Čist C++20

Obrasci održivog razvoja softvera i najbolja praksa

Prevod drugog izdanja

Stephan Roth

Apress®

ČIST C++ 20

Obrasci održivog razvoja softvera i najbolje prakse

Stephan Roth

Prevod II izdanja

kompjuter biblioteka

Apress®

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a, Beograd

Tel: 011/2520272

e-mail: kombib@gmail.com

internet: www.kombib.rs

Urednik: Mihailo J. Šolajić

Za izdavača, direktor:
Mihailo J. Šolajić

Autor: Stephan Roth

Prevod: Biljana Tešić

Lektura: Miloš Jevtović

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2021.

Broj knjige: 544

Izdanje: Prvo

ISBN: 978-86-7310-567-3

Clean C++20: Sustainable Software Development Patterns and Best Practices

Stephan Roth

ISBN 978-1-4842-5948-1

O AUTORU

Stephan Roth, rođen 15. maja 1969. godine, je kouč, savetnik i trener za sistemski i softverski inženjering u nemačkoj konsultantskoj kompaniji „oose Innovative Informatik eG“, čije je sedište u Hamburgu. Pre nego što se pridružio toj kompaniji, radio je dugi niz godina kao programer softvera, softverski arhitekta i sistemski inženjer u oblasti radio-izviđanja i šifrovanog sistema za komunikaciju. Razvio je sofisticirane aplikacije, posebno u sistemskom okruženju visokih performansi, i grafički korisnički interfejs koji koristi C++ i druge programske jezike. Stephan je takođe govornik na stručnim konferencijama i autor nekoliko publikacija. Kao član *Gesellschaft fur Sistems Engineering e.V.*, tj. nemačkog dela međunarodne organizacije za sistemski inženjering INCOSE, takođe je angažovan u zajednici Systems Engineering. Štaviše, on je aktivni pobornik pokreta Software Craftsmanship i bavi se principima i praksom razvoja čistog koda (CCD-om).

Stephan Roth živi sa suprugom Caroline i njihovim sinom Maximilianom u Bad Schwartau, banji u nemačkoj saveznoj državi Schleswig-Holstein, blizu Baltičkog mora.

Možete da posetite Stephanov veb sajt i blog o sistemskom inženjerstvu, softverskom inženjerstvu i izradi softvera pomoću URL-a roth-soft.de. Imajte na umu da su članci uglavnom napisani na nemačkom jeziku.

Osim toga, možete da kontaktirate sa njim pomoću e-pošte ili da ga pratite na ovde navedenim društvenim mrežama.

E-adresa: stephan@clean-cpp.com

Twitter: @_StephanRoth (https://twitter.com/_StephanRoth)

LinkedIn: www.linkedin.com/in/steproth

O tehničkom recenzentu

Marc Gregoire je softverski inženjer iz Belgije. Diplomirao je na Univerzitetu u Luvenu, u Belgiji, stekavši diplomu Burgerlijk ingenieur in de computer wetenschappen (koja je jednaka master studijama računarskog inženjerstva). Godinu dana kasnije je stekao diplomu *magistra* u oblasti veštačke inteligencije na istom univerzitetu. Nakon studija, Marc je počeo da radi u softverskoj konsultanskoj kompaniji „Ordina Belgium". Kao savetnik, radio je za firme „Siemens" i „Nokia Siemens Networks" na kritičnom 2G i 3G softveru pokrenutom na Solarisu za telekomunikacione operatere. To je zahtevalo rad u međunarodnim timovima koji se protežu od Južne Amerike i Sjedinjenih Američkih Država do Evrope, Bliskog istoka i Azije. Marc trenutno radi za „Nikon Metrology" na industrijskom softveru za 3D lasersko skeniranje.

Zahvalnice

Pisanje ovakve knjige nikada nije samo delo pojedinca, autora. Uvek postoje brojni, divni ljudi koji značajno doprinose ovako velikom projektu.

Prvi među tim ljudima bio je Steve Anglin iz izdavačke kuće „Apress“. On je stupio u kontakt sa mnom u martu 2016. godine, radi prvog izdanja knjige „Clean C++“. Nagovorio me je da nastavim svoj projekat sa kompanijom „Apress Media LLC“, koji je objavljen samo na Leanpubu. Platforma za samo izdavanje „Leanpub“ nekoliko godina je služila kao neka vrsta „inkubatora“, ali onda sam odlučio da knjigu završim i objavim uz pomoć izdavačke kuće „Apress“. Steve me je 2019. godine pitao da li želim da objavim drugo izdanje koje bi uzelo u obzir novi jezički standard C++ 20. Očigledno je bio prilično uspešan u nastojanjima da me ubedi da objavim drugo izdanje.

Zatim se zahvaljujem Marku Powersu, menadžeru uređivačkih radnji u „Apressu“, na velikoj podršci tokom pripreme rukopisa za oba izdanja. Mark nije samo uvek bio dostupan za odgovaranje na pitanja, već je njegovo neprestano praćenje napretka u pripremi rukopisa bilo pozitivan podsticaj za mene.

Osim Marku, veliko hvala i Matthew Moodieu, glavnom uredniku za razvoj u „Apressu“, koji mi je pružio odgovarajuću pomoć tokom čitavog procesa izdavanja ove knjige.

Posebnu zahvalnost upućujem mom tehničkom recenzentu Marcu Gregoireu. On je kritički ispitao svako poglavlje oba izdanja. Pronašao je mnoge probleme koje ja verovatno nikada ne bih pronašao. Snažno me je podstakao da poboljšam nekoliko odeljaka, a to mi je bilo zaista dragoceno.

Naravno, takođe se zahvaljujem celom produkcijskom timu u „Apressu“. Odlično su obavili posao koji se odnosi na finalizaciju (uređivanje kopija, indeksiranje, kompozicija/izgled, dizajn korica itd) cele knjige do distribucije konačnih datoteka za štampu (i e-knjige).

I na kraju, ali ne najmanje važno, zahvaljujem se svojoj voljenoj i jedinstvenoj porodici, posebno na razumevanju da projekat knjige oduzima mnogo vremena. Maximiliane i Carolina, baš ste divni!

Postanite član Kompjuter biblioteke

Kupovinom jedne naše knjige stekli ste pravo da postanete član Kompjuter biblioteke. Kao član možete da kupujete knjige u pretplati sa 40% popustai učestvujete u akcijama kada ostvarujete popuste na sva naša izdanja. Potrebno je samo da se prijavite preko formulara na našem sajtu. Link za prijavu: http://bit.ly/2TxekSa

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu

POGLAVLJE 1

Uvod

„Kako se to radi jednako je važno baš kao i to uraditi.“

—Eduardo Namur

Dragi čitaoci, predstavio sam prvo izdanje ove knjige rečima: „Još uvek je tužna stvarnost da su mnogi projekti za razvoj softvera u lošem stanju, a neki čak i u ozbiljnoj krizi.“ Bilo je to pre nešto više od tri godine i prilično sam siguran da opšta situacija od onda nije značajnije popravljena.

Razlozi zbog kojih mnogi projekti za razvoj softvera i dalje imaju poteškoća su višestruki. Postoji mnogo faktora rizika koji mogu prouzrokovati neuspeh projekata razvoja softvera. Na primer, neki projekti su „pogođeni“ lošim upravljanjem. U drugim projektima uslovi i zahtevi se stalno i brzo menjaju, ali proces razvoja ne podržava ovo visokodinamično okruženje. Štaviše, pronalaženju svih važnih zahteva i analizi slučaja upotrebe daje se malo prostora u nekim projektima. Konkretno, komunikacija između spoljnih zainteresovanih strana, poput stručnjaka za domene i programera, može biti teška, što dovodi do nesporazuma i razvoja nepotrebnih funkcija. I kao da sve ovo nije dovoljno loše, merama za osiguranje kvaliteta, poput testiranja, pridaje se premalo važnosti.

ZAINTERESOVANA STRANA

Termin zainteresovana strana u sistemskom i softverskom inženjerstvu obično se koristi za označavanje pojedinaca ili organizacija koji potencijalno mogu doprineti zahtevima razvojnog projekta ili koji definišu važna ograničenja za projekat.

Obično se razlikuju spoljne i unutrašnje zainteresovane strane. Primeri *spoljnih zainteresovanih strana* su kupci, svi korisnici sistema, stručnjaci za domene, administratori sistema, regulatorna tela, zakonodavci itd.

Unutrašnje zainteresovane strane su iz razvojne organizacije i mogu biti programeri i arhitekte softvera, poslovni analitičari, menadžeri proizvoda, inženjeri zahteva, odeljenje za osiguranje kvaliteta, marketinško osoblje itd.

Prethodno navedene tačke su sve tipični i dobro poznati problemi u profesionalnom razvoju softvera, ali, osim toga, postoji još jedna činjenica: **u nekim projektima baza koda je lošeg kvaliteta!**

To ne mora da znači da se kod ne izvršava pravilno. Njegov *spoljašnji kvalitet*, koji je izmerilo odeljenje za osiguranje kvaliteta (QA) pomoću integracionih testova ili testova prihvatanja, može biti prilično visok. Može proći QA bez prigovora, a u izveštaju o testiranju može se konstatovati da nije pronađeno ništa pogrešno. Korisnici softvera takođe mogu biti zadovoljni i srećni, a razvoj je možda čak završen na vreme i u okviru budžeta (što se retko događa). Čini se da je na prvi pogled sve u redu... Zaista, sve?

Ipak, *unutrašnji kvalitet* ovog koda, koji se izvršava pravilno, može biti veoma loš. Često je kod teško razumeti i on je komplikovan za održavanje i proširivanje. Nebrojene softverske jedinice, poput klasa ili funkcija, veoma su velike, neke od njih sa hiljadama linija koda, pa je izazov razumeti ih i čitati. Previše zavisnosti između softverskih jedinica dovodi do neželjenih efekata ako se nešto promeni. Softver nema uočljivu arhitekturu. Čini se da je njegova struktura nasumično nastala i neki programeri govore o „istorijski uzgojenom softveru“ ili „slučajnoj arhitekturi“. Klase, funkcije, promenljive i konstante imaju loše i tajanstvene nazive, a kod je prepun komentara: neki od njih su zastareli, neki samo opisuju očigledne „stvari“, a neki su, jednostavno, pogrešni. Programeri se plaše da nešto promene ili da prošire softver, jer znaju da je „truo“ i krhak i znaju da je pokrivenost jediničnim testovima slaba - ako uopšte postoje ti testovi. Upozorenje „Nikada ne dirajte sistem koji funkcioniše“ često se čuje od ljudi koji rade na takvim projektima. Za implementaciju nove funkcije nije potrebno samo nekoliko sati ili dana dok ne bude spremna za primenu, već je potrebno nekoliko nedelja ili, čak, meseci.

Ova vrsta lošeg softvera često se naziva *velika „lopta blata“*. Ovaj izraz su prvi put upotrebili 1997. godine Brian Foote i Joseph W. Yoder u svom radu za konferenciju Fourth Conference on Patterns Languages of Programs (PLoP ’97/EuroPLoP ’97). Foote i Yoder opisuju veliku „loptu blata“ kao „slučajno strukturisan, trom, traljav špageti kod koji se održava pomoću štapa i kanapa“. Takvi softverski sistemi su skupi i predstavljaju „noćne more“ kada je reč o održavanju i mogu razvojnu organizaciju da „bace na kolena“!

Upravo opisani patološki fenomeni mogu se naći u softverskim projektima u svim industrijskim sektorima i domenima. Programski jezik koji koriste nije važan. Naći ćete velike „lopte blata“ napisane na Javi, PHP-u, C-u, C#-u, C++-u i drugim manje ili više popularnim programskim jezicima. Zašto je to tako?

