Uvod

Jedina valjana mera kvaliteta softvera je: Šta je ovo/zapisnik



(c) 2008 Focus Shift

Reprodukovano sa ljubaznom dozvolom Thoma Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Koja vrata predstavljaju vaš kod? Koja vrata predstavljaju vaš tim ili vašu kompaniju? Zašto smo u toj sobi? Da li je ovo samo uobičajena provera koda ili smo pronašli niz užasnih problema ubrzo pošto ste kod pustili u primenu? Otklanjamo li greške u panici, opsednuti kodom za koji smo mislili da radi? Da li nas klijenti masovno napuštaju, a

menadžeri nam dišu za vrat? Kako da budemo sigurni da ćemo biti iza pravih vrata kad stvari postanu teške? Odgovor je: zanat.

Postoje dva dela za učenje zanata: znanje i rad. Morate steći znanje o principima, obrascima, praksama i heuristikama koje zanatlija poznaje, a to znanje morate usaditi i u svoje prste, oči i stomak radeći naporno i vežbajući.

Mogu da vas naučim fiziku vožnje bicikla. Zaista, klasična matematika je relativno jednostavna. Gravitacija, trenje, ugaoni moment, centar mase i tako dalje mogu se prikazati sa manje od jedne stranice jednačina. Tim formulama mogao bih da vam dokažem da je vožnja biciklom praktična i pružio bih vam svo znanje koje vam je potrebno da biste ga vozili. Ali, ipak kada biste prvi put seli na bicikl, odmah biste pali.

Pisanje koda se ne razlikuje od vožnje bicikla. Mogli bismo da zapišemo sve dobre principe jasnog i čistog koda i onda da vam poverimo da programirate (drugim rečima, neka padnete kada sednete na bicikl), ali kakvi bismo onda mi bili nastavnici kada bismo tako učinili, i kakvi biste vi bili učenici?

Ne. To nije način na koji će vas ova knjiga učiti.

Učenje pisanja jasnog koda je *naporan posao*. To zahteva više od znanja o principima i obrascima. Morate da se *preznojite* nad tim. Morate sami da vežbate i gledate kako ne uspevate. Morate da gledate kako drugi to rade i ne uspevaju. Morate da vidite kako se spotiču i vraćaju nazad. Morate da vidite njihovu muku u donošenju odluka i cenu koju plaćaju za donošenje pogrešnih odluka.

Budite spremni na naporan rad dok čitate ovu knjigu. Ova knjiga nije lako štivo koje možete pročitati u avionu pre nego što sleti. Ova knjiga će vas naterati da radite *i naporno radite*. Kakvim se poslom bavite? Čitaćete kod – mnogo koda. I biće vam izazov da razmišljate šta je tačno u kodu a šta nije. Od vas će se tražiti da nas pratite dok razdvajamo module i ponovo ih sastavljamo. Za ovo će trebati vremena i truda; ali mislimo da će biti vredno toga.

Knjigu smo podelili na tri dela. U prvih nekoliko poglavlja opisani su principi, obrasci i prakse pisanja jasnog koda. U ovim poglavljima postoji prilično malo koda, pa će biti izazov pročitati ih. Pripremiće vas za drugi deo. Ako ste odložili knjigu nakon što ste pročitali prvi deo, neka vam je sa srećom!

Drugi deo knjige je teži posao. Sastoji se od nekoliko studija slučaja stalno rastuće složenosti. Svaka studija slučaja je vežba čišćenja nekog koda – transformacije koda koji ima nekih problema – u kod koji ima manje problema. Detalji u ovom delu su *intenzivni*. Moraćete da prelazite između narativa i listinga programa. Moraćete da analizirate i razumete kod sa kojim radimo i da razumete naša obrazloženja za svaku promenu koju učinimo. Odvojte malo vremena jer *ovo bi moglo da potraje.*

Treći deo ove knjige predstavlja naplatu. To je jedno poglavlje koje sadrži spisak heuristike i osećaja prikupljenih pri izradi studija slučaja. Dok smo prolazili i čistili kod u studijama slučaja, dokumentovali smo svaki razlog za naše postupke za heuristiku ili za osećaj. Pokušali smo da razumemo sopstvene reakcije na kôd koji smo čitali i menjali i naporno smo radili da shvatimo zašto osećamo šta osećamo i zašto radimo to što radimo. Rezultat je baza znanja koja opisuje način na koji razmišljamo kada pišemo, čitamo i čistimo kod.

Ova baza znanja ima ograničenu vrednost ako ne radite pažljivo čitajući studije slučaja u drugom delu ove knjige. U tim studijama slučaja pažljivo smo označili svaku promenu koju smo izvršili uz upućivanje na heuristiku. Ove naredne reference su prikazane u uglastim zagradama, kao što je ovo: [H22]. Ovo vam omogućava da vidite *kontekst* u kojem su te heuristike primenjene i napisane! Nisu heuristike same po sebi toliko vredne, to je *odnos između tih heuristika i diskretnih odluka koje smo doneli dok smo čistili kod u studijama slučaja.*

Da bismo vam dodatno pomogli u tim odnosima, na kraju knjige smo dali referencu koja upućuje na broj stranice za svaku narednu referencu. Možete da ih upotrebite da potražite gde je primenjena određena heuristika.

Ako pročitate prvi i treći deo a preskočite studije slučaja, tada ćete pročitati još jedno lako štivo o pisanju dobrog softvera. Ali ako odvojite vreme da proučite studije slučaja, prateći svaki sitan korak, svaku pravovremenu odluku, ako se postavite na naše mesto i naterate se da razmišljate na isti način kao i mi, tada ćete steći mnogo bogatije razumevanje tih principa, obrazaca, praksi i heuristika. Sve to više neće biti površno znanje. Biće vam u telu, prstima i srcu. Sve to će postati deo vas na isti način na koji bicikl postaje produžetak vaše volje kada naučite da ga vozite.

Zahvalnosti

Zahvaljujem se mojim umetnicima, Jeniffer Kohnke i Angeli Brooks. Jennifer je odgovorna za kreativne slike na početku svakog poglavlja, kao i za portrete Kent Becka, Vard Cunninghama, Bjarne Stroustrupa, Ron Jeffriesa, Grady Boocha, Dave Thomasa, Michael Feathersa i mene.

Angela je odgovorna za inspirativne slike koje krase unutrašnjost svakog poglavlja. Tokom godina napravila je za mene priličan broj slika, uključujući mnoge slike u *Agile Software Develpment: Principles, Patterns, and Practices*. Ona je takođe moje prvo dete sa kojim sam veoma zadovoljan.

Posebnu zahvalnost upućujem mojim recenzentima Bobu Bogettiju, Georgeu Bullocku, Jeffreyu Overbeiu, a posebno Mattu Heusseru. Bili su brutalni. Bili su okrutni. Oni su bili neumoljivi. Snažno su me gurali da napravim neophodna poboljšanja.

Zahvaljujem svom izdavaču, Chris Guzikowskom, na podršci, ohrabrenju i radosnom izražavanju. Hvala i redakciji u Pearsonu, uključujući Rainu Chrobak što me je tretirala iskreno i tačno.

Hvala Micah Martinu i svim momcima iz 8th Lights (www.8thlight.com) na proverama i ohrabrenju.

Hvala svim mentorima, prošlim, sadašnjim i budućim, uključujući: Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottinger, Joleen Craig, Janice Brown, Susan Rosso i ostalima.

Hvala Jimu Newkirku, mom prijatelju i poslovnom partneru, koji me je naučio više nego što je toga svestan. Hvala Kent Becku, Martin Fowleru, Ward Cunninghamu,

Bjarnr Stroustrupu, Grady Boochu, i svim drugim mojim mentorima i sunarodnicima. Hvala John Vlissidesu što je bio sa nama kada je trebalo. Zahvaljujem momcima u Zebri što su mi dozvolili da pričam o tome koliko duga treba da bude funkcija.

I, konačno, hvala vama što ste pročitali ove zahvalnice.

