

# 3

## Upotreba predložaka u Javi

U poglavlju 2, govoreći o razvojnim metodologijama, naučili smo nešto o načinu razmišljanja Java programera. Ovo će poglavlje produbiti naše znanje jer predstavlja predloške (engl. *design patterns*) – tehnologiju čije će nas poznavanje učiniti učinkovitijim Java programerima.

Ovo nije knjiga o predlošcima; ovo je poglavlje uključeno u nju radi toga što su predlošci ključni za razumijevanje dizajna programskih sučelja u aplikacijama, alata i drugih aplikacija. Razlog leži u činjenici da je velika većina tehnologija ugrađena u predloške dizajna.

Kada bih trebao odabrati aspekt razvoja softvera u koji sam apsolutno zaljubljen, odabrao bih dizajn softvera. Kvalitetno dizajniranje je izazovna aktivnost i zahtijeva primjenu niza kreativnih umijeća u rješavanju problema. Iskustvo u stvaranju programskog rješenja također može biti izvorom osjećaja zadovoljstva. Nalazite li se još uvijek u fazi upoznavanja programskog jezika Java, možda će vam se dizajn softvera učiniti malo prezahtjevnim područjem. Softverski dizajn sličan je praznom slikarskom platnu okruženom velikim brojem boja između kojih je potrebno odabrati one prave. Odlučivanje u procesu dizajna nije laka zadaća jer, ako nemate dovoljno iskustva teško je znati kakav će učinak odluka u fazi dizajna imati na aplikaciju u kasnijim fazama njezinog razvoja.

Stjecanje znanja o predlošcima dizajna najbolji je način podizanja svojih programerskih sposobnosti na višu razinu. Tehnologije se mijenjaju vrlo brzim tempom. Kako bismo malo olakšali razumijevanje odnosa tehnologije i dizajna, recimo da je upoznavanje nove tehnologije poput čitanja dobre knjige. S druge pak strane, upoznavanje dizajna je kao kada učite čitati.

Ovo će poglavlje biti usmjereno na stjecanje znanja o važnosti predložaka dizajna i isticanja uobičajenih predložaka. Ako se već niste upoznali s predlošcima, ovo će vam poglavlje pomoći u tome.

*Mnogo je knjiga o predlošcima dizajna, no držim kako su sljedeće tri među najboljima: Improving the design of Existing Code autora Martina Fowlera; Design Patterns: Elements of Reusable Objected-Oriented Software autora Ericha Gamme, Richarda Helma, Ralpa Johnsona, i Johna Vlissidesa te Applying UML and Patterns: An Introduction to Objected-Oriented Analysis and Design and the Unified Process autora Craiga Larmana.*

U ovom ćete se poglavlju upoznati s definicijom predložaka; naučit ćete radi čega su isti toliko važni, naučit ćete trikove koji će vam pomoći u njihovu razumijevanju te ćete se upoznati s najvažnijim predlošcima u Javi. Poglavlje je podijeljeno u tri glavne cjeline. Prva cjelina objasnit će razloge radi kojih je važno poznavati predloške dizajna i predstaviti će neke primjere njihova korištenja u dizajnu softvera. Druga će cjelina govoriti o izgradnji predložaka iz dizajnerskih načela i provest će vas kroz niz vježbi koje pokazuju način izgradnje predložaka iz dizajnerskih načela. Naposljetku, posljednja cjelina donosi pregled najvažnijih predložaka dizajna i primjere njihove primjene.

## Zbog čega su važni predlošci dizajna

Jedan od najdražih citata mogeg oca bio je sljedeći: „Iskustvo je dobar učitelj, ali samo će budala učiti isključivo iz vlastitog iskustva“. I u razvoju softvera vlastito iskustvo je dobar učitelj, ali će vam i ono što naučite iz iskustva drugih programera pomoći svoja umijeća podići na višu razinu.

Predložak dizajna zapravo je naučena lekcija iz dizajna. Predložak dizajna je prokušano rješenje nekog problema, koje je moguće iskoristiti na razini dizajna. Svrha predložaka dizajna jest povezivanje problema s njihovim rješenjima te primjena rješenja na slične probleme. Korištenje predložaka programskoga koda je poželjno, ali je korištenje predložaka dizajna puno fleksibilnije. Predlošci dizajna mijenjaju se sa svakom aplikacijom na kojoj radite, no uvijek će postojati sličnosti. Sposobnost prepoznavanja ovih sličnosti uparena s poznavanjem predložaka dizajna pomoći će vam odluke o dizajnu softvera donositi s lakoćom.

Predlošci su jedan od najvažnijih resursa kojima ćete raspolagati u dizajnu objektno orijentiranoga softvera. Svakako će vam pomoći u potpunosti svladati programski jezik Java, biti produktivniji te da razvijati učinkovita Java programska rješenja.

## Ključni faktori razumijevanja Java programskoga jezika

Predlošci dizajna pomoći će vam u razumijevanju samog Java programskoga jezika. U usporedbi s drugim programskim jezicima, učenje Jave ima vrlo strmu krivulju. Nije riječ o tome da je Javu teže naučiti od ostalih programskih jezika: upravo je suprotno – Java ima vrlo čistu sintaksu i strukturu koja je slična drugim objektno orijentiranim jezicima.

No, Java postaje teška u trenutku kad se susretnete s ogromnim brojem API sučelja (Application Programming Interface) dostupnih u njoj. Ipak, njihov velik broj zapravo je dobra stvar: svako od API sučelja trebalo bi promatrati kao alat u punoj kutiji alata za rješavanje problema.

Iskorištavanje postojećeg softvera u cijelosti je temeljni predložak razmišljanja profesionalnog programera u Javi jer, ovakva će vam praksa pomoći u uštedi vremena i podizanju razine produktivnosti. Kolekcija API sučelja dostupna u verziji 1.5 JDK, kao i bezbroj open source projekata, zapravo predstavlja ono što nećete morati razvijati sami.

Ova knjiga predstavlja više aplikacijskih programskih sučelja, poput Collections, Java2D, JMX, XML, EJB, JMS, JDBC, RMI i Web Service. Popis je prilično dugačak, no njime smo tek zagrebali po površini stvarnoga bogatstva API sučelja. Naravno, ne možete naučiti sva ova sučelja, ali za takvo što i nema razloga. I upravo stoga su predlošci dizajna važni u svladavanju Jave.

Predlošci dizajna pomoći će vam u brzom svladavanju novog API sučelja. Razumijevanje predložaka koji se koriste u pojedinom API sučelju pomoći će vam u brzom razumijevanju, vrednovanju i potencijalnoj integraciji ovoga koda u vlastito programsko rješenje. Mnogo

je lakše koristiti i učiti iz postojećih API sučelja, nego uvijek iznova „izmišljati toplu vodu“ i počinjati ispočetka.

Ovo je posebno točno u radu s J2EE kosturom (Java 2 Platform Enterprise Edition). Doznate li radeći na nekom projektu kako je donesena odluka o ignoriranju mogućnosti distribuiranoga transakcijskog obrade J2EE aplikacijskog poslužitelja, a u korist nekoga posebno razvijenog rješenja, pobjegnite od tog projekta glavom bez obzira. Kao Java programer morate naučiti što više možete, ali primijeniti morate samo ono što je potrebno.

J2EE je standardno rješenje. Često se pogrešno misli da je J2EE kostur proizvod – J2EE nije proizvod. Riječ je samo o specifikaciji koju je tvrtka Sun izdala u obliku skupine dokumenata koji objašnjavaju kako bi se morali ponašati mrežni i EJB (Enterprise JavaBeans) spremnici. Prodavači softvera implementiraju ove specifikacije i prodaju ih kao dio standardnoga paketa. Ovo je važno reći jer se ljudi u Sunu jako dobro razumiju u predloške dizajna. Sva API sučelja temelje se na predlošcima, što je dobra vijest za vas kao programera te izvrstan razlog za učenje i razumijevanje predložaka dizajna. Ako svladate predloške dizajna, lako ćete shvatiti i iskoristiti sve što Sun proizvede.

## **Ključni faktori za razumijevanje alata koji se koriste u programiranju u Javi**

Osim bogatstva API sučelja dostupnih Java programerima, ovdje je i velik broj razvojnih alata namijenjenih poboljšanju procesa razvoja. Neki od ovih alata jesu ANT, JUnit, i XDoclet. Ovi alati donose proširene točke integracije, kao i dobre primjere važnosti predložaka dizajna.

### **ANT**

ANT je alat za gradnju aplikacije temeljen na XML-u, koji se može koristiti na više načina. Jedan od njih je automatiziranje gradnje softverskih cjelina tijekom razvoja softvera. Također, ovaj alat može se koristiti umjesto većine alata za skriptiranje, s time da nije ovisan o operativnom sustavu. Ovaj se alat sastoji od kombinacije nekoliko predložaka dizajna.

### **JUnit**

JUnit je kostur za testiranje cjelina softvera. Korištenje automatiziranih testova za programske cjeline izvrstan je način provjere izmjena u kodu, kao i sprječavanja pojave novih pogrešaka u softveru. Za korištenje JUnit alata potrebno je proširiti kostur. Razumijevanje predložaka dizajna s pomoću kojih je izgrađen JUnit pomoći će vam u iskorištavanju svih prednosti automatiziranog testiranja.

### **XDoclet**

XDoclet je kostur za generiranje programskoga koda. Omogućava umetanje metapodataka u komentare koda. Metapodaci se koriste za generiranje pomoćnoga koda, kao i za generiranje datoteka XML opisa. XDoclet olakšava sinkronizaciju izvedenih softverskih artefakata tijekom razvoja komponenti poput EJB-a, servleta i trajnih podatkovnih objekata, primjerice hibernate i JDO.