Softverska entropija

Pre svega, postoji prirodni zakon entropije ili nereda. Baš kao i svaki drugi zatvoreni i složeni sistem, softver vremenom postaje sve zamršeniji. Ova pojava naziva se *softverska entropija*. Termin je zasnovan na drugom zakonu termodinamike. U njemu se navodi da se nered zatvorenog sistema ne može smanjiti; može samo da ostane nepromenjen ili da se povećava. Izgleda da se softver ponaša na ovaj način. Uvek kada se doda nova funkcija ili kada se nešto promeni, kod postaje sve neuredniji. Postoje brojni faktori uticaja koji mogu doprineti softverskoj entropiji:

- Nerealni rasporedi projekata povećavaju pritisak i podstiču programere da kvare „stvari“ i da obavljaju svoj posao na loš i neprofesionalan način.
- ogromna složenost današnjeg softverskog sistema, kako tehnički, tako i kada je reč o zahtevima koje treba zadovoljiti
- programeri sa različitim nivoima veština i iskustvom
- globalno distribuirani, međukulturalni timovi, koji izazivaju problem komunikacije
- Razvoj se, uglavnom, fokusira na funkcionalne aspekte (funkcionalne zahteve i slučajeve korišćenja sistema) softvera, pri čemu se zahtevi za kvalitet (nefunkcionalni zahtevi), kao što su performanse, efikasnost, održivost, dostupnost, upotrebljivost, prenosivost, bezbednost itd, zanemaruju ili se, u najgorem slučaju, potpuno ignorišu.
- neprikladna razvojna okruženja i loše alatke
- Menadžment je fokusiran na brzu zaradu i ne razume vrednost održivog razvoja softvera.
- brzi i „prljavi“ hakovi i implementacije koje nisu u skladu sa dizajnom (*slomljeni prozori*)

TEORIJA SLOMLJENOG PROZORA

Teorija slomljenog prozora razvijena je u vezi sa američkim istraživanjima kriminala. Teorija ukazuje da jedan uništeni prozor na napuštenoj zgradi može biti okidač za uništenje čitavog naselja. Slomljeni prozor šalje fatalni signal okolini: „Vidi, nikoga nije briga za ovu zgradu!" To privlači dalje propadanje, vandalizam i druga nedruštvena ponašanja. Teorija slomljenog prozora korišćena je kao „temelj" za nekoliko reformi u kaznenoj politici, posebno za razvoj strategija nulte tolerancije.

U razvoju softvera ova teorija je preuzeta i primenjena je na kvalitet koda. Hakovi i loše implementacije koje nisu u skladu sa dizajnom softvera nazivaju se slomljeni prozori. Ako se ove loše implementacije ne poprave, u njihovom „susedstvu" može se pojaviti više hakova za bavljenje njima. Dakle, pokreće se uništenje koda.

Ne tolerišite slomljene prozore u kodu - *popravite ih*!

Zašto C++?

> *„C olakšava „pucanje sebi u nogu". C++ to otežava, ali kada pucate, onda ćete razneti celu nogu!"*
>
> Bjarne Stroustrup, Bjarne Stroustrup FAQ: Da li ste to zaista rekli?

Prvo i najvažnije, fenomeni poput softverske entropije, „mirisa" koda, anti-obrazaca i drugih problema koji se odnose na unutrašnji kvalitet softvera u osnovi su nezavisni od programskog jezika. Međutim, čini se da su C i C++ projekti posebno „skloni" neredu i obično prelaze u loše stanje. Čak je i World Wide Web pun loših, ali naizgled veoma brzih i visokooptimizovanih primera C++ koda. Često imaju „okrutnu" sintaksu koja u potpunosti zanemaruje elementarne principe dobrog dizajna i dobro napisanog koda. Zašto je to tako?

Jedan od razloga za ovo može biti činjenica da je C ++ programski jezik sa više paradigmi na srednjem nivou; odnosno, on sadrži jezičke funkcije na visokom i na niskom nivou. C++ je poput mesta koje spaja mnogo različitih ideja i koncepata. Pomoću ovog jezika možete pisati proceduralne, funkcionalne ili objektno-orijentisane programe, ili, pak, mešavinu sve te tri skupine jezika. Osim toga, C++ dozvoljava šablon metaprogramiranja (TMP), tj. tehniku u kojoj kompajler koristi takozvane šablone za generisanje privremenog izvornog koda koji se spaja sa ostatkom izvornog koda, a zatim kompajlira.

Od objavljivanja ISO standarda C++ 11 (ISO/IEC 14882: 2011 [ISO11]) u septembru 2011. godine, dodato je još više načina; na primer, funkcionalno programiranje pomoću anonimnih funkcija sada je na veoma elegantan način podržano lambda izrazima. Kao posledica ovih različitih mogućnosti, C++ ima reputaciju veoma složenog, komplikovanog i glomaznog jezika. Pojavom svakog standarda posle C++ 11 (C++ 14, C++ 17, a sada i C++ 20), dodato je mnogo novih funkcija koje su dodatno povećale složenost jezika.

Još jedan uzrok lošeg softvera mogao bi biti taj što mnogi programeri nisu imali znanje iz IT oblasti. Svako može započeti razvoj softvera u današnje vreme, bez obzira da li ima fakultetsku diplomu ili bilo koje drugo obrazovanje u računarstvu. Veliku većinu programera za C++ ne čine (ili nisu činili) stručnjaci. Naročito u tehnološkim oblastima, automobilskom saobraćaju, železničkom saobraćaju, vazduhoplovstvu, električnoj/elektronskoj industriji ili mašinskoj industriji mnogi inženjeri su ušli u programiranje tokom poslednjih decenija, a da nisu imali obrazovanje iz računarstva. Pošto se složenost povećavala, a tehnički sistemi su sadržavali sve više softvera, javila se hitna potreba za programerima. Ovu potražnju pokrivala je postojeća radna snaga. Inženjeri elektrotehnike, matematičari, fizičari i mnogi ljudi iz disciplina koje nisu tehničke počeli su da razvijaju softver. Naučili su da razvijaju softver uglavnom samoobrazovanjem i praktičnim radom. I činili su to onako kako najbolje znaju.

U osnovi, u ovome nema apsolutno ništa loše. Međutim, ponekad samo poznavanje alatki i sintakse programskog jezika nije dovoljno. Razvoj softvera nije isto što i programiranje. Svet je prepun softvera koje su kreirali nepravilno obučeni programeri. Mnogo je „stvari“ na apstraktnom nivou koje programer mora uzeti u obzir da bi kreirao održivi sistem - na primer, arhitekturu i dizajn. Kako sistem treba da bude strukturiran za postizanje određenih ciljeva kvaliteta? Za šta je dobra ova objektno-orijentisana „stvar“ i kako da je efikasno koristimo? Koje su prednosti i nedostaci određenog radnog okvira ili biblioteke? Koje su razlike između različitih algoritama i zašto se jedan algoritam ne uklapa u sve slične probleme? I šta je, do vraga, deterministički konačni automat i zašto on pomaže u suočavanju sa složenošću?!

Međutim, nema razloga da klonete duhom! Ono što je zaista važno za trajno „zdravlje“ softverskog programa je da neko brine o njemu, a čist kod je ključ!

Čist kod

Šta se tačno podrazumeva pod čistim kodom?

Glavni nesporazum je mešanje čistog koda sa nečim što se može nazvati lepi kod. Čist kod ne mora nužno biti lep (...šta god to značilo). Profesionalni programeri nisu plaćeni da pišu lepe ili divne kodove. Njih angažuju razvojne kompanije kao stručnjake za kreiranje vrednosti za kupca.

Kod je čist ako ga bilo koji član tima može lako razumeti i održavati.

Čist kod je osnova brzog koda. Ako je vaš kod čist i pokrivenost testom je velika, potrebno je samo nekoliko sati ili nekoliko dana da implementirate, da testirate i da primenite promenu ili novu funkciju - a ne nedelje ili meseci.

Čist kod je osnova za održivi softver; on održava projekat razvoja softvera pokrenut dugo vremena bez akumuliranja velike količine tehničkog duga. Programeri moraju aktivno voditi računa o softveru i osigurati da on ostane „u formi“, jer je kod presudan za opstanak organizacije za razvoj softvera.

Čist kod je takođe ključ za srećnijeg programera - doprinosi da njegov život bude bez stresa. Ako je vaš kod čist i jednostavan, možete biti mirni u svakoj situaciji, čak i kada se suočavate sa kratkim rokom za realizaciju projekta.

Sve ovde pomenute tačke su tačne, ali ključno je sledeće: ***čist kod štedi novac!*** U suštini, reč je o ekonomskoj efikasnosti. Svake godine razvojne organizacije izgube mnogo novca, jer je njihov kod u lošem stanju. Čist kod osigurava da vrednost koju dodaje razvojna organizacija ostane visoka. Kompanije mogu dugo zarađivati novac od svog čistog koda.

C++ 11: Početak nove ere

„Iznenađujuće, C++ 11 je kao novi jezik - delovi se jednostavno uklapaju bolje nego ranije i smatram da je stil programiranja na višem nivou prirodniji nego ranije i efikasniji nego ikada.“

Bjarne Stroustrup, C++11 - novi ISO C++ standard [Stroustrup16]

Posle objavljivanja jezičkog standarda C++ 11 (ISO/IEC 14882: 2011 [ISO11]) u septembru 2011. godine, neki ljudi su predvideli da će C++ doživeti preporod. Neki su čak govorili i o revoluciji.

Predvideli su da će se idiomatski stil na koji se odvija razvoj pomoću ovog „modernog C++ jezika“ znatno razlikovati i ne može se porediti sa stilom „istorijskog C++ jezika“ iz ranih devedesetih godina prošlog veka.

Bez sumnje, C++ 11 je doneo gomilu sjajnih inovacija i promenio način na koji razmišljamo o razvoju softvera u ovom programskom jeziku. Sa velikim samopouzdanjem mogu reći da je C++ 11 pokrenuo takve promene. U tom jeziku dobili smo semantiku premeštanja, lambda izraze, automatsko odbijanje tipova, izbrisane i zadate funkcije, mnoštvo poboljšanja standardne biblioteke i još mnogo korisnih „stvari".

Međutim, to je takođe značilo da su ove nove funkcije nastale povrh već postojećih. Nije moguće ukloniti značajnu funkciju iz jezika C++ bez deljenja velike količine postojećih baza koda. To znači da se složenost jezika povećala, jer je C++ 11 veći od svog prethodnika C++ 98 i samim tim je teže naučiti ovaj jezik u celini.

Njegov naslednik C++ 14 bio je evolutivni razvoj sa nekim ispravkama grešaka i manjim poboljšanjima. Ako planirate da pređete na moderni C++, trebalo bi bar da prvo koristite ovaj standard i preskočite C++ 11.

Tri godine kasnije, kada se pojavio C++ 17, dodate su brojne nove funkcije, ali je ovom revizijom uklonjeno i nekoliko funkcija. A u decembru 2020. godine odbor za standardizaciju C++ završio je i objavio novi standard C++ 20, koji neki ljudi nazivaju „sledeća velika stvar". Ovaj standard ponovo dodaje mnoštvo novih funkcija pored mnogih ekstenzija osnovnog jezika, standardne biblioteke i drugih „stvari", posebno takozvanu „veliku četvorku": koncepata, korutina, biblioteka opsega i modula.

