Poglavlje 3

Dijagrami klasa: osnovni pojmovi

Ako bi vam neko prišao u mračnom prolazu i rekao: „Psst, hoćeš li da vidiš UML dijagram?“, taj crtež bi verovatno bio dijagram klasa. Najveći broj UML dijagrama koje imam prilike da vidim, jesu dijagrami klasa.

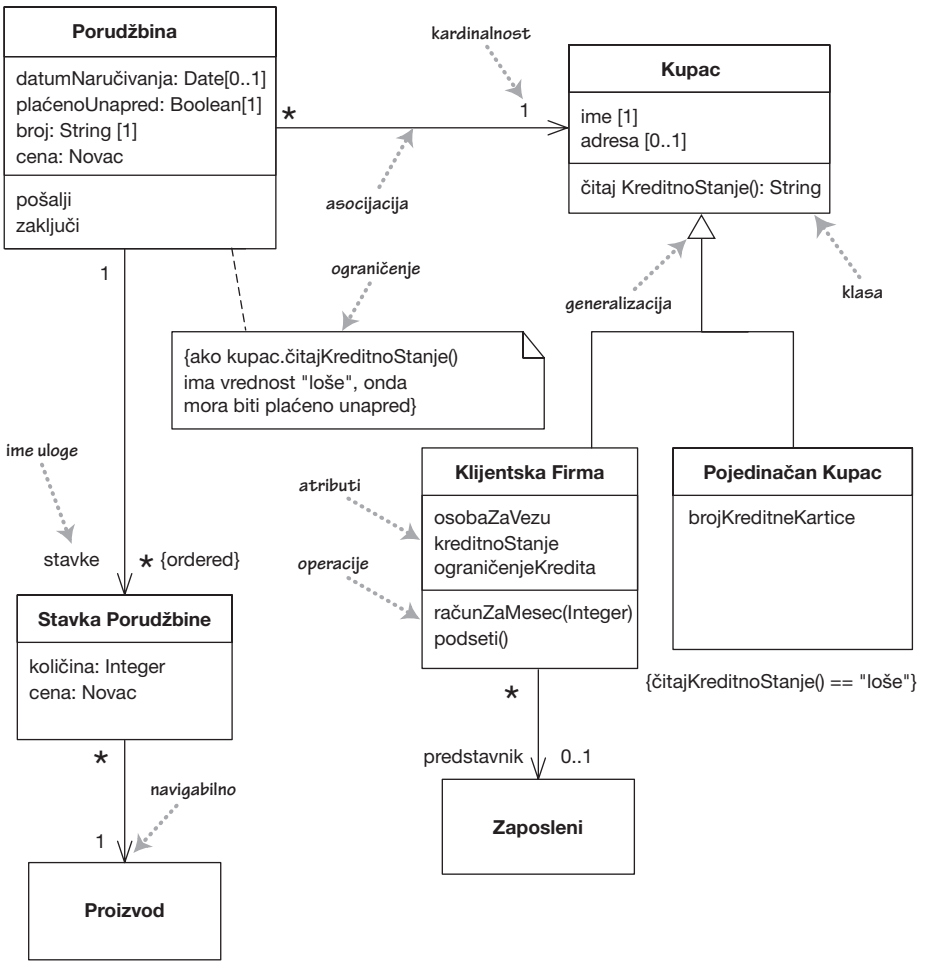
Ne samo da se dijagrami klasa često koriste, nego postoji i najveći skup tehnika za njihovo modelovanje. Iako su osnovni elementi svima potrebni, napredniji pojmovi se ređe koriste. Zato sam izlaganje podelio u dva dela: osnovni pojmovi (opisani su u ovom poglavlju) i napredni pojmovi (peto poglavlje).

Dijagram klasa (engl. *class diagram*) opisuje tipove objekata u sistemu i različite vrste statičkih veza koje postoje među njima. Dijagrami klasa takođe prikazuju svojstva i operacije klasa, kao i ograničenja načina povezivanja objekata. U jeziku UML koristi se opšti naziv **odlika** (engl. *feature*) i za svojstva i za operacije klase.

Na slici 3.1 prikazan je jednostavan model klasa, koji će vam odmah biti poznat ako ste radili s porudžbinama. Pravougaoni simboli na dijagramu jesu klase, a podeljeni su na tri odeljka: ime klase (podebljano), njeni atributi i njene operacije. Slika 3.1 prikazuje i dve vrste veza među klasama: asocijacije i generalizacije.

Svojstva

Svojstva (engl. *properties*) predstavljaju strukturne karakteristike klase. Da bi vam značenje svojstava bilo jasnije, zamislite da odgovaraju poljima klase. Kao što ćete videti, stvarnost je komplikovanija, ali je ova aproksimacija sasvim dobra za početak.



Slika 3.1. *Jednostavan dijagram klasa*

Svojstvo je jedan pojam, a pojavljuje se u dva potpuno različita oblika u dijagramu: kao atribut i kao asocijacija. Iako na dijagramu izgledaju sasvim različito, ta svojstva predstavljaju istu stvar.

Atributi

Atribut (engl. *attribute*) opisuje svojstvo u jednom redu teksta unutar klase. Potpuni oblik zapisa atributa je:

```
vidljivost ime: tip kardinalnost = podrazumevana-vrednost {opis-svojstva}
```

Primer za atribut je:

```
- name: String [1] = "Bez naslova" {readOnly}
```

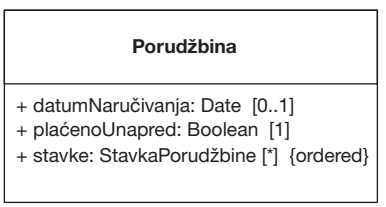
Obavezno je samo ime.

