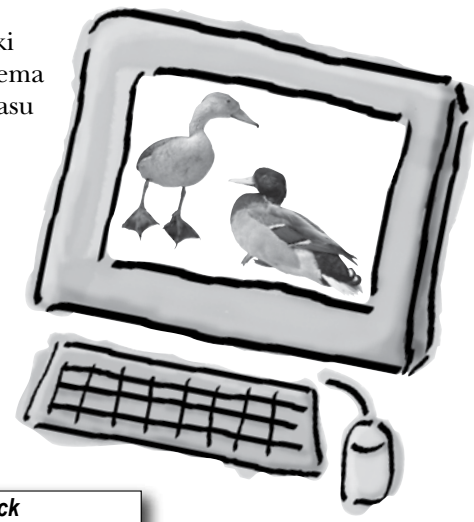
1 uvod u projektne obrasce

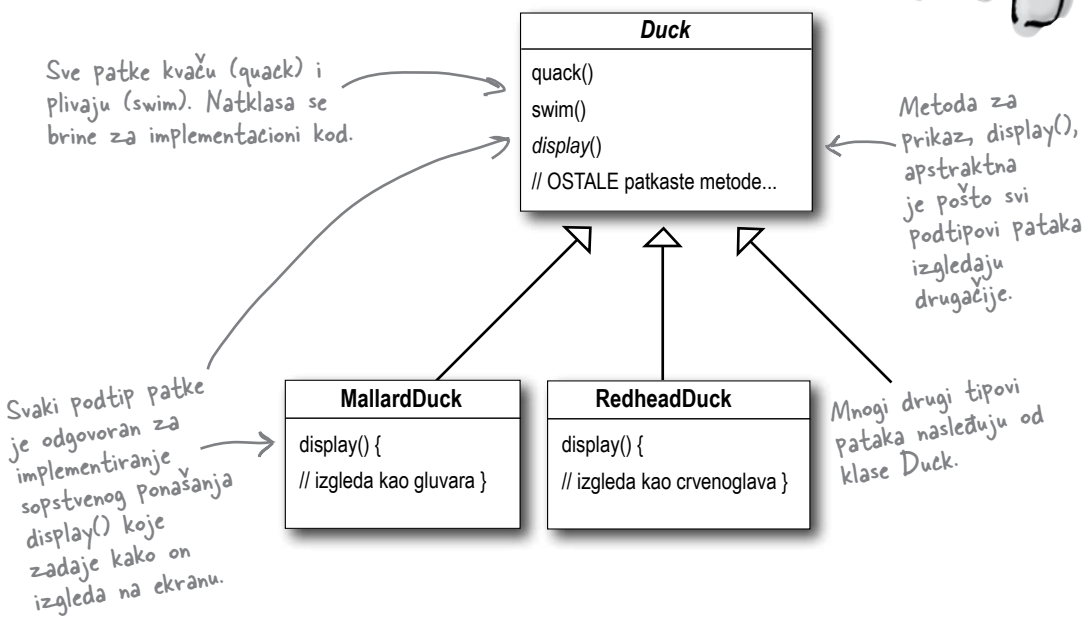
Dobro došli u projektne obrasce

Neko je već rešio vaše probleme. U ovom poglavlju ćete naučiti zašto (i kako) da koristite mudrosti i lekcije do kojih su došli drugi programeri koji su išli istim putem rešavanja problema pri projektovanju i preživeli. Pre nego što završimo, razmotrićemo upotrebu i koristi od projektnih obrazaca, pogledati neke principe objektno orijentisanog (OO) projektovanja i proći kroz primer rada jednog obrasca. Najbolji način da koristite obrasce jeste da njima *ispunite svoju glavu* i da potom *prepoznate mesta* u svojim projektima i postojećim aplikacijama gde biste mogli da *ih primenite*. Umesto ponovnog korišćenja koda, obrasci vam pružaju ponovno korišćenje *iskustva*.

Počelo je jednostavnom aplikacijom SimUDuck

Džo radi za kompaniju koja pravi veoma uspešnu igru simulacije jezera s patkama, *SimUDuck*. Igra prikazuje veliki broj vrsta pataka kako plivaju i kvaču. Prvi programeri sistema koristili su standardne OO tehnike i napravili jednu natklasu Duck (patka) od koje nasleđuju svi ostali tipovi pataka.





Tokom poslednje godine, kompanija je bila pod sve većim pritiskom konkurencije. Nakon sedmodnevne sesije mozganja uz izgranje golfa, direktori kompanije smatraju da je vreme za veliku inovaciju. Treba im nešto *zaista* impresivno što će pokazati na predstojećem sastanku deoničara na Mauiju *naredne sedmice*.

Ali patke sada treba da LETE

Direktori su rešili da simulatoru trebaju leteće patke da bi oduvali konkurenciju. I naravno, Džoov menadžer je rekao da Džou neće biti nikakav problem da nešto smisli za nedelju dana. „Ipak je on“, rekao je Džoov šef, „OO programer…*koliko bi to moglo biti teško*?“

Ali, nešto je pošlo naopako...

Džo, evo me na sastanku deoničara. Upravo su prikazali demo i u njemu je bilo **gumenih patkica** koje lete po ekranu. Mislio si da će to biti smešno?

Šta se desilo?

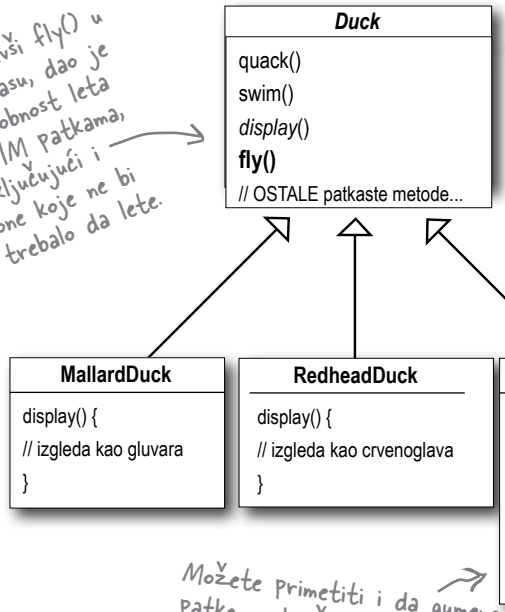
Džo nije primetio da ne bi trebalo *sve* potklase klase Duck da *lete*. Kada je dodao novo ponašanje natklasi Duck, dodao je i ponašanje koje *nije* odgovarajuće za neke potklase klase Duck. Sada ima leteće nežive objekte u programu SimUDuck.

Lokalizovano ažuriranje koda izazvalo je ne-lokalno sporedno dejstvo (leteće gumene patkice)!

Dobro, moj projekat ima malecni nedostatak. Ne znam zašto to prosto ne bi mogli da nazovu „posebnom osobinom". Pomalo je i slatko...

Ono što je Džo video kao odličnu upotrebu nasleđivanja u svrhu ponovne upotrebe, nije ispalo tako dobro kada je u pitanju održavanje.

Smestivši fly() u natklasu, dao je sposobnost leta SVIM patkama, uključujući i one koje ne bi trebalo da lete.



Možete primetiti i da gumene patke ne kvaču, pa je quack() pregaženo skvičanjem, „Squeak".

Džo razmišlja o nasleđivanju...

Mogao bih prosto da *preradim* metodu fly() za gumenu patku, kao što sam uradio sa metodom quack()...

RubberDuck

```
quack()  { // squeak}
display() { // rubber duck }
fly() {
  // preradi da ne radi ništa
}
```

Ali šta će se desiti kada programu dodamo drvene lažne patke? One ne treba ni da lete ni da kvaču...

DecoyDuck

```
quack() {
  // preradi da ne radi ništa
}

display() { // decoy duck}

fly() {
  // preradi da ne radi ništa
}
```

Evo još jedne klase u hijerarhiji; poput RubberDuck, ni ona ne leti, ali i ne kvače.

Naoštrite olovku

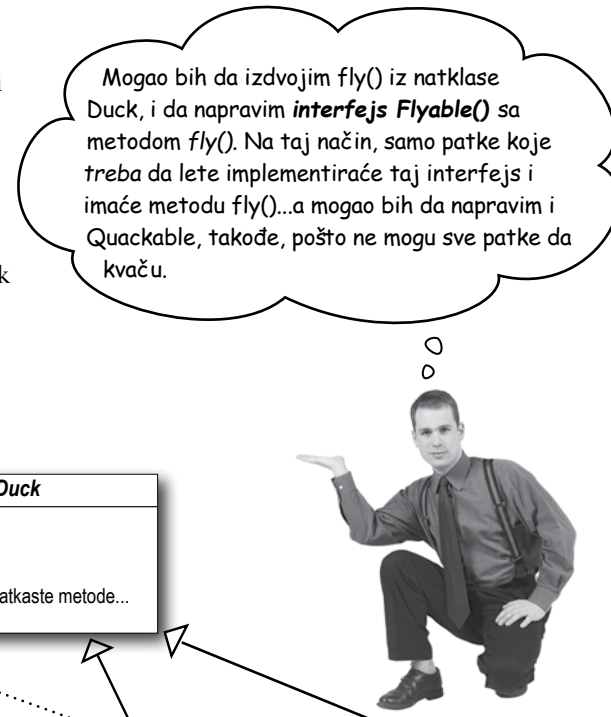
Šta od narednog čini *nasleđivanje* neodgovarajućim za definisanje ponašanja patke? (Odaberite sve što važi.)

