Objektovo-orientované programovanie

Objekty, Java a aspekty

Valentino Vranić

Objektovo-orientované programovanie

Objekty, Java a aspekty

Slovenská technická univerzita
v Bratislave
2008

PUBLIKÁCIU PODPORILO ZDRUŽENIE

GRATEX IT INŠTITÚT

v rámci fondu GraFIIT www.gratex.com

Lektori: doc. Ing. Pavol Herout, PhD.
doc. Ing. Ján Kollár, PhD.

Vydala Slovenská technická univerzita v Bratislave vo Vydavateľstve STU, Bratislava, Vazovova 5.

Text neprešiel jazykovou úpravou vydavateľstva.

Schválilo vedenie Fakulty informatiky a informačných technológií STU v Bratislave pre študijný program Informatika a študijný program Počítačové systémy a siete.

ISBN 978-80-227-2830-0

Obsah

Zoznam obrázkov

Predhovor

Táto kniha vznikla na základe mojich prednášok z rovnomenného predmetu na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave. Ako vo svojich prednáškach, tak aj v tejto knihe som sa snažil vystihnúť princípy objektovo-orientovaného programovania, ale aj poskytnúť úplný obraz jedného objektovo-orientovaného programovacieho jazyka (aj keď kniha predsa predpokladá základné vedomosti z programovacieho jazyka C alebo iného podobného procedurálneho jazyka). Nie náhodou je ním Java, ktorej popularita stále stúpa, a ktorej poznanie je v súčasnosti pre programátora veľkou výhodou.

Hovoriť dnes o objektovo-orientovanom programovaní a nespomenúť aspektovo-orientované programovanie by znamenalo zamlčať najperspektívnejší smer jeho rozvoja. Preto som jednu kapitolu venoval programovaciemu jazyku AspectJ, ktorý predstavuje reálne používané aspektovo-orientované rozšírenie Javy.

Chcel by som poďakovať recenzentom za pozorné preštudovanie rukopisu a cenné pripomienky, ktoré sa týkali aj trochu odvážnejšej terminológie (predovšetkým v oblasti objektovo-orientovaného a aspektovo-orientovaného programovania). Mojim cieľom bolo čo najviac sa priblížiť pôvodnej anglickej terminológii. V texte vždy upozorňujem na ďalšie termíny, ktoré sa používajú pre daný pojem.

Ďakujem Gratex IT Inštitútu, ktorý podporil vydanie tejto publikácie v rámci fondu GraFIIT.

Bratislava, 14. január 2008 Valentino Vranić

1 Úvod

Základným cieľom tejto knihy je oboznámiť čitateľa s objektovo-orientovaným programovaním. Splnenie tohto cieľa vyžaduje uvedenie čitateľa do určitého objektovo-orientovaného programovacieho jazyka. V prípade tejto knihy je to programovací jazyk Java.

Skutočné pochopenie určitého spôsobu programovania vyžaduje jeho praktické vyskúšanie v kontexte ostatných čŕt daného programovacieho jazyka. Preto v mnohom tento text zachádza do detailov Javy, ktoré sa bezprostredne nespájajú s objektovo-orientovaným programovaním, ako sú napríklad anonymné triedy, typické a rozsiahlo využívané v Jave.

Táto kniha sa nevenuje konštrukciám na riadenie vykonávania programu a operátorom, ktoré Java prakticky prevzala z jazyka C. Nevysvetľuje ani tvorbu vnorenej dokumentácie, tvorbu appletov a mechanizmus anotácií, ktoré však nie sú kľúčové pre objektovo-orientované programovanie.

Na druhej strane, objektovo-orientované programovanie je potrebné vnímať v širšom kontexte. Preto táto kniha pootvára dvere do troch ďalších oblastí. Objektovo-orientované programovanie je úzko prepojené s objektovo-orientovanou analýzou a návrhom, ktoré sa už takmer výlučne uskutočňujú v jazyku UML. Preto sa tento text dotýka aj jazyka UML a približuje čitateľovi tie jeho vlastnosti, ktoré patria do repertoára základných vedomostí objektovo-orientovaného programátora.