- Oznaka `vidljivosti` (engl. *visibility*) označava da li je atribut javni (+) ili privatni (-). Ostale vrste vidljivosti opisaću na strani 83.
- `Ime` atributa (engl. *name*) određuje kako se zove atribut u klasi i približno odgovara imenu polja u programskom jeziku.
- `Tip` atributa (engl. *type*) ukazuje na to da postoji ograničenje vrste objekata koji se mogu smestiti u atribut. O tipu atributa možete misliti kao o tipu polja u programskom jeziku.
- `Kardinalnost` (engl. *multiplicity*) objasniću na strani 38.
- `Podrazumevana vrednost` (engl. *default value*) jeste vrednost atributa u novom objektu, ako se drugačija vrednost zada za vreme nastanka objekta.
- `Opis svojstva` (engl. *property string*) omogućava da definišete dodatne osobine atributa. Na primer, koristiću opis `{readOnly}` da bih pokazao da klijentima nije dozvoljeno da menjaju svojstvo. Ako ga izostavim, smatrajte da je svojstvo promenljivo. U nastavku ću navesti i druge opise svojstava.

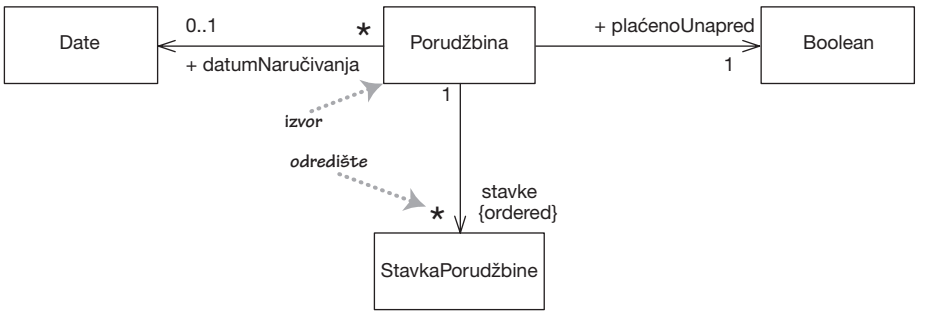
Asocijacije

Drugi način da označite svojstvo jeste asocijacija. Većina informacija koje možete prikazati atributom pojavljuje se i u asocijaciji. Slike 3.2 i 3.3 prikazuju ista svojstva na dva različita načina.

Asocijacija (engl. *association*) označava se punom linijom između dve klase, usmerenom od izvorne klase ka odredišnoj. Ime svojstva je na odredišnom kraju asocijacije, kao i njena kardinalnost. Odredišna klasa određuje tip svojstva.



Slika 3.2 *Svojstva klase Porudžbina prikazana kao atributi*



Slika 3.3 *Svojstva klase Porudžbina prikazana kao asocijacije*

Iako se većina informacija može prikazati na oba načina, neki elementi se razlikuju. Konkretno, kardinalnost asocijacije može biti definisana na oba kraja linije.

Pošto postoje dva načina označavanja iste stvari, postavlja se pitanje izbora. Atribute uglavnom koristim za mala svojstva, kao što su datumi ili logičke promenljive (uopšteno, za vrednosti prostih tipova – pogledajte stranu 73), a asocijacije za značajnije klase, kao što su kupci ili porudžbine. Osim toga, važne klase na dijagramu radije predstavljam pravougaonim oblicima, zbog čega se nameće upotreba asocijacija, dok atribute koristim za manje važne stvari na tom dijagramu. Izbor je zasnovan više na naglašavanju, nego na značenju pojmova.

Kardinalnost

Kardinalnost (engl. *multiplicity*) svojstva pokazuje na koliko objekata se odnosi svojstvo. Najčešće ćete sretati sledeće kardinalnosti:

- **1** (Jedna porudžbina mora imati tačno jednog kupca.)
- **0..1** (Za klijentsku firmu može biti zadužen poseban predstavnik, ali ne mora.)
- * (Kupac ne mora poslati porudžbinu, ali ne postoji gornja granica broja porudžbina jednog kupca – nula ili više porudžbina.)

Opštije rečeno, kardinalnosti su definisane gornjom i donjom granicom, na primer 2..4 za broj igrača kanaste. Donja granica može biti bilo koji pozitivan broj ili nula. Gornja granica je bilo koji pozitivan broj ili * (zvezdica znači da

nema ograničenja). Ako su donja i gornja granica iste, možete koristiti jedan broj, pošto je 1 ekvivalentno sa 1..1. S Pošto je 0..* uobičajen slučaj, umesto pune oznake koristi se samo *.

Nailazićete i na različite pojmove koji se odnose na kardinalnost atributa.

- **Opcioni** (engl. *optional*) znači da je 0 donja granica.
- **Obavezni** (engl. *mandatory*) znači da je donja granica veća ili jednaka 1.
- **Jedna vrednost** (engl. *single-valued*) znači da je 1 gornja granica.
- **Veći broj vrednosti** (engl. *multivalued*) znači da je gornja granica veća od 1, obično je to *.

Ime svojstva sa više vrednosti uglavnom navodim u množini.

Podrazumeva se da elementi sa više vrednosti čine skup, pa ako pitate kupca za njegove porudžbine, među njima neće postojati nikakav redosled (engl. *order*). Ako je značajan redosled porudžbina u asocijaciji, treba da dodate opis `{ordered}` na kraj linije asocijacije. Ukoliko hoćete da dozvolite duplikate, dodajte opis `{nonunique}`. (Ako želite da izričito naglasite standardne osobine, upišite `{unordered}` i `{unique}`.) Nailazićete i na tipove kolekcija, kao što je `{bag}`, što je oznaka za kolekcije koje ne zahtevaju jedinstvenost elemenata.