Sadržaj

Predgovor xv

Uvod xxi

Na korici xxv

Poglavlje 1: Jasan kod 1
- Neka bude kod 2
- Loš kod 2
- Ukupan trošak nereda 3
  - Veliko redizajnirnje na pomolu 4
  - Stav 5
  - Primalna zagonetka 5
  - Umetnost jasnog koda 6
  - Šta je jasan kod? 6
- Škole mišljenja 11
- Mi smo autori 12
- Pravilo izviđača 13
- Prethodište i principi 13
- Zaključak 14
- Bibliografija 14

Poglavlje 2: Smislena imena 15
- Uvod 15
- Koristite imena koja otkrivaju nameru 16
- Izbegavajte dezinformacije 17
- Pravite smislene razlike 18
- Koristite izgovorljive nazive 19
- Koristite pretraživa imena 20

Izbegavajte kodiranje....21
Mađarska notacija....21
Prefiksi....22
Interfejsi i implementacije....22
Izbegavajte mentalno mapiranje....22
Imena klase....23
Nazivi metoda....23
Ne budite slatki....23
Izaberite jednu reč po konceptu....24
Izbegavajte igru reči....24
Koristite imena domena koja se odnose na rešenje....25
Koristite imena domena koja se odnose na problem koji rešavate....25
Dodajte smisleno značenje....25
Nemojte dodavati nepotrebno značenje....28
Završne reči....28

Poglavlje 3: Funkcije....29
Mala!....32
Blokovi i uvlačenja....33
Činite jednu stvar....33
Sekcije u okviru funkcija....34
Jedan nivo apstrakcije po funkciji....34
Čitanje koda od vrha ka dnu: *Pravilo silaska*....35
Switch naredbe....35
Koristite opisna imena....37
Argumenti funkcije....38
Uobičajeni jednoargumentni oblici....39
Argumenti indikatori....39
Dvoargumentne fukcije....40
Troargumentne funkcije....40
Argumenti objekti....41
Argumenti liste....41
Glagoli i rezervisane reči....41
Bez nuspojava....42
Izlazni argumenti....43
Razlaganje komandi upita....43
Dajte prednost izuzecima a ne vraćanju poruke o grešci....44
Izdvajanje Try/Catch blokova....44
Obrada grešaka je jedna stvar....45
Magnetna zavisnost klase `Error.java`....45

Ne ponavljajte se....46
Strukturisano programiranje....46
Kako pišete ovakve funkcije?....47
Zaključak....47
`SetupTeardownIncluder`....48
Bibliografija....50

Poglavlje 4: Komentari....51
Komentari ne popravljaju loš kod....52
Izražavajte se kodom....53
Dobri komentari....53
Pravni komentari....53
Informativni komentari....53
Objašnjenje namere....54
Razjašnjenje....55
Upozorenje na posledice....55
Komentari TODO....56
Naglašavanje....57
Javadocs u javnim API-jima....57
Loši komentari....57
Mrmljanje....57
Suvišni komentari....58
Komentari koji vode na pogrešan put....60
Obavezni komentari....61
Komentari u vidu dnevnika....61
Beskorisni komentari....62
Baš nepotrebni komentari....64
Ne koristite komentar kada možete koristiti funkciju ili promenljivu....64
Oznake položaja....64
Komentari na zatvarajućim zagradama....65
Prateća objašnjenja....65
Kod pretvoren u komentar....66
HTML komentari....66
Informacije od opšteg značaja....67
Previše informacija....67
Neočigledna veza....68
Funkcije zaglavlja....68
Javadocs u kodu koji nije za javnu upotrebu....68
Primer....68
Bibliografija....72

Poglavlje 5: Formatiranje 73
Svrha formatiranja 74
Vertikalno formatiranje 74
Novinska metafora 75
Vertikalna otvorenost između pojmova 76
Vertikalna gustina 77
Vertikalna udaljenost 78
Vertikalan redosled 82
Horizontalno formatiranje 83
Horizontalna otvorenost i gustina 83
Horizontalno poravnanje 84
Uvlačenja 86
Jednoredni opseg 87
Timska pravila 87
Pravila formatiranja ujka Boba 88

Poglavlje 6: Objekti i strukture podataka 91
Apstrakcija podataka 91
Antisimetrija podataka/objekata 93
Demetrin zakon 96
Sudar vozova 96
Hibridi 97
Sakrivajuća struktura 97
Objekti za prenos podataka 98
Aktivni zapisi 99
Zaključak 99
Bibliografija 99

Poglavlje 7: Obrada grešaka 101
Koristite izuzetke, a ne povratne kodove 102
Napišite svoj Try-Catch-Finally kod 103
Koristite neproverene izuzetke 104
Obezbedite kontekst sa izuzecima 105
Definišite klase izuzetaka prema potrebama pozivaoca 105
Definišite normalan protok 107
Ne vraćajte Null 108
Ne prosleđujte Null 109
Zaključak 110
Bibliografija 110

Poglavlje 8: Granice 111
Korišćenje koda drugih proizvođača 112
Istraživanje i razumevanje granica 114
Razumevanje paketa log4j 114
Testovi za učenje su bolji od besplatnih 116
Korišćenje koda koji još ne postoji 116
Jasne granice 118
Bibliografija 118

Poglavlje 9: Jedinični testovi 119
Tri zakona programiranja zasnovanog na testiranju 120
Održavanje testova jasnim 121
Testovi pružaju razne mogućnosti 122
Jasni testovi 122
Jezik testiranja specifičnog za domen 125
Dvostruki standard 126
Jedna tvrdnja po testu 128
Jedan koncept po testu 129
F.I.R.S.T. 130
Zaključak 131
Bibliografija 131

Poglavlje 10: Klase 133
Organizacija klase 134
Kapsulacija 134
Klase treba da budu male! 134
Princip jedne odgovornosti 137
Kohezija 138
Održavanje rezultata kohezije u više malih klasa 139
Organizovanje za promene 145
Izolovanje od promene 148
Bibliografija 149

Poglavlje 11: Sistemi 151
Kako biste izgradili grad? 152
Odvojite izgradnju sistema od njegove upotrebe 152
Razdvajanje glavnog 153
Abstract Factory 153
Uvođenje zavisnosti 154
Proporcionalno uvećanje 155
Nadležnosti unakrsnog preseka 158

Java proksiji 159
Čist Java AOP radni okvir 161
AspectJ aspekti 164
Proba sistemske arhitekture 164
Optimizujte donošenje odluka 165
Koristite standarde mudro, onda kada dodaju *dokazivu* vrednost 165
Sistemima je potreban određen jezik domena 166
Zaključak 166
Bibliografija 167

Poglavlje 12: Nastajanje 169
Biti jasan pomoću dizajna u nastajanju 169
Pravilo 1 jednostavnog dizajna: Izvršava sve testove 170
Pravila 2-4 jednostavnog dizajna: Refaktorisanje 170
Bez duplikata 171
Izražajno 173
Što manji broj klasa i metoda 174
Zaključak 174
Bibliografija 174

Poglavlje 13: Konkurentnost 175
Zašto konkurentnost? 176
Mitovi i zablude 177
Izazovi 177
Principi odbrane konkurentnosti 178
Princip jedinstvene odgovornosti 178
Zaključak: Ograničite opseg podataka 178
Zaključak: Koristite kopije podataka 179
Zaključak: Niti moraju biti što je moguće više nezavisne 179
Upoznajte svoju biblioteku 180
Kolekcije koda sigurne za rad sa nitima 180
Upoznajte svoje modele izvršenja 180
Proizvođač-potrošač 181
Čitaoci-upisivači 181
Filozofi za stolom 182
Pazite na zavisnosti između sinhronizovanih metoda 182
Održavajte sinhronizovane delove malim 183
Pisanje ispravnog koda za isključivanje je teško 183
Testiranje koda za nitni rad 183
Lažne propuste tretirajte kao kandidate za probleme 184
Učinite da prvo radi kod koji nije nitni 184

Učinite da kod sa nitima bude uklopiv 184
Učinite da kod sa nitima bude podesiv 185
Radite sa više niti nego procesora 185
Izvršavajte kod na različitim platformama 185
Uradite instrumentaciju koda da biste isforsirali greške 185
Ručno kodiranje 186
Automatizovano menjanje 186
Zaključak 187
Bibliografija 188

Poglavlje 14: Postupno doterivanje 189
Implementacija Argova 190
Kako sam ovo uradio? 197
Args: Gruba skica 197
Tako da sam stao 210
O inkrementalizmu 211
Argumenti String 213
Zaključak 253

Poglavlje 15: JUnit iznutra 255
JUnit radni okvir 256
Zaključak 270

Poglavlje 16: Refaktorisanje klase `SerijalDate` **271**
Prvo, učinite da radi 272
Zatim uradite to kako treba 274
Zaključak 288
Bibliografija 288

Poglavlje 17: Mirisi i heuristike 289
Komentari 290
K1: *Neodgovarajuće informacije* 290
K2: *Nepotreban komentar* 290
K3: *Suvišan komentar* 290
K4: *Loše napisan komentar* 291
K5: *Isključen komentar koda* 291
Okruženje 291
O1: *Za izgradnju je potrebno više od jednog koraka* 291
O2: *Testovi zahtevaju više od jednog koraka* 291

