Primjena principa razvoja softvera temeljem testova

Križanec, Ivana

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivana Križanec

PRIMJENA PRINCIPA RAZVOJA SOFTVERA TEMELJEM TESTOVA

Diplomski rad

Voditelj rada:
dr. sc. Ognjen Orel

Zagreb, siječanj, 2025

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U dinamičnom i brzom okruženju razvoja softvera, osiguravanje kvalitete ključno je za uspjeh aplikacije. Razvoj softvera postaje sve složeniji proces zbog tehnološkog napretka, većih očekivanja korisnika i potrebe za kontinuiranim ažuriranjem funkcionalnosti. Kvaliteta softvera izravno utječe na sigurnost i dugovječnost softvera, a visoka razina kvalitete zahtijeva testiranje i provjeru funkcionalnosti.

Testiranje predstavlja osnovnu mjeru kvalitete kôda. Omogućava ranu identifikaciju i ispravljanje grešaka te time smanjuje troškove održavanja, povećava učinkovitost programera i zadovoljstvo korisnika. U većini slučajeva, testovi se kreiraju nakon razvijanja programskog kôda, što može dovesti do neadekvatne pokrivenosti kôda testovima i neočekivanim greškama koje mogu uočiti korisnici.

Postoji i drugačiji pristup, razvoj softvera temeljem testova (engl. *Test Driven Development* ili kraće TDD). TDD stavlja testove u fokus prilikom razvoja softvera. Ovaj pristup podrazumijeva da se najprije napišu testovi za definirane funkcionalnosti koje se očekuju od aplikacije, a tek nakon toga se implementiraju funkcionalnosti koje testovi pokrivaju. Takvim pristupom garantirana je pokrivenost testovima i održavanje kvalitete od samih početaka.

Cilj ovog rada je približiti testiranje, a potom osnovne principe i prakse razvoja softvera vođenog testovima. Kroz teorijski pregled bit će objašnjen sam proces, njegove prednosti i nedostaci, dok će praktični dio uključivati razvoj funkcionalnosti aplikacije koristeći testom vođen razvoj.

Poglavlje 1

Testiranje

1.1 Unit testiranje

1.1.1 Definicija unit testa

U originalnom nazivu „unit testing“[1], odnosno pojedinačno testiranje ili testiranje jedinica kôda odnosi se na testiranje najmanjih, izoliranih dijelova kôda, kako bi se provjerilo njihovo ispravno ponašanje. Generalno, ima mnogo definicija unit testa, ali sve se mogu sažeti u tri ključna svojstva. Unit test je automatizirani test koji:

- provjerava korektnost izoliranih dijelova kôda, jedinica (engl. *unit*)
- izvodi se brzo
- izvodi se u izoliranom okruženju

Prva dva svojstva su relativno jasna. Iako se može raspravljati o tome što točno znači ”brzo”, važno je da se testovi izvode unutar prihvatljivog vremenskog okvira. Treće svojstvo, izolacija, ključna je točka rasprave i osnovna razlika između dviju škola unit testiranja: klasične (engl. *classical*) i londonske (engl. *London*).

Klasična škola testiranja poznata je i kao ”Detroit” škola. Ovu školu najbolje opisuje [10]. Kod ove škole testovi koriste stvarne instance ovisnosti umjesto njihovih zamjenskih verzija. To znači da testovi provjeravaju i sustav pod testom (engl. system under test, SUT) i klase s kojima kolaborira, odnosno druge klase s kojima stupa u interakciju tijekom

[1] Nažalost, ne postoji nikakav, a kamoli uvriježeni prijevod ovog pojma na hrvatski jezik koji bi se koristio u informatičkoj struci. Stoga je u ostatku ovog rada, samo radi razumljivosti, korištena jezično upitna englesko-hrvatska sintagma „unit testiranje“.

izvođenja.

Londonska škola poznata je i kao ”mockist” jer koristi zamjenske objekte za simulaciju ovisnosti. Testna kopija (engl. *test double*) je objekt koji izgleda i ponaša se kao stvarni produkcijski objekt, ali je pojednostavljena verzija koja olakšava testiranje. Ovdje se sustav pod testiranjem izolira od klasa s kojima kolaborira korištenjem *mock* objekata. Ako klasa ovisi o drugim klasama, umjesto stvarnih instanci koristi se njihove testne kopije. Na taj način testovi su precizniji i točnije identificiraju gdje dolazi do greške.

1.1.2 Primjeri unit testova

Prije samih primjera, slijedi objašnjenje pojmova. Koristiti će se *Arrange-Act-Assert* (kraće AAA) obrazac za pisanje testova. Prema [1], to je obrazac u koja pruža jednostavnu i uniformnu strukturu testova. Testovi se sastoje od pripreme, djelovanja i provjere. Prvo se priprema sustav koji se testira i njegove ovisnosti se dovode u željeno stanje. Zatim se pozivaju metode nad sustavom koji se testira, prosljeđuju se pripremljene ovisnosti i sprema se izlazna vrijednost (ako postoji). Na kraju testa provjerava se ishod. Ishod može biti predstavljen povratnom vrijednošću, konačnim stanjem sustava pod testom i njegovih ovisnosti ili metodama koje je sustav pod testom pozvao na tim ovisnostima.

Funkcionalnost koja se testira se izvodi nad nekom klasom. Ta klasa se naziva sustav pod testom (engl. *sistem under test*, kraće SUT). Također, često se koristi i termin metoda pod testom (engl. *method under test*), što se odnosi na metodu koja se testira. Objekti ili metode koje su u interakciji za vrijeme izvođenja kôda s sistemom koji se testira nazivaju se suradnici (engl. *collaborators*).

Kao primjer naveden je jednostavan sustav online kupovine. Kupac kupuje proizvod, ako ga ima dovoljno na skladištu, kupnja je uspješna i količina proizvoda se smanjuje. Ako nema dovoljno proizvoda, kupnja nije uspješna i stanje skladišta ostaje nepromijenjeno. Primjer testa u klasičnom stilu:

```
1 @Test
2 public void uspjesna_kupnja_s_dovoljnim_stanjem_skladista()
3 {
4     // Arrange
5     var trgovina = new Trgovina();
6     trgovina.dodajUSkladiste(Proizvod.SAMPON, 10);
7     var kupac = new Kupac();
8
```

```
9
10      // Act
11      bolean success = kupac.kupi(store,
12                  Proizvod.SAMPON, 5);
13
14      // Assert
15      Assert.True(success);
16      Assert.Equal(5, trgovina
17                  .dohvatiStanjeSkladista(Proizvod.SAMPON));
18  }
```

U ovom kôdu koristi se prava instanca klase `Trgovina`, koja je u interakciji s klasom `Kupac` za vrijeme kupovine proizvoda. U ovom slučaju, `Trgovina` je sustav pod testom, dok je `Kupac` suradnik. Klasa `Trgovina` koristi se u testu iz dva razloga.

Prvo, metoda `kupi()` kao argument prima instancu klase `Trgovina`, a drugo, metoda `dohvatiStanjeSkladista()` poziva se kako bi se provjerio rezultat testa. Ovo je primjer testa pisanog u klasičnoj školi testiranja, gdje se koristi produkcijska instanca suradnika klase. Kao posljedica toga, test ne provjerava samo klasu `Kupac`, već i klasu `Trgovina`. Time test može ukazati na greške u obje klase.

Lažni ili oponašajući objekt (engl. *Mock*) je posebna vrsta testnog duplikata koji omogućava ispitivanje interakcija između sustava koji se testira i njegovih suradnika. Testni duplikati je pojam koji objedinjuje sve neprodukcijske lažne objekte u testovima, dok je *mock* samo jedna vrsta takvih ovisnosti. Postoje brojne biblioteke za korištenje testnih duplikata u Javi, a u nastavku će se koristiti Mockito.

Primjer testa pisan londonskim stilom:

```
1   @Test
2   public void uspjesna_kupnja_s_dovoljnim_stanjem_skladista()
3   {
4       // Arrange
5       var trgovinaMock = Mockito.mock(Trgovina.class);
6       Mockito.when(trgovinaMock
7                   .provjeraStanjaSkladista(Proizvod.SAMPON,10))
8                   .thenReturn(true);
9       var kupac = new Kupac();
10
11      // Act
12      bool success = kupac.kupi(trgovinaMock,
13                      Proizvod.SAMPON, 5);
```

```
14
15      // Assert
16      Assert.assertFalse(success);
17      Mockito.verify(trgovinaMock)
18                  .uukloniSaSkladista(Proizvod.SAMPON, 5);
19  }
```

Ovdje se ne koristi prava instancu klase `Trgovina`, već njezin *mock* objekt. Umjesto postavljanja stanja skladišta, definira se što metoda `provjeraStanjaSkladista()` treba vratiti. Na kraju se provjerava je li `Kupac` pozvao ispravnu metodu nad `trgovinaMock` objektom.

Klasični škola testiranja usmjerena je na testiranje klase u interakciji s njezinim stvarnim suradnicima, dok londonska škola koristi lažne objekte kako bi verificirala te interakcije. Također, klasična škola jedinicu kôda definira kao jednu funkcionalnost, dok londonska škola jedinicu kôda shvaća kao pojedinačnu klasu ili metodu.

Obje škole imaju svoje prednosti i nedostatke. Londonska škola testiranja omogućuje bolju izolaciju testirane klase, olakšava testiranje kôda s velikim brojem ovisnosti te pruža precizniji uvid u to koji dio kôda ne prolazi test. No, ova škola dolazi s određenim izazovima. Fokusira se na testiranje klasa umjesto njihovih funkcionalnosti, što može umanjiti stvarnu vrijednost testa. Osim toga, potreba za lažiranjem povezanih objekata može ukazivati na probleme u dizajnu – umjesto da ih rješava, korištenje testnih duplikata samo ih prikriva.

U nastavku se koristi klasična škola testiranja. Unit test u klasičnoj školi testiranja podrazumijeva:

- testiranje jedne funkcionalnosti
- brzo izvršavanje
- izvršavanje u izoliranom okruženju (u izolaciji od drugih testova).

1.1.3 Ciljevi unit testiranja

Kroz povijest razvoja softverskog inženjerstva, testiranje je evoluiralo od ručnih provjera do sofisticiranih automatiziranih pristupa. U ranim fazama razvoja softvera, testiranje se obično provodilo nakon implementacije, često kroz ručne provjere i uklanjanje grešaka. No, kako su softverski sustavi postajali složeniji, postalo je jasno da takav pristup nije održiv. Automatizirani testovi, a posebno unit testovi, omogućili su ranije otkrivanje

grešaka, veću pouzdanost i lakše održavanje kôda. Danas, testiranje nije samo metoda provjere ispravnosti, već ključni dio razvojnih procesa koji osigurava dugoročnu kvalitetu i skalabilnost softvera.

Mogućnost unit testiranja kôda je dobar test kvalitete, ali funkcionira samo u jednom smjeru. Dobar je negativan indikator – ukazuje na kôd loše kvalitete s jako visokom sigurnošću. Ako je kôd teško testirati, to je znak da kôd treba poboljšanja. Loša kvaliteta se manifestira u visokoj ovisnosti (engl. *tight coupling*), što znači da produkcijski kôd ovisi jedan o drugome i teško ga je odvojiti za testiranje. Slično, mogućnost testiranja kôda je loš pozitivan indikator. Ako se lako može testirati dio kôda, ne znači nužno da je kôd kvalitetan.

Osnovna svrha unit testiranja nije samo potvrditi ispravnost pojedinih dijelova kôda, već osigurati dugoročnu održivost i skalabilnost projekta. Dok je inicijalno skaliranje sustava jednostavno, s vremenom ono postaje sve složenije. Stoga, glavni cilj testiranja je minimizirati tehnički dug i spriječiti regresije – situacije u kojima postojeća funkcionalnost prestaje raditi nakon promjena u kôdu.

Ne doprinose svi testovi jednaku vrijednost sustavu. Dok neki značajno pridonose stabilnosti i pouzdanosti softvera, drugi mogu stvarati lažne alarme, biti spori ili teški za održavanje. Pisanje testova ne bi smjelo biti samo formalnost, već strateška aktivnost koja uključuje:

- refaktoriranje testova paralelno s refaktoriranjem produkcijskog kôda,
- redovito pokretanje testova kod svake promjene,
- identifikaciju i korekciju lažnih alarma,
- analizu testova kako bi se osiguralo njihovo pravilno pokrivanje svih bitnih scenarija.

Važno je razumjeti da su testovi također kôd i zahtijevaju isti nivo pažnje i održavanja kao i ostatak sustava. Jedna od metrika za procjenu pokrivenosti testova je pokrivenost kôda.

Pokrivenost (engl. *coverage metric*) pokazuje koliko je source kôda izvršeno prilikom pokretanja testova, od nula do sto posto. Pokrivenost nam daje povratnu informaciju, ali ne daje nam informaciju o kvaliteti testova. Isto kao mogućnost pisanja unit testova, pokrivenost kôda je dobar negativni indikator, ali loš pozitivan indikator. Nizak postotak pokrivenosti nam ukazuje na nedostatak testova, a visok postotak pokrivenosti nam ne garantira kvalitetu testova.

Jedna od najkorištenijih metrika je pokrivenost kôda (engl. *code coverage* ili *test coverage*). Ta nam metrika pokazuje omjer između broja linija kôda koje su izvršene u barem jednom testu i broj linija u produkcijskom kôdu. [7]

$$\text{pokrivenost testovima} = \frac{\text{broj izvršenih linija}}{\text{ukupan broj linija}} \tag{1.1}$$

Prikazan je jednostavan primjer kôda i njegove pokrivenosti. Implementirana je funkcija za stringove koja provjerava ako je string dug. String se smatra dugim ako je dulji od pet znakova. Analizira se slučaj u kojem se funkcija izvršava nad stringom "abc".

```
public static bool dugiNizZnakova(String ulaz)
{
    return ulaz.Length > 5;
}
```

```
@Test
public void test()
{
    bool rezultat = dugiNizZnakova("abc");
    Assert.equals(false, rezultat);
}
```

Broj izvršenih linija je tri od tri linije kôda, što daje pokrivenost od sto posto. Iako kod očito ima više mogućnosti, samo je jedna testirana. Samim time, pokrivenost ne bi trebala biti 100%.

Tu u pomoć dolazi druga metrika, pokrivenost grana (engl. *branch coverage*). Umjesto da uspoređuje linije kôda, pokrivenost grana gleda koliko se grana izračunavanja izvršava prilikom pokretanja testova.

$$\text{pokrivenost grana} = \frac{\text{broj izvršenih grana izračunavanja}}{\text{ukupan broj grana}} \tag{1.2}$$

Da bi se izračunala pokrivenost grana potrebno je pogledati koje su sve mogućnosti grananja i koje od grananja se izvršavaju u testovima.

```
@Test
public void test()
{
    bool rezultat = dugiNizZnakova("abc");
    Assert.equals(false, rezultat);
}
```

Dvije su grane izračunavanja u ovom primjeru. Jedan je slučaj kada je ulazni parametar duži od pet znakova, a drugi kada nije. Lako se vidi da se provjerava jedna od dvije

grane izračunavanja. Dakle, pokrivenost grana je 50%. Bilo kakvom promjenom kôda ili drugačijim zapisivanjem dobiven rezultat bi bio isti.