Jezik UML 1 dozvoljavao je i kardinalnosti s prekidima, kao što je 2, 4 (što znači 2 ili 4, kao u slučaju broja vrata na automobilima, pre pojave malih teretnih vozila). Takve kardinalnosti nisu naročito uobičajene, a u jeziku UML 2 nisu dozvoljene.

Podrazumevana kardinalnost atributa je [1]. Iako to važi za metamodel, ne možete pretpostavljati da je kardinalnost atributa na dijagramu [1] ako nije navedena, pošto informacija o kardinalnosti može biti izostavljena sa dijagrama. Zato radije izričito navodim kardinalnost [1] ako je ona značajna.

Predstavljanje svojstava u programu

Kao i u slučaju bilo čega drugog u jeziku UML, ne postoji način da se svojstva jednoznačno predstave u kodu. Uobičajeno se predstavljaju kao polja (engl. *field*) ili svojstva programskog jezika. Tako bi klasi Stavka Porudžbine sa slike 3.1 odgovarao sledeći kôd na jeziku Java:

```
public class StavkaPorudzbine...
  private int kolicina;
  private Novac cena;
  private Porudzbina porudzbina;
  private Proizvod proizvod
```

Na jeziku koji podržava svojstva, kao što je C#, klasa bi odgovarala kodu:

```
public StavkaPorudzbine...
  public int Kolicina;
  public Novac Cena;
  public Porudzbina Porudzbina;
  public Proizvod Proizvod;
```

Obratite pažnju na to da atribut uglavnom odgovara javnim svojstvima ako jezik podržava svojstva, a privatnim poljima ako to nije slučaj. U kodu na jeziku koji ne podržava svojstva možete videti polja, kojima se pristupa pomoću metoda čitača (get) i menjača (set). Nepromenljiv atribut nema menjačku metodu (u slučaju polja), odnosno aktivnost za menjanje (u slučaju svojstava). Zapazite da se kao ime svojstva obično koristi ime odredišne klase, ako ne zadate drugačije.

Tumačenje atributa kao privatnih polja zasnovano je u velikoj meri na realizaciji. U tumačenju zasnovanom na interfejsu više bismo se usredsredili na pristupne metode nego na osnovne podatke, pa bi atributi klase StavkaPorudzbine odgovarali sledećim metodama:

```
public StavkaPorudzbine...
  private int kolicina;
  private Proizvod proizvod;
  public int citajKolicinu() {
    return kolicina;
  }
  public void zadajKolicinu(int kolicina) {
  this.quantity = quantity;
  }
  public Novac citajCenu() {
  return proizvod.citajCenu().pomnozi(kolicina);
  }
```

U ovom slučaju ne postoji polje podatka za cenu, nego je to izračunata vrednost. Međutim, sa stanovišta klijenata klase StavkaPorudzbine, ova vrednost izgleda isto kao polje. Klijenti ne mogu odrediti šta je polje a šta je izračunato. To skrivanje informacija je suština kapsuliranja (engl. *encapsulation*).

Ako atribut ima više vrednosti, u kodu ga predstavlja kolekcija. Tako bi klasa Porudžbina sadržala kolekciju objekata klase StavkaPorudžbine. Pošto je pored kardinalnosti naznačeno uređenje (opis {ordered}), ta kolekcija mora biti uređena (kao List u Javi ili IList u .NET-u). Ukoliko je kolekcija neuređena, tada ne postoji nikakav značajan redosled i može se realizovati pomoću skupa, ali većina programera koristi liste za realizaciju neuređenih atributa. Neki koriste nizove, ali UML ne ograničava gornju granicu, pa ja uglavnom koristim kolekciju za tu strukturu podataka.

Svojstva sa više vrednosti proizvode drugačiju vrstu interfejsa u odnosu na svojstva s jednom vrednošću (na jeziku Java):

```
class Porudzbina {
  private Set stavke = new HashSet();
  public Set citajStavke() {
    return Collections.unmodifiableSet(stavke);
  }
  public void dodajStavku (StavkaPorudzbine arg) {
    stavke.add (arg);
  }
  public void ukloniStavku (StavkaPorudzbine arg) {
    stavke.remove(arg);
  }
```

U najvećem broju slučajeva, svojstvu sa više vrednosti ne dodeljujete vrednosti direktno, već ga ažurirate metodama za dodavanje i uklanjanje. Da bi upravljala svojim svojstvom stavke, klasa Porudzbina bi morala da upravlja pripadnošću toj kolekciji, te ne bi trebalo da prosleđuje celu kolekciju drugima. U takvom slučaju sam koristio zaštitnog posrednika, kako bih napravio nepromenljiv omotač kolekcije pre vraćanja iz metode citajStavke. Osim toga, možete napraviti nepromenljiv iterator ili napraviti kopiju kolekcije. Dozvoljeno je da klijenti menjaju objekte članove, ali ne bi trebalo da neposredno menjaju kolekciju.

Pošto atributi sa više vrednosti ukazuju na kolekcije, na dijagramu klasa skoro nikada nećete videti klase kolekcija. Prikazujte ih samo na dijagramima vrlo niskog nivoa, koji predstavljaju realizaciju kolekcija.