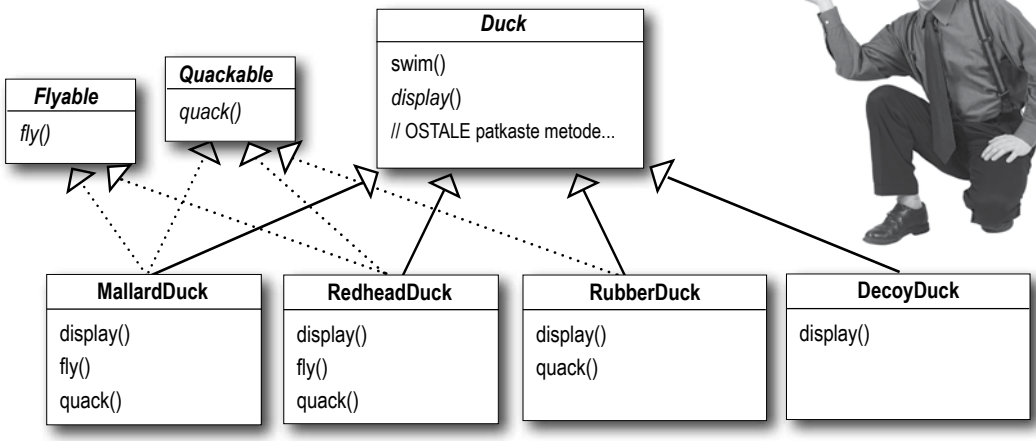
- ❑ A. Kod se duplicira u potklasama.
- ❑ B. Izmene ponašanja tokom izvršavanja su teške.
- ❑ C. Ne možemo zadati da patke igraju.
- ❑ D. Teško je saznati sva ponašanja pataka.
- ❑ E. Patke ne mogu istovremeno da lete i kvaču.
- ❑ F. Izmene mogu nenamerno da utiču na druge patke.

A šta je sa interfejsom?

Džo je shvatio da nasleđivanje verovatno nije rešenje, jer je upravo dobio dopis koji kaže da direktori sada žele da ažuriraju proizvod svakih šest meseci (na način koji još nisu smislili). Džo zna da će se specifikacije i dalje menjati i da će morati da razmatra, i verovatno redefiniše, fly() i quack() za svaku novu potklasu klase Duck koju bude dodao programu... *zauvek*.

Znači, treba mu čistiji način da samo *neki* (a ne *svi*) tipovi pataka lete ili kvaču.





Šta VI mislite o ovom dizajnu?

To je, ono, najgluplja ideja koju si mogao da smisliš. **Znače li ti nešto reči „dupliranje koda"?** Ako si mislio da je *redefinisanje nekoliko metoda* loše, kako bi se osećao kada bi morao da napraviš malu izmenu ponašanja letenja...*u svih 48 letećih potklasa klase Duck?!*

Šta biste vi uradili da ste Džo?

Znamo da ne bi trebalo da *sve* potklase imaju ponašanje leta ili kvakanja, pa nasleđivanje nije pravo rešenje. Ali mada implementiranje interfejsa Flyable i/ili Quackable rešava *deo* problema (više neće biti letećih gumenih pataka), ono potpuno uništava višekratnost koda za ta ponašanja, pa samo izaziva *drugačiju* noćnu moru za održavanje. I naravno, moglo bi postojati više vrsta ponašanja leta, čak i među patkama koje *lete*...

Možda sada očekujete da će projektni obrazac dojahati na belom konju i spasti vas nevolje. Ali to ne bi bilo zabavno! Ne, smislićemo rešenje na starinski način – *primenjujući dobre principe OO dizajna softvera.*

Zar ne bi bilo bajno kada bi postojao način da se softver izgradi tako da po potrebi možemo da ga izmenimo, sa najmanjim mogućim uticajem na postojeći kod? Mogli bismo manje vremena da provodimo prepravljajući kod, a više ubacujući zabavnije stvari u program...

Jedina konstanta u razvoju softvera

U redu, šta je jedina stvar na koju uvek možete računati pri razvoju softvera?

Bez obzira na to gde radite, šta pravite ili na kojem jeziku programirate, koja je jedina prava konstanta koja će vas uvek pratiti?

(upotrebite ogledalo da vidite odgovor)

Ma kako dobro da ste osmislili aplikaciju, tokom vremena ona mora da raste i menja se inače će *umreti*.

Naoštrite olovku

Mnoge stvari mogu pokrenuti promene. Navedite neke razloge zbog kojih ste morali da menjate kod u vašim aplikacijama (mi smo napisali par naših, za početak). Uporedite svoje odgovore sa rešenjem na kraju poglavlja pre nego što nastavite dalje.

Moji klijenti ili korisnici odluče da žele nešto drugo, ili žele novu funkcionalnost.

Moja kompanija je rešila da pređe na drugog prodavca baze podataka i on ih nabavlja od dobavljača koji koristi drugačiji format podataka. Uf!

Fokusiranje na problem...

Znamo da se nasleđivanje nije baš najbolje završilo pošto se ponašaje pataka menja u raznim potklasama, a nije odgovarajuće da *sve* potklase imaju ta ponašanja. Interfejsi Flyable i Quackable su ispočetka delovali obećavajuće – samo patke koje zaista lete biće Flyable, itd. – ali Javini interfejsi obično nemaju implementacioni kod, pa nema ponovnog korišćenja koda. U svakom slučaju, kad god morate da menjate ponašanje, često ste primorani da ga pratite i menjate u svim raznim potklasama gde je ono definisano, uz verovatnoću da ćete usput ubaciti i *nove* greške!

Srećom, postoji princip projektovanja upravo za tu situaciju.

Princip projektovanja

Prepoznajte aspekte aplikacije koji variraju i odvojte ih od onoga što ostaje isto.

Prvi od mnogo principa projektovanja. Njima ćemo se više baviti u celoj knjizi.

Drugim rečima, ako u kodu postoji neki aspekt koji se menja, recimo sa svakim novim zahtevom, tada znate da imate ponašanje koje treba da bude izvučeno i odvojeno od ostalih stvari koje se ne menjaju.

Evo još jednog načina na koji možete posmatrati ovaj princip: ***uzmite delove koji varijraju i kapsulirajte ih, tako da kasnije možete da izmenite ili proširite delove koji variraju, bez uticaja na one koji ne variraju.***

Iako je ovaj koncept veoma jednostavan, on čini osnovu gotovo svih projektnih obrazaca. Svi obrasci obezbeđuju način da se *nekom delu sistema omogući menjanje nezavisno od svih ostalih delova.*

U redu, vreme je da izvučemo pačje ponašanje van klasa Duck!

Uzmite ono što varira i „kapsulirajte ga“ tako da ne utiče na ostatak koda.

Rezultat? Manje slučajnih posledica izmena koda i više fleksibilnosti u vašim sistemima!

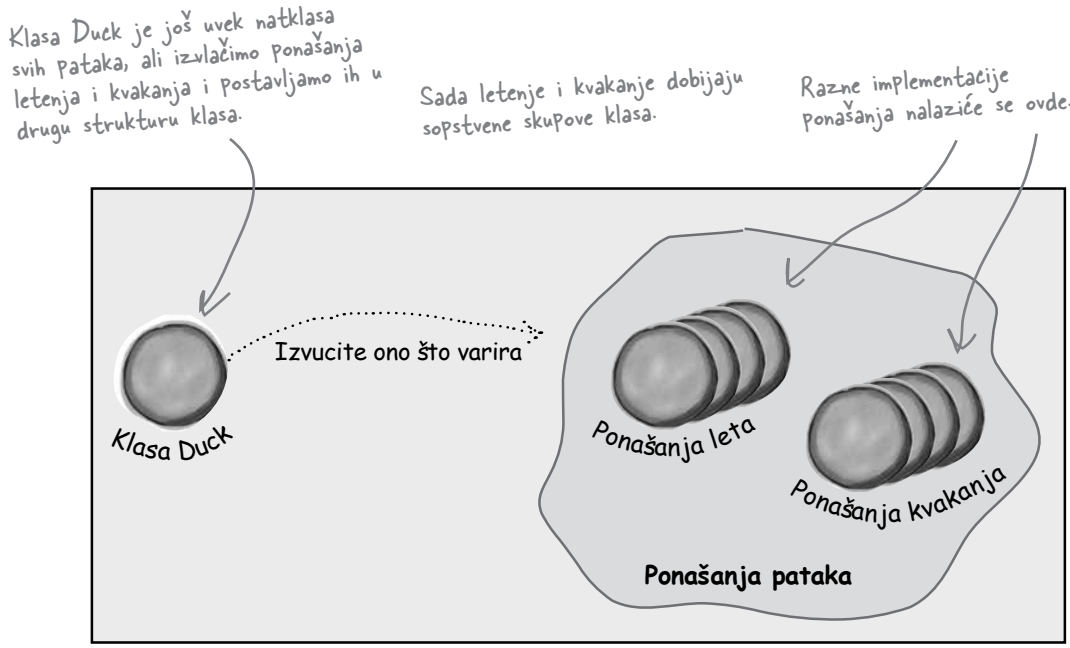
Odvajanje onoga što se menja od onoga što ostaje isto

Odakle da počnemo? Kako se čini, osim problema sa fly() i quack(), klasa Duck radi odlično i nema drugih delova koji često variraju ili se menjaju. Znači, osim nekoliko malih izmena, uglavnom ćemo ostaviti klasu Duck na miru.

