Poglavlja

	Uvod	xxv
1	Dobro došli u projektne obrasce: *uvod u projektne obrasce*	1
2	Neka objekti budu u toku: *obrazac Observer*	37
3	Dekorisanje objekata: *obrazac Decorator*	79
4	Dobrobiti OO dizajna: *obrazac Factory*	109
5	Jedinstveni objekti: *obrazac Singleton*	169
6	Kapsuliranje poziva: *obrazac Command*	191
7	Biti adaptivan: *obrasci Adapter i Facade*	237
8	Kapsuliranje algoritama: *obrazac Template Method*	277
9	Kolekcije kojima se dobro upravlja: *obrasci Iterator i Composite*	317
10	Stanje stvari: *obrazac State*	381
11	Upravljanje pristupom objektima: *obrazac Proxy*	425
12	Obrasci obrazaca: *kombinovani obrasci*	493
13	Obrasci u stvarnosti: *bolji život sa obrascima*	563
14	Dodatak: *preostali obrasci*	597

Sadržaj

Uvod

Vaš mozak o projektnim obrascima. I evo *vas*, pokušavate da *naučite* nešto, a vaš *mozak* vam čini uslugu starajući se da vam ništa od toga ne *ostane* u glavi. Vaš mozak misli, „Bolje da ostavim mesta za bitnije stvari, kao što su divlje životinje koje treba izbegavati i procenu da li je snoubording bez odeće loša zamisao." Pa, kako ćete prevatiti svoj mozak da misli da vam život zavisi od poznavanja projektnih obrazaca?

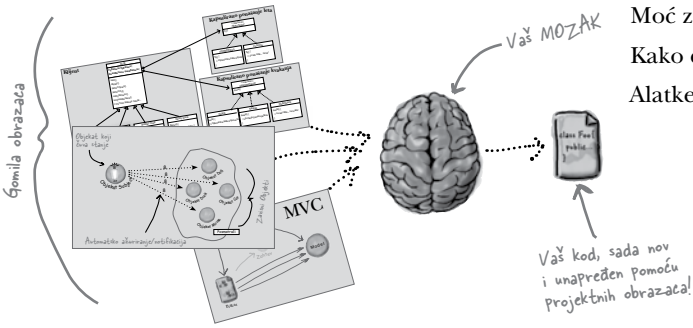
Kome je knjiga namenjena?	xxvi
Znamo šta mislite.	xxvii
I znamo šta misli vaš mozak.	xxvii
Čitaoca knjige „Um caruje" smatramo učenikom.	xxviii
Metakognicija: razmišljanje o razmišljanju	xxix
Evo šta smo MI uradili:	xxx
Evo šta VI možete da uradite da biste pokorili svoj mozak	xxxi
Pročitaj me	xxxii
Recenzenti	xxxiv
Zahvalnice	xxxvi

1 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*.



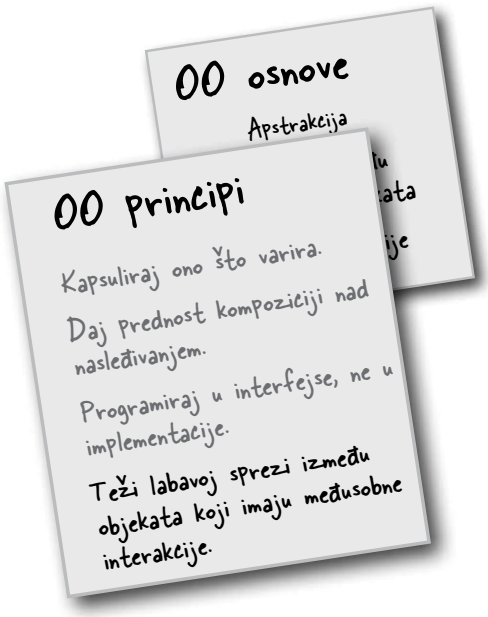
Počelo je jednostavnom aplikacijom SimUDuck 2
Ali patke sada treba da LETE 3
Ali, nešto je pošlo naopako... 4
Džo razmišlja o nasleđivanju... 5
A šta je sa interfejsom? 6
Šta biste vi uradili da ste Džo? 7
Jedina konstanta u razvoju softvera 8
Fokusiranje na problem... 9
Odvajanje onoga što se menja od onoga što ostaje isto 10
Projektovanje ponašanja pataka 11
Implementiranje pačjih ponašanja 13
Integrisanje pačjih ponašanja 15
Testiranje koda Duck 18
Dinamičko zadavanje ponašanja 20
Šira slika kapsuliranih ponašanja 22
IMA može biti bolje nego JESTE 23
Kad smo kod projektnih obrazaca... 24
Načuli smo u lokalnom restoranu... 26
Načuli smo za susednim stolom... 27
Moć zajedničkog rečnika obrazaca 28
Kako da koristim projektne obrasce? 29
Alatke za projektovanje 32



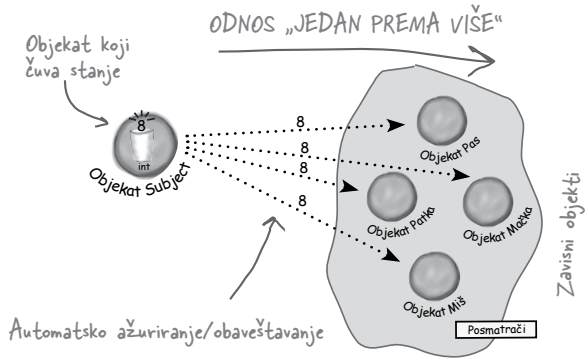
obrazac Observer

2 Neka objekti budu u toku

Ne želite da propustite neko zanimljivo dešavanje, zar ne? Imamo obrazac koji *upućuje* objekte kada se dešava nešto *do čega im je stalo*. To je obrazac Observer. To je jedan od najčešće korišćenih projektnih obrazaca, i neverovatno je koristan. Razmotrićemo sve zanimljive aspekte Posmatrača, kao što su *veza jedan-prema-više* i *labava sprega*. Imajući te koncepte na umu, kako možete odoleti da ne budete najzanimljiviji na Zabavi obrazaca?



Pregled aplikacije za praćenje vremena	39
Upoznajte obrazac Observer	44
Izdavači + pretplatnici = obrazac Observer	45
Definicija obrasca Observer	51
Moć labavog povezivanja	54
Projektovanje meteorološke stanice	57
Implementiranje meteorološke stanice	58
Pokretanje meteorološke stanice	61
Kodiranje aplikacije koja menja život	66
U međuvremenu, u Weather-O-Rami	69
Testiranje novog koda	71
Alatke za projektovanje	72
Izazov principa projektovanja	73



3 Dekorisanje objekata

Samo nazovite ovo poglavlje „Dizajnersko oko za naslednog tipa“. Preispitaćemo tipično preterano korišćenje nasleđivanja i saznaćete kako da dekorišete klase za vreme izvršavanja, koristeći jedan vid kompozicije objekata. Zašto? Kada upoznate tehnike dekorisanja, moći ćete da date svojim (ili tuđim) objektima nova zaduženja, *nepravećи nikakve izmene koda osnovnih klasa.*



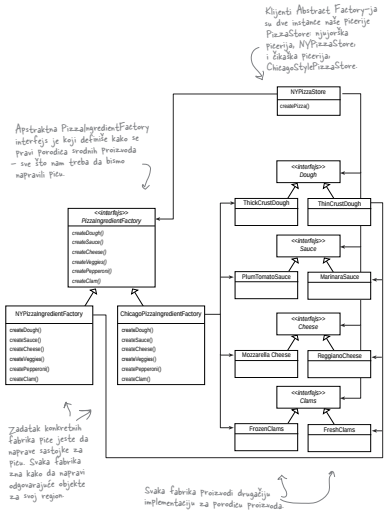
Dobro došli na Starbuzz kafu	80
Princip otvorenosti i zatvorenosti	86
Upoznajte obrazac Decorator	88
Konstruisanje porudžbine pomoću dekoratora	89
Definicija obrasca Decorator	91
Dekorisanje napitaka	92
Pisanje koda za Starbuzz	95
Kodiranje napitaka	96
Kodiranje dodataka	97
Posluživanje kafe	98
Stvarni dekoratori: Java I/O	100
Dekorisanje java.io klasa	101
Pisanje Java I/O dekoratora	102
Testirajte svoj prvi Java I/O dekorator	103
Alatke za projektovanje	105

obrazac Factory

4 Dobrobiti OO dizajna

Spremite za pravljenje slabo povezanih OO projekata.

Pravljenje objekata ne svodi se samo na korišćenje operatora **new**. Naučićete da je instanciranje aktivnost koju ne bi uvek trebalo da radite u javnosti i da ono često može voditi do *problema sa povezivanjem*. A to ne želimo, zar ne? Saznajte kako obrasci Factory mogu da vas poštede sramotnih zavisnosti.



Prepoznavanje aspekata koji variraju	112
Kapsuliranje pravljenja objekata	114
Pravljenje proste fabrike pica	115
Definicija Simple Factory-ja	117
Radni okvir za piceriju	120
Dopuštanje potklasama da odlučuju	121
Deklarisanje fabričke metode	125
Došlo je vreme da upoznate obrazac Factory Method	131
Paralelan pregled kreatora i proizvoda	132
Definicija obrasca Factory Method	134
Razmatranje zavisnosti objekata	138
Princip inverzije zavisnosti	139
Primenjivanje principa	140
Porodice sastojaka...	145
Izgradnja fabrika sastojaka	146
Prepravljanje pica...	149
Ponovna poseta picerijama	152
Šta smo uradili?	153
Definicija obrasca Abstract Factory	156
Poređenje Factory Method i Abstract Factory	160
Alatke za projektovanje	162

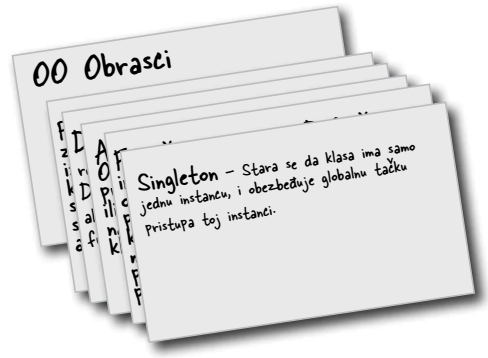


5 Jedinstveni objekti

Naša sledeća stanica je obrazac Singleton, naša karta za pravljenje jedinstvenih objekata za koje će uvek postojati samo jedna instanca. Možda će vas obradovati da znate da je od svih obrazaca, obrazac Singleton (eng. *Singleton Pattern*) najjednostavniji po pitanju dijagrama klasa; štaviše, dijagram ima samo jednu klasu! Ali nemojte se previše opuštati; uprkos jednostavnosti iz perspektive dizajna klasa, on zahteva ozbiljno objektno orijentisano razmišljanje za svoju implementaciju. Zato, udahnite duboko, pa da krenemo.

Seciranje klasične implementacije obrasca Singleton	173
Fabrika čokolade	175
Definicija obrasca Singleton	177
~~Hjustone~~ Hershey, PA, imamo problem...	178
Višenitni rad	180
Možemo li da popravimo višenitni rad?	181
U međuvremenu, u fabrici čokolade...	183
Alatke za projektovanje	186

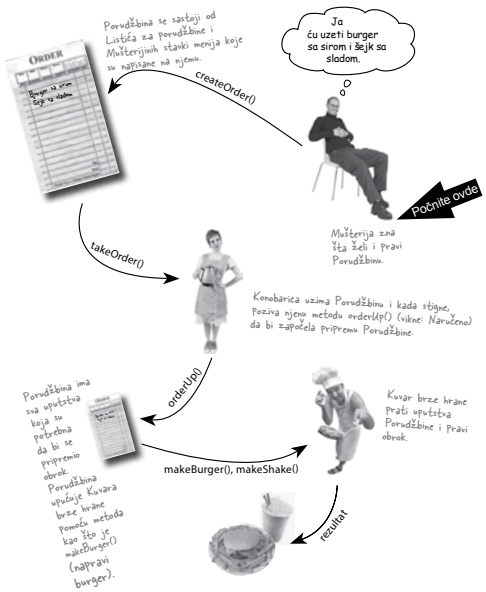




obrazac Command

6 Kapsuliranje poziva

U ovom poglavlju podižemo kapsuliranje na sasvim nov nivo: kapsuliraćemo pozivanje metode. Baš tako – kapusiliranjem pozivanja metode, možemo da iskristališemo delove izračunavanja, pa objekat koji poziva izračunavanje ne mora da brine o tome kako da nešto uradi, već prosto koristi našu iskristalisanu metodu da bi to obavio. Sa kapsuliranim pozivima metoda možemo da uradimo i neke lukave stvari, na primer da ih sačuvamo za evidentiranje ili da ih iznova koristimo da bismo u kodu implementirali funkcionalnost poništavanja akcije.