Ako pogledamo razvoj C++ jezika tokom poslednjih 10 godina, možemo uočiti da se složenost jezika znatno povećala. U međuvremenu je već započeo razvoj C++ 23. Preispitujem se da li je ovo pravi način na koji će se dugoročno nešto raditi. Možda bi u nekom trenutku bilo prikladno ne samo trajno dodati funkcionalnosti, već i pregledati postojeće funkcije, konsolidovati ih i ponovo pojednostaviti jezik.

Kome je namenjena ova knjiga

Kao trener i konsultant, imao sam priliku da posetim mnoge kompanije koje razvijaju softver. Osim toga, veoma pažljivo posmatram šta se događa na sceni programera. I otkrio sam prazninu.

Moj je utisak da su C++ programeri ignorisali one koji promovišu izradu softvera i čisti razvoj koda. Mnogi principi i tehnike koji su relativno poznati u

Java okruženju i u modernom svetu Veba ili razvoja igara izgleda da su u velikoj meri nepoznati u C++ svetu.

Ovom knjigom pokušavamo malo da smanjimo tu prazninu, jer čak i pomoću jezika C++ programeri mogu da napišu čist kod! Ako želite da naučite da napišete bolji C++ kod, knjiga „Čist C++ 20" je upravo ono što vam treba. Napisana je za C++ programere svih nivoa veština i na primerima se u njoj pokazuje kako se može napisati razumljiv, fleksibilan, održiv i efikasan C++ kod. Čak i ako ste sezonski programer za C++, u ovoj knjizi postoje informacije i tačke podataka koje će vam biti korisne u vašem radu.

Ova knjiga nije „bukvar" za C++! Da biste efikasno koristili znanje iz ove knjige, trebalo bi da poznajete osnovne pojmove jezika. Ako samo želite da započnete razvoj jezika C++, a još uvek nemate osnovno znanje, prvo treba da naučite osnovne pojmove, što možete uraditi pomoću drugih knjiga ili pomoću dobrog uvodnog treninga za C++. U ovoj knjizi se ne razmatraju detaljno svaka pojedina nova C++ 20 jezička funkcija ili funkcije prethodnice. Kao što sam već istakao, složenost jezika je sada relativno visoka. Postoje i druge veoma dobre knjige u kojima je predstavljen ovaj jezik „od A do Ž".

Štaviše, ova knjiga ne sadrži nikakav ezoterični hak ili „zakrpu". Znam da je pomoću jezika C++ moguće uraditi mnogo ludih i zapanjujućih „stvari", ali one obično nisu u duhu čistog koda i ne bi ih trebalo koristiti za kreiranje čistog i modernog C++ programa. Ako ste zaista oduševljeni misterioznom C++ pokazivačkom „gimnastikom", ova knjiga nije za vas.

Osim toga, ova knjiga je napisana da pomogne C++ programerima svih nivoa veština. Namenjena je za C++ programere svih nivoa veština i na primerima je pokazano kako se može napisati razumljiv, fleksibilan, održiv i efikasan C++ kod. Predstavljeni principi i tehnike mogu se primeniti na nove softverske sisteme, koji se ponekad nazivaju i *greenfield projekti*, ali i na stare sisteme sa dugom istorijom, koji se često pežorativno nazivaju *brownfield projekti*.

NAPOMENA Imajte na umu da ne podržava svaki kompajler za C++ trenutno sve nove jezičke funkcije, posebno ne one iz najnovijeg standarda C++ 20.

Konvencije korišćene u ovoj knjizi

U ovoj knjizi se koriste sledeće tipografske konvencije:

Kurzivni font se koristi za uvođenje novih termina i naziva.

Podebljani font se koristi u pasusima da bi bili naglašeni pojmovi ili važni iskazi.

`Font fiksne širine` se koristi u pasusima da bi ukazao na programske elemente, kao što su nazivi klasa, promenljivih ili funkcija, iskazi i ključne reči jezika C++. Ovaj font se takođe koristi za prikaz unosa u komandnoj liniji, adrese veb sajta (URL-a), redosleda pritiska tastera ili rezultata programa.

Izdvojeni tekst

Ponekad prenosim male delove informacija koji su tangencijalno povezani sa sadržajem oko njih, pa se mogu smatrati odvojenim od tog sadržaja. Takvi delovi su poznati kao izdvojeni tekst. Ponekad koristim izdvojeni tekst da bih predstavio dodatnu ili kontrastnu diskusiju o temi koja je povezana sa tim tekstom.

OVO ZAGLAVLJE SADRŽI NASLOV STRANICE

Ovo je izdvojeni tekst.

Napomene, saveti i upozorenja

Druga vrsta izdvojenog teksta za posebne namene koristi se za napomene, savete i upozorenja. Koristi se za obezbeđivanje nekih posebnih informacija, za obezbeđivanje korisnih saveta ili za upozoravanje na „stvari“ koje mogu biti opasne i koje treba izbegavati.

NAPOMENA Ovo je tekst napomene.

Uzorci koda

Primeri koda i isečci koda pojavljuju se odvojeno od teksta; istaknuti su sintaksom (ključne reči jezika C++ su podebljane) i napisani su fontom fiksne širine. Duži odeljci koda obično imaju numerisane naslove. Da bi u tekstu bilo ukazano na određene linije primera koda, uzorci koda ponekad uključuju brojeve linija (pogledajte listing 1-1).

LISTING 1-1 Uzorak koda sa numerisanim linijama

```
1 class Clazz {
2 public:
3 Clazz();
4 virtual ~Clazz();
5 void doSomething();
6
7 private:
8  int _attribute; 09
10  void function();
11};
```

Da biste se bolje fokusirali na određene aspekte koda, nevažni delovi su ponekad skriveni i predstavljeni komentarom sa tri tačke (...), kao u sledećem primeru:

```
void Clazz::function() {
  // ...
}
```

Stil kodiranja

Sada sledi nekoliko reči o stilu kodiranja koji koristim u ovoj knjizi.

Možda ćete steći utisak da moj stil programiranja umnogome liči na tipični Java kod, pomešan sa stilom Kernighan and Ritchie (K&R). Radio sam skoro 20 godina kao softverski programer, a čak i kasnije u svojoj karijeri naučio sam druge programske jezike (na primer, ANSI-C, Java, Delphi, Scala i nekoliko skriptnih jezika). Stoga sam usvojio sopstveni stil programiranja, koji je mešavina ovih različitih uticaja.

Možda vam se moj stil neće svideti, već više volite Kernel stil Linusa Torvalda, Allman stil ili bilo koji drugi popularni C++ standard kodiranja. Ovo je naravno sasvim u redu. Ja volim svoj stil, a vi svoj.

C++ osnovne smernice

Možda ste čuli za *C++ osnovne smernice*, koje se nalaze na adresi https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html. Ovo je kolekcija smernica, pravila i dobrih tehnika za programiranje pomoću modernog jezika C++. Projekat je smešten na GitHubu i objavljen je pod licencom u MIT stilu. Pokrenuo ga je Bjarne Stroustrup, ali ima mnogo više urednika i saradnika, kao što je Herb Sutter.

Broj pravila i preporuka u *C++ osnovnim smernicama* je prilično veliki. Trenutno postoji 30 pravila samo za interfejs, približno isti broj za upravljanje greškama i ne manje od 55 pravila za funkcije. I to nikako nije „kraj priče". Postoje dodatne smernice za klase, upravljanje resursima, performanse i šablone.

Prvo sam imao ideju da povežem teme u ovoj knjizi sa pravilima iz *C++ osnovnih smernica*. Međutim, to bi dovelo do bezbroj referenci za smernice i čak bi moglo smanjiti čitljivost knjige, pa sam se uglavnom uzdržao od toga, ali bih želeo da u ovom trenutku izričito preporučim *osnovne smernice za C++*. One su veoma dobar dodatak ovoj knjizi, iako ne podržavam sva pravila.

Prateći veb sajt i spremište izvornog koda

Ova knjiga ima prateći veb sajt - www.clean-cpp.com.

Veb sajt sadrži:

- diskusiju o dodatnim temama koje nisu razmatrane u ovoj knjizi
- verzije svih slika u ovoj knjizi u visokoj rezoluciji

Neki od primera izvornog koda u ovoj knjizi i drugi korisni dodaci dostupni su na GitHubu na adresi:

https://github.com/Apress/clean-cpp20

Kod možete pogledati pomoću Gita, koristeći sledeću komandu:

```
$> git clone https://github.com/clean-cpp/book-samples.git
```

Arhivu koda u .ZIP formatu možete pribaviti ako posetite adresu https://github.com/clean-cpp/book-samples i kliknete na dugme Download ZIP.

UML dijagrami

Neke ilustracije u ovoj knjizi su UML dijagrami. *Objedinjeni jezik za modelovanje (UML - UnifiedModelingLanguage)* je standardizovani grafički jezik koji se koristi za kreiranje modela softvera i drugih sistema. U svojoj trenutnoj verziji 2.5.1 UML sadrži 15 tipova dijagrama koji u potpunosti opisuju sistem.

Ne brinite ako ne poznajete sve tipove dijagrama; u ovoj knjizi koristim samo nekoliko njih. Povremeno predstavljam UML dijagrame da bih obezbedio kratak pregled određenih problema koji se možda ne mogu razumeti brzim čitanjem koda. Dodatak A sadrži kratak pregled korišćenih UML notacija.

POGLAVLJE 2

Izgradite bezbednosnu mrežu

„Testiranje je veština. Iako to može iznenaditi neke ljude, jednostavna je činjenica."

Mark Fewster i Dorothy Graham,
Software Test Automation, 1999.

To što glavni deo ove knjige započinjem poglavljem o testiranju može iznenaditi neke čitaoce, ali to činim zbog nekoliko dobrih razloga. Tokom poslednjih nekoliko godina testiranje na određenim nivoima postalo je osnovni temelj modernog razvoja softvera. Potencijalne koristi dobre strategije testiranja su ogromne. Sve vrste testova, ako su dobro pripremljene, mogu biti od pomoći i koristi. U ovom poglavlju opisujem zašto mislim da su posebno jedinični testovi neophodni da bi se osigurao osnovni nivo visokog kvaliteta softvera.

Imajte na umu da je ovo poglavlje posvećeno onome što se ponekad naziva „staro dobro jedinično testiranje" (POUT - plain old unit testing), a ne o dizajnu koji podržava razvoj vođen testovima (TDD), a koji ćemo razmotriti u Poglavlju 8.

Potreba za testiranjem

1962: NASA MARINER 1

Svemirska letelica Mariner 1 lansirana je u svemir 22. jula 1962. godine u sklopu misije obletanja Venere za planetarno istraživanje. Zbog problema na anteni za usmeravanje, raketa za lansiranje Atlas-Agena B radila je nepouzdano i izgubila je svoj upravljački signal sa zemaljske kontrole ubrzo nakon lansiranja.

Ovaj izuzetan slučaj razmatran je tokom projektovanja i konstrukcije rakete. Lansirno vozilo Atlas-Agena prebačeno je na automatsko upravljanje pomoću računara za navođenje. Nažalost, greška u softveru tog računara dovela je do netačnih komandi upravljanja koje su

prouzrokovale kritično odstupanje kursa i onemogućile upravljanje. Raketa je bila usmerena ka Zemlji i ukazivala je na kritično područje.

Na T+ 293 sekunde1, bezbednosni oficir poslao je destruktivnu komandu za uništavanje rakete. U NASA izveštaju o ispitivanju kao uzrok pominje se greška pri kucanju u izvornom kodu računara, tj. nedostatak crtice (-). Ukupan gubitak iznosio je 18,5 miliona dolara, što je u to vreme predstavljalo ogroman novac.

