

Komponentno i integraciono testiranje u programskom jeziku C++

David Gavrilović 1100/2019

15. januar 2020.

Sažetak

U ovom radu će biti obrađene različite vrste testiranja koda od kojih će akcenat biti na komponentnom testiranju i integracionom testiranju. Za ove vrste testiranja će biti prikazani i primeri rađeni u programskom jeziku C++ uz korišćenje određenih biblioteka. Pored pomenutih vrsta testiranja, biće obrađene i druge, ali ne toliko detaljno, poput testiranja jedinica koda, sistemskog testiranja kao i testova prihvatljivosti.

Sadržaj

1	Uvod	2
2	Komponentno testiranje	2
3	Integraciono testiranje	4
3.1	Pristup odozdo na gore	4
3.2	Pristup odozgo na dole	5
3.3	Modifikovani odozgo na dole pristup	5
4	Primeri testiranja u C++ jeziku	6
4.1	Primeri komponentnog testiranja	6
4.2	Primeri integracionog testiranja	9
5	Zaključak	13
	Literatura	14

1 Uvod

Moguće je testirati različite delove sistema. Na primer, možemo testirati pojedinačne komponente, neku grupu komponenti povezanih na neki način ili ceo sistem. Testove možemo podeliti po tome šta testiraju i ta podela se naziva podela po nivoima testiranja.

Po pomenutoj podeli postoje jedinični testovi, testovi komponenti, integracioni testovi, testovi sistema kao i testovi prihvatljivosti. Prvi se još nazivaju i Junit (eng. *Unit*) testovi i koriste se za proveru tačnosti manjih delova sistema. Ti delovi, u zavisnosti od konteksta, na primer mogu biti podprogrami ili klase. Za ovu vrstu testiranja je zaslužan programer. [1]

Komponentno testiranje (eng. *Component testing*) je testiranje komponenti koje nastaju spajanjem više jedinica koda. Te jedinice su već testirane pojedinačno, ali se u ovom testiranju testira da li one pravilno funkcionišu zajedno. Za ovu vrstu testiranja mogu biti zaduženi programeri ali su to češće testeri. Komponenta koja se testira se testira zasebno, pa ova vrsta testiranja podseća na jedinično testiranje ali na višem nivou. [1]

Integraciono testiranje (eng. *Integration testing*) se koristi da bi se proverile veze između različitih komponenti koje zajedno čine neki deo sistema. Dakle, proverava se da li komponente komuniciraju i funkcionišu zajedno na ispravan način. Ovu vrstu testiranja sprovode testeri. Primećimo da u ovu definiciju donekle spada i komponentno testiranje, ali se ono odnosi na niži nivo, iz tog razloga što se za komponentu integrišu različite jedinice koda. [1]

Sistemska testiranje (eng. *System testing*) se koristi da bi se ispitala funkcionalnost celog sistema. Cilj je da se raznim metodama proveri da li ceo sistem ispunjava zadatke koje je zadao klijent. Ovu vrstu testiranja, kao i prošlu, sprovode testeri. Pored funkcionalnosti ovim testiranjem se mogu ispitati i neki nefunkcionalni zahtevi. [1]

Nakon ovih faza postoje i testovi prihvatljivosti (eng. *Acceptance tests*) i to su testovi koje sprovode korisnici da bi se uverili da se napravljen sistem izvršava pravilno i na željeni način.

Svaka od ovih faza je bitna za testiranje i one se sprovode navedenim redosledom, od jediničnog pa do sistemskog testiranja i testova prihvatljivosti. To ima smisla iz tog razloga što ne želimo da testiramo sistem ako ne radi neka njegova komponenta. Međutim, ponekad se, zbog nedostatka vremena, koriste i drugačiji pristupi gde se na primer prvo ispituje kako radi sistem a onda neke njegove komponente po potrebi.

U ovom radu neće biti detaljno obrađeni svi od pomenutih načina testiranja već samo dva, komponentno i integraciono testiranje. Pored toga će u nastavku biti prikazano i kako je moguće i uz pomoć kojih biblioteka, izvršavati ove vrste testiranja u programskom jeziku C++.