Sad, da bismo odvojili „delove koji se menjaju od onih koji ostaju isti", napravićemo dva *skupa* klasa (potpuno odvojenih od Duck), jedan za *fly* i jedan za *quack*. Svaki skup klasa će čuvati sve implementacije odgovarajućeg ponašanja. Na primer, mogli bismo imati *jednu* klasu koja implementira *kvakanje*, *drugu* koja implementira *skvičanje*, i još jednu koja implementira *tišinu*.

Znamo da su fly() i quack() delovi klase Duck koji variraju zavisno od pataka.

Da bismo ta ponašanja odvojili od klase Duck, obe metode ćemo izvući ***iz klase* Duck i napraviti nov skup klasa koje će predstavljati svako ponašanje.**



Projektovanje ponašanja pataka

Kako ćemo osmisliti skup klasa koje implementiraju ponašanja leta i kvakanja?

Voleli bismo da stvari budu prilagodljive; na kraju krajeva, upravo nas je nefleksibilnost ponašanja pataka i dovela u nevolju. Znamo i da želimo da *dodeljujemo* ponašanja instancama klase Duck. Na primer, mogli bismo poželeti da napravimo novu instancu MallardDuck i da je inicijalizujemo sa posebnim *tipom* ponašanja leta. Kad smo kod toga, zašto se ne bismo postarali da možemo dinamički da menjamo ponašanje patke? Drugim rečima, trebalo bi u klase Duck da uključimo metode za zadavanje ponašanja tako da možemo da *menjamo* ponašanje leta patke MallardDuck u *vreme izvršavanja*.

S obzirom na ove ciljeve, pogledajmo naš drugi princip projektovanja:

Princip projektovanja

Programirajte u interfejs, ne u implementaciju.

Koristićemo interfejs da bismo predstavili svako ponašanje – na primer, FlyBehavior i QuackBehavior – i svaka implementacija *ponašanja* će primenjivati jedan od tih interfejsa.

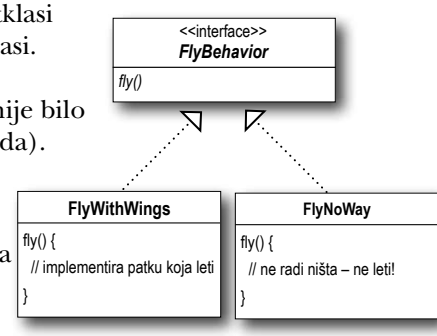
Tako ovaj put neće klase *Duck* implementirati interfejse leta i kvakanja. Umesto toga, napravićemo skup klasa čiji je čitav smisao postojanja da predstavljaju neko ponašanje (na primer, „skvičanje“), i onda će klasa *ponašanja*, a ne klasa Duck, implementirati interfejs ponašanja.

Ovo je suprotno načinu na koji smo ranije radili stvari, kada je ponašanje poticalo ili iz konkretne implementacije u natklasi Duck, ili iz specijalizovane implementacije u samoj potklasi. U oba slučaja smo se oslanjali na *implementaciju*. Bili smo ograničeni da koristimo te konkretne implementacije i nije bilo prostora za menjanje ponašanja (osim pisanja još više koda).

Sa novim dizajnom, potklase klase Duck koristiće ponašanje predstavljeno *interfejsom* (FlyBehavior i QuackBehavior), te tako stvarna *implementacija* ponašanja (drugim rečima, konkretno ponašanje uprogramirano u klasu koja implementira FlyBehavior ili QuackBehavior) neće biti zaključana u potklasi klase Duck.

Od sada će se ponašanja klase Duck nalaziti u odvojenoj klasi – klasi koja implementira određeni interfejs ponašanja.

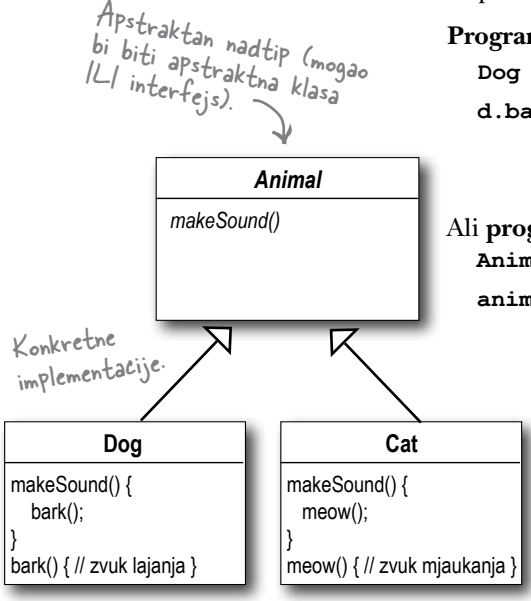
Na taj način, klase Duck neće morati da znaju ništa o detaljima implementacije sopstvenog ponašanja.



„Programirajte u *interfejs*“ zapravo znači „programirajte *u nadtip*“.

Reč *interfejs* je ovde preterana. Tu je *koncept* interfejsa, ali tu je i Javina *struktura* interfejsa. Možete da *programirate u interfejs* ne morajući zaista da koristite Javin interfejs. Poenta je da iskoristite polimorfizam programirajući u nadtip, tako da stvarni objekat koji se izvršava nije zaključan u kodu. Mogli bismo da preformulišemo „programirajte u nadtip“ u „deklarisani tip promenljivih trebalo bi da bude nadtip, obično apstraktna klasa ili interfejs, tako da objekti koji su dodeljeni tim promenljivama mogu da budu bilo koja konkretna implemetnacija nadtipa, što znači da klasa koja ih deklariše ne mora da zna za stvarne tipove objekata“!

Sve ovo ste verovatno već čuli, ali da bismo bili sigurni da mislimo na istu stvar, evo jednostavnog primera upotrebe polimorfnog tipa – zamislite jednu apstraktnu klasu Animal (životinja), sa dve konkretne implementacije Dog (pas) i Cat (mačka).



Programiranje u implementaciju bi bilo:

```
Dog d = new Dog();
d.bark();
```

Deklarisanje promenljive „d“ kao tip Dog (konkretna implementacija klase Animal) primorava nas da programiramo u konkretnu implementaciju.

Ali **programiranje u interfejs/nadtip** bilo bi:

```
Animal animal = new Dog();
animal.makeSound();
```

Znamo da je to Dog, ali sada možemo polimorfno da koristimo referencu animal.