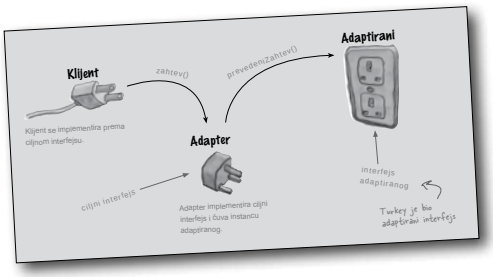
Automatizacija kuće ili propast	192
Pogled na klase dobavljača	194
Kratak uvod u obrazac Command	197
Od restorana do obrasca Command	201
Naš prvi komandni objekat	203
Korišćenje komandnog objekta	204
Dodeljivanje komandi pozicijama	209
Implementiranje sistema daljinskog upravljanja	210
Implementiranje komandi	211
Provera rada sistema daljinskog upravljanja	212
Vreme je da se napiše dokumentacija...	215
Šta radimo?	217
Vreme je za proveru kvaliteta tastera Undo	220
Korišćenje stanja za implementiranje Undo	221
Dodavanje Undo komandama plafonskog ventilatora	222
Svakom daljinskom treba režim Party!	225
Korišćenje makro komande	226
Još namena obrasca Command: zahtevi na čekanju	229
Još namena obrasca Command: evidencija zahteva	230
Obrazac Command u stvarnosti	231
Alatke za projektovanje	233

7 Biti adaptivan

U ovom poglavlju ćemo pokušati neverovatne poduhvate kao što je umetanje četvrtastog klina u oblu rupu. Zvuči nemoguće? Ne kada imamo projektne obrasce. Sećate se obrasca Decorator? **Omotavali smo objekte** da bismo im davali nova zaduženja. Sada ćemo omotati objekte sa drugačijom namerom: da bi njihovi interfejsi izgledali kao nešto drugo. Zašto bismo to radili? Da bismo dizajn koji očekuje jedan interfejs prilagodili klasi koja implementira drugačiji interfejs. I to nije sve; kad smo već kod toga, pogledaćemo još jedan obrazac koji omotava objekte da bi pojednostavio njihov interfejs.



Adapteri su svuda oko nas	238
Objektno orijentisani adapteri	239
Ako hoda kao patka i kvače kao patka, onda to mora može biti patka ćurka umotana u pačji adapter...	240
Proba adaptera	242
Objašnjenje obrasca Adapter	243
Definicija obrasca Adapter	245
Adapteri objekta i klase	246
Adapteri u stvarnom svetu	250
Adaptiranje enumeracije na iterator	251
„Dome, slatki dome“ bioskop	257
Gledanje filma (na teži način)	258
Svetla, kamera, Facade!	260
Konstruisanje fasade kućnog bioskopa	263
Implementiranje pojednostavljenog interfejsa	264
Vreme je za gledanje filma (na lakši način)	265
Definicija obrasca Facade	266
Princip najmanjeg znanja	267
Kako NE steći prijatelje i uticati na objekte	268
Obrazac Facade i princip najmanjeg znanja	271
Alatke za projektovanje	272



obrazac Template Method

8 Kapsuliranje algoritama

Zahuktali smo se sa kapsuliranjem; kapsulirali smo pravljenje objekata, pozivanje metoda, složene interfejse, patke, pice... šta bi moglo da bude sledeće?

Bacićemo se na kapsuliranje delova algoritama, tako da klase mogu da se prikače na izračunavanje kad god požele. Učićemo i o principu projektovanja koji je inspirisan Holivudom. Da počnemo...



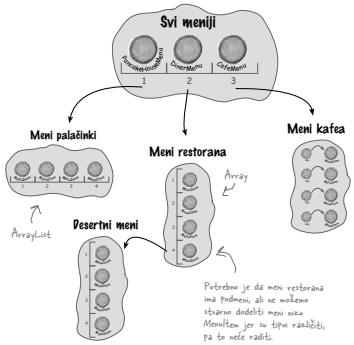
Vreme je za još kofeina	278
Mućkanje kafe i čaja (u Javi)	279
Apstrahujmo Coffee i Tea	282
Dalje doterivanje projekta...	283
Apstrahovanje prepareRecipe()	284
Šta smo uradili?	287
Upoznajte Template Method	288
Šta nam je doneo Template Method?	290
Definicija obrasca Template Method	291
Priključeni na šablonsku metodu...	294
Korišćenje priključaka	295
Holivudski princip i Template Method	299
Šablonske metode u divljini	301
Sortiranje pomoću Template Method	302
Treba da soritramo neke patke...	303
Šta je compareTo()?	303
Poređenje pataka i pataka	304
Hajde da sortiramo patke	305
Izrada mašine za sortiranje pataka	306
Razmahivanje sa okvirima	308
Namenske liste pomoću AbstractList	309
Alatke za projektovanje	313

9 Kolekcije kojima se dobro upravlja

Ima mnogo načina da se objekti naguraju u kolekciju.

Smestite ih u bilo koji niz, stek, listu, heš mapu – izaberite sami. Svaki način ima svoje prednosti i nedostatke. Ali u nekom trenutku, vaši klijenti će želeti da prave iteracije nad tim objektima, a kada se to desi, da li ćete im otkriti svoju implementaciju? Iskreno se nadamo da nećete! To prosto ne bi bilo profesionalno. Pa, ne morate da rizikujete svoju karijeru; u ovom poglavlju ćete videti kako možete klijentima dozvoliti da prolaze kroz objekte, a da nikada ne vide na koji način skladištite objekte. Naučićete i kako da napravite vrhunske kolekcije objekata koje u jednom skoku mogu da preskoče impresivne strukture podataka. A ako to nije dovoljno, naučićete i ponešto o odgovornostima objekata.



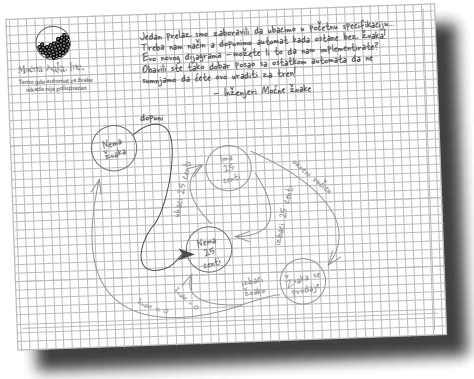


Najnovija vest: spajanje restorana Objektgrad i palačinkarnice Objektgrad	318
Pogledajte stavke menija	319
Implementiranje specifikacija: prvi pokušaj	323
Možemo li da kapsuliramo iteraciju?	325
Upoznajte obrazac Iterator	327
Dodavanje iteratora u DinerMenu	328
Prepravljanje menija restorana pomoću Iterator-a	329
Popravljanje koda Waitress	330
Testiranje koda	331
Analiza trenutnog dizajna...	333
Raščišćavanje pomoću java.util.Iterator	335
Definicija obrasca Iterator	338
Struktura obrasca Iterator	339
Princip jedne odgovornosti	340
Upoznajte Javin iterfejs Iterable	343
Javina poboljšana petlja for	344
Pogled na meni kafea	347
Iteratori i kolekcije	353
Da li je konobarica spremna za udarni termin?	355
Definicija obrasca Composite	360
Dizajniranje menija sa Composite-om	363
Implementacija MenuComponent	364
Implementacija MenuItem	365
Implementiranje kompozitne klase Menu	366
A sad testiranje...	369
Alatke za projektovanje	376

obrazac State

10 Stanje stvari

Manje poznata činjenica: obrasci Strategy i State su blizanci razdvojeni na rođenju. Pomislili biste da će oni živeti sličnim životima, ali obrazac Strategy je nakon toga stvorio veoma uspešan biznis širom međusobno zamenjivih algoritama, dok je obrazac State pošao, možda, plemenitijim putem pomaganja objektima da upravljaju svojim ponašanjem putem menjanja svog unutrašnjeg stanja. Ipak, koliko god da su njihovi putevi postali različiti, u pozadini ćete pronaći gotovo potpuno isti dizajn. Kako je to moguće? Kao što ćete videti, strategija i stanje imaju veoma različite svrhe. Pozabavimo se prvo onim o čemu se radi u obrascu State, a njihov odnos ćemo istražiti na kraju poglavlja.



Java žvaka	382
Osnove mašina stanja	384
Pisanje koda	386
Provera pre isporuke	388
Znali ste da će se desiti... promena zahteva!	390
Zbrkano STANJE stvari...	392
Novi dizajn	394
Definisanje interfejsa i klasa State	395
Prepravljanje automata za žvake	398
Pogledajmo sada celu klasu GumballMachine...	399
Implementiranje još stanja	400
Definicija obrasca State	406
Treba da završimo igru 1 od 10	409
Dovršavanje igre	410
Demo za direktora Moćne žvake, Inc.	411
Zdravorazumska provera...	413
Zamalo da zaboravimo!	416
Alatke za projektovanje	419

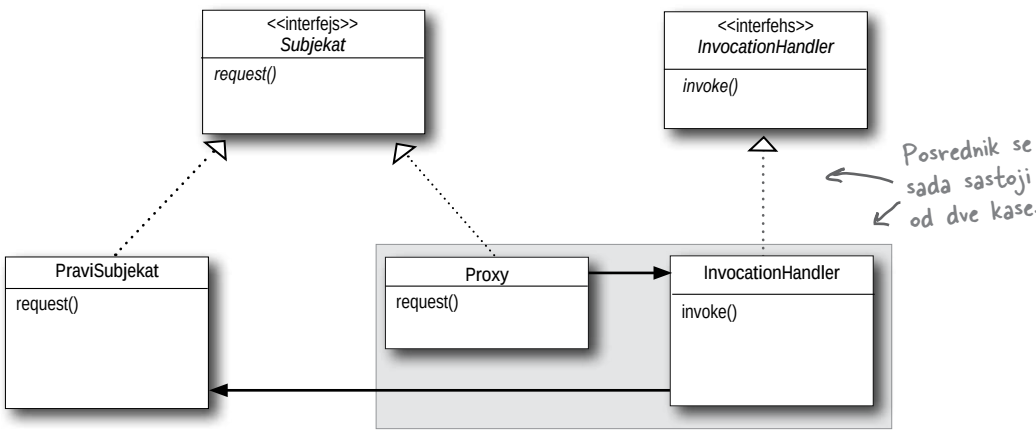


11 Upravljanje pristupom objektima

Da li ste se ikada igrali dobrog i lošeg policajca? Vi ste dobar policajac i pružate sve svoje usluge na fin i prijateljski način, ali ne želite da vam svako traži usluge, pa imate lošeg policajca koji kontroliše koliko ćete biti dostupni. To je ono što rade posrednici: kontrolišu i upravljaju pristupom. Videćete da, postoji mnogo načina na koje posrednici zamenjuju objekte kojima posreduju. Posrednici su poznati po tome da za objekte kojima posreduju mogu da preuzmu čitave pozive za metode preko interneta; i da strpljivo odmenjuju neke prilično lenje objekte.