2 Komponentno testiranje

Komponentno testiranje (naziva se još i testiranje modula (eng. *Module testing*)) se sprovodi odmah nakon stvaranja komponente i nakon što su jedinice koje se nalaze u toj komponenti testirane (za šta se koristi jedinično testiranje). Bitno je dobro testirati komponente zato što se nakon komponentnog testiranja sprovodi integraciono testiranje. Što su nam komponente bolje konstruisane to će se manje grešaka primetiti u integracionom testiranju i te greške će se više odnositi na okruženje nego na pojedinačne komponente.

Razlikujemo dve različite vrste komponentnog testiranja, izolovano (eng. *Component testing in small* - CTIS) i neizolovano (eng. *Component testing in large* - CTIL). Prva vrsta se odnosi na komponente koje ne zavise od nekih drugih komponenti, dok se druga koristi kada rad jedne komponente zavisi od rada neke druge komponente pa je stoga nije moguće izolovano testirati. Napomenimo da se ovde i dalje testira samo jedna komponenta, dok se druge komponente od koje ona zavisi ne testiraju. Ovo može da stvori probleme zato što je ponekad potrebno testirati neku komponentu kojoj rad nekih delova zavisi od delova neke druge komponente koja još uvek nije završena. Ovo može da se zaobiđe dodavanjem lažnih (eng. *dummy* ili *mock*) delova komponenti (na primer funkcija ili klasa). Ti delovi se zovu stabovi (eng. *stubs*) i drajveri (eng. *drivers*). [2]

Stabovi se koriste kada je potrebno testirati delove višeg nivoa u softveru kada delovi nižeg nivoa koji su mu potrebni nisu završeni. Drajveri su suprotni od stabova pa se koriste za testiranje nižih delova sistema kada su delovi viših nivoa kojima su ti niži delovi potrebni nezavršeni. Dakle, namena im je ista, samo se koriste u različitim situacijama. [3]

Ako je komponenta koja se testira nezavisna, testeri je mogu testirati, a ako je zavisna od nekih drugih, potrebno je prvo napraviti stabove i drajvere i onda izvršiti testiranje. U drugom slučaju su za testiranje zaduženi programeri kojima je takođe posao da naprave pomenute lažne komponente. [1]

Ova vrsta testiranja se često zamenjuje sa jediničnim iz tog razloga što oba testiraju nekakav zaseban deo koda. Razlika je u tome što jedinično testiranje obuhvata jedinice, a to su manji delovi koda koji se mogu testirati i testira prvenstveno njihovu tačnost, dok komponentno testira da li veći i složeniji delovi, koji su sačinjeni od jedinica, rade pravilno. Dakle, komponentno se izvršava na višem nivou integracije. Kao što je već napomenuto, jedinično testiranje se sprovodi pre komponentnog, u toku pisanja softvera. Takođe, jedinično testiranje je posao programera, dok komponente često testiraju testeri.

Pored jediničnog, komponentno testiranje ima sličnosti i sa integracionim iz tog razloga što se zajedno testiraju jedinice koje su spojene u jednu komponentu i proverava se kako one funkcionišu zajedno.

Kao primer komponentnog testiranja navedena je sledeća situacija. Potrebno je napraviti neku veb aplikaciju koja ima veliki broj komponenti. Jedna od tih komponenti je stranica koja služi za logovanje i želimo da je testiramo. Može se testirati šta se dešava kada korisnik unese podatke u ispravnom formatu kao i šta se dešava kada ih unese u neispravnom, poput nedozvoljenih karaktera u korisničkom imenu ili pogrešno unete dužine lozinke. Ovo se dešava nakon što su jedinice koje čine stranicu (tekstualna polja i dugmići na primer) već testirane. Testovi mogu da uključe i proveru da li uneto korisničko ime postoji kao i da li mu je uneta šifra odgovarajuća. Da bi se ovo testiranje izvršilo potrebno je te podatke proslediti nekoj drugoj komponenti koja može da proveri da li je pristup uspešan ili ne. Ta komunikacija se može lažirati tako da ta komponenta nazad pošalje samo neku poruku o uspešnosti pristupa. Takođe, ako veb stranica koju nakon uspešnog logovanja korisniku treba prikazati ne postoji, onda se ona može lažirati. Na primer, umesto nje moguće je prikazati neku drugu stranicu sa porukom da je pristup uspešan.

3 Integraciono testiranje

Integraciono testiranje se sprovodi nakon što je svaka komponenta testirana zasebno i proverava da li komponente funkcionišu pravilno kada se se spoje u jednu grupu.