Funkcije 292
F1: *Previše argumenata* 292
F2: *Izlazni argumenti* 292
F3: *Argumenti zastavice* 292
F4: *Mrtva funkcija* 292
Generalno 292
G1: *Više jezika u jednoj izvornoj datoteci* 292
G2: *Očigledno ponašanje je neimplementirano* 292
G3: *Netačno ponašanje na granicama* 293
G4: *Zaobilaženje bezbednosti* 293
G5: *Udvostručavanje* 293
G6: *Kod na pogrešnom nivou apstrakcije* 294
G7: *Osnovne klase zavise od njihovih derivata* 295
G8: *Previše informacija* 295
G9: *Mrtav kod* 296
G10: *Vertikalno odvajanje* 296
G11: *Nedoslednost* 296
G12: *Nered* 297
G13: *Veštačko povezivanje* 297
G14: *Osobina zavisti* 297
G15: *Argumenti selektori* 298
G16: *Nejasna namera* 299
G17: *Pogrešno postavljena odgovornost* 299
G18: *Neprikladna statika* 300
G19: *Koristite objašnjavajuće varijable* 300
G20: *Imena funkcija treba da govore šta rade* 301
G21: *Razumevanje algoritma* 301
G22: *Učinite logičke zavisnosti fizičkim* 302
G23: *Dajte prednost polimorfizmu u odnosu na If/Else ili Switch/Case* 303
G24: *Pridržavajte se standardnih konvencija* 304
G25: *Zamenite čarobne brojeve imenovanim konstantama* 304
G26: *Budite precizni* 305
G27: *Struktura nad konvencijom* 305
G28: *Kapsuliranje uslova* 305
G29: *Izbegavajte negativne uslove* 306
G30: *Funkcije treba da rade jednu stvar* 306
G31: *Skrivena vremenska povezivanja* 307
G32: *Ne strukturišite kod proizvoljno* 308

G33: *Kapsulirajte granične uslove*.....308
G34: *Funkcije treba da se spuštaju samo na jedan nivo apstrakcije*.....308
G35: *Čuvajte podatke koji se mogu konfigurisati na visokim nivoima*.....310
G36: *Izbegavajte tranzitivnu navigaciju*.....311
Java.....311
J1: *Izbegavajte duge spiskove uvoza koristeći džokere*.....311
J2: *Ne nasleđujte konstante*.....312
J3: *Konstante naspram enuma*.....313
Imena.....314
I1: *Izaberite opisna imena*.....314
I2: *Izaberite imena na odgovarajućem nivou apstrakcije*.....315
I3: *Koristite standardnu nomenklaturu gde god je to moguće*.....316
I4: *Jednoznačna imena*.....316
I5: *Koristite dugačka imena za velike opsege*.....317
I6: *Izbegavajte kodiranje*.....317
I7: *Imena treba da opisuju sporedna dejstva*.....317
Testovi.....317
T1: *Nedovoljni testovi*.....317
T2: *Koristite alat za prikrivanje!*.....318
T3: *Ne preskačite jednostavne testove*.....318
T4: *Zanemaren test je pitanje o dvosmislenosti*.....318
T5: *Testiranje graničnih uslova*.....318
T6: *Detaljno testiranje u blizini grešaka*.....318
T7: *Obrasci otkazivanja se otkrivaju*.....318
T8: *Obrasci ispitivanja mogu se otkriti*.....318
T9: *Testovi treba da budu brzi*.....319
Zaključak.....319
Bibliografija.....319

Dodatak A: Konkurentnost II.....321
Primer klijent/server.....321
Server.....321
Dodavanje niti.....322
Nadgledanje servera.....323
Zaključak.....325
Mogući putevi izvršenja.....325
Broj putanja.....326
Kopanje dublje.....327
Zaključak.....330

Upoznajte svoju biblioteku 330
Izvršni radni okvir 330
Neblokirajuća rešenja 331
Bezbedne klase bez niti 332
Zavisnosti između metoda mogu pokvariti konkurentan kod 333
Tolerisanje otkazivanja 334
Zaključavanje na strani klijenta 334
Zaključavanje na strani servera 336
Povećanje propusnosti 337
Proračun protoka sa jednom niti 338
Proračun protoka sa više niti 338
Blokada 339
Uzajamno isključivanje 340
Zaključaj i čekaj 340
Ne preko reda 340
Kružni zastoj 340
Prekid uzajamnog isključivanja 341
Prekid uslova zaključaj i čekaj 341
Prekid uslova ne preko reda 342
Prekid kružnog zastoja 342
Testiranje višenitnog koda 342
Alati za podršku testiranja višenitnog koda 345
Zaključak 346
Vežba: Primeri punog koda 346
Klijent/server bez niti 346
Klijent/server pomoću niti 350

Dodatak B: `org.jfree.date.SerialDate` 353

Dodatak C: Uporedne reference poglavlja Mirisi i heuristike 425

Epilog 427

Indeks 429

1

Jasan kod



Ovu knjigu čitate iz dva razloga. Prvi razlog je što ste programer. Drugi, zato što želite da budete bolji programer. Odlično. Potrebni su nam bolji programeri.

Ovo je knjiga o dobrom programiranju. Prepuna je koda. Kod ćemo gledati iz svakog ugla. Gledaćemo na njega odozgo, odozdo na gore i iznutra ka spolja. Kad završimo, znaćemo mnogo o kodu. Štaviše, moći ćemo da utvrdimo razliku između dobrog i lošeg koda. Znaćemo kako da napišemo dobar kod. I znaćemo kako da loš kod pretvorimo u dobar kod.

Neka bude kod

Moglo bi da se tvrdi da je knjiga o kodu nekako zastarela – kod više nije problem, i da bi trebalo umesto toga da brinemo o modelima i zahtevima. Zaista, postoje mišljenja da smo blizu kraja rada sa kodom, jer će se, navodno, uskoro svi kodovi generisati a ne pisati. U tom slučaju programeri neće biti potrebni jer će poslovni ljudi generisati programe iz specifikacija.

Besmislica! Nikada se nećemo osloboditi od koda, jer kod predstavlja detalje zahteva. Na nekom nivou ti se detalji ne mogu zanemariti ili apstrahovati, moraju da budu specifikovani. Tako detaljno preciziranje da mašina može da ih izvrši naziva se *programiranje*. Ta specifikacija je *kod.*

Očekujem da će nivo apstrakcije naših jezika i dalje rasti. Takođe očekujem da će i broj namenskih jezika i dalje rasti. To će biti dobra stvar. Ali to neće eliminisati kod. Zaista, sve specifikacije napisane na tom višem nivou i namenskom jeziku biće kod! I dalje će morati da bude strog, tačan i toliko formalan i detaljan da ga mašina može razumeti i izvršiti.

Ljudi koji misle da će kod jednog dana nestati su poput matematičara koji se nadaju da će jednog dana otkriti matematiku koja ne mora biti formalna. Oni se nadaju da ćemo jednog dana otkriti način stvaranja mašina koje mogu da rade ono šta želimo, a ne ono šta kažemo. Ove mašine će morati tako dobro da nas razumeju da mogu da prevedu naše nejasne zahteve u savršeno izvršavanje programa koji tačno zadovoljavaju željene potrebe.

To se nikada neće dogoditi. Ni ljudi, sa svom intuicijom i kreativnošću, nisu u stanju da stvore uspešne sisteme na osnovu nejasnih osećanja korisnika. Zaista, ako nas je disciplina specifikacije zahteva išta naučila, onda je to da su tačno određeni zahtevi formalni kao i sam kod i mogu delovati kao izvršni testovi tog koda!

Zapamtite da je kod zaista jezik na kojem na kraju izražavamo zahteve. Možemo stvoriti jezike koji su bliži zahtevima ili možemo stvoriti alate koji će nam pomoći da analiziramo i sastavimo te zahteve u formalne strukture. Ali nikada nećemo eliminisati potrebnu preciznost – zato će uvek postojati kod.

Loš kod

Nedavno sam pročitao predgovor Kent Beckove knjige *Implementation Patterns*[1]. U njemu piše: „...ova knjiga je zasnovana na prilično krhkoj premisi: da je dobar kod važan... “ *Krhka* premisa? Ne slažem se! Mislim da je ta premisa jedna od najatraktivnijih,

1. [Beck07]

podržanih i preopterećenih od svih u našem poslu (i mislim da Kent to zna). Znamo da je dobar kod važan jer smo toliko dugo morali da se bavimo njegovim nedostacima.



Znam jednu kompaniju u kojoj je krajem 80-ih godina prošlog veka razvijena *strašna* aplikacija. Bila je veoma popularna, a mnogi profesionalci su je kupili i koristili. Ali ciklusi objavljivanja novih verzija aplikacije su počeli da se produžavaju. Greške nisu popravljene od jedne do druge verzije. Vreme učitavanja je raslo, a rušenja su postala učestala. Sećam se dana kad sam frustrirano zatvorio aplikaciju i nikad je više nisam koristio. Kompanija je ubrzo nakon toga prestala da postoji.

Dve decenije kasnije sreo sam jednog od nekadašnjih zaposlenih u toj kompaniji i pitao ga šta se dogodilo. Odgovor je potvrdio moje bojazni. Požurili su da izbace proizvod na tržište a imali su veliki nered u kodu. Dodavanjem sve više i više funkcija, kod je postajao sve lošiji i lošiji sve dok jednostavno nisu više mogli da imaju kontrolu nad njim. Loš kod je srušio kompaniju.