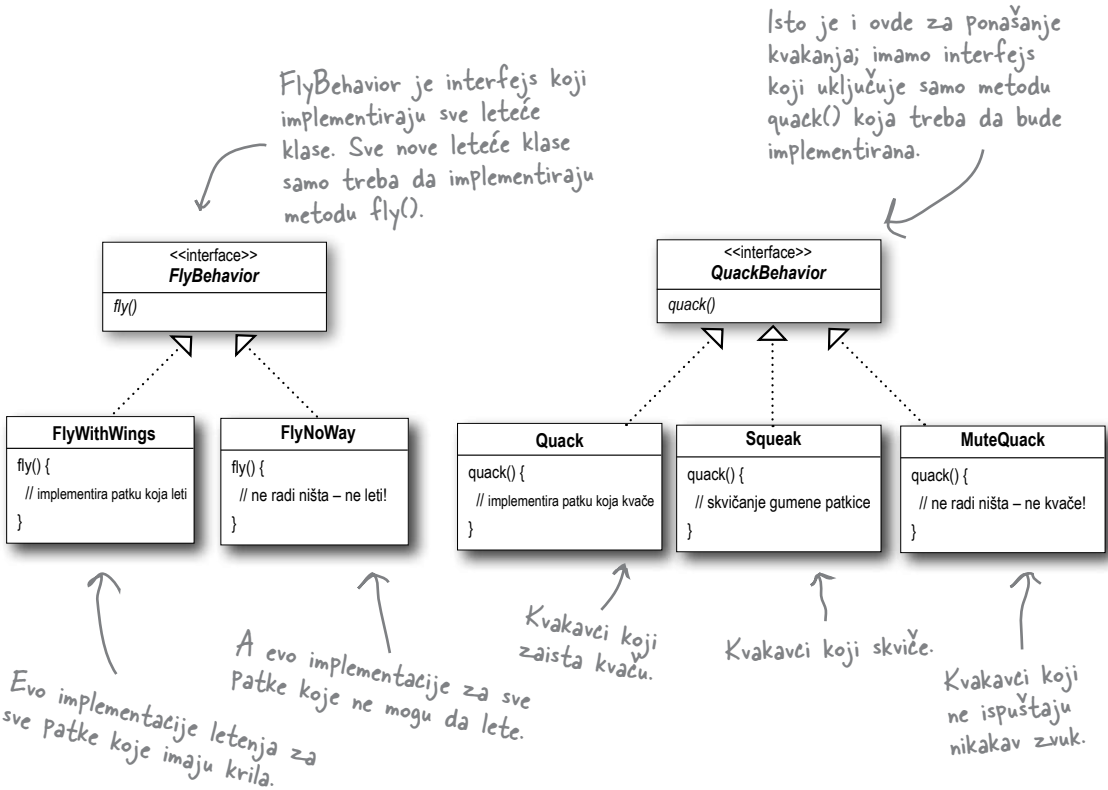
Još bolje bi bilo da, umesto da u kod upišemo instanciranje podtipa (kao što je new Dog()), **dodelimo konkretan implementacioni objekat pri izvršavanju:**

```
a = getAnimal();
a.makeSound();
```

Ne znamo KOJI je stvarni podtip životinje... zanima nas samo da ona zna kako da reaguje na makeSound().

Implementiranje pačjih ponašanja

Ovde imamo dva interfejsa, FlyBehavior i QuackBehavior, kao i odgovarajuće klase koje implementiraju svako konkretno ponašanje:



Sa ovakvim projektovanjem, drugi tipovi objekata mogu da iskoriste naša ponašanja leta i kvakanja jer ta ponašanja više nisu sakrivena u klasama Duck!

A možemo dodati nova ponašanja ne menjajući nijednu postojeću klasu ponašanja i ne dirajući nijednu klasu Duck koja *koristi* ponašanja leta.

Tako dobijamo prednosti PONOVNE UPOTREBE bez tereta koji donosi nasleđivanje.

ne postoje Glupa pitanja

P: Da li uvek moram prvo da implementiram aplikaciju, vidim gde se stvari menjaju, a onda da se vraćam da bih odvojio i kapsulirao te stvari?

O: Ne uvek; često, kada projektujete aplikaciju, možete da predivdite oblasti koje će varirati, i u kod ugraditi fleksibilnost koja će se pobrinuti za njih. Primetićete da principi i obrasci mogu da budu primenjeni u bilo kojoj fazi razvojnog ciklusa.

P: Da li bi i Duck trebalo da bude interfejs?

O: Ne u ovom slučaju. Kao što ćete videti kada sve povežemo, imaćemo koristi od toga što Duck nije interfejs i što određene patke, kao što je MallardDuck, nasleđuju zajednička svojstva i metode. Pošto smo ono što varira uklonili iz nasleđivanja od klase Duck, možemo bez problema da koristimo prednosti ove strukture.

P: Deluje pomalo čudno da imamo klasu koja je samo ponašanje. Zar klase ne bi trebalo da predstavljaju stvari? Zar klase ne bi trebalo da imaju i stanje I ponašanje?

O: U OO sistemu, da, klase predstavljaju stvari koje uopšteno govoreći imaju i stanje (promenljive instanci) i metode. U ovom slučaju, stvar je ponašanje. Ali čak i ponašanje i dalje može da ima stanje i metode; ponašanje leta bi moglo imati promenljive instanci koje predstavljaju atribute za ponašanje leta (lepet krila u minutu, maksimalna visina, brzina itd.).

Naoštrite olovku

1. Koristeći naš novi projekat, šta biste uradili ako bi aplikaciji SimUDuck trebalo da dodate letenje na raketni pogon?

2. Možete li da se dosetite klase koja bi koristila ponašanje Quack, a da nije patka?

Odgovori:

1) Napravite klasu FlyRocketPowered koja implementira interfejs FlyBehavior.

2) Jedan primer, vabilica (naprava koja proizvodi pačje zvukove).

Integrisanje pačjih ponašanja

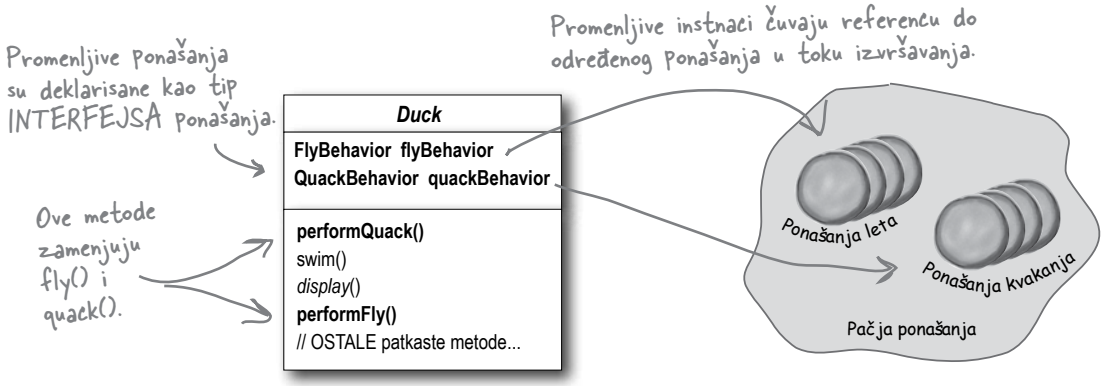
Evo ključa: Duck će sada delegirati svoja ponašanja leta i kvakanja, umesto da koristi metode kvakanja i leta definisane u klasi Duck (ili potklasi).

Evo kako:

1 **Prvo ćemo dodati dve promenljive instanci tipa FlyBehavior i QuackBehavior** – nazovimo ih flyBehavior i quackBehavior. Svaki konkretan objekat patke dodeljivaće tim promenljivama *posebno* ponašanje prilikom izvršavanja, kao što je FlyWithWings (let krilima) za letenje ili Squeak (skvičanje) za kvakanje.

Pored toga, uklonićemo metode fly() i quack() iz klase Duck (i svih potklasa) jer smo to ponašanje premestili u klase FlyBehavior i QuackBehavior.

Zamenićemo fly() i quack() u klasi Duck dvema sličnim metodama, nazvanim performFly() i performQuack(); u nastavku ćete videti kako one rade.



2 **Sada implementiramo performQuack():**

```
public abstract class Duck {
   QuackBehavior quackBehavior;
   // još

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

Svaka patka Duck ima referencu do nečega što implementira interfejs QuackBehavior.

Umesto da se sam bavi ponašanjem kvakanja, objekat Duck delegira to ponašanje na objekat koji referencira quackBehavior.

Prilično jednostavno, a? Da bi izvela kvakanje, Duck traži od objekta koji je referenciran u quackBehavior da kvače umesto nje. U ovom delu koda ne marimo za to *koja je vrsta* objekta konkretna patka, ***zanima nas samo da objekat zna da kvače – quack()!***

Još integracije...

❸ Dobro, vreme je da brinemo o tome **kako** se promenljive instanci flyBehavior i quackBehavior **zadaju**. Pogledajmo klasu MallardDuck:

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("Ja sam prava gluvara");
    }
}
```

MallardDuck koristi klasu Quack da bi se pobrinula za svoje kvakanje, pa kada se pozove performQuack(), odgovornost za kvakanje se delegira objektu Quack i dobijamo stvarno kvakanje.

Ona koristi FlyWithWings kao svoj tip ponašanja FlyBehavior.

Sećate se, MallardDuck nasleđuje promenljive instanci quackBehavior i flyBehavior od klase Duck.

Kvakanje za MallardDuck je pravo **kvakanje** žive patke, nije **skvičanje** i nije **nemo kvakanje** (eng. *mute quack*). Kada se MallardDuck instancira, njen konstruktor inicijalizuje nasleđenu promenljivu instance quackBehavior na novu instancu tipa Quack (konkretnu implementacionu klasu QuackBehavior).

Isto važi i za ponašanje leta patke – konstruktor za MallardDuck inicijalizuje nasleđenu promenljivu instance flyBehavior instancom tipa FlyWithWings (konkretnom implementacionom klasom FlyBehavior).

Dobro uočeno, to je upravo ono što radimo... *zasad*.

Kasnije u knjizi imaćemo u našoj kutiji sa alatom više obrazaca koji će moći da nam pomognu da to popravimo.