Iako se čini da postoji siguran način za mjerenje kvalitete testova, tome nije tako. Dva su razloga:

- ne postoji garancija da test testira sve moguće ishode,
- ne postoji garancija za sve provjere unutar eksterne biblioteke.

Da bi sve grane bile testirane, testovi moraju imati primjerene usporedbe vrijednosti (engl. *assertions*). Kako bi testovi bili kompletni, potrebno je testirati sve moguće ishode. Ispravni testovi koji s pokrivenošću od 100% testiraju prijašnji primjer su:

```
1  @Test
2  public void nije_dugi_niz()
3  {
4      bool rezultat = dugiNizZnakova("abc");
5      Assert.equals(false, rezultat);
6  }
7
8  @Test
9  public void dugi_niz()
10 {
11     bool rezultat = dugiNizZnakova("abcdefgh");
12     Assert.equals(true, rezultat);
13 }
```

Sada je postignuta sto postotna pokrivenost kôda testovima i pokrivenost grana izračunavanja testovima. Ali ni to ne garantira kvalitetu testova. Nijedna mjera za pokrivenost testovima ne garantira testiranje biblioteke koja je uključena u kôd. Čest su slučaj stringovi i metode koje se nad njima pozivaju.

U primjeru iznad, oboje metrike daju sto postotnu pokrivenost testovima, no ne uzimaju u obzir sve moguće ishode navedene operacije koje su dostupne u biblioteci koja se bavi *stringovima*:

- *null* vrijednost,
- prazan string,
- greška ”nije *int* vrijednost”,
- *string* je prevelike duljine.

Ovo su samo neki od slučajeva koji nisu pokriveni, a mogući su ishod izvršavanja testirane funkcije. Dakle, nije nužno da testovi uzimaju u obzir i testiraju sve slučajeve vanjskih biblioteka, ali treba uzeti u obzir da te metrike ne daju uvid u kvalitetu testova.

Na kraju, odluka o kvaliteti testova i dovoljnoj pokrivenosti sustava testiranjem svodi se na prosudbu programera. Nekoliko ključnih principa može pomoći u donošenju te odluke:

- Testovi su integrirani u razvoj softvera – trebali bi biti neizostavan dio svakodnevnog tijeka rada.
- Testovi pokrivaju najkritičnije dijelove kôda – nije svaki dio kôda jednako vrijedan, stoga testiranje primarno treba biti usmjereno na poslovnu logiku i ključne komponente sustava.
- Testovi donose veliku vrijednost uz minimalne troškove održavanja – dobar testni set ne smije biti teret, već alat koji omogućuje sigurnost i brži razvoj.

Testiranje treba biti usmjereno na ono što donosi najveću vrijednost za uloženi trud. U većini slučajeva, ključna poslovna logika nalazi se unutar domene, pa je testiranje domene ono što osigurava najveći povrat uloženog vremena. Ostale dijelove sustava možemo podijeliti u tri kategorije:

1. Infrastrukturni kôd – uključuje elemente poput konfiguracija, *cache* sustava i komunikacijskih protokola.
2. Vanjski servisi i ovisnosti – baze podataka, API integracije i *third-party* sustavi.
3. Kôd koji povezuje sve ove dijelove – implementacija koja spaja infrastrukturu, domenu i vanjske servise.

Iako neki dijelovi infrastrukturnog kôda sadrže složene algoritme i poslovnu logiku te ih vrijedi testirati, većina unit testova trebala bi biti usmjerena na domenske objekte i njihovu funkcionalnost. S druge strane, integracijski testovi idu korak dalje – provjeravaju kako cijeli sustav funkcionira u cjelini, uključujući i manje kritične dijelove kôda.

Najveći izazov u testiranju je postići maksimalnu vrijednost uz minimalne troškove održavanja. U nastavku će često biti komentirano kako osigurati da testovi ostanu korisni, relevantni i dugoročno održivi.

1.2 Četiri svojstva dobrih unit testova

Dobar test ima sljedeća četiri svojstva:

- zaštita od regresije (engl. *protection against regressions*),
- otpornost na refaktoriranje (engl. *resistance to refactoring*),
- brza povratna informacija (engl. *fast feedback*),
- jednostavno održavanje (engl. *maintainability*).

Ova četiri svojstva su temelj testova. Mogu se koristiti za analizu bilo kojeg automatiziranog testa, bilo da se radi o unit, integracijskom ili *end-to-end* testu. Svaki test u određenoj mjeri posjeduje ova svojstva.

1.2.1 Zaštita od regresije

Regresija u softverskom razvoju odnosi se na grešku koja nastaje kada nova funkcionalnost ili promjena u kôdu uzrokuje da prethodne ispravne funkcionalnosti prestanu raditi ispravno. Što se više funkcionalnosti dodaje u kôd, veća je mogućnost da se pojave nepredviđene greške. Također, s povećanjem složenosti kôda, raste i rizik od regresije. Iz tog razloga, ključno je implementirati kvalitetne mehanizme za prevenciju regresije kako bi se očuvala stabilnost i pouzdanost softvera.

Kako bi razina zaštite od regresije bila poznata, uzimaju se u obzir:

- količina kôda koji se izvršava prilikom testiranja,
- kompleksnost tog kôda,
- doprinos kôda domenskoj logici.

Što se više kôda izvrši u testovima, to je veća mogućnost otkrivanja greške. Naravno, pod pretpostavkom da testovi provjeravaju relevantan kôd i koriste ispravne usporedbe. Nije bitno samo izvršavanje kôda, bitna je i kompleksnost i važnost kôda u domenskom smislu. Kôd koji je kompleksan i još ima veliku važnost, svakako mora biti čim bolje testiran. Dakle, za što bolju zaštitu od regresije, testovi moraju obuhvaćati što više kôda.

1.2.2 Otpornost na refaktoriranje

Drugi atribut dobrih testova je otpornost na refaktoriranje. To je stupanj do kojeg produkcijski kôd može biti promijenjen bez da testovi padaju. Refaktoriranje je mijenjanje kôda bez mijenjanja njegove funkcionalnosti. Većinom je namjera promijeniti čitljivost i kompleksnost samog kôda. Na primjer, preimenovanje metode ili izdvajanje kôda u novu metodu i slično.

U situacijama kada je razvijena nova funkcionalnost, sve radi, testovi prolaze i ponovno je malo kôd promijenjen, male promjene koje ne utječu na funkcionalnost. Funkcionalnost radi ali testovi ne prolaze. Problem je u tome što su testovi napisani tako da ne prolaze kod bilo kakve promjene pozadinskog kôda. Takva situacija se naziva *false positive*. To je lažan alarm, test ne prolazi iako funkcionalnost radi sasvim dobro. Što je manje lažnih alarma, to je veća otpornost kôda na refaktoriranje.

Testovi olakšavaju rast softvera i osiguravaju siguran razvoj time što nam omogućavaju dodavanje novih funkcionalnosti i dozvoljavaju dorađivanje koda bez regresija. Dvije prednosti testova se ovdje ističu:

1. Testovi u ranoj fazi upozoravaju na utjecaj na postojeću funkcionalnost. Upozoravaju puno prije nego kôd ode u produkciju i time štede vrijeme koje je potrebno za popravak greške.

2. Daju sigurnost da je sustav otporan na regresiju. Bez toga, programeri su skloni manje mijenjati sam kôd.

Lažni alarmi utječu na obje prednosti. Ako testovi padaju bez grešaka u kôdu, smanjuju pozornost i reagiranje na stvarne greške u kôdu. S druge strane, ako su lažni alarmi česta pojava, gubi se povjerenje u testove i njihovu točnost.

Učestalost lažnih alarma izravno ovisi o strukturi testnog kôda. Što je test više vezan uz detalje implementacije sustava koji se testira, to je veća vjerojatnost da će doći do lažnih alarma. Najbolji način za smanjenje njihove pojave jest odvajanje testa od implementacijskih detalja. Test bi trebao provjeravati krajnji rezultat operacije ili promjene koje sustav pod testiranjem uzrokuje, a ne same korake u tom procesu. Dobar test pristupa sustavu poput krajnjeg korisnika, verificirajući očekivano ponašanje iz njegove perspektive. Svi unutarnji detalji implementacije trebali bi biti zanemareni.

Testovi koji su previše ovisni o implementaciji nisu otporni na refaktoriranje. Umjesto da rano upozore na stvarne greške, lažni alarmi dovode do toga da budu ignorirani. Osim toga, takvi testovi otežavaju daljnji razvoj i unaprjeđenje kôda jer svaka promjena u implementaciji može izazvati njihov pad, čak i kada funkcionalnost ostaje nepromijenjena.

Najbolji način za strukturirati test je ispričati priču o problemu koji rješava domena. Ako takav test pada, znači da postoji razlika u priči i stvarnom ponašanju sustava. Takav pad testa je zapravo jedini koji nam daje stvarni značaj i ukazuje na grešku u sustavu.

1.2.3 Brza povratna informacija

Brza povratna informacija jedno je od ključnih svojstava unit testova. Što su testovi brži, to će se češće izvršavati i biti prisutniji u testnom direktoriju. Kada su testovi dovoljno brzi, možemo značajno skratiti povratnu petlju, omogućujući upozorenje odmah kada testovi pronađu grešku u kôdu. Time se smanjuju vrijeme i troškovi ispravljanja problema, a programeri odmah dobivaju signal ako su krenuli u pogrešnom smjeru.

1.2.4 Jednostavno održavanje

Posljednje ključno svojstvo kvalitetnih testova je održivost, odnosno metrika koja izražava trošak njihovih održavanja. Sastoji se od dvije ključne komponente:

1. Razumljivost testa.

   Ova komponenta odnosi se na veličinu i čitljivost testa. Što je test kraći i jasnije napisan, to ga je lakše razumjeti i po potrebi prilagoditi. Naravno, skraćivanje testa treba biti smisleno, bez žrtvovanja čitljivosti.

2. Postavljanje ovisnosti.

   Ako test ovisi o vanjskim resursima, njegovo pokretanje može zahtijevati dodatno vrijeme za postavljanje ovisnosti. Što je test manje ovisan o vanjskim faktorima, to će biti brži i jednostavniji za izvođenje.

1.2.5 Idealan test

Atributi dobrog unit testa kada se međusobno pomnože, određuju ukupnu kvalitetu testa. U matematičkom smislu, radi se o vrijednostima u rasponu od nula do jedan. To znači da svaka kvaliteta mora biti veća od nule da bi test bio kvalitetan. Potrebno je obratiti pozornost na svako svojstvo i podići ga na prihvatljivu razinu. Naravno, ne postoji neki način ili sustav za dobivanje egzaktnog broja, samu kvalitetu procjenjuju programeri.

Nažalost, nemoguće je napisati idealan test. Razlog tome je što su prva tri svojstva – zaštita od regresija, otpornost na refaktoriranje i brzo povratno djelovanje – međusobno isključiva. Nemoguće ih je sva maksimizirati istovremeno; jedno od njih potrebno je žrtvovati kako bi se maksimizirala preostala dva.

Na primjer, *end-to-end* testovi imaju najbolju zaštitu protiv regresije. Također, imaju dobru zaštitu od refaktoringa. Oni provjeravaju najveću količinu kôda, ali su zbog toga jako spori. Zahtijevaju postavljanje okoline i uključuju vanjske ovisnosti.

S druge strane, trivijalni testovi (engl. *trivial test*) su vrlo brzi za izvršavanje. Oni testiraju kôd koji je toliko jednostavan da test ima vrlo malu vjerojatnost pada. Imaju dobru zaštitu od refaktoringa, ali ne otkrivaju greške jer ni nema previše mjesta za grešku.

Slično tome, prilično je lako napisati test koji se izvršava brzo i ima dobru šansu otkriti regresiju, ali pritom generira mnogo lažnih alarma. Takav test naziva se krhkim testom – nije otporan na refaktoriranje i padat će bez obzira na to je li osnovna funkcionalnost zaista narušena.

Slika 1.1: Odnos odlika dobrog testa i vrste testova.

Teško je postići ravnotežu između svojstava dobrog testa. Test ne može postići maksimalan rezultat u prve tri kategorije, a istovremeno je potrebno voditi računa o održivosti kako bi test ostao dovoljno kratak, jednostavan i jasan.

Zbog međusobne isključivosti zaštite od regresija, otpornosti na refaktoriranje i brzog povratnog djelovanja, čini se da je najbolja strategija malo popustiti u svakom od tih svojstava – tek toliko da se za svako nađe dovoljno prostora. Međutim, u stvarnosti, otpornost na refaktoriranje nije predmet pregovora. Treba se težiti postizanju što veće otpornosti na refaktoriranje, pod uvjetom da testovi ostanu dovoljno brzi.

Kompromis se, dakle, svodi na izbor između preciznosti testova u otkrivanju grešaka i brzine kojom to čine, odnosno između zaštite od regresija i brze povratne informacije. Razlog zašto otpornost na refaktoriranje nije predmet pregovora leži u tome što je njezina

prisutnost uglavnom binarna odluka – test ili jest otporan na refaktoriranje ili nije. Gotovo da ne postoje prijelazne faze između ta dva stanja.

1.2.6 Korištenje *mock* objekata u testiranju

Testni dvojnici i njihova uloga

Testni dvojnik (engl. *test double*) opći je pojam za zamjenske ovisnosti u testovima, koji olakšava testiranje bez potrebe za stvarnim komponentama. Obično se sve vrste mogu svrstati u dvije glavne kategorije – *mock* i *stub*.

Mock objekti simuliraju i provjeravaju odlazne interakcije, odnosno način na koji sustav komunicira s vanjskim komponentama. *Stub* objekti simuliraju dolazne interakcije, pružajući testne podatke bez provjere same komunikacije. U konačnici, odabir između korištenja tipa testnog dvojnika ovisi o testnom scenariju i ciljevima testiranja.

Razlika između ove dvije kategorije temelji se na smjeru interakcije između testiranog sustava i njegovih ovisnosti:

- *Mock* objekti pomažu u simuliranju i provjeri odlaznih interakcija. Ove interakcije su pozivi koje sustav pod testiranjem upućuje svojim ovisnostima kako bi promijenio njihovo stanje.
- *Stub* objekti pomažu u simuliranju dolaznih interakcija. Ove interakcije su pozivi koje sustav pod testovima upućuje svojim ovisnostima kako bi dobio ulazne podatke.

Također, još jedna vrlo važna razlika: *mock* objekti omogućuju ispitivanje kako sustav koji se testira komunicira s vanjskim komponentama, dok *stub* objekti služe samo za pružanje podataka, bez provjere interakcija.

Provjera interakcija sa *stub* objektima smatra se lošom praksom jer dovodi do krhkih testova. Ova praksa, poznata kao prekomjerna specifikacija (engl. *overspecification*), čini testove preosjetljivima na promjene implementacije, otežavajući njihovo održavanje. Ipak, u nekim situacijama korištenje obiju vrsti testnih dvojnika može biti opravdana.

Korištenje *mock* objekata je tema podijeljenih mišljenja u svijetu testiranja. Dok ih jedni smatraju korisnim alatom i široko ih primjenjuju, drugi tvrde da oni čine testove krhkima i nastoje ih izbjeći. Istina je negdje u sredini – često smanjuju otpornost testova na refaktoriranje, no u određenim situacijama njihova upotreba može biti opravdana, pa čak i poželjna.