Postoje i brojni drugi alati dostupni Java programeru. Razumijevanje predložaka dizajna na kojima su ovi alati razvijeni uklonit će tajnovitost njihova rada, a njihovo poznavanje pomoći će vam u korištenju istih alata u razvoju boljega softvera.

## Ključni faktori razvoja učinkovitih Java rješenja

Predlošci će vam pomoći u razvoju učinkovitih softverskih rješenja korištenjem Jave. Oni pomažu u prenošenju informacija o konceptima dizajna, kao o u pridobivanju korisnog znanja o temeljnim načelima dizajna.

### Razvoj zajedničkog rječnika dizajna

U nazivu predložaka leži velika vrijednost; nazive predložaka programski inženjeri koriste u međusobnoj komunikaciji. Predlošci u ovoj knjizi preuzeti su iz široko prihvaćenog izvora – GoF skupine predložaka.

Primjerice, recimo da postoji potreba za međusobnom komunikacijom dvaju dijelova sustava, iako imaju različita sučelja. U ovakvom slučaju, koristit ćete predložak Adapter. U situaciji gdje je za rješenje nekog problema potrebno više algoritama, koristiti ćete predložak Strategy. Ovo poglavlje pruža detaljan uvid u ova dva predložka, ali i u nekoliko drugih. Ovdje smo ih spomenuli samo kako bismo naglasili vrijednost poznavanja predložaka.

### Poznavanja temeljnih postavki dizajna

Ovaj razlog za učenje predložaka meni osobno je vrlo drag. Na početku svoje programerske karijere, kad sam se tek upoznao s objektno orijentiranim programiranjem, nisam shvaćao važnost objektno orijentiranih koncepata i činilo mi se da je riječ samo o više posla koji će rezultirati slabijim rezultatima. Naklonost prema objektno orijentiranim konceptima počeo sam stjecati tek kad sam se susreo s predlošcima dizajna.

Predlošci će vam pomoći da do kraja shvatite temeljna načela dizajna. Razumijevanje temelja softverskog dizajna ključan je korak na putu da postanete samouvjeren softverski programer. Predlošci donose konkretne primjere primjene različitih dizajnerskih načela. U osnovi, dizajn se sastoji od donošenja odluka. Znanje o tome koje odluke vode razvoju dobrog softvera, a koje vode prema budućim problemima, čini cijelu razliku u razvoju djelotvornih programskih rješenja.

Dizajnerske odluke usmjerene su na određivanje dijelova softverskog sustava i prepoznavanje njihova međudjelovanja, a s ciljem postizanja ciljeva koje ste postavili pred sustav. Dobar dizajn rezultat je lekcija naučenih na težak način, prolaženjem kroz „noćne more“ loših dizajnerskih odluka.

*Apstrakcija*, *polimorfizam* i *nasljeđivanje* tri su temeljna načela objektno orijentiranog dizajna. Apstrakcija (engl. *abstraction*) je postupak modeliranja relevantnih aspekata stvarnosti. Polimorfizam (engl. *polymorphism*) je zamjena za tipove podataka, kako bi se jednoj klasi omogućilo da zauzme mjesto druge. Nasljeđivanje (engl. *inheritance*) je postupak stvaranja veza specijalizacije i generalizacije između klasa. Neki kriteriji dizajna koje je potrebno uzeti u obzir za vrijeme razvoja Java softverskih rješenja uključuju:

- ◆ **Zaštićene varijacije.** Ovo znači da u aplikacijama morate izbjegavati proizvoljnost. Osjećate li da bi se neka komponenta aplikacije mogla promijeniti, odvojite ju od drugih korištenjem sučelja. Sučelja će vam pomoći promijeniti klasu implementacije, bez utjecaja na podređene aplikacije.
- ◆ **Nisko uparivanje.** Svrha ovog koncepta dizajna jest osiguravanje da promjene izvedene u jednoj sekciji programskoga koda ne utječu na druge, nepovezane sekcije. Primjerice, iziskuje li promjena u korisničkom sučelju i promjenu baze podataka? Ako iziskuje, aplikacija bi mogla postati krhka u slučaju male promjene u softverskom sustavu.
- ◆ **Visoka kohezija.** To je postupak čvrstoga međusobnog povezivanja srodnih komponenti.

Važan je za razumijevanje predložaka dizajna jer svaki predložak zapravo primjena jednoga ili više dizajnerskih načela. Kad shvatite apstrakciju, polimorfizam i nasljeđivanje, lakše ćete shvatiti i kako rad s predlošcima može pojednostaviti dizajn softvera.

Ciljevi kod dizajna softvera su važni, ali postoji veliki jaz između ciljeva i implementiranog softvera. Predlošci premošćuju ovaj jaz i pomažu ostvarivanju ciljeva. Sljedeći odjeljak govori o temeljnim aktivnostima kada započnete raditi s predlošcima.

## Razvoj predložaka na temelju načela dizajna

U srži svakog predloška nalazi se zbirka dizajnerskih načela. Ovaj odjeljak donosi jednostavan i nekonvencionalan pristup izgradnji predložaka od temelja prema vrhu. Pristup se sastoji od započinjanja s jednostavnom dizajnom i postupnim unošenjem promjena tako da dizajn ostane fleksibilniji. Svaka promjena u dizajnu postaje korak u izgradnji kompleksnijih predložaka dizajna. Vježba u ovom poglavlju pomoći će vam u shvaćanju kako principi dizajna čine softver fleksibilnijim te će vam omogućiti postupno razumijevanje mehanizama iza predložaka dizajna.

Odjeljak započinje s dizajniranjem jedne klase. Ovom se dizajnu zatim dodaje veza, a nakon nje sučelje. Ova dva koraka dizajnu od jedne klase omogućavaju fleksibilnost. Razumijevanje ove fleksibilnosti ima važne posljedice za razumijevanje predložaka dizajna. Posljednji odjeljak donosi primjere spajanja koncepata veze i nasljeđivanja, što je čest slučaj kod mnogih predložaka dizajna.

### Dizajniranje jedne klase

Jedna klasa ne čini predložak dizajna, ali ipak je i ovdje riječ o dizajnu. (A nema ničega lošeg u jednostavnosti.) Dio procesa dizajna jest dodjeljivanje odgovornosti objektu, kao što prikazuje slika 3-1.

Teacher
-name
+getName() +getSSN() +teachClass() +takeAttendance() +proctorTest() +gradePaper() +reportGrades()

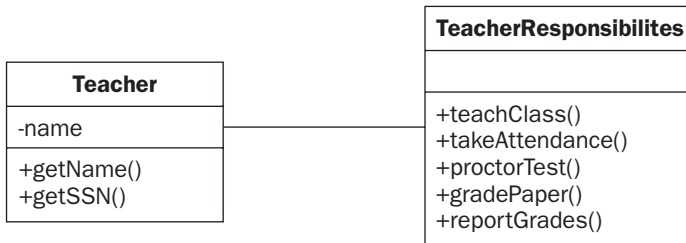
Slika 3-1

Često se dogodi da klasa postane pretrpana metodama koje uopće nisu povezane s apstrakcijom koju klasa predstavlja. To u kasnijim fazama može dovesti do problema s određenim klasama i odmak je od načela visoke kohezije. U ovom primjeru, klasa Teacher sastoji se od nekoliko metoda povezanih s odgovornostima učitelja. Rješenje se sastoji u postupku *pomicanja udesno* (engl. *push to the right*) ili delegiranja metoda kojima nije mjesto u ovoj apstrakciji. Treba pritom reći kako je fraza „kojima nije mjesto“ subjektivna – svaka odluka u dizajnu može biti pogrešna. No, sve dok svoje odluke možete opravdati

jasnim principima objektno orjentiranog dizajna, nemate se za što brinuti – odluke uvijek možete promijeniti u kasnijim fazama, kad problem postane jasniji.

## Stvaranje asocijacija između klasa

Sve su odgovornosti učitelja upućene klasi `TeacherResponsibilities`. I ovdje je potrebno vizualizirati pomicanje metoda udesno. Slika 3-2 prikazuje način delegiranja odgovornosti kroz asocijaciju između klasa.



Slika 3-2

Kako bi klasa `TeacherResponsibilities` radila za klasu `Teacher`, potrebno je stvoriti vezu između njih. Objekt `Teacher` sadrži referencu na `TeacherResponsibilities`. U osnovi, postoje tri načina da se to postigne:

1. Objekt `TeacherResponsibilities` proslijeđuje se objektu `Teacher` kao parametar.

```
Teacher teacher = new Teacher("Heather");
TeacherResponsibilities responsibilities= new TeacherResponsibilities ();
teacher.setResponsibilities ( responsibilities);
```

2. Objekt `Teacher` stvara objekt `TeacherResponsibilities`.

```
public class Teacher {

    private TeacherResponsibilities responsibilities = new TeacherResponsibilites();

}
```

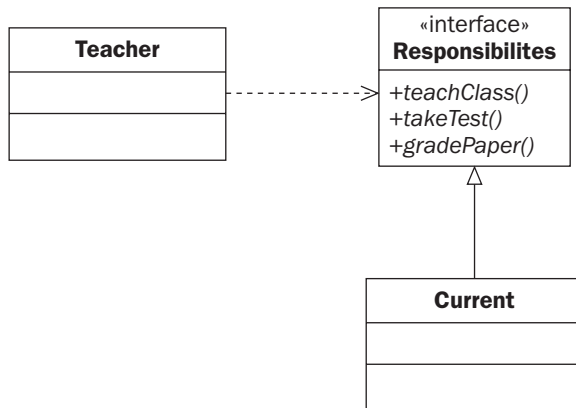
3. Objekt `TeacherResponsibilities` proslijeđen je natrag iz poziva metode.

```
public class Teacher {
    public Teacher() {
        Administration admin = new Administration();
        responsibilities = admin.getResponsibilites();
    }
}
```

Ove tri metode određuju međusobnu vidljivost objekata tijekom stvaranja asocijacije između njih. Postupak dizajna ovime je možda završen, ali potrebno je posvetiti pažnju još jednom principu dizajna: *labavom uparivanju* (engl. *loose coupling*). Pri zadavanju asocijacije stvorena je snažna ovisnost između klasa `Teacher` i `TeacherResponsibilities`. Veza je ograničena na tipove `Teacher` i `TeacherResponsibilities`. Ovo samo po sebi nije loše, no možda će postojati potreba da se odgovornosti promijene tijekom vremena. *Kako olabaviti vezu i riješiti ovu neodređenost?* Odgovor je u pomicanju sučelja prema gore (engl. *push up*).