Ipak, obratite pažnju na to da iako *zaista* podešavamo ponašanja na konkretne klase (tako što instanciramo klase ponašanja kao što je Quack ili FlyWithWings i dodeljujemo ih promenljivoj koja referencira ponašanje), to bismo mogli *lako* da promenimo u vreme izvršavanja.

Znači, ovde ima još mnogo fleksibilnosti. S druge strane, nismo se baš proslavili sa fleksibilnim inicijalizovanjem promenljivih instance. Ali razmislite o tome: pošto je promenljiva instance quackBehavior tip interfejsa, mogli bismo (pomoću magije polimorfizma) dinamički da dodelimo drugačiju implementacionu klasu QuackBehavior u vreme izvršavanja.

Razmislite neko vreme o tome kako biste implementirali patku tako da njeno ponašanje može da se menja prilikom izvršavanja. (Za nekoliko strana ćete videti kod koji to radi.)

Testiranje koda Duck

1 Upišite i kompajlirajte donju klasu Duck (Duck.java), i klasu MallardDuck napisanu dve strane ranije (MallardDuck.java).

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   public Duck() { }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

   public void swim() {
      System.out.println("Sve patke plutaju, čak i lažne!");
   }
}
```

Deklarišite dve promenljvie referenci za tipove interfejsa ponašanja. Sve potklase patki (u istom paketu) nasleđivaće ih.

Delegirajte klasi ponašanja.

2 Upišite i kompajlirajte interfejs FlyBehavior (FlyBehavior.java) i dve klase za implementaciju ponašanja (FlyWithWings.java i FlyNoWay.java).

```
public interface FlyBehavior {
   public void fly();
}
```

Interfejs koji sve klase implementiraju za ponašanje leta.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("Ja letim!!");
   }
}
```

Implementacija ponašanja leta za patke koje zaista LETE...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("Ne mogu da letim");
   }
}
```

Implementacija ponašanja leta za patke koje NE lete (kao što su gumene i lažne patke).

Testiranje koda Duck, nastavak...

3 Upišite i kompajlirajte interfejs QuackBehavior (QuackBehavior.java) i tri klase za implementaciju ponašanja (Quack.java, MuteQuack.java i Squeak.java).

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Kvak");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Tišina >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Skvik");
   }
}
```

4 Upišite i kompajlirajte klasu za testiranje (MiniDuckSimulator.java).

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

Ovo poziva nasleđenu metodu performQuack() patke MallardDuck, koja potom delegira objektovom ponašanju QuackBehavior (tj, poziva quack() na patkinoj nasleđenoj referenci quackBehavior).

Potom činimo isto sa nasleđenom metodom performFly() patke MallardDuck.

5 Izvršite kod!

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Kvak
Ja letim!!
```

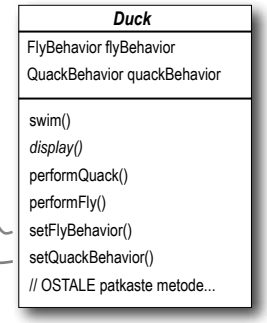
Dinamičko zadavanje ponašanja

Kakva je šteta što je sav ovaj dinamički talenat ugrađen u naše patke, a mi ga ne koristimo! Zamislite da hoćete da zadate tip ponašanja patke kroz metodu za zadavanje u klasi Duck, umesto instanciranjem ponašanja u konstruktoru patke.

❶ Dodajte dve nove metode klasi Duck:

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Ove metode možemo da pozovemo bilo kad kada poželimo da promenimo ponašanje patke u letu.

Napomena urednika: neopravdana igra reči – ispraviti

❷ Napravite nov tip Duck (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("Ja sam model patke");
    }
}
```

Naš model patke počinje svoj život prizemljen... bez načina da leti.

❸ Napravite nov tip FlyBehavior type (FlyRocketPowered.java).

To je u redu, pravimo ponašanje leta na raketni pogon.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("Ja letim raketom!");
    }
}
```

4 **Izmenite klasu za testiranje (MiniDuckSimulator.java), dodajte ModelDuck, i obezbedite patki ModelDuck raketu.**

```
public class MiniDuckSimulator {

    public static void main(String[] args) {

          Duck mallard = new MallardDuck();

          mallard.performQuack();

          mallard.performFly();

          Duck model = new ModelDuck();

          model.performFly();

          model.setFlyBehavior(new FlyRocketPowered());

          model.performFly();

    }

}
```

Prvi poziv za performFly() delegira objketu flyBehavior podešenom u konstruktoru za ModelDuck, koja je instanca FlyNoWay.

Ovo poziva nasleđenu metodu za zadavanje ponašanja modela, i...voilà! Model odjednom ima sposobnost letenja na raketni pogon!

Ako radi, model patke je dinamički promenio svoje ponašanje leta! OVO ne možete da uradite ako se implementacija nalazi unutar klase Duck.

5 **Pokrenite ga!**