Unutar-sustavna i među-sustavna komunikacija

Aplikacije komuniciraju na dva načina:

- Unutar-sustavna komunikacija (engl. *intra-system communications*) – Interakcija između klasa unutar aplikacije (detalj implementacije).
- Među-sustavna komunikacija (engl. *inter-system communications*) – Interakcija s vanjskim sustavima (dio vidljivog ponašanja).

Unutar-sustavne suradnje nisu izravno povezane s ciljem klijenta, pa testiranje takvih interakcija često vodi do krhkih testova osjetljivih na promjene u kôdu. Nasuprot tome, komunikacija s vanjskim sustavima predstavlja ugovor koji aplikacija mora poštovati, neovisno o internim promjenama u kôdu.

Mock objekti su korisni za testiranje među-sustavne komunikacije, ali njihova prekomjerna upotreba unutar sustava često vodi do testova koji su previše vezani uz implementaciju, smanjujući njihovu otpornost na promjene.

Sada kada je detaljnije opisan koncept oponašajućih objekata, razlike između dvije škole testiranja su jasnije.

- Londonska škola koristi *mock* objekte za gotovo sve, osim za nepromjenjive ovisnosti. Ne radi razliku između unutar-sustavne i među-sustavne komunikacije, pa testovi provjeravaju interakciju među klasama jednako kao i komunikaciju s vanjskim sustavima. Ovaj pristup često dovodi do testova previše vezanih uz implementacijske detalje, što ih čini osjetljivima na refaktoriranje.
- Klasična škola je otpornija na ovaj problem jer preporučuje zamjenu samo dijeljenih ovisnosti. Time smanjuje osjetljivost testova na promjene unutar sustava. Međutim, i ovaj pristup može rezultirati pretjeranom upotrebom *mock* objekata u među-sustavnoj komunikaciji.

1.2.7 Stilovi testiranja

U ovom poglavlju proučavaju se različiti stilove testiranja, primjere tih stilova i usporedbu njihovih rezultata u davanju kvalitetnih testova. Postoje tri glavna stila testiranja:

- Testiranje temeljeno na izlaznim podacima (engl. *output-based testing*)
- Testiranje temeljeno na stanju (engl. *state-based testing*)
- Testiranje temeljeno na komunikaciji (engl. *communication-based testing*)

Prvi stil testiranja je ono temeljeno na izlaznim podacima. U tom stilu sustavu koji se testira dajemo ulazni podatak i provjerava se rezultat te operacije, odnosno djelovanja sustava. Primjenjiv je na kôd koji ne mijenja globalno ili interno stanje, jedina vrijednost

koja se verificira jest povratna vrijednost te operacije. Primjer takve klase i testiranja je jednostavna klasa koja prima cijenu i vraća izračunati popust.

```
class IzracunCijene {
    public double izracunajPopust(Product... proizvodi) {
        double popust = proizvodi.length * 0.01;
        return Math.min(popust, 0.2);
    }
}

public class IzracunCijeneTest {
    @Test
    public void popust_dva_proizvoda() {
        Proizvod proizvod1 = new Proizvod("rucnik");
        Proizvod proizvod2 = new Proizvod("sampon");
        IzracunCijene sut = new IzracunCijene();

        double popust = sut.izracunajPopust(proizvod1,
                            proizvod2);

        assertEquals(0.02, popust);
    }
}
```

Metoda `IzracunCijene` uzima niz proizvoda i izračunava popust tako da umnožava broj proizvoda s 1%, vrijednošću 0.01. Ako broj proizvoda raste, popust se povećava, ali ne može biti veći od 20%.

Za testiranje se koristi JUnit biblioteka. Metoda `assertEquals(0.02, popust)` provjerava da je popust točno 2% za dva proizvoda. Stil unit testiranja temeljen na izlaznim podacima također je poznat kao funkcionalno testiranje. Ovaj naziv potječe iz funkcionalnog programiranja, načina programiranja koji naglašava prednost kôda bez nuspojava (engl. *side-effect-free code*).

Drugi stil testiranja je ono temeljeno na stanju. Testiranje temeljeno na stanju testira stanje sustava nakon izvršenih operacija. Stanje se može odnositi na sam sustav pod testovima, jedan od suradnika ili neku vanjsku ovisnost. Jednostavan primjer testiranja stanja je dodavanje proizvoda u košaricu:

```
class Narudzba {

    private final List<Proizvod> proizvodi = new ArrayList<>();

    public List<Proizvod> getProizvodi() {
        return proizvodi;
    }

    public void dodajProizvod(Proizvod proizvod) {
        proizvodi.add(proizvod);
    }
}

public class NarudzbaTest {

    @Test
    public void proizvod_dodan_u_narudzbu() {
        Proizvod proizvod = new Proizvod("sampon");
        Narudzba sut = new Narudzba();

        sut.dodajProizvod(proizvod);

        assertEquals(1, sut.getProizvodi().size());
        assertEquals(proizvod, sut.getProizvodi().get(0));
    }
}
```

Kôd provjerava stanje liste proizvoda nakon što je dodan proizvod.

Treći stil testiranja je ono koje verificira komunikaciju između sustava pod testom i njegovih suradnika (engl. *collaborators*). Čest primjer je slanje emaila, provjerava se poziv metode koja implementira slanje maila.

```
public class ControllerTest {
    @Test
    public void slanje_maila() {
        EmailGateway emailMock = mock(EmailGateway.class);
        Controller sut = new Controller(emailMock);

        sut.posaljiMail("username@email.hr");

```

```
9           verify(emailMock, times(1))
10                  .posaljiMail("username@email.hr");
11      }
12  }
```

Klasična škola testiranja preferira testiranje stanja nad komunikacijskim testiranjem, dok londonska preferira testiranje komunikacije nad testiranjem stanja. Oboje škole koriste testiranje izlaznih podataka. Zanimljiv je utjecaj stila testiranja na kvalitetu testova.

Zaštita od regresije ne ovisi o određenom stilu testiranja. Ona je rezultat sljedeća tri čimbenika: količine kôda koji se izvršava tijekom testa, složenosti tog kôda, njegove značajnosti za domenu. Općenito, može se napisati test koji pokriva koliku god količinu kôda – nema određenog stila testiranja koji donosi prednost u tom pogledu. Isto vrijedi i za složenost kôda te njegovu važnost za domenu. Jedina iznimka je stil testiranja temeljen na komunikaciji: pretjerana upotreba može rezultirati površnim testovima koji provjeravaju samo tanak sloj kôda, dok se za preostali kôd koriste *mock* objekti.

Postoji vrlo mala povezanost između stilova testiranja i brzine povratne informacije. Sve dok testovi ne koriste vanjske ovisnosti te ostaju unutar granica unit testiranja, svi stilovi proizvode testove s otprilike istom brzinom izvođenja. Testiranje temeljeno na komunikaciji može biti nešto sporije, jer *mock* objekti mogu uvesti dodatnu latenciju prilikom izvršavanja.

Kada je riječ o otpornosti na refaktoriranje, situacija je drugačija. Otpornost na refaktoriranje mjeri koliko lažnih alarma rezultata testovi generiraju tijekom refaktoriranja. Oni nastaju kada su testovi povezani s implementacijskim detaljima kôda, umjesto s njegovim vidljivim ponašanjem. Testiranje temeljeno na izlaznim vrijednostima pruža najbolju zaštitu od lažnih alarma jer su testovi povezani isključivo s metodom koja se testira. Jedini način da se takvi testovi povežu s implementacijskim detaljima jest ako je i sama metoda koju testiraju zapravo implementacijski detalj.

Testiranje temeljeno na stanju obično je podložnije kreiranjem lažnih alarma. Osim metode koju testira, takav test radi i sa stanjem klase. Statistički gledano, što je test jače povezan s produkcijskim kôdom, to je veća šansa da će se osloniti na implementacijske detalje. Budući da testovi temeljeni na stanju ovise o širem API-ju, veća je vjerojatnost da će biti povezani s implementacijskim detaljima.

Testiranje temeljeno na komunikaciji najosjetljivije je na lažne alarme. Većina testova koji provjeravaju interakciju s testnim duplikatima sklona je lomljivosti. To se uvijek odnosi na *stub* objekte (koji ne bi smjeli biti provjereni u testovima), dok su *mock* objekti prihvatljivi samo kada se provjeravaju interakcije koje prelaze granicu aplikacije i čiji su učinci vidljivi izvan sustava. Međutim, kao što površnost nije nužno obilježje testiranja temeljenog na komunikaciji, tako ni lomljivost nije njegova neizbježna karakteristika. Broj lažnih alarma rezultata može se smanjiti povezivanjem testova isključivo s vidljivim

ponašanjem. Ipak, razina pažnje potrebna za to varira ovisno o stilu unit testiranja.

Održivost testova usko je povezana sa stilovima unit testiranja. Ne može se učiniti mnogo da bi se poboljšala situacija. Održivost testova procjenjuje troškove održavanja testnog kôda i definira se kroz sljedeće dvije karakteristike:

- koliko je test teško razumjeti, što ovisi o njegovoj veličini,
- koliko je test teško pokrenuti, što ovisi o tome koliko vanjskih ovisnosti koristi.

Veći testovi su teži za održavanje jer ih je teže razumjeti ili mijenjati kada je to potrebno. Isto tako, test koji izravno koristi vanjske ovisnosti (npr. bazu podataka) manje je održiv jer zahtijeva dodatno vrijeme za upravljanje tim ovisnostima. U usporedbi s druga dva stila testiranja, testiranje na temelju izlaznih podataka je najodrživije. Rezultat su testovi koji su kratki i sažeti, što ih čini lakšima za održavanje. Prednost ovog pristupa proizlazi iz njegove jednostavne strukture, daje ulaznu vrijednost nekoj metodi i provjerava njezin izlaz.

Testovi temelji na stanju obično su manje održivi od testova temeljnih na izlaznim vrijednostima. Razlog tome je što provjera stanja često zauzima više prostora od provjere izlaznih vrijednosti. Čak i u jednostavnom slučaju s malo promjena, provjera stanja zauzima većinu linija samog testa.

Testovi koji se temelje na validaciji komunikacije imaju najgoru održivost u usporedbi s ostalim stilovima. Razlog tome je što oni zahtijevaju postavljanje testnih duplikata i definiranje interakcija, što povećava veličinu i složenost testova. Najveći problem su "lanci *mock* objekata" (engl. *mock chain*), gdje: *mock* objekti vraćaju druge *mock* objekte, ti objekti opet vraćaju nove *mock* objekte i to se ponavlja kroz više slojeva. Takvi lanci čine testove teškima za čitanje, razumijevanje i održavanje.

Slika 1.2 prikazuje usporedbu stilova unit testiranja i atributa dobrog unit testa. Kao što je objašnjeno u ovom poglavlju, sva tri stila imaju jednaku zaštitu od regresija i istu brzinu povratne informacije. Zbog toga ove metrike nisu uključene u usporedbu.

Testiranje koje verificira izlazne podatke daje najbolje rezultate. Ovaj stil testiranja se rijetko povezuje s implementacijskim detaljima, što znači da ne zahtijeva puno prilagodbi kako bi ostao otporan na refaktoriranje. Najodrživiji je, jer su testovi kratki i nemaju vanjske ovisnosti.

Testovi koji verificiraju stanje i komunikaciju su lošiji. Veća je vjerojatnost da će se povezati s implementacijskim detaljima, što povećava osjetljivost na promjene u kôdu. Skuplji su za održavanje, jer su veći i složeniji.

Potrebno je pisati najviše testova koji provjeravaju stanje. Nažalost, to je lakše reći nego učiniti. Ovaj stil testiranja primjenjiv je samo na kôd koji je napisan na funkcionalan način, što nije čest slučaj u objektno orijentiranim jezicima.

	Testiranje temeljeno na izlaznim podacima	Testiranje temeljeno na stanju	Testiranje temeljeno na komunikaciji
Otpornost na refaktoriranje	Niska	Srednja	Srednja
Trošak održavanja	Nizak	Srednji	Visok

Slika 1.2: Odnos odlika dobrog testa i stila testiranja.

1.3 Integracijsko testiranje

1.3.1 Važnost integracijskog testiranja

Ne može se tvrditi da sustav ispravno radi u cjelini oslanjajući se isključivo na unit testove. Unit testovi su izvrsni za provjeru poslovne logike, ali nije dovoljno testirati tu logiku izolirano. Potrebno je provjeriti kako različiti dijelovi sustava međusobno komuniciraju, kao i njihovu integraciju s vanjskim sustavima – bazom podataka, sustavom za razmjenu poruka i slično.

Unit test ispunjava sljedeća tri kriterija:

- provjerava jednu jedinicu (*unit*) ponašanja,
- izvršava se brzo,
- izvršava se u izolaciji od ostalih testova.

Ako test ne zadovoljava barem jedan od ovih kriterija, nije unit test, već spada u kategoriju integracijskih testova [9]. U praksi, integracijski testovi gotovo uvijek provjeravaju kako sustav radi u integraciji s vanjskim ovisnostima (engl. *out-of-process dependencies*). To znači da integracijski testovi pokrivaju kôd iz sloja kontrolera. Unit testovi pokrivaju domenski model. Integracijski testovi provjeravaju kako se taj domenski model povezuje s vanjskim ovisnostima.

Potrebno je održavati ravnotežu između unit i integracijskih testova. Rad s vanjskim ovisnostima čini integracijske testove sporima. Također, oni su skuplji za održavanje, zato što:

- vanjske ovisnosti moraju biti operativne tijekom testiranja,

- testovi postaju veći i složeniji zbog većeg broja suradnika.

Ali integracijski testovi imaju prednosti, pokrivaju veći dio kôda (napisan kôd i biblioteke koje se koriste u kôdu), bolje štite od regresija (jer provjeravaju kako sustav radi kao cjelina), otporniji su na refaktoriranje, jer nisu usko povezani s implementacijskim detaljima.

Omjer unit i integracijskih testova ovisi o specifičnostima projekta, ali općenito pravilo glasi: unit testovi trebaju pokriti što je moguće više rubnih slučajeva (engl. *edge cases*) poslovne logike, a integracijski testovi trebaju pokriti barem jedan uspješan put izvršavanja (engl. *happy path*) i rubne slučajeve koji se ne mogu testirati unit testovima.

Happy path je uspješno izvršenje poslovnog scenarija (npr. korisnik uspješno dovrši kupnju). *Edge case* je iznimka ili greška koja se može dogoditi tijekom izvršenja scenarija (npr. korisnik unese neispravan broj kreditne kartice).

Većina testiranja treba biti prebačena na unit testove, jer to smanjuje troškove održavanja. Ali, nekoliko integracijskih testova po poslovnom scenariju osigurava da cijeli sustav ispravno radi.

Integracijski testovi provjeravaju kako se sustav integrira s vanjskim ovisnostima. Postoje dva načina za implementaciju takve provjere: koristiti pravu vanjsku ovisnost ili zamijeniti tu ovisnost *mock* objektom.

Sve vanjske ovisnosti spadaju u dvije kategorije:

- Upravljane ovisnosti (engl. *managed dependencies*) – To su vanjske ovisnosti nad kojima imamo potpunu kontrolu. One su dostupne samo u aplikaciji, a njihove interakcije nisu vidljive vanjskom svijetu. Na primjer baza podataka – vanjski sustavi obično ne pristupaju vašoj bazi podataka direktno, već to rade preko API-ja aplikacije.