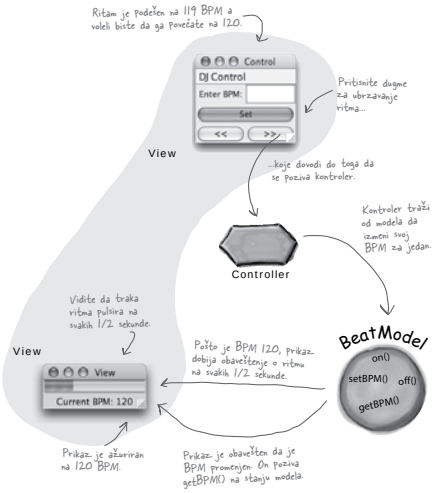
Kodiranje monitora	427
Testiranje monitora	428
Osnove udaljenih metoda	433
Pripremanje GumballMachine da bude udaljeni servis	446
Registrovanje u registru RMI...	448
Definicija obrasca Proxy	455
Spremite se za virtuelnog posrednika	457
Projektovanje virtuelnog posrednika za korice albuma	459
Pisanje klase ImageProxy	460
Korišćenje Proxyja Java API-ja za pravljenje zaštitnog posrednika	469
Upoznavanje u Objektgradu	470
Implementacija interfejsa Person	471
Petominutna drama: zaštita subjekata	473
Šira slika: pravljenje dinamičkog posrednika za Person	474
Zoološki vrt posrednika	482
Alatke za projektovanje	485
Kod za prikazivač korica albuma	489



kombinovani obrasci

12 Obrasci obrazaca

Ko bi ikad pomislio da bi projektni obrasci mogli da rade zajedno? Već ste bili svedoci oštrih Ćaskanja uz vatricu (a čak niste ni videli rubrike Borbe obrazaca na život i smrt, jer nas je urednik primorao da ih uklonimo iz knjige), pa ko bi mogao pomisliti da obrasci zapravo mogu dobro da rade jedan s drugim? Verovali ili ne, neki od najmoćnijih OO projekata koriste istovremeno više obrazaca. Spremite se da svoje veštine korišćenja obrazaca podignete na naredni nivo; vreme je za kombinovane obrasce.

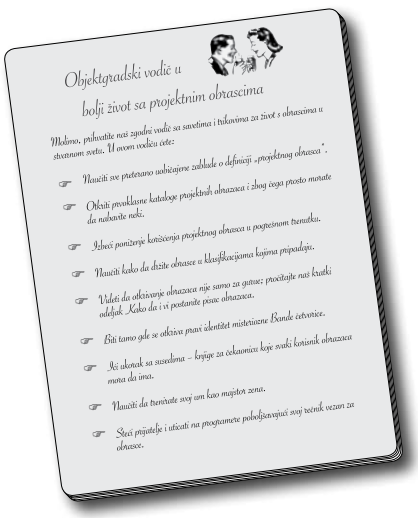




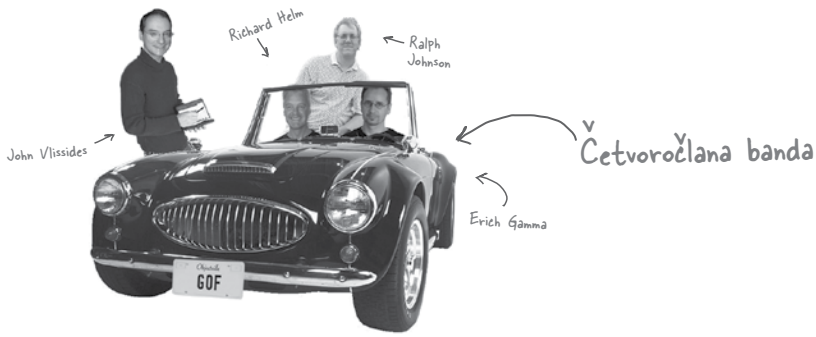
Udruženi rad	494
Ponovno okupljanje pataka	495
Šta smo uradili?	517
Pogled iz pačje perspektive: dijagram klasa	518
Kralj kombinovanih obrazaca	520
Upoznajte Model-View-Controller	523
Detaljniji pogled...	524
Razumevanje MVC-a kao skupa obrazaca	526
Upotreba MVC-a za upravljanje ritmom...	528
Izgradnja delova	531
Pogledajmo sada konkretnu klasu BeatModel	532
Prikaz	533
Implementiranje prikaza	534
A sada kontroler	536
Sklapanje svega...	538
Istraživanje strategije	539
Prilagođavanje modela	540
A sada proba...	542
Alatke za projektovanje	545

13 Obrasci u stvarnosti

Ahhhh, sada ste spremni za sjajan novi svet ispunjen projektnim obrascima. Ali, pre nego što počnete da otvarate sva ta vrata mogućnosti, treba da se pozabavimo još nekim pojedinostima na koje ćete nailaziti u stvarnom svetu – tako je, stvari su nešto složenije nego što su ovde, u Objektgradu. Idemo, na narednoj strani imamo fini vodič koji će vam pomoći sa prelaskom...

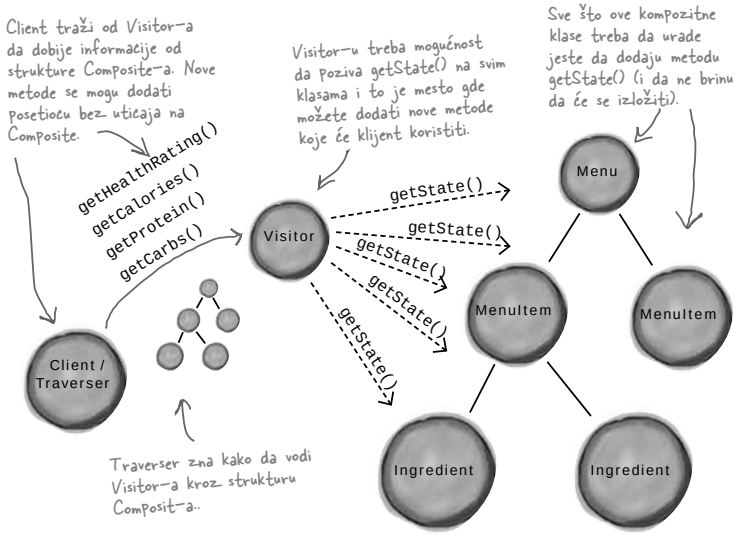


Definicija projektnog obrasca	565
Pažljivije razmatranje definicije projektnog obrasca	567
Neka sila bude sa vama	568
Želiš li biti pisac projektnih obrazaca	573
Organizovanje projektnih obrazaca	575
Razmišljanje u obrascima	580
Vaš um o obrascima	583
Zen um	583
Ne zaboravite moć zajedničkog rečnika	585
Krstarenje Objektgradom sa Četvoročlanom bandom	587
Vaše putovanje je tek počelo...	588
Zoološki vrt obrazaca	590
Uništavanje zla pomoću anti-obrazaca	592
Alatke za projektovanje	594
Napuštamo Objektgrad...	595



14 Dodatak: preostali obrasci

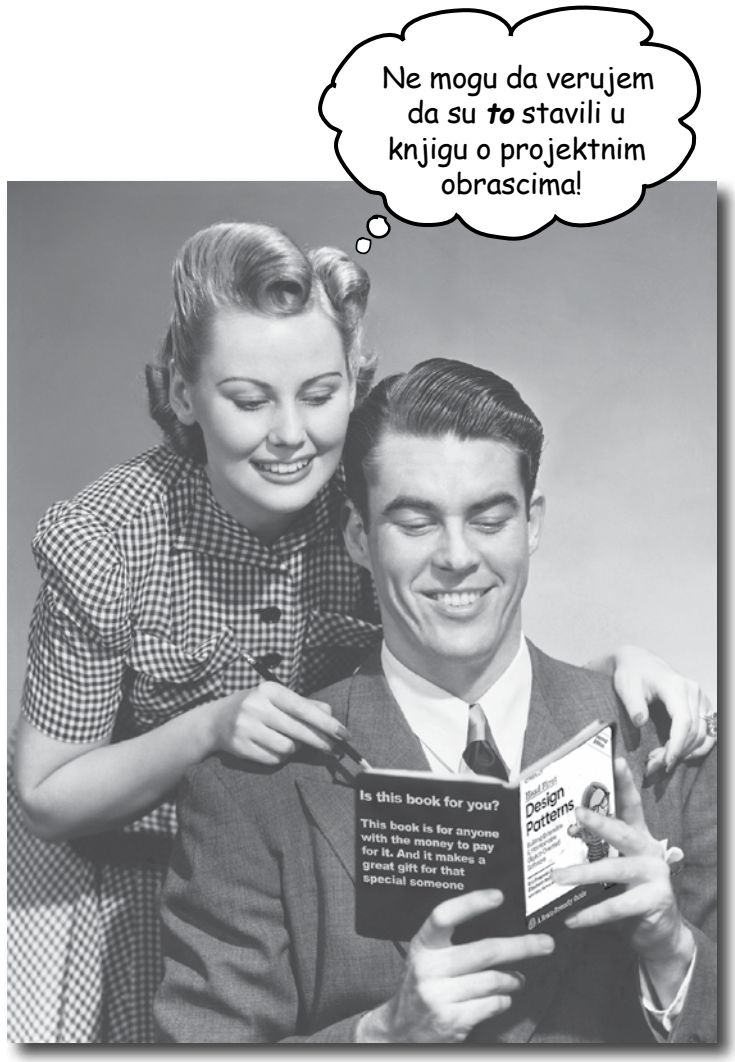
Ne mogu svi biti najpopularniji. Mnogo toga se promenilo u poslednjih 25 i više godina. Otkako je knjiga *Design Patterns: Elements of Reusable Object-Oriented Software* prvi put izdata, programeri su primenili ove obrasce hiljadama puta. Obrasci čiji rezime dajemo u ovom dodatku kompletni su, punopravni i zvanični GoF obrasci, ali se ne koriste toliko često kao obrasci koje smo do sada istraživali. Međutim, ovi obrascdi su sami po sebi zadivljujući i ako ih vaše okolnosti zahtevaju, trebalo bi da ih s ponosom primenite. Naš cilj u ovom dodatku je da vam na visokom nivou predstavimo šta su ti obrasci.



Bridge 598
Builder 600
Chain of Responsibility 602
Flyweight 604
Interpreter 606
Mediator 608
Memento 610
Prototype 612
Visitor 614

i Indeks 617

Uvod



U ovom odeljku odgovaramo na goruće pitanje: „Zbog čega su, u stvari, stavili to u knjigu o projektnim obrascima?"

Kome je knjiga namenjena?

Ako na sva naredna pitanja odgovorite sa „da”:

Svi naši primeri su na Javi, ali trebalo bi da možete da razumete glavne koncepte knjige ako znate neki drugi objektno orijentisani jezik.

1. Da li poznajete **Javu** (ne morate biti guru) ili drugi objektno orijentisani jezik?
2. Želite li da **naučite**, **razumete**, **zapamtite** i ***primenjujete*** projektne obrasce, uključujući principe OO projektovanja na kojima su projektni obrasci zasnovani?
3. Da li više volite **stimulativan razgovor uz večeru** nego suvoparna, dosadna, akademska predavanja?

ova knjiga je za vas.

Ko bi verovatno trebalo da odustane od ove knjige?

Ako odgovorite sa „da” na bilo koje od ovih pitanja:

1. Da li ste potpuni početnik u **objektno orijentisanom programiranju?**
2. Da li ste projektant/programer koji razbija objektno orijentisano programiranje i traži ***knjigu enciklopedijskog tipa***?
3. Da li ste arhitekta koji traži projektne obrasce za ***preduzeće***?
4. **Plašite li se da probate nešto drugačije**? Da li biste radije da vam izvade koren zuba nego da nosite prugasto sa kariranim? Verujete li da stručna knjiga ne može biti ozbiljna ako su koncepti objektno orijentisanog programiranja antropomorfizovani?

ova knjiga nije za vas.



[Napomena marketinškog odeljenja: ova knjiga je za sve sa kreditnom karticom.]

Znamo šta mislite.

„Otkud je ovo ozbiljna knjiga o programiranju?”

„U čemu je stvar sa svim ovim slikama?”

„Mogu li zaista da učim na ovaj način?”

I znamo šta misli vaš mozak.