Izbegavajte klase koje su sačinjene samo od kolekcije polja i njihovih čitača. Objektno orijentisano projektovanje treba da proizvede objekte sa raznovrsnim ponašanjem, a ne da služe samo za snabdevanje drugih objekata podacima. Ako često tražite podatke pomoću čitača, to je znak da neko ponašanje treba premestiti u objekat koji ima te podatke.

Navedeni primeri potkrepljuju činjenicu da se između UML-a i koda ne može brzo i lako uspostaviti veza, mada sličnosti postoje. Dogovori unutar projektnog tima olakšaće pronalaženje te veze.

Bez obzira na to da li je svojstvo realizovano kao polje ili kao izračunata vrednost, objekat ga uvek poseduje. Nemojte koristiti svojstvo za modelovanje privremene veze, na primer, objekta koji se prosleđuje kao parametar metodi, a koristi se samo u okviru te interakcije.

Dvosmerne asocijacije

Asocijacije koje smo do sada posmatrali su jednosmerne (engl. *unidirectional*). Druga uobičajena vrsta asocijacija jeste dvosmerna asocijacija (engl. *bidirectional*). Primer je prikazan na slici 3.4.

Slika 3.4 *Dvosmerna asocijacija*

Dvosmerna asocijacija je par svojstava koja su povezana kao uzajamno inverzna. Klasa Auto ima svojstvo `vlasnik:Osoba[1]`, a klasa Osoba ima svojstvo `automobili:Auto[*]`. (Zapazite da sam svojstvu `automobili` dodelio ime koje je množina tipa svojstva, što je uobičajena, ali nenormativna konvencija.)

Inverzna veza među njima ukazuje da ćete se, ako pratite oba svojstva, vratiti do skupa koji sadrži polaznu tačku. Na primer, mogu da krenem od nekog vozila, pronađem njegovog vlasnika, a zatim tražim automobile te osobe. Taj skup automobila treba da sadrži vozilo od koga sam pošao.

Umesto označavanja asocijacije imenom svojstva, mnogi projektanti – posebno oni koji su modelovali podatke – obeležavaju asocijaciju glagolskim oblikom (slika 3.5), tako da se ime veze može koristiti u rečenici. To je ispravno, a asocijaciji možete dodati strelicu da bi se izbegla dvosmislenost. Većina projektanata radije koristi ime svojstva, pošto ono više odgovara odgovornostima i operacijama.

Neki ljudi imenuju svaku asocijaciju. Ja imenujem asocijaciju samo ako to doprinosi boljem razumevanju. Video sam previše asocijacija sa imenima poput „sadrži“ ili „u vezi je sa“.

Na slici 3.4, dvosmernost asocijacije naglašava se **strelicama navigabilnosti** (engl. *navigability arrows*) na oba kraja asocijacije. Na slici 3.5 nisu nacrtane strelice; UML dozvoljava da koristite ovaj način kada ukazujete na dvosmernu asocijaciju ili kada ne prikazujete navigabilnost. Preporučujem vam da koristite strelicu na oba kraja asocijacije, kao na slici 3.4, kada hoćete da naglasite da je dvosmerna.

Realizacija dvosmerne asocijacije na programskom jeziku često nije sasvim jednostavna, pošto morate usaglasiti ta dva svojstva. Sledi primer realizacije dvosmerne asocijacije na jeziku C#:

Slika 3.5 *Upotreba glagolskog oblika za imenovanje asocijacije*

```
class Auto...
  public Osoba Vlasnik {
    get {return _vlasnik;}
    set {
      if(_vlasnik != null) _vlasnik.prijateljskiAutomobili().Remove(this);
      _vlasnik = value;
      if (_vlasnik != null) _vlasnik.prijateljskiAutomobili().Add(this);
    }
  }
  private Osoba _vlasnik;
...

class Osoba ...
  public IList Automobili {
    get {return ArrayList.ReadOnly(_automobili);}
  }
  public void DodajAuto(Auto arg) {
    arg.Vlasnik = this;
  }
  private IList _automobili = new ArrayList();
  internal IList prijateljskiAutomobili() {
    //korišćenje je dozvoljeno samo objektu Auto.Vlasnik
    return _automobili;
  }
....
```

Jednoj strani asocijacije mora se omogućiti da upravlja vezom. Ako je to moguće, vezom treba da upravlja strana s jednom vrednošću. Da bi to funkcionisalo, podređena strana (Osoba) treba svoje kapsulirane podatke da učini dostunim nadređenoj strani. Tako se podređenoj klasi dodaje jedna opasna metoda, koja ne bi trebalo tu da se nalazi, osim ako jezik posebno dobro upravlja pristupom. Upotrebio sam reč „prijatelj“ u imenu, pošto bi menjačka metoda nadređene klase u jeziku C++ bila prijateljska metoda podređene klase. Kao i u najvećem delu koda kojim se realizuju svojstva, i ovde ima dosta ponavljanja, pa zato projektanti rado koriste neki alat za generisanje programa.

Navigabilnost nije značajno svojstvo u konceptualnim modelima, pa u njima ne prikazujem strelice za označavanje navigabilnosti.

Operacije

Operacije (engl. *operations*) jesu aktivnosti koje klasa ume da obavi. Sasvim je očigledno da operacije odgovaraju metodama klase. Najčešće se ne prikazuju jednostavne operacije nad svojstvima, pošto se one obično mogu izvesti.

Potpuna sintaksa operacija na jeziku UML glasi:

```
vidljivost ime (lista-parametara) : tip-rezultata {opis-svojstva}
```

- `Vidljivost` je javna (+) ili privatna (-). Ostale su opisane na 83. strani.
- `Ime` je niz znakova.
- `Lista-parametara` je lista parametara operacije.
- `Tip-rezultata` je tip rezultata operacije, ako ona ima rezultat.
- `Opis-svojstva` ukazuje na svojstva operacije.