```
File  Edit  Window  Help  Yabbadabbadoo
%java MiniDuckSimulator
Kvak
Ja letim!!
Ne mogu da letim
Ja letim raketom!
```

Da biste promenili ponašanje patke tokom izvršavanja, samo pozovite njenu metodu za zadavanje tog ponašanja.

Šira slika kapsuliranih ponašanja

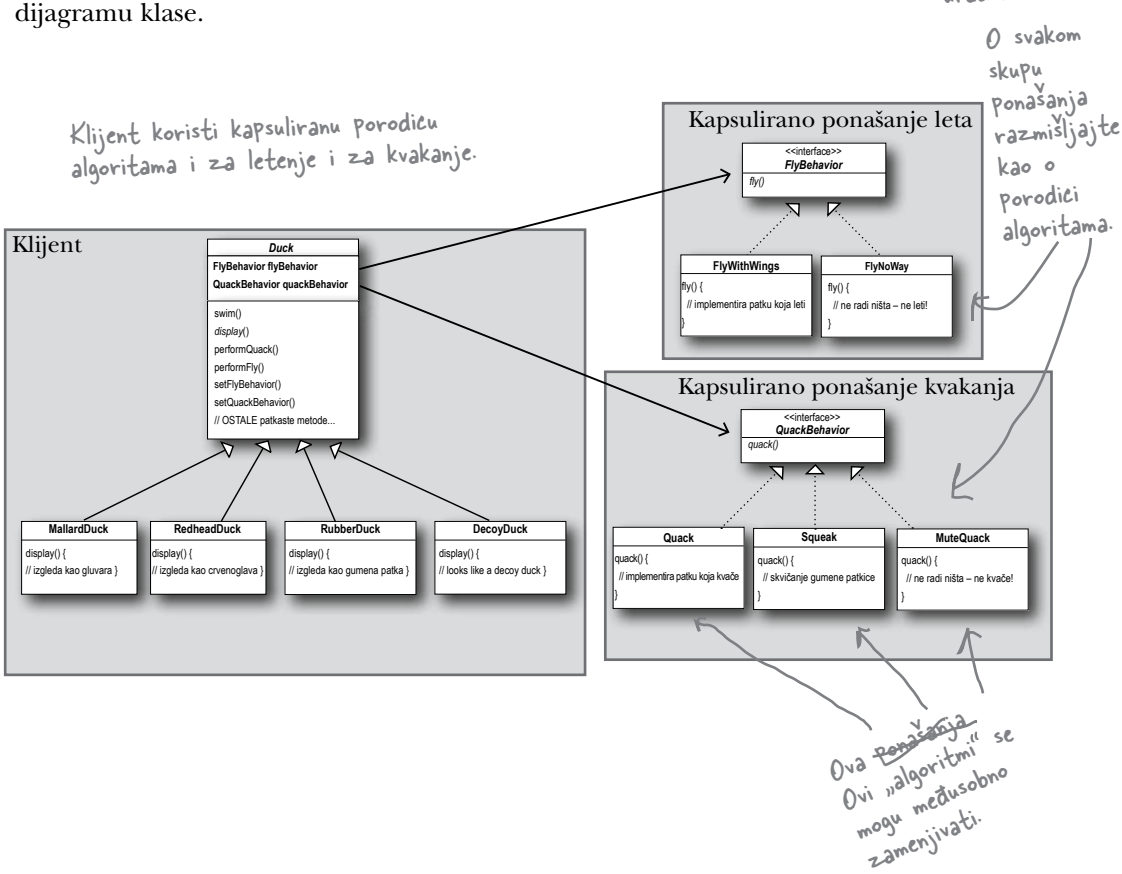
Pošto smo već duboko zaronili u projekat simulatora patke, vreme je da izronimo da udahnemo vazduh i osmotrimo širu sliku.

Ispod je cela prepravljena struktura klase. Imamo sve što biste mogli očekivati: patke koje nasleđuju klasu Duck, ponašanja leta koja implementiraju FlyBehavior i ponašanja kvakanja koja implementiraju QuackBehavior.

Primetićete i da smo počeli da opisujemo stvari na nešto drugačiji način. umesto da razmišljamo o ponašanjima patke kao o *skupu ponašanja*, počećemo da o njima razmišljamo kao o *porodici algoritama*. Razmislite o tome: u projektu SimUDuck, algoritmi predstavljaju stvari koje bi patke radile(razne načine kvakanja ili letenja), ali jednako lako bismo istu tehniku mogli da koristitimo za skup klasa koje implementiraju načine da se izračuna porez na promet u raznim državama.

Obratite pažnju na *odnose* između klasa. Zapravo, uzmite olovku i napišite odgovarajući odnos (JESTE, IMA i IMPLEMENTIRA) na svakoj strelici u dijagramu klase.

Obavezno uradite ovo.



IMA može biti bolje nego JESTE

Odnos IMA (eng. *HAS*) je zanimljiv: svaka patka ima FlyBehavior i QuackBehavior kojima delegira letenje i kvakanje.

Kada na ovaj način spojite dve klase, vi koristite **kompoziciju**. Umesto da *nasleđuju* svoje ponašanje, patke dobijaju svoje ponašanje tako što su *ukomponovane* sa krutim objektom ponašanja.

To je bitna tehnika; zapravo, to je osnova našeg trećeg principa projektovanja:

Princip projektovanja

Dajte prednost kompoziciji nad nasleđivanjem.

Kao što ste videli, pravljenje sistema pomoću kompozicije pruža mnogo više prilagodljivosti. Ne samo da vam omogućava da kapsulirate porodicu algoritama u njihov sopstveni skup klasa, već omogućava i da ***menjate ponašanje tokom izvršavanja*** sve dok objekat s kojim pravite kompoziciju implementira odgovarajući interfejs ponašanja.

Kompozicija se koristi u *mnogim* projektnim obrascima i u ovoj knjizi ćete pročitati još mnogo o o njenim prednostima i nedostacima.

Vabilica za patke (eng. *duck call*) uređaj je koji lovci koriste da bi imitirali zov (kvakanje) pataka. Kako biste implementirali sopstvenu vabilicu koja *ne* nasleđuje od klase Duck?

Guru i učenik...

Guru: *Reci mi šta si naučio o objektno orijentisanim načinima.*

Učenik: *Guru, naučio sam da je objektno orijentisani način posvećen ponovnom korišćenju.*

Guru: *Nastavi...*

Učenik: *Guru, putem nasleđivanja sve dobre stvari se mogu ponovo koristiti i tako drasitčno skraćujemo vreme za razvoj, kao što brzo sasecamo bambus u šumi.*

Guru: *Da li se na kod troši više vremena **pre** ili **nakon** završavanja razvoja?*

Učenik: *Odgovor je **nakon**, Guru. Uvek trošimo više vremena održavajući i menjajući softver nego na njegov početni razvoj.*

Guru: *Dakle, da li bi trebalo uložiti trud u ponovnu upotrebljivost **pre nego** u održivost i proširivost?*

Učenik: *Guru, verujem da u tome ima istine.*

Guru: *Vidim da treba još mnogo da učiš. Idi i još malo meditiraj o nasleđivanju. Kao što si video, nasleđivanje ima svoje probleme, i postoje drugi načini da se postigne višekratnost koda.*

Kad smo kod projektnih obrazaca...

Čestitamo vam na prvom obrascu!

Upravo ste primenili svoj prvi projektni obrazac – obrazac STRATEGY. Baš tako, koristili ste obrazac Strategy (eng. *Strategy Pattern*) da biste prepravili aplikaciju SimUDuck.
Zahvaljujući tom obrascu, simulator je spreman za svaku izmenu koju bi direktori mogli da zakuvaju na svom sledećem poslovnom putovanju na Maui.

Pošto smo vas naterali da pređete dug put da biste ga naučili, evo zvanične definicije ovog obrasca:

> **Obrazac Strategy** definiše porodicu algoritama, kapsulira svaku od njih i čini ih međusobno zamenjivim. Strategija omogućava da algoritam varira nezavisno od klijenata koji je koriste.

Upotrebite OVU definiciju kada budete hteli da zadivite drugare i utičete na bitne direktore.

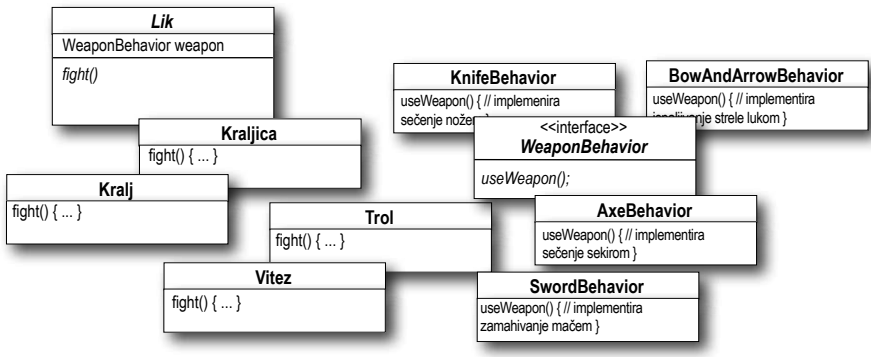
Projektna zagonetka

Ispod ćete naći zbrku klasa i interfejsa za akcionu avanturističku igru. Pronaći ćete klase za karaktere likova igre i klase za ponašanja oružja koja likovi mogu da koriste u igri. Svaki lik može da koristi samo jedno oružje u jednom trenutku, ali može da promeni oružje bilo kad tokom igre. Vaš posao je da sve to razvrstate...

(Rešenja su na kraju poglavlja.)

Vaš zadatak:

1. Poređajte klase.
2. Označite jednu apstraktnu klasu, jedan interfejs i osam klasa.
3. Nacrtajte strelice između klasa.
   a. Ovakvu strelicu nacrtajte za nasleđivanje („proširuje“).
   b. Ovakvu strelicu nacrtajte za interfejs („implementira“).
   c. Ovakvu strelicu nacrtajte za IMA.
4. Postavite metodu setWeapon() u odgovarajuću klasu.




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Načuli smo u lokalnom restoranu...

Koja je razlika između ove dve porudžbine? Nikakva! Potpuno su iste osim što Alis koristi dvaput više reči i iskušava strpljenje mrzovoljnog prodavca brze hrane.

Šta je Flo dobila što Alis nije? **Zajednički rečnik** s prodavcem brze hrane. Ne samo da joj to olakšava komunikaciju s prodavcem, već i prodavac ima manje toga za pamćenje pošto su mu svi obrasci večere već u glavi.

Projektni obrasci vam daju zajednički rečnik sa ostalim programerima. Kada imate rečnik, lakše ćete komunicirati s drugim programerima i inspirisati one koji ne poznaju obrasce da počnu da ih uče. Oni takođe podižu vaše razmišljanje o arhitekturi na viši nivo omogućavajući vam da **razmišljate na nivou *obrasca***, a ne na detaljnom nivou *objekta*.

