Razvoj softvera vođen testiranjem

Franić, Tin

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tin Franić

RAZVOJ SOFTVERA VOĐEN TESTIRANJEM

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tin Franić

Matični broj: 44914/16–R

Studij: Poslovni sustavi

RAZVOJ SOFTVERA VOĐEN TESTIRANJEM

ZAVRŠNI RAD

Mentor:

Dr. sc. Marko Mijač

Varaždin, kolovoz 2021.

Tin Franić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Cilj ovog završnog rada je proučavanje razvoja softvera vođenog testiranjem (TDD), što je kombinacija testiranja i samog razvoja programskog koda. Temeljna pretpostavka takvog razvoja softvera je da prije bilo kakve promjene u kodu napišemo jedinični test koji tu promjenu provjerava. Primarna svrha takvog načina razvoja softvera je pisanje samo neophodnog koda uz što manje grešaka. U teorijskom dijelu ovog istraživanja precizno je definiran i opisan proces razvoja programa vođenog testiranjem, razlika TDD i tradicionalnog razvoja, vrste vođenog razvoja, njegove pozitivne i negativne strane, kao i najbitnije prakse i uzorci ovakvog načina razvoja. Primjenom teorijskog znanja u praktičnom dijelu razvijena je jednostavna aplikacija, a dinamika njenog razvoja vidljiva je u GitHub-u.

Ključne riječi: Razvoj softvera, testiranje, TDD, jedinično testiranje, agilni razvoj, programiranje

Sadržaj

1. Uvod

Od samih početaka razvoja softvera programeri su težili razvoju metoda programiranja koje bi omogućile što kvalitetniji softver uz korištenje što je manje resursa moguće. Tokom godina razvijene su mnoge metode, a ovaj rad proučava jednu od njih; razvoj softvera vođen testiranjem.

Razvoj softvera vođen testiranjem (engl. *Test-Driven development - TDD*) možemo definirati kao razvojnu metodu koja se sastoji od testiranja i klasičnog načina razvoja programskog koda. Kod takvog razvoja programa prije pisanja novog koda uvijek moramo napisati jedinični test koji testira novonastali programski kod. Nakon što napišemo kod koji zadovoljava test, taj kod možemo refaktorirati ukoliko smatramo da je to potrebno, ili možemo krenuti s pisanjem sljedećeg testa. U svojim začecima TDD je smatran jednom od praksi ekstremnog programiranja, koje je jedna od metoda agilnog razvoja. U današnjem svijetu TDD se vrlo često koristi u agilnim, ali sve češće i u planskim pristupima razvoju softvera.

Razlog zbog kojeg je razvoj softvera vođen testiranjem postao tako popularan je činjenica da ima mnoge prednosti u odnosu na tradicionalni razvojni proces, a najznačajnije su jednostavnost koda, bolji dizajn, fleksibilnost i manji broj grešaka. Upravo te prednosti ovakvog načina razvoja programa su značajan motiv za odabir ove teme za završni rad.

U teorijskom dijelu ovog istraživanja pokušat ću precizno definirati i opisati proces razvoja programa vođenog testiranjem koji se sastoji od tri temeljne faze. Dotaknuti ću se i testiranja, točnije jediničnog testiranja, koje je neizostavni dio ove metode razvoja softvera. Opisati ću pozitivne i negativne strane TDD-a, razlike u odnosu na tradicionalni razvoj softvera, vrste vođenog razvoja, kao i najbitnije prakse i uzorke koji se koriste kod ovakvog načina razvoja programa i rješavanja problema koji ga prate.

Primjenom teorijskog znanja u praktičnom dijelu razvit ću jednostavnu aplikaciju korištenjem TDD-a. Primjer će biti jedna od kata koda, aplikacija koja upisani broj pretvara u broj zapisan rimskim znamenkama. Izradit ću ju u C# jeziku, pri čemu ću koristiti Visual Studio razvojno sučelje. Za praćenje dinamike svake faze razvoja aplikacije koristiti ću se sustavom za kontrolu verzija GitHub.

Nakon svega navedenoga pokušat ću dokučiti koliko je zaista korisno učiti razvijati softver ovom metodom i zaslužuje li TDD razinu popularnosti koju uživa u današnjem svijetu razvoja softvera.