- Neupravljane ovisnosti (engl. *unmanaged dependencies*) – To su vanjske ovisnosti nad kojima nemamo potpunu kontrolu. Interakcije s njima su vidljive izvana. Na primjer SMTP server (za slanje e-mailova), *message bus* – oba proizvode nuspojave vidljive drugim aplikacijama.

Komunikacija s upravljanim ovisnostima dio je implementacijskih detalja, a komunikacija s neupravljanim ovisnostima dio je vidljivog ponašanja sustava. Pravilo glasi: za upravljane ovisnosti koriste se prave instance; za neupravljane koriste se *mock* objekti. Ako postoji objekt koji ima i upravljane i neupravljane ovisnosti, dio koji smatramo upravljanim bi trebali koristiti, a za neupravljani dio koristiti *mock* objekte.

1.3.2 Dobre prakse integracijskih testova

Postoje generalne smjernice koje ukazuju na kvalitetne integracijske testove:

- jasno definiranje granica domenskog modela
- smanjenje broja slojeva u aplikaciji
- eliminacija kružnih ovisnosti.

Kao i obično, najbolje prakse koje su korisne za testove također često poboljšavaju općeniti kôd.

Potrebno je jasno definirati mjesto za domenski model u kôdu. Domenski model je zbirka domena znanja o problemu koji tvoj projekt treba riješiti. Dodjeljivanje domenskom modelu jasne granice pomaže boljoj vizualizaciji i razmišljanju o tom dijelu kôda. Ova praksa također pomaže kôd testiranja. Unit testovi ciljaju na domenski model i algoritme, dok integracijski testovi ciljaju na kontrolere. Jasna granica između domena i kontrolera čini lakše uočljivom razliku između unit i integracijskih testova.

Većina programera prirodno gravitira prema apstrakciji i generalizaciji kôda uvođenjem dodatnih slojeva indiskrecije. U tipičnoj aplikaciji na razini poduzeća lako je primijetiti nekoliko takvih slojeva. U ekstremnim slučajevima, aplikacija može imati toliko puno slojeva apstrakcije da postaje previše teško razumjeti kôd i logiku čak i kôd najjednostavnijih operacija.

Previše slojeva indiskrecije negativno utječe na sposobnost razmišljanja o kôdu. Kada svaka funkcionalnost ima svoju reprezentaciju u svakom od tih slojeva, potrebno je potrošiti značajan napor kako bi se svi dijelovi sastavili u koherentnu sliku. Ovo stvara dodatno mentalno opterećenje koje otežava cijeli proces razvoja.

Potrebno je imati što manje slojeva indiskrecije. U većini *backend* sustava dovoljno je imati samo tri: domenski model, sloj aplikacijskih usluga (kontroleri) i infrastrukturni sloj. Infrastrukturni sloj obično se sastoji od algoritama koji ne pripadaju domenskom modelu, kao i kôda koji omogućuje pristup vanjskim ovisnostima.

Druga praksa koja može drastično poboljšati održivost kôda i olakšati testiranje je eliminacija kružnih ovisnosti. Kružna ovisnost su dvije ili više klase koje izravno ili neizravno ovise jedna o drugoj da bi pravilno funkcionirale. Baš kao i prekomjeran broj slojeva apstrakcije, kružne ovisnosti stvaraju ogromno kognitivno opterećenje kada je potrebno čitati i razumjeti kôd. Razlog je taj što kružne ovisnosti ne daju jasnu polazišnu točku od koje je moguće početi istraživati rješenje. Čak i mali skup međusobno povezanih klasa može brzo postati previše težak za shvatiti.

Kružne ovisnosti također ometaju testiranje. Često je potrebno posegnuti za sučeljima i imitiranjima kako bi se graf klasa podijelio i izolirao jednu jedinicu ponašanja, što je opet

problematično kada se radi o testiranju domenskog modela.

Sljedeća dobra praksa odnosi se na strukturu testa. Imati više od jednog *arrange*, *act* ili *assert* dijela u testu loš je znak. To je znak da taj test provjerava više jedinica ponašanja, što opet otežava održivost testa.

Na primjer, ako se testiraju dva povezana slučaja korištenja — registracija korisnika i brisanje korisnika — možda će biti primamljivo provjeriti oba ova slučaja u jednom integracijskom testu. Takav test mogao bi imati sljedeću strukturu:

- *Arrange* - Priprema podataka za registraciju korisnika.
- *Act* - Pozivanje metode `UserController.registerUser()`.
- *Assert* - Upit na bazu podataka za provjeru uspješne registracije.
- *Act* - Pozivanje `UserController.DeleteUser()`.
- *Assert* - Upit na bazu podataka za provjeru uspješnog brisanja.

Ovaj pristup je primamljiv jer korisnički tokovi prirodno slijede jedan za drugim, a prva radnja (registracija korisnika) može istovremeno poslužiti kao priprema za sljedeću radnju (brisanje korisnika). Problem je što takvi testovi gube fokus i brzo postaju previše napuhani. Najbolje je podijeliti test tako da se izdvoji svaka radnja u test za sebe. Kada svaki test fokusira na jednu jedinicu ponašanja, ti testovi postaju lakši za razumjeti i izmijeniti kada bude potrebno.

Izuzetak od ove smjernice su testovi koji rade s vanjskim ovisnostima koje je teško dovesti u željeno stanje. Recimo, na primjer, da registracija korisnika rezultira stvaranjem bankovnog računa u vanjskom bankovnom sustavu. Banka je osigurala *sandbox* za potrebe testiranja, gdje bi se *sandbox* koristio u *end-to-end* testovima.

Sandbox je izolirano testno okruženje u kojem se može sigurno izvršavati i testirati kôd, aplikacije ili datoteke bez utjecaja na glavni sustav. Često se koristi za sigurnosne testove, razvoj softvera i analizu zlonamjernog kôda. Problem je što je *sandbox* prespor ili možda banka ograničava broj poziva. U takvom scenariju, korisno je kombinirati više radnji u jedan test kako bi smanjio broj interakcija s problematičnom vanjskom ovisnošću.

Poglavlje 2

Testom vođen razvoj

2.1 Definicija testom vođenog razvoja

Testiranje je ključan dio razvoja softvera jer osigurava kvalitetu i funkcionalnost kôda. Potrebno je osigurati dosljedno pisanje testova, dovoljnu pokrivenost testovima i kvalitetu testova.

Jedan od pristupa koji rješava taj problem je testom vođen razvoj (engl. *Test Driven Development*, kraće TDD). To je način razvoja softvera u kojem testovi imaju centralnu ulogu i pišu se prije samog kôda. Glavna ideja TDD-a je strukturirani proces razvoja u kojem se svaka nova funkcionalnost implementira kroz unaprijed definirane korake [6].

Proces TDD-a sastoji se od sljedećih koraka:

1. Brzo dodati test.
2. Pokrenuti sve testove, zadnje dodan test bi morao pasti.
3. Napraviti malu promjenu.
4. Ponovno pokrenuti testove, svi testovi prolaze.
5. Refaktorizirati kôd, eliminirati duplikacije.

Na početku procesa potrebno je napraviti listu funkcionalnosti koje softver treba podržavati te ga prema potrebi u procesu nadopunjavati i prilagođavati. Ova praksa osigurava održiv kôd koji je lako testirati. Testom vođen razvoj se odvija u tri osnovne faze.

2.2 Faze unutar testom vođenog razvoja

2.2.1 Crvena faza

Crvena faza počinje pisanjem testa za kôd koji još nije napisan. Budući da funkcionalnost nije implementirana, test ne prolazi i simbolično se „crveni“. Ovaj korak prisiljava programera da jasno definira što očekuje od kôda i time precizno definira zahtjeve na sustav.

Pri pisanju testa programer sam odlučuje o veličini koraka unutar ove faze. Ako na listi funkcionalnosti vidi intuitivno jasan test, može ga odmah napisati, ako ne, može neku od funkcionalnosti podijeliti na manje dijelove i time si olakšati korak koji će kasnije nadograditi.

2.2.2 Zelena faza

Cilj zelene faze je najjednostavniji kôd za koji test prolazi, simbolično „zeleni“ se. Potrebno je napisati najjednostavniji mogući kôd koji zadovoljava zahtjeve postavljene u testu. Naglasak je na prolaznosti testa i funkcionalnosti dok se kvaliteta ostavlja za slijedeću fazu. Ovaj pristup omogućava brze iteracije i konstantnu provjeru napretka.

2.2.3 Faza refaktoriranja

Faza refaktoriranja fokusira se na poboljšavanje kvalitete kôda. U ovoj fazi se eliminiraju duplikacije, optimizira se postojeći kôd i primjenjuju se najbolje prakse razvoja softvera.

Ključna prednost ove faze je postojanje testova koji osiguravaju da promjene u kôdu ne narušavaju postojeću funkcionalnost. Pri svakoj promijeni u kodu pokreću se svi testovi, čime se osigurava da postojeće funkcionalnosti ostaju korektne. Time se minimiziraju greške, a stabilnost i korektnost sustava su garantirane.

Također, preporuča se jako mali korak u svakom refatoriranju, jedna mala promjena. Na taj način je programer svjestan promjena i lakše otkrije grešku.

2.3 Obrasci za dobro testiranje i obrasci za svaku fazu unutar testom vođenog razvoja

Kent Beck u svojoj knjizi *Test-Driven Development by Example* [10] predstavlja niz obrazaca (engl. *pattern*) koji olakšavaju primjenu testom vođenog razvoja metodologije, vodeći prema pisanju kvalitetnog, održivog i lako proširivog kôda. U nastavku poglavlja kratko su komentirani ti obrasci.

2.3.1 Obrasci testom vođenog razvoja

Kvalitetno testiranje ima direktan utjecaj na proces razvoja softvera. Viša razina stresa rezultira manjkom testova, manjak testova u više grešaka, a više grešaka u više stresa. Da bi se takav začarani krug izbjegao, potrebno je napisati test za svaku funkcionalnost. Postoji test koji provjerava da prethodno dodane funkcionalnosti nisu oštećene i nisu predstavljene nove greške. To je važan argument testom vođenog razvoja. Također, da bi bili sigurni da jedna greška ili novi kôd ne ruši sve testove, testovi trebaju biti izolirani. Dakle, svaki test testira jednu funkcionalnost. Takvim pristupom se lakše i brže uočavaju greške.

Odabir podataka nad kojima se izvode testovi ključno je za osiguravanje kvalitete testova. Nekoliko smjernica koje je poželjno slijediti:

1. Podaci korišteni u testovima trebaju biti što jednostavniji, osim kada razlika u podacima zahtijeva razliku u implementaciji. Na primjer, ako je za različite vrijednosti potreba različita implementacija, njihova primjena testira specifično ponašanje sustava i pomaže u otkrivanju potencijalnih problema.

2. Korištenje podataka koji simuliraju stvarne scenarije ključno je za testiranje relevantnosti sustava. Takvi podaci omogućuju testiranje sustava u uvjetima sličnima onima u uporabi.

3. Podaci korišteni u testovima trebaju jasno odražavati namjeru i svrhu testiranja. Na taj se način osigurava da su testovi razumljivi kako autorima, tako i budućim programerima koji će raditi na istom sustavu. Dokumentiranje razloga i motivacije iza korištenih podataka daje jasnije upute za održavanje i daljnji razvoj.

2.3.2 Obrasci u crvenoj fazi testiranja

Jedna od važnijih smjernica testom vođenog razvoja je postupno dodavanje testova. Iako je korisno pamtiti sve zadatke, još je bolje organizirati ih u obliku liste, u papirnatom obliku ili digitalno, na računalu. Takve liste daju jasan pregled napretka, omogućuju lakše praćenje preostalih zadataka i lakšu podjelu posla unutar tima. Održavanje liste zadataka osobito je korisno u situacijama kada određeni zadaci ostaju nedovršeni ili se odgađaju za kasnije održavanje. Ovaj pristup omogućava ostavljanje sustava u stanju u kojem svi testovi prolaze, a pokazuje i jasnu sliku u kojoj je fazi razvoj softvera i koji su sljedeći koraci. Stavke na listi trebaju biti dovoljno jednostavne da se mogu brzo implementirati, time se izbjegavaju zastoji i zbunjenost. Jednostavni zahtjevi omogućuju brzo prepoznavanje odnosi li se određeni zadatak dio postojeće funkcionalnosti, nova funkcionalnost unutar postojeće klase ili korak prema dodavanju nove klase i njene funkcionalnosti.

- Objašnjavajući test (engl. *explanation test*). Ova vrsta testa objašnjava specifičnu situaciju i što se dešava u tom slučaju. Objašnjavajući testovi su veoma korisni kôd

prvog upoznavanja sa sustavom jer, kako i samo ime kaže, objašnjavaju specifičan dio kôda.

- Učeći test (engl. *learning test*). Ova vrsta testa koristi se pri korištenju vanjskih resursa, kao što je API. Učeći testovi pokazuju što API vraća u različitim slučajevima i pomažu pri učenju kako se API koristi prije nego što se implementira u infrastrukturu softvera.

- Regresijski test (engl. *regression test*). Ovi testovi pokazuju što se očekuje za određeni korisnički unos i često se koriste kôd rješavanja prijavljenih grešaka od strane korisnika. Idealno bi bilo da su napisani kôd same implementacije prije kako ne bi došlo do greške. Ovi nas testovi uče nas kako unaprijediti proces testiranja i programiranja.

Ako razvoj naiđe na ozbiljne probleme, uvijek postoji opcija brisanja kôda i početka ispočetka. Rad u timu može uvelike pomoći jer rasprava problema s kolegama često vodi do novih ideja i rješenja. Problem koji za jednog člana tima djeluje složeno, drugome može biti lako rješiv. Zajednička razmjena ideja ključna je za učinkovit razvoj i održavanje sustava.

2.3.3 Obrasci u zelenoj fazi testiranja

U zelenoj fazi testiranja cilj je da test što prije prođe. U ovoj fazi dopuštene su greške i privremena rješenja jer je primarni fokus na prolaznosti testa, a kvaliteta i čistoća kôda se postižu u kasnijoj fazi razvoja.

Postoje tri osnovna pristupa koji nam omogućuju prolazak testa:

1. Lažiranje (engl. *Fake It*)
   Konstanta je najjednostavnija povratna vrijednost metode. Test prolazi. Konstanta se zatim postepeno zamjenjuje varijablom, uz prilagođavanje implementacije prema očekivanom rezultatu. Inicijalno vraćanje konstante garantira da je finalna implementacija također korektna jer svaki korak osigurava prolazak testa.

2. Triangulacija (engl. *Triangulation*)
   Pravila su dosta jednostavna u odnosu na *Fake It*. Piše se test i *assertion*. U metodi koja se koristi, kao povratnu vrijednost stavi se očekivani rezultat testa. Tada triangulacija omogućava dodavanje još jednog assertiona u test iz kojeg se vidi implementaciju koja bi trebala biti u metodi koja se testira.