Ovu vrstu testiranja sprovode isključivo testeri i ona spada u testove crne kutije. Kod takve vrste testiranja tester ne poznaje unutrašnju strukturu onoga šta testira, bila to neka komponenta ili ceo sistem, već samo njegovu funkcionalnost. Dakle, tester zna koji rezultat bi trebalo dobiti, ali mu nije poznato na koji način. Obuhvata proveru izlaza iz sistema za date ulaze. Koristi se da bi se testiralo ponašanje aplikacije ili nekog njenog dela kao i za proveru da li aplikacija ispunjava sve što se od nje zahteva. [1]

Integraciono testiranje ne testira samo funkcionalnost softvera već stavlja akcenat na testiranje relacija i komunikacije između komponenti kao i načina na koji one sarađuju. Glavni fokus integracionog testiranja su pojmovi poput arhitekture i načina na koji je konstruisan softver. [1]

Ova vrsta testiranja je bitna iz nekoliko razloga. Prvo, vrlo je verovatno da su komponente koje se testiraju zajedno napravljene od strane različitih programera pa je stoga potrebno proveriti da li one pravilno funkcionišu zajedno. Drugi razlog je taj što komponente razmenjuju neke podatke koje u tom procesu razmene menjaju, tako da jedna komponenta može da primi nešto što joj može stvoriti neke probleme. Takođe, u sistem mogu da spadaju i neki alati koje nije razvijao tim koji radi na projektu (na primer nekakav *API*), već neko drugi. U tom slučaju treba proveriti da li ti alati primaju podatke koje bi trebalo da primaju, kao i da li oni kao izlaz generišu željene podatke ili poruke. Pored toga, moguće je da neke komponente ili jedinice nisu testirane dovoljno dobro i da zbog toga postoji neka nedetektovana greška. U tom slučaju će se integracionim testiranjem te greške primetiti. [4]

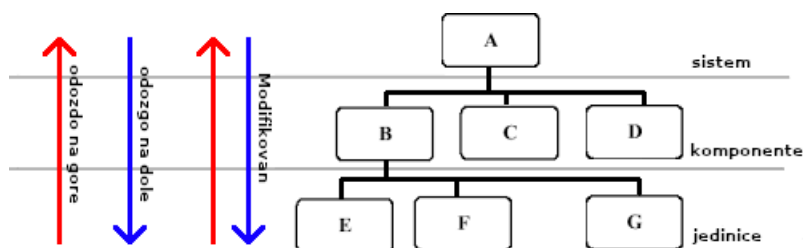
Kao primer se može navesti ista aplikacija navedena kao primer u sekciji za komponentno testiranje. Jedna komponenta je veb stranica koja služi za logovanje, dok je druga, nazovimo je profil korisnika, takva da nju treba prikazati nakon uspešnog logovanja. U integracionom testiranju želimo da proverimo da li se prilikom uspešnog logovanja korisniku stvarno prikazuje ispravna profil strana, kao i kakva je komunikacija između te dve komponente ali i kakva je razmena podataka sa bazom.

Moguće je integrisati sve komponente odjednom i onda pokrenuti testiranje. Ovo nije baš dobar pristup za velike sisteme zato što se greška (ako postoji) teško pronalazi, ali je dobar za male, koji imaju mali broj komponenti. Iz tog razloga se integraciono testiranje često obavlja tako što se integrišu komponente jedna po jedna i tako testiraju, pa je mnogo lakše locirati grešku.

Postoji nekoliko različitih vrsta integracionog testiranja iz tog razloga što postoje različiti načini integracije softvera. Postoje tri različita pristupa, odozdo na gore, odozgo na dole i modifikovani odozgo na dole. Vrste se mogu videti na slici 1.

3.1 Pristup odozdo na gore

Ovaj pristup prvo testira najniže delove softvera (jedinice i module) i kreće se ka vrhu integracijom manjih celina u veće. Takođe se naziva i klasični pristup zato što je prirodno krenuti od manjih ka većim celinama. Za najmanje celine je potrebno pisati i stabove i drajvere, dok se



Slika 1: Vrste integracionog testiranja

za veće celine mogu pisati samo dražveri ukoliko su potrebni, zato što se već postojeće, manje celine mogu upotrebiti kao stabovi.