Da li *vam* je ikad jako zaglavio loš kod? Ako ste programer sa bilo kakvim iskustvom, onda ste to mnogo puta osetili. Postoji i ime za to. Mi to nazivamo *probijanje*. Probili smo se kroz loš kod. Probijamo se kroz mora zapetljanog trnja i skrivenih zamki. Borimo se da pronađemo svoj put, nadajući se nekom nagoveštaju, nekom tragu onome šta se dešava, ali sve što vidimo je sve besmisleniji kod.

Naravno da ste bili zaglavljeni u lošem kodu. Pa, zašto ste ga napisali?

Da li ste žurili? Verovatno je tako. Možda ste mislili da nemate vremena da dobro uradite posao, da će se vaš šef ljutiti na vas ako potrošite vreme da „ispeglate“ svoj kod. Možda ste se samo umorili od rada na tom programu i želeli ste da ga završite. Ili ste možda pogledali koliko zaostaju druge stvari koje ste obećali da ćete uraditi i shvatili da morate da „zbrzate” ovaj modul da biste mogli da pređete na sledeći. Svi smo to radili.

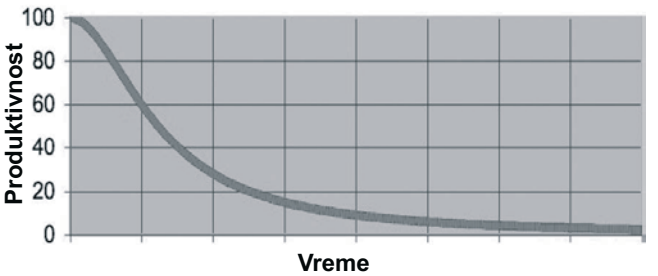
Svi smo videli nered koji smo napravili, a zatim smo odlučili da ga ostavimo za drugi dan. Svi smo osetili olakšanje kada smo videli kako naš neuredan program radi i odlučili da je radni nered bolji nego ništa. Svi smo rekli da ćemo se vratiti i očistiti kasnije. Naravno, u one dane nismo poznavali LeBlancov zakon: *Kasnije znači nikad.*

Ukupan trošak nereda

Ako ste programer duže od dve ili tri godine, verovatno vas je usporio neuredan kod. Stepen usporavanja može biti značajan. Posle godinu ili dve, timovi koji su na početku projekta radili vrlo brzo, mogu se naći u situaciji da se kreću puževim koracima. Svaka promena koda utiče na dva ili tri druga dela koda. Nijedna promena nije beznačajna. Svako

dodavanje ili modifikacija sistema zahteva da se slojevi, zavoji i čvorovi „razumeju“ tako da se može dodati još više zapleta, zavoja i čvorova. S vremenom nered postaje toliko veliki i tako dubok i tako visok da ga ne mogu očistiti. Ne postoji način za to.

Kako nered raste, produktivnost tima nastavlja da pada, asimptotski se približavajući nuli. Kako se produktivnost smanjuje, menadžment čini jedino što može. Povećava broj saradnika u nadi da će povećati produktivnost. Ali ti novi saradnici nisu upućeni u dizajn sistema. Oni ne prave razliku između promene koja odgovara svrsi projekta i promene koja je u neskladu sa ciljem projekta. Štaviše, oni i svi ostali u timu su pod strašnim pritiskom da povećaju produktivnost. Dakle – prave sve više i više nereda, usmeravajući produktivnost prema nuli. (Vidi sliku 1-1.)



Slika 1-1 Produktivnost u odnosu na vreme

Veliko redizajnirnje na pomolu

Na kraju, tim se pobuni i obaveštava menadžment da ne može da nastavi da razvija tu neprijatnu kolekciju kodova. Zahtevaju redizajn; menadžment ne želi da troši resurse na potpuno novi redizajn projekta, ali ne može da porekne da je produktivnost na užasno niskom nivou. Na kraju, rukovodioci se povinuju zahtevima programera i dozvoljavaju veliko redizajniranje.

Izabran je novi „opaki“ tim. Svi žele da budu u njemu jer se radi o projektu koji počinje od nule (eng. *Green-filed project*). Saradnici u timu počinju ispočetka i stvaraju nešto zaista lepo. Ali samo najbolji i najsjajniji su izbarani za opaki tim. Svi ostali moraju nastaviti da održavaju trenutni sistem.

Sada se radi o trci dve ekipe. Opaki tim mora da izgradi novi sistem koji radi sve što radi i stari sistem. I ne samo to, tim mora da bude u toku sa promenama koje se neprestano dešavaju u starom sistemu. Menadžment neće zameniti stari sistem sve dok novi sistem ne može da uradi sve ono što može i stari.

Ova trka može da traje veoma dugo. Dešavalo se da traje i 10 godina. I do trenutka kada je to urađeno, prvobitni članovi opake ekipe su odavno otišli, a sadašnji članovi traže da se novi sistem redizajnira, jer je veliki nered.

Ako ste iskusili čak i jedan mali deo situacije koju sam upravo opisao, tada već znate da trošenje vremena na čišćenje koda nije samo isplativo, to je stvar profesionalnog opstanka.

Stav

Da li ste ikada prošli kroz toliko nereda da su vam bile potrebne nedelje da prođete ono što je trebalo da traje satima? Jeste li videli da ono što je trebalo da bude jednoredna promena, umesto toga napravljeno je u stotine različitih modula? Ovi simptomi su previše česti.

Zašto se to dešava kodu? Zašto se dobar kod tako brzo pretvori u loš kod? Postoji mnogo objašnjenja za to. Žalimo se da su se zahtevi promenili na takav način da remete prvobitan projekat. Pozivamo se na rokove koji su bili tesni da bi se stvari odvijale kako treba. Blebetamo o glupim menadžerima, netolerantnim kupcima i beskorisnim marketinškim akcijama. Ali krivica, dragi Perice, nije u višoj sili, već u nama samima. Mi smo neprofesionalni.

Ovo je bi mogla biti gorka pilula koju treba progutati. Kako bi ovaj nered mogao da bude *naša* greška? Šta je sa zahtevima? Šta je sa rasporedom? Šta je sa glupim menadžerima i beskorisnim marketingom? Zar oni ne snose deo krivice?

Ne. Menadžeri i prodavci traže od *nas* informacije potrebne za davanje obećanja i obaveza; čak i kad nas ne gledaju, ne treba se stideti reći im šta mislimo. Korisnici traže potvrdu na koji će način zahtevi ući u sistem. Rukovodioci projekata zahtevaju od nas da pomognu u kreiranju rasporeda rada. U velikoj meri učestvujemo u planiranju projekta i delimo veliku odgovornost za eventualne propuste, pogotovo ako ti propusti imaju veze sa lošim kodom!

„Ali čekajte!“ reći ćete. „Ako ne učinim ono što moj šef kaže, biću otpušten.“ Verovatno nećete. Većina menadžera želi istinu, čak i kada se ne ponašaju tako. Većina menadžera želi dobar kod, čak i kada su pritisnuti rokovima. Oni mogu sa strašću braniti rokove i zahteve, ali to je njihov posao. *Vaš* posao je da sa istom strašću branite kod.

Vratimo se na temu, šta da ste hirurg i da imate pacijenta koji zahteva da prestanete sa glupim pranjem ruku u pripremama za operaciju, jer to troši previše vremena?[2] Jasno je da je pacijent glavni, ali ipak lekar treba apsolutno da odbije taj zahtev. Zašto? Zato što lekar zna više od pacijenta o rizicima od bolesti i infekcija. Bilo bi neprofesionalno (a pogotovo kriminalno) da se lekar povinuje pacijentu.

Isto tako je neprofesionalno i da se programeri povinuju volji menadžera koji ne razumeju rizike pravljenja nereda.

Primalna zagonetka

Programeri se suočavaju sa zagonetkom od suštinskog značaja. Svi programeri koji imaju više od nekoliko godina iskustva znaju da ih početni nered u pisanju koda usporava. Pa ipak, svi programeri osećaju pritisak pa prave nered da bi ispunili rokove.

Pravi profesionalci znaju da je drugi deo zagonetke pogrešan. *Nećete* ispuniti rok tako što ćete napraviti nered. Zaista, nered će vas odmah usporiti i naterati da propustite rok. *Jedini* način da se rok postigne – jedini način da se radi brzo – je da se kod u svakom trenutku održava što čistijim.

2. Kada je 1847. godine Ignaz Semmelveis, prvi put preporučio lekarima pranje ruku, to je odbijeno na osnovu toga što su lekari bili previše zauzeti i nisu imali vremena da peru ruke.

2

Smislena imena

Tim Ottinger



Uvod

Imena su svuda u softveru. Imenujemo naše varijable, funkcije, argumente, klase i pakete. Imenujemo izvorne datoteke i direktorijume koji ih sadrže. Mi imenujemo naše jar datoteke i war datoteke i ear datoteke. Imenujemo i imenujemo i imenujemo. Budući da to radimo mnogo toga, bilo bi bolje da to dobro radimo. Sledi nekoliko jednostavnih pravila za dobijanje dobrih imena.