Parametri iz liste označavaju se slično kao atributi. Sintaksa:

```
smer ime: tip = podrazumevana-vrednost
```

- `Ime`, `tip` i `podrazumevana-vrednost` imaju isto značenje kao u sintaksi atributa.
- `Smer` ukazuje na to da li je parametar ulazni (`in`), izlazni (`out`) ili ulazno-izlazni (`inout`). Ako nije označen smer, podrazumeva se `in`.

Primer operacije klase Account:

```
+ stanjeNa (datum: Datum) : Novac
```

U konceptualnim modelima ne treba da koristite operacije za specifikaciju interfejsa klase. Umesto toga ih koristite da ukažete na glavne odgovornosti te klase, sažimajući u nekoliko reči neku CRC odgovornost (detaljnija uputstva potražite na strani 62).

Korisno je napraviti razliku između operacija koje menjaju stanje sistema i operacija koje ga ne menjaju. Jezik UML definiše **upit** (engl. *query*) kao operaciju koja čita neku vrednost iz klase, ne menjajući stanje sistema – drugim rečima, bez sporednih efekata. Takvu operaciju označavate opisom svojstva `{query}`. Operacije koje menjaju stanje nazivam **modifikatori** (engl. *modifiers*) ili komande (engl. *commands*).

Precizno govoreći, upiti i modifikatori razlikuju se po tome da li menjaju vidljivo stanje [Meyer]. Vidljivo stanje je ono što se može uočiti spolja. Operacija koja ažurira privremenu memoriju promenila bi unutrašnje stanje, ali ne bi imala nikakvog efekta uočljivog spolja.

Mislim da je korisno naglasiti upite, pošto možete promeniti njihov redosled izvršavanja, a da ne promenite ponašanje sistema. Uobičajena konvencija je da modifikatori, ako je moguće, nemaju rezultat, pa se možete osloniti na činjenicu

da su operacije koje vraćaju neku vrednost upiti. [Meyer] to naziva pravilom razdvajanja komande i upita. Može biti naporno stalno primenjivati to pravilo, ali treba ga što više koristiti.

Ponekad ćete videti i pojmove čitačke metode (engl. *getting methods*) i menjačke metode (engl. *setting methods*). Menjačka metoda upisuje vrednost u polje (ne radi ništa drugo). Spolja posmatrano, ne bi trebalo da klijent može da prepozna da li upitu odgovara čitačka metoda, odnosno da li se modifikator odnosi na menjačku metodu. Izbor pristupnih metoda u potpunosti je unutrašnja stvar klase.

Postoji i razlika između značenja pojmova operacija i metoda. **Operacija** se primenjuje nad objektom (deklaracija procedure), dok je **metoda** telo procedure. One se razlikuju kada postoji polimorfizam. Ako imate nadtip sa tri podtipa i svaki od njih redefiniše operaciju `getPrice` nadtipa, tada postoji jedna operacija i četiri metode koje je realizuju.

Uglavnom se termini *operacija* i *metoda* koriste kao sinonimi, ali ima situacija kada je korisno precizno naglasiti razliku.

Generalizacija

Tipičan slučaj **generalizacije** su Pojedinačni kupci i Klijentske firme, u primeru trgovine (slika 3.1). Postoje razlike, ali i mnoge sličnosti. Zajedničke osobine se mogu smestiti u opštu klasu Kupac, koja predstavlja nadtip, a Pojedinačan kupac i Klijentska firma su podtipovi.

Ova pojava je i predmet različitih tumačenja sa raznih gledišta modelovanja. Sa stanovišta koncepata, možemo reći da je Klijentska firma podtip klase Kupac ako su sve instance klase Klijentska firma takođe, po definiciji, instance klase Kupac. Tada je Klijentska firma posebna vrsta klase Kupac. Osnovna ideja je da sve što kažemo o klasi Kupac (asocijacije, atributi, operacije) važi i za klasu Klijentska firma.

Iz perspektive softvera je očigledno da se to realizuje nasleđivanjem (engl. *inheritance*): Klijentska firma je potklasa (engl. *subclass*) klase Kupac. U najpopularnijim objektno orijentisanim jezicima potklasa nasleđuje sve karakteristike natklase (engl. *superclass*) i može redefinisati sve njene metode.

Važna posledica nasleđivanja je **zamenljivost** (engl. *substitutability*). Trebalo bi da u svakoj situaciji mogu da koristim Klijentsku firmu umesto klase Kupac i da sve dobro funkcioniše. U suštini, umesto objekta tipa Kupac, mogu slobodno upotrebiti objekat bilo kog podtipa te klase. Usled polimorfizma,

objekat klase Klijentska firma može na neke komande odgovoriti drugačije nego objekat klase Kupac, ali klijent ne bi trebalo da brine o razlikama. (Da biste saznali više pojedinosti o tome, pogledajte Liskovljevo pravilo zamenljivosti u knjizi [Martin].)

Iako je nasleđivanje moćan mehanizam, često komplikuje strukturu klasa više nego što treba da bi se postigla zamenljivost. Dobar primer za to nalazimo u prvom periodu korišćenja jezika Java, kada se mnogima nije dopadala realizacija ugrađene klase Vector i hteli su da je zamene nečim jednostavnijim. Međutim, jedini način da se zameni Vector bio je da se napravi potklasa te klase, a to je značilo da će biti nasleđeno mnogo nepoželjnih podataka i ponašanja.