Kniha sa venuje aj objektovo-orientovaným vzorom, ktorých poznanie je predpokladom úspešného vývoja objektovo-orientovaných aplikácií.

Prostredníctvom tejto knihy čitateľ bude mať príležitosť dozvedieť sa aj o aspektovo-orientovanom programovaní. Ide o prístup, ktorý významne rozširuje možnosti objektovo-orientovaného programovania. V knihe je prezentovaný v súvislosti s jazykom AspectJ, ktorý plynulo nadväzuje na Javu, a ktorý sa používa už aj v praxi.

Pre lepšiu názornosť princípov objektovo-orientovaného programovania v texte sa prelína výklad Javy so všeobecnejším pohľadom na objektovo-orientované programovanie v kontexte objektovo-orientovaného vývoja softvéru ako takého:

- Kapitola 2 poskytuje vhľad do objektovo-orientovaného programovania.
- Kapitola 3 uvádza čitateľa do programovacieho jazyka Java.

- Kapitoly 4–6 vysvetľujú principiálne záležitosti objektovo-orientovaného programovania a ich realizáciu v Jave.
- Kapitoly 7–13 sú venované ďalším prvkom Javy a Java API (rozhrania aplikačného programovania), ktoré sa nedajú označiť priamo za objektovo-orientované, ale sú postavené na objektovo-orientovaných základoch jazyka a je nevyhnutné poznať ich pre plnohodnotné využitie tohto jazyka.
- Kapitoly 14–16 ponesú čitateľa za zdanlivé hranice objektovo-orientovaného programovania — k modelovaniu, vzorom a aspektom.

2 Vhľad do objektovo-orientovaného programovania

Spolu s procedurálnym programovaním objektovo-orientované programovanie predstavuje najpoužívanejší prístup k programovaniu. Táto kapitola približuje objektovo-orientované programovanie prostredníctvom príkladu.

2.1 Vznik objektovo-orientovaného programovania

Podľa Thomasa Kuhna k zmene paradigmy — prevládajúceho vedeckého názoru v danej oblasti — dochádza zlomom, teda revolúciou, nie postupne, evolúciou. K zlomu prichádza, keď sa problémy platnej paradigmy stanú neúnosnými. Klasickým príkladom takého zlomu je zmena z newtonovskej mechaniky na teóriu relativity [Kuh70].

Aj keď sa nástup objektovo-orientovaného programovania často vníma ako zlom paradigmy, korene objektovo-orientovaného programovania sú v skutočnosti v procedurálnom programovaní. Prvý objektovo-orientovaný jazyk, Simula 67, ktorý ako súčasť svojho názvu má rok vzniku, za základ mal procedurálny jazyk Algol 60. Autori Simuly 67, Ole-Johan Dahl a Kristen Nygaard, prví použili pojem objekt v zmysle objektovo-orientovaného programovania.[1]

Programovací jazyk Smalltalk[2] predstavuje významný posun vo vývoji objektovo-orientovaného programovania. Tento jazyk je založený na postuláte „Všetko je objekt.“ a dôsledne ho uplatňuje, takže objektmi sú aj samotné riadiace konštrukcie (ako napríklad **if**). Kým v Simule 67 išlo predovšetkým o organizáciu kódu, v Smalltalku je dôraz na spolupráci objektov ako entít vo vykonávaní programu, ktorých manipulácia je veľmi pružná vďaka tomu, že ide o interpretovaný jazyk.

[1] `http://staff.um.edu.mt/jskl1/talk.html`
[2] `http://www.smalltalk.org`

2.2 Objekty a triedy

Objektovo-orientované programovanie predstavuje programovanie pomocou objektov. Objekt je entita, ktorá má [Boo94]:

- stav
- správanie
- identitu

Stav objektu zahŕňa všetky vlastnosti objektu a ich hodnoty. Správanie objektu predstavuje konanie objektu pri zmenách stavu a aktivácii operácií, ktoré poskytuje. Identita objektu predstavuje jeho jednoznačnú identifikáciu.