3. Očita implementacija (engl. *Obvious implementation*)
   Ako se na listi nalazi stavka koja predstavlja jednostavnu operaciju s poznatom im-

plementacijom, moguće je odmah napisati implementaciju. Ako tijekom implementacije neki test ne prolazi, razvoj se može razlomiti u manje korake ili primijeniti pristup *Fake It* ili triangulaciju. Iako je ova metoda učinkovita, preporučuje se umjerena upotreba kako bi se izbjeglo stvaranje nepotrebno složenog kôda. Osim funkcionalnosti, važno je da kôd slijedi principe čistog kôda (engl. *Clean Code*).

2.3.4 JUnit obrasci

JUnit je programski okvir koji se koristi za testiranje. Obrasci za kvalitetno pisanje testova i smjernice za organizaciju unutar JUnit okvira uključuju:

1. Uspoređivanje (engl. *Assertion*)
   Kod pisanja testova ključno je koristiti detaljne i specifične usporedbe dobivenih i očekivanih vrijednosti. Na primjer, ako se provjerava površina lika, nije dovoljno provjeriti da je površina različita od nule, već provjeriti da je dobivena vrijednost točna izračunatoj površini lika. Provjere jednakosti moraju posve biti prepuštene računalu, bez potrebe za ljudskom procjenom. Dodatno, korisno je dodati specifične poruke o greškama za lakše debuggiranje.

2. Fiksirani set objekata (engl. *Fixture*)
   Fiksirani set objekata predstavlja zajednički set objekata koji se koristi u više testova. Postupak uključuje pisanje metode `setUp()` u kojoj se inicijaliziraju zajednički resursi. Time se smanjuje duplikacija kôda u testovima i olakšava održavanje u slučaju promjena.

3. Vanjski resurs (engl. *External Fixture*)
   U slučaju kada testovi koriste vanjske resurse, na primjer datoteke, preporučuje se korištenje metoda `setUp` i `tearDown`. Metoda `setUp` otvara datoteku, a `tearDown` je zatvara. Time se izbjegava ponavljanje kôda i greške poput nezatvorene datoteke.

4. Testne metode (engl. *Test Method*)
   Testne metode trebaju biti jasno organizirane i grupirane prema funkcionalnostima koje testiraju. Imena testova trebaju jasno označavati što se testira i koji rezultat očekujemo.

5. Test koji vraća iznimku (engl. *Exception test*)
   Testovi koji vraćaju iznimku koriste se za provjeru situacija u kojima dolazi do iznimki (engl. *exceptions*). Na primjer, takav test može provjeriti što se događa ako pokušamo pristupiti objektu koji ne postoji. Takvi testovi nam osiguravaju da sustav pravilno upravlja greškama.

6. Testovi koji se odnose na isti paket (engl. *package*) trebaju biti grupirani unutar istog paketa testova. Primjerice, prilikom dodavanja podklase, potrebno je osigurati da se zajedno s testovima za podklasu pokreću i testovi za osnovnu klasu, čime se osigurava dosljednost i integracija između klasa.

2.3.5 Obrasci u trećoj fazi razvoja (engl. *Refactoring*)

Refaktoriranje je važan korak u razvoju softvera koji omogućava poboljšanje kvalitete kôda bez promjene njegove funkcionalnosti. Fokus je na testovima koji prolaze i kôd koji je proizašao iz pisanja tih testova. Cilj je olakšati razumijevanje kôda, smanjiti duplikacije kôda i olakšati održavanje kôda.

Spajanje razlika

Kad postoje očite sličnosti između dijelova kôda, one se mogu spojiti radi pojednostavljivanja kôda. Manje očite sličnosti se mogu postepeno usklađivati kako bi se olakšalo refaktoriranje. Cilj je izbjeći kompliciranije promjene kod kojih je bitna kontrola toka i vrijednost podataka, ne uklapaju se u strategiju postupnih i malih koraka. Iako se nikada ne može do kraja izbjeći skokovito refaktoriranje, može se smanjiti njihovo pojavljivanje. Postupci spajanja sličnog kôda uključuju:

- Spajanje dviju sličnih petlji u jednu.
- Uklanjanje uvjeta kada su dvije grane slične.
- Brisanje jedne metode kada su dvije metode slične.
- Spajanje sličnih klasa i brisanje viška.

Izolirana promjena (engl. *Isolate Change*)

Kod potrebe za promjenom dijela metode ili objekta izolira se dio koji je potrebno promijeniti i na taj dio se stavlja fokus. Takav pristup omogućava preciznije izmjene i smanjuje rizik greške. Izoliranu promjenu moguće je provesti kroz ekstrakciju metode (engl. *Extract Method*), ekstrakciju objekta (engl. *Extract Object*) ili kreiranjem objekta iz metode (engl. *Method Object*).

Promjena reprezentacije podataka (engl. *Migrate Data*)

Kako bi se promijenila reprezentacija podataka često se koristi privremeno dupliciranje podataka. Koraci prema promijeni uključuju:

1. Dodavanje varijable instance u novom formatu.
2. Stari format varijable zamjenjuje se novim formatom.
3. Brisanje starog formata varijable.
4. Po potrebi, promjena sučelja kako bi podržavalo novi format.

Smanjenje metoda (engl. *Extract Method*)

Da bi se duga i kompliciranа metoda učinila čitljivom, potrebno je manji dio te metode odvojiti u novu zasebnu metodu. *Extract Method* je jedan od kompliciranijih refaktoringa, ali je vrlo često automatski implementiran, što olakšava situaciju. Koraci uključuju:

1. Pronalaženje dijela kôda metode koji ima smisla kao zasebna metoda. Vrlo često su to tijela petlje, cijele petlje ili uvjeti.
2. Provjera da u tom dijelu nema mijenjanja vrijednosti varijablama koje su inicijalizirane van tog dijela.
3. Kopiranje kôda u novu metodu.
4. Dodavanje parametra za svaku privremenu varijablu koju taj dio kôda koristi.
5. Zamjena starog dijela kôda pozivom nove metode.

Ovaj obrazac je koristan kada postoje slični dijelovi unutar metoda ili kada metoda postane prekomplicirana za razumijevanje.

Linijska metoda (engl. *Inline Method*)

Ovaj obrazac se koristi za pojednostavljivanje kontrole toka zamjenom poziva metode njezinim tijelom. Time dobijemo kôd u kojem nema potrebe za dodatnim proučavanjem metoda koje se pozivaju. Koraci su:

1. Kopiranje tijela metode.
2. Zalijepiti tijelo metode na mjesto poziva.
3. Zamjena parametara stvarnim parametrima i pridruživanje vrijednosti lokalnoj varijabli, ako je to potrebno.

Dodavanje sučelja (engl. *Interface*)

Kada više sličnih klasa sadrže metode istog imena i sličnog ponašanja, poželjno je dodavanje sučelja koje će te klase naslijediti. Time se smanjuje duplikacija kôda i smanjuje se mogućnost zaboravljanja implementiranja nekih metoda. Kod naknadnog dodavanja sučelja koriste se koraci:

1. Deklaracija sučelja. Ponekad je ime postojeće klase najbolje za sučelje, pa se ime klase treba promijeniti.
2. Nasljeđivanje sučelja od strane postojećih klasa.
3. Dodavanje potrebnih metoda u sučelje i po potrebi mijenjanje vidljivosti metoda u klasi.
4. Zamjena korištenja metode klase s metodama sučelja gdje je to moguće.

Dodavanje parametara metodi

Koraci za dodavanje parametra u metodu:

1. Dodavanje parametra u sučelje ako metoda pripada sučelju.
2. Dodavanje parametra u metodu.
3. Prilagodba svih poziva metode novim parametrima.

Dodavanje parametra je često kod implementacije novih funkcionalnosti. Također, koristi se kod mijenjanja reprezentacije podataka. Prvo se doda parametar, zamijeni se korištenje starog s ovim, te se na kraju korištenje starog parametra obriše.

Premještanje parametara u konstruktor objekta

Ako koristimo parametar u više metoda unutar istog objekta, možemo pojednostaviti kôd prosljeđivanjem tog parametra jednom, u konstruktoru. Možemo refaktorizirati i u obrnutom smjeru, na primjer, ako se parametar koristi samo u jednoj metodi objekta. Koraci uključuju:

1. Dodavanje parametara u konstruktor.
2. Dodavanje varijable istog imena kao parametar.
3. Stavljanje varijable u konstruktor.

4. Redom, mijenjanje reference parametra u `this.parametar`.

5. Kada nema više referenci na parametar, brisanje parametra iz metode i svih poziva metode.

6. Micanje `this` iz referenci na parametar.

7. Preimenovanje varijable u korektno ime.

2.3.6 Kvaliteta testova

Ako testovi osiguravaju sigurnost u kôd, vrlo vjerojatno su dobri. Generalno, ovo je lista koju je svakako potrebno testirati:

- Kondicionali
- Petlje
- Operacije
- Polimorfizam

Sada kad postoje potrebni testovi, potrebno je provjeriti njihovu kvalitetu. Postoji nekoliko pokazatelja koji ukazuju na to da testovi nisu dovoljno dobri:

- Dugo postavljanje okoline. Ako je potrebno više linija kôda za postavljanje uvjeta i objekata nego za pisanje testova, to je znak da su objekti preveliki.
- Duplikacija postavljanja okoline. Ako ne postoji zajedničko mjesto za postavljanje okoline unutar jednog testa, objekti su previše slični ili previše ovise jedan o drugome.
- Dugo vrijeme izvršavanja testova. Ako testovi u TDD pristupu traju dugo, neće biti često izvršavani, a ako nisu često izvršavani, vrlo vjerojatno ne rade. To je znak da postoji problem u testiranju izoliranih dijelova kôda, što je problem u dizajnu softvera.
- Preosjetljivi testovi. Testovi koji neočekivano ne prolaze su znak da jedan dio aplikacije neočekivano utječe na drugi dio. Potrebna je promjena u dizajnu kako bi se utjecaji eliminirali, ili odvajanjem dijelova ili njihovim spajanjem.

Testovi bi trebali pružati sigurnost da sustav radi kako se očekuje. Na primjer, ako se koristi trokut, u testovima koji provjeravaju dobivenu površinu trebali bi provjeriti barem sve vrste trokuta. Ako dođe do potreba za brisanjem testova, to je prihvatljivo ako brisanje ne smanjuje samopouzdanje u korektnost ponašanja sustava. Također, ako test govori o ponašanju sustava u raznim scenarijima, trebao bi ostati.

Zahvaljujući strukturiranim fazama razvoja, fokus je na jednom problemu koji se rješava, samim time svaki korak bude kvalitetnije odrađen i detaljnije promišljen.

2.4 Prednosti razvoja softvera vođenog testiranjem

Razvoj softvera vođen testiranjem donosi brojne prednosti koje doprinose kvaliteti i održivosti softverskog kôda. Ključna karakteristika testom vođenog razvoja je da testovi postaju osnovni produkt procesa razvoja, što osigurava pokrivenost kôda testovima, često u vrlo visokom postotku.

Jedna od glavnih prednosti je eliminacija dupliciranog kôda. Budući da se u procesu razvoja dodaje minimalno kôda koji omogućuju prolazak testova, kôd je sažet i optimiziran. Nadalje, treća faza razvoja, refaktoriranje, omogućuje dodatno poboljšanje kvalitete kôd, a istovremeno se osigurava očuvanje funkcionalnosti.

Testom vođen razvoj potiče programere na redovito pisanje testova prema definiranim zahtjevima na funkcionalnost, čime se osigurava konzistentna dokumentacija softverskog sustava. Osim toga, ovaj pristup naglašava važnost dizajna i strukture kôda, što rezultira modularnošću kôda i naknadnim lakšim održavanjem. Dakle, razvoj softvera vođen testiranjem ima značaje prednosti, među kojima se ističu visoka pokrivenost kôda testovima, kvaliteta kôda i stvaranje jasne dokumentacije razvojnog procesa.

2.5 Nedostaci razvoja softvera vođenog testiranjem

Unatoč svojim prednostima, razvoj softvera vođen testiranjem nije bez izazova i potencijalnih nedostataka. Visoka pokrivenost testovima, iako važna, ne garantira nužno kvalitetu samih testova.

Jedan od izazova je mogućnost da fokus na pisanje testova postane važniji od stvarne funkcionalnosti sustava. Time dobivamo sustav koji je dobro testiran, ali ne zadovoljava funkcionalne zahtjeve korisnika.

Testom vođen razvoj zahtijeva značajnu pripremu i učenje. Usvajanje ovog pristupa može biti vremenski zahtjevno, osobito za timove koji ne provode praksu redovitog pisanja testova. Dodatno, potrebno je održavanje i prilagodba testova kod dodavanja novih funkcionalnosti ili modificiranja postojećih. Iako testom vođen razvoj nudi iznimne mogućnosti

za poboljšanje kvalitete softvera, potrebno je uložiti vrijeme i resurse u učenje samog pristupa kako bi se koristio sa što više prednosti.

Poglavlje 3

Primjer testom vođenog razvoja

3.1 Opis sustava

U sklopu rada proveden je razvoj modula Inovacije unutar Informacijskog sustava znanosti Republike Hrvatske (CroRIS). Sustav CroRIS je izgrađen prema Idejnom rješenju, koje definira sustav kao nacionalni CRIS [3] temeljen na Common European Research Information Format (CERIF) [2] modelu podataka, s ciljem objedinjavanja i interoperabilnosti informacija o znanstvenoj djelatnosti u Hrvatskoj [4]. Izvedbeno rješenje nadovezuje se na ovu viziju, osiguravajući pouzdane podatke i podršku poslovnim procesima u znanosti i istraživanju [5].

Razvoj modula Inovacije vođen je testovima, a temeljna uloga modula je rad s podacima o inovacijama. Ključne su operacije nad vrijednostima objekata i njihovo ispravno spremanje u bazu podataka. Kako domenska logika nije kompleksna i ne zahtijeva složene algoritme, fokus je stavljen na točnost i konzistentnost podataka koji se pohranjuju. U takvim situacijama, prema knjizi [11], testiranje *repository* sloja nije nužno jer je obuhvaćeno integracijskim testovima.

U skladu s tim, kreiraju se unit testovi za validaciju *form* objekata te unit test za *converter* – klasu koja *form* objekt koji se koristi za prikaz na sučelju pretvara u domenski objekt koji se sprema u bazu i obrnuto. Time se osigurava točnost i konzistentnost podataka.

Zahtjevi na sustav opisani su u dokumentu, a uključuju kreiranje, ažuriranje, tablični prikaz inovacija, prikaz detalja inovacija i brisanje inovacija. Inovacija sprema podatke o autoru, povezanom profilu autora i povezanom profilu ustanove vlasnika. To su osnovni podaci o autoru koji su obavezni za unos. Naziv, jezik, ključne riječi, opis, prednosti, tržište i napomena su podaci za koje mora postojati originalan naziv, moraju postojati podaci na engleskom jeziku i mora biti omogućen unos podataka na više od jednog jezika. Od tih podataka, naziv, jezik, ključne riječi i opis su obavezni.

Što se tiče tehničkih detalja same inovacije, znanstveno područje, vrsta intelektualnog vlasništva, oblik industrijskog vlasništva i stupanj razvoja (TRL) obavezni su podaci. Inovacija se može povezati s drugim podacima iz baze podataka: osobama, projektima, ustanovama, patentima i uslugama. Mogu se evidentirati poveznice i nagrade dodijeljene inovaciji.