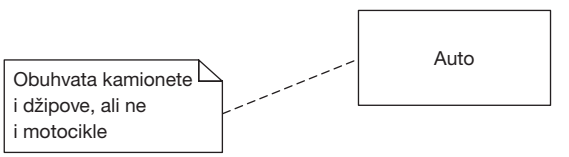
Postoje i mnogi drugi mehanizmi pravljenja zamenljivih klasa. Zato mnogi naglašavaju razliku između izrade podtipova (engl. *subtyping*), odnosno nasleđivanja interfejsa, i izrade potklasa (engl. *subclassing*), odnosno nasleđivanja realizacije. Klasa je **podtip** ako može zameniti nadtip, bez obzira na to da li je od njega izveden nasleđivanjem. **Izrada potklasa** se koristi kao sinonim za stvarno nasleđivanje.

Mnogi mehanizmi omogućavaju da pravite podtipove bez izrade potklasa. Primeri za to su realizacija interfejsa (strana 69) i mnogi standardni projektni obrasci navedeni u [Gang of Four].

Napomene i komentari

Napomene (engl. *notes*) jesu komentari na dijagramu. Mogu biti nezavisni od drugih elemenata dijagrama ili spojeni isprekidanom linijom sa elementima koje objašnjavaju (slika 3.6). Mogu se pojaviti na svakoj vrsti dijagrama.

Isprekidana linija je nezgodna ako ne možete tačno da ustanovite gde se završava. Zato je uobičajeno da se na kraj linije ucrta mali krug. Ponekad je korisno dodati komentar elementu dijagrama. To se postiže dodavanjem dve crtice (--) ispred teksta komentara.



Slika 3.6 *Napomena se koristi kao komentar jednog ili više elemenata dijagrama*

Zavisnost

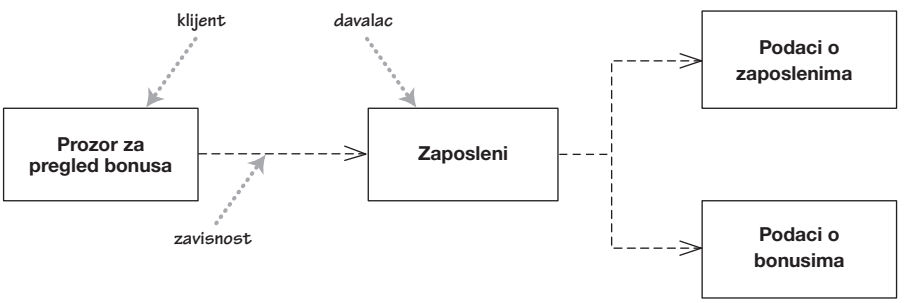
Između dva elementa postoji **zavisnost** (engl. *dependency*) ako promene definicije jednog elementa, **davaoca** (engl. *supplier*), mogu izazvati promene drugog, **klijenta** (engl. *client*). Postoje različiti razlozi zavisnosti među klasama: jedna klasa šalje poruku drugoj; jedna klasa sadrži drugu; objekat jedne klase prosleđuje objekat druge kao parametar neke operacije. Ako se promeni interfejs klase, može se dogoditi da poruke poslate toj klasi ne budu više ispravne.

Kako se računarski sistemi razvijaju, sve više se vodi računa o upravljanju zavisnostima. Ako zavisnosti izmaknu kontroli, svaka promena sistema se prenosi na ostale elemente koji se moraju menjati. Što više elemenata treba promeniti, to je promena teža.

UML omogućava da prikažete zavisnosti između svih vrsta elemenata. Koristite ih kad god hoćete da pokažete kako bi promene jednog elementa mogle izazvati promene drugih.

Slika 3.7 prikazuje neke zavisnosti koje se mogu nalaziti u višeslojnoj aplikaciji. Klasa Prozor za pregled bonusa jeste klasa **korisničkog okruženja** (engl. *presentation class*) i zavisi od klase Zaposleni, **objekta oblasti problema** (engl. *domain object*), koji obuhvata osnovna ponašanja sistema, u ovom slučaju pravila poslovanja. To znači da bi klasu prozora trebalo promeniti ako se promeni interfejs klase Zaposleni.

Ovde je značajno to što je zavisnost jednosmerna i usmerena od klase korisničkog okruženja ka klasi domena problema. Tako znamo da promene klase prozora neće uticati na klasu Zaposleni, niti na druge klase domena problema. Ustanovio sam da je korisno poštovati pravilo strogog razdvajanja logike okruženja i logike oblasti problema, pri čemu okruženje zavisi od oblasti problema, a obrnuto ne važi.



Slika 3.7 *Primeri zavisnosti*

Možete primetiti da na dijagramu ne postoji neposredna zavisnost klase prozora od dve klase s podacima. Ako se ove klase promene, možda će morati da se promeni i klasa Zaposleni. Međutim, uticaj promene se tu završava ako se menja samo realizacija klase Zaposleni, a ne i njen interfejs.

UML može da prikaže mnogo vrsta zavisnosti i svaka od njih se opisuje na drugačiji način pomoću rezervisanih reči. Osnovnu zavisnost koju sam ovde istakao smatram najkorisnijom i obično je koristim bez rezervisanih reči. Da biste je detaljnije opisali, dodajte odgovarajuću rezervisanu reč iz tabele 3.1

Osnovna zavisnost nije tranzitivna. Primer tranzitivnog odnosa je „ima dužu bradu". Ako Jim ima dužu bradu nego Grady i Grady ima dužu bradu nego Ivar, možemo zaključiti da Jim ima dužu bradu nego Ivar. Neke vrste zavisnosti su tranzitivne, na primer zamenljivost, ali uglavnom postoji značajna razlika između neposredne i posredne zavisnosti, kao na slici 3.7.