Objektovo-orientovaný program sa uskutočňuje ako *interakcia objektov*. Správanie objektu v objektovo-orientovaných jazykoch so statickými typmi (ako je Java) definuje jeho *typ*. Typ objektu sa označuje ako *trieda* (class). V zdrojových textoch programov sa najčastejšie definujú triedy, a objekty pritom predstavujú ich inštancie. Preto sa niekedy zdá, že ide skôr o programovanie pomocou tried. Iný prístup je v dynamických objektovo-orientovaných jazykoch, v ktorých niekedy úplne absentuje pojem triedy ako šablóny pre tvorbu objektov a objekty sa vytvárajú priamo.

2.3 Stav a správanie objektu

Hovorili sme o objektoch, ich spolupráci a triedach ako šablónach pre tvorbu objektov. Pozrime sa na tieto pojmy bližšie prostredníctvom príkladu.

Predpokladajme, že vytvárame hru. Jeden z objektov v hre je `statocnyRytier`. Jeho stav je daný týmito vlastnosťami:

- `statocnyRytier.poloha` — kde sa `statocnyRytier` nachádza
- `statocnyRytier.energia` — akú `statocnyRytier` má energiu

Bodkou vyjadrujeme to, že dané vlastnosti patria spomínanému objektu. Vlastnosti objektu sa označujú ako *atribúty*.

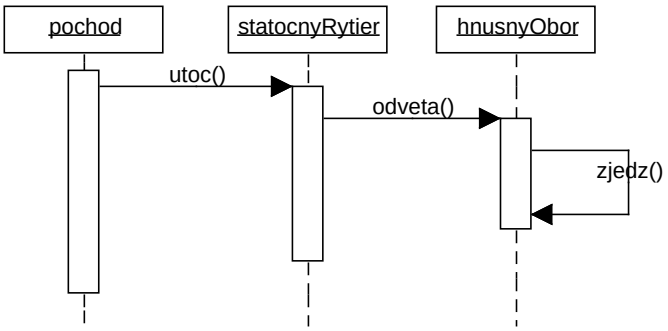
Správanie objektu `statocnyRytier` je dané operáciami, ktoré poskytuje. V našom príklade pôjde o jedinú operáciu: `statocnyRytier.utoc()`. Jej aktivácia spôsobí to, že `statocnyRytier` zaútočí na nepriateľa.

Ďalší z objektov v hre je `hnusnyObor`. Jeho stav je daný týmito vlastnosťami:

- `hnusnyObor.poloha` — kde sa `hnusnyObor` nachádza
- `hnusnyObor.energia` — akú `hnusnyObor` má energiu
- `hnusnyObor.hladny` — či je `hnusnyObor` hladný

Správanie objektu `hnusnyObor` je znovu len jedno: `hnusnyObor.odveta()`. Aktivácia tejto operácie spôsobí odvetu nepriateľovi, ktorý zaútočil na objekt `hnusnyObor`.

Stret rytiera a obra môžeme potom vnímať ako interakciu objektov, ako je to znázornené na obr. 2.1. Interakciu iniciuje objekt `pochod`, o ktorého detaily sa v tomto okamihu nezaujímame. Objekty interagujú prostredníctvom tzv. posielania *správ*, ktoré vlastne predstavuje volanie *operácií*.



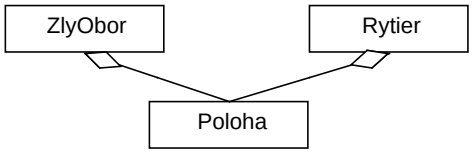
Obrázok 2.1: Stret rytiera a obra.

2.4 Typ objektu

Ako už bolo povedané, správanie objektu definuje jeho typ. Typ objektu sa označuje ako trieda (class). Napríklad `hnusnyObor` predstavuje jedného zo zlých obrov — jeho typ je daný triedou `ZlyObor`. Ďalším príkladom je `statocnyRytier`, ktorý je objektom triedy `Rytier`, alebo `poloha`, ktorá je objektom triedy `Poloha`.

