznorodosť OO prístupov

Korene OO programovania

- Prvý OO jazyk: Simula 67
  - Základ: procedurálny jazyk Algol 60
  - Autori jazyka prví použili pojem objekt v zmysle OO programovania: Ole-Johan Dahl a Kristen Nygaard[1]
- Smalltalk[2]
  - „Všetko je objekt“
  - Dôraz na spoluprácu objektov prostredníctvom posielania správ
- Vývoj OO prístupu
  - Z procedurálneho programovania (Simula, C++, Java...)
  - Z funkcionálneho programovania (CLOS – Common Lisp Object System)

[1] http://staff.um.edu.mt/jskl1/talk.html#History%20I
[2] http://www.smalltalk.org

Základný prvok OO programovania

- Objekt alebo trieda?
- Objekt má:[3]
  - stav – zahŕňa všetky vlastnosti objektu a ich hodnoty
  - správanie – ako objekt koná a reaguje v zmysle zmeny stavu a volania metód
  - identitu – každý objekt je jednoznačne identifikovateľný (nezamieňať objekt s jeho referenciou)
- Objekt je inštanciou triedy
- Programovanie založené na triedach: Simula, Smalltalk, C++, Java...

[3] G. Booch. Object-Oriented Analysis and Design with Applications. Addison-Wesley, 2nd edition, 1994.

Prototypovo-orientované programovania

- Len objekty – bez tried
- Dedenie je medzi objektmi: objekt sa odvádza od iného objektu – svojho *prototypu*
- Jazyky Self a JavaScript

Mechanizmy OO programovania

- Hlavné mechanizmy OO programovania:[4]
  - abstrakcia
  - zapuzdrenie (encapsulation)
  - modulárnosť
  - hierarchia
- Vedľajšie mechanizmy OO programovania:
  - typovosť
  - súbežnosť
  - perzistencia

[4] G. Booch. Object-Oriented Analysis and Design with Applications. Addison-Wesley, 2nd edition, 1994.

Abstrakcia

Abstrakcia

- Spôsob vysporiadania sa so zložitosťou
  - lat. *abstrahere* – vytiahnuť; opak: konkretizácia
- Abstrakcia označuje aj proces, aj abstraktné koncepty, ktoré jeho použitím vznikajú
  - Objekt v OOP ako abstrakcia objektu reálneho sveta
- Sústredenie na pohľad na objekt zvonka
- Odčlenenie správania objektu od jeho implementácie
- Tvorba abstrakcií: inkrementálny (po častiach) a iteratívny (opakovane vykonávaný) proces

Kvalita abstrakcie

- Zviazanosť (*coupling*) – čím voľnejšie triedy z hľadiska závislostí medzi nimi
- Súdržnosť (*cohesion*) – v triede majú byť veci, ktoré navzajom súvisia
- Dostatočnosť – trieda musí poskytovať všetky potrebné operácie
- Úplnosť (rozhrania) – rozhranie triedy má poskytovať všetky zmysluplné operácie
- Primitívnosť (operácií) – poskytnuté operácie majú byť primitívne (nie zamerané na komplexné prípady spracovania, ktoré sa dajú pokryť viacerými jednoduchými operáciami)

Zapuzdrenie

Zapuzdrenie (encapsulation)

- Skrývanie informácií – vnútorných detailov
- Štruktúru objektu a implementáciu jeho metód je väčšinou vhodné skryť
  - V OO jazykoch sa väčšinou sa dá riadiť – modifikátory prístupu v Jave
- Navonok má byť dostupné len rozhranie
  - Operácie prostredníctvom ktorých klientske objekty môžu s daným objektom interagovať
- Sprístupnenie vnútorných detailov sťažuje zmeny: klientsky kód môže od nich závisieť

Príklad: zmena vnútornej reprezentácie (1)

- Jednoduchá evidencia študentov:

```
public class Student {
    public String meno;
    public String priezvisko;
    ...
}
```

