9 - Razvoj vođen testovima

Problem sa programerima je
što nikada ne možete da budete sigurni šta rade,
sve dok ne bude prekasno.
Sejmur Krej

9.1 Testiranje softvera

Testiranje softvera je postupak proveravanja da li se razvijeni softver ponaša na ispravan način, u nekim izabranim reprezentativnim slučajevima. Testiranje ima veoma važnu ulogu u okviru obezbeđivanja kvaliteta proizvoda, pa predstavlja jedan od nezaobilaznih poslova u okviru razvojnog procesa. Obično je najzastupljenije u završnim fazama razvoja softvera ili njegovih pojedinačnih delova, ali neke vrste testova imaju svoje mesto i u drugim fazama razvoja.

Važno je da istaknemo da sprovedeno testiranje ne može da predstavlja *dokaz* ispravnosti softvera, već samo može da potvrdi da softver radi ispravno za jedan unapred određen skup test primera. Eventualno dokazivanje korektnosti je predmet formalne verifikacije softvera i ne može da se ostvari testiranjem. Ipak, dobro izabran skup testova može da značajno umanji verovatnoću postojanja neprimećenih grešaka u kodu, pa zbog toga dobro isplanirano i dovoljno temeljno i široko testiranje ima veliku težinu u okviru procenjivanja kvaliteta softvera. Zbog toga se testiranje često naziva i *neformalnom* verifikacijom.

Pri razvoju softvera se u kontekstu provere ispravnosti upotrebljavaju (i često mešaju) dva termina: *verifikacija softvera* je proveravanje da li je softver razvijen u skladu sa ustanovljenim formalnim i neformalnim specifikacijama i kriterijumima kvaliteta, dok je *validacija softvera* proveravanje da li softver zadovoljava stvarne

potrebe naručioca. Često kažemo da je verifikacija proveravanje da li se softver *dobro pravi*, a da je validacija proveravanje da li pravimo *dobar softver*. Većina testova se odnosi na verifikaciju.

Da bi testiranje softvera imalo značaja, ono mora da bude obavljano sistematično i sa punim razumevanjem softvera koji se testira. Pri planiranju i projektovanju testova neophodno je sveobuhvatno i temeljno poznavanje ciljeva i projektnih zahteva, kao i svih elemenata arhitekture i implementacije softvera.

U razvojnom procesu svoje mesto pronalazi veći broj različitih vrsta testova, koji se primenjuju u različitim periodima razvoja i sa različitim ciljevima. Testovi mogu da se razlikuju i klasifikuju u odnosu na neka od svojstava, kao što su, na primer, cilj testiranja, predmet testiranja, obim testiranja, vreme testiranja i drugo. Najpre ćemo predstaviti vrste testova prema cilju i obimu, a zatim ćemo se temeljnije posvetiti testovima jedinica koda.

Vrste testova prema cilju

U odnosu na cilj testiranja prepoznaju se tri velike kategorije testova:

- *funkcionalni testovi*;
- *nefunkcionalni testovi* i
- *holistički testovi*.

Funkcionalni testovi obuhvataju sve vrste testova koji imaju za cilj proveravanje da li jedinice koda, komponente ili softver kao celina rade u skladu sa zahtevima, specifikacijama i dokumentacijom:

- *testovi ispravnosti* proveravaju da li funkcije i metodi izračunavaju ispravne rezultate i proizvode ispravne promene stanja sistema, kao i da li upotreba aplikativnog interfejsa softvera proizvodi planirane posledice;
- *testovi kompatibilnosti* služe da se proveri da li se softver povezuje sa drugim softverom na način na koji je to predviđeno.

Nefunkcionalni testovi obuhvataju sve one vrste testova kojima nije primarni cilj proveravanje ispravnosti rezultata rada softvera i njegovih komponenti:

- *testovi upotrebljivosti* imaju za osnovni cilj proveravanje da li je korisnički interfejs dovoljno upotrebljiv za ciljne korisnike; kao specijalni slučajevi testova upotrebljivosti prepoznaju se još:
  - *testovi upotrebljivosti za korisnike sa posebnim potrebama*, koji služe da se provere specifični aspekti korisničkog interfejsa, i

  - *regionalni testovi*, koji služe da se proveri prilagođenost softvera različitim jezicima ili lokalnim okolnostima upotrebe;
- *testovi performansi* podrazumevaju višestruko izvršavanje značajnih operacija u različitim okolnostima, radi procenjivanja efikasnosti sistema i njegovih delova u realnim uslovima[33]; prema vrsti testiranih performansi ovi mogu da obuhvate:
  - *testovi efikasnosti*, koji mere brzinu rada ili vreme odziva softvera u uslovima predviđenog radnog opterećenja;
  - *testovi opterećenja*, koji mere opterećenje razliitih komponenti i prate njihovo ponašanje pod povećanim opterećenjem[34];
  - *testovi stabilnosti* procenjuju kako se softver ponaša u uslovima ekstremnog opterećenja, koje prevazilazi okvire za koje je softver razvijen; fokus je na ustanovljavanju da li dolazi do značajnih padova performansi ili čak i potpunog otkaza rada softvera;
  - *testovi istrajnosti* proveravaju kako se softver ponaša u uslovima produžene neprekidne upotrebe;
  - *testovi skalabilnosti*, koji proveravaju da softver može da se skalira i koliko se to lako ostvaruje u slučaju povećanog opterećenja i
  - *testovi uskih grla* su fokusirani na najopterećenije delove softvera i hardvera; slični su testovima opterećenja i testovima stabilnosti
  - i drugo;
- *testovi bezbednosti* proveravaju da li su svi bezbednosni aspekti sistema ispravno implementirani – na primer, da li neidentifikovan korisnik može da pristupi zaštićenim elementima softvera, da li identifikovan korisnik može da pristupi elementima za koje nema autorizaciju i drugo;
- *testovi instalacije* obuhvataju instaliranje softvera u različitim uslovima i proveravanje da li su na kraju instalacije svi delovi sistema ispravno konfigurisani i pripremljeni za rad;
- *destruktivni testovi* proveravaju koliko je sistem otporan na (namerne i slučajne) pokušaje da se dovede u neaktivno stanje;

[33] Iako slabe performanse mogu da dovedu u pitanje praktičnu funkcionalnost softvera, ipak je uobičajeno da se pitanja vezana za performanse softvera svrstavaju u nefunkcionalne zahteve.

[34] Merenje vremena odziva i propusnosti softvera ili njegovih delova se svrstava i u testove opterećenja i u testove efikasnost.

Holistički testovi su celoviti testovi softvera od strane korisnika. Prema trenutku u kome se testiranje odvija i vrsti korisnika koji vrše testiranje, prepoznajemo nekoliko tipova holističkih testova:

- *razvojni testovi* predstavljaju redovno celovito testiranje upotrebe softvera tokom njegovog razvoja od strane članova razvojnog tima, koji su specijalizovani za poslove testiranja;
- *testovi prihvatljivosti* se odvijaju u odnosu na celovit softverski proizvod i proveravaju da li on ispunjava uslove i zadovoljava zahteve određene projektnim zadatkom; predmet testova prihvatljivosti može biti konačan proizvod, rezultat pojedinačne iteracije ili čak samo deo sistema koji predstavlja ostvarenje pojedinačnog slučaja upotrebe ili korisničke celine;
- *alfa testiranje* je celovito testiranje od strane uže grupe korisnika, obično relativno dobro tehnički obrazovanih, čija je namena da se u relativno ranoj fazi proizvodnje (kada su uglavnom sve mogućnosti softvera bar delimično implementirane, ali još ne i do kraja doterane) prikupe kritička mišljenja; obično primarni cilj nije uočavanje bagova u implementaciji već grešaka i slabosti u konceptima, planovima i korisničkom interfejsu;
- *beta testiranje* je celovito testiranje koje se sprovodi od strane šire grupe korisnika, koji se biraju tako da što približnije odgovaraju ciljnoj grupi korisnika gotovog proizvoda; primarni cilj je blagovremeno uočavanje bagova, ali se testeri podstiču da komentarišu i konceptualne greške.