2.5 Agregácia

Triedy `ZlyObor` a `Rytier` obsahujú triedu `Poloha`. Táto skutočnosť je znázornená na obr. 2.2. Takýto spôsob spájania tried do väčších celkov sa označuje ako *agregácia*.



Obrázok 2.2: Agregácia.

Agregácia predstavuje spôsob tvorenia *hierarchie*: agregát, t.j. prvok, do ktorého sa zahŕňajú agregované prvky, je nadradeným prvkom vzhľadom na tieto prvky. Teda, v našom príklade triedy `ZlyObor` a `Rytier` sú nadradené triede `Poloha`.

2.6 Zapuzdrenie

Pokúsme sa bez zavádzania jazykových formalizmov načrtnúť implementáciu triedy `ZlyObor` — jej atribúty a operácie:

```
class ZlyObor {
   Poloha poloha;
   int energia;
   boolean hladny;

   void odveta(Rytier r) {
      if (hladny)
         zjedz(r);
   }

   void zjedz(Rytier r) {
      int e = zistiEnergiu();
      r.uberEnergiu(e);
      zvysEnergiu(e);
   }

   int zistiEnergiu() {
      return energia;
   }

   void zvysEnergiu(int i) {
      energia = energia + i;
   }
```

```
    void znizEnergiu(int i) {
        energia = energia - i;
    }
}
```

Jedným z kľúčových princípov objektovo-orientovaného programovania je *zapuzdrenie* (encapsulation), označované ešte aj ako skrývanie informácií. Tento princíp hovorí, že implementácia objektu má zostať skrytá a prístup k objektu má byť zabezpečený prostredníctvom *rozhrania* (interface), ktoré tvoria vybrané operácie.

Napríklad energia zlých obrov sa nemení priamo siahnutím na atribút `energia`, ale prostredníctvom operácií `zvysEnergiu`() a `znizEnergiu`(). Cudzie objekty by nemali mať možnosť prístupu k tomuto atribútu, na čo — ako uvidíme v nasledujúcej kapitole — jestvujú príslušné jazykové konštrukcie. V uvedenom príklade však pre jednoduchosť tieto konštrukcie nie sú uvedené a princíp zapuzdrenia je porušený aj z iného hľadiska: na zmenu polohy by mali tiež byť poskytnuté zodpovedajúce operácie.

2.7 Dedenie

`ZlyObor` je len jeden možný druh obrov. Niekedy sa potrebujeme vyjadriť o obroch vo všeobecnosti bez ohľadu na špecifické druhy obrov. *Zovšeobecnením* (generalization) obrov by v našom príklade mohla byť trieda `Obor`, ktorá zahŕňa spoločné vlastnosti všetkých obrov:

```
class Obor {
    boolean hladny;
    int energia;

    void odveta(Rytier r) {
        r.znizEnergiu(1);
    }
    int zistiEnergiu() {
        return energia;
    }
    void zvysEnergiu(int i) {
        energia = energia + i;
    }
    void znizEnergiu(int i) {
        energia = energia - i;
    }
}
```

`ZlyObor` potom predstavuje *špeciálizáciu* triedy `Obor`:

```
class ZlyObor extends Obor {

   void odveta(Rytier r) {
      if (hladny)
         zjedz(r);
   }
   void zjedz(Rytier r) {
      int e = zistiEnergiu();
      r.uberEnergiu(e);
      zvysEnergiu(e);
   }
}
```

`ZlyObor` *dedí správanie a štruktúru* triedy `Obor`. Pod štruktúrou väčšinou rozumieme atribúty alebo kód ako taký, kým pod správaním rozumieme funkcionalitu, ktorú implementujú operácie. Inak povedané, trieda `ZlyObor` *rozširuje a konkretizuje* triedu `Obor` — pridáva nové detaily do *abstrakcie* obra. Preto je v Jave kľúčové slovo pre dedenie práve **extends**.