## Stvaranje sučelja

Sučelje (engl. *interface*) je softverska veza između klasa. Korištenjem sučelja, trenutačnoj klasi je omogućena implementacija. Ako se u budućnosti implementacija promijeni, trenutačnu klasu možete zamijeniti novom klasom. Budući da klasa `Teacher` ovisi samo o sučelju `Responsibilities`, neće ju biti potrebno mijenjati. UML dijagram za ovaj dizajn prikazan je na slici 3-3.



Slika 3-3

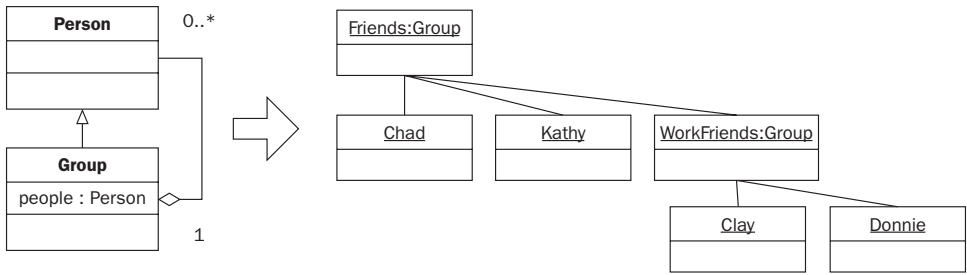
*Ovdje je potrebno reći kako svaki artefakt koji dodajete dizajnu predstavlja dodatnu stvar o kojoj će biti potrebno voditi brigu. Sučelja su sjajna za uspostavljanje odnosa između komponenti s ciljem izoliranja proizvoljnosti, ali nije ih potrebno primjenjivati baš posvuda.*

U sljedećem odjeljku bavit ćemo se kombiniranjem delegiranja i nasljeđivanja, koncepta koje smo predstavili u prethodnim odjeljcima, a s ciljem stvaranja snažnih objektnih struktura.

## Stvaranje petlje nasljeđivanja

Povezivanje dviju klasa asocijacijom i nasljeđivanjem, moguće je stvoriti stabla i grafove, što možete zamisliti kao *penjanje uz* hijerarhiju klasa. Relacija nasljeđivanja čvorove u objektnoj strukturi čini polimorfnim. U primjeru prikazanom na slici 3-4, grupom `WorkFriends` manipulira se korištenjem sučelja koje je deklarirala klasa `Person`. Sljedeći prikladan primjer je slično ponašanje datoteka i mapa u datotečnom sustavu – i jedni i drugi koriste zajedničke funkcionalnosti poput kopiranja, brisanja i sličnog.

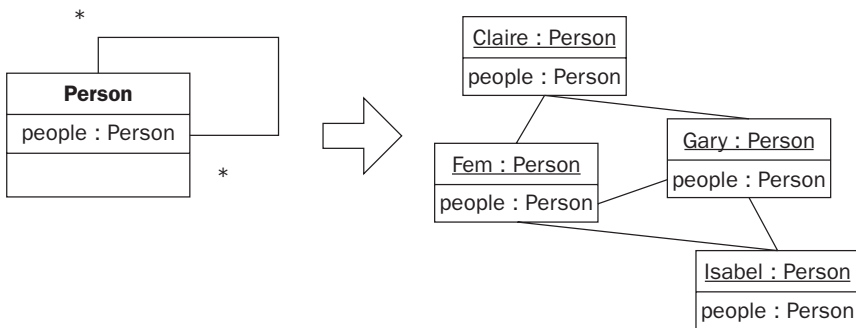
Slika 3-4 prikazuje pregled rezultirajućih klasa i objekata petlje nasljeđivanja (engl. *inheritance loop*). Ovo je uobičajena struktura koja se koristi u mnogim predlošcima dizajna, uključujući kompoziciju.



Slika 3-4

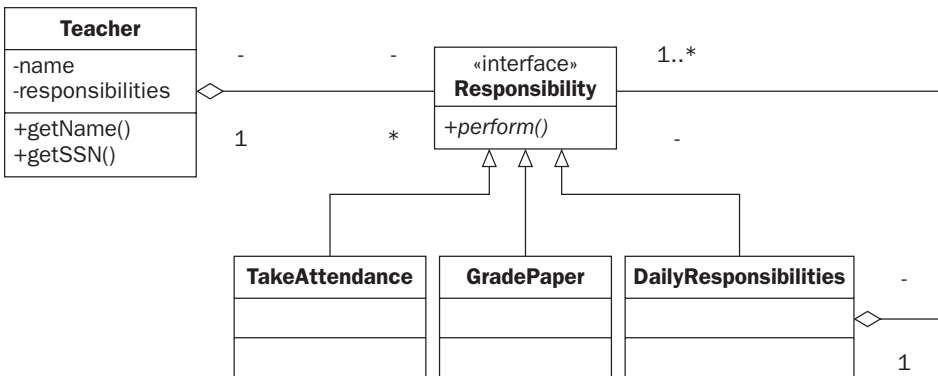
O petlji nasljeđivanja govorim kao o penjanju uz hijerarhiju, kao što je prikazano na slici 3-4. Kretanjem prema gore u hijerarhiji stvarate vezu koja se naziva *obrnuto zadržavanje* (engl. *reverse containment*). Zadržavanjem kolekcije više klase od strane niže klase, moguće je manipulirati različitim podtipovima, kao i kolekcijama koje dijele isto sučelje.

Slika 3-5 prikazuje neznatnu promjenu u odnosu na sliku 3-4. Promjenom osnovne asocijacije između viših i nižih tipova u *više prema više*, moguće je prikazati i grafove i stabla.



Slika 3-5

I naposljetku, slika 3-6 petlji nasljeđivanja dodaje veze nižeg reda prikazujući kompleksnu strukturu podataka s metodama koje je moguće pozvati s polimornim sučeljem.



Slika 3-6

Stvorili smo i zajedničko sučelje koje će omogućiti dodavanje novih odgovornosti, uz ograničeni utjecaj na aplikaciju.

Svrha ovog odjeljka bila je iznijeti „trikove“ koji će vam pomoći u shvaćanju predložaka. Stvaranjem asocijacija i korištenjem nasljeđivanja, iz istih smo načela izgradili neke kompleksne primjere dizajna. Naučili smo primjenjivati ove principe korištenjem jednostavnih akcija: pomicanje udesno, pomicanje prema gore, penjanje. Ovi će nam trikovi pomoći u razumijevanju važnih predložaka u sljedećem odjeljku.

## Važni Java predlošci

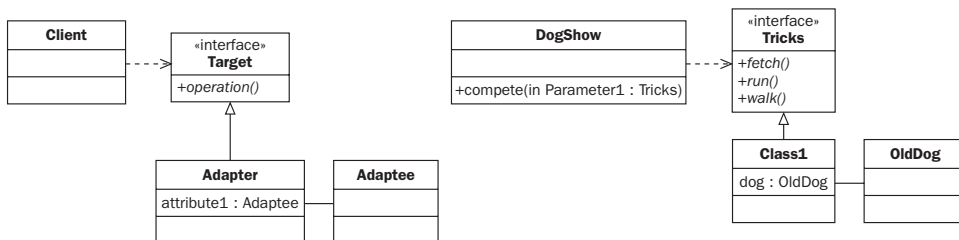
Ovaj odjeljak donosi primjere važnih predložaka. Stjecanjem znanja o svakome od njih, razvit ćete vlastiti rječnik predložaka i dodati ga u kutiju s alatom za razvoj softvera. Svaki od opisanih predložaka uključuje opis problema koje predložak rješava, temeljna načela dizajna koja su uključeni u ovaj predložak, klase od kojih se predložak sastoji te načine za njihovo povezivanje i zajednički rad.

Namjera ovog poglavlja nije opisivanje predložaka u tradicionalnom smislu, već iznošenje primjera koda i različitih problema koje je moguće riješiti svakim od iznesenih predložaka. Svi predlošci predstavljeni u ovom odjeljku jesu prilagođeni GoF predlošci - Adapter, Model-View-Controller, Command, Strategy i Composite.

U opisu svakog predloška potrebno je posebno istaknuti način na koji klase od kojih se predložak sastoji djeluju međusobno u rješavanju specifičnog problema. Svaki predložak bit će predstavljen s pomoću teksta i slike koja prikazuje predložak te s pomoću pokaznih klasa koje igraju odgovarajuću ulogu u predlošku.

## Adapter

Predložak Adapter omogućava komunikaciju između komponenti koje imaju nekompatibilna sučelja. Dobar je primjer upotrebe koncepta objektno orijentiranog dizajna - i to iz razloga što je vrlo jasan. U isto vrijeme, ovaj predložak na sjajan način prikazuje tri važna načela dizajna: upućivanja, nasljeđivanja i apstrakcije. Slika 3-7 prikazuje strukturu klasa u predlošku Adapter, kao i pokazne klase.