Alfa i beta testiranje se po pravilu primenjuju u slučaju softvera koji je namenjen širokom krugu korisnika. Mogu da se primenjuju i u slučaju softvera koji se radi namenski (npr. informacioni sistem nekog preduzeća) ali onda imaju nešto drugačiju prirodu. Nasuprot tome, testovi prihvatljivosti imaju veći značaj u slučaju namenskog softvera, dok se kod softvera za širi krug korisnika umesto njih često koriste nešto širi razvojni testovi.

Testovi prihvatljivosti su konceptualno drugačiji od ostalih vrsta testova, zato što se primarno bave načinom upotrebe i rezultatima upotrebe softvera iz ugla korisnika, a ne samo tehničkim aspektima realizacije. Oni se najviše bave testiranjem ponašanja sistema kao celine. Implementiraju se drugačijim alatima nego druge vrste testova. Često se definišu specifičnim skript jezikom, koji prateći alati mogu da izvršavaju simulirajući realnu upotrebu softvera (kroz simuliranje događaja koji nastaju upotrebom tastature ili miša i slično).

Vrste testova prema obimu

Prema obimu ili nivou testiranja obično se prepoznaju:

- *testovi jedinica koda*;
- *testovi komponenti;*
- *testovi integracije* i
- *testovi sistema*.

Testovi jedinica koda služe za proveravanje ispravnosti najmanjih funkcionalnih jedinica koda. Najpre se pojedinačni testovi upotrebljavaju za testiranje ispravnosti funkcija i metoda. Svaki pojedinačan test bi trebalo da proverava tačno jedan aspekt ponašanja testirane jedinice. Ispravno izgrađen skup testova bi trebalo da proverava sve aspekte ponašanja testirane funkcije ili metoda, uključujući ponašanje za različite ulazne vrednosti, kao i u različitim stanjima sistema. Neophodno je da se posebno proverava ponašanje za sve granične slučajeve, kao i za neispravne ulazne vrednosti.

Kada se više jedinica koda povezuje u jednu celinu, onda način testiranja ispravnosti te celine zavisi od načina povezivanja delova. Ako se delovi povezuju korišćenjem osnovnog načina povezivanja jedinica koda u konkretnom programskom jeziku (npr. pozivanje funkcija ili razmenjivanje poruka između objekata), onda ispravnost celine može da se proverava testovima jedinica koda. Sa druge strane, ako se delovi povezuju putem nekog drugog protokola (na primer, putem protokola HTTP), onda se testiranje ispravnosti takvog interfejsa obično ne obavlja testovima jedinica koda, već moraju da se koriste testovi komponenti ili testovi integracije[35].

Testovi komponenti se nalaze negde između testova jedinica koda i testova integracije. Često se u literaturi ne navode kao posebna vrsta testova, već se svrstavaju u testove integracije. Primarni predmet testiranja komponente je testiranje ispravnosti implementacije njenog interfejsa. Interno funkcionisanje delova komponente, pa čak i njenog celovitog funkcionisanja, sve do nivoa javnog interfejsa ali bez njega, se obično proverava već u okviru testiranja jedinica koda.

Interfejs komponente se najčešće definiše pomoću jedne klase, kojom se zaklanja implementacija. To je tipična primena obrasca za projektovanje *Fasada*. U tom slučaju funkcionalnost komponente i njenog interfejsa u suštini može da se svede na testiranje interfejsa fasadne klase, što može da se radi odgovarajućim testovima jedinica koda, ako fasadna klasa ima i interni osnovni interfejs.

[35] Naravno, čak i takvo testiranje može da se sprovede pomoću testiranja jedinica koda, ali je problem u tome što upotreba dodatnih protokola obično zahteva širu pripremu (na primer instaliranje, konfigurisanje i pokretanje modula koji se povezuju) i njegova efikasnost je za red veličine niža od većine „običnih“ testova jedinica koda, pa se zato to obično radi malo drugačije.

Međutim, spoljašnji interfejs komponente (onaj interfejs koji vide i koriste druge komponente, tj. interfejs koji se koristi u realnom okruženju) može da koristi neke protokole povezivanja koji se razlikuju od osnovnih načina povezivanja jedinica koda u konkretnom programskom jeziku. Tada njegovo testiranje ne može da se obavi testovima jedinica koda. Tehnika testiranja interfejsa pojedinačne komponente se obično ne razlikuje značajno od tehnike sprovođenja testova integracije, pa se zato testiranje komponenti često posmatra kao donekle specifičan oblik testova integracije.

Testovi integracije (ili *integralni testovi*) služe za proveravanje da li se komponente ispravno povezuju u veće celine. Neophodan uslov za testiranje integracije je da je prethodno uspešno sprovedeno testiranje svih pojedinačnih komponenti koje se integrišu u celinu.

Za testove integracije nije dovoljno pisanje jednostavnih testova, kao za testove jedinica koda. Obično se pišu posebni programi koji simuliraju povezivanje komponenti u ciljnom okruženju i njihovu upotrebu u relativno složenom kontekstu. Ako se komponente povezuju i upotrebljavaju putem nekog od standardizovanih protokola (na primer, Veb servisi ili REST), onda mogu da se upotrebljavaju i komercijalni alati za testiranje, koji obično omogućavaju pisanje skriptova za testiranje.

Testovi sistema se odnose na testiranje celog sistema. Mogu da imaju odlike testova integracije (ako se sistem posmatra iz tehničkog ugla) ili odlike testova prihvatljivosti (ako se sistem posmatra iz ugla korisnika), a najčešće predstavljaju njihovu kombinaciju. U svakom slučaju, testovi sistema podrazumevaju testiranje čitavog proizvoda i svih njegovih karakteristika.

Za razliku od ostalih navedenih vrsta testova, testovi sistema mogu da se bave i aspektima validacije, a ne samo verifikacije softvera. Tokom čitavog razvoja validacija može samo da se bavi pitanjem da li su specifikacije i kriterijumi kvaliteta ustanovljeni u skladu sa stvarnim potrebama klijenta ili ne, a tek kada je proizvod gotov, onda može da obuhvati i neke praktične provere. Dok se ostali testovi u suštini bave samo proveravanjem da li su zadovoljene specifikacije i formalni kriterijumi kvaliteta (tj. bavi se verifikacijom), testovi sistema mogu da obuhvate i neformalne kriterijume i da proveravaju da li je zaista razvijen proizvod kakav je bio potreban, ili je iz nekog razloga razvijen pogrešan proizvod.

9.2 Testovi jedinica koda

Kao što je već navedeno, osnovna namena testova jedinica koda je proveravanje ispravnosti najmanjih funkcionalnih jedinica koda. Suština testiranja jedinice koda je u tome da se na praktičan način, proveravanjem ponašanja u nekim uobičajenim,

specifičnim ili graničnim uslovima proveri da li se pri upotrebi programskog interfejsa jedinice koda dobijaju rezultati koji odgovaraju specifikaciji.

Najčešći predmet testiranja su funkcionalni zahtevi. Oni se testiraju kroz proveravanje da li se pri upotrebi interfejsa jedinice koda potvrđuju očekivane zavisnosti postuslova od preduslova:

- Da li rezultat funkcije ili metoda odgovara datim ulaznim vrednostima i datom početnom stanju sistema?
- Da li se na ispravan način menja stanje sistema?

Pored funkcionalnih zahteva može da se testira i robusnost jedinice koda:

- Da li se funkcije i metodi ponašaju na odgovarajući način u slučaju neispravnih ulaznih vrednosti?
- Da li se ispravno izveštava o problemima?
- Da li se ispravno izbacuju izuzeci?

Jedinica koda koja se testira može da bude jedna operacija, funkcija ili metod, neka struktura podataka ili klasa. Jedinica koda može da predstavlja i skup više manjih integrisanih jedinica koda. Ako ima odgovarajući programski interfejs, koji može da posluži za testiranje, onda kao jedinica koda može da se testira čak i softverski paket ili neki interni ili eksterni podsistem.