Vaš mozak žudi za novitetima. On uvek traga, pregleda, *iščekuje* nešto neobično. Tako je napravljen i to vam pomaže da preživite.

Danas je mala verovatnoća da ćete postati tigrova užina. Ali vaš mozak ipak motri. Nikad se ne zna.

Pa, šta mozak radi sa svim rutinskim, običnim, normalnim stvarima na koje nailazite? Sve što *može* da bi ih sprečio da ometaju *pravi* posao mozga – beleženje bitnih stvari. On se ne bakće čuvanjem dosadnih stvari; one nikada ne prođu filtar „ovo očigledno nije bitno”.

Kako vaš mozak *zna* šta je bitno? Pretpostavimo da ste otišli na pešačenje i tigar iskoči pred vas. Šta se dešava u vašoj glavi i vašem telu?

Neuroni se zapale. Emocije prorade. *Hemikalije prostruje.*

Tako vaš mozak zna...

Ovo mora da je bitno! Nemoj da zaboraviš!

Ali zamislite da ste kod kuće ili u biblioteci. Bezbedno je, toplo, nigde tigrova. Učite. Spremate se za ispit. Ili pokušavate da naučite neku tešku stručnu temu za koju vaš šef smatra da će vam trebati sedmica ili deset dana u vrh glave.



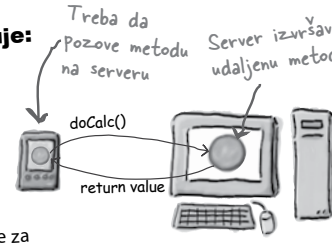
Postoji samo jedan problem. Vaš mozak pokušava da vam učini veliku uslugu. Trudi se da taj, *očigledno* nebitan, sadržaj ne zauzima oskudne resurse. Resurse koje je bolje utrošiti na čuvanje stvarno krupnih stvari. Kao što su tigrovi. Kao što je opasnost od vatre. Kao što je znanje da nikada više ne treba da skijate na dasci u šortsu.

I ne postoji jednostavan način da mu kažete, „Hej, mozgu, baš ti hvala, ali koliko god da je ova knjiga dosadna, i koliko god da sam slab po emocionalnoj Rihterovoj skali, ja zaista želim da zapamtim ove stvari.”

Čitaoca knjige „Um caruje" smatramo učenikom.

Pa, šta je potrebno da bi se nešto *naučilo*? Prvo, to nešto treba da *razumete*, a potom da se pobrinete da ga ne *zaboravite*. Nije stvar u natrpavanju činjenica u glavu. Na osnovu najnovijih istraživanja u kognitivnoj nauci, neurologiji i pishologiji obrazovanja, za *učenje* je potrebno mnogo više od teksta na strani. Mi znamo kako se vaš mozak uključuje.

Neki od principa učenja u knjigama Um caruje:



Neka bude vizuelno. Slike se lakše pamte od samih reči, a učenje postaje mnogo efikasnije (po istraživanju, do 89% poboljšanja u prisećanju i prenošenju). One čine stvari i razumljivijim. **Stavite reči unutar ili blizu slika** s kojima imaju veze, umesto na dno ili na drugu stranu, i udvostručićete verovatnoću da učenici reše probleme vezane za sadržinu.

Koristite konverzacijski i personalizovan stil. U novijim istraživanjima, učenici su na testovima nakon učenja postizali do 40% bolje rezultate ako se tekst obraćao direktno čitaocu, u prvom licu, konverzacijskim stilom umesto formalnim tonom. Ispričajte priču umesto da držite predavanje. Koristite opušten jezik. Ne uzimajte sebe preozbiljno. Kome biste *vi* posvetili više pažnje: društvu na stimulativnoj večernjoj zabavi ili nekom predavaču?

Truba je biti apstraktna metoda, a pri tome nemaš telo.



```
abstract void roam();
```

Metoda bez tela! Završava tačkom sa zarezom.

Navedite učenika da razmišlja dublje. Drugim rečima, ako aktivno ne istežete svoje neurone, u vašoj glavi se neće dešavati ništa naročito. Čitalac mora da bude motivisan, zaokupljen, radoznao i inspirisan da rešava probleme, izvlači zaključke i generiše novo znanje. I zbog toga su vama potrebni izazovi, vežbe i pitanja koja pobuđuju razmišljanje, kao i aktivnosti koje uključuju obe polovine mozga i više čula istovremeno.

Ima li smisla reći kada JESTE kupatilo? Kupatilo JESTE kada? Ili da li je to odnos IMA?



Zadobijte – i sačuvajte – pažnju čitaoca. Svi smo već prošli kroz situaciju „Stvarno želim ovo da naučim, ali uvek zaspim posle prve strane". Vaš mozak obraća pažnju na stvari koje su neuobičajene, zanimljive, čudne, privlačne, neočekivane. Učenje nove, teške, stručne teme ne mora biti dosadno. Vaš mozak će učiti mnogo brže ako nije tako.

Pogodite ih. Znamo da vaša sposobnost da nešto zapamtite uveliko zavisi od emotivnosti sadržaja. Pamtite ono do čega vam je *stalo*. Pamtite kada nešto *osetite*. Ne, ne govorimo o srceparajućim pričama o dečaku i njegovom psu. Govorimo o osećanjima kao što su iznenađenje, radoznalost, zabava, „šta, do…?", i o osećaju „Pa ja razbijam ovo!" koji dolazi kada rešite zagonetku, naučite nešto što svi drugi smatraju komplikovanim ili shvatite da znate nešto što Bane „Ja sam stručniji od tebe" iz inženjerskog odeljenja *ne zna*.



Metakognicija: razmišljanje o razmišljanju

Ako zaista želite da učite, i želite da učite brže i dublje, obratite pažnju na način na koji obraćate pažnju. Razmišljajte o načinu na koji razmišljate. Učite o načinu na koji učite.

Pitam se kako bih mogao da pređem mozak pa da ovo zapamti...

Većina nas nije pohađala kurseve o metakogniciji ili teoriji učenja kada smo odrastali. Od nas se *očekivalo* da učimo, ali retko su nas *učili* kako da učimo.



Ako držite ovu knjigu, pretpostavljamo da zaista želite da naučite projektne obrasce. I verovatno ne želite da potrošite mnogo vremena. I želite da *zapamtite* ono što čitate i da to kasnije primenite. I zbog toga morate da *razumete* ono što čitate. Da biste izvukli maksimum iz ove knjige, ili iz *bilo koje* knjige ili procesa učenja, preuzmite odgovornost za svoj mozak. Vaš mozak kada je u pitanju *ovaj* sadržaj.

Stvar je u tome da naterate mozak da novu materiju koju učite posmatra kao Stvarno Važnu. Presudnu za vaše dobrostanje. Jednako bitnu kao što je tigar. Inače, bićete u neprekidnoj borbi u kojoj će vaš mozak davati sve od sebe da ništa od novog sadržaja ne bude sačuvano.

Kako ćete naterati svoj mozak da projektne obrasce smatra jednako bitnim kao što je to tigar?

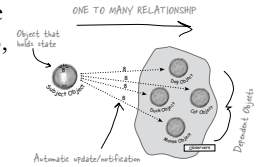
Postoji spor, naporan način, i brži, efikasniji. Spor način je čisto ponavljanje. Očigledno znate da *jeste* sposobni da naučite i zapamtite čak i najdosadnije teme, ako neprekidno bubate istu stvar. Uz dovoljno ponavljanja, vaš mozak će reći, „Ovo mi ne *deluje* bitno, ali on neprekidno gleda jedno te isto *iznova* i *iznova*, pa pretpostavljam da jeste važno."

Brži način je da uradite ***bilo šta što povećava moždanu aktivnost***, a pogotovo razne *vrste* moždane aktivnosti. Stvari sa prethodne strane su su veliki deo rešenja i sve dokazano pomažu mozgu da radi u vašu korist. Na primer, istraživanja pokazuju da smeštanje reči *unutar* slika koje opisuju (umesto negde drugde na strani, recimo u zaglavlju slike ili glavnom tekstu) tera mozak da pokuša da razume odnos reči i slike, a time se aktivira više neurona. Više aktivnih neurona = veća verovatnoća da mozak *shvati* da je to nešto vredno pažnje, a verovatno i pamćenja.

Konverzacijski stil pomaže jer ljudi obraćaju više pažnje kada imaju osećaj da razgovaraju, jer se od njih očekuje da prate razgovor i ravnopravno učestvuju u njemu. Zadivljujuća stvar je da vaš mozak uopšte ne *mari* što se „razgovor" vodi između vas i knjige! S druge strane, ako je stil pisanja formalan i suvoparan, mozak to doživljava isto kao kada vam neko drži predavanje ili kada sedite u sobi punoj pasivnih učesnika. Nema potrebe da ostanete budni.

Ali slike i konverzacijski stil su tek početak.

Evo šta smo MI uradili:



Koristili smo ***slike***, jer je mozak podešen za vizuelno, a ne za tekst. Što se mozga tiče, slika zaista vredi 1.024 reči. A kada slike i tekst rade zajedno, ugradili smo tekst u slike jer je vaš mozak efikasniji kada je tekst unutar stvari o kojoj govori, umesto u zaglavlju ili zakopan negde u tekstu.

Koristili smo ***prekomernost***, govoreći istu stvar na *razne* načine i koristeći razne tipove medijuma i *više čula*, da bismo povećali verovatnoću da se sadržaj ureže u više oblasti vašeg mozga.



Guru za obrasce

Koristili smo koncepte i slike na ***neočekivane*** načine jer vaš mozak je podešen da primećuje novitete, i koristili smo slike i ideje sa bar *malo* ***emotivnog*** *sadržaja*, pošto je vaš mozak podešen da obraća pažnju na biohemiju emocija. Pre ćete zapamtiti ono što vas natera da nešto *osetite*, čak i ako je to samo ***zabava***, ***iznenađenje*** ili ***zainteresovanost***.

Koristili smo personalizovan, ***konverzacijski stil***, jer je vaš mozak podešen tako da obraća više pažnje kada veruje da razgovarate, nego kada misli da pasivno slušate neku prezentaciju. Vaš mozak to čini čak i kada *čitate*.

Dodali smo preko 90 ***aktivnosti***, pošto je vaš mozak podešen tako da više uči i pamti kada nešto ***radite***, nego kada o nečemu čitate. I napravili smo vežbe koje su zahtevne, ali izvodljive, jer se većini *ljudi* to više dopada.



Koristili smo ***više stilova učenja***, jer *vi* možda više volite postupke, dok neko drugi želi prvo da razume širu sliku, a neko treći da vidi primer koda. Međutim, bez obzira na ono što vama više leži, *svi* imaju koristi od prikazivanja istog sadržaja na više načina.

Dodali smo sadržaj za ***obe hemisfere mozga***, jer što se mozak više angažuje, veća je verovatnoća da ćete naučiti i zapamtiti, i duže ćete ostati fokusirani. Pošto naprezanje jedne polovine mozga često znači da druga ima priliku da se odmara, produktivnije ćete učiti tokom dužeg perioda.



Dodali smo i ***priče*** i vežbe koje predstavljaju ***više gledišta***, jer je mozak podešen tako da uči dublje kada je primoran da procenjuje i prosuđuje.

Dodali smo ***izazove*** kroz vežbe i ***pitanja*** koja nemaju uvek pravi odgovor, jer mozak je podešen da uči i pamti kada mora da *radi* na nečemu. Razmislite o tome – ne možete svoje *telo* dovesti u formu *posmatranjem* ljudi u teretani. Dali smo sve od sebe da ono na čemu naporno radite bude *prava* stvar. Da *ne potrošite nijedan suvišan dendrit* obrađujući teško shvatljiv primer, niti tumačeći težak tekst, prepun stručnih izraza, a ni neki preterano sažet.



U pričama, primerima, slikama itd, koristili smo ***ljude***, jer i *vi ste* osoba. A vaš mozak obraća više pažnje na *ljude* nego na *stvari*.