Slika 3-7

## Predložak Adapter je rezultat međudjelovanja četiriju klasa

Četiri klase od kojih se sastoji predložak Adapter jesu Target, Client, Adaptee i Adapter. U ovom primjeru prikazana je učinkovitost predloška Adapter u rješavanju problema nekompatibilnih sučelja: klasa Adaptee ne implementira sučelje Target. Rješenje se sastoji

u stvaranju međuklase Adapter, koja će umjesto klase Adaptee implementirati sučelje Target. Korištenjem polimorfizma, klijent može koristiti ili sučelje Target ili klasu Adapter, i ne mora se brinuti o kojoj je točno komponenti riječ.

## Sučelje Target

Počnite sa sučeljem Target. Ono opisuje kako će se objekt morati ponašati. U nekim je slučajevima moguće implementirati sučelje Target u objekt, dok u drugim slučajevima ovo nije moguće. Naprimjer, sučelje bi moglo sadržavati nekoliko metoda, ali vama je potrebno prilagođeno ponašanje samo jedne od njih. Paket `java.awt` sadrži Window adapter namijenjen isključivo rješavanju ovog problema. U drugomu bi primjeru objekt naziva Adaptee koji želite prilagoditi mogao biti zapravo kod koji nije moguće mijenjati:

```
package wrox.pattern.adapter;

public interface Tricks {

    public void walk();
    public void run();
    public void fetch();
}
```

## Client

Slijedi kod klase Client, koja također koristi ovo sučelje. Ovo je jednostavan prikaz metoda u sučelju. Metoda `compete()` podređena je sučelju Tricks. Mogli biste je modificirati kako bi podržavala sučelje Adaptee ali bi to povećalo kompleksnost koda Client. Umjesto toga, bolje je kod Client ne modificirati, a klasu Adaptee modificirati da surađuje sa sučeljem Tricks:

```
public class DogShow {

    public void compete( Tricks target){
        target.run( );
        target.walk( );
        target.fetch( );
    }
}
```

## Adaptee

Adaptee sadrži kod koji ćete morati koristiti, ali na način da izlaže sučelje Target bez njezove direktne implementacije:

```
package wrox.pattern.adapter;

public class OldDog {
    String name;

    public OldDog(String name) {
        this.name= name;
    }
    public void walk() {
        System.out.println("walking..");
    }
    public void sleep() {
```

```

        System.out.println("sleeping..");
    }
}

```

## Adapter

Kao što je vidljivo, klasa `OldDog` ne implementira ni jednu metodu sučelja `Tricks`. Kodom koji slijedi ovu ćemo klasu proslijediti klasi `Adapter`, koja implementira sučelje `Target`:

```

package wrox.pattern.adapter;

public class OldDogTricksAdapter implements Tricks {
    private OldDog adaptee;

    public OldDogTricksAdapter(OldDog adaptee) {
        this.adaptee= adaptee;
    }
    public void walk() {
        System.out.println("this dog can walk.");
        adaptee.walk();
    }
    public void run() {
        System.out.println("this dog doesn't run.");
        adaptee.sleep();
    }
    public void fetch() {
        System.out.println("this dog doesn't fetch.");
        adaptee.sleep();
    }
}

```

`Adapter` se može koristiti svugdje gdje se koristi sučelje `Tricks`. Prosljeđivanje `OldDogTricksAdapter` u klasu `DogShow` omogućit će vam korištenje koda napisanoga za sučelje `Tricks`, kao i korištenje nemodificirane klase `OldDog`.

Slijedeći odlomak koda prikazuje kako uspostaviti asocijacije i izvesti primjer:

```

package wrox.pattern.adapter;

public class DogShow {
    // Metode su izostavljene.

    public static void main(String[] args) {

        OldDog adaptee = new OldDog("cogswell");
        OldDogTricksAdapter adapter = new OldDogTricksAdapter( adaptee );
        DogShow client = new DogShow( );
        client.compete( adapter );

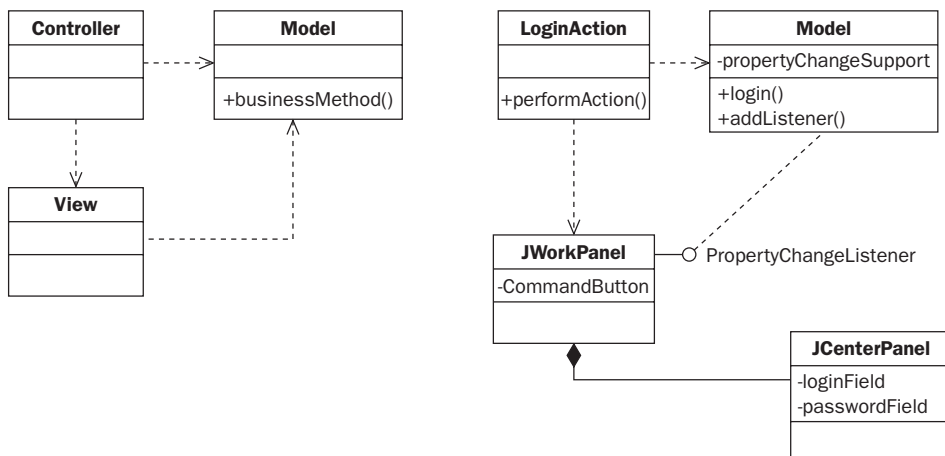
    }
}

```

## Predložak Model-View-Controller

Svrha predložka `Model-View-Controller` jest odvajanje logike korisničkog sučelja od sloja poslovne logike. Na ovaj je način moguće više puta upotrijebiti sloj poslovne logike i spriječiti da promjene u sučelju utječu na poslovnu logiku. Predložak `Model-View-Controller`,

poznat i kao Model-2, često se koristi u Web razvoju. Zbog toga je poglavlje 8 u cijelosti posvećeno ovoj temi. O razvoju Swing klijenata možete naučiti i u četvrtom poglavlju. Slika 3-8 prikazuje strukturu klasa predložka Model-View-Controller, zajedno s klasama koje u ovom primjeru implementiraju taj predložak:

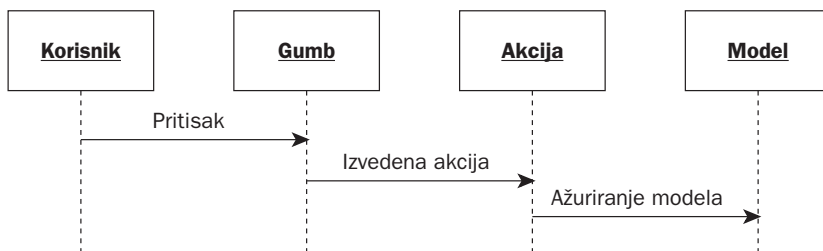


Slika 3-8

Sljedeći primjer upotrebe predložaka bit će jednostavna Swing aplikacija (Swing biblioteka: grafička sastavnica osnovnih Java klasa – Java Foundation Classes, JFC). Aplikacija će uključivati osnovnu funkcionalnost za prijavljivanje (login), no važnije od ove funkcionalnosti jest odvajanje načela dizajna koji omogućavaju labavo uparivanje modela (podaci), kontrolera (akcije) i pregleda (Swing obrazac). Model-View-Controller zapravo je i više od jednostavnog predložka jer predstavlja podjelu odgovornosti, koja je česta u dizajnu aplikacija. Aplikacija koja podržava načela predložka Model-View-Controller mora odgovoriti na ova tri pitanja: Na koji način aplikacija mijenja model? Na koji se način promjene modela reflektiraju na pregled? Na koji se način uspostavljaju asocijacije između modela, pregleda i klasa-kontrolera? Sljedeći odjeljak prikazuje na koji se način implementiraju scenariji ovog primjera uz korištenje Swing aplikacije.

## Scenarij 1: Izmjena modela

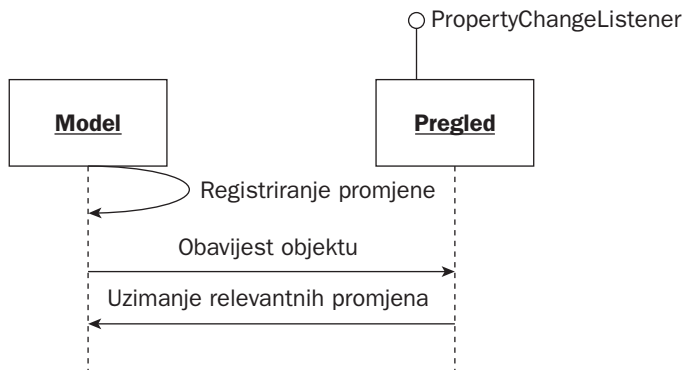
Izmjene modela izvode se izvana prema unutra. U ovom primjeru se kao sučelje koristi Java Swing. Korisnik pritisće gumb, što izaziva događaj koji prima kontrolna akcija. Akcija zatim mijenja model (slika 3-9).