Kod povezivanja podataka, potrebno je omogućiti select izbornik koji prikazuje podatke za povezivanje iz baze podataka. Kod osoba, ustanova, projekata, patenata i usluga to su te vrste podataka koje su spremljene u bazi podataka. Kod uloga tih objekata i znanstvenog područja, vrste intelektualnog vlasništva, oblika industrijskog vlasništva i stupnja razvoja to su predefinirane vrijednosti također spremljene u bazu podataka pod određenom klasifikacijom.

U implementaciji se koriste unaprijed definirani objekti iz temeljnog (engl. *core*) projekta sustava, koji služi kao ovisnost (engl. *dependency*) te omogućuje nasljeđivanje i ponovno korištenje postojećih objekata. Među ključnim elementima ističu se form objekti i select izbornici.

Form objekti koriste se za prikaz podataka na korisničkom sučelju te dolaze s unaprijed definiranim anotacijama koje omogućuju opisivanje atributa, poput vrste objekta i placeholder vrijednosti. Ove anotacije osiguravaju konzistentnost podataka i olakšavaju validaciju unosa.

Select izbornici standardiziraju prikaz i selekciju podataka unutar korisničkog sučelja. Podaci za izbornike dohvaćaju se putem `Select2Controllera`, koji putem odgovarajućih endpointa komunicira s *repository* slojem. Uz to, *select* polja dolaze s validacijom koja osigurava da korisnik ne ostavi izbornik prazan, odnosno da određena vrijednost bude odabrana.

Za potrebe validacije koristi se i `javax.validation` biblioteka, koja omogućuje definiranje pravila za prihvatljive veličine podataka te provjeru smiju li određeni atributi ostati nedefinirani.

3.2 Primjeri i primjenjivanje dobrih praksi

Prema metodologiji testom vođenog razvoja ključno je održavati listu funkcionalnosti koje je potrebno implementirati. Početni popis sadrži sljedeće stavke:

- kreiranje inovacija,
- ažuriranje inovacija,
- prikaz detalja inovacije,
- tablični prikaz i pretraživanje inovacija,

- brisanje inovacije.

Nijedna stavka na popisu nije dovoljno jednostavna da bi se mogla implementirati odmah. Potrebno je stavku kreiranja inovacije podijeliti na manje dijelove.

- kreiranje inovacija,
- *form* objekt – unos podataka, obavezni podaci, višejezičnost, povezanost,
- domenski objekt,
- converter – pretvara *form* objekt u domenski i obrnuto,
- ažuriranje inovacija,
- prikaz detalja inovacije,
- tablični prikaz i pretraživanje inovacija,
- brisanje inovacije.

Prvi korak u implementaciji je definiranje form objekta koji predstavlja inovaciju i osiguravanje njegove ispravne validacije. To podrazumijeva provjeru unesenih podataka kako bi se zadovoljili svi preduvjeti.

3.2.1 Unit testovi – validacija *form* objekata

Form objekti služe za prikaz podataka na korisničkom sučelju te omogućuju njihovo ažuriranje od strane korisnika. Pomoću anotacija provodi se validacija unesenih podataka, pri čemu se provjerava ispravnost obaveznih polja, kao i ograničenja poput maksimalne dopuštene duljine teksta. Na primjer, unos naziva inovacije je obavezan, dok opis ne smije sadržavati više od šest tisuća znakova.

U crvenoj fazi razvoja prvo se piše test koji validira formu, čak i prije nego što ona postoji. Na temelju definiranih specifikacija kreiraju se testni scenariji koji pokrivaju sve slučajeve.

Pri testiranju objekta `InovacijaForm`, naziv testa slijedi pravila TDD metodologije, pa se klasa naziva `InovacijaFormTest`, dok nazivi metoda opisuju očekivani ishod testa i konkretnu situaciju, odvojeni donjom crtom.

```
@Test
public void validna_forma() {
    // arrange
    ValidatorFactory factory = Validation
                        .buildDefaultValidatorFactory();
    validator = factory.getValidator();
    InovacijaForm form = new InovacijaForm();

    form.setJezik("HR");
    form.setNaziv("Nova inovacija");
    form.setKljucneRijeci("Nova inovacija");
    ...

    // act
    Set<ConstraintViolation<InovacijaForm>> violations =
                            validator.validate(form);

    // assert
    assertTrue(violations.isEmpty());
}
```

Dodaje se test koji postavlja vrijednosti atributa *form* objekta i provjerava ispravnost njene validacije. Budući da klasa još ne postoji, kao ni metode za postavljanje podataka (*setter* metode), kôd se crveni. Prilikom pokretanja testa, postupak izgradnje ne može se dovršiti zato što ne postoji klasa `InovacijaForm`.

```
@Setter
public class InovacijaForm {

    private Integer id;
    private String naziv;
    private String kljucneRijeci;
    private String opis;
    private String napomena;
    ...
}
```

Nakon kreiranja klase `InovacijaForm` i potrebnih *setter* metoda, test prolazi, čime se prelazi u zelenu fazu razvoja. U ovoj fazi implementiran je minimalan kôd potreban za prolazak testa.

Slijedi faza refaktoriranja. Podaci unutar forme i izgled same forme moraju zadovoljavati uvjete opisane na početku poglavlja. Atributi koji zahtijevaju odabir vrijednosti putem select izbornika pretvaraju se u `Select2` objekte, a dio kôda u testovima u kojem se priprema okolina za testiranu situaciju prilagođava se novoj strukturi. Primjer postavljanja vrijednosti unutar *select* objekta koji se koristi u testovima:

```
Select2Option povezaniAutori = Select2Option.builder()
                                .id("1")
                                .text("Ivan Horvat")
                                .build();
```

Sukladno tome, mijenjaju se odgovarajući atributi u `InovacijaForm` klasi:

```
private Select2 povezaniAutori =
        new Select2AsyncMultiple(OSOBA_ENDPOINT);
```

Sljedeći korak u razvoju je osigurati provjeru obaveznih podataka te ograničenja duljine unesenih vrijednosti. U tim slučajevima očekuje se da forma neće biti validna. Testovi se definiraju tako da unose podatke koji ne ispunjavaju navedene uvjete, čime se provjerava ispravnost validacije i očekivano javljanje grešaka.

U testovima se u takvim slučajevima vrijednost obaveznim podacima postavlja na `null`, a podacima koji imaju ograničenu duljinu, na duljinu veću od ograničene. Kako postoji više obaveznih podataka, potrebno je provjeriti prisutnost svakog od njih. To dovodi do ideje testova u kojima u svakom pojedinom testu nedostaje jedan od obaveznih podataka. Međutim, takav pristup rezultirao bi velikim brojem sličnih testova unutar testne klase.

Kako bi se izbjegla redundancija, koristi se koncept parametriziranih testova. U tu svrhu kreira se klasa `InovacijeTestData`, koja sadrži iste atribute kao `InovacijaForm` i služi za postavljanje i korištenje vrijednosti u testovima. Vrijednosti za testiranje dodjeljuju se pomoću `Arguments Provider`-a, komponente iz JUnit biblioteke, koja omogućuje dinamičko prosljeđivanje testnih podataka. Na primjer, `Arguments Provider` može postaviti vrijednost `null` atributu `naziv`, dok su svi ostali atributi ispravno popunjeni. Na taj način provjerava se ispravnost validacije obaveznog atributa `naziv`.

```
@Override
public Stream<? extends Arguments> provideArguments(
    ExtensionContext context) {
     return Stream.of(
         Arguments.of(new InovacijeTestData("HR", null,
                 "Kljucne Rijeci", "Opis", "Prednosti",
                 "Trziste", "Napomena", "Ivan Horvat",
```

```
                "Ivan Horvat", "Ustanova", "Discipline",
                "industrijsko", "patent", "TRL1"))
    );
}
```

Za svaki od obaveznih podataka treba napisati test i navesti koja se poruka očekuje u tom testu:

```
assertEquals(1, violations.size());
ConstraintViolation<InovacijaForm> violation =
                violations.iterator().next();
assertEquals("{validation.notEmpty.message}",
                violation.getMessage());
```

Time se osigurava da je samo to greška u formi i da je došlo do očekivane greške.

U svakom testu potrebno je postaviti okruženje za `ValidatorFactory`. To se može odvojiti u zasebnu metodu koja se poziva prije svakog testa. To nam sugerira TDD metodologija kao princip fiksiranog objekta, pogledati 2.3.4. JUnit nudi anotaciju `@BeforeEach`, koja omogućuje postavljanje zajedničkog okruženja za svaki test.

```
private Validator validator;

@BeforeEach
public void setUp() {
    ValidatorFactory factory =
              Validation.buildDefaultValidatorFactory();
    validator = factory.getValidator();
}
```

Uvođenjem pomoćnih klasa i izdvajanjem metode, test postaje parametriziran. Sljedeći primjer prikazuje kako izgleda test za provjeru ispravnosti validacije forme kada nedostaju obavezni podaci:

```
@ParameterizedTest
@Transactional
@ArgumentsSource(InovacijeTestInvalidDataProvider.class)
public void forma_nije_validna_bez_obaveznih_podataka(
        InovacijeTestData inovacijaTestData)
{
    InovacijaForm form = new InovacijaForm();
    form.setJezik(inovacijaTestData.getJezik());
```

```
    form.setNaziv(inovacijaTestData.getNaziv());
    ...

    form.getPovezaniAutori().setSelectionSingle(
            inovacijaTestData.getPovezaniAutori());
    ...

    Set<ConstraintViolation<InovacijaForm>> violations =
                    validator.validate(form);

    assertEquals(1, violations.size());
    ConstraintViolation<InovacijaForm> violation =
                violations.iterator().next();
    assertEquals("{validation.notEmpty.message}",
                violation.getMessage());
}
```

Kako bi testovi prošli, potrebno je dodati odgovarajuće validacije u `InovacijaForm` klasu. Na primjer, za atribut `naziv` koji je obavezan i ne smije imati više od 200 znakova, koristi se `Size` i `NotEmpty` anotacije:

```
private static final int CONTRAINT_SIZE_STANDARD = 200;

@Input(type = InputType.TEXT,
       attrs = @Attr(key = "labelText",
                     value = "#{label.naslov}"))
@Size(max = CONTRAINT_SIZE_STANDARD,
      message = "{validation.size.message}")
@NotEmpty(message = "{validator.notEmpty.message}")
private String naziv;
```

Za polja koja koriste `select` izbornike, a obavezni su podaci, primjenjuju se anotacije naslijeđene iz baznog projekta. Na primjer:

```
@Select2NotEmpty(message = "{validator.notEmpty.message}")
private Select2 povezaniAutori = new
                Select2AsyncMultiple(OSOBA_ENDPOINT);
```

Sada test prolazi jer je `InovacijaForm` klasa ažurirana kako bi zadovoljila sve zahtjeve.

Vrijeme je za ažuriranje liste.

- kreiranje inovacija,
- *form* objekt - ~~unos podataka~~, ~~obavezni podaci~~, višejezičnost, povezanost,
- domenski objekt,
- *converter* – pretvara *form* objekt u domenski i obrnuto,
- ažuriranje inovacija,
- prikaz detalja inovacije,
- tablični prikaz i pretraživanje inovacija,
- brisanje inovacije.

Potrebno je omogućiti unos više jezičnih podataka, pri čemu je obavezan unos podataka na engleskom jeziku. Također, potrebno je odabrati jedan jezik kao originalan jezik podataka. Za tu potrebu kreira se nova forma koja predstavlja listu višejezičnih formi (`InovacijaMLListForm`), a u inicijalnoj `InovacijaForm` klasi uključuje se ta klasa. Dodaje se i prilagođena anotacija i validacija koja provjerava da postoji engleski jezik podataka i da nije odabran više od jedan originalan jezik. Validator prolazi kroz listu `InovacijaMLForm` klasa i provjerava zahtijevane uvjete. Korištenje prilagođene validacije:

```
@Valid
@InovacijaMLValid
private InovacijaMLListForm inovacijaMLListForm =
        new InovacijaMLListForm();
```

Sličnim koracima kao i za `InovacijaForm` klasu, gradi se funkcionalnost `InovacijaMLForm`klase. Kao krajnji rezultat, dobiva se klasa koja evidentira naziv, ključne riječi, opis, prednosti, tržište, napomenu i najvažnije, jezik tih podataka i oznaku originalnog jezika podataka. Korektnost ove forme će biti testirana kasnije u integracijskim testovima.

```
@Getter
@Setter
@NoArgsConstructor
public static class InovacijaMLForm {

    private static final int CONSTRAINT_SIZE_MEDIUM = 1500;
```

```
7
8       ...
9
10      @Input(type = InputType.CHECKBOX,
11            attrs = @Attr(key = "labelText",
12                          value = "#{label.trans}"))
13      private String trans;
14
15      @Input(type = InputType.SELECT,
16             attrs = {@Attr(key = "labelText",
17                            value = "#{label.jezik}"),
18                      @Attr(key = "sortAlphabetically",
19                            value = "false")})
20      @Select2NotEmpty(message = "{validator.notEmpty.message}")
21      private Select2 language =
22                     new Select2SyncSingle(getAllJezik());
23
24      ...
25
26      @Input(type = InputType.TEXT, attrs = @Attr(
27             key = "labelText", value = "#{label.notice}"))
28      @Size(max = CONSTRAINT_SIZE_MEDIUM,
29            message = "{validation.size.message}")
30      private String napomena;
31  }
```

Potrebno je implementirati funkcionalnost povezivanja inovacija s drugim podacima unutar sustava, kao što su osobe, projekti, ustanove, poveznice, patenti, usluge i nagrade. Kako bi se izbjeglo preopterećenje forme `InovacijaForm`, koja je postala prevelika za daljnje dodavanje tih povezanosti, kreira se nova forma `PovezanostForm` koja sadrži popis povezanih formi. Ovim pristupom omogućava se lakše razdvajanje zadataka tijekom programiranja, kao i jednostavniji prikaz tih povezanosti na korisničkom sučelju.