Teorijski ovo je dobar pristup ali se praksi u teško implementira. Dobra strana ovog pristupa je ta što se testira svaka jedinica ili komponenta, i samo ako je ispravna može učestvovati u integraciji. Takođe, sistem je robusniji ako se potvrdi da njegovi pojedinačni delovi rade ispravno. [1]

Loše strane su to što se najvažnije (veće) komponente testiraju poslednje, što može dovesti do toga da jednostavno nema vremena da se one temeljno testiraju tako da ceo sistem možda neće raditi kako treba. Komponente se kreiraju jedna po jedna što može dovesti do toga da se puno vremena potroši na testiranje, što može poremetiti rokove u razvoju softvera. I na kraju, moguće je da će biti potrebno napisati veliki broj stabova i dražvera, koji se na kraju odbacuju, na šta se takođe može potrošiti dosta vremena. Pored toga mora se paziti da oni ne izazovu neke dodatne probleme. [1]

3.2 Pristup odozgo na dole

Pristup odozgo na dole je suprotan od prethodnog, što se lako može zaključiti iz naziva. Prvo se testiraju veći delovi sistema dok se najmanje komponente testiraju poslednje. Obično se prvo testira korisnički interfejs. U ovom pristupu dražveri verovatno neće biti potrebni zato što se već testirani delovi iz višeg nivoa mogu upotrebiti umesto njih, ali će zbog toga vrlo verovatno biti potrebno implementirati stabove.

Prednosti su te što je moguće veoma rano zaključiti da li sistem ispunjava zadate zahteve kao i da li sistem ima neke propuste u dizajnu. Takođe nema trošenja vremena na pisanje dražvera.

Naravno, ovaj sistem ima i mane od kojih je najveća ta što se jedinice i komponente često ne testiraju pojedinačno što može dovesti do nekih propusta. Moguće je i da će pisanje stabova potrošiti dosta vremena. [1]

3.3 Modifikovani odozgo na dole pristup

Ovo je kombinacija prethodna dva pristupa i pokušaj da se njihove negativne karakteristike kombinovanjem uklone. Integracija će i dalje biti izvršena odozgo na dole ali će se u ovom pristupu neke jedinice i komponente koje su definisane kao kritične testirati zasebno, pre integracije. Ovo znači da se neće testirati sve jedinice što doprinosi uštedi na vremenu u odnosu na odozdo na gore pristup ali neke jedinice i komponente (one najbitnije) će ipak biti testirane, što dovodi do poboljšanja u odnosu na pristup odozgo na dole.

Pošto je ovo i dalje odozgo na dole pristup potrebno je napisati potencijalno veliki broj stabova. Pored njih je potrebno napisati i drajvere za bitnije delove sistema koji se testiraju. Takođe može doći do potencijalnih problema pri određivanju kritičnih, najbitnijih delova sistema, što se radi prilikom dizajna softvera. [1]

Ovaj pristup može dovesti do toga da se sistem testira više od jedan put, jednom odozgo na dole prilikom integracije, a nakon toga odozdo na gore, kada se testiraju najbitnije komponente. To znači da je ova vrsta integracionog testiranja sporija od klasičnog odozgo na dole pristupa, ali i dalje daje vremenski bolje rezultate od odozdo na gore metoda.

4 Primeri testiranja u C++ jeziku

Postoji veliki broj jezika i veliki broj alata koji se mogu upotrebiti za testiranje softvera. U ovoj sekciji će biti prikazani primeri komponentnog i integracionog testiranja u programskom jeziku C++. Ovaj jezik podržava neke testove pomoću funkcija iz zaglavlja `<cassert>` ali se testiranje može izvršiti i pomoću velikog broja raznih biblioteka. Ove biblioteke omogućavaju automatizaciju procesa testiranja i mogu se koristiti za testiranje raznih nivoa softvera.

Svaka vrsta testiranja se izvodi u nekoliko faza. Prva faza je planiranje i u njoj se određuje šta će se sprovesti testiranjem i uz pomoć kojih alata. Druga je analiza i implementacija testova, gde se određuje način na koji se testira softver i gde se testovi kreiraju. U trećoj fazi, koja se zove primena testova, se primenjuju testovi koji su kreirani u prethodnoj i poslednja je evaluacija tih testova, gde se posmatraju dobijeni rezultati.

4.1 Primeri komponentnog testiranja

Komponentno ili modul testiranje služi da bi se testirala neka komponenta zasebno. Isto se može reći i za jedinično testiranje, samo su jedinice znatno manje od komponenti. Iz tog razloga se za komponentno testiranje mogu koristiti neke od biblioteka koje su napravljene za testiranje jedinica. Napomenimo i da ne postoji neka definicija šta je tačno komponenta jednog sistema pa se taj pojam razlikuje od sistema do sistema, u zavisnosti od dizajna softvera.