Slika 3-9

## Scenarij 2: Obnavljanje nakon izmjene modela

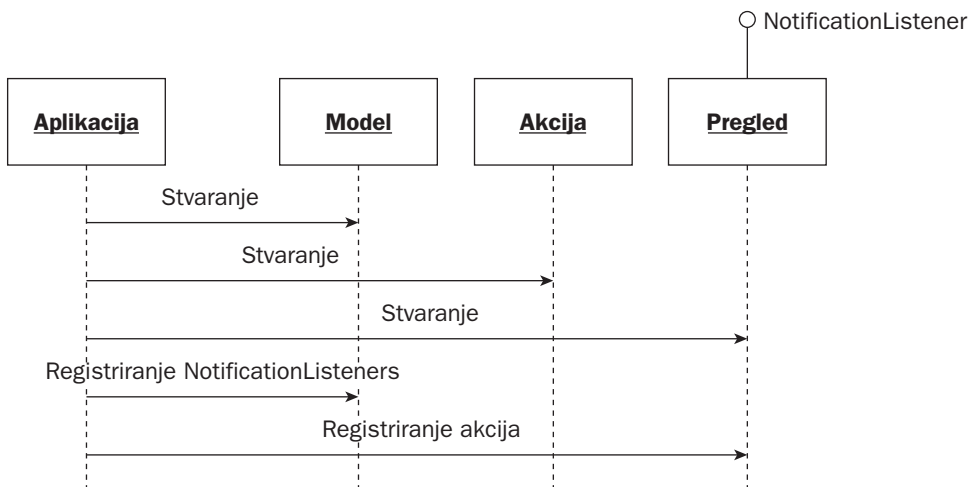
U sljedećem scenariju pretpostavka je da je model bio izmijenjen tijekom neke akcije. Pregledi (engl. *views*) će možda trebati imati tu informaciju, ali izravno pozivanje pregleda od strane modela će uzrokovati kršenje načela separacije, budući da bi u tom slučaju model trebao imati informacije o pregledu. Za rješavanje ovog problema, Java nudi predložak *Observer*, koji dopušta da izmjene na modelu budu proslijeđene komponentama pregleda. Svi pregledi koji ovise o modelu morat će se registrirati kao objekti listeneri *ChangeListener*. Nakon registracije, bit će obavještavani o izmjenama na modelu. Obavijest se sastoji od naputka pregledu da informacije koje su mu potrebne izvuče izravno iz modela (slika 3-10).



Slika 3-10

## Scenarij 3: Inicijalizacija aplikacije

Treći scenarij prikazuje kako inicijalizirati objekte akcije, modela i pregleda te zatim uspostaviti veze između komponenti (slika 3-11).



Slika 3-11

Pregledi su registrirani u modelu, a akcije u pregledima. Registracijom upravlja klasa aplikacije.

Nakon što smo prikazali scenarije suradnje između modela, pregleda i kontrolnih komponenti, u sljedećem odjeljku predstaviti ćemo svaku od komponenata.

## Model

Model može biti jedan ili više Java objekata koji predstavljaju temeljne podatke u aplikaciji, a često ih se naziva domenskim modelima. U ovom primjeru, koristit ćemo jedan objekt u Javi i nazvati ga Model.

Funkcionalnost koju mora obavljati Model u ovom primjeru jest podrška funkciji prijavljivanja. U stvarnoj aplikaciji, Model bi uključivao izvore podataka poput relacijskih baza podataka ili usluga imenika:

```
package wrox.pattern.mvc;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Model {
```

Prva stvar koja nas zanima kod ovog modela jest varijabla članica `PropertyChangeSupport`. Ona je dio komponente Event Delegation Model (EDM), dostupne u Javi od verzije JDK 1.1. EDM je mehanizam *izdavač-pretplatnik* (engl. *publisher subscriber*) koji pregledima omogućava registraciju u modelu i primanje obavijesti o promjenama njegova stanja:

```
private PropertyChangeSupport changeSupport= new PropertyChangeSupport(this);
private boolean loginStatus;
private String login;
private String password;
public Model() {
    loginStatus= false;
}
public void setLogin(String login) {
    this.login= login;
}
public void getPassword(String password) {
    this.password= password;
}
public boolean getLoginStatus() {
    return loginStatus;
}
```

Ovdje možemo vidjeti kako metoda `setLoginStatus()` pokreće izmjenu atributa:

```
public void setLoginStatus(boolean status) {
    boolean old= this.loginStatus;
    this.loginStatus= status;
    changeSupport.firePropertyChange("model.loginStatus", old, status);
}

public void login(String login, String password) {
    if ( getLoginStatus() ) {
        setLoginStatus(false);
    } else {
        setLoginStatus(true);
    }
}
```

`addChangeListener()` jest metoda koja svim zainteresiranim pregledima omogućava registraciju u modelu i primanje obavijesti o događajima:

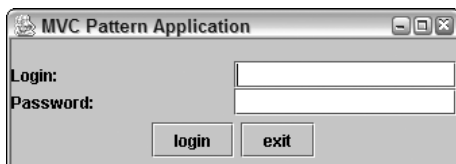
```
public void addChangeListener(PropertyChangeListener listener) {
    changeSupport.addChangeListener(listener);
}
}
```

Primijetite kako u samomu modelu ne postoje nikakve reference prema komponentama korisničkog sučelja, što znači da su omogućene promjene u pregledima koje neće imati nikakvog utjecaja na djelovanje modela. Također, moguće je stvoriti i sekundarno sučelje. Primjerice, mogli biste stvoriti API sučelje korištenjem Web usluga, koje će omogućiti automatizirano prijavljivanje na daljinu.

## Pregled

Komponenta pregleda aplikacije sastoji se od Swing sučelja. Slika 3-12 prikazuje što će korisnik vidjeti prilikom pokretanja aplikacije.

Ovo korisničko sučelje sastoji se od dvije `JPanel` komponente. Prva od njih jest klasa `CenterPanel`, koja sadrži polja za unos korisničkog imena i lozinke. Druga komponenta je klasa `WorkPanel` koja se sastoji od komandnih gumba za prijavljivanje i odjavljivanje te klasu `CenterPanel`.



Slika 3-12

`CenterPanel` je uobičajen obrazac za upis podataka. Važno je spomenuti da u ovoj klasi ne postoji programski kod za izvođenje operacije prijavljivanja – njezin zadatak je napraviti korisničko sučelje:

```
package wrox.pattern.mvc;
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class CenterPanel extends JPanel {

    private JTextField login= new JTextField(15);
    private JTextField password= new JTextField(15);

    public CenterPanel() {
        setLayout(new GridLayout(2, 2));
        add(new JLabel("Login:"));
        add(login);
        add(new JLabel("Password:"));
        add(password);
    }
}
```

```

    public String getLogin() {
        return login.getText();
    }
    public String getPassword() {
        return password.getText();
    }
}

```

Sljedeća komponenta korisničkog sučelja, `WorkPanel`, sadrži klasu `CenterPanel`. Primijetite kako ne postoje reference iz klase `CenterPanel` u `WorkPanel`. Slijedi kompozicije koja omogućava da klasa `CenterPanel` bude isključena u korist drugog obrasca ili pregledavana u drugačijem okviru:

```

package wrox.pattern.mvc;
import java.awt.BorderLayout;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;

```

Kao što je vidljivo iz deklaracije klase, `WorkPanel` je Swing komponenta. Također, implementira sučelje `PropertyChangeListener` što joj omogućava registraciju u modelu aplikacije i primanje obavijesti o izmjenama u modelu. Komponenta `WorkPanel` u modelu je registrirana kao objekt listener `PropertyChangeListener`, što omogućava nisko uparivanje između sučelja i logike domene, uključujući promjene pregleda sukladne promjenama modela:

```

public class WorkPanel extends JPanel implements PropertyChangeListener {
    private Model model;

    private JPanel center;
    private JPanel buttonPanel= new JPanel();
    private JLabel loginStatusLabel= new JLabel(" ");

    public WorkPanel(JPanel center, Model model) {
        this.center= center;
        this.model= model;
        init();
    }
    private void init() {
        setLayout(new BorderLayout());
        add(center, BorderLayout.CENTER);
        add(buttonPanel, BorderLayout.SOUTH);
        add(loginStatusLabel, BorderLayout.NORTH);
    }
}

```

Kod izmjene modela poziva se metoda `propertyChange()` za sve klase registrirane u modelu:

```

public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getPropertyName().equals("model.loginStatus")) {
        Boolean status= (Boolean)evt.getNewValue();
        if (status.booleanValue()) {
            loginStatusLabel.setText("Login was successful");
        }
    }
}

```

```

    } else {
        loginStatusLabel.setText("Login Failed");
    }
}
}
}

```

Metoda `addButton()` omogućava dvije stvari. Prvo, omogućit će vam konfiguriranje proizvoljnog broja gumba i drugo, sadrži klase akcija koje određuju što će se točno dogoditi nakon pritiska određenoga gumba. Akcija predstavlja konačni dio MVC predloška – kontroler. Kontroler ćemo predstaviti u sljedećem odjeljku.

```

public void addButton(String name, Action action) {
    JButton button= new JButton(name);
    button.addActionListener(action);
    buttonPanel.add(button);
}
}
}

```

## Kontroler

Namjena kontrolera jest da služi kao prolaz za izvođenje izmjena u modelu. U sljedećem primjeru, kontroler se sastoj od dviju `java.swing.Action` klase. Ove su klase akcija registrirane u jednoj ili više grafičkih komponenti, preko metode komponentata `addActionListener()`. U ovoj aplikaciji postoje dvije `Action` klase. Prva se pokušava prijaviti u model, a druga služi za izlazak iz aplikacije:

```

package wrox.pattern.mvc;

import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;

```

`LoginAction` klasa proširuje klasu `AbstractAction` i premošćuje metodu `actionPerformed()`. Metodu `actionPerformed()` poziva komponenta, u ovom slučaju gumb, nakon što se pritisne. Akcija nije ograničena na registraciju u samo jednoj komponenti korisničkog sučelja. Korist izdvajanja kontrolne logike u zasebnu klasu je u tome što je akciju moguće registrirati u izbornicima, prečacima s tipkovnice i alatnim vrpčama a to sprječava ponavljanje logike akcije za svaku komponentu korisničkoga sučelja:

```

public class LoginAction extends AbstractAction {

    private Model model;
    private CenterPanel panel;