Koristili smo pristup ***80/20***. Ako planirate doktorat iz dizajna softvera, verovatno ovo neće biti jedina knjiga koju ćete pročitati. Zato ne govorimo o *svemu*. Samo o stvarima koje će vam zaista *trebati*.



Evo šta VI možete da uradite da biste pokorili svoj mozak

Mi smo uradili ono što je do nas. Ostalo je na vama. Ovi saveti su polazište; slušajte svoj mozak i zaključite šta kod vas pali, a šta ne. Probajte nove stvari.

1. **Usporite. Što više razumete, to** manje morate da pamtite.

   Nemojte samo da *čitate*. Zastanite i razmislite. Kada vas u knjizi nešto pitamo, nemojte prosto skočiti na odgovor. Zamislite da vam neko postavlja pitanje. Što dublje terate mozak da misli, to je veća verovatnoća da ćete naučiti i zapamtiti.

2. **Radite vežbe. Pravite beleške.**

   Mi smo ih ubacili, ali ako bismo ih radili umesto vas, to bi bilo kao da neko radi fizičke vežbe umesto vas. I nemojte samo *gledati* vežbe. **Koristite olovku**. Ima mnogo dokaza da fizička aktivnost *tokom* učenja može da poboljša učenje.

3. **Čitajte odeljke „Ne postoje glupa** pitanja"

   I to sve. Oni nisu opcioni izdvojeni tekst – ***oni su deo glavnog sadržaja***! Ne preskačite ih.

4. **Neka ovo bude poslednje što ćete pročitati pred spavanje. Ili bar poslednja *zahtevna* stvar.**

   Deo učenja (naročito prebacivanje u dogoročno pamćenje) dešava se *nakon* što odložite knjigu. Mozgu je potrebno vreme za sebe, da još malo obrađuje. Ako mu dodate još nešto tokom perioda obrade, deo onoga što ste upravo naučili biće izgubljen.

5. **Pijte vodu. Mnogo vode.**

   Vaš mozak najbolje radi dok se brčka u tečnosti. Dehidracija (koja se može desiti i pre nego što osetite žeđ) smanjuje kognitivnu funkciju.

6. **Razgovarajte o tome. Naglas.**

   Govor aktivira drugi deo mozga. Ako pokušavate nešto da razumete, ili da povećate verovatnoću da ćete se nečega kasnije setiti, izgovorite to naglas. Ili, još bolje, pokušajte to naglas da objasnite nekom drugom. Učićete brže i možda ćete otkriti ideje za koje niste znali da postoje dok ste čitali.

7. **Slušajte svoj mozak.**

   Obratite pažnju na to da li vam je mozak preopterećen. Ako primetite da čitate površno ili da zaboravljate ono što ste upravo pročitali, vreme je za pauzu. Kada pređete određenu granicu, nećete učiti brže pokušavajući da natrpate više materije, čak biste mogli omesti učenje.

8. ***Osetite*** nešto!

   Vaš mozak mora da zna da je ovo *bitno*. Unesite se u priče. Izmislite svoje potpise za slike. Gunđanje zbog loše šale *bolje* je nego ne osećati baš ništa.

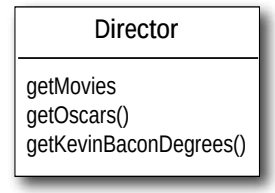
9. *Projektujte* nešto!

   Primenite ovo na nešto novo što projektujete, ili refaktorišite stari projekat. Samo uradite *nešto* da biste stekli iskustvo izvan vežbi i aktivnosti iz ove knjige. Treba vam samo olovka i problem za rešavanje… problem koji bi mogao imati koristi od jednog ili više projektnih obrazaca.

Pročitaj me

Ovo je program učenja, a ne enciklopedija. Namerno smo uklonili sve što bi moglo smetati učenju onoga na čemu radimo u određenom delu knjige. U prvom prolazu treba da krenete od početka, jer knjiga pretpostavlja da ste nešto već videli i naučili.

Koristimo jednostavniju, izmenjenu verziju UML-a.



Koristimo jednostavne dijagrame slične objedinjenom jeziku za modelovanje.

Iako ste verovatno već naišli na objedinjeni jezik za modelovanje (UML), on nije objašnjen u ovoj knjizi i nije preduslov za njeno čitanje. Ako nikada pre niste videli UML, ne brinite, daćemo vam nekoliko smernica usput. Drugim rečima, ne morate istovremeno da brinete i o projektnim obrascima i o UML-u. Neši dijagrami „*liče na* UML" – mada se trudimo da budemo verni UML-u, povremeno malo zaobilazimo pravila, obično iz naših ličnih umetničkih razloga.

Ne objašnjavamo sve projektne obrasce koji su ikada napravljeni.

Postoji *mnogo* projektnih obrazaca: originalni osnovni obrasci (poznati kao GoF obrasci), Javini obrasci za preduzeća, arhitektonski obrasci, obrasci za dizajn igara i mnogi drugi. Naš cilj je bio da ova knjiga teži manje od osobe koja je čita, pa ih ne opisujemo sve. Fokusirali smo se na *bitne* osnovne obrasce iz originalnih objektno orijentisanih GoF obrazaca, i pobrinuli se da stvarno, istinski duboko razumete kako i kada da ih kroistite. Pronaći ćete kratak pregled nekih drugih obrazaca (onih koje ćete verovatno daleko manje koristiti) u Dodatku. U svakom slučaju, kada završite *Um caruje: projektni obrasci*, moći ćete da uzmete bilo koji katalog obrazaca i vrlo brzo se uhodate.

Aktivnosti NISU opcione.

Vežbe i aktivnosti nisu dodaci; one su deo glavnog sadržaja knjige. Neke služe da vam pomognu u pamćenju, neke u razumevanju, a neke u primeni naučenog. ***Ne preskačite vežbe***. Ukrštenice su jedina stvar koju ne *morate* da radite, ali dobra su prilika da vaš mozak razmisli o rečima u drugačijem konkekstu.

Reč „kompozicija" koristimo u opštem OO smislu, koji je fleksibilniji od striktne upotrebe „kompozicije" u UML-u.

Kada kažemo „kompozicija dva objekta" mislimo da između njih postoji odnos IMA (eng. HAS-A). Naša upotreba odražava tradicionalnu upotrebu tog pojma i ona se koristi u tekstu GoF-a (kasnije ćete naučiti šta je to). Nedavno je UML preradio ovaj pojam u nekoliko tipova kompozicije. Ako ste stručnjak za UML, i dalje ćete moći da čitate ovu knjigu i trebalo bi da lako možete da uparite upotrebu kompozicije sa doteranijim pojmovima dok budete čitali.

Prekomernost je namerna i bitna.

Upečatljiva razlika u knjigama Um caruje jeste to što mi želimo da vi *zaista* shvatite. I želimo da kada završite knjigu zapamtite ono što ste naučili. Većina enciklopedijskih knjiga nema za cilj zadržavanje znanja i prisećanje, ali ova knjiga se bavi *učenjem*, pa će se neki koncepti pojavljivati više puta.

Primeri koda su kratki koliko god je to moguće.

Čitaoci su nam rekli da ih nervira bauljanje kroz 200 redova koda u potrazi za dva reda koje treba da razumeju. Većina primera u ovoj knjizi data je u najmanjem mogućem kontekstu da bi deo koji učite bio jasan i jednostavan. Ne očekujte robustan, pa čak ni kompletan kod – primeri su pisani baš za učenje i nisu uvek potpuno funkcionalni.

U nekim situacijama nismo uključili sve bitne i potrebne naredbe, ali pretpostavljamo, na primer, da ćete kao programer na Javi znati da je `ArrayList` u java.util. Ako naredbe import nisu deo običnog osnovnog JSE API-ja, pomenućemo to. Sav izvorni kod smo postavili i na veb, pa ga možete preuzeti. Pronaći ćete ga na lokaciji `http://wickedlysmart.com/head-first-design-patterns`.

Pored toga, zarad fokusiranja na učenje kroz kod, nismo stavili naše klase u pakete (drugim rečima, sve su u Javinom neimenovanom paketu). Ne proporučujemo da tako radite u stvarnosti, i kada budete pruzeli primere koda iz ove knjige, videćete da klase *jesu* u paketima.

Vežbe Snaga uma nemaju odgovore.

Za neke od njih nema pravog odgovora, a u drugima je deo iskustva učenja da sami zaključite jesu li vaši odgovori tačni. U nekim vežbama Snaga uma pronaći ćete smernice koje vas upućuju na pravu stranu.

Recenzenti

Valentin Crettaz



Jef Cumps



Barney Marispini



Ike Van Atta





Neustrašivi vođa ekstremnog tima recenzenata UCPO

Johannes deJong

U sećanje na Filipa Makea, 1960 – 2004. Tvoja zadivljujuća stručnost, neizmerni entuzijazam i iskrena briga za učenike zauvek će nas inspirisati.

Jason Menard





Dirk Schreckmann

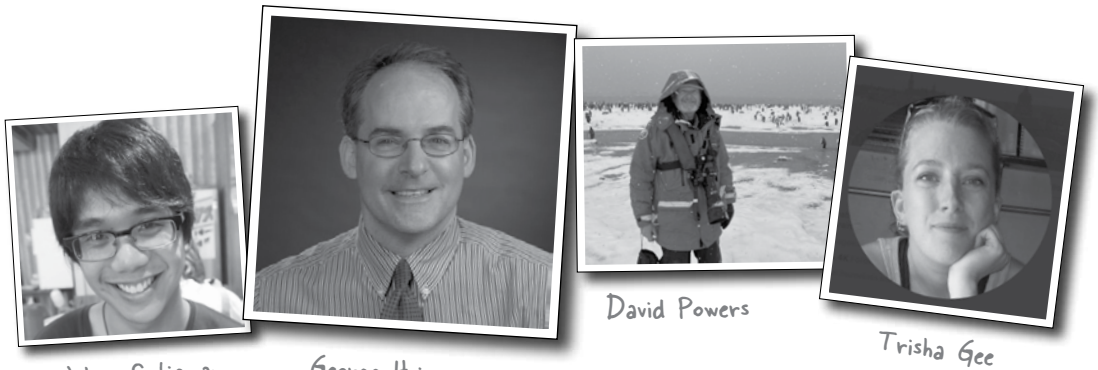


Philippe Maquet



Mark Spritzler

Recenzenti, drugo izdanje



Zahvalnice

Iz prvog izdanja

U kompaniji O'Reilly:

Najveće hvala **Majku Loukidesu** iz kompanije O'Reilly, jer je sve pokrenuo i pomogao u oblikovanju koncepta Um caruje u ediciju. I veliko hvala pokretačkoj sili iz edicije Um caruje, **Timu O'Rajliju**. Hvala mudroj „mami" edicije Um caruje, **Kajli Hart**, „kralju In Designa" **Ronu Bilodou**, rokenrol zvezdi **Eli Volkhauzen** za inspirisan dizajn korica, **Melani Jarbrug** za vođenje produkcije, **Kolin Gorman** i **Rejčel Monagan** za tvrdokornu korekturu i **Bobu Faleru** za znatno poboljšan indeks. Najzad, hvala **Majku Hendriksonu** i **Megan Blanšet** za podršku ovoj knjizi i izgradnju tima.

Našim neustrašivim recenzentima:

Izuzetno smo zahvalni direktoru tehničke recenzije **Johanesu deJongu**. Ti si naš junak, Johanese. Duboko cenimo doprinos pomoćnika menadžera recenzentskog tima **Javaranch**, pokojnog **Filipa Makea**. Sam si uneo svetlost u živote hiljada programera, i uticaj koji si imao na njihove (i naše) živote trajaće večno. **Džef Kamps** je zastrašujuće dobar u pronalaženju problema u našim nacrtima poglavlja, i opet je značajno uticao na knjigu. Hvala ti Džefe! **Valentin Kretas** (tip za AOP), koji je sa nama od prve knjige Um caruje, dokazao je (kao i uvek) koliko nam je neophodna njegova tehnička stručnost i uvid. Ti si car, Valentine (samo skini kravatu).