Neke od najkorišćenijih biblioteka za jedinično testiranje za C++ programski jezik su *GoogleTest* (gTest), *BoostTest*, *UnitTest++*, *CppTest*, *Catch2* i mnoge druge. U nastavku će biti korišćena *Catch2* biblioteka [5] iz tog razloga što je jednostavna i otvorenog je koda. Napomenimo to da se neće ulaziti u detalje na koji način *Catch2* funkcioniše.

Savetuje se da fajl u kojem se nalaze testovi ne bude fajl u kome se nalazi kod sistema. Testovi koji služe za testiranje različitih stvari se obično nalaze u različitim fajlovima.

Da bi se biblioteka pokrenula potrebno je postaviti direktorijum u kome se ona nalazi, nazvan *catch2*, negde gde mu je moguće pristupiti (na primer u direktorijum sa test datotekama) i napisati blok koda koji je prikazan u kodu broj 1. Kod na liniji broj jedan se sme pojaviti samo u jednom fajlu. [5]

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

Kod 1: Kod neophodan za pokretanje testova

Testiranje se izvršava pomoću makroa koji služi za grupisanje testova, definisanog kao *TEST_CASE(test_name, [tags])*. Grupa testova se može dalje podeliti u sekcije makroom *SECTION*. Primer kojim se započinje testiranje je veoma jednostavan i verovatno nije primenljiv u praksi ali je dobar za ilustraciju. Recimo da imamo neki sistem kome je jedna komponenta definisana kao u kodu broj 2.

```
1 class component {
2 public:
3     component(int arg) : m_arg(arg) {}
4
5     int run() {
6         return unit1(unit2(unit3(m_arg)));
7     }
8 private:
9     int unit1(int x) {
10         return x + 1;
11     }
12     int unit2(int x) {
13         return x * 2;
14     }
15     int unit3(int x) {
16         return (x + 3) * 2;
17     }
18
19     int m_arg;
20 };
```

Kod 2: Jednostavna klasa

Navedena klasa predstavlja neku komponentu koja je sastavljena od jednostavnih jedinica koje se mogu testirati zasebno. Nakon toga želimo da testiramo celu komponentu. To se u ovom slučaju može uraditi testiranjem funkcije *run()* na način prikazan u kodu 3. Makro *REQUIRE(bool)* predstavlja neku tvrdnju za koju želimo da se ispostavi kao tačna.

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch2/catch.hpp"
3 #include "component.h"
4
5 TEST_CASE("Testing component", "[component]") {
6     SECTION("Positive number") {
7         component c(1);
8         REQUIRE(c.run() == 17);
9     }
10    SECTION("Negative number") {
11        component c(-1);
12        REQUIRE(c.run() == 9);
13    }
14    SECTION("Zero") {
15        component c(0);
16        REQUIRE(c.run() == 13);
17    }
18 }
```

Kod 3: Jednostavan test primer

Nakon izvršavanja će se dobiti poruka o uspešnosti testiranja. Ako su svi testovi prošli uspešno onda će se ispisati odgovarajuća poruka, a ako nisu onda se ispisuje broj neuspešnih tvrdnji kao i brojevi linija koda gde se te pogrešne tvrdnje nalaze.

Za eventualnu kreaciju lažnih delova dugih komponenti moguće je koristiti biblioteke poput *GoogleTest Mocking* (gMock), *Isolator++* i druge. Naravno, programeri mogu razvijati i svoje lažne komponente koje će koristiti u testiranju.

Drugi primer će biti nadogradnja primera iz sekcije [Komponentno testiranje](#), gde je komponenta veb strana koja se koristi za logovanje. Na toj veb strani se nalaze dva tekstualna polja koja redom predstavljaju korisničko ime i šifru, kao i jedno dugme kojim se logovanje započinje. Ovde bi testiranje bilo testiranje crne kutije u kome unosimo različite vrednosti u tekstualna polja i klikom na dugme proveravamo da li je logovanje uspešno ili ne. Dakle, testiranje ne zahteva pisanje testova, već samo proveru funkcionalnosti preko korisničkog interfejsa. Treba uneti korisničko ime i šifru u pravilnom i nepravilnom formatu i proveriti da li je logovanje uspelo. Ako se unesu korisničko ime i šifra u pravilnom formatu treba proveriti da li se taj korisnik nalazi u bazi podataka sa korisnicima i shodno tome dozvoliti pristup ili prikazati poruku da korisnik ili ne postoji ili da nije uneta ispravna šifra.

Ovde može doći do problema ako baza podataka u trenutku testiranja komponente ne postoji. Tada je treba lažirati (eng. *Mock*). To se u ovom slučaju može uraditi na jednostavan način bez korišćenja ikakvih biblioteka. Recimo da postoji komponenta koja bi trebalo da komunicira sa bazom, koja ima funkciju koja kao argumente prima dve niske (ime i šifru) i nazad vraća uređeni par vrednosti logičkog (eng. *bool*) tipa. Ako su oba elementa para tačna onda se korisniku dozvoljava logovanje. U suprotnom se korisniku prikazuje odgovarajuća poruka, da uneti korisnik ili ne postoji ili postoji ali je šifra pogrešna.

Lažirana funkcija je nazvana *mock_db* i ima istu povratnu vrednost kao funkcija umesto koje se poziva. Kao što se iz koda 4 može zaključiti, jednostavna je i uzima u obzir postojanje samo jednog korisnika u bazi podataka, kome su korisničko ime i lozinka redom *John123* i *abcde*. Ako su ti podaci zaista uneti u program, onda se logovanje dozvoljava i korisnik se preusmerava na neku drugu veb stranu.