```

Uobičajeno je da kontroler može vidjeti i model i njegove preglede. Ipak, model ne može izravno pozivati akcije, čime je osigurano očuvanje odvojenosti slojeva poslovne logike i sučelja:

```

public LoginAction(Model model, CenterPanel panel ) {
    this.model= model;
    this.panel = panel;
}

public void actionPerformed(ActionEvent e) {
    System.out.println("Login Action: "+ panel.getLogin() +" "+ panel.getPassword() );
    model.login( panel.getLogin(), panel.getPassword() );
}
}
}

```

Klasa `ExitAction` kontrolira ponašanje korisničkog sučelja. Prikazuje poruku kod pritiskanja gumba `Exit`, tražeći potvrdu prekida rada aplikacije:

```
package wrox.pattern.mvc;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
public class ExitAction extends AbstractAction {
```

Naposlijetku, pogledajmo klasu `Application`. Ona je odgovorna za inicijalizaciju i stvara asocijaciju kojom se uspostavlja MVC separacija logičkih načela dizajna:

```
package wrox.pattern.mvc;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

public class Application extends JFrame {
    private Model model;
```

Swing aplikacija stvara asocijaciju s klasom `Model`, što je prikazano u sljedećem kodu aplikacijskoga konstruktora:

```
public Application(Model model) {
    this.model= model;
```

Nakon ovoga, stvorit ćemo preglede za prikazivanje Swing sučelja:

```
CenterPanel center= new CenterPanel();
WorkPanel work= new WorkPanel(center, model);
```

Slijedi stvaranje akcijskih klasa koje predstavljaju kontroler i njihova registracija kod gumba:

```
work.addButton("login", new LoginAction(model, center));
work.addButton("exit", new ExitAction() );
model.addPropertyChangeListener(work);
setTitle("MVC Pattern Application");
```

Za prikazivanje aplikacije, koristit ćemo Swing održavanje sustava:

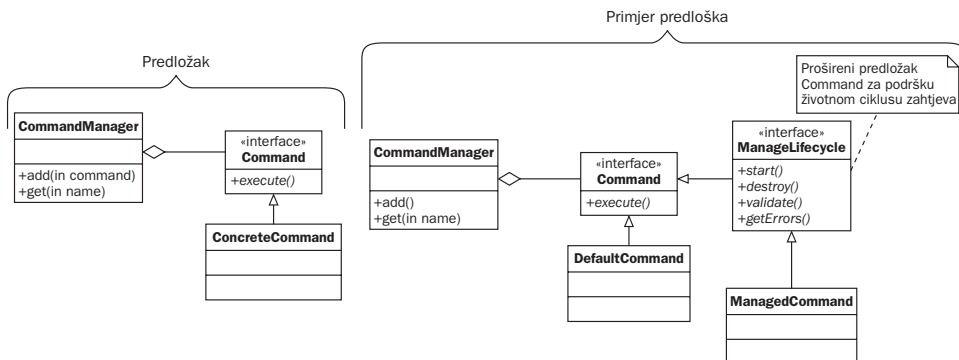
```
getContentPane().add(work);
pack();
show();
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
}
public static void main(String[] args) {
    Model model= new Model();
    Application application= new Application(model);
}
}
```

Predložak MVC jest kombinacija najboljih praksi u softverskom dizajnu jer ubrzava potrebno razdvajanje korisničkog sučelja i poslovnih slojeva aplikacije. Iznesenim smo primjerom predstavili veći broj predložaka dizajna – kompoziciju, akciju i događaj

objavljivanja u pretplate. Sljedeći predložak je Command, a omogućava dosljednu obradu zahtjeva korisnika.

## Predložak Command

Predložak Command predstavlja standardno sučelje za obradu korisničkih zahtjeva. Svaki zahtjev učahuren (engl. *encapsulated*) je u objekt s imenom Command. Slika 3-13 prikazuje klase uključene u predložak Command.



Slika 3-13

Tri klase u predlošku Command jesu Command, CommandManager i Invoker. Klasa Command zapravo je učahureno ponašanje (engl. *behavior*). Svako ponašanje, poput Save ili Delete jest aplikacija, koju je moguće modelirati u naredbu. Tako ponašanje aplikacije postaje zbirka komandnih objekata. Da bismo aplikaciji dodali ponašanje, sve što moramo napraviti jest implementirati dodatne komandne objekte. Sljedeća komponenta predloška Command je CommandManager. Ova klasa odgovorna je za pristup naredbama koje su dostupne aplikaciji. Posljednja komponenta jest Invoker, klasa koja je odgovorna za izvođenje komandnih klasa na dosljedan način. Sljedeći odjeljak će vam predstaviti anatomiju klase Command.

## Sučelje Command

Prvi dio predloška Command jest sučelje Command identificirano jednom metodom:

```

package wrox.pattern.command;

public interface Command {

    public void execute();
}

```

Ovdje se životni ciklus razlikuje od onoga kod pozivanja obične metode. Primjer je proslijeđivanje parametra objekta, kao kod ove metode:

```

public void getTotal(Sale) {
    // Izračunava vrijednost prodaje.
}

```

Kao naredbu, napisat ćemo:

```
public CalculateSale implements Command {
    private Sale sale;

    public void setSale( Sale sale ) {
        this.sale = sale;
    }
    public void execute( ) {
        // Izračunava vrijednost prodaje.
    }
}
```

Za demonstraciju interakcije između klasa u ovom predlošku koristit ćemo praznu naredbu:

```
package wrox.pattern.command;

public class DefaultCommand implements Command {

    public void execute() {
        System.out.println("executing the default command");
    }
}
```

U sljedećem odjeljku predstaviti ćemo klasu koja upravlja naredbom za aplikaciju.

## Klasa **CommandManager**

Klasa `CommandManager` obradit će sve zahtjeve. Korištenjem komponente `HashMap` sve će naredbe biti inicijalizirane prije obrade zahtjeva i zatim pozivane po nazivu. Naredbe se pohranjuju korištenjem metode `add()`, a pozivaju se metodom `getCommand()`:

```
package wrox.pattern.command;
import java.util.HashMap;
import java.util.Map;
public class CommandManager {
    private Map commands= new HashMap();

    public void add(String name, Command command) {
        commands.put(name, command);
    }
    public Command getCommand(String name) {
        return (Command)commands.get(name);
    }
}
```

## Invoker

Izvođenje predloška `Command` prikazat ćemo s pomoću jednog klijenta. Nakon pozivanja, konstruktor `Client` dodaje naredbu `DefaultCommand` komponenti za upravljanje:

```
package wrox.pattern.command;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

public class Client {
```

```

private CommandManager manager= new CommandManager();

public Client() {
    manager.add("default", new DefaultCommand());
}

```

Slijedi primjer ugrađivanja podataka izravno u izvorni kod (hard coding) preslikavanjem naredbe. Robusnija implementacija bi inicijalizirala preslikavanje naredbi iz resursne datoteke:

```

<commands>
  <command name="default" class="wrox.Pattern.command.DefaultCommand" />
</commands>

```

Nakon što metoda `invoke (String name)` primi zahtjeve, pronalazi naziv naredbe u komponenti `CommandManager` i vraća objekt `Command`:

```

public void invoke(String name) {
    Command command= manager.getCommand(name);
    command.execute();
}

public static void main(String[] args) {
    Client client= new Client();
    client.invoke("default");
}
}

```

Ovdje smo prikazali važnu sastavnicu većine mrežnih kostura, poput `Strutsa` ili `WebWorka`. U `WebWork` kosturu postoji specifični `Command` predložak naziva `xWork`, koji je podrobno opisan u poglavlju 8.

Tretiranjem svakoga zahtjeva kao `Command` objekta, svim je naredbama moguće dodijeliti zajedničke usluge poput sigurnosti, provjere valjanosti unosa i revizije. Sljedeći odjeljak proširit će trenutačni predložak `Command` i prikazati implementiranje sučelja `ManagedLifecycle`, koje će definirati skup metoda koje se pozivaju tijekom svakoga zahtjeva:

```

package wrox.Pattern.command;

import java.util.Collection;
import java.util.Map;

public interface ManagedLifecycle extends Command {

    public void initialize();
    public void setApplicationContext(Map context);
    public boolean isValid();
    public Collection getErrors( );
    public void destroy();

}

```

Sučelje `ManagedLifecycle` predstavlja odnos između objekta `Command` i klijentskoga koda. Slijedi primjer implementacije sučelja `ManagedLifecycle`:

```

package wrox.pattern.command;
import java.util.Collection;
import java.util.Map;

```

```

import java.util.HashMap;

public class ManagedCommand implements ManagedLifecycle {
    private Map context;
    private Map errors= new HashMap( );
    public void initialize() {
        System.out.println("initializing..");
    }
    public void destroy() {
        System.out.println("destroying");
    }
    public void execute() {
        System.out.println("executing managed command");
    }
    public boolean isValidate() {
        System.out.println("validating");
        return true;
    }
    public void setApplicationContext(Map context) {
        System.out.println("setting context");
        this.context= context;
    }
    public Collection getErrors() {
        return errors.getValues();
    }
}

```

Sljedeći kod ilustrira inicijalizaciju i pozivanje dvaju tipova naredbi - standardnih i upravljanih:

```

package wrox.pattern.command;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

public class Client {
    private Map context= new HashMap();
    private CommandManager manager= new CommandManager();

    public Client() {
        manager.add("default", new DefaultCommand());
    }
}

```

Klasi CommandManager dodan je novi ManagedCommand:

```

manager.add("managed", new ManagedCommand());
}
public void invoke(String name) {
    Command command= manager.getCommand(name);
}