Ako se programeri softvera pitaju zašto su testovi dobri i neophodni, pretpostavljam da bi najčešći odgovor bio zbog smanjenja grešaka ili mana. Nema sumnje da je to u suštini tačno: testiranje je osnovni deo osiguranja kvaliteta.

Softverske greške se, obično, doživljavaju kao neprijatna smetnja. Korisnike iritira pogrešno ponašanje programa koji proizvodi nevažeće rezultate ili ih nervira redovno otkazivanje rada programa. Ponekad su, čak, i neke sitnice, poput skraćenog teksta u okviru za dijalog korisničkog interfejsa, dovoljne da korisnike softvera značajno ometaju u svakodnevnom radu. Posledica može biti sve veće nezadovoljstvo softverom, a u najgorem slučaju njegova zamena drugim proizvodom. Osim što se može desiti finansijski gubitak, imidž proizvođača softvera „pati" i od grešaka. U najgorem slučaju kompanija ulazi u ozbiljne probleme i mnogi poslovi će biti izgubljeni.

Međutim, prethodno opisani scenario se ne odnosi na svaki softver. Implikacije grešaka mogu biti mnogo dramatičnije.

1986: KATASTROFALNE POSLEDICE MEDICINSKOG AKCELERATORA THERAC-25

Ovo je slučaj sa verovatno najviše posledica u istoriji razvoja softvera. Therac-25 je bio uređaj za zračenje. Razvilo ga je i proizvodilo državno preduzeće „Atomic Energy of Canada Limited (AECL)" od 1982. do 1985. godine. Jedanaest uređaja proizvedeno je i instalirano u klinikama u Sjedinjenim Američkim Državama i Kanadi.

Zbog grešaka u kontrolnom softveru, nedovoljnog procesa osiguranja kvaliteta i drugih nedostataka, tri pacijenta su izgubila život usled previsokih doza zračenja. Tri pacijenta su pri zračenju pretrpela trajna teška zdravstvena oštećenja.

Analizom ovog slučaja utvrđeno je da je, između ostalog, softver napisala samo jedna osoba, koja je bila odgovorna i za testove.

1 NASA National Space Science Data Center (NSSDC): Mariner 1, http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1, preuzeto 2021-0305.

Kada ljudi misle na računare, obično imaju na umu lični računar, prenosivi računar, tablični računar ili pametni telefon. A ako razmišljaju o softveru, obično razmišljaju o veb prodavnicama, kancelarijskim paketima ili poslovnim IT sistemima.

Međutim, ove vrste softvera i računara čine veoma mali procenat svih sistema sa kojima imamo svakodnevni kontakt. Većina softvera koji nas okružuju kontroliše mašine koje fizički komuniciraju sa svetom. Čitavim našim životom upravlja softver. Ukratko: **bez softvera danas nema života!** Softver je svuda i važan je deo naše infrastrukture.

Ako se vozimo liftom, naš život je u rukama softvera. Avionima se upravlja pomoću softvera, a čitav sistem kontrole letenja širom sveta zavisi od softvera. Savremeni automobili sadrže značajnu količinu malih računarskih sistema (sa softverom koji komunicira pomoću mreže), koji su odgovorni za mnoge bezbednosno ključne funkcije vozila. Bez obzira na to šta radimo, uvek dolazimo u kontakt sa softverom, koji sadrže klima uređaji, automatska vrata, medicinski uređaji, vozovi, automatizovane proizvodne linije u fabrikama.... Zahvaljujući *digitalnoj revoluciji* i *Internet stvarima (IoT-u)*, značaj softvera u našim životima će se znatno povećati. Ova činjenica ne može postati očiglednija nego u autonomnom automobilu (bez vozača).

Nepotrebno je naglašavati da bi bilo koja greška u ovim softverski intenzivnim sistemima mogla imati katastrofalne posledice. Greška ili kvar važnog sistema može predstavljati pretnju životu ili fizičkom stanju. U najgorem slučaju, stotine ljudi bi moglo izgubiti život tokom avionske nesreće, verovatno prouzrokovane pogrešnim iskazom `if` u potprogramu podsistema Fly-by-Wire. O ovakvim sistemima se ni pod kojim uslovima ne može pregovarati. **Nikada!**

Međutim, čak i u sistemima bez funkcionalnih bezbednosnih zahteva, greške mogu imati ozbiljne implikacije, posebno ako su suptilnije u svojoj destruktivnosti. Lako je zamisliti da bi greške u finansijskom softveru mogle da izazovu svetsku bankarsku krizu. Zamislite da je finansijski softver arbitrarne velike banke dva puta dovršio svako knjiženje zbog greške, a da ovaj problem nije primećen nekoliko dana.

1990: KRAH AT&T-a

Telefonska mreža AT&T se „srušila“ 15. januara 1990. godine i 75 miliona telefonskih poziva nije uspostavljeno tokom narednih devet sati. Otkazivanje rada je prouzrokovano jednom linijom koda (pogrešnim iskazom `break`) u nadgradnji softvera koje je kompanija AT&T primenila na svih 114 svojih računarskih elektronskih prekidača (4ESS) u decembru 1989. godine. Problem je počeo da se ispoljava u toku popodneva 15. januara, kada je kvar u AT&T-ovom kontrolnom centru na Menhetnu doveo do lančane reakcije i onemogućio funkcionisanje prekidača na polovini mreže.

Procenjeni gubitak za AT&T bio je 60 miliona dolara. Takođe su verovatno ogromne gubitke pretrpela i preduzeća koja su se oslanjala na tu telefonsku mrežu.

Uvod u testiranje

U projektima za razvoj softvera postoje različiti nivoi mera za osiguranje kvaliteta. Ovi nivoi se često vizuelizuju u obliku piramide - takozvane *piramide testiranja*. Osnovni koncept razvio je američki programer Mike Cohn, jedan od osnivača Scrum alijanse. Piramidu automatizacije testiranja opisao je u svojoj knjizi „Succeeding with Agile“ [Cohn09]. Pomoću piramide, Cohn opisuje stepen automatizacije neophodan za efikasno testiranje softvera. Narednih godina piramidu testiranja su dalje razvijali različiti ljudi. Ona koja je prikazana na slici 2-1 je moja verzija.

Slika 2-1 Piramida testiranja

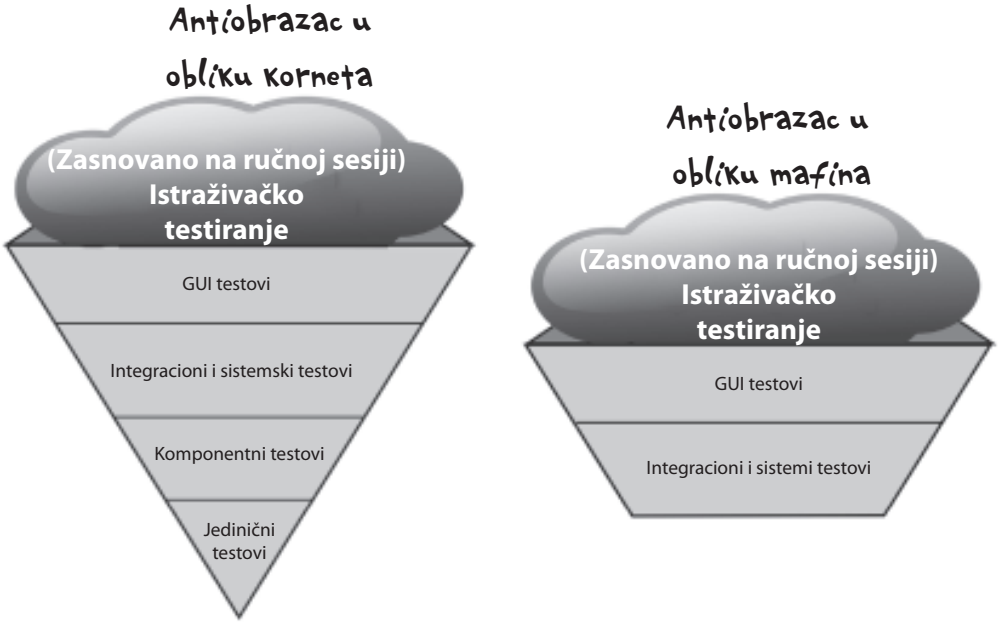
Oblik piramide, naravno, nije slučajnost. Poruka iza toga je da bi trebalo da imate mnogo više jediničnih testova niskog nivoa (približno 100 odsto pokrivenosti koda) od bilo koje druge vrste testova. Zašto?

Iskustvo je pokazalo da se ukupni troškovi sprovođenja i održavanja testova povećavaju prema vrhu piramide. Veliki sistemski testovi i ručni testovi prihvatanja korisnika su, obično, složeni, jer često zahtevaju opsežnu organizaciju i ne mogu se lako automatizovati. Na primer, teško je napisati automatizovani test korisničkog interfejsa, koji je ne retko krhak i relativno spor. Stoga se ovi testovi često izvode ručno, što je pogodno za odobrenje kupca (testove prihvatanja) i redovne istraživačke testove koje vrši QA odeljenje, ali previše dugotrajno i skupo za svakodnevnu upotrebu tokom razvoja softvera.

Osim toga, veliki sistemski testovi ili testovi koji su vođeni korisničkim interfejsom potpuno su neprikladni za proveru svih mogućih putanja izvršavanja u celom sistemu. Mnogo je koda u softverskom sistemu koji se bavi alternativnim putanjama, upravljanjem izuzecima i greškama, nadležnostima unakrsnog preseka (bezbednošću,upravljanjem transakcijama, evidentiranjem...) i drugim pomoćnim funkcijama koje su potrebne, ali do kojih se često ne može doći pomoću uobičajenog korisničkog interfejsa.

Povrh svega, ako test na nivou sistema nije uspešan, tačan uzrok greške je teško locirati. Sistemski testovi se obično zasnivaju na slučajevima korišćenja sistema. Tokom korišćenja uključene su mnoge komponente. To znači da se izvršava mnogo stotina ili, čak, i hiljada linija koda. Koja je od tih linija odgovorna za neuspešan test? Na ovo pitanje se često ne može lako odgovoriti i zahteva dugotrajnu i skupu analizu.

Nažalost, u nekoliko projekata razvoja softvera naći ćete degenerisane piramide testiranja, kao što je prikazano na slici 2-2. U takvim projektima se ulaže ogroman napor u testove na višem nivou, dok se osnovni testovi zanemaruju (*antiobrazac u obliku korneta*). U krajnjem slučaju, jedinični testovi u potpunosti nedostaju (*antiobrazac u obliku mafina*).



Slika 2-2 Degenerisana piramida testiranja (antiobrasci)

Stoga, široka baza jeftinih, dobro izrađenih, veoma brzih, redovno održavanih i potpuno automatizovanih jediničnih testova, koji su podržani izborom korisnih komponentnih testova, može biti čvrsta osnova za osiguranje prilično visokog kvaliteta softverskog sistema.

Jedinični testovi

„Refaktorisanje bez testova nije refaktorisanje, već samo pomeranje 'stvari' naokolo."