```
1 pair<bool, bool> db_connector::mock_db(  
2     string username, string password) {  
3     if (username == "John123") {  
4         if (password == "abcde") {  
5             return std::pair<bool, bool>{true, true};  
6         } else {  
7             return std::pair<bool, bool>{true, false};  
8         }  
9     } else {  
10        return std::pair<bool, bool>{false, false};  
11    }  
12 }
```

Kod 4: Lažna komunikacija sa bazom

Funkciju koja se u testiranoj komponenti poziva pritiskom na dugme nazovimo *login()*. Iz tela te funkcije se poziva lažna funkcija *mock_db* i njen izlaz se obrađuje. Obrada uključuje jednostavnu proveru vrednosti.

U slučaju pogrešnih vrednosti korisniku se prikazuje poruka uz pomoć klase *message_window*.

Pored toga da li testirana komponenta komunicira sa bazom treba proveriti i da li se u slučaju unosa postojećih informacija za prijavljivanje korisnik prosleđuje na neku drugu, unapred određenu, veb stranu. Prikaz te nove veb strane se takođe može lažirati na neki način, na primer izbacivanjem poruke ili prebacivanjem na neku privremenu veb stranu koja nema nikakvu drugu ulogu. U ovom slučaju se korisnik funkcijom *forward_to* prosleđuje na neku veb stranu nazvanu *Dummy page*. Ceo proces je prikazan u kodu broj 5.