Testovi za forme imaju sličnu strukturu, razlikuju se samo po vrsti podataka i njihovoj ulozi. Stoga je dan primjer testa samo za jednu od formi – povezanost osoba s inovacijom. Forma će biti validna samo ako su uneseni podaci o osobi i njenoj ulozi u inovaciji. Potrebno je napisati test za validnu formu, za formu u kojoj nije evidentirana osoba i za formu u kojoj nije evidentirana uloga osobe. Primjer testa u kojem nije evidentirana osoba:

```
@Test
public void forma_nije_validna_bez_osobe() {
    PovezanostOsobaGroupForm form = new PovezanostOsobaGroupForm
       ();
    Select2Option uloga = Select2Option.builder()
                    .id("1")
                    .text("uloga␣osobe")
                    .build();
    form.getPovezanaOsobaUlogaId().setSelectionSingle(uloga);

    Set<ConstraintViolation<PovezanostOsobaGroupForm>> violations
        = validator.validate(form);

    assertEquals(1, violations.size());
    ConstraintViolation<PovezanostOsobaGroupForm> violation =
       violations.iterator().next();
    assertEquals("{validation.notEmpty.message}", violation.
       getMessage());
    assertEquals("povezanaOsobaId", violation.getPropertyPath().
       toString());
}
```

Dobivena forma koja povezuje osobu s inovacijom i evidentira ulogu osobe na inovaciji:

```
@Getter
@Setter
@NoArgsConstructor
public class PovezanostOsobaGroupForm extends Form {
    @Input(type = InputType.SELECT, attrs = {
            @Attr(key = "minimumInputLength", value = "3"),
            @Attr(key = "labelText", value = "#{label.povezanost.
               osoba.select}"),
            @Attr(key = "placeholder", value = "#{label.
               odaberiPlaceholder}")
    })
    @Select2NotEmpty()
    private Select2 povezanaOsobaId =
       new Select2AsyncSingle(Select2Endpoint.OSOBA_ENDPOINT);

    @Input(type = InputType.SELECT, attrs = {
            @Attr(key = "labelText", value = "#{label.povezanost.
```

```
                uloga.select}"),
16      })
17      @Select2NotEmpty()
18      private Select2 povezanaOsobaUlogaId =
19              new Select2SyncSingle(getClassByAlias(
20                      KLASA_OSOBE_INOVACIJA_KONTAKT.getCfAlias()));
21  }
```

Ovim je dobivena klasa `InovacijaForm` koja zadovoljava zahtijevane funkcionalnosti. Iako se čini da su neki koraci možda bili nepotrebni, TDD sugerira što manje korake sa što više refaktoriranja.

Stanje liste funkcionalnosti:

- kreiranje inovacije,
- prikaz detalja inovacije,
- tablični prikaz i pretraživanje inovacija,
- ~~form objekt~~ - ~~unos podataka~~, ~~obavezni podaci~~, ~~višejezičnost~~, ~~povezanost~~,
- domenski objekt,
- *converter* – pretvara *form* objekt u domenski i obrnuto,
- ažuriranje inovacije,
- brisanje inovacije.

Sljedeći je korak pisanje *converter* klase.

3.2.2 Unit testovi – validacija *converter* objekata

Converter je klasa koja služi za ”pretvaranje” objekta forme u domenski objekt, a pri dohvatu podataka iz baze, obrnuto – pretvara domenski objekt u formu koja se koristi za prikaz na korisničkom sučelju. Testovi za ovu funkcionalnost osiguravaju da su podaci koje je korisnik unio pravilno konvertirani u domenski objekt, što nam jamči da će podaci biti ispravno pohranjeni u bazu podataka.

`InovacijaForm` klasa već postoji. Pretpostavka je da će nakon konverzije, vrijednosti atributa u formi i domenskom objektu biti usklađene.

Primjer testa za pretvorbu forme u domenski objekt:

```
@Test
void convert_InovacijaForm_into_Inovacija() {
    InovacijaForm form = new InovacijaForm();
    form.setId(1);
    Select2Option povezaniAutori =  Select2Option.builder()
                        .id("1")
                        .text("Ivan␣Horvat")
                        .build();

    ...

    form.getPovezaniAutori()
        .setSelectionSingle(povezaniAutori);

    Inovacija inovacija = inovacijaConverter.convert(form);

    assertEquals(1, inovacija.getId());
    assertEquals("Ivan␣Horvat", inovacija.getAutori());

    ...

    assertEquals(1, inovacija.getPovezaniAutori());
}
```

U crvenoj fazi test pada jer ne postoji `InovacijaConverter` klasa, ni `Inovacija` klasa. Prvo se dodaje `InovacijaConverter` klasa. Kod pretvorbe form objekta u domenski:

```
@Component
public class InovacijaConverter implements Converter<
   InovacijaForm, Inovacija, Integer> {

    @Override
    public Inovacija convert(final InovacijaForm inovacijaForm) {
        Inovacija inovacija = new Inovacija();
        inovacija.setId(inovacijaForm.getId());
        inovacija.setVrstaIntelektualnogVlasnistva(
            inovacijaForm.getVrstaIntelektualnogVlasnistva().
               getIntSelectionSingle());
        ...

```

```
12          List<Osoba> autoriList = new ArrayList<>();
13          for (var autor : inovacijaForm.getPovezaniAutori().
               getSelectedOptions()) {
14              autoriList.add(new Osoba(Integer.parseInt(autor.getId
                   ())));
15          }
16          inovacija.setPovezaniAutori(autoriList);
17
18          ...
19
20          return inovacija;
21      }
22  }
```

Slično i u obrnutom smjeru, pretvorba iz domenskog objekta u form objekt. Potrebno je svakako dodati domenski objekt. Iz `converter` klase je jasno koji su atributi potrebni i koja je struktura povezanih podataka. Reprezentirani su kao liste objekata, a osnovni podaci se čuvaju u `Inovacija` klasi.

```
1  @Getter
2  @Setter
3  @NoArgsConstructor
4  @EqualsAndHashCode
5  public class Inovacija {
6      private Integer id;
7      private String autori;
8      ...
9      private List<Osoba> povezaniAutori;
10     private List<Ustanova> povezaneUstanove;
11     private List<Disciplina> discipline;
12     private List<InovacijaML> inovacijaMLList;
13 }
```

U ovom koraku je dobiven `converter`. Lista funkcionalnosti sada izgleda ovako:

- kreiranje inovacije,
- prikaz detalja inovacije,
- tablični prikaz i pretraživanje inovacija,
- ~~Form objekt - unos podataka, obavezni podaci, višejezičnost, povezanost,~~

- ~~domenski objekti~~ – ~~inovacija~~, ~~inovacijaML~~, ~~povezanost~~,
- ~~converter~~ – ~~pretvara form objekt u domenski i obrnuto~~,
- ažuriranje inovacije,
- brisanje inovacije.

Sljedeći je korak implementacija funkcionalnosti ažuriranja inovacije.

3.2.3 Integracijski testovi – validacija *controller* objekta

Kontroler ne smije sadržavati nikakvu poslovnu logiku – njegova jedina uloga je pozivanje odgovarajućih metoda. Sva logika i implementacija nalaze se u infrastrukturnom sloju aplikacije, koji se sastoji od `repository`, `facade` i `DAO` klasa.

Modul treba prikazivati početnu stranicu, stranicu sa tabličnim prikazom inovacija i stranicu koja prikazuje detalje inovacija. Na prikazu detalja inovacije treba postojati mogućnost uređivanja i brisanja inovacija. Za početak, fokusiramo se na prikaz detalja inovacije.

Anotacija `@AutoConfigureMockMvc` se u testovima koristi za automatsko konfiguriranje `MockMvc` objekta u Spring aplikacijama. HTTP protokol (engl. *HyperText TransferProtocol*) je mrežni protokol aplikacijkskog sloja za prijenos podataka koji služi za komunikaciju između poslužitelja (servera) i klijenta.

`MockMvc` je alat koji omogućuje testiranje HTTP zahtjeva i odgovora bez potrebe za pokretanjem cijelog aplikacijskog konteksta ili servera. To znači da se mogu testirati kontroleri i njihove rute (engl. endpoints) na jednostavan način, simuliranjem HTTP zahtjeva i provjerom odgovora, a sve to bez potrebe za stvarnim HTTP pozivima.

```
1  @SpringBootTest(classes =
       FullApplicationContextConfigurationScanner.class)
2  @AutoConfigureMockMvc
3  public class InovacijeControllerTest {
4
5      private final String contextPath;
6      private final Korisnik securityUserContextMock;
7      private final MockMvc mockMvc;
8      private final String EXISTING_INOVACIJA_ID = "296";
9
10     @Autowired
```

```
11      public InovacijeControllerTest(final MockMvc mockMvc,
12                  @Value("${server.servlet.context-path}") final
                        String contextPath) {
13          this.mockMvc = mockMvc;
14          this.contextPath = contextPath;
15          this.securityUserContextMock =
                SecurityPermissionsEnhancer.
                getEnhancedSecurityContextMock();
16      }
17
18      @Test
19      void dohvati_InovacijaForm_za_validan_id() throws Exception {
20          mockMvc.perform(get(contextPath + "/inovacija/" +
                EXISTING_INOVACIJA_ID)
21                  .with(user(this.securityUserContextMock))
22                  .contextPath(contextPath))
23              .andExpect(model().attributeExists("inovacijaForm", "
                    inovacijaId"))
24              .andExpect(status().isOk())
25              .andExpect(view().name("inovacijaDetails"));
26      }
27  }
```

Test u crvenoj fazi ne prolazi. Potrebno je dodati jednostavan kontroler koji kreira novu praznu formu, postavlja atribute modela na `null` i vraća odgovarajući view. Također, treba dodati HTML datoteku s imenom *view*-a kojeg kontroler očekuje. HTML je kratica za *HyperText Markup Language*, što znači prezentacijski jezik za izradu *web* stranica.

Nakon što se izvrše ove izmjene, test prolazi.

```
1   @Controller
2   @RequestMapping("/inovacija")
3   public class InovacijeController extends BaseController {
4
5       @GetMapping({"/{inovacijaId}"})
6       public String getInovacijaForm(
7                   @PathVariable final Integer inovacijaId,
8                   final Model model) {
9           InovacijaForm inovacijaForm = new InovacijaForm();
10          model.addAttribute("inovacijaForm", null);
11          model.addAttribute("inovacijaId", null);
12
```

```
13          return "inovacijaDetails";
14      }
15  }
```

U fazi refaktoringa potrebno je osigurati da atribut modela `inovacijaForm` bude inovacija s zadanim identifikatorom. Za to je potreban dohvat iz baze podataka. Kako je podijeljena funkcionalnost za povezanost inovacije i njenih osnovnih podataka, ista struktura se slijedi kod `repository` i `DAO` klasa. `Facade` klasa je tu da te rezultate spoji i prezentira kontroleru.

Potrebno je dodati metodu u `repository` koja će dohvatiti inovaciju s danim identifikatorom i napisati njenu implementaciju.

```
1  public interface InovacijeRepository extends CrudRepository<
       Inovacija, Integer> {
2      InovacijaForm getFormById(Integer id);
3  }
```

Implementacija te metode u `InovacijeRepository` klasi:

```
1  @Override
2  @Transactional(readOnly = true)
3  public InovacijaForm getFormById(final Integer inovacijaId) {
4      final var inovacija = this.getById(inovacijaId);
5      return this.inovacijaConverter.convertFrom(inovacija);
6  }
```

Klasa `InovacijeDao` nasljeđuje `CrudDao` i koristi implementaciju metode `getById`. `InovacijeDao` je povezan s MyBatis mapperom koji sadrži SQL kod koji se izvršava za svaku od metoda.

```
1  @Mapper
2  public interface InovacijeDao extends
3                          CrudDao<Inovacija, Integer> {}
```

`Facade` ujedinjuje rezultate i servira ih kontroleru.

```
1  @Component
2  public class InovacijeFacade {
3      private final InovacijeRepository inovacijeRepository;
4      private final PovezanostRepository povezanostRepository;
```

```
5
6       public InovacijeFacade(
7                   final InovacijeRepository inovacijeRepository,
8                   final PovezanostRepository povezanostRepository)
9       {
10          this.inovacijeRepository = inovacijeRepository;
11          this.povezanostRepository = povezanostRepository;
12      }
13
14      public InovacijaForm getInovacijaFormById(final Integer
            inovacijaId) {
15          InovacijaForm inovacijaForm = inovacijeRepository.
                getFormById(inovacijaId);
16          inovacijaForm.setPovezanostForm(povezanostRepository.
                getFormForInovacijaById(inovacijaId));
17          return inovacijaForm;
18      }
19  }
```

Sada se u kontroleru može koristiti dohvat `InovacijaForm` po `inovacijaId`, a test će potvrditi da je dohvat uspješan.

```
1   @GetMapping({"/{inovacijaId}"})
2   public String getInovacijaForm(
3           @PathVariable final Integer inovacijaId,
4           final Model model) {
5       InovacijaForm inovacijaForm = inovacijeFacade.
            getInovacijaFormById(inovacijaId);
6       model.addAttribute("inovacijaForm", inovacijaForm);
7       model.addAttribute("inovacijaId", inovacijaId);
8
9       return "inovacijaDetails";
10  }
```

Na popisu poslova može se prekrižiti prikaz detalja inovacije:

- kreiranje inovacije,
- ~~prikaz detalja inovacije~~,
- tablični prikaz i pretraživanje inovacija,
- ~~form objekt - unos podataka~~, ~~obavezni podaci~~, ~~višejezičnost~~, ~~povezanost~~,

- ~~domenski objekti~~ – ~~inovacija~~, ~~inovacijaML~~, ~~povezanost~~,
- ~~converter~~ – ~~pretvara form objekt u domenski i obrnuto~~,
- ažuriranje inovacije,
- brisanje inovacije.

Sada se može napisati test koji testira ažuriranje inovacije. Kako već `InovacijaFormTest` ispituje velik broj slučajeva, ovdje je potrebno ispitati samo one koje se ne mogu obuhvatiti u unit testovima. Takav primjer je inovacija kojoj su dva jezika postavljena kao obavezna. U tom slučaju forma nije validna.

```
1  @Test
2  @Transactional
3  void test_pada_za_dva_originalna_jezika() throws Exception {
4
5      final String postUrl = contextPath + "/inovacija/{id}/update"
           ;
6      final String expectedRedirectPath = contextPath + "/inovacija
           /" + EXISTING_INOVACIJA_ID + "/update";
7
8      MvcResult mvcResult = mockMvc
9              .perform(post(postUrl, EXISTING_INOVACIJA_ID)
10             .with(user(this.securityUserContextMock))
11             .contextPath(contextPath)
12             .param("inovacijaMLListForm.inovacijaMLForms[0].trans"
                   , "on")
13
14             ...
15             .param("inovacijaMLListForm.inovacijaMLForms[1].trans"
                   , "on")
16             ...
17             .andExpect(flash().attribute("errorNotificationMessage
                   ", notNullValue()))
18             .andReturn();
19
20     String redirectPath = mvcResult.getResponse().
           getRedirectedUrl();
21     assertEquals(expectedRedirectPath, redirectPath);
22 }
```

Za tu provjeru koristi se prilagođeni validator za kojeg je sada testirana funkcionalnost.

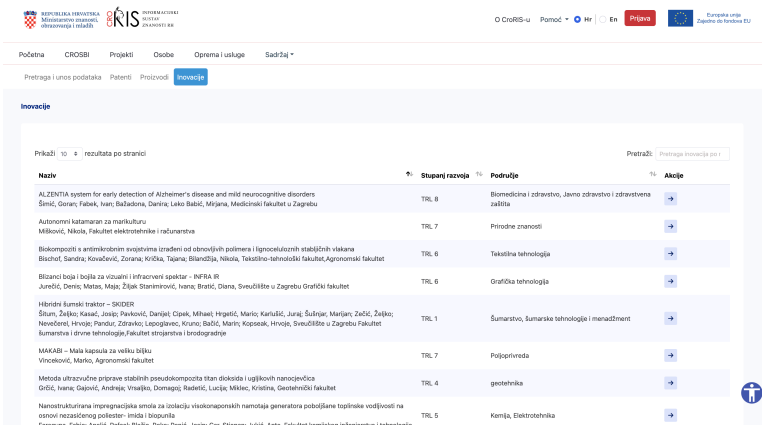
```
@Valid
@InovacijaMLValid
private InovacijaMLListForm inovacijaMLListForm = new
    InovacijaMLListForm();
```

Pripadnu metodu potrebno je implementirati u kontroleru i slično kao u prošlom primjeru, u facade, repository i DAO klasi. Krajnji rezultat je `InovacijeController` koji obavlja sve potrebne akcije nad inovacijama, a svaka od tih akcija je testirana, čime se osigurava njihova ispravnost.