Načuli smo za susednim stolom...

I tako, napravio sam ovu klasu za emitovanje. Ona evidentira sve objekte koji je slušaju i svaki put kada se pojavi neki nov podatak ona šalje poruku svakom slušaocu. Kul je to što slušaoci mogu bilo kada da se pridruže emitovanju ili čak i da se uklone. Stvarno je dinamična i labavo vezana!

SNAGA UMA

Možete li da se dosetite još nekog zajedničkog rečnika koji se koriste van OO projektovanja i razgovora u restoranu? (Mala pomoć: šta mislite o automehaničarima, drvodeljama, vrhunskim kuvarima i kontrolorima leta?) Koji kvaliteti se prenose zajedno sa žargonskim jezikom?

Možete li se setiti aspekata OO projektovanja koji se prenose zajedno sa imenima obrazaca? Koji se kvaliteti prenose sa imenom „obrazac Strategy“?

Rik, zašto prosto ne kažeš da koristiš **obrazac Observer**?

Tako je. Ako komuniciraš u obrascima, drugi programeri odmah i *precizno* znaju koji dizajn opisuješ. Samo nemoj da zapadneš u groznicu obrazaca... znaćeš da je imaš kada počneš da koristio obrasce za Zdravo svete...

Moć zajedničkog rečnika obrazaca

Kada komunicirate koristeći obrasce, radite više od prostog deljenja ŽARGONA.

Zajednički rečnici obrazaca su MOĆNI. Kada sa drugim programerom u svom timu razgovarate koristeći obrasce, ne prenosite samo ime obrasca već i ceo skup kvaliteta, karakteristika i ograničenja koje obrazac predstavlja.

„Mi koristimo obrazac Strategy da bismo implementirali razna ponašanja pataka." Ovo vam govori da je ponašanje pataka kapsulirano u sopstveni skup klasa koje se mogu lako proširivati i menjati, čak i tokom izvršavanja, ako je potrebno.

Obrasci omogućavaju da kažete više sa manje reči. Kada koristite obrazac u opisu, drugi programeri će brzo precizno znati koji dizajn imate na umu.

Razgovaranje na nivou obrazaca omogućava da duže ostanete „u projektu". Govoreći o softverskim sistemima koristeći obrasce omogućeno vam je da održavate diskusiju na niovu projekta, ne morajući da zalazite u dubine detalja implementiranja objekata i klasa.

Koliko je projektnih sastanaka kojima ste prisustvovali brzo spalo na detalje implementacije?

Zajednički rečnici mogu da dodaju turbo-punjenje vašem razvojnom timu. Tim koji je dobro upućen u projektne obrasce brže će se kretati sa manje prostora za nerazumevanje.

Kada vaš tim počne da deli projektne ideje i iskustva u pojmovima obrazaca, izgradićete zajendicu korisnika obrazaca.

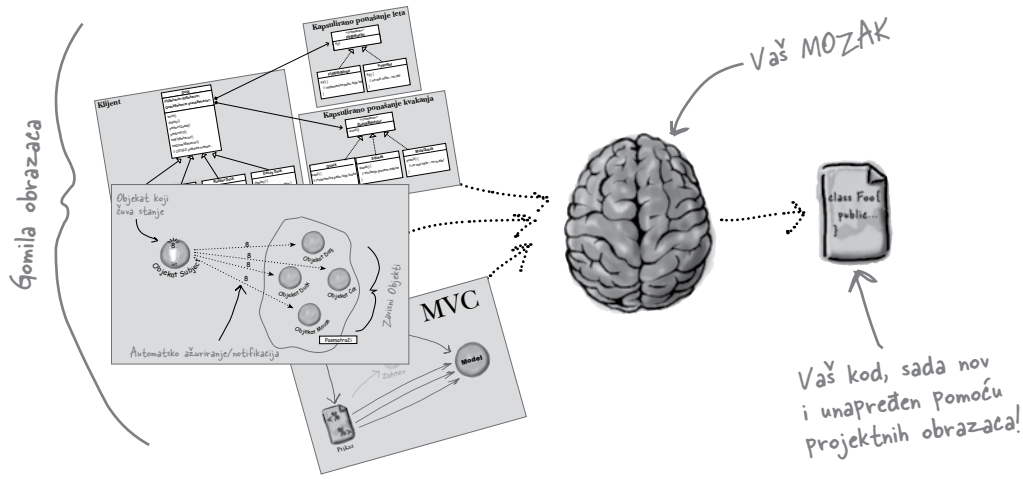
Zajednički rečnici ohrabruju mlađe programere da se ubrzaju. Mlađi programeri se ugledaju na iskusne projektante. Kada stariji projektanti koriste projektne obrasce, i mlađi su motivisani da ih nauče. Izgradite zajednicu korisnika obrazaca u svojoj organizaciji.

Mogli biste pokrenuti grupu za učenje obrazaca u svojoj organizaciji. Možda biste bili i plaćeni dok učite...

Kako da koristim projektne obrasce?

Svi smo koristili gotove biblioteke i radne okvire. Uzmemo ih, napišemo nekolicinu koda naspram njihovih API-ja, kompajliramo ih u naše programe i izvučemo korist iz gomile kodova koji je napisao neko drugi. Pomislite na Javine API-je i sve funkcionalnosti koje vam oni pružaju: network, GUI, IO itd. Biblioteke i radni okviri su prošli dug put do razvojnog modela u kojem možemo prosto da izaberemo komponente i da ih uklopimo. Međutim... oni nam ne pomažu da strukturišemo sopstvene aplikacije tako da se lakše razumeju, održavaju i budu fleksibilnije. Tu na scenu stupaju projektni obrasci.

Projektni obrasci ne ulaze direktno u vaš kod, oni prvo ulaze u vaš MOZAK. Kada napunite mozak dobrim funkcionalnim poznavanjem obrazaca, možete početi da ih primenjujete u sopstvenim projektima i da prepravljate svoj stari kod kada primetite da počinje da degradira u neprilagodljivi rusvaj.



ne postoje Glupa pitanja

P: Ako su projektni obrasci toliko dobri, zašto neko ne bi napravio biblioteku sa njima, pa da ne moram ja?

O: Projektni obrasci su viši nivo od biblioteka. Projektni obrasci nam govore kako da strukturišemo klase i objekte da bismo rešili određene probleme i naš je posao da prilagodimo dizajn projekta da odgovara našoj konkretnoj aplikaciji.

P: Zar nisu i biblioteke i radni okviri takođe i projektni obrasci?

O: Radni okviri i biblioteke nisu projektni obrasci; oni obezbeđuju specifične implementacije koje ulančavamo u naš kod. Ponekad, međuim, biblioteke i radni okviri koriste projektne obrasce u svojim implementacijama. To je odlično, jer kada jednom shvatite projektne obrasce, brže ćete razumeti API-je koji su strukturisani oko njih.

P: Znači, nema biblioteka sa projektnim obrascima?

O: Nema, ali kasnije ćete učiti o katalozima obrazaca sa listama obrazaca koje možete primeniti na svoje aplikacije.

Skeptična programerka

Prijateljski guru za obrasce

Programerka: Dobro, hmm, ali zar nije sve ovo samo dobar objektno-orijentisani dizajn; mislim, sve dok se pridržavam kapsuliranja i znam za apstrakciju, nasleđivanje i polimorfizam, zar zaista moram da razmišljam o projektnim obrascima? Zar to nije prilično jednostavno? Zar nisam zbog ovoga išla na sve one OO kurseve ? Mislim da su projektni obrasci korisni za ljude koji nisu upoznati sa dobrim OO dizajnom.

Guru: A, ovo je jedna od pravih zabluda objektno-orijentisanog programiranja: da ćete time što poznajete OO osnove automatski biti dobri u izgradnji fleksibilnih, višekratnih i održivih sistema.

Programerka: Neću?

Guru: Nećeš. Kako ispada, konstruisanje OO sistema koji imaju ta svojstva nije uvek očigledno i otkriva se tek napornim radom.

Programerka: Mislim da počinjem da shvatam. Ovi, ponekad neočigledni, načini konstruisanja objektno-orijentisanih sistema sakupljeni su...

Guru: ...da, u skup obrazaca koji se naziva projektnim obrascima.