Koristite imena koja otkrivaju nameru

Lako je reći da imena treba da otkriju nameru. Ono što želimo da naglasimo je da se *ozbiljno* posvetimo tome. Odabir dobrih imena iziskuje vreme, ali i doprinosi uštedi na vremenu. Zato vodite računa o imenovanju i menjajte imena kada god pronađete bolja. Svi koji čitaju vaš kod (uključujući i vas) biće sretniji ako to učinite.

Naziv promenljive, funkcije ili klase trebalo bi da odgovara na sva važna pitanja. Trebalo bi vam kaže zašto postoji, šta radi i kako se koristi. Ako je za ime potreban komentar, ime ne otkriva našu nameru.

```
int d; // proteklo vreme u danima
```

Ime d ne otkriva ništa. Ne evocira osećaj proteklog vremena, niti dana. Treba da odberemo ime koje određuje šta se meri i jedinicu te mere:

```
int minutedTimeInDais;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDais;
```

Odabirom imena koja otkrivaju nameru može se znatno olakšati razumevanje i promena koda. Koja je namena ovog koda?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4) list1.add(x);
  return list1;
}
```

Zašto je teško reći šta ovaj kod radi? Nema složenih izraza. Razmak i uvlake su razumni. Spominju se samo tri promenljive i dve konstante. Ne postoje čak ni neke maštovite klase ili polimorfne metode, samo spisak nizova (ili tako izgleda).

Problem nije u jednostavnosti koda, već u značenju koda: stepen do koga smisao nije izričit u samom kodu. Kod implicitno zahteva da znamo odgovore na pitanja kao što su:

1. Kakve su stvari u `theList`?
2. Kakav je značaj stavke sa indeksom nula u `theList`?
3. Kakav je značaj vrednosti `4`?
4. Kako bih koristio listu koja se vraća?

Odgovori na ova pitanja ne postoje u gornjem primeru koda, *ali mogli su da budu*. Recimo da se radi o igri čišćenje minskog polja (eng. *mine sweeper*). Tabla na kojoj se igra ta igrica sastavljena je od ćelija (eng. *cell*) i zove se `theList`. Preimenujmo je u `gameBoard`.

Svaka ćelija na tabli predstavljena je jednostavnim nizom. Dalje, otkrivamo da je nulta lokacija statusne vrednosti i da statusna vrednost 4 znači „označena zastavicom“ (eng. *flagged*). Samo davanjem imena tim pojmovima možemo znatno poboljšati kod:

```
public List <int []> getFlaggedCells() {
  List <int []> flaggedCells = nova ArrayList <int []>();
  for (int [] cell: gameBoard)
    if (cell [STATUS_VALUE] == FLAGGED)
      flaggedCells.add (cell);
  return flaggedCells;
}
```

Obratite pažnju da se jednostavnost koda nije promenila. I dalje ima sasvim isti broj operatora i konstanti, sa potpuno istim brojem ugnežđenja. Ali, kod je postao mnogo eksplicitniji.

Možemo ići dalje i napisati jednostavnu klasu za ćelije umesto da koristimo niz `ints`. Može da se doda funkcija otkrivanja namera (nazovimo je `isFlagged`) za sakrivanje magičnih brojeva. To rezultira novom verzijom funkcije:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = nova ArrayList<Cell>();
  for (Cell cell: gameBoard)
    if (cell.isFlagged())
      flaggedCells.add (ćelija);
  return flaggedCells;
}
```

Uz ove jednostavne promene imena, nije teško razumeti o čemu se radi. U tome se ogleda snaga izbora dobrih imena.

Izbegavajte dezinformacije

Programeri moraju da izbegavaju ostavljanje loših tragova koji prikrivaju značenje koda. Treba da izbegavamo reči čija ukorenjena značenja skreću od naših nameravanih značenja. Na primer, `hp`, `aix` i `sco` bi bila loša imena promenljivih jer su to imena Unix platformi ili varijanti. Čak i ako kodirate hipotenuzu, a `hp` izgleda kao dobra skraćenica, to može predstavljati dezinformaciju.

Ne označavajte grupu naloga (eng. *account*) kao accountList, osim ako nije u pitanju `List`. Reč List znači nešto specifično za programere. Ako kontejner sadrži nalog to zapravo nije `List` i može dovesti do pogrešnih zaključaka.[1] Dakle `accountGroup` ili `bunchOfAccounts` ili samo prosto `account` bi bilo bolje.

Pazite pri korišćenju imena koja se razlikuju u malim detaljima. Koliko vremena treba da se uoči suptilna razlika između `XYZControllerForEfficientHandlingOfStrings` u jednom modulu i, negde malo udaljenije, `XYZControllerForEfficientStorageOfStrings`? Reči imaju zastrašujuće slične oblike.

Pisanje sličnih koncepata na sličan način je *informacija*. Korišćenje nedoslednih pravopisa je *dezinformacija*. Sa savremenim Java okruženjima uživamo u automatskom dovršavanju koda. Pišemo nekoliko znakova imena i pritisnemo neku kombinaciju tastera

1. Kao što ćemo kasnije videti, čak i ako je kontejner `List`, verovatno je bolje ne kodirati vrstu kontejnera u ime.

(ako je to tako) i dobijemo spisak mogućih kompletiranja za to ime. Veoma je korisno ako su imena za slične stvari poređana po abecedi i ako su razlike očigledne, jer je verovatno da će programer odabrati objekat po imenu a da ne vidi vaše bogate komentare ili čak listu metoda koje nudi ta klasa.

Zaista užasan primer dezinformativnih imena bila bi upotreba malih slova `L` ili velikih slova `O` kao promenljivih imena, naročito u kombinaciji. Problem je, naravno, što oni izgledaju gotovo u potpunosti kao konstante jedan i nula.

```
int a = l;
if (O == l)
  a = O1;
else
  l = O1;
```

Čitalac može ovo da smatra zgodnim, ali imali smo prilike da vidimo kod u kome je takvih stvari bilo u izobilju. U jednom slučaju autor koda predložio je korišćenje različitog fonta kako bi razlike bile očiglednije; rešenje koje bi trebalo preneti svim budućim programerima usmenim putem ili u pisanom dokumentu. Konačno, problem je u potpunosti i bez novih dodataka premošćen jednostavnim preimenovanjem.

Pravite smislene razlike



Programeri sami sebi stvaraju probleme kada pišu kod isključivo da bi zadovoljili kompajler (sastavljač) ili interpreter (prevodilac). Primera radi, iz razloga što ne možete da koristite isto ime koje se odnosi na dva različita pojma u istom opsegu; možda ćete doći u iskušenje da promenite jedno ime na proizvoljan način. Ponekad se to desi greškom u pisanju imena, što dovodi do iznenađujuće situacije da kada ispravite slovne greške nije moguće uraditi kompajliranje programa.[2]

Nije dovoljno dodati nizove brojeva ili zvučne reči, iako je kompajler zadovoljan. Ako imena moraju biti različita, onda treba i da znače nešto različito.

Imenovanje nizova brojeva (`a1, a2, .. aN`) suprotno je imenovanju sa namerom. Takva imena nisu dezinformativna – ona su neinformativna; ne daju nikakav zaključak o autorovoj nameri. Razmotrite:

```
public static void copyChars (char a1 [], char a2 []) {
  for (int i = 0; i < a1.length; i++) {
    a2 [i] = a1 [i];
  }
}
```

2. Razmotrite, na primer, zaista užasnu praksu imenovanja promenljive imenom `klass` samo zato što je `class` ime upotrebljeno za nešto drugo.

Ova funkcija se mnogo bolje čita kada se pojmovi `source` i `destination` koriste za nazive argumenata.

Prazne reči su još jedna besmislena razlika. Zamislite da imate klasu `Product`. Ako imate drugu sa imenom `ProductInfo` ili `ProductData`, napravili ste različita imena, ali ne čineći ih po značenju različitim. `Info` i `Data` su nejasne prazne reči poput `a`, `an` i `the`.

Imajte na umu da nema ničeg lošeg u korišćenju za prefiks reči kao što su `a` i `the` sve dok one daju smislenu razliku. Na primer, možete koristiti `a` za sve lokalne varijable i `the` za sve argumente funkcija.[3] Problem nastaje kada odlučite da nazovete promenljivu `theZork`, a već imate drugu promenljivu koja se zove `zork`.

Prazne reči su suvišne. Reč `promenljiva` nikada ne bi trebalo da se pojavljuje u imenu promenljive. Reč `tabela` nikada ne bi trebalo da se pojavljuje u nazivu tabele. Kako je `ImeStringa` bolje od `Ime`? Da li će `Ime` ikada označavati broj sa pokretnim zarezom? Ako je tako, krši se ranije pravilo o dezinformacijama. Zamislite da u kodu nađete jednu klasu koja se zove `Kupac`, a druga da se zove `KupacObjekat`. Kako da shvatite razliku? Koja će predstavljati najbolji put do istorije plaćanja kupca?