Corey Haines (@coreyhaines), 20. decembar 2013, na Twitteru

Jedinični test je deo koda koji izvršava mali deo baze proizvodnih kodova u određenom kontekstu. Test će vam pokazati u deliću sekunde da vaš kod funkcioniše onako kako očekujete. Ako je pokrivenost jediničnim testom prilično velika i za manje od minuta možete proveriti da li svi delovi vašeg sistema u razvoju funkcionišu pravilno, to će imati brojne prednosti:

- Brojna ispitivanja i studije dokazale su da je otklanjanje grešaka nakon isporuke softvera mnogo skuplje od izvršenja jediničnih testova.
- Jedinični testovi obezbeđuju trenutne povratne informacije o vašoj celokupnoj bazi koda. Pod uslovom da je pokrivenost testovima dovoljno velika (približno 100 odsto), u roku od samo nekoliko sekundi programeri znaju da li kod funkcioniše pravilno.
- Jedinični testovi omogućavaju programerima da prepravljaju svoj kod, bez straha da će učiniti nešto pogrešno što bi pokvarilo kod. U stvari, strukturalna promena u osnovi koda bez bezbednosne mreže jediničnih testova jeste opasna i ne bi je trebalo nazvati refaktorisanje.
- Velika pokrivenost jediničnim testovima može sprečiti dugotrajne i frustrirajuće sesije otklanjanja grešaka. Često jednočasovna traženja uzroka greške pomoću debagera mogu se dramatično skratiti. Naravno, nikada nećete moći da potpuno eliminišete upotrebu debagera. Ova alatka se i dalje može koristiti za analizu suptilnih problema ili za pronalaženje uzroka neuspelog jediničnog testa. Međutim, debager više neće biti ključna alatka za programere koja osigurava kvalitet koda.
- Jedinični testovi su vrsta izvršive dokumentacije, jer pokazuju tačno kako je kod dizajniran za upotrebu.
- Jedinični testovi mogu lako otkriti regresije; odnosno, mogu odmah da prikažu „stvari" koje su nekad funkcionisale, ali su neočekivano prestale da rade nakon izvršene promene.

- Jedinično testiranje podstiče kreiranje čistih i dobro oblikovanih interfejsa. Može pomoći da se izbegnu neželjene zavisnosti između jedinica. *Dizajn za testiranje* je takođe dobar *dizajn za upotrebljivost*, tj. ako se deo koda može lako dodati na uređaj za ispitivanje, onda se obično može i integrisati sa manje napora u proizvodni kod sistema.
- Jedinično testiranje ubrzava razvoj softvera.

Izgleda da je poslednja stavka na ovom spisku paradoksalna i zahteva malo objašnjenja. Jedinično testiranje pomaže razvoju da napreduje brže - kako je to moguće? To ne izgleda logično.

Nema sumnje: pisanje jediničnih testova zahteva napor. Prvo i najvažnije, menadžeri samo vide taj napor i ne razumeju zašto programeri treba da ulože vreme u ove testove. Posebno tokom početne faze projekta pozitivan efekat jediničnog testiranja na brzinu razvoja možda neće biti vidljiv. U ovim ranim fazama, kada je složenost sistema relativno mala i uglavnom sve funkcioniše, čini se da je za pisanje jediničnih testova u početku potrebno samo da se potrudite. Međutim, vremena se menjaju...

Kada sistem postaje sve veći i veći (sa više od 100.000 linija koda) i kada se složenost poveća, postaje sve teže razumeti i verifikovati sistem (setite se softverske entropije opisane u Poglavlju 1). Kada mnogi programeri iz različitih timova rade na ogromnom sistemu, svakodnevno se susreću sa kodom koji pišu drugi programeri. Bez jediničnih testova ovo može postati veoma frustrirajući posao. Siguran sam da svi znaju za te besmislene, beskrajne sesije debagovanja i „hodaju“ kroz kod u režimu jednog koraka tokom analiziranja vrednosti promenljivih iznova i iznova i iznova... Ovo je ogromno gubljenje vremena i znatno usporava brzinu razvoja.

Naročito u srednjim i kasnim fazama razvoja i u fazi održavanja nakon isporuke proizvoda dobri jedinični testovi postaju veoma dragoceni. Najveća ušteda vremena od jediničnog testiranja sledi nekoliko meseci ili nekoliko godina nakon pisanja testa, kada jedinicu ili njen API treba promeniti ili proširiti.

Ako je pokrivenost testom velika, skoro je nevažno da li je deo koda koji uređuje programer napisao on sam ili neki drugi programer. Dobri jedinični testovi pomažu programerima da brzo razumeju deo koda koji je napisala druga osoba, čak i ako je napisan pre tri godine. Ako test nije uspešan, on tačno pokazuje gde je ponašanje prekinuto. Programeri mogu da veruju da sve i dalje funkcioniše pravilno ako svi testovi budu uspešni. Dugotrajne i dosadne sesije debagovanja postaju retkost, a debager služi uglavnom za brzo pronalaženje uzroka neuspelog testa ako taj uzrok nije očigledan, što je odlično, jer je zabavno raditi na taj način. Takav rad je motivišući i dovodi do bržih i boljih rezultata.

Programeri će imati veće poverenje u bazu koda i biće im jednostavna. Da li mogu menjati zahteve ili zahteve za nove funkcije? Mogu, jer programeri mogu novi proizvod da isporuče brzo, često i odličnog kvaliteta.

RADNI OKVIRI ZA JEDINIČNO TESTIRANJE

Za C++ razvoj dostupno je nekoliko različitih radnih okvira za jedinično testiranje - na primer, CppUnit, Boost.Test, CUTE, Google Test i Catch, odnosno Catch2.

U principu, svi ovi radni okviri prate osnovni dizajn takozvanog *xUnita*, što je zajednički naziv za nekoliko radnih okvira za jedinično testiranje koji svoju strukturu i funkcionalnost izvode iz Smalltalkovog *SUnita*. Osim što sadržaj ovog poglavlja nije fokusiran na određeni radni okvir za jedinično testiranje, jer je njegov sadržaj primenljiv na generalno jedinično testiranje, potpuno i detaljno poređenje svih dostupnih radnih okvira neće biti razmatrano u ovoj knjizi. Dalje, odabir odgovarajućeg radnog okvira zavisi od mnogo faktora. Na primer, ako vam je veoma važno da možete brzo da dodate nove testove uz minimalnu količinu posla, to može biti „izbacivački" kriterijum za određene radne okvire.

Šta je sa QA?

Programer bi mogao imati sledeći stav: „Zašto treba da testiram svoj softver? Imamo ispitivače i odeljenje za kontrolu kvaliteta - to je njihov posao."

Suštinsko pitanje je da li je kvalitet softvera isključiva briga odeljenja za osiguranje kvaliteta.

Jednostavan i jasan odgovor: **Nije!**

Bilo bi krajnje neprofesionalno predati deo softvera QA odeljenju, ako znate da on sadrži greške. Profesionalni programeri nikada ne prebacuju odgovornost za kvalitet sistema na druga odeljenja. Suprotno tome, profesionalni programeri softvera grade produktivna partnerstva sa ljudima iz QA odeljenja. Trebalo bi blisko da sarađuju i da se dopunjuju.

Naravno, veoma je ambiciozan cilj isporučiti softver koji nema ni jednu grešku. Povremeno, QA odeljenje pronađe nešto pogrešno. I to je dobro. QA odeljenje je naša druga bezbednosna mreža. Ono proverava da li su prethodne mere osiguranja kvaliteta bile dovoljne i efikasne.

Možemo učiti na svojim greškama. Profesionalni programeri odmah popravljaju te nedostatke kvaliteta otklanjanjem grešaka koje je pronašao QA i pisanjem automatizovanih jediničnih testova da bi takve greške bile pronađene u budućnosti. Onda bi trebalo pažljivo razmisliti kako se moglo dogoditi da je previđen ovaj problem. Rezultat ove retrospektive trebalo bi da posluži kao povratna informacija za poboljšanje razvojnog procesa.

Pravila za dobre jedinične testove

Video sam mnogo jediničnih testova koji su prilično beskorisni. Jedinični testovi trebalo bi da dodaju vrednost vašem projektu. Da biste postigli ovaj cilj, morate slediti neka osnovna pravila koja sam opisao u ovom odeljku.

Kvalitet koda za testiranje

Isti visokokvalitetni zahtevi za proizvodni kod moraju da važe i za jedinični kod za testiranje. U idealnom slučaju, ne bi trebalo praviti razliku između proizvodnog koda i koda za testiranje - oni su jednaki. Ako kažemo da postoji proizvodni kod sa jedne strane i kod za testiranje sa druge strane, razdvajamo „stvari“ koje pripadaju jedna drugoj. Ne radite to!

Razmišljanje o proizvodnom kodu i kodu za testiranje kao o dve kategorije postavlja osnovu za zanemarivanje testova kasnije u projektu.

Imenovanje jediničnog testa

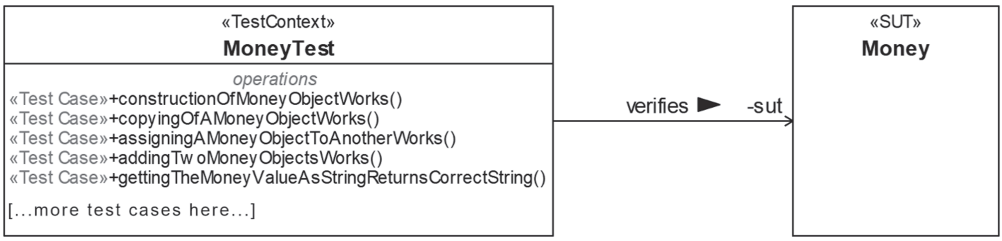
Ako jedinično testiranje nije uspešno, programer želi odmah da zna:

- Kako se zove jedinica čiji test nije bio uspešan?
- Šta je testirano i kakvo je okruženje testa (scenario za testiranje)?
- Koji je bio očekivani rezultat testa, a koji je bio stvarni rezultat neuspelog testa?

Stoga je ekspresivni i opisni standard imenovanja za vaše jedinične testove veoma važan. Moj savet je da uspostavite standarde imenovanja za sve testove.

Pre svega, dobra je praksa imenovati modul za jedinično testiranje (u zavisnosti od radnog okvira za jedinično testiranje, ti moduli se nazivaju *oprema za ispitivanje* ili *uređaj za ispitivanje*) na takav način da se iz njega može lako izvesti jedinični test.

Jedinični testovi bi trebalo da imaju naziv, kao što je `<Unit_under_Test>Test`, pri čemu se čuvar mesta `<Unit_under_Test>` zamenjuje nazivom testa. Na primer, ako je vaš sistem koji se testira jedinica (SUT) `Money`, odgovarajući uređaj za ispitivanje koji se pripaja toj jedinici i sadrži sve slučajeve jediničnog testiranja treba da se zove `MoneyTest` (pogledajte sliku 2-3).



Slika 2-3 Sistem koji se testira, jedinica (SUT) Money i uređaj za ispitivanje MoneyTest

Osim toga, jedinični testovi moraju imati ekspresivne i opisne nazive. Nije od pomoći ako jedinični testovi imaju besmislene nazive, kao što su `testConstructor()`, `test4391()` ili `sumTest()`. Evo dva predloga za pronalaženje dobrih naziva za njih.

Generalno, možete upotrebiti višenamenske klase koje se mogu koristiti u različitim kontekstima, a ekspresivni naziv može sadržati sledeće delove:

- preduslov scenarija za testiranje, odnosno stanje SUT-a pre izvršenja testa
- testirani deo jedinice koja se testira, obično naziv testirane procedure, funkcije ili metoda (API-a)
- očekivani rezultat testa

To dovodi do šablona naziva za procedure/metode jediničnog testa poput ovog:

```
<PreconditionAndStateOfUnitUnderTest>_<TestedPartOfAPI>_<ExpectedBehavior>
```

Listing 2-1 sadrži nekoliko primera.