```
1 void loginWindow::login() {
2     /**
3      * telo funkcije koje za ovaj
4      * primer nije bitno prikazati
5      */
6
7     // poziv mock funkcije
8     std::pair<bool, bool> loginInfo
9         = db_conn.mock_db(username, password);
10
11     if (loginInfo.first == true) {
12         if (loginInfo.second == true) {
13             // uspeh, prosledjivanje
14             forward_to("Dummy page");
15         } else {
16             // pogresna lozinka
17             message_window msg;
18             msg.add_text("Neispravna lozinka");
19             msg.show();
20         }
21     } else {
22         // nepoznat korisnik
23         message_window msg;
24         msg.add_text("Korisnik ne postoji");
25         msg.show();
26     }
27 }
```

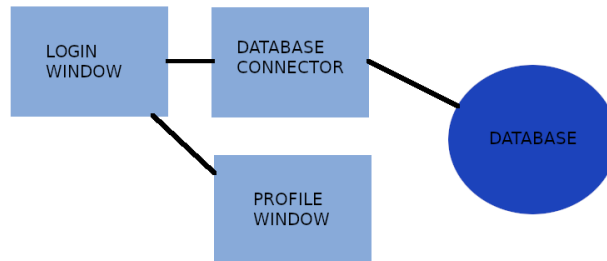
Kod 5: Poziv lažne funkcije

4.2 Primeri integracionog testiranja

Integracionim testiranjem se testira više komponenti istovremeno. Kao i kod komponentnog testiranja, kod integracionog je neke testove moguće izvršiti pomoću biblioteka za jedinično testiranje. U ovoj sekciji će, kao i u prošloj, biti korišćena *Catch2* biblioteka.

Potrebno je prvo definisati sistem na kome ćemo demonstrirati ovu vrstu testiranja. Veoma je sličan primeru iz prošle sekcije i jedan njegov deo čine komponente kao što su baza podataka, komponenta koja komunicira sa bazom podataka, komponenta koja omogućava pristup sistemu kao i komponenta koja predstavlja profil korisnika. Ovaj deo sistema je zamišljen tako da se korisniku prvo prikazuje komponenta za pristup gde

on izvršava identifikaciju. Ako se njegovo korisničko ime i šifra nalaze u bazi podataka onda mu se prikazuje njegov profil na aplikaciji.



Slika 2: Deo sistema koji će biti testiran

Pretpostavimo da je svaka komponenta testirana zasebno. Prve komponente koje želimo da testiramo zajedno su baza podataka i komponenta koja sa njom upravlja (nazvana *dbConn*). Tačnije, želimo da proverimo da li komponenta razmenjuje potrebne podatke sa bazom. Deklaracija komponente *dbConn* se nalazi u kodu 6. U bazi se nalazi tabela koja ima tri kolone, korisničko ime, lozinku, kao i broj kredita koji korisnik ima trenutno na raspolaganju i koji se može koristiti dalje u aplikaciji.

```
1 class dbConn {
2 public:
3     dbConn(std::string path_to_db);
4     ~dbConn();
5
6     bool exists(
7         std::string user, std::string passwd);
8
9     std::optional<int> read_credit(
10        std::string user, std::string passwd);
11
12 private:
13
14     std::ifstream m_in;
15 };
```

Kod 6: DB connection komponenta

Da bi ova komponenta bila testirana sa bazom potrebno je pozvati njene funkcije koje sa bazom komuniciraju i proveriti njihovu povratnu vrednost. Proces je prikazan u kodu broj 7.

```
1 TEST_CASE("Database testing", "[database]") {
2     SECTION("Exists in the database") {
3         dbConn db(path_to_database);
4         REQUIRE(db.exists("username", "password")
5             == true);
6         REQUIRE(db.exists("john123", "password")
```

```

7         == false);
8     }
9
10    SECTION("Get user credit") {
11        dbConn db(path_to_database);
12        REQUIRE(db.read_credit("username", "password")
13                == 10);
14        REQUIRE(db.read_credit("name", "nekasifra")
15                == std::nullopt);
16    }
17 }

```

Kod 7: Testiranje komunikacije sa bazom

Nakon ovog testiranja želimo da integrišemo novu komponentu, a to je komponenta za pristup sistemu (*loginWindow*). Ova komponenta čeka unos korisničkog imena i lozinke od strane korisnika i nakon toga pokušava da pristupi sistemu. Logovanjem se šalju uneti podaci (ako zadovoljavaju kriterijum da su duži od 5 a kraći od 15 karaktera) komponenti *dbConn* koja proverava da li se oni nalaze u bazi. Ukoliko se nalaze korisniku se odobrava pristup, dok se u suportnom zahtev odbija. Ako su uneti podaci tačni korisnik se preusmerava na novu komponentu koja predstavlja njegov profil na aplikaciji. Definicija profil komponente, nazvana *profileWindow*, kao i definicija *loginWindow* komponente se nalaze u kodu broj 8.

```

1  class loginWindow {
2  public:
3      loginWindow(
4          std::string username, std::string password);
5
6      bool login();
7      profileWindow forward_to_profile();
8
9  private:
10     std::string m_username;
11     std::string m_password;
12 };
13
14 class profileWindow {
15 public:
16     profileWindow(
17         std::string username, std::string password);
18
19     std::string password();
20     std::string username();
21
22 private:
23     std::string m_password;
24     std::string m_username;
25 };

```

Kod 8: Komponente za pristup i prikaz profila

Integraciono testiranje sa ovim komponentama je prikazano u kodu broj 9. Prvo što je potrebno proveriti su unešeni korisničko ime i lozinka.

Treba da ispunjavaju uslov da su odgovarajuće dužine kao i da se nalaze u bazi. Prvo više spada u komponentno nego integraciono testiranje, a drugo se može proveriti tako što se proverí povratna vrednost funkcije *login*. Ukoliko ona vraća istinitosnu vrednost logovanje je uspešno dok u suprotnom nije. Ova dva testa se nalaze pod grupom testova nazvanom *Login testing*.

Nakon toga potrebno je proveriti da li se otvorila profil strana korisnika čiji su podaci uneti. To se može proveriti poređenjem informacija koje su unete sa informacijama profil stranice koja je otvorena. Prikaz toga se može videti pod grupom testova nazvanom *From login to profile*.