```

Nadalje, provjera se implementira li naredba koja se izvodi sučelje ManagedLifecycle:

```

if (command instanceof ManagedLifecycle) {
    ManagedLifecycle managed= (ManagedLifecycle)command;
    managed.setApplicationContext(context);
    managed.initialize();
    if (managed.isValidiate()) {
        managed.execute();
    }
}

```

```

    } else {
        Collection errors = managed.getErrors();
    }
    managed.destroy();
} else {
    command.execute();
}
}
}

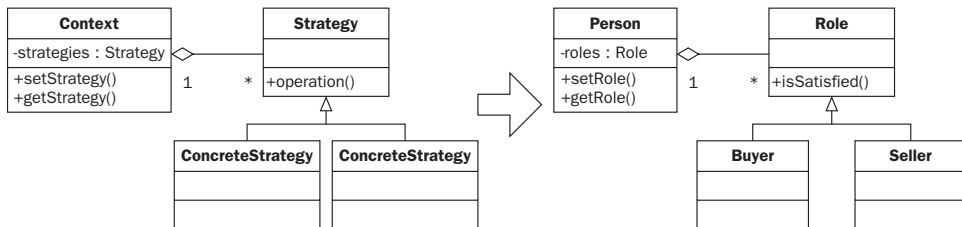
```

Sekvenca pozivanja sučelja `ManagedLifecycle` sadrži više funkcionalnosti, u usporedbi sa svojom inačicom koja sadrži samo jednu metodu. Najprije prosljeđuje potrebne aplikacijske podatke, zatim poziva inicijalizacijsku metodu, izvodi provjeru valjanosti unosa te poziva izvedbenu metodu.

*Omogućavanje pozivatelju klijenta da proslijedi resurse naredbi vrlo je moćan koncept, koji se naziva inverzija kontrole. Primjenom ovog koncepta nestaje potreba da klasa `Command` utvrđuje koje su usluge i resursi dostupni pozivatelju.*

## Predložak Strategy

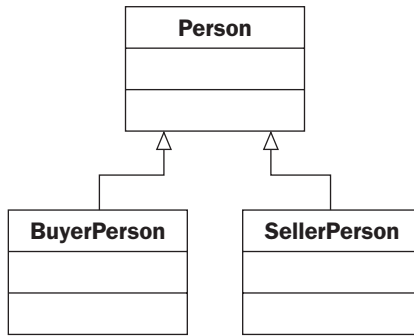
Predložak Strategy omogućava zamjenu algoritama za vrijeme izvođenja. Za implementaciju ovog rješenja, svaki algoritam potrebno je predstaviti kao stratešku klasu. Aplikacija će tada trenutačnoj strateškoj klasi delegirati izvođenje algoritma specifičnog za strategiju. Slika 3-14 prikazuje UML dijagram za strateški predložak, s uključenim primjerom iz ovog odjeljka.



Slika 3-14

Čest problem kod modeliranja domene jest pretjerana upotreba stvaranja *podtipova*. Podtip (engl. *subtype*) bi trebalo stvoriti samo u slučaju kada je između podtipa i njemu nadređenog tipa moguće opisati određenu „jeste“ (engl. *is-a*) vezu. Primjerice, kod stvaranja modela za neku osobu unutar modela domene, čini se primamljivim za svaku ulogu koju ova osoba ima stvoriti zasebni podtip. Ne postoji pogrešan način modeliranja problema, no u ovom konkretnom slučaju, svaka od uključenih osoba može imati po nekoliko uloga i stvaranje podtipova je ovdje pogrešno rješenje. Ponašanje svake uključene osobe ovisi o njezinoj ulozi, što je koncept koji se može izraziti predloškom Strategy.

U ovom ćemo odjeljku za primjer prikazati na koji je način različite uloge kupaca i prodavača moguće uključiti u strategiju. Naime, pogrešno je svakoj osobi dodijeljivati isključivo jednu ili drugu ulogu. Potreba promjene ponašanja, odnosno skakanja iz jedne u drugu klasu ponašanja u hijerarhiji klasa bit će nam motiv za korištenje predloška Strategy. Slika 3-15 prikazuje pogrešan način modeliranja veze između osobe i njezinih uloga:



Slika 3-15

Predložak Strategy sastoji se od sučelja koje definira ponašanje, implementacijske podklase za definiranje ponašanja i objekt, kako bi klasa mogla obaviti svoju zadaću.

## Strategija

Rješenje ovog problema jest oblikovanje svake uloge kao klase i delegiranje ponašanja karakterističnoga za svaku ulogu iz klase Person u klasu Role. Najprije ćemo pogledati ponašanje koje će se razlikovati u trenutačnom objektu stanja. U ovom primjeru, sučelje Role koristi se za deklaraciju ponašanja strategije, a dvije konkretne klase, Buyer i Seller, za implmentaciju različitih ponašanja.

Kako bismo primjeru dali više životnosti, pretpostavimo da kupac Buyer i prodavač Seller pokušavaju postići dogovor o cijeni proizvoda. Metoda `isSatisfied()` prosljeđuje se u `Product` i u `Price` te zatim i kupac i prodavač moraju odlučiti hoće li prihvatiti uvjete dogovora.

```

package wrox.pattern.strategy;

public interface Role {

    public boolean isSatisfied( Product product, double price );
}
  
```

Naravno, prodavač i kupac imaju različite ciljeve: prodavač želi ostvariti što veći profit i na svaki proizvod postavlja maržu od 20%. Sljedeći kod pravi takvu pretpostavku:

```

package wrox.pattern.strategy;
public class Seller implements Role {

    /*
     * Seller will be happy if they make 20% profit on whatever they sell.
     * (non-Javadoc)
     * @see wrox.Pattern.strategy.Role#isSatisfied(wrox.Pattern.strategy.Product,
     double)
     */
    public boolean isSatisfied(Product product, double price) {
        if (price - product.getCost() > product.getCost() * .2) {
            return true;
        } else {
            return false;
        }
    }
}
  
```

S druge strane, kupac traži proizvod koji se nalazi unutar cjenovnog okvira koji si može priuštiti. Važno je primijetiti kako klasa Buyer nije ograničena samo na metode opisane u sučelju Role, što omogućava postavljanje granične varijable članice limit u klasu Buyer, a koja neće biti prisutna u klasi Seller.

Algoritam prihvatljivih uvjeta je proizvoljan dio ovog primjera. Postavljen je na način da kupac ne može potrošiti više od odabranog limita i neće platiti više od iznosa koji je dva puta veći od početne cijene. Uloga klase Buyer izražena je u metodi isSatisfied():

```
package wrox.Pattern.strategy;
public class Buyer implements Role {

    private double limit;

    public Buyer(double limit) {
        this.limit= limit;
    }
    /*
     * The buyer is happy if he can afford the product,
     * and the price is less then 200% over cost.
     * @see wrox.Pattern.strategy.Role#isSatisfied(wrox.Pattern.strategy.Product,
double)
     */
    public boolean isSatisfied(Product product, double price) {
        if ( price < limit && price < product.getCost() * 2 ) {
            return true;
        } else {
            return false;
        }
    }
}
```

Kod koji slijedi za apstrakciju proizvoda koristi klasu. Riječ je o podatkovnom objektu koji je dio scenarija:

```
package wrox.pattern.strategy;
public class Product {
    private String name;
    private String description;
    private double cost;

    public Product(String name, String description, double cost) {
        this.name = name;
        this.description = description;
        this.cost = cost;
    }
}
```

Sljedeći odjeljak predstavlja klasu koja koristi strategiju uključivanja.

## Kontekst

Nadalje, pogledajmo klasu Person, koja radi s objektima Role. Ova klasa je asocijacijom povezana sa sučeljem Role. Osim toga, važno je reći kako sučelje Role ima definirane komponente za postavljanje i uzimanje, što omogućava promjenu uloga osobe tijekom izvođenja programa. Također, ovakav pristup rezultira i puno čistijim kodom. U ovomu

primjeru koriste se dvije uloge: Buyer i Seller. U budućnosti u klasu Role mogu biti dodavani i drugi objekti, poput Wholesaler, Broker i slično, budući da ne postoji podređenost nekoj podklasi.

```
package wrox.pattern.strategy;

public class Person {
    private String name;
    private Role role;
    public Person(String name) {
        this.name= name;
    }
    public Role getRole() {
        return role;
    }
    public void setRole(Role role) {
        this.role= role;
    }
}
```

Sljedeći ključan moment jest delegiranje specifičnog ponašanja Role od strane zadovoljene metode klase Person njezinom sučelju Role. Polimorfizam omogućava odabir ispravnih pozadinskih objekata:

```
public boolean satisfied(Product product, double offer) {
    return role.isSatisfied(product, offer);
}
}
```

Kod predložka ovime je implementiran. Sada ćemo vidjeti kako se aplikacija ponaša nakon njegove implementacije. Za početak, uvedimo Products, People i Roles:

```
package wrox.pattern.strategy;

public class Person {
    // Prethodne metode su izostavljene.

    public static void main(String[] args) {
        Product house= new Product("house", "4 Bedroom North Arlington", 200000);
        Product condo= new Product("condo", "2 Bedroom McLean", 100000);
        Person tim= new Person("Tim");
        Person allison= new Person("Allison");
    }
}
```

Bavite se kupoprodajom kuća. Sljedeći korak jest uvođenje početnih uloga i njihovo dodjeljivanje osobama. Nakon ovoga, osobe će se ponašati u skladu s ulogama koje su im dodijeljene:

```
tim.setRole(new Buyer(500000));
allison.setRole(new Seller());

if (!allison.satisfied(house, 200000)) {
    System.out.println("offer of 200,000 is no good for the seller");
}
if (!tim.satisfied(house, 600000)) {
    System.out.println("offer of 600,000 is no good for the buyer");
}
if (tim.satisfied(house, 390000) && allison.satisfied(house, 390000)) {
    System.out.println("They Both agree with 390,000 ");
}
```