Programerka: Znači, poznajući obrasce, mogu da preskočim naporan rad i odmah pređem na dizajn projekta koji uvek ispravno radi?

Guru: Da, do neke mere, ali zapamti, dizajn je umetnost. Uvek postoje kompromisi. Međutim, ako se pridržavaš dobro promišljenih, oprobanih projektnih obrazaca, bićeš u velikoj prednosti.

Programerka: Šta da radim ako ne mogu da nađem obrazac?

Upamti, poznavanje koncepata kao što su apstrakcija, nasleđivanje i polimorfizam ne čine te dobrim objektno orijentisanim projektantom. Projektni guru razmišlja o tome kako da napravi prilagodljive projekte koji su održivi i mogu da se izbore sa promenom.

Guru: Postoje neki principi objektno orijentisanog programiranja koji su osnova obrazaca i njihovo poznavanje će ti pomoći da se izboriš kada ne možeš da pronađeš obrazac koji odgovara tvom problemu.

Programerka: Principi? Misliš osim apstrakcije, kapsulacije i...

Guru: Da, jedna od tajni pravljenja održivih OO sistema jeste razmišljanje o tome kako bi se oni mogli izmeniti u budućnosti, a principi se bave tim pitanjima.

Alatke za projektovanje

Gotovo da ste završili prvo poglavlje! Već ste stavili nekoliko alata u svoju OO kutiju s alatom; hajde da ih nabrojimo pre nego što pređemo na Poglavlje 2.

OO osnove

Apstrakcija

Kapsuliranje

Polimorfizam

Nasleđivanje

Pretpostavljamo da poznajete OO osnove kao što su apstrakcija, kapsuliranje, polimorfizam i nasleđivanje. Ako ste malo zarđali, izvucite svoju omiljenu knjigu o objektno orijentisanom programiranju i podsetite se, a onda ponovo preletite preko ovog poglavlja.

OO principi

Kapsuliraj ono što varira.

Daj prednost kompoziciji nad nasleđivanjem.

Programiraj u interfejse, ne u implementacije.

Usput ćemo se detaljnije baviti ovima, a listi ćemo dodati i još neke.

OO obrasci

Strategy – definiše porodicu algoritama, kapsulira svaku od njih i čini ih međusobno zamenjivim. Strategija omogućava da algoritam varira nezavisno od klijenata koji je koriste.

Kroz celu knjigu razmišljajte o tome kako se obrasci oslanjaju na OO osnove i Principe.

Jedan gotov, ima još mnogo!

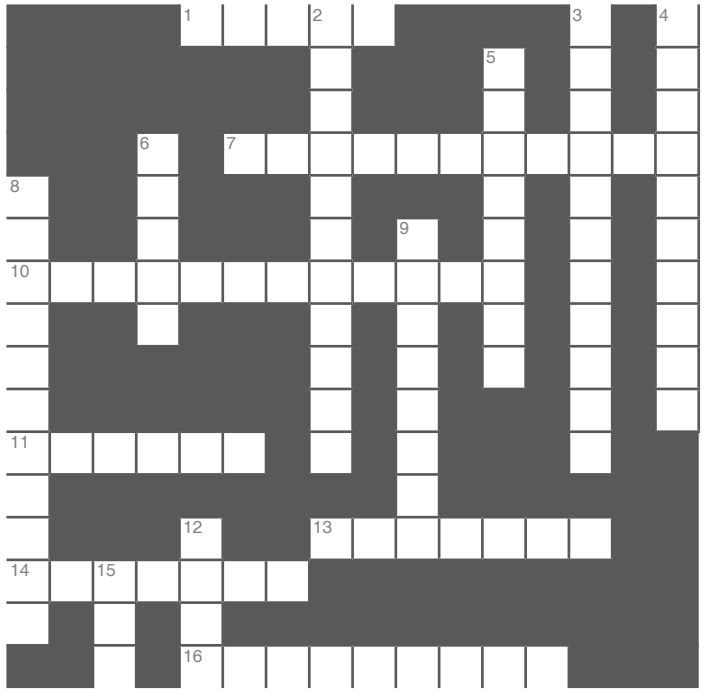
PRAVO U CENTAR

- Poznavanje OO osnova vas ne čini dobrim OO programerom.
- Dobar OO dizajn je višekratan, proširiv i održiv.
- Obrasci vam pokazuju kako da izgradite sisteme sa dobrim kvalitetima OO dizajna.
- Obrasci su dokazano objektno orijentisano iskustvo.
- Obrasci vam ne daju kod već opšta rešenja za projektne probleme. Vi ih primenjujete u svojoj konkretnoj aplikaciji.
- Obrasci nisu *izmišljeni*, oni su *otkriveni*.
- Većina obrazaca i principa bavi se pitanjima *promene* u sotveru.
- Većina obrazaca omogućava da neki deo sistema varira nezavisno od svih ostalih delova.
- Često pokušavamo da uzmemo ono što varira u sistemu i da ga kapsuliramo.
- Obrasci obezbeđuju zajednički jezik koji može da maksimira vrednost vaše komunikacije sa drugim programerima.

Ukrštenica projektnih obrazaca

Hajde da uposlimo vašu desnu hemisferu mozga.

Ovo je standardna ukrštenica; sva rešenja su reči iz ovog poglavlja.



VODORAVNO

1. Životinja iz igre SimUDuck.
7. Primenjuje se na ono što varira.
10. Obrazac koji je popravio simulator.
11. Projektni obrasci vam daju zajednički _____ sa drugim programerima.
13. Konstanta u razvoju aplikacija.
14. Pačji govor.
16. Programirajte u ovo umesto u implementaciju.

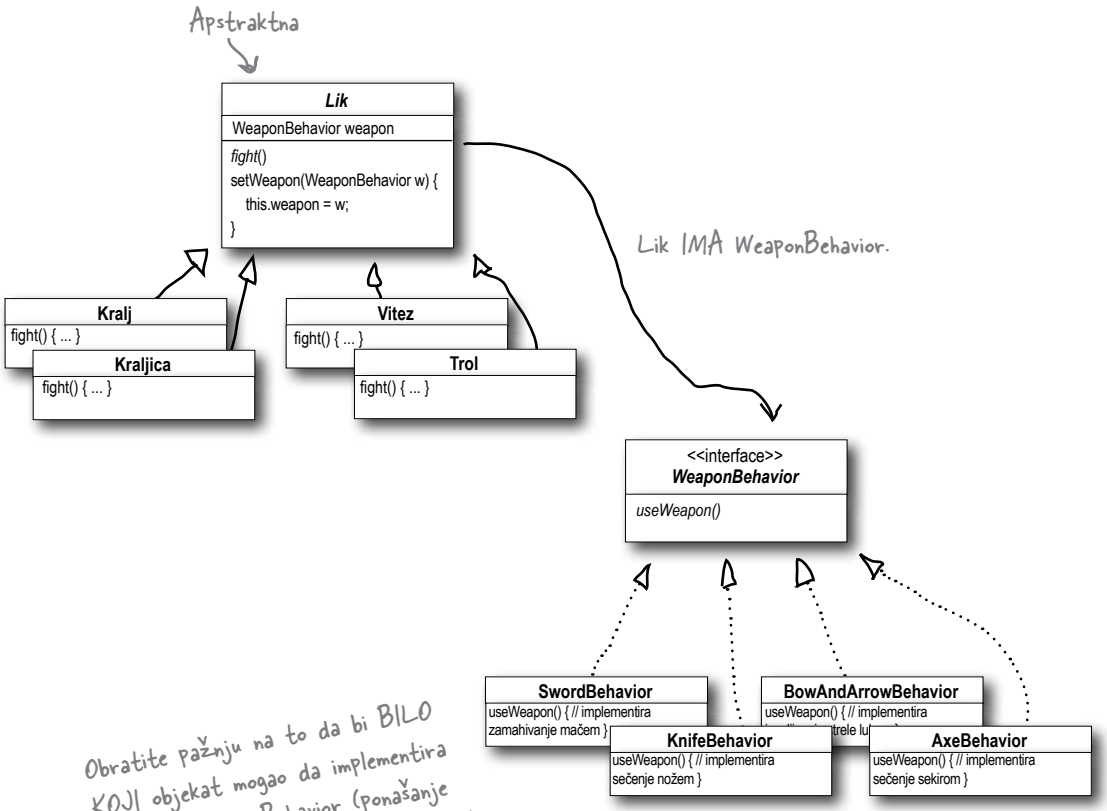
USPRAVNO

2. Dajte prednost ovome nad nasleđivanjem.
3. Desert koji mrzovoljni kuvar naziva brod.
4. Strategije mogu biti _____.
5. Većina obrazaca proističe iz OO _____.
6. Obrasci ulaze u vaš _____.
8. Obrazac kojim je Rik bio oduševljen.
9. U učenju koristite tuđe _____.
12. Ostrvo na kom je prikazan demo s patkama.
15. Java IO, Networking, Sound.

Rešenje projektne zagonetke

Lik je apstraktna klasa za sve ostale likove (Kralj, Kraljica, Vitez i Trol), dok je WeaponBehavior interfejs koji implementiraju sva ponašanja oružja. Tako su svi konkretni likovi i oružja konkretne klase.

Da bi promenio oružje, svaki lik poziva metodu setWeapon() koja je definisana u natklasi Lik. Tokom borbe se metoda useWeapon() poziva za tekuće oružje zadato za dati lik da bi se nanela velika telesna povreda drugom liku.



Naoštrite olovku Rešenje

Šta od narednog čini pravljenje potklasa neodgovarajućim za obezbeđivanje ponašanja konkretnih pataka? (Odaberite sve što važi.) Evo našeg rešenja.

- ☑ A. Kod se duplicira u potklasama.
- ☑ B. Izmene ponašanja tokom izvršavanja su teške.
- ☐ C. Ne možemo zadati da patke igraju.
- ☑ D. Teško je saznati sva ponašanja pataka.
- ☐ E. Patke ne mogu istovremeno da lete i kvaču.
- ☑ F. Izmene mogu nenamerno da utiču na druge patke.

Naoštrite olovku Rešenje

Navedite neke faktore koji su pokrenuli promene u vašim aplikacijama? Vaša lista bi mogla biti sasvim drugačija, ali evo nekoliko naših. Izgledaju li poznato? Evo našeg rešenja.

Moji klijenti ili korisnici odluče da žele nešto drugo, ili žele novu funkcionalnost.

Moja kompanija je rešila da pređe na drugog dobavljača baze podataka i na drugog dobaljvača podataka koji koristi različit format podataka. Uf!

Pa, tehnologija se menja i moramo da ažuriramo naš kod da bi koristio protokole.

Naučili smo mnogo gradeći naš sistem i voleli bismo da se vratimo i uradimo stvari malo bolje.

Ukrštenica projektnih obrazaca
Rešenje

			1 P	A	T	2 K	A				3 B		4 V	
						O				5 P	A		I	
						M				R	N		Š	
		6 M		7 K	A	P	S	U	L	I	R	A	NJ	E
8 P		O				O				N		N		K
O		Z				Z		9 I		C		A		R
10 S	T	R	A	T	E	G	I	J	S	K	I		S		A
M			K				C		K		P		P		T
A							I		U		A		L		N
T							J		S				I		E
11 R	E	Č	N	I	K		A		T				T		
A									V						
Č				12 M			13 P	R	O	M	E	N	A		
14 K	V	15 A	K	A	NJ	E									
I		P		U											
		I		16 I	N	T	E	R	F	E	J	S			