- Klientsky kód:

```
Student s = new Student();
s.meno = "Jan";
s.priezvisko = "Petrov";
```

- Čo ak sa rozhodneme zmeniť reprezentáciu?

```
public class Student {
    public String meno; // meno a priezvisko
}
```

Príklad: zmena vnútornej reprezentácie (2)

- Lepšie riešenie:

```
public class Student {
    private String meno;
    private String priezvisko;

    public void setMeno(String meno) { . . . }
    public void setPriezvisko(String priezvisko) { . . . }
    public String getMeno() { . . . }
    public String getPriezvisko() { . . . }
}
```

- Klientsky kód je nezávislý od vnútornej reprezentácie:

```
Student s = new Student();
s.setMeno("Jan");
s.setPriezvisko("Petrov");
```

Princíp otvorenosti a uzavretosti

- Skrývanie atribútov tried je dôsledkom všeobecnejšieho princípu otvorenosti a uzavretosti (open-closed principle)[5]:
  - Softvérové entity (triedy, moduly, funkcie atď.) majú byť otvorené pre rozšírenie, ale uzavreté pre zmeny.
- Tento princíp nie je obmedzený len na OOP
- Ale OOP prostredníctvom polymorfizmu ho umožňuje lepšie dodržať

[5] R. C. Martin. The Open-Closed Principle. C++ Report, 1996. http://www.objectmentor.com/resources/articles/ocp.pdf

Princíp otvorenosti a uzavretosti – príklad

- Grafické útvary, ktoré implementujú rozhranie `Kresleny`

```
void nakresliUtvary(Kresleny... k) {
    for (int i = 0; i < k.length; i++)
        k[i].nakresli();
}
```

- Otvorenosť – možno pridávať ďalšie útvary
- Uzavretosť – pre to nie sú potrebné zmeny v kóde na kreslenie

Modulárnosť

Modulárnosť

- Modul predstavuje ucelenú a vymedzenú časť kódu (príslušnými konštrukciami)
- Prostredníctvom modularizácie sa vysporadúvame so zložitosťou
- Moduly možno identifikovať na viacerých úrovniach – trieda, metóda alebo dokonca blok kódu
- Aj keď jeho význam je podstatne širší, pojem modulu sa bežne spája so súbormi
- V niektorých programovacích jazykoch súbor skutočne predstavuje modul – napr. v C a C++
- V Jave však ako moduly možno vnímať skôr balíky, a tie sú nezávisle od súborov

Trieda ako modul

- V OOP triedu možno vnímať ako modul
- Úzko súvisí s abstrakciou a zapuzdrením
- Závislosť zapuzdrenia od modularizácie je v Jave daná modelom riadenia prístupu, v ktorom významnú rolu hrá zaradenie triedy do balíka

Zviazanosť a súdržnosť modulov

- Moduly majú byť voľne zviazané a vysoko súdržne (high cohesion)
- *Voľná zviazanosť* (loose coupling) znamená minimalizáciu vzájomných väzieb medzi modulmi
- Bez väzieb by však nebolo programu
- Vhodné vymedzenie nevyhnútneho rozhrania triedy podstatne zníži potrebu zmien pri zmene vnútornej reprezentácie
- Všetok kód týkajúci sa jednej záležitosti by mal byť sústredený na jednom mieste – *vysoká súdržnosť* (angl. cohesion)
- V OOP nie je vždy možné dosiahnuť súdržnosť na uspokojivej úrovni – zvlášť príznačné pre tzv. pretínajúce záležitosti (vrátime sa k tomu na prednáške o aspektovo-orientovanom programovaní)