Kako bismo prikazali dodatne mogućnosti predložka Strategy, početnog prodavača pretvorit ćemo u kupca, pozivanjem metode `setRole()` na objektu `Person`. Pretvaranje prodavača u kupca moguće je izvesti bez modificiranja objekta `Person`:

```

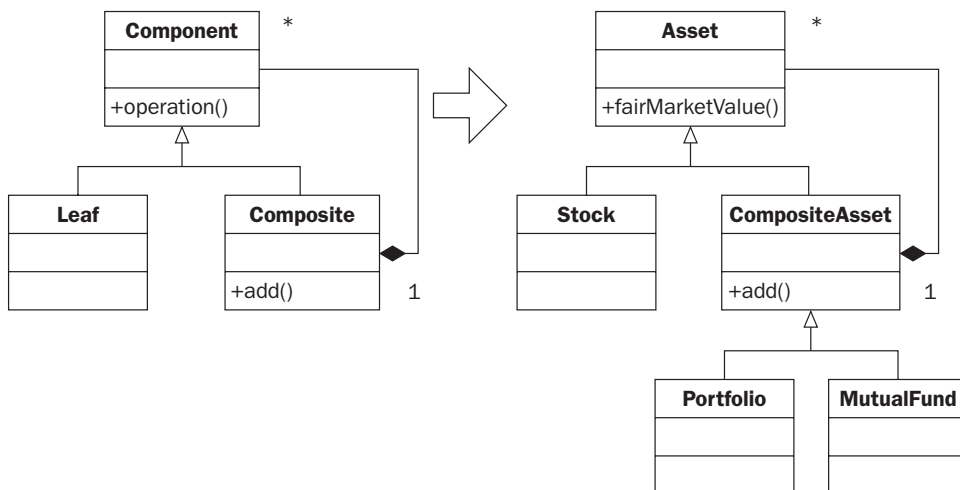
allison.setRole(new Buyer(190000));
    if (allison.satisfied(condo, 110000)) {
        System.out.println("As a buyer she can afford the condo ");
    }
}
}
}
}

```

Implementacijom predložka Strategy omogućena je promjena ponašanja objekta tijekom izvođenja i bez učinka na njegovu implementaciju. Ovo je vrlo snažan alat u softverskom dizajnu. U sljedećem odjeljku bavit ćemo se stvaranjem složenih predložaka na principu apstrakcije ponašanja, a koji će nam pomoći da hijerarhiji klasa pristupamo s pomoću istoga zajedničkog sučelja.

## Predložak Composite

Predložak Composite omogućit će nam tretiranje cijele kolekcije objekata kao da je riječ o samo jednom objektu. Na ovaj način smanjit ćemo kompleksnost programskoga koda u odnosu na kod koji bi nam bio potreban za tretiranje svake kolekcije zasebno. Slika 3-16 prikazuje strukturu predložka Composite koji je povezan s klasama koje ga implementiraju.



Slika 3-16

Primjer koji ovdje koristimo za ilustriranje ponašanja je sustav za upravljanja portfeljom vrijednosnica, koji se sastoji od dionica i zajedničkih fondova. Zajednički fondovi jesu skupine dionica, ali želimo koristiti isto sučelje i za dionice i za zajedničke fondove, kako bismo pojednostavili manipulaciju i jednom i drugom grupom. To će nam omogućiti izvođenje operacija poput izračuna tržišne cijene vrijednosnica, njihove kupnje ili prodaje te izračuna postotka prinosa, a sve korištenjem jednoga sučelja. Predložak Composition u svakom će slučaju smanjiti kompleksnost stvaranja ovih operacija. On se sastoji od klasa `Leaf` i `Composite` a ilustrira se na slici 3-16.

## Sučelje Component

Prvo sučelje je `Component`, koje deklarira zajedničko sučelje koje će koristiti i pojedinačni i složeni čvorovi. Slijedi primjer korištenja metode `fairMarketValue`, operacije koja se može koristiti sa dionicama, zajedničkim fondovima i portfeljima:

```
package wrox.pattern.composite;

public interface Asset {

    public double fairMarketValue();
}
```

## Klasa stranica Leaf

Klasa `Leaf` predstavlja pojedinačne temeljne tipove podataka koji implementiraju komponentu sučelja. U ovom primjeru, klasa `Stock` predstavlja čvor stranice predloška. Klasa `Stock` je čvor stranice u smislu da ne drži referencu na bilo koji drugi `Asset` objekt:

```
package wrox.pattern.composite;

public class Stock implements Asset {

    private String name;
    private double price;
    private double quantity;

    public Stock(String name, double price, double quantity) {
        this.name= name;
        this.price= price;
        this.quantity= quantity;
    }
}
```

Cijena dionica izračunava se množenjem cijene jedne dionice s brojem dionica:

```
public double fairMarketValue() {

    return price * quantity;
}
}
```

## Composite

Sljedeći odjeljak prikazuje `Composite` objekt po imenu `CompositeAsset`. Potrebno je napomenuti kako je `CompositeAsset` deklariran apstraktno. Dopuštena složena imovina, poput zajedničkoga fonda ili portfelja, proširuje ovu apstraktnu klasu:

```
package wrox.pattern.composite;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public abstract class CompositeAsset implements Asset {
    private List assets= new ArrayList();

    public void add(Asset asset) {
        assets.add(asset);
    }
}
```

Izvedimo iteraciju kroz investicije nižega reda. U slučaju da je neka od ovih investicija u isto vrijeme i složena imovina, bit će ponovno obrađena - bez potrebe za stvaranjem posebnog slučaja. Ovako bi, primjerice, mogli imati zajednički fond koji se sastoji od zajedničkih fondova:

```
public double fairMarketValue() {
    double total = 0;
    for (Iterator i= assets.iterator(); i.hasNext(); ) {
        Asset asset= (Asset)i.next();
        total = total + asset.fairMarketValue();
    }
    return total;
}
}
```

Nakon što smo s time završili, slijedi stvaranje konkretnih složenih objekata: MutualFund i Portfolio. Za stvaranje klase MutualFund nije nam potrebno puno - ona će svoje ponašanje naslijediti od objekta CompositeAsset:

```
package wrox.pattern.composite;

public class MutualFund extends CompositeAsset{

    private String name;

    public MutualFund(String name) {
        this.name = name;
    }

}
```

I klasa Portfolio proširuje CompositeAsset. Razlika je u tome što izravno poziva klasu višeg reda i modificira rezultate kalkulacije tržišne cijene – od iznosa oduzima 2% kao naknadu za upravljanje vrijednosnicama:

```
package wrox.pattern.composite;

public class Portfolio extends CompositeAsset {
    private String name;
    public Portfolio(String name) {
        this.name= name;
    }
    /* Market value - Management Fee
    * @see wrox.Pattern.composite.CompositeAsset#fairMarketValue()
    */
    public double fairMarketValue() {
        return super.fairMarketValue() - super.fairMarketValue() * .02;
    }

}
```

Ostalo nam je još samo izvođenje koda. Sljedeća klasa jest klasa investitora - Investor. To je zapravo klijentski kod koji također koristi prednosti složenog predložka Composite:

```
package wrox.pattern.composite;

public class Investor {
```

```

private String name;
private Portfolio portfolio;
public Investor(String name, Portfolio portfolio) {
    this.name= name;
    this.portfolio= portfolio;
}

```

Određivanjem tržišne cijene investitorovog portfelja, predložak Composite napraviti će presjek kolekcije dionica i zajedničkih fondova - s ciljem utvrđivanja vrijednosti kolekcije, a bez straha od narušavanja strukture objekta:

```

public double calcNetworth( ){

    return portfolio.fairMarketValue();
}

public static void main(String[] args) {
    Portfolio portfolio= new Portfolio("Frequently Used Money");
    Investor investor= new Investor("IAS", portfolio);

    portfolio.add(new Stock("wrox", 450, 100));

    MutualFund fund= new MutualFund("Don Scheafer's Intellectual Capital");
    fund.add(new Stock("ME", 35, 100) );
    fund.add(new Stock("CV", 22, 100) );
    fund.add(new Stock("BA", 10, 100) );
    portfolio.add(fund);

    double total =investor.calcNetworth();

    System.out.println("total =" + total);
}
}

```

Korištenjem predložka composite vrlo je lako pojednostaviti operacije koje se izvode nad kompleksnim strukturama podataka.

## Sažetak

U ovom smo se poglavlju upoznali s prednostima koje predložci dizajna imaju u razvoju Java programskih rješenja. Predložci su presudni pri učenju iz iskustava drugih programera, ali i pri razumijevanju API sučelja koja se koriste na Java platformi.

Upoznali smo se s predlošcima dizajna, naučili zašto su važni i saznali „trikove“ koji omogućavaju njihovo lakše razumijevanje, a detaljnije smo upoznali nekoliko predložaka koji su važni za programiranje u Javi.

Budući da ste dosad naučili kako razmišljaju Java programeri, ostatak knjige bit će posvećen praktičnim Java softverskim rješenjima. Poglavlja se neće sastojati od detaljnog predstavljanja tehnologija, već od primjera iz stvarnoga života, koji se rješavaju primjenom različitih tehnologija.

Prvo poglavlje ovog novog dijela knjige jest poglavlje 4 „Razvoj učinkovitih sučelja korištenjem JFC alata“. U tom ćemo poglavlju naučiti kako koristiti Swing biblioteke za razvoj samostalnih Java aplikacija.