Postoji aplikacija za koju znamo gde je to tako. Promenili smo imena da bismo zaštitili krivca, ali evo kako je to tačno izgledalo:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Kako programeri koji rade na ovom projektu da znaju koju od ovih funkcija treba da pozovu?

U nedostatku posebnih konvencija, promenljiva `novacIznos` ne razlikuje se od `novac`, `kupacInfo` se ne razlikuje od `kupac`, `racunPodaci` se ne razlikuje od `racun`, a `theMessage` se ne razlikuje od `message`. Razložite imena na takav način da onaj ko čita kod zna šta te razlike nude.

Koristite izgovorljive nazive

Ljudi su dobri sa rečima. Značajan deo našeg mozga posvećen je konceptu reči. A reči su, po definiciji, izgovorljive. Šteta bi bilo ne iskoristiti prednost velikog dela našeg mozga koji je evoluirao da bi se nosio sa govornim jezikom. Zato neka vaša imena budu izgovorljiva.

Ako nešto ne možete da izgovorite, ne možete o tome ni da razgovarate, a da ne zvučite kao idiot. „Pa, ovde, na frgrmn, imamo agazagbag, vidite?“ Ovo je veoma važno jer programiranje predstavlja društvenu aktivnost.

U jednoj kompaniji postoji pojam `genymdhms` (datum generisanja, godinu (`y`), mesec (`m`), dan (`d`), sat (`h`), minut (`m`) i sekunda (`s`)), tako da se izgovara: „gen vaj em di ejč em es“. Imam neprijatnu naviku da sve izgovaram onako kako je napisano, pa sam i ja počeo da govorim „gen vaj em di ejč em es“. Brojni dizajneri i analitičari su nas kritikovali

3. Ujka Bob je to radio u C++ ali je odustao od tog načina zbog modernih IDE-a koji su ga učinili nepotrebnim.

3

Funkcije



U ranim danima programiranja sklopili smo naše prve sisteme rutina i podrutina. Zatim, u doba Fortrana i PL/1, sastavili smo naše sisteme programa, potprogram i funkcija. Od tih početaka, preživele su samo funkcije, kao prva linija organizacije u bilo kom programu. Tema ovog poglavlja je pisanje funkcija na dobar način.

Razmotrite kod u listingu 3-1. Teško je pronaći dugačku funkciju u alatu FitNesse[1], ali nakon malo pretraživanja naći ćete je. Ne samo da je dugačka, već ima duplirani kod, puno čudnih nizova, mnogo neobičnih i neočiglednih tipova podataka i interfejsa za programiranje aplikacija (eng. *application programming interface, API*). U naredna tri minuta pogledajte koliko smisla možete da nađete u svemu tome.

Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
```

1. Alat za testiranje otvorenog koda, www.fitnesse.org

Listing 3-1 (nastavak)
HtmlUtil.java (FitNesse 20070619)

```
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Da li posle tri minuta proučavanja razumete ovu funkciju? Verovatno ne. Previše toga se ovde događa na previše različitih nivoa apstrakcije. Postoje neobični nizovi i čudni pozivi funkcije pomešani sa dvostruko ugnežđenim `if` naredbama kojima upravljaju logičke vrednosti.

Međutim, sa samo nekoliko jednostavnih metoda ekstrakcije, nekim preimenovanjima i malim restrukturiranjem, uspeo sam da uhvatim nameru funkcije u devet linija u listingu 3-2. Pogledajte u naredna tri minuta da li možete to da shvatite.

Verovatno ne razumete sve detalje, osim ako niste koristili FitNesse. Ipak, verovatno razumete da ova funkcija uključuje početne vrednosti (eng. *setup*) i raščlanjivanje (eng. *teardown*) test stranice i zatim pravi HTML stranu. Ako ste upoznati sa JUnitom[2] verovatno shvatate da ova funkcija pripada nekoj vrsti radnog okruženja za testiranje veb strana. To je tačno. Dolaženje do tih informacija na osnovu listinga 3-2 prilično je jednostavno, ali je prilično nejasno u listingu 3-1.

2. Alat za testiranje otvorenog koda u Javi. www.junit.org

Šta je to što funkciju iz listinga 3-2 čini čitkom i razumljivom? Kako da učinimo da funkcija saopšti svoju nameru? Koje osobine možemo dati funkcijama tako da povremenom čitaocu omoguće da oseti vrstu programa koje one sadrže?

Listing 3-2
HtmlUtil.java (refaktorisano)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Mala!

Prvo pravilo funkcija je da treba da budu male. Drugo pravilo funkcija je *da bi trebalo da budu manje i od toga*. Ovo nije tvrdnja koju mogu da opravdam. Ne mogu da dam ni jednu referencu koja pokazuje da su vrlo male funkcije bolje, ali ono što mogu da kažem je da već skoro četiri decenije pišem funkcije različitih veličina. Napisao sam nekoliko gadosti od 3.000 linija koda. Napisao sam mnogo funkcija u opsegu od 100 do 300 linija koda, ali i funkcije koje su bile dužine od 20 do 30 linija koda. Dugi niz pokušaja i grešaka naučio me je da bi funkcije trebalo da budu veoma male!

Osamdesetih godina prošlog veka govorili smo da funkcija ne treba da bude veća od veličine ekrana. Naravno da je to rečeno u vreme kada su ekrani VT100 bili veličine 24 reda i 80 kolona, a editori su za svoje potrebe koristili 4 reda. Danas sa manjim fontovima i lepim velikim monitorom, na ekran može da stane 150 znakova u jednom redu i 100 ili više redova. Ipak, redovi ne smeju da budu dugački 150 znakova, a funkcije ne bi trebalo da budu dugačke 100 redova. Ustvari, funkcije nikako ne bi trebalo da budu duže od 20 redova.

Koliko kratka bi trebalo da bude funkcija? 1999. godine posetio sam Kenta Becka u njegovoj kući u Oregonu. Zajedno smo napisali nekoliko programa, a on mi je u jednom trenutku pokazao simpatičan mali program napisan u jeziku Java/Swing koji je nazvao Sparkle. Program je na ekranu proizveo vizuelni efekat vrlo sličan onome koji stvara čarobnom štapić dobre vile u filmu Pepeljuga. Pomeranjem miša, iz kursora bi iskakale sjajne iskre i padale na dno ekrana u simuliranom gravitacionom polju. Kad mi je Kent pokazao kod, zadivilo me je koliko su sve funkcije bile male. Navikao sam na

funkcije napisane u programu Swing koje su bile dugačke „kilometrima“. Svaka funkcija u ovom programu bila je dugačka samo dva, tri ili četiri reda. Svaka od njih je bila očigledna i jasna sama po sebi. I svaka od njih vas je vodila do sledeće, ubedljivim redosledom. Tako kratke bi trebalo da budu vaše funkcije![3]

Koliko kratke treba da budu vaše funkcije? Obično bi trebalo da budu kraće od listinga 3-2! Zaista, listing 3-2 bi trebalo skratiti na listing 3-3.

Listing 3-3
`HtmlUtil.java (re-refactored)`

```
public static String renderPageWithSetupsAndTeardowns(
PageData pageData, boolean isSuite) throws Exception {
if (isTestPage(pageData))
includeSetupAndTeardownPages(pageData, isSuite);
return pageData.getHtml();
}
```

Blokovi i uvlačenja

Podrazumeva se da bi blokovi unutar naredbi `if`, `else`, `while` itd. trebalo da budu dugački jedan red. Taj red bi verovatno trebalo da predstavlja pozivanje funkcije. To ne samo da čini funkciju malom, već joj dodaje i dokumentacionu vrednost jer funkcija koja se poziva unutar bloka može imati lepo opisno ime.

Ovo podrazumeva da funkcije ne bi trebalo da budu toliko velike da bi sadržale ugnežđene strukture. Prema tome, unutar funkcije ne sme biti više od jednog ili dva nivoa uvlačenja, što, naravno, čini funkcije lakšim za čitanje i razumevanje.

Činite jednu stvar



Trebalo bi da bude jasno da listing 3-1 radi puno više od jedne stvari. Između ostalog, on pravi bafere, preuzima stranice, traži nasleđene stranice, prikazuje putanje, dodaje prikrivene nizove i generiše HTML kod. Listing 3-1 je veoma zauzet radeći razne stvari, dok sa druge strane, listing 3-3 radi jednu jednostavnu stvar – uključuje početne vrednosti i raščlanjivanja u test stranice.

Sledeći saveti se pojavljuju u jednom ili drugom obliku već 30 ili više godina.

> ***Funkcije treba da rade jednu stvar. Treba da je rade dobro i treba da rade samo to.***

3. Pitao sam Kenta da li još uvek ima taj kod, ali nije uspeo da ga pronađe. Pregledao sam i sve svoje stare računare, ali bezuspešno. Ostalo mi je samo sećanje na taj program.