Dve pridošlice u recenzentskom timu UC, **Barni Marispini** i **Ajk Van Ata**, obavili su đavolski dobar posao na knjizi – momci, vi ste nam dali neke *zaista* krucijalne povratne informacije. Hvala što ste se pridružili timu.

Imamo i odličnu tehničku pomoć moderatora/gurua iz Javarancha **Marka Spriclera**, **Džejsona Menarda**, **Dirka Šrekmana**, **Thomasa Pola** i **Margarite Isaeve**. I, kao i uvek, posebno hvala supervizoru javaranch.com, **Polu Vitonu**.

Hvala finalistima Javaranchovog takmičenja „Izaberite korice za *Um caruje: projektni obrasci*". Pobednik, Si Bruster, poslao je pobednički esej koji nas je ubedio da izaberemo ženu koju vidite na koricama. U ostale finaliste spadaju Endru Es, Đan Franko Kasula, Helen Krozbi, Fo Tek, Helen Tomas, Sateš Komineni i Džef Fišer.

Za ažurirano izdanje knjige iz 2014, zahvalni smo i sledećim tehničkim recenzentima: Džordžu Hoferu, Tedu Hilu, Todu Bartožkievicu, Silvanu Tenijeu, Skotu Dejvidsonu, Kevinu Rajanu, Riču Vordu, Marku Fransisu Jegeru, Marku Maseu, Glenu Reju, Bajardu Fetleru, Polu Higinsu, Metu Karpenteru, Džuliji Vilijams, Metu Makalou i Meri En Belarmino.

Zahvalnice

Iz drugog izdanja

U kompaniji O'Reilly:

Pre svega, **Meri Treseler** je supermoć koja sve omogućava i večno ćemo joj biti zahvalni za sve što radi za kompaniju O'Reilly, Um caruje i autore. **Melisa Dafild** i **Mišel Kronin** su raskrčile mnoge staze koje su omogućile ovo drugo izdanje. **Rejčel Monagan** je uradila zadivljujuću korekturu, dajući našem tekstu nov sjaj. **Kristen Braun** je učinila da knjiga bude prelepa na internetu i u štampi. **Eli Volkhauzen** je upotrebila svoju magiju i dizajnirala briljantne nove korice za drugo izdanje. Hvala svima!

Recenzenti drugog izdanja:

Zahvalni smo tehničkim recenzentima drugog izdanja što su nastavili sa zadatkom 15 godina kasnije. **Dejvid Pauers** je naš odabrani recenzent (on je naš, nemojte ni pomišljati da mu tražite da pregleda *vašu* knjigu) jer mu ništa ne promiče. **Džordž Hajneman** je premašio očekivanja sa svojim detaljnim komentarima, predlozima i povratnim informacijama, i zaslužio je nagradu za najvrednijeg tehničkog igrača za ovo izdanje. **Triša Dži** i **Džulijan Setiavan** obezbedili su neprocenjivo poznavanje Jave koje nam je trebalo da bismo izbegli greške kojih bismo se kasnije stideli. Hvala svima!

Posebna zahvalnost

Veoma posebno hvala **Eriku Gami**, koji je prevazišao svoje dužnosti u recenziranju ove knjige (čak je poneo nacrt sa sobom na odmor). Eriče, tvoje interesovanje za ovu knjigu nas je inspirisalo, a tvoja temeljna tehnička recenzija ju je neizmerno poboljšala. Hvala i celoj **Četvoročlanoj bandi** za podršku i interesovanje, i jer su gostovali u Objektgradu. Mnogo su nas zadužili i **Vord Kaningam** i zajednica korisnika obrazaca koji su napravili Portland Pattern Repository – nezamenjiv izvor tokom pisanja ove knjige.

Veliko hvala **Majku Lukidesu**, **Majku Hendriksonu** i **Megan Blanšet**. Majk L. je bio s nama celim putem. Majk, tvoje pronicljive povratne informacije pomogle su nam da oblikujemo knjigu, a tvoje ohrabrenje nas je podsticalo da nastavimo dalje. Majk H., hvala za upornost tokom pet godina koliko si nas nagovarao da napišemo knjigu o obrascima; najzad smo to učinili i drago nam je što smo čekali Um caruje.

Potrebno je celo selo da se napiše stručna knjiga: **Bil Pag** i **Ken Arnold** su nam dali stručne savete o jedinstvenoj instanci. **Džošua Marinači** je obezbedio sjajne savete o Swingu. Rad **Džona Bruera** „Why a Duck?" (Zašto patka?) inspirisao je SimUDuck (i drago nam je da on voli patke). **Den Fridman** je inspirisao primer sa malom jedinstvenom instancom. **Daniel Stajnberg** je služio kao naša „tehnička veza" i mreža emocionalne podrške. Hvala **Džejmsu Dempsiju** iz Applea jer nam je dozvolio da upotrebimo njegovu pesmu MVC. I hvala **Ričardu Varburtonu** koji se pobrinuo da naš kod na Javi 8 bude prihvatljiv za ažurirano izdanje knjige.

Najzad, lično hvala **recenzentskom timu Javarancha** za vrhunske recenzije i toplu podršku. U ovoj knjizi vas ima više nego što znate.

Pisanje knjige Um caruje je divlja vožnja sa naša dva zadivljujuća vodiča: **Keti Sierom** i **Bertom Bejtsom**. Sa Keti i Bertom odbacujete sve konvencije za pisanje knjiga i ulazite u svet pun pripovedanja, teorije učenja, kognitivne nauke i pop kulture, gde čitalac uvek vlada.

1 uvod u projektne obrasce

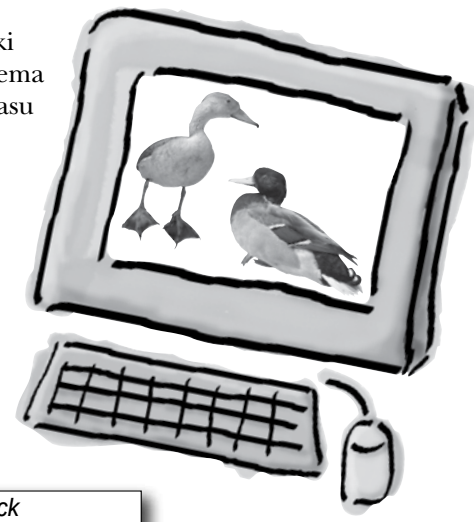
Dobro došli u projektne obrasce

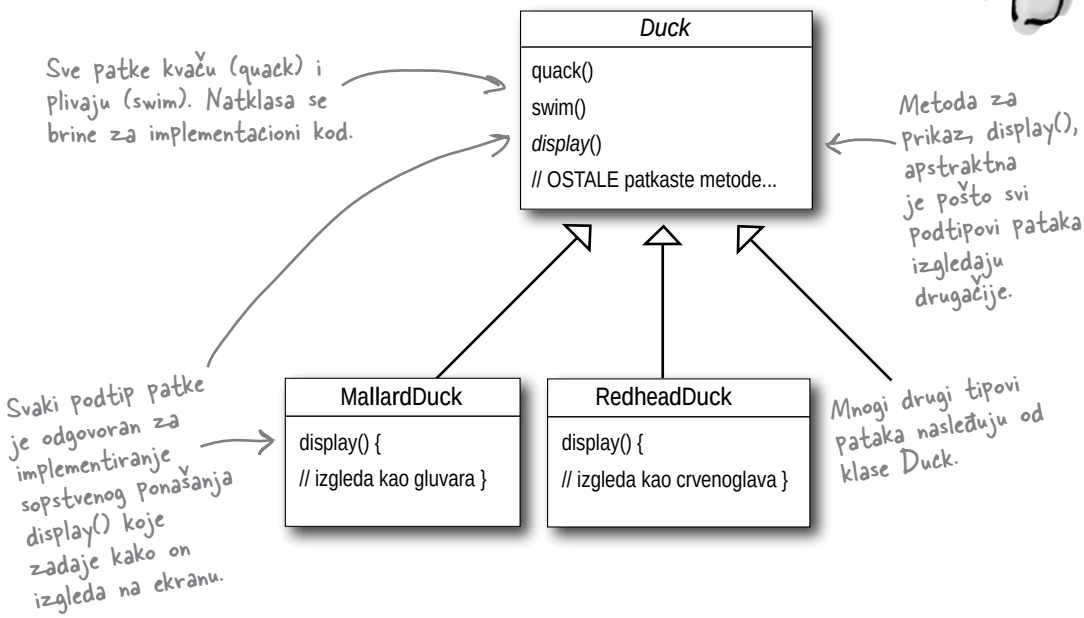


Neko je već rešio vaše probleme. U ovom poglavlju ćete naučiti zašto (i kako) da koristite mudrosti i lekcije do kojih su došli drugi programeri koji su išli istim putem rešavanja problema pri projektovanju i preživeli. Pre nego što završimo, razmotrićemo upotrebu i koristi od projektnih obrazaca, pogledati 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.





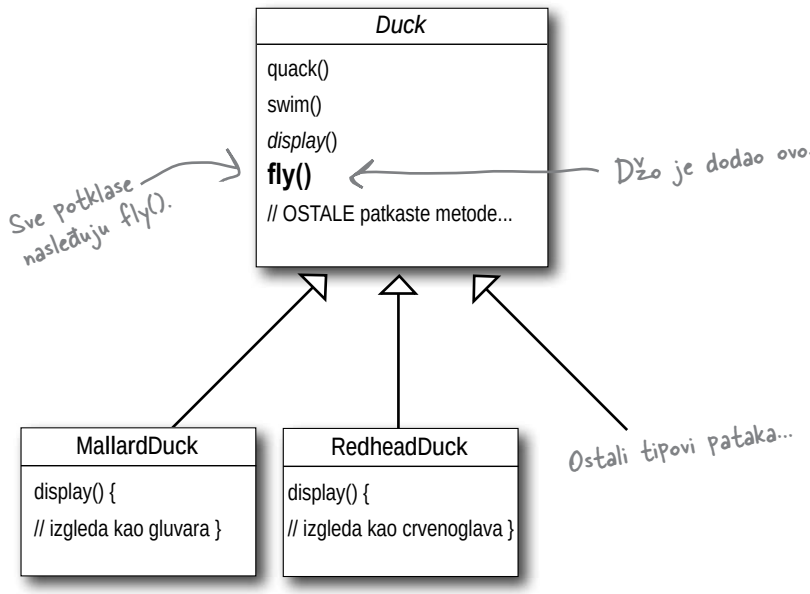
Tokom poslednje godine, kompanija je bila pod sve većim pritiskom konkurencije. Nakon sedmodnevne sesije mozganja uz izgranje golfa, direktori kompanije smatraju da je vreme za veliku inovaciju. Treba im nešto *zaista* impresivno što će pokazati na predstojećem sastanku deoničara na Mauiju *naredne sedmice*.

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

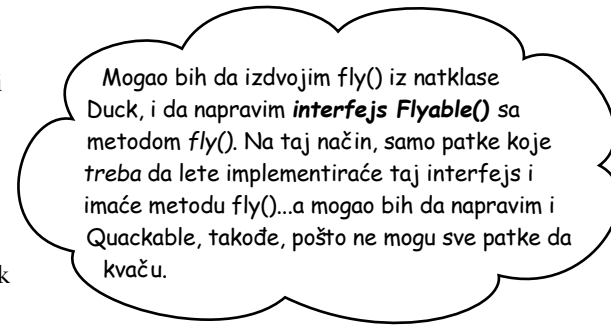
Šta od narednog čini *nasleđivanje* neodgovarajućim za definisanje ponašanja patke? (Odaberite sve što važi.)