LISTING 2-1 Primeri dobrih i ekspresivnih naziva jediničnih testova

```
void CustomerCacheTest::cacheIsEmpty_addElement_sizeIsOne();

void CustomerCacheTest::cacheContainsOneElement_removeElement_sizeIsZero();

void ComplexNumberCalculatorTest::givenTwoComplexNumbers_add_Works();

void MoneyTest:: givenTwoMoneyObjectsWithDifferentBalance_Inequality Comparison_
Works();
```

```
void MoneyTest::createMoneyObjectWithParameter_getBalanceAsString_
returnsCorrectString();

void InvoiceTest::invoiceIsReadyForAccounting_getInvoiceDate_returnsToday();
```

Drugi mogući pristup za izradu ekspresivnih naziva jediničnih testova je manifestovanje specifičnog zahteva u nazivima. Ovi nazivi obično odražavaju zahteve domena aplikacije. Na primer, oni mogu biti izvedeni iz zahteva zainteresovanih strana. Pogledajte listing 2-2.

LISTING 2-2 Još nekoliko primera naziva jediničnih testova koji verifikuju zahteve specifične za domen

```
void UserAccountTest::creatingNewAccountWithExisting
EmailAddressThrowsException();

void ChessEngineTest::aPawnCanNotMoveBackwards();

void ChessEngineTest::aCastlingIsNotAllowedIfInvolvedKingHasBeenMovedBefore();
void ChessEngineTest::aCastlingIsNotAllowedIfInvolvedRookHasBeenMovedBefore();
void HeaterControlTest::ifWaterTemperatureIsGreaterThan92DegTurnHeaterOff();
void BookInventoryTest::aBookThatIsInTheInventoryCanBeBorrowedByAuthorized
People();

void BookInventoryTest::aBookThatIsAlreadyBorrowedCanNotBeBorrowedTwice();
```

Dok budete čitali ove nazive metoda testiranja, trebalo bi da vam postane jasno da se, čak i ako implementacija testova i metodi testiranja nisu prikazani ovde, može lako doći do mnogo korisnih informacija. To je takođe velika prednost ako takav test ne bude uspešan. Svi poznati radni okviri za jedinično testiranje ispisuju naziv neuspelog testa pomoću `stdouta` na interfejsu komandne linije ili ga prikazuju u posebnom izlaznom prozoru IDE-a. Dakle, lociranje grešaka je u velikoj meri olakšano.

Nezavisnost jediničnih testova

Svaki jedinični test mora biti nezavisan od svih ostalih. Bilo bi kobno kada bi se testovi morali izvoditi po određenom redosledu, zato što se jedan test zasnivao na rezultatima prethodnog. Nikada nemojte pisati jedinični test čiji je rezultat preduslov za naredni test. Nikada ne ostavljajte jedinicu za testiranje u izmenjenom stanju, jer je to preduslov za sledeće testove.

Glavne probleme mogu da izazovu globalna stanja - na primer, upotreba Singletona ili statičnih članova u jedinici koja se testira. Ne samo da Singletoni povećavaju spregu između softverskih jedinica, već takođe često održavaju globalno stanje koje „zaobilazi" nezavisnost jediničnog testiranja. Na primer, ako je određeno globalno stanje preduslov za uspešan test, ali je prethodni test promenio to globalno stanje, mogu nastati ozbiljni problemi.

Naročito u starijim sistemima, koji su često zatrpani Singletonima, postavlja se pitanje kako se možete osloboditi svih onih „gadnih" zavisnosti tih Singletona i učiniti kod lakšim za testiranje. To je važno pitanje koje razmatram u odeljku „Injektovanje zavisnosti" u Poglavlju 6.

UPOTREBA ZASTARELIH SISTEMA

Ako ste suočeni sa zastarelim sistemima i sa mnogim poteškoćama dok pokušavate da dodate jedinične testove, preporučujem knjigu „Working Effectively with Legacy Code" [Feathers07], koju je napisao Michael C. Feathers. Ta knjiga sadrži mnoge strategije za rad sa velikim, neproverenim nasleđenim bazama kodova. Takođe uključuje katalog od 24 tehnike za prekidanje zavisnosti. Te strategije i tehnike nećemo razmatrati u ovoj knjizi.

Jedna asertacija po testu

Moj savet je da ograničite jedinični test samo na jednu asertaciju, kao što je prikazano u listingu 2-3. Znam da je ovo kontroverzna tema, pa ću pokušati da objasnim zašto mislim da je važna.

LISTING 2-3 Jedinični test koji proverava operatora nejednakosti klase Money

```
void MoneyTest::givenTwoMoneyObjectsWithDifferentBalance_
InequalityComparison_Works() {
   const Money m1(-4000.0);
   const Money m2(2000.0);
   ASSERT_TRUE(m1 != m2);
}
```

Možemo reći da biste takođe mogli da proverite da li drugi operatori poređenja (na primer, `Money::operator==()`) funkcionišu pravilno u ovom jediničnom testu. Bilo bi to lako učiniti jednostavnim dodavanjem dodatnih asertacija, kao što je prikazano u listingu 2-4.

LISTING 2-4 Pitanje: Da li je dobra ideja proveriti sve operatore poređenja u jednom jediničnom testu?

```
void MoneyTest::givenTwoMoneyObjectsWithDifferentBalance_
testAllComparisonOperators() {
   const Money m1(-4000.0);
   const Money m2(2000.0);
   ASSERT_TRUE(m1 != m2);
   ASSERT_FALSE(m1 == m2);
   ASSERT_TRUE(m1 < m2);
   ASSERT_FALSE(m1 > m2);
   // ...more assertions here...
}
```

Mislim da su problemi u vezi sa ovim pristupom očigledni:

- Ako test ne bude uspešan zbog više razloga, programerima može da bude teško da brzo pronađu uzrok greške. Pre svega, rana asertacija koja nije uspešna prikriva dodatne greške, odnosno sakriva naredne asertacije, jer je izvršavanje testa zaustavljeno.
- Kao što je objašnjeno u odeljku „Imenovanje jediničnih testova“, trebalo bi da imenujemo test na precizan i ekspresivan način. Sa višestrukim asertacijama, jedinični test testira mnoge „stvari“ (što je, inače, kršenje principa jedinstvene odgovornosti; pogledajte Poglavlje 6) i bilo bi teško pronaći dobar naziv za to. Naziv `...testAllComparisonOperators()` nije dovoljno precizan.

Nezavisna inicijalizacija okruženja jediničnih testova

Ovo pravilo je donekle slično nezavisnosti jediničnih testova. Kada se čisto sproveden test završi, sva stanja povezana sa njim moraju nestati. Tačnije rečeno, prilikom izvođenja svih jediničnih testova svaki test mora biti izolovana parcijalna instanca aplikacije. Svaki test mora samostalno da postavi i inicijalizuje svoje potrebno okruženje. Isto se odnosi i na čišćenje nakon izvršenja testa.

Isključite gettere i settere

Nemojte pisati jedinične testove za uobičajene gettere i settere klase, kao u listingu 2-5.

LISTING 2-5 Jednostavni getter i setter

```
void Customer::setForename(const std::string& forename) {
  this->forename = forename;
}

const std::string& Customer::getForename() const {
  return forename;
}
```

Da li zaista očekujete da bi nešto moglo da pođe „po zlu“ sa tako jednostavnim metodima? Ove funkcije članova su, obično, toliko jednostavne da bi bilo besmisleno pisati jedinične testove za njih. Štaviše, uobičajeni getteri i setteri se implicitno testiraju pomoću drugih i važnijih jediničnih testova.

Obratite pažnju da sam upravo napisao da nije potrebno testirati **uobičajene i jednostavne** gettere i settere. Ponekad getteri i setteri nisu mnogo jednostavni. Prema principu skrivanja informacija (pogledajte odeljak o skrivanju informacija u Poglavlju 3), o kojem ćemo kasnije govoriti, trebalo bi getter sakriti od klijenta ako je jednostavan i besmislen ili ako mora da izvršava složene zadatke da bi utvrdio svoju povratnu vrednost. Zbog toga, ponekad može biti korisno napisati eksplicitni test za getter ili setter.

Isključite kod drugih proizvođača

Nemojte pisati testove za kod drugih proizvođača. Ne morate da proveravate da li biblioteke ili radni okviri funkcionišu na očekivani način. Na primer, možete mirne savesti pretpostaviti da korišćena funkcija člana `std::vector::push_back()` iz C++ standardne biblioteke funkcioniše pravilno. Suprotno tome, možete očekivati da kod drugih proizvođača ima svoje jedinične testove. Može da bude mudra arhitektonska odluka da u projektu ne koristite biblioteke ili radne okvire koji nemaju svoje jedinične testove i čiji je kvalitet sumnjiv.

Isključite spoljne sisteme

Isto važi i za spoljne sisteme - nemojte pisati testove za spoljne sisteme koji su deo konteksta vašeg sistema, koji treba da se razvijaju i zbog toga nisu u vašoj odgovornosti. Na primer, ako vaš finansijski softver koristi postojeći spoljni sistem za konverziju valuta koji je povezan pomoću Interneta, ne bi trebalo to da testirate. Osim činjenice da takav sistem ne može dati definisan odgovor (faktor konverzije između valuta varira iz minuta u minut) i da je takav sistem nemoguće kreirati zbog problema mreže, mi nismo odgovorni za spoljni sistem.

Moj savet je da simulirate ove „stvari" i da testirate vaš kod, a ne kod spoljnih sistema. Kasnije u ovom poglavlju biće reči o testiranju duplikata (lažnih objekata).

Šta radimo sa bazom podataka?

Mnogi IT sistemi u današnje vreme sadrže (relacione) baze podataka. Od njih se zahteva da sačuvaju ogromne količine objekata ili podataka u dugotrajnijem skladištu, tako da se ovi objekti ili podaci mogu jednostavno zahtevati i „preživeti" isključivanje sistema.

Važno pitanje je šta radite sa bazom podataka tokom jediničnog testiranja.

> *„Moj prvi i najvažniji savet na ovu temu je: Kada postoji bilo koji način testiranja bez baze podataka, testirajte bez baze podataka!"*
>
> Gerard Meszaros, *xUnit Test Patterns*

Baze podataka mogu izazvati različite, a ponekad i suptilne probleme tokom jediničnog testiranja. Na primer, ako mnogi jedinični testovi koriste istu bazu podataka, baza podataka postaje velika centralna memorija koju ti testovi moraju deliti za različite svrhe. Ovo deljenje može negativno uticati na nezavisnost jediničnih testova, o kojoj je već bilo reči u ovom poglavlju. Može biti teško garantovati traženi preduslov za svako jedinično testiranje. Izvršenje pojedinačnog jediničnog testa može izazvati neželjene efekte drugih testova pomoću najčešće korišćene baze podataka.

Drugi problem je što su baze podataka u osnovi spore. One su mnogo sporije od lokalne računarske memorije. Jedinični testovi koji stupaju u interakciju sa bazom podataka obično izvode veličine sporije od testova koji se u potpunosti mogu pokrenuti u memoriji. Zamislite da imate nekoliko stotina jediničnih testova, a da je za svaki od njih u proseku potreban dodatni vremenski opseg

od 500 ms koji je uzrokovan upitima baze podataka. Sve u svemu, svi testovi sa bazom podataka traju nekoliko minuta duže od testova bez baze podataka.

Moj savet je da simulirate bazu podataka (pogledajte odeljak o duplim/lažnim objektima za testiranje kasnije u ovom poglavlju) i da izvršite sve jedinične testove isključivo u memoriji. Ne brinite: baza podataka, ako postoji, biće uključena na nivou integracije i testiranja sistema.