Pojedinačni testovi testiraju pojedinačne aspekte ponašanja pojedinačnih funkcija ili metoda. Sistematično izgrađene kolekcije testova se upotrebljavaju za testiranje ponašanja čitavih klasa ili paketa. Testiranju složenog ponašanja klase, tj. ispravnosti složenije upotrebe objekata i metoda klase, može da se pristupi tek nakon što se prethodno testira elementarno ponašanje svakog pojedinačnog metoda klase. Ako više klasa sarađuju međusobno u većem podsistemu, onda i ispravnost takvog podsistema može da se proverava testovima jedinica koda.

Testovi jedinica koda se pišu kao tzv. *neprodukcioni* kod za testiranje, tj. programski kod koji se piše radi testiranja neće biti sastavni deo proizvoda, već mu je isključiva namena testiranje. Obično se pri pisanju testova jedinica koda koriste odgovarajuće biblioteke za testiranje, a u specifičnim slučajevima može da bude potrebno čak i razvijanje sopstvenih delova koda koji služe samo za olakšavanje implementiranja testova jedinica koda.

Uobičajeno je da se testovima jedinica koda testira samo javni interfejs klasa i paketa. Ipak, u nekim kompleksnijim slučajevima je dobro da se omogući i testiranje privatnih delova, u kom slučaju se dodaju posebni metodi ili klase čija isključiva namena je da pomognu pri testiranju. Takve delove nekada nije lako izbaciti iz

produkcionog koda, pa se dešava da ostanu u njemu. Poželjno je da se ipak uloži dodatni napor da se kod organizuje tako da oni mogu da se izbace[36].

Testovi jedinica koda nisu dovoljni da dokažu ispravnost softvera. Dobro oblikovani testovi će dovesti do ispoljavanja većine bagova, ali oni praktično nikada ne mogu da obuhvate sve moguće uslove u kojima softver može da se nađe. Ako je potrebno da se dokaže korektnost softvera, onda je neophodno da se upotrebljavaju neke formalne metode verifikacije, bilo manuelne ili automatske. Sa druge strane, dobro oblikovana kolekcija testova će moći da nam pomogne da prepoznamo i otklonimo većinu bagova.

9.3 Biblioteka Catch2

Postoji veliki broj biblioteka koje pružaju podršku testiranju jedinica koda na programskom jeziku C++. Kao primer ćemo da ukratko predstavimo biblioteku *Catch2* [*Catch2*], koja tokom poslednjih nekoliko godina postaje sve zastupljenija[37].

Kod većine starijih biblioteka postoji problem netrivijalnog pisanja i izvođenja testova. Najčešće svaki test ili grupa testova mora da se na neki način registruje da bi bila izvedena. Pri razvoju biblioteke *Catch2* je osnovna motivacija bila prevazilaženje tih uobičajenih problema. Zaista, jedan od osnovnih razloga što je biblioteka *Catch2* postala popularna, jeste jednostavnost pisanja, prevođenja i izvođenja testova. Ona ne zahteva nikakvo dodatno povezivanje ili registrovanje testova – sve se odvija automatski. Kao što ćemo videti, programer je dužan samo da napiše testove i ne mora da se zatim stara o njihovoj tehničkoj implementaciji. Da bismo preveli i povezali testove dovoljno je da prevedemo i povežemo sve module koji su potrebni za testiranje (modul programa za testiranje i module sa testovima) i ostale module koje koristimo pri testiranju.

Najveća mana biblioteke *Catch2* je u tome što se distribuira i prevodi kao biblioteka u jednom zaglavlju, koja intenzivno koristi parametarski polimorfizam. Iako to čini upotrebu jednostavnijom, za posledicu ima relativno sporo prevođenje

[36] Na primer, jedna tehnika za testiranje privatnih elemenata klase `A` je da se deklariše prijateljska klasa (npr. `friend class TestHelperA`) i da se u njoj napišu odgovarajući metodi. Ne predstavlja problem ako takva deklaracija ostane u produkcionom kodu, ali ne bi bilo dobro da u njemu ostane i definicija te klase (`TestHelperA`).

[37] Broj 2 u imenu biblioteke nije oznaka verzije nego deo imena. Ova biblioteka se originalno zvala *Catch* ali je autor primetio da takvo ime pravi probleme kada se traži na vebu, pa je sa prelaskom na verziju 2 biblioteci promenjeno ime u *Catch2*. U planu je da nove verzije zadrže isto ime, pa je tako u vreme pisanja ove knjige u toku dovršavanje verzije 3, koja se i dalje zove *Catch2*.

testova i programa za testiranje. Iz tog razloga se od verzije 3 prelazi na upotrebu statičke biblioteke.

U primerima ćemo da koristimo aktuelnu stabilnu verziju 2 (u trenutku pisanja ovog teksta to je verzija 2.13.10), koja se distribuira kao biblioteka u jednom zaglavlju. U slučaju verzije 3 jedina razlika je u tome što je potrebno da se uključe druga zaglavlja i da se poveže statički prevedena biblioteka.

Pisanje programa za testiranje

Biblioteka *Catch2* sadrži makroe za testiranje i program koji izvršava testove. Da bismo u naš program ugradili komandni program za testiranje, tj. funkciju `main` iz biblioteke *Catch2*, koja izvodi testiranje, potrebno je da napišemo samo:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
```

Prvi red definiše konfiguracioni makro koji sugeriše da je potrebno da se ugradi funkcija `main` programa za testiranje. Drugi red uključuje zaglavlje, koje ako je definisan prethodni konfiguracioni makro, uključuje sva druga potrebna zaglavlja i odgovarajuću definiciju funkcije `main`.

Program za testiranje možemo da pokrećemo uz upotrebu velikog broja konfiguracionih opcija. U najvažnije opcije spadaju opcije za odabir testova koje je potrebno izvesti, što nam omogućava da se u slučaju većih kolekcija testova fokusiramo samo na određene podskupove.

Svaki test ima ime. Kao parametar pokretanja programa za testiranje možemo da navedemo ime testa koji je potrebno da se izvrši, ali možemo da koristimo i džoker znake. Takođe, možemo da odaberemo i testove koje ne želimo da izvodimo.

Pored imena, svaki test može da ima i jednu ili više *oznaka* (engl. *tag*) koje nam služe za grupisanje ili klasifikovanje testova. Na primer, oznaka može da predstavlja ime klase čije metode testiramo, ime skupa operacija ili bilo šta drugo. Ako među oznakama nekog testa navedemo i neku koja počinje tačkom (ili upravo oznaku tačka „`[.]`“), onda je test podrazumevano *skriven*, tj. izvodiće se samo ako to eksplicitno zatražimo odgovarajućim opcijama.

Na primer, sledeći argumenti određuju da je potrebno da se izvrše svi testovi sa oznakom `Razlomak`, osim onih čije ime počinje sa `Kons`:

```
... ~Kons* [Razlomak]
```

Pisanje testova

Moduli sa testovima se pišu sasvim jednostavno. Dovoljno je da se uključe zaglavlje biblioteke i odgovarajuća zaglavlja jedinice koda koja se testira i da se

napišu testovi. Na primer, jedan ispravan modul za testiranje bi mogao da izgleda ovako:

```
#include "catch2/catch.hpp"

TEST_CASE( "Jednostavan primer", "[Primeri]" ) {
    CHECK( 1 + 2 = 3 );
}
```

Osnovna jedinica organizacije je *test* (engl. *test case*). Jedan test može da obuhvata veći broj *provera*, koje mogu dodatno da se organizuju po *sekcijama*. Testovi mogu da se grupišu upotrebom *oznaka* (tagova). Test se definiše u bloku koji počinje makroom:

```
TEST_CASE( naziv_testa [, oznake] )
```

Prvi parametar je obavezan naziv testa. Drugi parametar je opciona niska sa oznakama. Svaka oznaka se navodi unutar uglastih zagrada, bez dodatnih separatora. U prethodnom primeru testu „`Jednostavan primer`“ je dodeljena oznaka `Primeri`.

U okviru jednog testa može da se navede veći broj *provera* (ili *pretpostavki*, engl. *assertion*). Bibilioteka *Catch2* podržava veći broj različitih provera ali se najčešće koriste dve najjednostavnije: `REQUIRE` i `CHECK`.

Provera `REQUIRE( uslov )` proverava da li je dati uslov ispunjen i u slučaju neuspeha prekida izvođenje testa u kome se nalazi. Provera `CHECK( uslov )` u osnovi radi isto, ali u slučaju neuspeha ne prekida izvođenje testa. Provera `CHECK` je praktičnija, zato što omogućava da uočimo više neispravnosti u istom testu. Međutim, ako mora da bude ispunjen neki uslov da bi mogle da se izvedu naredne provere, onda je neophodno da se koristi `REQUIRE` za proveravanje tog uslova. Na primer, ako želimo prvo da proverimo da li je pokazivač ispravan, pa da zatim proverimo da li objekat na koji pokazuje ima neka svojstva, onda pišemo provere poput:

```
REQUIRE( pRazlomak );
CHECK( pRazlomak->Imenilac() == 10 );
```

Pored ovih osnovnih provera postoje i mnoge druge, na primer:

- `REQUIRE_FALSE( uslov )`, `CHECK_FALSE( uslov )` – proveravaju da li je uslov netačan (slično kao kada se koristi negacija uslova u proverama `REQUIRE` i `CHECK`, ali negacija može da ometa automatsko razlaganje izraza, pa se preporučuje ovaj oblik);

- `REQUIRE_NOTHROW( izraz )`, `CHECK_NOTHROW( izraz )` – proveravaju da li izraz (ne) izbacuje izuzetak;
- `REQUIRE_THROWS( izraz )`, `CHECK_THROWS( izraz )` – proveravaju da li izraz (ne) izbacuje izuzetak;
- `REQUIRE_THROWS_AS( izraz, tip_izuzetka )`, `CHECK_THROWS_AS( izraz, tip_izuzetka )` – proveravaju da li izraz izbacuje izuzetak datog tipa;
- provere poklapanja (engl. *matcher*), pre svega za niske i kolekcije; na primer, može da se proveri da li se niska poklapa sa nekim šablonom
- i druge.

Jedan test može da sadrži više sekcija. Svaka sekcija predstavlja blok koji počinje makroom:

```
SECTION( naziv_sekcije [, opis_sekcije ] )
```

Programski kod koji se nalazi pre prve sekcije jednog testa predstavlja *inicijalizacioni kod*, koji se izvršava pre svake pojednačne sekcije (u engl. terminologiji testiranja se naziva *setup*). Slično, programski kod koji se nalazi posle poslednje sekcije predstavlja *deinicijalizacioni* kod, koji se izvršava posle svake pojedinačne sekcije (u engl. terminologiji testiranja se naziva *teardown*). Između sekcija ne bi trebalo da se nalazi nikakav programski kod.

Na primer, sledeći test sadrži dve sekcije, koje koriste istu početnu vrednost vektora `v`:

```
TEST_CASE( "Sekcije", "[Primeri]" ) {
    std::vector<int> v { 0, 1, 2 };
    SECTION( "Prvi" ) {
        CHECK( v.size() == 3 );
        CHECK( v[1] == 1 );
        v[1] = 101;
        CHECK( v[1] == 101 );
        v.pop_back();
        CHECK( v.size() == 2 );
    }
    SECTION( "Drugi" ) {
        CHECK( v.size() == 3 );
        CHECK( v[0] == 0 );
        CHECK( v[1] == 1 );
        CHECK( v[2] == 2 );
    }
}
```

Semantika sekcija je definisana tako da svaka sekcija predstavlja jedan alternativni tok izvršavanja testa. Zbog toga sekcije mogu da se nalaze i u okviru drugih sekcija, pri čemu važe ista pravila inicijalizacije i deinicijalizacije.

Napredne mogućnosti

Biblioteka *Catch2* ima još mnogo toga što ovde nećemo detaljnije opisivati. Neke od naprednijih mogućnosti biblioteke su:

- iako sekcije omogućavaju izvođenje relativno složenih testova, postoje i alternativne organizacije kao *testiranje vođeno ponašanjem* (engl. *Behavior Driven Development – BDD*), ili pravljenje posebnih celina (kompleta) testova (engl. *test fixtures*);
- testovi mogu da se pišu i u obliku šablona, tako da se jednom napisan test izvršava za različite konkretne tipove;
- postoje tzv. *generatori podataka* (engl. *data generators*), koji mogu da budu od pomoći pri pravljenju kolekcija podataka na kojima se izvodi testiranje, ali i da omoguće da se iste provere izvode na većem broju različitih primera;
- možemo da napišemo i svoju funkciju `main`, a da pri tome iskoristimo neke elemente izvođenja testova koje nudi biblioteka;
- možemo da koristimo više različitih generatora izveštaja (engl. *reporters*)
- i drugo.

9.4 Razvoj vođen testovima

Klasičan redosled aktivnosti pri razvoju softvera podrazumeva da se softver najpre projektuje, pa zatim implementira i na kraju testira. Takav redosled može da se primeni na svim nivoima razvoja, uključujući i projekat kao celinu, ali i svaku pojedinačnu razvijanu jedinicu koda.

Da bi testiranje moglo da se sprovede, neophodno je da se najpre napravi odgovarajući skup test primera. Da bi skup test primera bio dovoljno reprezentativan, potrebno je da test primeri budu usklađeni sa konkretnom implementacijom. Ali, da bi to bilo moguće, zaključujemo da test primere mora da pravi neko ko je dobro upoznat sa predmetom testiranja i potencijalnim slabim tačkama koje se testovima posebno ciljaju. Zato testove obično pravi neko ko je učestvovao u projektovanju ili implementiranju odgovarajućeg dela koda, ili specijalizovani stručnjak za testiranje, koji ima punu saradnju ostalih članova tima.

Slika 31 – Klasičan redosled aktivnosti pri razvoju softvera

Kao što vidimo, testiranje je složen postupak koji obuhvata nekoliko različitih poslova. U prethodnom pasusu smo naveli neke od poslova koji čine testiranje. Potpuniji spisak poslova obuhvata:

- određivanje ciljeva testiranja;
- prepoznavanje predmeta testiranja;
- analiziranje predmeta testiranja;
- određivanje nivoa detaljnosti testiranja;
- oblikovanje testova;
- ustanovljavanje očekivanih rezultata;
- implementiranje testova;
- izvršavanje testova;
- analiziranje neuspešnih testova
- i druge poslove.

Pravljenje testova na samom kraju razvoja nije nimalo jednostavan posao. Da li bi možda bilo jednostavnije i bolje da se testovi prave u nekom drugom trenutku?

Pogledajmo sada isečak iz jednog primera testa:

```
...
CHECK( Razlomak(1,2).Brojilac() == 1 );
CHECK( Razlomak(1,2).Imenilac() == 2 );
CHECK( Razlomak(2,1).Brojilac() == 2 );
CHECK( Razlomak(2,1).Imenilac() == 1 );
...
```

Čak i bez poznavanja deklaracije ili definicije klase `Razlomak` i makroa biblioteke *Catch2*, iz navedenog zapisa testa možemo da uočimo da konstruktor klase `Razlomak` ima za argumente vrednosti brojioca i imenioca, kao i da naši testovi proveravaju da li su međusobno usklađeni konstruktor i metodi koji izdvajaju vrednosti brojioca i

imenioca. Pri izvođenju tog zaključka smo napisane testove upotrebili kao vid dokumentacije. Zaista, testovi mogu da veoma dobro ilustruju ponašanje testiranih jedinica koda. Ako su testovi dobro napravljeni, onda oni mogu da opišu način upotrebe testiranih jedinica jednako dobro i precizno kao tehnička specifikacija.

Ako imamo u vidu da je uvek poželjno (pa i neophodno) da se pre implementacije funkcija i metoda napravi njihova specifikacija, onda bismo mogli da na samom početku, pre implementiranja programskog koda, najpre napišemo testove kao vid specifikacije, pa da tek onda pišemo implementaciju, a da na kraju izvedemo testiranje pomoću tako napisanih testova. Takva izmena redosleda aktivnosti može da ima više pozitivnih posledica:

- napisani testovi mogu da predstavljaju vid specifikacije jedinica koda koje ćemo tek da napišemo; štaviše, ako dovoljno jasno napišemo testove, onda obično ne moramo da žurimo da napišemo i odgovarajuću tekstualnu specifikaciju;
- nakon što napišemo testove kao vid specifikacije, program ili neće moći da se prevede ili testovi neće proći, zato što odgovarajuća jedinica koda nije uopšte napisana ili joj ponašanje nije implementirano u skladu sa novom „specifikacijom“, pa zato
- možemo da kažemo da nam je *cilj pisanja programskog koda* postalo *zadovoljavanje napisanih testova*.

Slika 32 – Redosled aktivnosti pri razvoju vođenom testovima

Takvom izmenom redosleda aktivnosti smo na praktičan način predstavili osnovnu ideju *razvoja vođenog testovima*. Osnovni principi razvoja vođenog testovima su da (1) testovi prethode kodu i da je pri tome (2) neophodan visok nivo sistematičnosti.

Za razliku od klasičnog pristupa razvoju softvera, koji promoviše testiranje softvera nakon razvoja, u slučaju razvoja vođenog testovima zahteva se da *testovi prethode kodu*, tj. da pre pisanja koda moramo:

- da isplaniramo kako je potrebno da se kod ponaša;
- da te planove pretočimo u vid specifikacije – u obliku testova;
- da testovima obuhvatimo sve uobičajene i granične slučajeve i
- da uvek kritički razmotrimo da li je potrebno da se testovima obuhvate i još neki specijalni slučajevi.

Na taj način se već pre započinjanja pisanja ili menjanja nekog dela koda dobija odgovarajuća kolekcija testova. Napisani testovi predstavljaju ciljne kriterijume kvaliteta koda. Posle toga, cilj pisanja produkcionog koda nije ništa drugo do zadovoljavanje novih postavljenih kriterijuma, a bez narušavanja već ranije dostignutih kriterijuma kvaliteta softvera.

Posle pisanja programskog koda, testovi se izvode (izvršavaju). Ako je testiranje uspelo (*testovi su prošli*), onda to znači da je programski kod zadovoljio postavljene kriterijume kvaliteta. Ako testiranje nije uspelo, onda programski kod nije dovoljno dobar i mora da se popravlja. Popravljanje koda se nastavlja sve dok se ne omogući uspešno testiranje.

Razvoj vođen testovima promoviše visok nivo sistematičnosti, do te mere da se nijedna funkcija programa ne razvija sve dok ne postoji test koji ne uspeva zbog njenog odsustva. Štaviše, nijedna linija koda se ne dodaje niti menja sve dok ne postoji test koji zbog nje ne uspeva.

Razvoj vođen testovima (engl. *Test Driven Development*) se odnosi primarno na testove jedinica koda, ali može da se primenjuje i šire, na primer na korisničke celine ili slučajeve upotrebe, pa čak i na određene testove integracije ili testove sistema.

Redosled koraka

1. Analiza i projektovanje

Svaka iteracija razvoja započinje analizom i projektovanjem (tj. planiranjem) novih funkcija, metoda ili klasa, ili potrebnih izmena postojećih funkcija, metoda ili klasa.

Imajući u vidu da će u narednom koraku da se naprave testovi kao vid formalne specifikacije, rezultat projektovanja i planiranja može da bude i neformalna specifikacija. Alternativno, ovaj korak može da se integriše sa narednim.

2. Pisanje testova kao formalne specifikacije zadatka

Kada znamo šta je potrebno da se razvija, onda možemo to i formalno da zapišemo – u obliku testova jedinica koda koje ćemo razvijati ili menjati. Pisanjem testova definišemo kako će izgledati interfejs novih jedinica koda i kako će one da se ponašaju.

Testovi koje napišemo u ovoj fazi vrlo često ne mogu ni da se prevedu, zato što još ne postoje odgovarajući metodi ili klase koji se u testovima upotrebljavaju.

3. Pisanje kostura koda

Nakon što su napisani testovi, piše se kostur programskog koda, koji omogućava da se testovi prevedu. To odgovara pisanju interfejsa odgovarajućih jedinica koda. Dodaju se nove funkcije, metodi i klase ili se menjaju postojeći interfejsi, radi usklađivanja sa novom specifikacijom. Ako je potrebno da novi metodi vraćaju neki rezultat, obično se u ovoj fazi privremeno implementiraju tako da vraćaju neku konstantnu vrednost.

Ovaj korak se završava proveravanjem da li su napravljeni svi neophodni metodi i klase. To se radi prevođenjem programa za testiranje – ako uspe da se prevede, onda je ovaj deo posla završen. Naravno, još uvek ne očekujemo da testovi uspešno prolaze, zato što još nije implementirano odgovarajuće ponašanje.

4. Pisanje programskog koda koji omogućava da testovi prođu

Posle pisanja kostura koda, potrebno je da se novi metodi, jedan po jedan, implementiraju tako da zadovolje „specifikaciju“, tj. testove. Tokom implementiranja mogu (i preporučuje se) da se dodaju novi testovi, koji odgovaraju različitim posebnim slučajevima. To je važno zato što se pri implementiranju koda najbolje sagledavaju potencijalne osetljive tačke. Dok su u prethodnim koracima mogli da se predvide neki granični slučajevi samo na osnovu interfejsa, sada mogu da se uoče i neki novi granični slučajevi, koji su posledica konkretne implementacije, pa zato i oni moraju da se testiraju.

Male iteracije

Preporučuje se da opisane iteracije razvoja budu što manje. Nije dobro da se prvo napiše veliki broj testova pa da se zatim implementira mnogo koda. Takav pristup ima više potencijalnih slabosti. Pre svega, testovi će moći da se izvrše tek kada budu napisani svi odgovarajući delovi koda, a ako to zahteva mnogo posla, onda je lako moguće da se usput napravi veći broj grešaka. Što je više grešaka koje moraju da se rešavaju, to je njihovo pronalaženje i otklanjanje složenije i zahteva više vremena.

Zato je mnogo bolje da se piše u malim iteracijama. Na početku iteracije se napiše mali broj novih testova, koji se odnose na samo jednu jedinicu koda. Zatim se piše ili menja ta jedinica koda. Kada testiranje pokaže da je jedinica koda dostigla traženi kvalitet, iteracija se završava i počinje nova.

Iteracije pisanja testova i koda bi trebalo da se se veoma brzo smenjuju, često na nekoliko minuta. Na taj način testovi i kod zajedno evoluiraju, tako da su testovi tek malo ispred koda.

Sistematičnost

Za uspešno sprovođenje razvoja vođenog testovima je od presudnog značaja da se testovi pišu sistematično:

- svaka karakteristika softvera mora da se pokrije testovima;
- svi granični slučajevi moraju da budu obuhvaćeni testovima;
- svaka linija koda mora da bude testirana;
- svaka naredba izbora i svaka petlja moraju da budu obuhvaćeni testovima; svaka grana mora da bude testirana;
- svaka operacija, funkcija i metod moraju da se testiraju;
- svaka polimorfna upotreba objekata mora da se testira na različitim klasama
- i drugo.

Kada se naknadno uoče bagovi u gotovom proizvodu, bilo pri holističkom testiranju ili pri upotrebi, to najčešće može da se objasni kao posledica nedovoljno sistematičnih testova – da su testovi bili dovoljno dobro definisani, oni bi na vreme ukazali na problem, umesto da se on kasnije ispoljava kao bag.

Testiranje jedinica koda se odnosi prvenstveno na javne metode klasa. Ne postoji opšta saglasnost oko toga da li je dobro da se pišu testovi jedinica koda i za privatne metode. Za razliku od javnog interfejsa koji bi trebalo da bude relativno stabilan, privatni elementi mogu da trpe određene značajnije promene ponašanja tokom razvoja, na primer zbog optimizacije, pa onda to povlači i češće održavanje testova. Drugi problem je tehničkog karaktera, zato što elementi biblioteka za testiranje obično nemaju neposredan pristup privatnim metodima naših klasa. Jedan način da se ovaj problem prevaziđe je da se obezbede prijateljske klase, koje mogu da pristupe privatnim elementima i koriste se samo radi testiranja, ali to predstavlja vid narušavanja enkapsulacije.

Neki privatni metodi su jednostavni i od propuštanja njihovog testiranja nas verovatno neće boleti glava. U nekim drugim slučajevima se privatni metodi koriste prilično neposredno u javnim metodima tako da se dobrim testiranjem javnih metoda praktično u potpunosti (iako implicitno) testiraju i privatni metodi, pa i tada nemamo problem. Ipak, u nekim slučajevima je privatni metod tako definisan da ne možemo da testiramo sve različite slučajeve njegove upotrebe samo testiranjem javnih metoda koji ga koriste. Tada moramo da proverimo (1) da li su možda ti

slučajevi zabranjeni (i ako jesu, onda možemo da popravimo robusnost uvođenjem pretpostavki, a ne testiranjem) i (2) da li bi možda takav metod trebalo da bude javan (zato što ima veće mogućnosti primene nego postojeći javni interfejs)? Tek ako na oba pitanja dobijemo negativan odgovor, onda bi trebalo da razmislimo o testiranju (i načinu testiranja) privatnog metoda.

Smer razvoja

Softver se obično razvija u jednom od dva smera – ili od vrha prema dnu, ili od dna prema vrhu. Ako se softver razvija od vrha prema dnu, onda se najpre razvijaju glavni algoritmi i koncepti, dok se implementacija detalja ostavlja za kasnije. Da bi programi mogli da se prevode i testiraju, umesto svih onih delova koji još nisu razvijeni prave se tzv. *umeci* (engl. *stubs*). Umetak je privremeni deo koda koji definiše interfejs ili druge metode, ali ih implementira samo *praznim* kodom. U slučaju potprograma koji ne izračunavaju rezultat, umetak najčešće ne radi doslovno ništa. U slučaju funkcija ili metoda koji moraju da vrate neki rezultat, umetak obično vraća neku konstantnu vrednost. Jedina namena umetaka je da omoguće da se klase, komponente i program prevedu. Naravno, tako prevedene klase, komponente i programi neće raditi ispravno, ali će bar biti moguće da se testiraju pojedini delovi implementacija, kao i da se proveri da li na visokom nivou algoritimi rade kako je predviđeno.

Ako se razvoj softvera odvija u suprotnom smeru, od dna prema vrhu, onda se za testiranje upotrebljavaju tzv. *izvođači* (engl. *drivers*). Izvođači su privremeni delovi koda koji povezuju manje funkcionalne celine isključivo radi testiranja. U odsustvu većih funkcionalnih celina produkcionog koda, čiji razvoj tek sledi, pravljenje izvođača je vro pristupačan način da se neki delovi povežu u celinu i testiraju.

9.4.1 Primer

Ilustrovaćemo koncepte razvoja vođenog testovima predstavljanjem nekoliko koraka razvoja klase `Razlomak`. Za implementiranje testova ćemo da koristimo biblioteku *Catch2*, verziju 2. Pretpostavićemo da je zaglavlje biblioteke u datoteci: `catch2/catch.hpp`.

Započećemo od prazne klase, koju ćemo implementirati u zaglavlju `razlomak.h`[38]:

```
class Razlomak
```

[38] Ako bi klasa `Razlomak` postala složenija, onda bi trebalo da se deo implementacije prebaci iz zaglavlja u datoteku `razlomak.cpp` i da se doda `razlomak.cpp` u naredbu za prevođenje. U našem primeru to nije neophodno.

```
{};
```

Testove naše klase ćemo da pišemo u datoteci `razlomak.test.cpp`:

```
#include "catch2/catch.hpp"
#include "razlomak.h"
```

Glavni program za testiranje implementiramo u datoteci `main.test.cpp`:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
```

Prevođenje i testiranje možemo da izvodimo korišćenjem prevodioca `g++` na `Linuxu`, pomoću sledećeg skripta zapisanog u datoteci `tst.sh`:

```
#!/bin/bash
g++ main.test.cpp razlomak.test.cpp -o test
if [ $? -eq 0 ]; then
    ./test
fi
```

ili korišćenjem paketa *Visual Studio* na operativnom sistemu *Windows*, pomoću skripta zapisanog u datoteci `tst.cmd`:

```
cl /EHsc main.test.cpp razlomak.test.cpp /Fetest.exe
@if not errorlevel 1 test.exe
```

Korak 1 – Konstruktor

Počinjemo od pravljenja konstruktora, koji kao argumente ima brojilac i imenilac. Najpre pišemo odgovarajuće testove:

razlomak.test.cpp

```
#include "catch2/catch.hpp"
#include "razlomak.h"

TEST_CASE( "Konstruktori klase Razlomak", "[Razlomak]" ) {
    Razlomak r(2,5);
    CHECK( r.Imenilac() == 5 );
    CHECK( r.Brojilac() == 2 );
}
```

Zatim pišemo konstruktor i dodajemo podatke `Brojilac_` i `Imenilac_`. Da bismo mogli da proverimo uspešnost konstruktora, potrebno je da napravimo i pristupne metode, kojima dohvatamo vrednosti brojioca i imenioca.

razlomak.h

```
class Razlomak
{
public:
    Razlomak( int b, int i )
        : Imenilac_(i),
          Brojilac_(b)
    {}

    int Imenilac() const
        { return Imenilac_; }

    int Brojilac() const
        { return Brojilac_; }

private:
    int Imenilac_;
    int Brojilac_;
};
```

Prevodimo i pokrećemo testove:

```
===============================================================================
All tests passed (2 assertions in 1 test case)
```

Korak 2 – Pozitivan imenilac

Dodajemo pretpostavku da imenilac nikada ne sme da bude ni nula ni negativan. Ako je negativan, promenićemo znak i imenioca i brojioca, a ako je nula, onda ćemo da izbacimo izuzetak.

Prvo dodajemo odgovarajuće testove. Dodajemo sekcije da bismo razdvojili različite aspekte testiranja konstruktora.

razlomak.test.cpp

```
#include "catch2/catch.hpp"
#include "razlomak.h"

TEST_CASE( "Konstruktori", "[Razlomak]" ) {
    SECTION( "Jednostavan konstruktor"){
        Razlomak r(2,5);
        CHECK( r.Imenilac() == 5 );
        CHECK( r.Brojilac() == 2 );
    }
    SECTION( "Konstruktor sa imeniocem nula"){
        CHECK_THROWS_AS( Razlomak( 3, 0 ), std::domain_error );
    }
    SECTION( "Konstruktor sa negativnim imeniocem"){
```

```
        Razlomak r(2,-5);
        CHECK( r.Imenilac() == 5 );
        CHECK( r.Brojilac() == -2 );
    }
}
```

Prevodimo testove i pokrećemo ih:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.exe is a Catch v2.0.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
Konstruktori
  Konstruktor sa imeniocem nula
-------------------------------------------------------------------------------
razlomak.test.cpp(10)
.....................................................................

razlomak.test.cpp(11): FAILED:
  CHECK_THROWS_AS( Razlomak( 3, 0 ), std::domain_error )
because no exception was thrown where one was expected:

-------------------------------------------------------------------------------
Konstruktori
  Konstruktor sa negativnim imeniocem
-------------------------------------------------------------------------------
razlomak.test.cpp(13)
.....................................................................