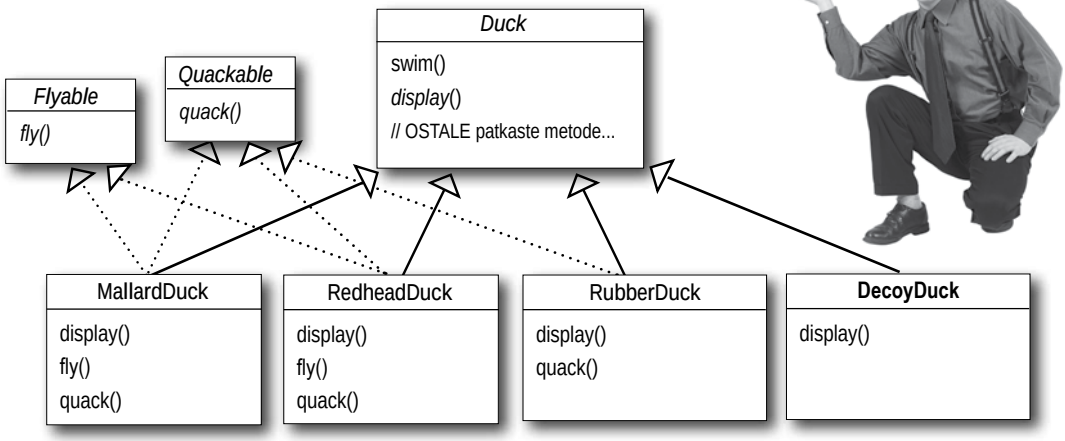
- ❑ A. Kod se duplicira u potklasama.
- ❑ B. Izmene ponašanja tokom izvršavanja su teške.
- ❑ C. Ne možemo zadati da patke igraju.
- ❑ D. Teško je saznati sva ponašanja pataka.
- ❑ E. Patke ne mogu istovremeno da lete i kvaču.
- ❑ F. Izmene mogu nenamerno da utiču na druge patke.

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!

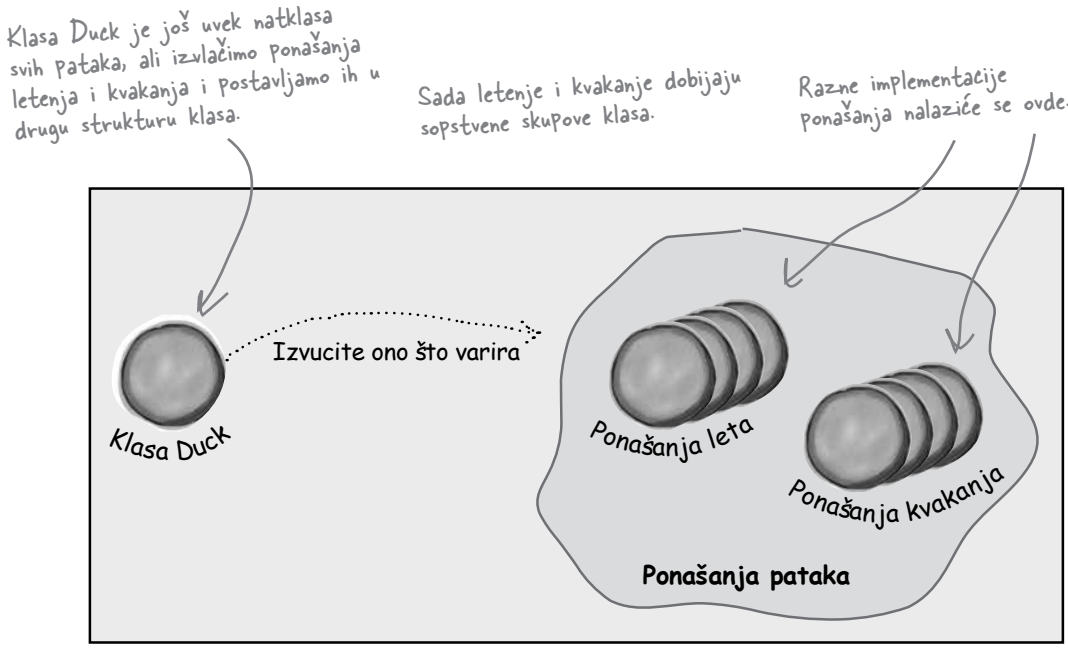
Odvajanje onoga što se menja od onoga što ostaje isto

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.

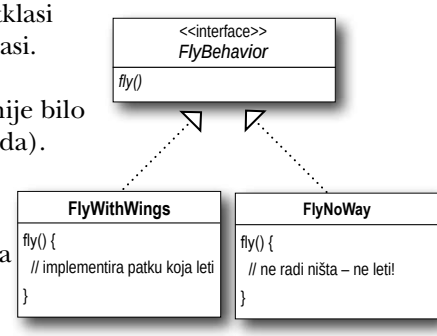
Tako ovaj put neće klase *Duck* implementirati interfejse leta i kvakanja. Umesto toga, napravićemo skup klasa čiji je čitav smisao postojanja da predstavljaju neko ponašanje (na primer, „skvičanje“), i onda će klasa *ponašanja*, a ne klasa Duck, implementirati interfejs ponašanja.

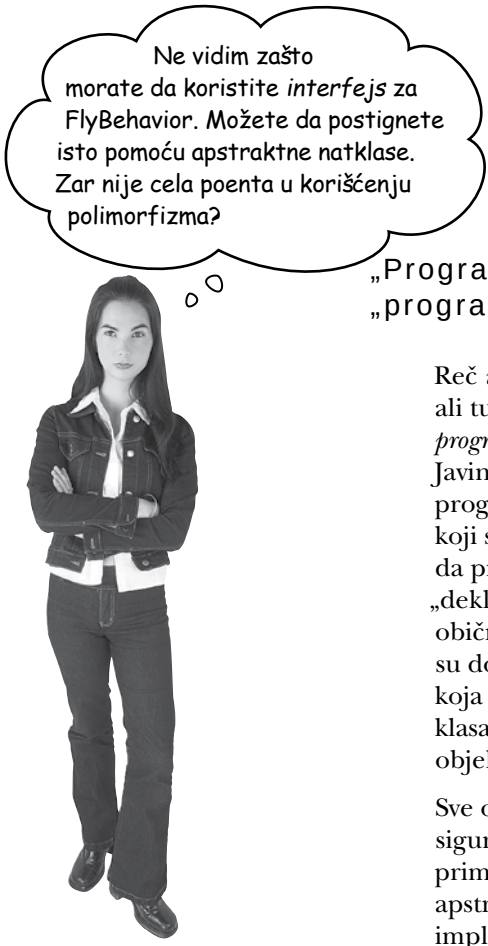
Ovo je suprotno načinu na koji smo ranije radili stvari, kada je ponašanje poticalo ili iz konkretne implementacije u natklasi Duck, ili iz specijalizovane implementacije u samoj potklasi. U oba slučaja smo se oslanjali na *implementaciju*. Bili smo ograničeni da koristimo te konkretne implementacije 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.

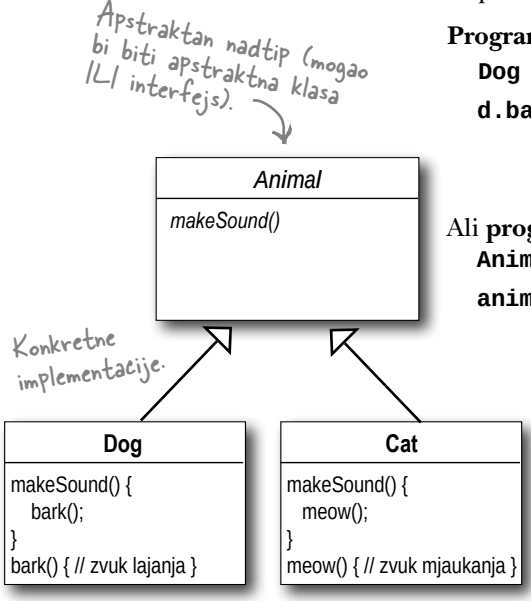




„Programirajte u *interfejs*“ zapravo znači „programirajte *u nadtip*“.

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.

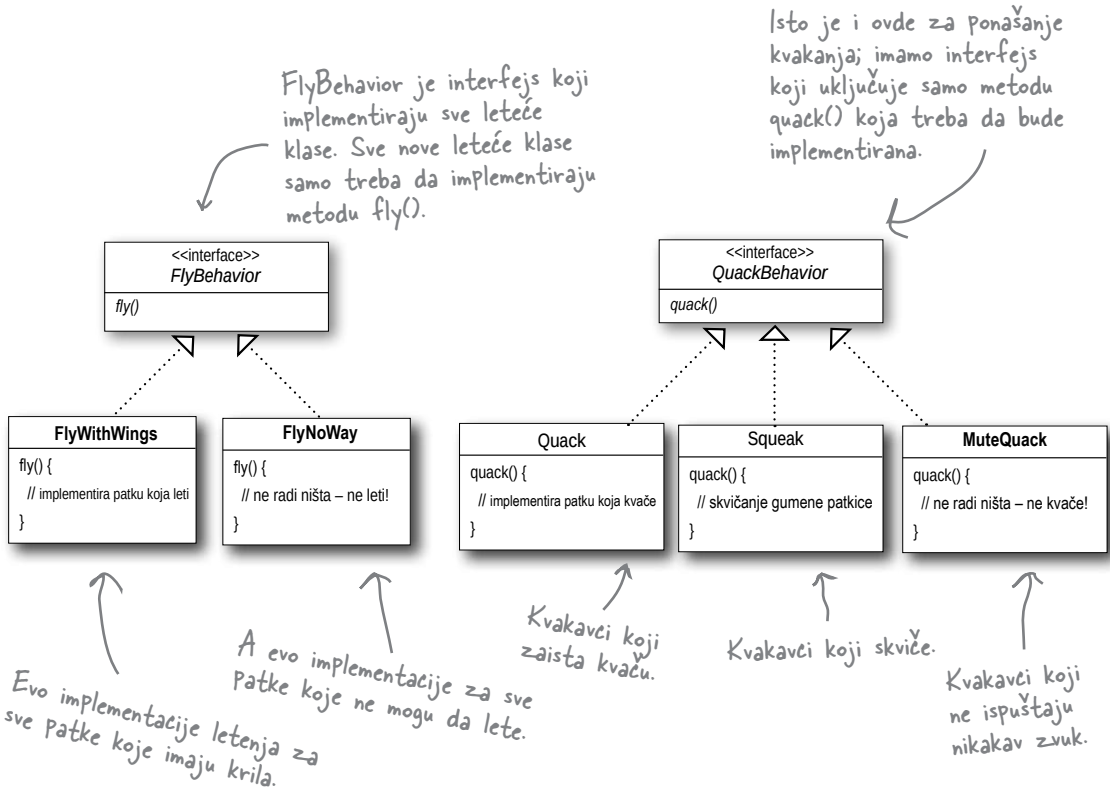
Još bolje bi bilo da, umesto da u kod upišemo instanciranje podtipa (kao što je new Dog()), **dodelimo konkretan implementacioni objekat pri izvršavanju:**

```
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

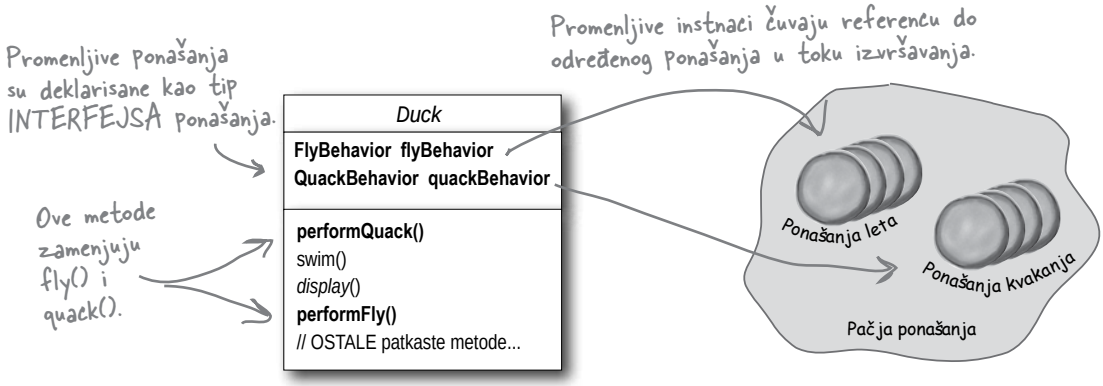
Evo ključa: Duck će sada delegirati svoja ponašanja leta i kvakanja, umesto da koristi metode kvakanja i **leta definisane u klasi Duck (ili potklasi).**

Evo kako:

❶ **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.



❷ **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!!
```

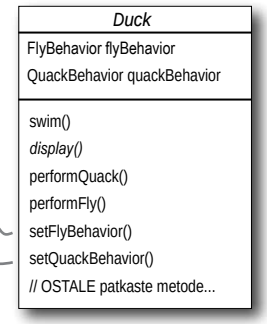
Dinamičko zadavanje ponašanja

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.