Hierarchia

Hierarchia

- Jednoduché dedenie
  - *is-a* – je (druhom)
  - Zovšeobecňovanie (generalizácia)/špecializácia
- Viacnásobné dedenie
  - Súčasťou jazyka C++; využitie pre tzv. *mixin* triedy
  - Problémy: kolízia názvov a opakované dedenie
  - Často sa dá nahradiť jednoduchým dedením a agregáciou
- Agregácia
  - *has-a* (*part-of*) – má (obsahuje)
  - Agregácia hodnotou a referenciou
- Pozicia v hierarchii určuje úroveň abstrakcie

Liskovej princíp substitúcie

- Najdôležitejšie kritérium pre použitie dedenia je či jestvuje vzťah typ-podtyp[6]
- O tom je Liskovej princíp substitúcie (Liskov substitution principle):[7]
  - Ak pre každý objekt o_1 typu S jestvuje objekt o_2 typu T taký, že pre všetky programy P definované v zmysle T správanie P je nezmenené, keď sa o_1 nahradí o_2, potom je S podtypom T.
- Príklady:
  - Typy geometrických útvarov
  - Typy zamestnancov (aj šéf, aj sekretárka sú zamestnanci)
- Veci však nie sú tak jednoduché v praxi
- Treba dbať o pohľad klientskeho kódu

[6] J. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.

[7] B. Liskov. Data abstraction and hierarchy, ACM SIGPLAN Notices 23(5), 1987.

Kedy *nepoužiť* dedenie

- Niekedy nie je možné nájsť takýto vzťah medzi typmi – vtedy dedenie netreba použiť
  - Príklad: zoznam a množina
  - Vzťah podobnosti (*is-like-a*) sa dá riešiť dedením od spoločnej triedy
- Niekedy láka nesprávne použitie dedenia na rozšírenie triedy za účelom jej zovšeobecnenia
  - Príklad: odvodenie elipsy od kruhu (pridaním ďalšieho stredu a polomeru)
- Toto neznamená, že odvodené triedy nesmú rozširovať základnú triedu – dôležitý je význam vzhľadom na princíp substitúcie
  - Možno všade kde sa očakáva kruh podsunúť elipsu?

Kruh a elipsa (1)

- Kruh je špeciálny prípad elipsy – matematicky
- Naozaj je vhodné použiť dedenie pre kruh a elipsu?[8]

```
public class Elipsa extends Utvar {
    private Bod f1;
    private Bod f2;
    private int a;
    private int b;

    public Elipsa(Bod f1, Bod f2, int a, int b) {
        ...
    }
    public Bod getF1() { return f1; }
    public Bod getF2() { return f2; }
    public void setF1(Bod b) {
        f1.setX(b.getX());
        f1.setY(b.getY());
    }
    public void setF2(Bod b) {
        f2.setX(b.getX());
        f2.setY(b.getY());
    }
    ...
```

Kruh a elipsa (2)

```
    ...
    public int getA() { return a; }
    public int getB() { return b; }
    public void setA(int d) { a = d; }
    public void setB(int d) { b = d; }
}
```

- Bod – pre úplnosť:

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
}
```

Kruh a elipsa (3)

- Časť údajov bude naviac, ale dá sa

```
public class Kruh extends Elipsa {
    public Kruh(Bod c, int r) {
        super(c, c, r, r);
    }
    public void setF1(Bod f1) {
        super.setF1(f1);
        super.setF2(f1);
    }
    public void setF2(Bod f2) {
        super.setF1(f2);
        super.setF2(f2);
    }
    public void setA(int d) {
        super.setA(d);
        super.setB(d);
    }
    public void setB(int d) {
        super.setA(d);
        super.setB(d);
    }
}
```

Kruh a elipsa (4)

- Čo však bude s klientskym kódom?
- Vďaka polymorfizmu môžeme všade kde sa očakáva elipsa použiť kruh
- Aj tam kde sa s tým nepočítalo:

```
public class C {
    public void magnify(Elipsa e, float m) { // m−násobné zväčšenie elipsy
        e.setA((int) (m * e.getA()));
        e.setB((int) (m * e.getB()));
    }
}
```

- Kruh sa však zväčší nie m-násobne, ale m^2-násobne!
- To, že sa nemyslelo na správanie odvodeného typu na mieste pôvodného, predstavuje porušenie Liskovej princípu substitúcie

Design by Contract (1)

- Liskovej princíp substitúcie súvisi s návrhom podľa zmluvy (design by contract)[9]
- V návrhu podľa zmluvy operácie vystupujú ako zmluvné strany
- Každá operácia definuje:
  - predpoklad (precondition)[10] – podmienka, ktorá musia platiť pred operáciou
  - dôsledok (postcondition)[11] – podmienka, ktorej platnosť zaručuje operácia po jej vykonaní, ak pred tým platil predpoklad
- Operácia garantuje, že ak platí predpoklad, po jej vykonaní bude platiť dôsledok

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[10] niekedy *vstupná podmienka*

[11] niekedy *výstupná podmienka*

Design by Contract (2)

- Príklad: operácia `setA()` triedy `Elipsa` by mala garantovať, že sa po jej aplikácii atribút `B` nezmení
- Pre elipsu podmienka bude dodržaná, ale pre kruh nie
- Došlo k zoslabeniu dôsledku
- Aby bol dodržaný Liskovej princíp substitúcie, pri podtypoch (prekonávanie):
  - predpoklad sa môže len zachovať alebo zoslabiť
  - dôsledok sa môže len zachovať alebo zosilniť

Design by Contract (3)

- Prekonávajúca operácia nemôže požadovať nič nad rámec toho, čo požaduje prekonaná operácia (predpoklad sa môže len zachovať alebo zoslabiť)
  - K prekonávajúcej operácií sa dá dostať cez prekonanú operáciu, lebo sa objekt podtypu môže vyskytnúť na mieste objektu nadtypu
  - Klientsky kód neoverí podmienky, ktoré musia platiť pred vyvolaním prekonanej operácie (ani ich nemusí poznať), a vyvolá ju
- Prekonávajúca operácia musí dodržať všetko, čo sa prekonaná operácia zaviazala splniť (dôsledok sa môže len zachovať alebo zosilniť)
  - Na mieste objektu nadtypu sa objaví objekt podtypu
  - Klientsky kód očakáva, že po vyvolaní prekonávajúcej operácie budú platiť podmienky, ktoré sa zaviazala splniť prekonaná operácia, a vyvolá ju

Typovosť a polymorfizmus

Typovosť

- Pojem typu primárne evokuje štruktúru údajov, ale o typoch sa dá hovoriť aj bez priameho uvádzania štruktúry – prostrednictvom *správania* (čo sa dá s inštanciami daného typu robiť)
- Abstraktné typy údajov
  - Spôsob algebraickej špecifikácie
  - Vstupno-výstupné deklarácie operácií + axiómy, ktoré pri ich volaní platia
- Trieda určuje typ, ale typ nemusí byť určený len triedou
  - Rozhrania v Jave
- Typovosť znamená, že ak je objekty rozdielneho typu možné zamieňať, tak len presne vymedzeným spôsobom
  - Podtyp, polymorfizmus, upcasting, downcasting
- Silná/slabá typovosť jazyka a beztypovosť

Typovosť (2)

- Viazanie (*binding*) názvov metód vo volaniach s určitým kódom
  - statické (skoré) – počas kompilácie
  - dynamické (neskoré) – v čase vykonávania programu
- Polymorfizmus

Polymorfizmus

- Prekonávanie metód: dynamické viazanie + dedenie
- Preťaženie metód: statické viazanie
- Parametrický polymorfizmus – v C++ (*templates*), ale aj v Jave 5.0
- Viacnásobný polymorfizmus; multimetódy v jazyku CLOS
- Idióm *double dispatch* pre C++ – platí aj pre Javu
  - Základ návrhového vzoru *Visitor*
- Neobmedzený (unbounded) dynamický polymorfizmus v Smalltalku – prítomnosť volanej metódy sa zisťuje až pri vykonávaní

Viacnásobný polymorfizmus v Jave

- Double dispatch na príklade
- Pracujeme s položkami, ktoré treba zobrazovať na rôznych zariadeniach
- Rôzne typy položiek: slová, zoznamy...
- Rôzne typy zariadení: horizontálny zobrazovač, vertikálny zobrazovač...
- Každá položka poskytuje metódu display() na vlastné zobrazenie
- Môžu pribudnúť ďalšie položky a zariadenia
- Pridanie ďalších zariadení nemá ovplývniť položky

Položky