razlomak.test.cpp(15): FAILED:
  CHECK( r.Imenilac() == 5 )
with expansion:
  -5 == 5

razlomak.test.cpp(16): FAILED:
  CHECK( r.Brojilac() == -2 )
with expansion:
  2 == -2

===============================================================================
test cases: 1 | 1 failed
assertions: 5 | 2 passed | 3 failed
```

Zatim popravljamo konstruktor tako da pretpostavka o pozitivnosti imenioca bude uvek zadovoljena.

razlomak.h

```
class Razlomak {
```

```
...
    Razlomak( int b, int i )
        : Imenilac_(i),
          Brojilac_(b)
    {
        if( !Imenilac_ )
            throw std::domain_error( "Imenilac je nula!" );
        else if( Imenilac_ < 0 ){
            Imenilac_ = - Imenilac_;
            Brojilac_ = - Brojilac_;
        }
    }
...
};
```

Prevodimo testove i pokrećemo ih:

```
===============================================================
All tests passed (4 assertions in 1 test case)
```

Korak 3 – Poređenje jednakosti

Dodajemo operator ==. Najpre pišemo testove, pa onda i operator. Zatim prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Poredjenje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5) == Razlomak(2,5) );
    CHECK( Razlomak(2,5) == Razlomak(-2,-5) );
    CHECK( Razlomak(2,5) == Razlomak(4,10) );
}
```

razlomak.h

```
class Razlomak {
...
    bool operator==( const Razlomak& r ) const
    {
        return Imenilac() == r.Imenilac()
            && Brojilac() == r.Brojilac();
    }
...
};
```

Primećujemo da poslednji test ne prolazi. Dva razlomka koja poredimo su suštinski ista, ali se njihovi brojioci i imenioci razlikuju. Da bi prošao i taj test, moramo da popravimo poređenje[39].

```
class Razlomak {
...
   bool operator==( const Razlomak& r ) const
   {
      return Imenilac() * r.Brojilac() == r.Imenilac() * Brojilac();
   }
...
};
```

Korak 4 – Pisanje i čitanje

Pri implementaciji pisanja i čitanja trebalo bi da vodimo računa da ove dve operacije budu usaglašene, tako da zapis koji pravimo pri ispisivanju može kasnije da se pročita. Počinjemo od pisanja. Pri testiranju koristimo `std::ostringstream`.

razlomak.test.cpp

```
...
#include <sstream>
...
TEST_CASE( "Pisanje i citanje", "[Razlomak]" ) {
    std::ostringstream str;
    str << Razlomak(2,5);
    CHECK( str.str() == "2/5" );

    ostr.str("");
    ostr.clear();
    ostr << Razlomak(2,-5);
    CHECK( ostr.str() == "-2/5" );
}
```

razlomak.h

```
class Razlomak { ... };

inline std::ostream&
operator<<( std::ostream& ostr, const Razlomak& r )
{
    ostr << r.Brojilac() << '/' << r.Imenilac();
    return ostr;
}
```

[39] Alternativno rešenje je da pri konstrukciji uvek uprostimo razlomak.

Zatim prelazimo na čitanje.

razlomak.test.cpp

```
...
#include <sstream>
...
TEST_CASE( "Pisanje i citanje", "[Razlomak]" ) {
    ...
    std::istringstream istr( ostr.str() );
    Razlomak r(3,4);
    istr >> r;
    CHECK( r == Razlomak(-2,5) );
}
```

razlomak.h

```
class Razlomak {
...
};

inline std::istream& operator>>( std::istream& istr, Razlomak& r )
{
    int i,b;
    char c;
    istr >> b >> c >> i;
    if( istr ){
        if( c == '/' )
            r = Razlomak( b, i );
        else
            istr.setstate( std::ios::failbit );
    }
    return istr;
}
```

Sada znamo kako je implementirano čitanje, pa dodajemo još odgovarajućih testova i zatim prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Pisanje i citanje", "[Razlomak]" ) {
    ...
    istr.str( "7;8" );
    istr.clear();
    istr >> r;
    CHECK( !istr );
    CHECK( r == Razlomak(-2,5) );

    istr.str( "7/a8" );
    istr.clear();
    istr >> r;
```

```
    CHECK( !istr );
    CHECK( r == Razlomak(-2,5) );

    istr.str( "7/8" );
    istr.clear();
    istr >> r;
    CHECK( istr );
    CHECK( r == Razlomak(7,8) );
}
```

Korak 5 – Poređenje „manji od“

Pišemo testove za operator poređenja „<“. Zatim pišemo operator, prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Poredjenje", "[Razlomak]" ) {
    ...
    CHECK( Razlomak(2,5) < Razlomak(3,5) );
    CHECK( Razlomak(2,5) < Razlomak(2,4) );
    CHECK( Razlomak(-3,5) < Razlomak(-2,5) );
    CHECK( Razlomak(-2,5) < Razlomak(2,5) );
}
```

razlomak.h

```
class Razlomak {
...
    bool operator<( const Razlomak& r ) const
    {
        return Brojilac()*r.Imenilac() < r.Brojilac()*Imenilac();
    }
...
};
```

Korak 6 – Skraćivanje razlomka

Dodajemo metod `Skrati`, koji skraćuje razlomak. Prvo pišemo testove, pa implementiramo metod.

razlomak.test.cpp

```
...
TEST_CASE( "Skracivanje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(2,5).Skrati().Brojilac() == 2 );

    CHECK( Razlomak(25,15).Skrati().Imenilac() == 3 );
    CHECK( Razlomak(25,15).Skrati().Brojilac() == 5 );
```

```
    CHECK( Razlomak(15,25).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(15,25).Skrati().Brojilac() == 3 );

    CHECK( Razlomak(15,-25).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(15,-25).Skrati().Brojilac() == -3 );
}
```

razlomak.h

```
class Razlomak {
...
    Razlomak& Skrati()
    {
        int n = nzd( abs(Brojilac_), Imenilac_ );
        Imenilac_ /= n;
        Brojilac_ /= n;
        return *this;
    }

private:
    static unsigned nzd( unsigned n1, unsigned n2 )
    {
        if( n1 < n2 )
            swap( n1, n2 );
        else if( !n1 )
            return 1;
        while( n2 ){
            unsigned r = n1 % n2;
            n1 = n2;
            n2 = r;
        }
        return n1;
    }
...
};
```

Korak 7 – Sabiranje i oduzimanje

Kao završni deo ovog primera, dodajemo operatore sabiranja i oduzimanja. I u ovom slučaju, prvo pišemo testove, pa implementiramo operatore, pa onda prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Sabiranje i oduzimanje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5) + Razlomak(-1,5) == Razlomak(1,5) );
    CHECK( Razlomak(2,5) - Razlomak(-1,5) == Razlomak(3,5) );
    CHECK( Razlomak(2,5) + Razlomak(1,2) == Razlomak(9,10) );
    CHECK( Razlomak(2,5) - Razlomak(1,2) == Razlomak(-1,10) );
```

```
    CHECK( Razlomak(3,10) + Razlomak(4,15) == Razlomak(17,30) );
    CHECK( Razlomak(3,10) - Razlomak(4,15) == Razlomak(1,30) );
    CHECK( Razlomak(3,10) + Razlomak(-4,15) == Razlomak(1,30) );
    CHECK( Razlomak(3,10) - Razlomak(-4,15) == Razlomak(17,30) );
}
```

razlomak.h

```
class Razlomak {
...
    Razlomak operator+( const Razlomak& r ) const
    {
        return Razlomak(
                    Brojilac() * r.Imenilac()
                        + Imenilac() * r.Brojilac(),
                    Imenilac() * r.Imenilac()
                ).Skrati();
    }

    Razlomak operator-( const Razlomak& r ) const
    {
        return Razlomak(
                    Brojilac() * r.Imenilac()
                        - Imenilac() * r.Brojilac(),
                    Imenilac() * r.Imenilac()
                ).Skrati();
    }
...
};
```

Rezultat testiranja:

```
===============================================================================
All tests passed (36 assertions in 5 test cases)
```

U prethodnih nekoliko koraka smo predstavili kako bi trebalo da teče razvoj vođen testovima. Primer je sasvim jednostavan, ali smo imali priliku da vidimo i specijalne i granične slučajeve, sa imeniocem koji je manji od nule ili jednak nuli. Zadržali smo se na elementarnim mogućnostima biblioteke *Catch2*, zato da bi pažnja bila usmerena na postupak, a ne na alat. Primetimo da smo testirali sve javne elemente klase `Razlomak`, ali ne i privatni statički metod `nzd`.

9.5 Uloga testova jedinica koda

Osnovna uloga testova jedinica koda je proveravanje da li se napisani delovi programskog koda ponašaju u skladu sa odgovarajućom tehničkom specifikacijom. Oni predstavljaju važno sredstvo za obezbeđivanja kvaliteta softvera. Međutim,

testovi jedinica koda mogu da imaju i neke dodatne uloge u procesu razvoja softvera.

Testovi predstavljaju vid parcijalne verifikacije softvera. Kolekcija testova omogućava programeru da proverava da li jedinica koda radi ispravno u nekim unapred definisanim okolnostima. U opštem slučaju testovi ne mogu da obuhvate sve moguće okolnosti upotrebe, pa zato ne mogu da predstavljaju potpunu verifikaciju softvera. Ipak, i parcijalna verifikacija je veoma važna za obezbeđivanje željenog kvaliteta softvera.

Testiranje programa može da bude veoma efikasno sredstvo
da se pokaže postojanje bagova,
ali je beznadežno neadekvatno za pokazivanje da ih nema

Edsher Daikstra

Testovi jedinica koda bi trebalo da proveravaju ispravnost svake pojedinačne jedinice koda: funkcije, metoda, klase i drugih vrsta jedinica koda. Mnogi zameraju da se pisanjem testova usporava razvoj produkcionog koda, ali to usporavanje ima i pozitivan uticaj na razvoj – sprečava se srljanje. Testovi jedinica koda, posebno ako se pišu na način koji propisuje razvoj vođen testovima, omogućavaju programeru da pre pisanja svake pojedinačne jedinice koda zastane i zapita se „Šta je potrebno da napravim?“ i „Kako je potrebno da se ponaša?“. Na taj način se sprečava lakomisleno zaletanje u pisanje niza možda nepotrebnih operacija.

Razvoj vođen testovima omogućava da se blagovremeno uoče eventualne nepotpunosti ili neispravnosti implementacije, ali i dizajna pa i konceptualnog projekta. Tako veliki doprinos testova potiče otud što dok piše testove programer mora da razmišlja i kao korisnik jedinice koda koju testira, a ne samo kao njen implementator. Takva promena ugla posmatranja može značajno da pomogne pri uočavanju eventualnih slabosti projektovanog interfejsa i drugih grešaka na nivou koncepta ili projekta. Pri pisanju testova u prvom planu su interfejs testirane jedinice koda i apstrakcija njenog ponašanja. Sa druge strane, pri implementiranju se može nehotice izgubiti iz vida značaj interfejsa, pa i apstrakcije ponašanja, zato što je programer prinuđen da se bavi sasvim konkretnim problemima i pojedinostima u kodu. Na ovaj način testovi doprinose pisanju *lako upotrebljivog* koda.

Da bi jedinice koda mogle da se dobro testiraju, one moraju da budu jasno uobličene i uokvirene. Veoma je teško napraviti dobar skup testova za jedinicu koda u kojoj postoji slaba kohezija elemenata, pa njeno ponašanje nije jasno i dobro uobličeno. Blagovremeno testiranje i posebno razvoj vođen testovima, teraju programera da sve jedinice koda što bolje oblikuje i međusobno razgraniči, tako da svaka ima jasno određeno ponašanje i smisao postojanja. Na taj način se vrši dodatni

pritisak na programera da pravi bolje dizajniran kod, jer samo *dobro dizajniran kod* može da bude i *lako proverljiv*.

Testovi jedinica koda su posebno važni za primenu refaktorisanja. Refaktorisanje je postupak unapređivanja dizajna koda ali tako da se ne menja njegovo ponašanje (vidi poglavlje *10 - Refaktorisanje*). Ključni problem je: kako da budemo sigurni da menjanjem dizajna nismo promenili i ponašanje? Ako postoje dobro implementirani testovi, onda je odgovor jednostavan – posle svake promene koda izvedemo testiranje i tako proverimo da li smo slučajno izmenili ponašanje.

Ako je u nekoj kasnijoj fazi razvoja potrebno da se menja ranije napisan kod, onda jedan od otežavajućih faktora može da bude otežana proverljivost – kako napraviti neophodne promene ponašanja, a da se ne promene i još neki aspekti ponašanja? Kao i u slučaju refaktorisanja, i za ovaj problem rešenje su nam testovi jedinica koda. Oni omogućavaju da se lako uoče eventualne greške u kodu nastale prilikom naknadnih izmena.

Jedna od najvažnijih „sporednih" uloga testova je da oni predstavljaju vid dokumentacije. Testovi jedinica koda uvek imaju oblik primera ispravne upotrebe interfejsa, a ako su dobro sistematizovani, onda mogu da predstavljaju i oblik formalne specifikacije testirane jedinice koda. Oni opisuju uslove funkcionisanja i različite načine upotrebe interfejsa jedinice koda. Posebno, testovi obično dobro ilustruju različite granične slučajeve. Jedna od najznačajnijih karakteristika ovog vida dokumentacije je *stalna ažurnost*. Jedan od osnovnih problema sa skoro svim drugim vidovima dokumentacije je neažurnost. Ali testovi jedinica koda ne mogu da budu neažurni, jer inače ne bi uspešno prolazili. Ako se kolekcija testova redovno održava i raste uporedo sa programskim kodom, onda ona pored vida neformalne verifikacije predstavlja i *najkompletniji* i *najažurniji* vid dokumentacije.

Testovi jedinica koda predstavljaju veliku pomoć i u procesu debagovanja. Svaki put kada se naknadno uoči neka neispravnost, koja nije prepoznata postojećim testovima, to može da bude signal da postojeći testovi nisu dovoljno dobri. Onda se obično pravi novi test, ili više testova, koji ne prolaze zbog postojanja baga. Slično kao u slučaju razvoja vođenog testovima, na ovaj način i debagovanje može da se svede na menjanje (tj. na popravljanje) koda radi zadovoljavanja testova. Štaviše, za razliku od privremenih provera, koje se često koriste pri debagovanju, testovi jedinica koda mogu da ostanu kao trajne provere i potvrde ostvarene ispravnosti.

9.6 Umesto zaključka

Danas se smatra da je programski kod *zastareo*, ako ne obuhvata odgovarajuće testove jedinica koda. Danas testovi jedinica koda nisu samo osnovno sredstvo za proveru ispravnosti napisanog programskog koda, već i veoma važan vid dokumentacije. Ako nemamo odgovarajuće testove jedinica koda, veliko je pitanje da

li zaista znamo kako se neki programski kod ponaša i u kojoj meri odgovara onome što piše u pratećoj tekstualnoj dokumentaciji, koja nam dolazi uz taj programski kod. Da li je implementirano sve što je bilo predviđeno? Da li je možda program naknadno izmenjen tako da više nije u skladu sa dokumentacijom?

Tehnike za implementiranje i izvršavanje testova su postale neizostavni deo ozbiljnih razvojnih alata, a sami testovi obavezno gradivo na iole naprednijim programerskim obukama. Jedna od knjiga koje su ostvarile veliki uticaj u tom smeru je knjiga Kenta Beka iz 2002. godine [*Back 2002*]. Dobar pregled različitih biblioteka za testiranje jedinica koda za programski jezik C++ napisao je Noel Lopis [*Llopis 2010*]. Iako je to relativno star pregled (iz 2010. godine), iz njega se mogu videti osnovni pristupi razvoju biblioteka za testiranje.