Mnogi odnosi u UML-u podrazumevaju zavisnost. Navigabilna asocijacija od klase Porudžbina do klase Kupac na slici 3.1 znači da klasa Porudžbina zavisi od klase Kupac. Potklasa zavisi od svoje natklase, ali obrnuto ne važi.

Tabela 3.1 *Neke rezervisane reči za opisivanje zavisnosti*

Rezervisana reč	Značenje
`<<call>>`	Izvor poziva operaciju odredišta.
`<<create>>`	Izvor pravi instance odredišta.
`<<derive>>`	Izvor je izveden iz odredišta.
`<<instantiate>>`	Izvor je instanca odredišta. Zapazite da, ako je izvor veze klasa, onda je to instanca klase klasa, što znači da je odredište metaklasa (engl. *metaclass*).
`<<permit>>`	Odredište dozvoljava izvoru da pristupa privatnim karakteristikama.
`<<realize>>`	Izvor je realizacija specifikacije ili interfejsa koji su definisani odredištem (69. strana).
`<<refine>>`	Preciziranje detalja (engl. *refinement*) ukazuje na odnos između različitih nivoa značenja. Na primer, izvor može biti klasa nastala tokom projektovanja, a odredište odgovarajuća klasa napravljena tokom analize.
`<<substitute>>`	Izvor može zameniti odredište (45. strana).
`<<trace>>`	Koristi se za evidentiranje, na primer, zahteva upućenih klasama ili veza između promena jednog modela i drugih promena.
`<<use>>`	U realizaciji izvora koristi se odredište veze.

Treba koristiti što manje zavisnosti, naročito onih koje se prostiru preko velikih delova sistema. Posebno obratite pažnju na ciklične zavisnosti, pošto one mogu dovesti do cikličnih promena. Ne smetaju mi uzajamne zavisnosti između tesno povezanih klasa, ali pokušavam da odstranim cikluse na širem nivou, posebno između paketa.

Prikazivanje svih zavisnosti na dijagramu klasa je beskorisno, pošto ih ima puno i često se menjaju. Budite izbirljivi i prikazujte zavisnosti samo ako su neposredno značajne za temu koju hoćete da opišete. Da biste razumeli zavisnosti i upravljali njima, najbolje je da ih koristite u dijagramima paketa (strana 89).

Zavisnosti na dijagramu klasa obično koristim za prikazivanje privremene veze, na primer kada se jedan objekat prosleđuje kao parametar drugom. Pored takvih veza videćete rezervisane reči <<parameter>>, <<local>> i <<global>>. Te oznake možete videti pored asocijacija i na dijagramima verzije 1 UML-a, kada ukazuju na privremene veze, a ne na svojstva. Navedene rezervisane reči ne postoje u jeziku UML 2.

Zavisnosti se mogu prepoznati čitanjem koda, pa ih je najbolje analizirati pomoću posebnih alata. Najefikasnije je da se alatom za povratnu analizu dobiju slike zavisnosti.

Ograničenja

Dijagrami klasa uglavnom opisuju ograničenja (engl. *constraints*). Slika 3.1 pokazuje da porudžbinu može dostaviti samo jedan kupac. Ovaj dijagram pokazuje i da se svaka stavka porudžbine razmatra odvojeno, na primer, u porudžbini se navodi „40 braon proizvoda, 40 plavih proizvoda i 40 crvenih proizvoda“, a ne „120 proizvoda“. Osim toga, dijagram pokazuje da za klijentsku firmu postoji kredit, a za pojedinačne kupce ne postoji.

Osnovni elementi, kao što su asocijacija, atributi i generalizacija u velikoj meri definiču važna ograničenja, ali ne mogu da ukažu na sva ograničenja. Preostala ograničenja ipak moraju negde biti prikazana, a dobro mesto za to je dijagram klasa.

Jezik UML dozvoljava da opišete ograničenja kako god želite. Jedino pravilo je da opis morate smestiti unutar vitičastih zagrada ({}). Možete koristiti prirodni jezik, neki programski jezik ili UML-ov formalni jezik ograničenja objekata (engl. *Object Constraint Language*, *OCL*), koji je opisan u [Warmer i Kleppe] i zasnovan na predikatskom računu. Upotreba formalne notacije otklanja opasnost od pogrešnih tumačenja, koja mogu nastati zbog nepreciznosti prirodnog

jezika. Međutim, ona unosi novu opasnost od pogrešnog tumačenja, ako autori i čitaoci ne razumeju dovoljno OCL. Zato bih preporučio upotrebu prirodnog jezika ako vaši čitaoci ne poznaju predikatski račun.

Ograničenju možete dodeliti ime, navodeći ga na početku i završavajući znakom dve tačke, na primer: {onemogućiti incest: muž i žena ne smeju biti u srodstvu}.

Projektovanje po ugovoru

Projektovanje po ugovoru (engl. *Design by Contract)* jeste tehnika projektovanja koju je razvio Bertrand Meyer [Meyer]. Ova tehnika je glavna karakteristika njegovog jezika Eiffel ali nije ograničena na njega, već se može koristiti u svakom programskom jeziku.

U središtu projektovanja po ugovoru nalazi se pretpostavka. **Pretpostavka** (engl. *assertion*) logički je iskaz koji nikada ne treba da bude netačan (ako je netačan došlo je do greške). Obično se pretpostavke proveravaju samo tokom otkrivanja i otklanjanja grešaka, a ne proveravaju se tokom izvršavanja konačne verzije programa. U programu ne treba da se vodi računa o tome da li se proveravaju pretpostavke.