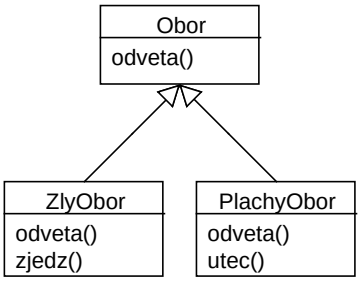
Ďalším príkladom obra môže byť `PlachyObor`, ktorý je, ako vidíme z operácie `odveta`(), naozaj plachý.:

```
class PlachyObor extends Obor {
   void utec() {
      . . .
   }
   void odveta(Rytier r) {
      utec();
   }
}
```

Dôležité je uvedomiť si, že popri agregácii, dedenie predstavuje ďalší spôsob tvorenia hierarchie v objektovo-orientovanom programovaní. Dokonca, keď sa povie len „hierarchia tried", myslí sa na hierarchiu dedenia. Hierarchiu obrov názorne ukazuje obr. 2.3.

2.8 Polymorfizmus

V dedení zďaleka nejde len o zdieľanie spoločného kódu ako sa na prvý pohľad môže zdať. Oveľa dôležitejšie je, že objekt každej triedy, ktorá je v hierarchii dedenia



Obrázok 2.3: Dedenie — hierarchia obrov.

nadradená danej triede, je zároveň aj objektom tejto nadradenej triedy. Presnejšie povedané, typ definovaný podtriedou je podtypom typu definovaným nadtriedou.

V našom príklade aj objekty typu `PlachyObor`, aj objekty typu `ZlyObor` sú typu `Obor` — preto môžeme premenným typu `Obor` priradiť objekty týchto podtypov:

```
Obor o1;
Obor o2;

o1 = new ZlyObor();
o2 = new PlachyObor();
```

Na vytvorenie objektov bolo použité kľúčové slovo **new**. Proces vytvárania objektov je detailne vysvetlený v nasledujúcej kapitole a v tomto okamihu nie je dôležitý.

Nad objektmi `o1` a `o2` môžeme volať všetky operácie definované pre typ `Obor`. Aké však bude správanie v prípade vyvolania operácie, ktorú každý z objektov definuje inak? Príkladom je operácia `odveta`(). Za predpokladu, že premenná `r` predstavuje rytiera, ktorý na obra zaútočil, pri odvete obra `o1`

```
o1.odveta(r);
```

rytier bude zjedený, kým pri odvete obra `o2`

```
o2.odveta(r);
```

sa mu nestane nič, lebo obor `o2` utečie, ako je definované v operáciách `odveta`() príslušných tried, a nie v operácii `odveta`() triedy `Obor`.

Výhoda tohto javu, ktorý sa označuje ako *polymorfizmus* začína byť zrejmá pri vyšších počtoch objektov a dopredu neznámom počte typov. Tak napríklad pre hordu sto

obrov, ktorí nemusia byť len typov `ZlyObor` a `PlachyObor`, pri ich strete s rovnako početnou rytierskou výpravou môžeme napísať jednoducho

```
for (int i = 0; i < 100; i++) {
   obor[i].odveta(r[i]);
}
```

a vieme, že každý obor urobí práve to, čo je pre jeho typ charakteristické. Pre úplnosť treba povedať, že výrazom `obor[]` označujeme pole premenných typu `Obor`, podobne ako v jazyku C, čo bude podrobne vysvetlené v nasledujúcej kapitole. Všimnite si veľmi dôležitú skutočnosť, že nepotrebujeme žiadne podmienené príkazy, a že v prípade zmeny v typoch obrov tento kód vôbec nie je potrebné upravovať.

2.9 Sumarizácia

Skôr ako začneme rozoberať jednotlivé záležitosti objektovo-orientovaného programovania a špecifiká programovacieho jazyka Java, táto kapitola poskytla vhľad do objektovo-orientovaného programovania. Pojmy v tejto kapitole boli definované s istou voľnosťou a príklady kódu boli prezentované bez presnej definície syntaxe a sémantiky, ale ak sa podarilo aspoň trochu demystifikovať problematiku objektovo-orientovaného programovania, kapitola splnila svoj účel.