```
public interface DItem {
    void display(DDevice d);
}

// slovná položka
public class WordItem implements DItem {
    String word;
    public WordItem(String s) {
        word = s;
    }
    public void display(DDevice d) { // double dispatch
        d.write(this);
    }
}
```

Položky (2)

```
// položká v tvare zoznamu
public class ListItem implements DItem {
   List list;
   public ListItem(List l) {
      list = l;
   }
   public void display(DDevice d) { // double dispatch
      d.write(this);
   }
}
```

Zariadenia

```
public interface DDevice { // zariadenia
    void write(WordItem item);
    void write(ListItem item);
}
// horizontálny zobrazovač
public class HorDevice implements DDevice {
    public void write(WordItem item) {
        for(int i = 0; i < item.word.length(); i++)
            System.out.print(item.word.charAt(i));
    }
    public void write(ListItem item) {
        for(int i = 0; i < item.list.size(); i++) {
            System.out.print(item.list.get(i));
            System.out.print(", ");
        }
    }
}
```

Zariadenia (2)

```
// vertikálny zobrazovač
public class VerDevice implements DDevice {
    public void write(WordItem item) {
        for(int i = 0; i < item.word.length(); i++)
            System.out.println(item.word.charAt(i));
    }
    public void write(ListItem item) {
        for(int i = 0; i < item.list.size(); i++) {
            System.out.print(item.list.get(i));
            System.out.println(", ");
        }
    }
}
```

Použitie

- Vytvoríme položku a zariadenie:

```
DItem it = new WordItem("word");
DDevice dev = new HorDevice();
```

- Vyvolanie zobrazenia:

```
it.display(dev); // double dispatch
```

- Double dispatch:

```
public void display(DDevice d) {
   d.write(this);
}
```

  - Najprv sa dynamicky rozhodne o metóde `display()` na základe skutočného typu objektu v referencii `it`
  - Následne sa v metóde `display()` dynamicky rozhodne o metóde `write()` na základe skutočného typu zariadenia

Sumarizácia

Sumarizácia

- Prehľad niektorých dôležitých princípov OO programovania
- Niektoré z týchto princípov sú platné aj všeobecnejšie
- V programoch reálnej zložitosti a v reálnom čase dostupnom na ich vývoj nie je možné úplne dodržať všetky princípy
- Miera ich uplatnenia však rozhoduje o flexibilnosti návrhu
- Tieto princípy sa často predkladajú ako pravidlá (usmernenia), idiómy alebo (návrhové) vzory
- Návrhovým vzorom bude venovaná jedna z nasledujúcich prednášok

Čítanie

- Dnešná prednáška: OJA, kapitola 6[12]
- Z dostupných materiálov môžete pozrieť ešte:
  - R. C. Martin. The Liskov Substitution Principle. C++ Report, 1996. `http://www.objectmentor.com/resources/articles/lsp.pdf`
  - R. C. Martin. The Open-Closed Principle. C++ Report, 1996. `http://www.objectmentor.com/resources/articles/ocp.pdf`
  - The Ellipse-Circle Dilemma. `http://ootips.org/ellipse-circle.html`
- Ďalšia prednáška bude o výnimkách, RTTI a zoskupeniach v Jave: OJA, kapitoly 8–10

[12] **Errata**: `http://fiit.stuba.sk/~vranic/pub/oja-err.pdf`.