Prilikom projektovanja po ugovoru koriste se tri vrste pretpostavki: preduslovi (engl. *pre-conditions*), završni uslovi (engl. *post-conditions*) i invarijante (engl. *invariants*). Preduslovi i završni uslovi primenjuju se na operacije. **Završni uslov** je iskaz koji opisuje kako bi svet trebalo da izgleda posle izvršenja neke operacije. Na primer, ako definišemo operaciju „kvadratni koren broja“, završni uslov bi bio *ulaz = rezultat * rezultat*, gde je *rezultat* izlazna, a *ulaz* ulazna vrednost. Završni uslov je koristan način da saopštimo šta radimo, bez opisivanja kako to radimo – drugim rečima, to je razdvajanje interfejsa od realizacije.

Preduslov je iskaz o tome kako očekujemo da svet izgleda pre izvršenja neke operacije. Možemo definisati preduslov *ulaz >= 0* za operaciju „kvadratni koren“. Taj preduslov saopštava da je pogrešno računati „kvadratni koren“ negativnog broja i da posledice posledice takvog zahteva nisu definisane.

Na prvi pogled deluje da je ideja provere prilikom računanja loša, pošto bi negde trebalo proveriti da li je ispravan poziv operacije „kvadratni koren“. Značajno je pitanje čija je to odgovornost.

Preduslovom se izričito saopštava da je za proveru odgovoran onaj ko poziva operaciju. Bez eksplicitnog iskaza o odgovornostima, provera može biti nedovoljna (ako obe strane pretpostavljaju da je ona druga

odgovorna) ili suvišna (ako obe strane proveravaju uslove). Suvišna provera je loša zato što udvostručava kôd za proveru i značajno usložnjava program. Eksplicitno isticanje nosioca odgovornosti pojednostavljuje program. Ukoliko pozivalac zaboravi da proveri uslove, to će najverovatnije biti otkriveno prilikom otklanjanja grešaka i testiranja, pošto se tada proveravaju pretpostavke.

Na osnovu navedenih definicija preduslova i završnog uslova, dobijamo preciznu definiciju pojma **izuzetak** (engl. **exception**). Izuzetak nastaje kada se pozove operacija čiji je preduslov zadovoljen, a po njenom izvršenju nije zadovoljen završni uslov.

Invarijanta je pretpostavka o klasi. Na primer, klasa Račun može imati invarijantu koja tvrdi da je *stanje == suma(stavka.iznos())*. Invarijanta je „uvek" tačna za sve instance klase. Ovde „uvek" znači „kad god se može pozvati operacija nad objektom".

U suštini, to znači da se invarijanta dodaje preduslovima i završnim uslovima povezanim sa svim javnim operacijama klase. Invarijanta može postati netačna tokom izvršavanja metode, ali bi trebalo da ponovo postane tačna do trenutka kada bilo koji drugi objekat može da pozove metodu te klase.

Pretpostavke mogu igrati značajnu ulogu u izradi potklasa. Jedna od opasnosti nasleđivanja jeste redefinisanje operacija potklase tako da ne budu usaglašene sa operacijama natklase. Prepostavke smanjuju verovatnoću da se to dogodi. Invarijante i završni uslovi klase moraju se primenjivati na sve potklase. Potklase mogu uvesti strože pretpostavke, ali ih ne smeju oslabiti. S druge strane, preduslov ne sme biti stroži, ali može biti oslabljen.

Na prvi pogled to pravilo izgleda suvišno, ali je važno da bi funkcionisalo dinamičko povezivanje (engl. *dynamic binding*). Po pravilu zamenljivosti, trebalo bi da uvek možete koristiti objekat potklase kao da je instanca natklase. Ako je preduslov potklase stroži, prilikom izvršavanja operacija natklase moglo bi doći do greške.

Kada treba koristiti dijagrame klasa

Dijagrami klasa su noseći stub UML-a, pa ćete ih stalno koristiti. U ovom poglavlju opisani su osnovni pojmovi, dok su u 5. poglavlju razmotreni napredni elementi.

Pri crtanju dijagrama klasa problem je mnoštvo elemenata čiju je upotrebu teško savladati. Sledi nekoliko preporuka:

- Ne pokušavajte da koristite sve elemente koji su vam dostupni. Počnite od jednostavnih stvari, opisanih u ovom poglavlju: klasa, asocijacija, atributa, generalizacije i ograničenja. Uvodite i druge elemente iz 5. poglavlja, ali samo kada vam trebaju.
- Otkrio sam da su konceptualni dijagrami klasa veoma korisni za istraživanje poslovne terminologije. Zato se morate potruditi da izostavite softver iz rasprave i da koristite jednostavnu notaciju.
- Nemojte crtati modele za sve oblasti, nego se usredsredite na najvažnije. Bolje je imati nekoliko dijagrama koje koristite i stalno ažurirate, nego puno zaboravljenih i zastarelih modela.

Pri korišćenju dijagrama klasa najopasnije je to što se možete usredsrediti na strukturu i zanemariti ponašanje. Dakle, kada crtate dijagrame klasa da biste razumeli softver, obavezno koristite i neku tehniku za opisivanje ponašanja. Ako radite kako treba, primetićete da te tehnike primenjujete naizmenično.

Preporučena literatura

Sve knjige o UML-u koje sam spomenuo u prvom poglavlju detaljnije se bave dijagramima klasa. Upravljanje zavisnostima je značajna karakteristika većih projekata. Najbolja knjiga o toj temi je [Martin].