2. Razvoj softvera vođen testiranjem

Razvoj vođen testiranjem je pristup razvoju softvera pri kojem jedinični testovi imaju svrhu specificiranja i validacije onoga što kod treba izvršiti. Jednostavnijim riječima, test za svaku funkcionalnost je prvo napisan i pokrenut, nakon čega na red dolazi pisanje koda koji odgovara zahtjevima testa. [1]

Cilj takvog razvoja je čist kod koji radi, i to iz sljedećih razloga: [2]

- Pisanje čistog koda koji radi je predvidljiv način razvoja koda; uvijek znamo kada smo gotovi, bez brige o mogućim greškama u kodu
- Ako pri pisanju koda uvijek pišemo ono što nam prvo padne na pamet, ne stignemo razmisliti o drugim, potencijalno boljim rješenjima
- Čist kod poboljšava iskustvo krajnjeg korisnika softvera
- Čist kod olakšava suradnju između kolega u timu
- Pisanje čistog koda pruža zadovoljstvo onome tko ga piše

Svaki proces razvoja softvera vođenog testiranjem sastoji se od 3 temeljne faze, a to su: [2]

1. Crveno – napisati manji test koji ne prolazi, a možda se ne može niti kompajlirati
2. Zeleno – učiniti da test prolazi što je brže moguće, bez obzira na moguće greške
3. Refaktoriranje – eliminacija dupliciranja koda nastalog u prethodnom koraku

Nakon izvršenja ova tri koraka proces se ponavlja dokle god programer vidi načina za poboljšanje učinkovitosti koda.

Kako bi pojasnio potrebu za ovakvim, isprva neobičnim pristupom programiranju i testiranju, tvorac TDD-a K. Beck usporedio je programiranje s vađenjem vode iz bunara. Kada je kanta vode manja, njezino vađenje iz bunara je lako, no kad je kanta velika i puna vode umorit ćemo se prije nego ju izvadimo iz bunara. Stoga nam treba mehanizam sa zupčanikom kako bismo mogli odmoriti kada nam to bude potrebno. Teža kanta vode zahtijeva mehanizam sa gušćim rasporedom zubaca kako bi mogli odmoriti nakon manjih pomaka. Ti zupci predstavljaju jedinične testove TDD-a, što znači da teži programski zadatak zahtijeva pokrivanje manjih dijelova koda svakim testom. Svaki test koji prolazi nas čini korak bliže zacrtanom cilju, baš kao što nas svaki pomak na zupčaniku čini bliže vađenju kante iz bunara. [2]

2.1. Podrijetlo TDD-a

Razvoj softvera vođen testiranjem uglavnom povezujemo s agilnim metodama razvoja softverskog sustava, iako je danas prihvaćen i u planskim pristupima razvoju. Pojam TDD nastao je 1999. godine kao jedna od praksi ekstremnog programiranja (*engl. Extreme Programming - XP*). [3]

Ekstremno programiranje je metodologija razvoja softvera koja je potpuno podređena klijentu, a u središtu razvojnog procesa je kod. Zbog promjene fokusa sa zahtjeva prema sustavu na njegovu implementaciju, ekstremno programiranje izostavlja detaljne aktivnosti analize i dizajna koda. Uz TDD, neke od praksi ekstremnog programiranja su programiranje u paru, jednostavan dizajn, kontinuirana integracija, uključenost klijenta u razvoj itd. [4]

Idejni tvorac razvoja softvera vođenim programiranjem smatra se Kent Beck, iako i on sam priznaje da je originalna ideja mnogo starija. U svojoj knjizi *Test-Driven Development By Example* zahvalio se nepoznatom autoru knjige, koju je pročitao kao dječak, koji je sugerirao da prvo zapišemo očekivane izlazne podatke, a zatim razvijamo kod sve dok rezultati ne budu jednaki očekivanima. [2]

2.2. Jedinično testiranje

Neizostavan dio razvoja softvera je provjera ispravnosti koda, odnosno testiranje. Kod tradicionalnog integriranog pristupa testiranju, ono se izvodi nakon što je kod napisan, dok je kod TDD-a obrnuto, točnije prvo napišemo jedinični test koji nam služi kao polazna točka za razvoj koda. [5]

„Jedinični (engl. *Unit test*) test je test jedne izolirane komponente, sa ponavljanjem. Izolirani test znači da se komponenta koja se testira tretira u izolaciji, bez ostalih komponenti sustava (...) što je razlika u odnosu na integracijska testiranja.“ [5]

Komponenta koju testiramo može biti što god izaberemo, no obično je to linija koda, metoda, ili klasa. Ipak, općenito govoreći, ovdje vrijedi pravilo da je manje bolje. Testiranje manjih komponenti nam daje bolji uvid u performanse napisanog koda, odnosno lakše možemo uočiti u kojem dijelu koda se nalazi problem. Također, manji jedinični testovi se brže izvršavaju, a to je vrlo bitno kada imamo velik broj testova. [6]

Testove je poželjno ponavljati jer može doći do razlika u rezultatima pri jednom izvršavanju testa komponente i pri ponavljanju izvršavanja. Dakle, ako očekujemo da će određena komponenta testirana iz stanja A, poslije izvršavanja testa dati rezultat B, isto se

mora dogoditi i pri svakom naknadnom ponavljanju izvođenja tog testa. Ipak, promjena stanja komponente se tolerira ukoliko je unaprijed precizno definirana. [4]