```

1  TEST_CASE("Login testing", "[login]")
2  {
3      SECTION("Wrong format of username or password") {
4          REQUIRE_THROWS(
5              loginWindow("usernameusername", "p"));
6          REQUIRE_THROWS(
7              loginWindow("user", "pass"));
8          REQUIRE_THROWS(
9              loginWindow("username", "pass"));
10         REQUIRE_THROWS(
11             loginWindow("user", "password"));
12     }
13
14     SECTION("Correct format of username and password")
15     {
16         loginWindow logW1("username", "password");
17         REQUIRE(logW1.login() == true);
18
19         loginWindow logW2("john123", "password");
20         REQUIRE(logW2.login() == false);
21
22         loginWindow logW3("username", "nemasifre");
23         REQUIRE(logW3.login() == false);
24     }
25
26     TEST_CASE("From login to profile", "[profile]") {
27         SECTION("Forward to profile") {
28             loginWindow lw("john123", "abcdef");
29             profileWindow profile("john123", "abcdef");
30
31             REQUIRE(lw.forward_to_profile().username()
32                     == profile.username());
33             REQUIRE(lw.forward_to_profile().password()
34                     == profile.password());
35         }
36     }

```

Kod 9: Testiranje integrisanja komponenti za profil i pristup

Makroi *REQUIRE_THROWS(arg)* se koriste kada tvrdimo da se pozivom argumenta *arg* izbacuje nekakav izuzetak. Ukupno je napisano trinaest tvrdnji u ovoj sekciji, grupisanih u tri dela u zavisnosti od toga šta testiraju. Kada se pokrenu da bi se proverila trenutna implementacija ovog

dela sistema dobija se izlaz da su svi testovi prošli uspešno. Dobijena poruka izgleda ovako:

```
1 =====  
2 All tests passed (13 assertions in 3 test cases)
```

Kod 10: Rezultat testiranja

5 Zaključak

Postoji nekoliko različitih vrsta testiranja od kojih je svaka bitna i ima drugačiju namenu. Prvo se testiraju jedinice softvera koje se spajaju u neke veće komponente koje se onda zasebno testiraju. Onda se te komponente integrišu u sistem koji se takav testira. Na kraju se primenjuju testovi prihvatljivosti.

U ovom radu su obrađene dve vrste testiranja detaljnije od drugih, a to su komponentno i integraciono. Komponentno je veoma bitno za funkcionalnost softvera i povećava robusnost sistema. Integraciono testiranje proverava način na koji komponente rade zajedno što je veoma bitna stavka za softver.

Na samom kraju su prikazani neki primeri testiranja u C++ kao i alati koje je za to testiranje moguće koristiti. Primeri komponentnog testiranja su uključili i pisanje jednostavnih lažnih delova drugih komponenti potrebnih da bi se testiranje izvršilo. U integracionom testiranju su testirane zajedno komponente koje čine deo nekog većeg sistema. Ispitana je komunikacija između njih kao i da li pravilno funkcionišu zajedno. Za obe vrste testiranja je korišćena *Catch2* biblioteka.

Literatura

- [1] Milind G. Limaye. *Software Testing*. Tata McGraw-Hill Education, 2009. on-line at: https://books.google.rs/books?id=zUm8My7SiakC&printsec=copyright&redir_esc=y#v=onepage&q&f=false.
- [2] Guru99.com. What is component testing? techniques, example test cases. on-line at: <https://www.guru99.com/component-testing.html>.
- [3] Professionalqa.com. What is meant by stubs and drivers? 2019. on-line at: <http://www.professionalqa.com/stubs-and-drivers>.
- [4] Softwaretestinghelp.com. What is integration testing (tutorial with integration testing example). 2019. on-line at: <https://www.softwaretestinghelp.com/what-is-integration-testing/>.
- [5] Catch Org. Catch2 library. on-line at: <https://github.com/catchorg/Catch2>.