4

Komentari



„Ne komentarišite loš kod – napišite ga ponovo.“
– Brajan V. Kernighan i P. J. Plaugher[1]

Ništa ne može biti toliko korisno kao dobro ubačen komentar. Ništa ne može zagušti modul više od neozbiljnih priča u prazno u komentarima. Ništa ne može biti toliko štetno kao stari surov komentar koji propagira laž i dezinformacije.

Komentari nisu poput Šindlerove liste. Oni nisu „nasušna potreba.“ Zaista, u najboljem slučaju komentari su nužno zlo. Da su naši programski jezici dovoljno izražajni i da imamo talenat da suptilno koristimo te jezike da bismo izrazili svoje namere, komentari nam ne bi bili toliko potrebni, a možda bismo mogli i bez njih.

1. [KP78], p. 144.

Pravilna upotreba komentara je kompenzacija za naš neuspeh da se izrazimo u kodu. Obratite pažnju da sam upotrebio reč reč *neuspeh*. To sam i mislio. Komentari su uvek nedostaci. Moramo da ih imamo jer ne možemo uvek da smislimo kako da se izrazimo bez njih, ali njihova upotreba nije razlog za slavlje.

Dakle, kada se nađete u položaju da morate da napišete komentar, razmislite o tome i vidite da li postoji način da se izrazite kroz kod. Svaki put kada se izrazite kodom, čestitajte sebi. Svaki put kada napišete komentar, napravite grimasu i budite svesni svoje nesposobnosti izražavanja.

Zašto sam takav po pitanju komentara? Zato što lažu. Ne uvek i ne s namerom, ali se to događa previše često. Što je komentar stariji, dalje je od koda koji opisuje i verovatno je pogrešan. Razlog je jednostavan. Programeri ne mogu da održavaju komentare ažurnim.

Kod se menja i razvija. Delovi koda se premaštaju tamo-ovamo. Ti delovi koda se dele i razmnožavaju i ponovo okupljaju u grupama. Nažalost, komentari ih ne prate uvek, štaviše *ne mogu* uvek da ih prate. Previše često se komentari odvajaju od koda koji opisuju i postaju siročad sve manje tačnosti. Na primer, pogledajte šta se dogodilo sa ovim komentarom i redom koji je trebalo da opiše:

```
MockRequest request;

private final String HTTP_DATE_REGEXP =
"[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
"[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT"; private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Ostale instance promenljive koje su verovatno dodate kasnije, ubačene su između konstante `HTTP_DATE_REGEXP` i pratećeg komentara.

Moguće je reći da programeri moraju da budu disciplinovani i da treba da održavaju komentare na visokom stepenu ažurnosti, relevantnosti i tačnosti. Slažem se, trebalo bi. Ali ja bih radije da ta energija ide ka tome da kod bude toliko jasan i izražen da mu uopšte nisu potrebni komentari.

Netačni komentari su daleko gori od svih komentara. Oni zavaravaju i dovode u zabludu. Oni postavljaju očekivanja koja se nikada neće ispuniti. Oni govore o starim pravilima koja više nisu potrebna ili pravilima kojih ne treba da se držimo.

Istina se može naći samo na jednom mestu: u kodu. Samo kod vam zaista može reći šta se dešava. To je jedini izvor zaista tačnih informacija. Stoga, iako su komentari ponekad potrebni, potrošićemo značajnu energiju da ih minimalizujemo.

Komentari ne popravljaju loš kod

Jedan od najčešćih motiva za pisanje komentara je loš kod. Pišemo modul i znamo da je zbunjujući i neorganizovan. Znamo da je u neredu, pa kažemo sebi: „Bolje bi bilo da napišem komentare!“ Ne! Bolje je da kod učinite jasnim!

Jasan i izražajan kod s nekoliko komentara, daleko je bolji od pretrpanog i složenog koda s puno komentara. Umesto da trošite vreme na pisanje komentara koji objašnjavaju nered koji ste napravili, vreme upotrebite na čišćenje tog nereda.

Izražavajte se kodom

Sigurno postoje slučajevi kada je kod loše sredstvo za izražavanje. Nažalost, mnogi programeri su prihvatili da je kod retko, ako je uopšte, dobro sredstvo za izražavanje. Bez ikakve dileme to je pogrešan stav. Šta biste radije videli? Ovo:

```
// Proverava da li je zaposleni ispunio uslove za beneficije
if ((zaposleni.indikator & INDIKATOR _ SATI) &&
(zaposleni.imagodina > 65))
```

Ili ovo?

```
if (zaposleni.jesteIspunioUsloveZaBeneficije())
```

Potrebno vam je samo nekoliko sekundi razmišljanja da objasnite većinu vaših namera u kodu. U mnogim slučajevima to je jednostavno pisanje funkcije koja izražava istu stvar kao i komentar koji biste napisali.

Dobri komentari

Neki komentari su neophodni ili korisni. Pogledaćemo nekoliko stvari koje smatram vrednim svakog bita koji koriste. Međutim, imajte na umu da je jedini zaista dobar komentar onaj za koji ste našli način da ga ne napišete.

Pravni komentari

Ponekad nas korporativni standardi pisanja koda iz pravnih razloga prisiljavaju da pišemo određene komentare. Na primer, izjave o autorskim pravima i autorstvu su neophodne i razumne stvari koje treba staviti u komentar na početku svake izvorne datoteke.

Evo, na primer, standardnog zaglavlja komentara koje postavljamo na početak svake izvorne datoteke u FitNesseu. Srećan sam što mogu da kažem da naš IDE skriva ove komentare ne prikazujući ih.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2
// or later.
```

Komentari poput ovog ne bi trebalo da budu ugovori ili pravni dokumenti. Ako je moguće, pogledajte standardnu licencu ili druge postojeće dokumente, umesto da u komentar stavljate sve uslove i odredbe.

Informativni komentari

Ponekad je korisno u komentaru pružiti osnovne informacije. Primera radi, razmotrite ovaj komentar koji objašnjava povratnu vrednost apstraktne metode:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Komentar kao ovaj ponekad može biti koristan, ali bolje je da se, tamo gde je to moguće, za prenošenje informacija koristi ime funkcije. Na primer, u ovom slučaju komentar može biti suvišan preimenovanjem funkcije u `responderBeingTested`.

Evo slučaja koji je malo bolji:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

U ovom slučaju komentar nam daje do znanja da je regularni izraz namenjen da upari vreme i datum koji su formatirani funkcijom `SimpleDateFormat.format`, koristeći navedeni niz znakova. Ipak, moglo bi biti i bolje i jasnije, da je ovaj kod premešten u posebnu klasu koja pretvara formate datuma i vremena. Tada bi komentar verovatno bio suvišan.

Objašnjenje namere

Ponekad komentar nadilazi samo korisnu informaciju o primeni i daje dublji uvid. U sledećem slučaju vidimo zanimljivu odluku dokumentovanu komentarom. Upoređujući dva objekta, autor je odlučio da sortira predmete svoje klase veće od objekata bilo kojeg drugog.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // veći smo jer smo ispravnog tipa.
}
```

Evo još boljeg primera. Možda se ne slažete sa programerovim rešenjem problema, ali bar znate šta je on hteo da uradi.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
  String text = "'''bold text'''";
  ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
  AtomicBoolean failFlag = new AtomicBoolean();
  failFlag.set(false);

  //Ovo je najboji pokusaj da dobijemo bolje uslove
  //stvaranjem velikog broja niti (threads).
  for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
```

```
        new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
      Thread thread = new Thread(widgetBuilderThread);
      thread.start();
    }
    assertEquals(false, failFlag.get());
  }
```

Razjašnjenje

Ponekad je korisno pojasniti značenje nekog nejasnog argumenta ili vrednosti koja se vraća, u nešto što je čitljivo. Generalno, bolje je pronaći način da se taj argument ili povratna vrednost jasno pojave u sopstvenom obliku, ali kada je deo standardne biblioteke ili je u kodu koji ne možete da izmenite, komentar koji objašnjava može biti od koristi.

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0); // a == a
  assertTrue(a.compareTo(b) != 0); // a != b
  assertTrue(ab.compareTo(ab) == 0); // ab == ab
  assertTrue(a.compareTo(b) == -1); // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);  // b > a
  assertTrue(ab.compareTo(aa) == 1); // ab > aa
  assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Naravno, postoji značajan rizik da je komentar netačan. Prođite kroz prethodni primer i videćete koliko je teško da proverite da li su tačni. Ovo objašnjava zašto je razjašnjenje potrebno, ali i zašto je rizično. Pre nego što napišete komentare poput ovih, vodite računa o tome da budu tačni.