Ne mešajte kod za testiranje sa proizvodnim kodom

Ponekad programeri dođu na ideju da svoj proizvodni kod opreme kodom za testiranje. Na primer, klasa može da sadrži kod za upravljanje zavisnošću kolaborativne klase tokom testa, kao što je prikazano u listingu 2-6.

LISTING 2-6 Jedno moguće rešenje za upravljanje zavisnošću tokom testiranja

```
#include <memory>
#include "DataAccessObject.h"
#include "CustomerDAO.h"
#include "FakeDAOForTest.h"
using DataAccessObjectPtr = std::unique_ptr<DataAccessObject>;
class Customer {
public:
  Customer() = default;
  explicit Customer(const bool testMode) : inTestMode(testMode) {}
  void save() {
    DataAccessObjectPtr dataAccessObject = getDataAccessObject();
    // ...use dataAccessObject to save this customer...
  }
  // ...

private:
  DataAccessObjectPtr getDataAccessObject() const {
    if (inTestMode) {
      return std::make_unique<FakeDAOForTest>();
    } else {
      return std::make_unique<CustomerDAO>();
    }
  }
  // ...more operations here...
  bool inTestMode{ false };
  // ...more attributes here...
};
```

`DataAccessObject` je apstraktna osnovna klasa specifičnih DAO-va, u ovom slučaju DAO-va `CustomerDAO` i `FakeDAOForTest`. Poslednji DAO je lažni objekat, koji je jednostavno testni duplikat (pogledajte odeljak o testnim duplikatima kasnije u ovom poglavlju). Služi da zameni pravi DAO, jer ne želimo da ga testiramo i ne želimo da „spašavamo" klijenta tokom testiranja (setite se mojih saveta o bazama podataka). Član Bulovih podataka `inTestMode` utvrđuje koji DAO treba da se koristi.

Dakle, ovaj kod bi funkcionisao, ali rešenje ima nekoliko nedostataka.

Pre svega, proizvodni kod je pretrpan kodom za testiranje. Iako se na prvi pogled ne pojavljuje dramatično, kod za testiranje može da poveća složenost i da smanji čitljivost proizvodnog koda. Potreban nam je dodatni član da bismo razlikovali režim testiranja i proizvodnu upotrebu našeg sistema. Ovaj Bulov član nema nikakve veze sa klijentom, a ni sa domenom našeg sistema. I lako je zamisliti da je ova vrsta člana potrebna u mnogim klasama u našem sistemu.

Štaviše, klasa `Customer` zavisi od DAO-va `CustomerDAO` i `FakeDAOForTest`. Možete da vidite ove DAO-ve na listi `includes` pri vrhu izvornog koda. To znači da je lažni DAO za testiranje `FakeDAOForTest` takođe deo sistema u proizvodnom okruženju. Može se očekivati da se kod testnog duplikata nikada ne poziva u proizvodnom režimu, već se kompajlira, povezuje i primenjuje.

Naravno, postoje elegantniji načini na koje možete da se „nosite" sa ovim zavisnostima i da zaštitite proizvodni kod od koda za testiranje. Na primer, možemo injektovati određeni DAO kao referentni parametar u `Customer::save()`. Pogledajte listing 2-7.

LISTING 2-7 Izbegavanje zavisnosti od koda za testiranje (1)

```
class DataAccessObject;
class Customer {
public:
  void save(DataAccessObject& dataAccessObject) {
    // ...use dataAccessObject to save this customer...
  }
  // ...
};
```

Alternativno, ovo izbegavanje zavisnosti od koda za testiranje može se izvesti tokom konstruisanja instanci tipa `Customer`. U ovom slučaju moramo referencirati DAO kao atribut klase. Štaviše, moramo da potisnemo automatsko generisanje podrazumevanog konstruktora pomoću kompajlera, jer ne želimo

da bilo koji korisnik klase `Customer` kreira njenu nepravilno inicijalizovanu instancu. Pogledajte listing 2-8.

LISTING 2-8 Izbegavanje zavisnosti od koda za testiranje (2)

```
class DataAccessObject;
class Customer {
public:
  Customer() = delete;
  explicit Customer(DataAccessObject& dataAccessObject) :
  dataAccessObject_(dataAccessObject) {}
  void save() {
    // ...use member dataAccessObject to save this customer...
  }
  // ...
private:
  DataAccessObject& dataAccessObject_;
  // ...
};
```

IZBRISANE FUNKCIJE

U jeziku C++ kompajler automatski generiše takozvane *specijalne funkcije člana* (podrazumevani konstruktor, konstruktor kopije, operator dodele kopija i destruktor) za tip ako ne deklariše svoj [C++11]. Od pojave jezika C++ 11 na ovu listu posebnih funkcija članova dodati su konstruktor pomeranja i operator dodele pomaka. C++ 11 i novije verzije obezbeđuju jednostavan i deklarativan način za suzbijanje automatskog kreiranja bilo koje specijalne funkcije člana, ali i uobičajenih funkcija članova i funkcija koje nisu članovi: možete ih izbrisati. Na primer, možete sprečiti kreiranje podrazumevanog konstruktora na sledeći način:

```
class Clazz {
public:
  Clazz() = delete;
};
```

I još jedan primer: možete izbrisati operator `new` da biste sprečili dinamičko raspoređivanje klasa u hip memoriji:

```
class Clazz {

public:

  void* operator new(std::size_t) = delete;

};
```

Treća alternativa bi mogla biti da određeni DAO kreira „fabrika“ (pogledajte odeljak „Fabrika“ u Poglavlju 9 o projektnim obrascima) koju `Customer` „zna“. Ova „fabrika“ se može konfigurisati spolja da bi kreirala vrstu DAO-a koja je potrebna ako je sistem pokrenut u okruženju za testiranje. Bez obzira koje od ovih mogućih rešenja odaberemo, klasa `Customer` nema kod za testiranje. Ne postoje zavisnosti od specifičnih DAO-va u klasi `Customer`.

Testovi moraju da se izvršavaju brzo

U velikim projektima jednog dana ćete doći do tačke u kojoj ćete imati hiljade jediničnih testova. Ovo je odlično kada je reč o kvalitetu softvera. Međutim, neprijatan neželjeni efekat može biti to što će ljudi prestati da izvode ove testove pre nego što se prijave u spremište izvornog koda, jer će izvršavanje testova predugo trajati.

Lako je zamisliti da postoji snažna korelacija između vremena potrebnog za izvođenje testova i produktivnosti tima. Ako izvođenje svih jediničnih testova traje 15 minuta, pola sata ili duže, programeri su ometeni u obavljanju posla i gube vreme u čekanju rezultata testa. Čak i ako izvršavanje svakog jediničnog testa u proseku traje „samo“ pola sekunde, za izvođenje 1.000 testova potrebno je više od osam minuta. To znači da će izvršavanje čitavog paketa testova 10 puta dnevno trajati skoro sat i po. Kao rezultat toga, programeri će ređe izvoditi testove.

Moj savet je: **testovi moraju da se izvršavaju brzo!** Jedinični testovi bi trebalo da uspostave brzu povratnu spregu za programere. Izvođenje svih jediničnih testova za veliki projekat trebalo bi da traje tri minuta. Za brže izvršavanje lokalnog testa (nekoliko sekundi) tokom razvoja radni okvir za testiranje treba da obezbedi jednostavan način za privremeno isključivanje nerelevantnih grupa testova.

U automatizovanom sistemu za razvoj softvera svi testovi moraju da se izvode bez izuzetka neprekidno pre nego što se finalni proizvod izradi. Razvojni tim bi trebalo odmah da dobije obaveštenje ako jedan ili više testova nisu uspešni u sistemu za razvoj softvera. Obaveštenje se može poslati pomoću e-pošte ili pomoću optičke vizuelizacije (na primer, zbog ravnog ekrana na zidu

ili „semafora“ kojim upravlja sistem za razvoj softvera) na istaknutom mestu. Ako čak i samo jedan test ne bude uspešan, ni pod kojim uslovima ne smete objaviti i isporučiti proizvod!

Kako pronalazite ulazne podatke testa?

Softver može reagovati veoma različito, u zavisnosti od podataka koji se koriste kao ulazni podaci. Ako bi jedinični testovi trebalo da dodaju vrednost vašem projektu, brzo ćete doći do pitanja kako možete pronaći sve testne slučajeve koji su neophodni za obezbeđivanje dobre detekcije grešaka.

Sa jedne strane, želite da imate veoma visoku, idealno potpunu pokrivenost testom. Sa druge strane, ekonomski aspekti, kao što su trajanje projekta i budžet, takođe se moraju imati na umu. To znači da često nije moguće izvršiti opsežno testiranje za svaki skup testnih podataka, posebno ako postoji veliki skup ulaznih kombinacija, a na kraju ćete dobiti skoro beskonačan broj testnih slučajeva.

Da bi bio pronađen dovoljan broj testnih slučajeva, postoje dva centralna i važna koncepta u osiguranju kvaliteta softvera: *ekvivalentna particija*, koje se ponekad naziva i *particioniranje klase ekvivalencije* (ECP – equivalence class partitioning), i *analiza granične vrednosti*.

Ekvivalentna particija

Ekvivalentna particija, koja se ponekad naziva i klasa ekvivalencije, skup je ili deo ulaznih podataka za koje bi deo softvera, kako u okruženju za testiranje, tako i u svom operativnom okruženju, trebalo da pokaže slično ponašanje. Drugim rečima, pretpostavlja se da je ponašanje sistema, komponente, klase ili funkcije/metoda isto, na osnovu njihovih specifikacija.

Rezultat ekvivalentne particije može se koristiti za izvođenje testova iz ovih particija sličnih ulaznih podataka. U principu, testni slučajevi su dizajnirani tako da je svaka particija pokrivena bar jednom.

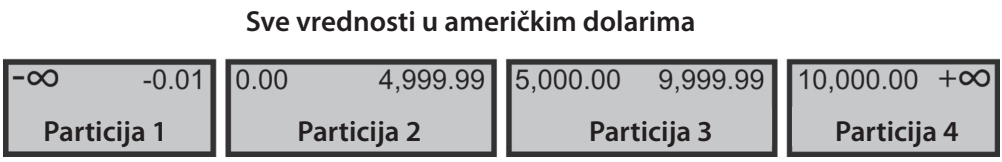
Kao pristup zasnovan na specifikacijama, tehnika ekvivalentne particije je tehnika testiranja dizajna u „crnoj kutiji“ - tj. unutrašnjost softvera koji se testira obično nije poznata. Međutim, to je takođe veoma koristan pristup za tehnike testiranja „bele kutije“, tj. za jedinično testiranje i pristupe test-first, kao što je TDD (pogledajte Poglavlje 8).

Sada ćemo pogledati primer. Pretpostavimo da moramo da testiramo C++ klasu koja izračunava kamatu na bankovni račun. Prema specifikaciji zahteva, račun treba da se ponaša na sledeći način:

- Banka naplaćuje četiri odsto zatezne kamate na prekoračenja.
- Banka nudi 0,5 odsto kamate za prvih 5.000 USD štednje.
- Banka nudi jedan odsto kamate za sledećih 5.000 USD štednje.
- Za ostatak banka nudi dva odsto kamate.
- Kamata se obračunava svakodnevno.