1 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

2 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.

3 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

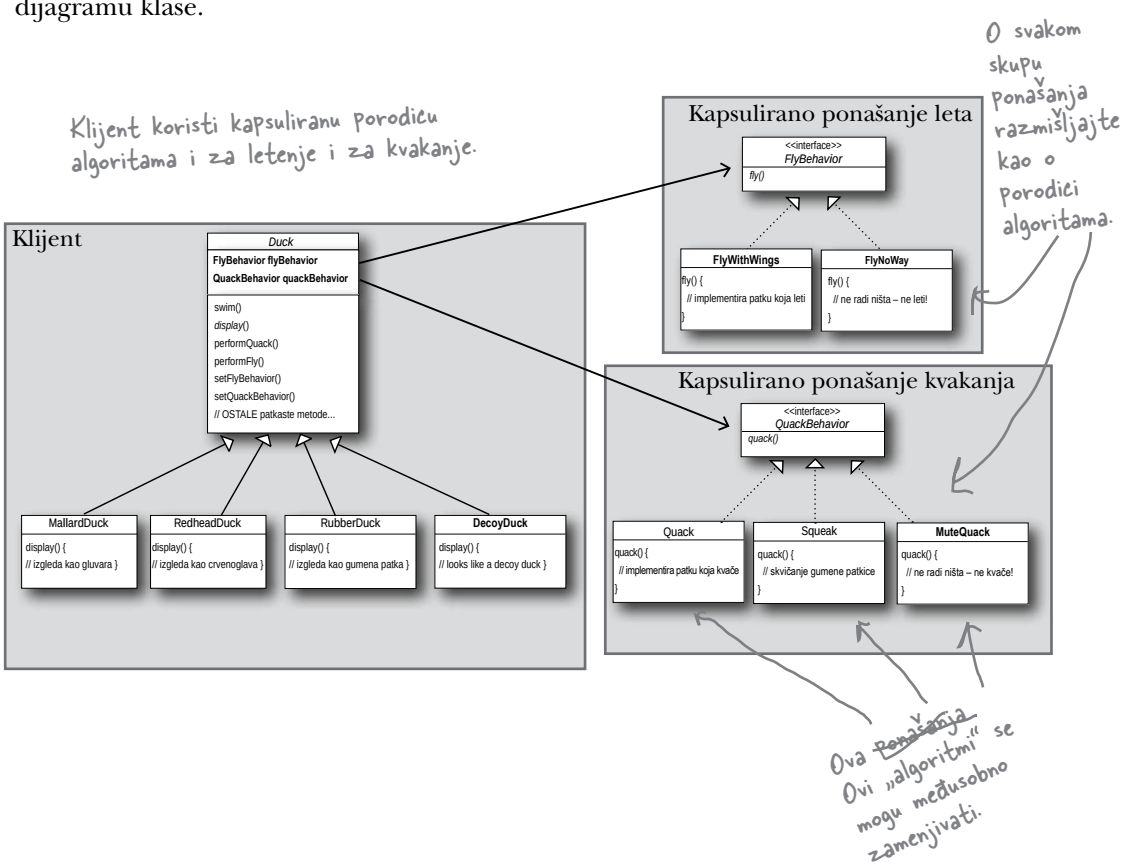
Pošto smo već duboko zaronili u projekat simulatora patke, vreme je da izronimo da **udahnemo vazduh i osmotrimo širu sliku.**

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.



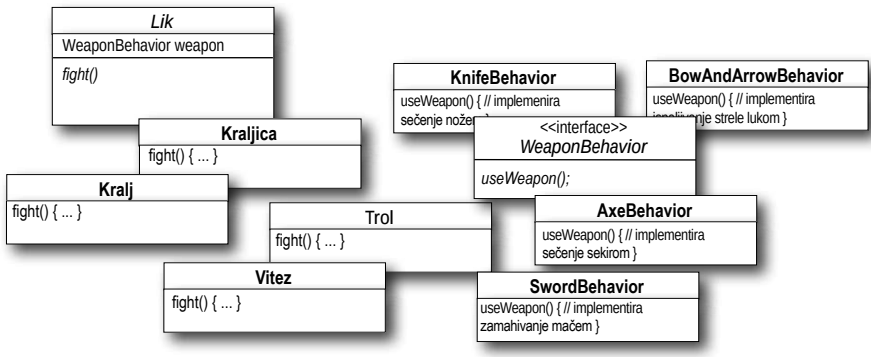
Projektna zagonetka

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.

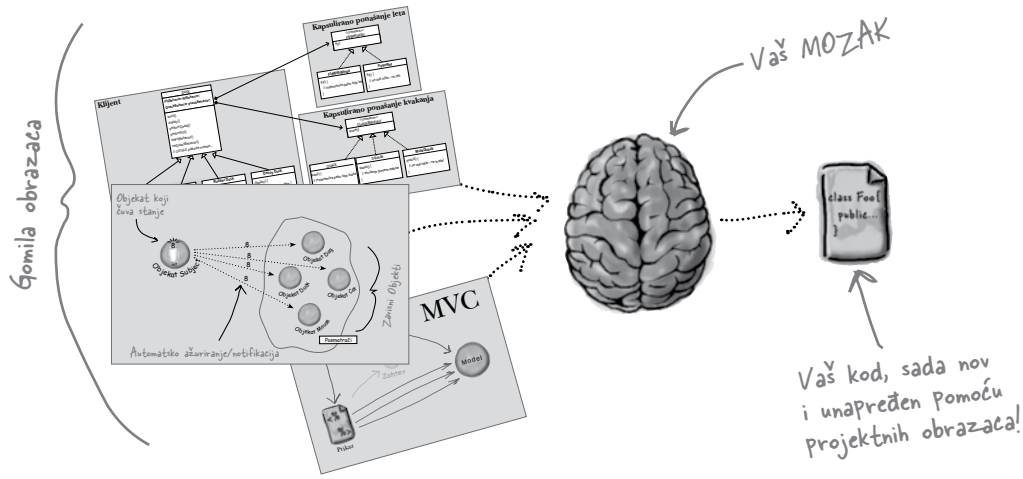
Zajednički rečnici ohrabruju mlađe programere da se ubrzaju. Mlađi programeri se ugledaju na iskusne projektante. Kada stariji projektanti koriste projektne obrasce, i mlađi su motivisani da ih nauče. Izgradite zajednicu korisnika obrazaca u svojoj organizaciji.

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

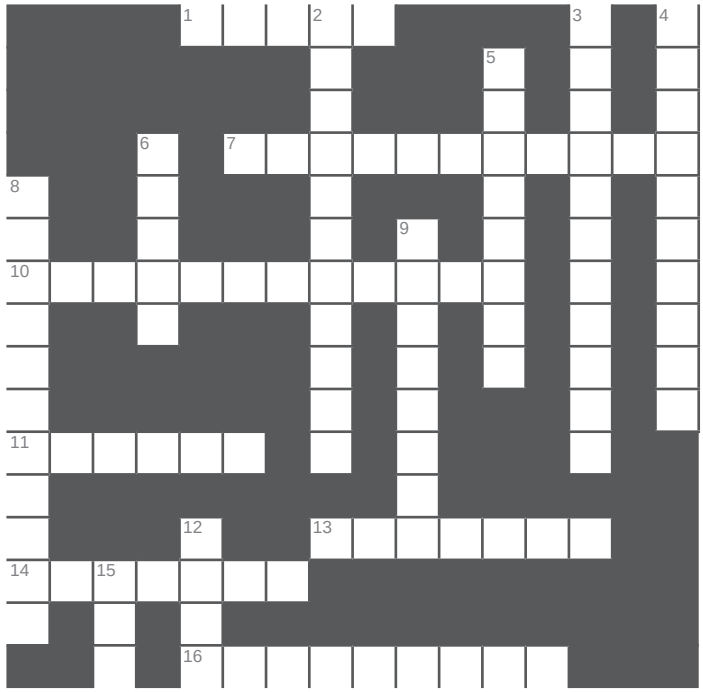
- Poznavanje OO osnova vas ne čini dobrim OO programerom.
- Dobar OO dizajn je višekratan, proširiv i održiv.
- Obrasci vam pokazuju kako da izgradite sisteme sa dobrim kvalitetima OO dizajna.
- Obrasci su dokazano objektno orijentisano iskustvo.
- Obrasci vam ne daju kod već opšta rešenja za projektne 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

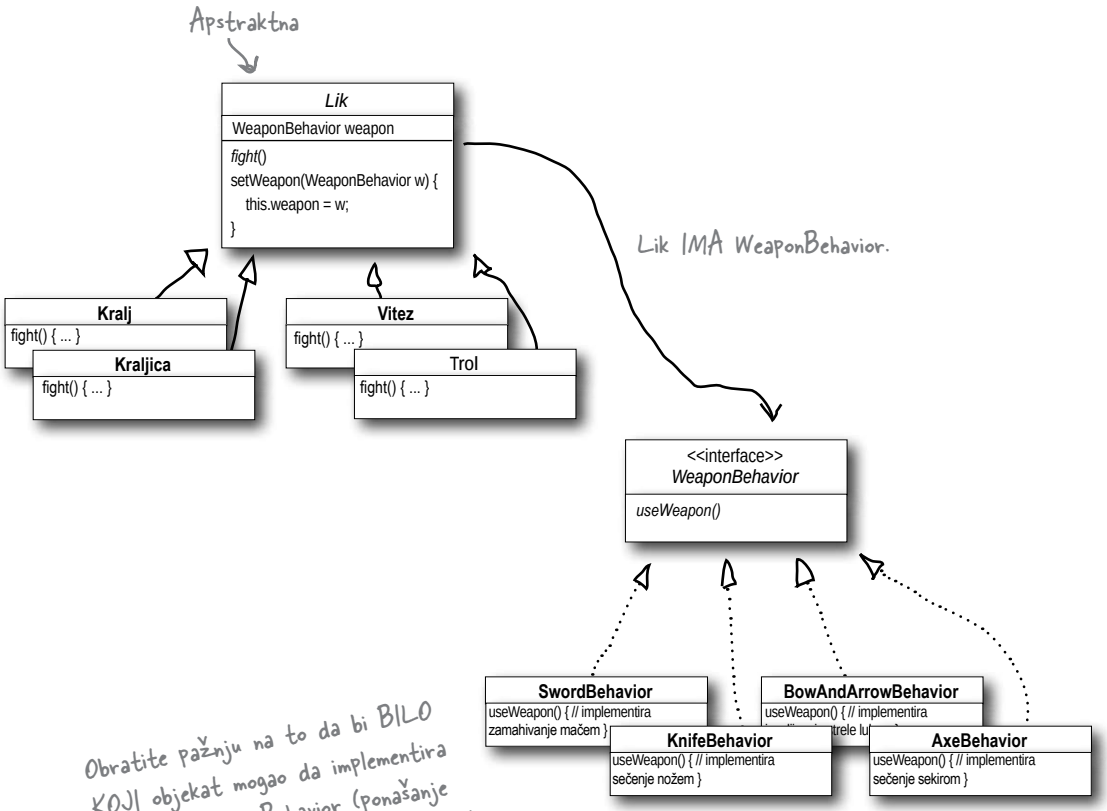
2. Dajte prednost ovome nad nasleđivanjem.
3. Desert koji mrzovoljni kuvar naziva brod.
4. Strategije mogu biti _____.
5. Većina obrazaca proističe iz OO _____.
6. Obrasci ulaze u vaš _____.
8. Obrazac kojim je Rik bio oduševljen.
9. U učenju koristite tuđe _____.
12. Ostrvo na kom je prikazan demo s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