Upozorenje na posledice

Ponekad je korisno upozoriti druge programere na određene posledice. Primera radi, evo komentara koji objašnjava zašto je određeno testiranje isključeno:



```
// Ne pokreci osim ako nisi dokon
public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);
```

5

Formatiranje



Kad čitaoci „zavire pod haubu“ našeg koda, želimo da ih impresioniramo urednošću, doslednošću i osećajem za detalje. Želimo da ih urednost ne ostavi ravnodušnim, da izviju obrve dok prolaze kroz module i da shvate da su profesionalci uradili svoj posao. Ako umesto toga vide izmešanu masu koda koja izgleda kao da ju je napisala gomila pijanih mornara, onda će verovatno zaključiti da je ista pažnja posvećena detaljima i svakom drugom aspektu projekta.

Treba da vodite računa da vaš kod bude lepo formatiran. Trebalo bi da odaberete skup jednostavnih pravila koja upravljaju formatiranjem vašeg koda, a zatim da ih dosledno primjenjujete. Ako radite u timu, trebalo bi prihvatiti jedan skup pravila formatiranja kojih svi članovi tima moraju da se pridržavaju. U tome vam može pomoći automatizovana alatka koja primenjuje ta pravila.

Svrha formatiranja

Da budemo jasni, formatiranje koda je *važno*. Previše je važno da bi se ignorisalo i previše je važno pa mu se moramo posvetiti. Formatiranje koda odnosi se na komunikaciju, a komunikacija je od prvorazrednog značaja za profesionalnog programera.

Možda ste mislili da je za profesionalnog progrmera prvo poslovno pravilo „da proradi“. Nadam da vam je ova knjiga do sada razjasnila tu zabludu. Funkcionalnost koju danas stvarate ima dobre šanse da se promeni u sledećem verzijama, ali čitljivost vašeg koda imaće dubok uticaj na sve promene koje će se ikada desiti. Stil kodiranja i čitljivost nastavljaju da utiču na održivost i proširivost koda dugo nakon što je originalna verzija promenjena do neprepoznatljivosti. Vaš stil i disciplina preživljavaju, čak i ako to nije slučaj sa vašim kodom.

Koje su bitne stvari u vezi formatiranja koje nam pomažu da najbolje komuniciramo?

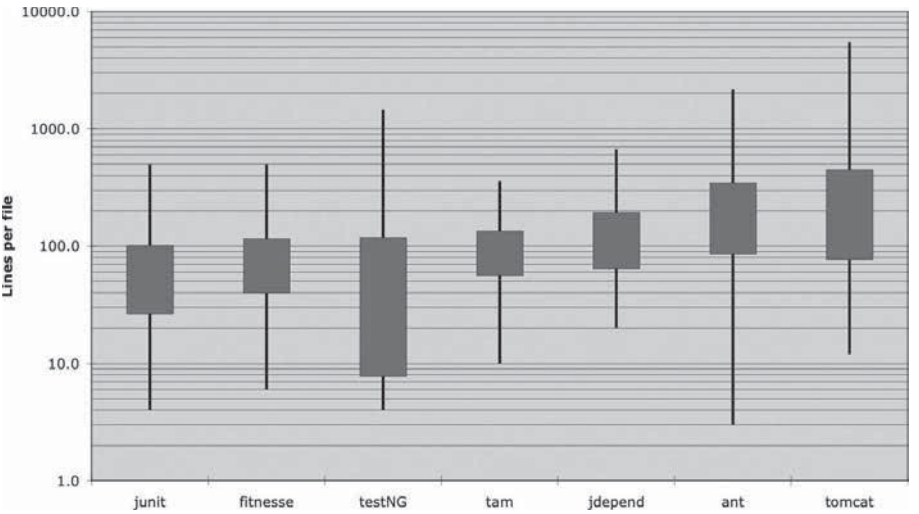
Vertikalno formatiranje

Krenimo od veličine po vertikali. Koliko bi trebalo da bude dugačak izvorni kod? U Javi je veličina koda usko povezana sa veličinom klase. Razgovaraćemo o veličini klasa kada budemo govorili o klasama. Za sada, razmotrimo samo veličinu.

Koliko je velika većina izvornih kodova pisanih u Javi? Otkriva se da postoji ogroman raspon veličina i neke izuzetne razlike u stilu. Slika 5-1 prikazuje neke od tih razlika.

Prikazano je sedam različitih projekata. Junit, FitNesse, testNG, Time and Money, JDepend, Ant i Tomcat. Linije kroz pravougaono polje prikazuju minimalnu i maksimalnu dužinu koda svakog projekta. Polje obuhvata otprilike jednu trećinu (jedna standardna devijacija)[1] dužine koda. Sredina polja je srednja vrednost. Dakle, prosečna veličina datoteke u projektu FitNesse iznosi oko 65 redova koda, a približno jedna trećina datoteka sadrži između 40 i više od 100 redova koda. Najveće polje u FitNesseu je oko 400 redova, a najmanje 6 redova. Imajte na umu da se radi o prikazu sa logaritamskom skalom, tako da mala razlika u poziciji po vertikali znači vrlo veliku razliku u apsolutnoj veličini.

1. Okvir prikazuje sigmu/2 iznad i ispod srednje vrednosti. Znam da raspodela dužina datoteke nije normalna, pa i standardna devijacija nije matematički precizna, ali ovde ne pokušavamo da budemo precizni. Pokušavamo samo da shvatimo o čemu se radi.



Slika 5-1 Raspodela dužina datoteke na LOG skali (visina pravougaonika = sigma)

Junit, FitNesse i Time and Money sastoje se od relativno malih datoteka. Nijedna nije veća od 500 redova, a većina njih je kraća od 200 redova. Tomcat i Ant, s druge strane, imaju neke datoteke koje su duge po nekoliko hiljada redova, a blizu polovine su sa više od 200 redova koda.

Šta nam to znači? Čini se da je moguće izgraditi značajne sisteme (FitNesse je blizu 50.000 redova) od datoteka koje su obično duge 200 redova, sa gornjom granicom koja iznosi 500 redova. Iako ovo ne bi trebalo da bude važno pravilo, trebalo bi ga smatrati vrlo poželjnim. Male datoteke je obično lakše razumeti nego velike.

Novinska metafora

Razmislite o dobro napisanom novinskom članku. Pročitali ste ga vertikalno. Na vrhu vas očekuje naslov koji će vam reći o čemu se radi i omogućava vam da odlučite da li je to nešto što želite da pročitate. Prvi pasus daje sažetak celog članka, skrivajući sve detalje dok vam daje široku sliku. Kako nastavljate da čitate dalje, detalji se pojavljuju sve dok ne dobijete uvid u sve datume, imena, citate, tvrdnje i druge sitnice.

Želeli bismo da izvorni kod bude poput novinskog članka. Naziv treba da bude jednostavan, ali objašnjavajući. Ime bi samo po sebi trebalo da bude dovoljno da nam kaže da li smo u pravom modulu ili ne. Delovi na vrhu izvornog koda treba da pružaju koncepte i algoritme visokog nivoa. Pojedinosti bi trebalo da se povećavaju kako nastavljamo sa čitanjem, dok na kraju ne pronađemo najniže nivoe funkcija i detalje izvornog koda.

Novine su sastavljene od puno članaka; većina njih su vrlo kratki. Neki su malo duži, ali vrlo malo njih je sa toliko teksta da bi mogli da zauzmu celu stranicu. Zbog toga su novine *upotrebljive*. Da su novine samo jedna duga priča koja sadrži neorganizovanu kolekciju činjenica, datuma i imena, mi ih jednostavno ne bismo čitali.

Vertikalna otvorenost između pojmova

Skoro ceo kod se čita s leva na desno i odozgo na dole. Svaki red koda predstavlja izraz ili uslov, a svaka grupa redova predstavlja potpunu misao. Te misli treba međusobno odvojiti praznim redovim.

Razmotrimo, na primer, listing 5-1. Postoje prazni redovi koji razdvajaju deklaraciju paketa, uvoz(e) i svaku od funkcija. Ovo krajnje jednostavno pravilo ima dubok uticaj na vizuelni izgled koda. Svaki prazan red je vizuelni znak koji identifikuje novi i zasebni koncept. Dok prolazite kroz listing, pogled vam se lepi na prvi red koji sledi iza praznog reda.

Listing 5-1
BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Izbacivanje ovih praznih redova, kao u listingu 5-2, ima izuzetan uticaj na čitljivost koda.

Listing 5-2
BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Ovaj utacaj je još izraženiji kada niste fokusirani. U prvom primeru nameću vam se različite grupe redova, dok drugi primer izgleda kao zbrka. Razlika između ova dva listinga je samo zbog vertikalne otvorenosti.

Vertikalna gustina

Ako otvorenost razdvaja koncepte, tada vertikalna gustina podrazumeva blisku povezanost. Dakle, redovi koda koji su usko povezani treba da izgledaju vertikalno gusti. Obratite pažnju kako beskorisni komentari u listingu 5-3 prekidaju blisku povezanost dve instance.

Listing 5-3
public class ReporterConfig {

```
  /**
   * The class name of the reporter listener
   */
  private String m_className;
  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();
  public void addProperty(Property property) {
    m_properties.add(property);
  }
```