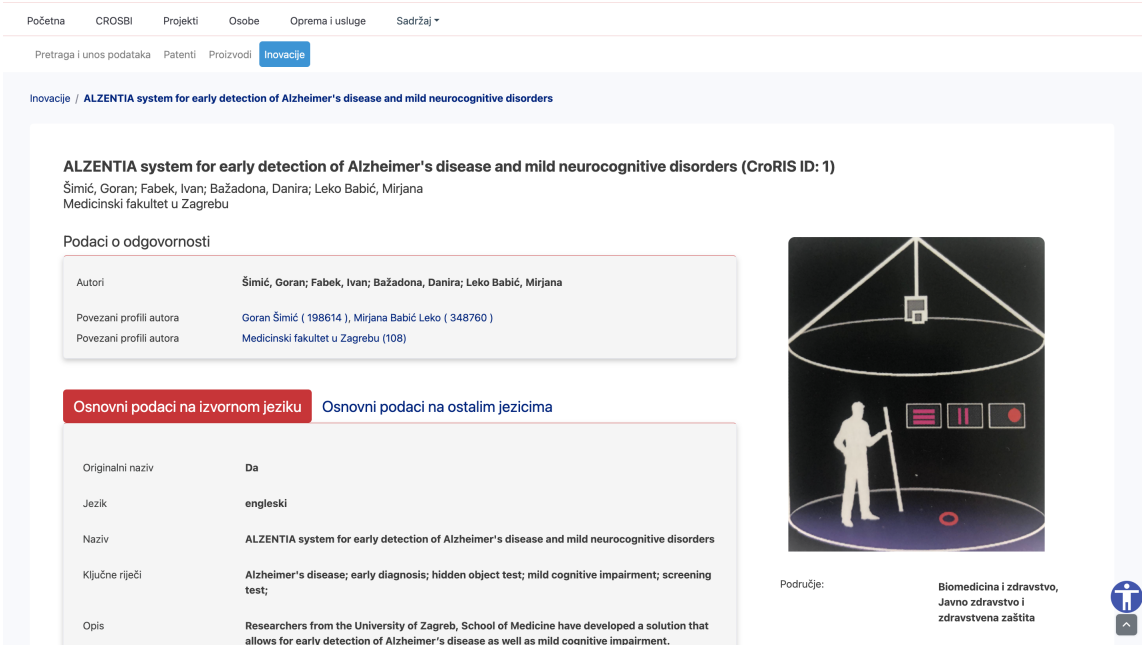
3.3 Rezultati primjene testom vođenog razvoja

Može se primijetiti da testiranje validacije forme možda nije nužno, s obzirom na to da se u testovima kontrolera mogu obuhvatiti sve mogućnosti formi. To bi značilo uklanjanje testova za *form* objekte, ali istovremeno povećanje broja testnih slučajeva u controller testovima. Posljedica toga bila bi veća kompleksnost i dulje vrijeme izvršavanja testova controllera. U unit testovima potrebno je testirati što više slučajeva, a integracijskim testovima je dovoljno provjeriti jedan "*happy path*" i slučajeve koje unit testovi ne mogu pokriti.

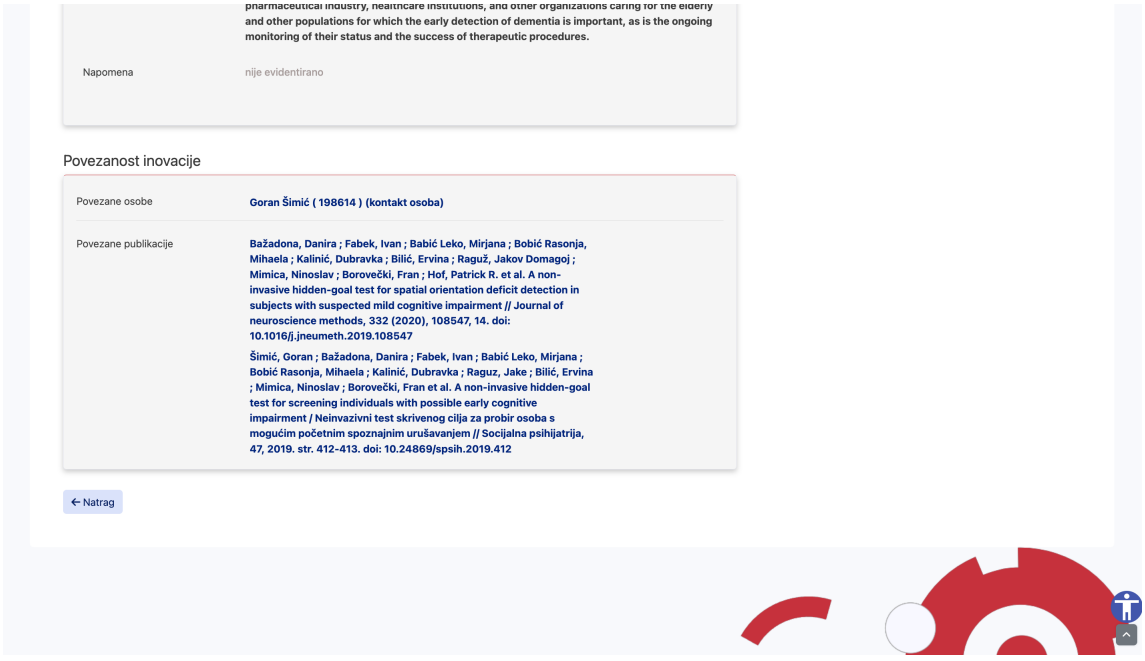
Sučelje implementiranog modula:



Slika 3.1: Tablični prikaz i pretraživanje inovacija.



Slika 3.2: Prikaz detalja inovacije.



Slika 3.3: Nastavak prikaza detalja inovacije.

Testom vođen razvoj rezultirao je modulom s visokom razinom pokrivenosti testovima. S obzirom na specifičnosti sustava, otprilike je jednak broj unit i integracijskih testova.

Iako su funkcionalnosti vezane uz proizvode i patente već ranije implementirane, potrebno ih je testirati kako bi se osigurala ispravnost sustava. Kao rezultat toga, sustav je temeljito testiran. Za provjeru pokrivenosti testovima korišten je alat *JaCoCo* [8]. Postignuta je pokrivenost testovima od 87% i pokrivenost grana testovima od 93% posto.

patenti

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
hr.srce.croris.patenti.infrastructure.repository.impl		77%		86%	63	256	204	884	48	201	0	17
hr.srce.croris.patenti.web.controller.sharedhandler		7%		50%	34	37	45	53	33	36	1	2
hr.srce.croris.patenti.web.form		89%		79%	52	316	22	390	43	289	0	27
hr.srce.croris.patenti.domain		97%		97%	43	983	6	348	14	419	0	32
hr.srce.croris.patenti.web.controller.select2		46%		0%	17	30	48	89	14	27	0	1
hr.srce.croris.patenti.infrastructure.parser.search		50%		25%	15	27	18	54	9	21	0	3
hr.srce.croris.patenti.security.accesspolicy.evaluator		8%		0%	9	11	31	35	6	8	0	2
hr.srce.croris.patenti.security.accesspolicy.policy		9%		0%	12	13	25	29	7	8	0	1
hr.srce.croris.patenti.security.accesspolicy.policy.impl		13%		n/a	13	17	19	23	13	17	0	2
hr.srce.croris.patenti.web.datatable		92%		94%	11	118	3	73	6	69	0	6
hr.srce.croris.patenti.infrastructure.facade		91%		n/a	3	27	6	89	3	27	0	6
hr.srce.croris.patenti.web.controller		97%		70%	8	67	3	203	1	55	0	7
hr.srce.croris.patenti.domain.search		83%		n/a	3	12	3	24	3	12	0	3
hr.srce.croris.patenti.errorhandling		65%		n/a	3	9	3	10	3	9	0	1
hr.srce.croris.patenti		16%		n/a	1	2	3	4	1	2	0	1
hr.srce.croris.patenti.security.accesspolicy.constants		0%		n/a	1	1	2	2	1	1	1	1
hr.srce.croris.patenti.web.converter		99%		93%	7	95	1	530	1	46	0	5
hr.srce.croris.patenti.domain.constants		98%		n/a	1	7	1	36	1	7	0	2
hr.srce.croris.patenti.web.validator		100%		98%	1	62	0	90	0	35	0	7
hr.srce.croris.patenti.security		100%		n/a	0	4	0	24	0	4	0	1
hr.srce.croris.patenti.security.aclrepository		100%		n/a	0	9	0	12	0	9	0	3
hr.srce.croris.patenti.web.validator.utils		100%		n/a	0	1	0	6	0	1	0	1
Total	2,143 of 17,464	87%	106 of 1,602	93%	297	2,104	443	3,008	207	1,303	2	131

Slika 3.4: Prikaz rezultata pokrivenosti testovima alata *JaCoCo*.

Poglavlje 4

Zaključak

Prvo poglavlje obrađuje koncept unit testiranja, pri čemu su objašnjeni njegovi ciljevi i prednosti, uz konkretne primjere primjene. Također su prikazane ključne karakteristike unit testova te njihova povezanost s korištenjem *mock* objekata i različitim stilovima testiranja. Na kraju poglavlja definirani su integracijski testovi i istaknuta je njihova važnost u procesu testiranja softvera.

Drugo poglavlje posvećeno je metodologiji razvoja vođenog testiranjem. Detaljno su opisane faze razvoja prema ovom pristupu te su predstavljeni obrasci koji pomažu u kreiranju kvalitetnih testova. Poseban naglasak stavljen je na obrasce primjenjive u svakoj fazi TDD-a kako bi se osigurao strukturiran i učinkovit razvoj softvera.

Treće poglavlje donosi praktičnu primjenu razvoja vođenog testiranjem kroz implementaciju modula Inovacije. Opisani su zahtjevi sustava, implementirani unit i integracijski testovi te analizirani njihovi rezultati. Također su prikazani ostvareni ishodi, uključujući korisničko sučelje i postotak pokrivenosti testovima.

Zaključno, kvalitetno testiranje nije samo tehnička nužnost, već i ključni preduvjet za dugoročnu stabilnost i skalabilnost sustava. Primjenom najboljih praksi testiranja osigurava se pouzdan i lako proširiv sustav spreman za buduće nadogradnje.

Bibliografija

[1] AAA Unit Testing, (2023), https://medium.com/@rojasjimenezjosea/aaa-unit-testing-688e3e61902a.

[2] Common European Research Information Format (CERIF), (2023), https://eurocris.org/services/main-features-cerif.

[3] Current research information system, (2023), https://eurocris.org/why-does-one-need-cris.

[4] Idejno rješenje Informacijskog sustava o hrvatskoj znanstvenoj djelatnosti - CroRIS, (2023), https://www.srce.unizg.hr/sites/default/files/srce/usluge/croris/idejnorjesenjecroriskonacno.pdf.

[5] Izvedbeno rješenje Informacijskog sustava o hrvatskoj znanstvenoj djelatnosti (CroRIS), (2023), https://www.srce.unizg.hr/sites/default/files/srce/usluge/croris/izvedbenorjesenjecroriskonacno.pdf.

[6] TDD, (2023), https://www.agilealliance.org/glossary/tdd/.

[7] Code Coverage Testing in Software Testing, (2024), https://www.geeksforgeeks.org/code-coverage-testing-in-software-testing/.

[8] Intro to Jacoco, (2024), https://www.baeldung.com/jacoco.

[9] Unit testing vs integration testing, (2024), https://circleci.com/blog/unit-testing-vs-integration-testing/.

[10] Kent Beck, Test Driven Development: By Example, Addison-Wasley, 2023.

[11] Vladmir Khorikov, Unit Testing: Principles, Practices, and Patteerns, Manning, 2020.

Sažetak

Testiranje softvera ključno je za osiguravanje održivosti, ispravnosti i pouzdanosti sustava. Kroz pravilno implementirane testove mogu se identificirati i otkloniti potencijalne pogreške u ranoj fazi razvoja, čime se smanjuju troškovi održavanja i poboljšava kvaliteta konačnog rješenja.

Prednosti testiranja su višestruke – od osiguravanja stabilnosti sustava, preko povećanja povjerenja u ispravnost koda, do omogućavanja sigurnih nadogradnji bez narušavanja postojećih funkcionalnosti. Korektnim i dosljednim testiranjem moguće je postići visoku razinu sigurnosti u rad sustava, a time i unaprijediti cjelokupni proces razvoja.

Kako bi testiranje bilo učinkovito, važno je osigurati da svi potrebni testovi budu prisutni i pravilno napisani. U tom kontekstu, pristup testom vođenog razvoja predstavlja dobru praksu koja omogućava sustavan način rada u kojem se testovi pišu prije same implementacije funkcionalnosti. Na taj način ne samo da se osigurava pokrivenost koda testovima, već se i potiče razvoj modularnog, čitljivog i održivog softvera.

Praktična primjena testom vođenog razvoja ilustrirana je na modulu Inovacije unutar Informacijskog sustava znanosti Republike Hrvatske (CroRIS). Metodologija se provodi kroz jasno definirane korake razvoja, koji osiguravaju visoku pokrivenost testovima i kvalitetan softverski kod.

Summary

Software testing is crucial for ensuring the sustainability, correctness, and reliability of a system. Through properly implemented tests, potential errors can be identified and fixed in the early stages of development, thus reducing maintenance costs and improving the quality of the final solution.

The benefits of testing are numerous – from ensuring system stability, to increasing confidence in the correctness of the code, and enabling safe updates without disrupting existing functionalities. Correct and consistent testing makes it possible to achieve a high level of security in system operation, thereby improving the overall development process.

For testing to be effective, it is important to ensure that all necessary tests are present and correctly written. In this context, the Test-Driven Development (TDD) approach represents a good practice that enables a systematic workflow in which tests are written before the actual implementation of functionality. This way not only is code coverage ensured by tests, but it also promotes the development of modular, readable, and maintainable software.

The practical application of test-driven development is illustrated in the Innovation module within the Croatian Research Information System (CroRIS). The methodology is implemented through clearly defined development steps, ensuring high test coverage and high-quality software code.

Životopis

Rođena sam 15. travnja 1998. godine u Varaždinu, gdje sam završila Srednju glazbenu školu i Prirodoslovno-matematičku gimnaziju. Tijekom školovanja sudjelovala sam na matematičkim natjecanjima i bila dio Centra izvrsnosti iz matematike, gdje sam razvila interes za studij matematike. 2017. godine upisujem Prirodoslovno-matematički fakultet u Zagrebu, smjer matematika, a 2022. godine stječem titulu sveučilišnog prvostupnika.

Nakon završetka preddiplomskog studija, odlučila sam nastaviti obrazovanje upisom na diplomski studij Računarstva i matematike. Tijekom studija stekla sam iskustvo u programiranju u Javi. Trenutno radim u Sveučilišnom računarskom centru Sveučilišta u Zagrebu (Srce), gdje kontinuirano učim i usvajam nove vještine.