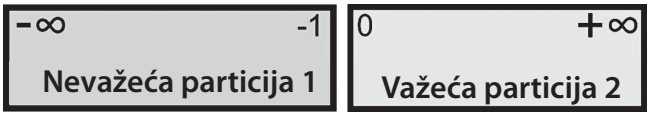
Prema ovim specifikacijama, API kalkulator kamate ima dva parametra: novčani iznos na osnovu kamate koja se izračunava na dnevnoj bazi i broj dana za koje ovaj iznos važi. To znači da moramo izraditi klase ekvivalencije za dva ulazna parametra.

Ekvivalentna particija za iznos novca prikazana je na slici 2-4.



Slika 2-4 Klase ekvivalencije ulaznog parametra za novčani iznos

Klase ekvivalencije za period važenja u danima su nešto jednostavnije i prikazane su na slici 2-5.



Slika 2-5 Klase ekvivalencije ulaznog parametra za broj dana

Kakve pouke sada možemo izvući iz ovoga za kreiranje testnih slučajeva?

Pre svega, imajte na umu da ulazni parametar za novčani iznos dozvoljava beskonačno velike pozitivne ili beskonačno velike negativne vrednosti. Nasuprot tome, negativne vrednosti za broj dana nisu dozvoljene.

Ovo je trenutak kada bi bilo preporučljivo uključiti poslovne zainteresovane strane i stručnjake za domen.

Prvo, treba razjasniti da li je gornja ili donja granica novčanog iznosa zaista beskonačna. Odgovor na ovo pitanje ne utiče samo na testne slučajeve, već i na tip podataka koji će se koristiti za ovaj parametar. Dalje, specifikacija ne razjašnjava šta bi trebalo da se dogodi ako se negativna vrednost koristi za

broj dana. Negativna vrednost bi bila nevažeća, ali kakvu bi reakciju trebalo da pokaže kalkulator kamate?

Drugo pitanje na koje bi takva analiza mogla da odgovori bilo bi, na primer, da li su kamatne stope zaista toliko fiksne (konstante) koliko i specifikacija zahteva. Možda su kamatne stope promenljive, a možda su promenljivi i novčani iznosi povezani sa njima.

Međutim, testovi se sada mogu sistematski izvesti iz ove analize. Ideja iza particioniranja ekvivalencije je da je dovoljno izabrati samo jednu vrednost iz svake particije za testiranje. Hipoteza koja stoji iza ove tehnike je da **ako jedan uslov/vrednost u particiji „prođe" test, svi drugi u istoj particiji će takođe „proći" test**. Slično tome, ako jedan uslov/vrednost u particiji nije uspešan/uspešna, svi drugi uslovi/vrednosti u toj particiji takođe neće biti uspešni/uspešne. Ako postoji više parametara, kao u našem slučaju, potrebno je formirati odgovarajuće kombinacije.

Analiza granične vrednosti

„Greške vrebaju u uglovima i sakupljaju se na granicama."

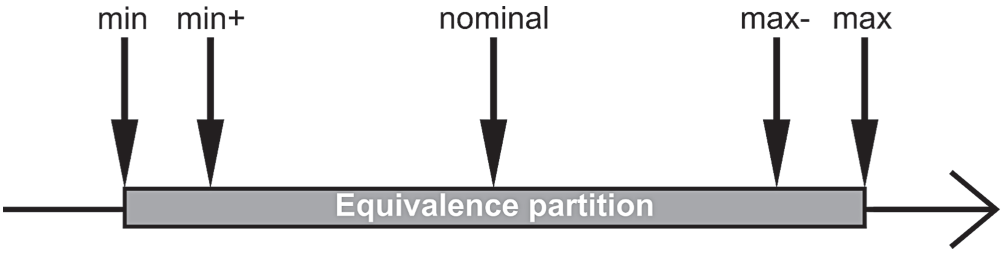
Boris Beizer, Software TestingTechniques [Beizer90]

Mnoge softverske greške mogu se pratiti do pojave poteškoća u graničnim oblastima klasa ekvivalencije - na primer, pri prelazu između dve važeće klase ekvivalencije, između važeće i nevažeće klase ekvivalencije ili pri ekstremnoj vrednosti koja nije uzeta u obzir. Stoga se izrada klasa ekvivalencije dopunjava analizom granične vrednosti.

U disciplini testiranja analiza granične vrednosti je tehnika koja pronalazi tačke prelaska između klasa ekvivalencije i bavi se ekstremnim vrednostima. Rezultat takve analize je koristan za odabir ulaznih vrednosti numeričkog parametra za testove:

- tačno na svom minimumu
- nešto iznad minimuma
- nominalna vrednost uzeta negde od sredine particije ekvivalencije
- odmah ispod maksimuma
- tačno na svom maksimumu

Ove vrednosti se takođe mogu prikazati na numeričkoj liniji, kao na slici 2-6.



Slika 2-6 Ulazni parametri izvedeni iz analize granične vrednosti

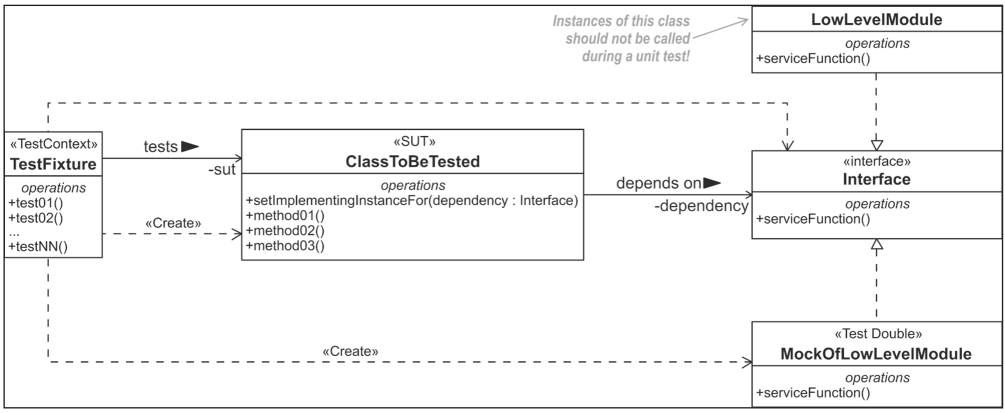
Ako se granične vrednosti utvrde i ispitaju za svaku particiju ekvivalencije, onda se u praksi može postići veomalo dobra pokrivenost testom uz relativno mali napor.

Testni duplikati (lažni objekti)

Jedinični testovi treba da se zovu „jedinični testovi" samo ako su jedinice koje se testiraju potpuno nezavisne od „saradnika" tokom izvođenja testa, odnosno ako jedinica koja se testira ne koristi druge jedinice ili spoljne sisteme. Na primer, iako je uključivanje baze podataka tokom integracionog testa nekritično i potrebno, jer je to svrha integracionog testa, pristup (na primer, upit) ovoj bazi podataka tokom stvarnog jediničnog testa je zabranjen (pogledajte odeljak „Šta da radimo sa bazom podataka?" ranije u ovom poglavlju).

Prema tome, zavisnosti jedinice koja će se testirati od drugih modula ili spoljnih sistema treba zameniti takozvanim *testnim duplikatima*, koji su poznati i kao *lažni objekti*, ili *„makete"*.

Da bismo na elegantan način koristili takve testne duplikate, trebalo bi da težimo labavoj sprezi jedinice koja se testira (pogledajte odeljak „Labava sprega" u Poglavlju 3). Na primer, apstrakcija (konkretno, interfejs u obliku čiste apstraktne klase) može se na pristupnoj tački predstaviti neželjenom „saradniku", kao što je prikazano na slici 2-7.



Slika 2-7 Interfejs olakšava zamenu LovLevelModule testnim duplikatima

Pretpostavimo da želite da razvijete aplikaciju koja koristi spoljnu veb uslugu za konverziju valuta u realnom vremenu. Tokom jediničnog testa ne možete prirodno koristiti ovu spoljnu uslugu, jer ona isporučuje različite faktore konverzije svakog minuta.

Osim toga, usluga se traži pomoću Interneta, što je u osnovi sporo i možda neće biti uspešno. Nemoguće je simulirati granične slučajeve, pa morate konverziju stvarne valute da zamenite testnim duplikatom tokom jediničnog testa.

Prvo, moramo da uvedemo tačku varijacije u kod da bismo mogli da zamenimo modul koji komunicira sa uslugom konverzije valuta pomoću testnog duplikata. To se ne može učiniti pomoću interfejsa, koji je na jeziku C++ apstraktna klasa sa isključivo čistim virtuelnim funkcijama članova. Pogledajte listing 2-9.

LISTING 2-9 Apstraktni interfejs za konvertore valuta

```
class CurrencyConverter {

public:
   virtual ~CurrencyConverter() { }
   virtual long double getConversionFactor() const = 0;
};
```

Pristup usluzi konverzije valuta pomoću Interneta je inkapsuliran u klasu koja implementira interfejs `CurrencyConverter`. Pogledajte listing 2-10.

LISTING 2-10 Klasa koja pristupa usluzi konverzije valuta u realnom vremenu

```
class RealtimeCurrencyConversionService : public CurrencyConverter {

public:
  virtual long double getConversionFactor() const override;
  // ...more members here that are required to access the service...
};
```

Za potrebe testiranja, postoji druga implementacija - testni duplikat `CurrencyConversionServiceMock`. Objekti ove klase će vratiti definisani i predvidljivi faktor konverzije koji je potreban za jedinično testiranje. Osim toga, objekti ove klase obezbeđuju mogućnost podešavanja faktora konverzije spolja - na primer, za simulaciju graničnih slučajeva. Pogledajte sliku 2-11.

LISTING 2-11 Testni duplikat

```
class CurrencyConversionServiceMock : public CurrencyConverter {

public:
  virtual long double getConversionFactor() const override {
    return conversionFactor;
  }

  void setConversionFactor(const long double
    value) { conversionFactor = value;
  }

private:
  long double conversionFactor{0.5};
};
```

Na mestu u proizvodnom kodu na kojem se koristi konvertor valuta interfejs se sada koristi za pristup usluzi. Zbog ove apstrakcije, potpuno je transparentno za kod klijenta koja vrsta implementacije se koristi tokom izvršavanja - pravi konvertor valuta ili njegov testni duplikat. Pogledajte listinge 2-12 i 2-13.

LISTING 2-12 Zaglavlje klase koja koristi uslugu

```
#include <memory>
class CurrencyConverter;
```

```
class UserOfConversionService {
public:
  UserOfConversionService() = delete;
  explicit UserOfConversionService(const std::shared_
  ptr<CurrencyConverter>& conversionService);
  void doSomething();
  // More of the public class interface follows here...

private:
  std::shared_ptr<CurrencyConverter> conversionService_;
  //...internal implementation...
};
```

LISTING 2-13 Izvod iz implementacione datoteke

```
UserOfConversionService::UserOfConversionService  (const std::shared_
ptr<CurrencyConverter>& conversionService) :
  conversionService_(conversionService) { }
void UserOfConversionService::doSomething() {
  long double conversionFactor = conversionService_->getConversionFactor();
  // ...
}
```

U jediničnom testu za klasu `UserOfConversionService` testni slučaj sada može proslediti lažni objekat konstruktoru inicijalizacije. Sa druge strane, tokom uobičajenog rada prava usluga se može proslediti konstruktoru. Ova tehnika je projektni obrazac koji se naziva *injektovanje zavisnosti*, o čemu se detaljno govori u istoimenom odeljku Poglavlja 9. Pogledajte listing 2-14.

LISTING 2-14 UserOfConversionService dobija potreban objekat CurrencyConverter

```
auto serviceToUse =
  std::make_shared</* name of the desired class here
*/>(); UserOfConversionService user(serviceToUse);
// The instance of UserOfConversionService is ready for
use... user.doSomething();
```