Kod jediničnog testiranja programski kod često slijedi 3A uzorak pisanja testova koji se sastoji od tri dijela: [7]

- Arrange – sadrži sve predradnje neophodne za stvaranje okruženja za izvršavanje testa (npr. instanciranje objekata, inicijaliziranje varijabli..)
- Act – izvršavanje samog testa komponente i dohvaćanje rezultata testa
- Assert – provjerava se podudarnost dobivenih rezultata testa s očekivanim vrijednostima

2.3. Proces TDD-a

Prvi korak procesa razvoja svakog softvera je analiza zahtjeva korisnika, pa tako ni TDD nije iznimka. Nakon toga radi se grubi izgled i ideja softverskog sustava. Ideja cjelokupnog softvera se često mijenja, pa tako u svakom novom razvojnom ciklusu može doći do dodavanja novih metoda ili izbacivanja metoda koje su prethodno bile u planu. Pri razvijanju softvera klijenti nakon svakog razvojnog ciklusa dobiju uvid u trenutno stanje softvera, što olakšava razvoj i omogućuje da krajnji proizvod bude u skladu s potrebama klijenta. [3]

Kao što je već napomenuto razvoj softvera vođen testiranjem se svodi na to da prvo napišemo jedinične testove, a zatim kod koji odgovara zahtjevima tih testova. Testove pišemo jedan po jedan, što znači da tek nakon što napišemo kod komponente koja zadovoljava prvi test, krećemo na pisanje i zadovoljavanje drugog testa. Cilj pisanja i rješavanja testova je postizanje željene funkcionalnosti softvera uz što kraći i pregledniji kod. [3]

Kada prvi puta napišemo i pokrenemo test, on će pokušati instancirati klasu ili funkciju koja još nije implementirana, tako da test neće proći. Stoga je prvi korak ka rješavanju testa izrada klase i metode koja nam je potrebna. Zatim pišemo kod metode koji mora zadovoljiti test, a pri tome se ne moramo zamarati s onime što bi ostale metode klase trebale implementirati. Efekt takvog razvoja je kod jednostavniji za pregledavanje i održavanje. Nakon toga napisani kod refaktoriramo kako bi se riješili duplikacije koda, prepravili mu izgled u skladu s dobrom programerskom praksom itd. [3]

Ciklus razvoja softvera vođenog testiranjem se općenito dijeli na tri faze; crveno, zeleno i refaktoriranje. Takav poredak faza u razvoju softvera osigurava da imamo testove za kod koji ćemo pisati i da pišemo samo kod koji nam je neophodan kako bi napisani testovi prošli. [3]

2.3.1. Crveno

U prvoj fazi pišemo testove za kod koji u tom trenutku još ne postoji. Zbog toga se testovi neće kompajlirati, što znači da neće ni proći, zbog čega se ova faza i naziva crvenom. [3] Kent Beck ovu fazu uspoređuje sa pisanjem priče. Savjetuje da vizualiziramo kako bi funkcija u našoj glavi trebala izgledati u kodu i da zamislimo sučelje kakvo bismo htjeli. Uključivanje svih elemenata kojih se možemo dosjetiti u priču je nužno kako bismo mogli doći do pravih odgovora. [2]

2.3.2. Zeleno

Cilj druge faze osnovnog ciklusa TDD-a je taj da učinimo da test prolazi, zbog čega ovu fazu možemo zvati zelenom. Što brže postizanje tog cilja je u ovoj fazi daleko najbitnije. Ako je rješenje problema brzo, čisto i očito, možemo ga implementirati. S druge strane, ako je rješenje čisto i očito, ali mislimo da bi moglo potrajati nešto duže, bolje bi bilo napisati napomenu vezanu uz to i vratiti se na glavni problem, a to je što brži prolazak testa. U ovoj fazi brzi prolazak testa opravdava sve potencijalne greške koje putem učinimo. Ovakva drastična promjena u načinu programiranja može biti teška pojedinim iskusnijim programerima koji su navikli pratiti pravila kvalitetnog razvoja koda. [2]

2.3.3. Refaktoriranje

U trećoj fazi ciklusa TDD-a pokušavamo poboljšati kvalitetu koda, što omogućuje veću jednostavnost održavanja. Također, u ovoj fazi se rješavamo dupliciranja koda koja smo možda napravili u prethodnoj fazi. Ovdje nam testovi služe kako bismo uvidjeli utječu li načinjene promjene na prolaznost testova. Sve dok testovi prolaze znamo da je kod u redu. Nakon što završimo s fazom refaktoriranja koda i uvjerimo se da više ne postoje očiti načini poboljšanja koda, ciklus je završen i krećemo ispočetka s prvom fazom i novim testom. [3]

2.4. Razlika TDD-a i tradicionalnog razvoja

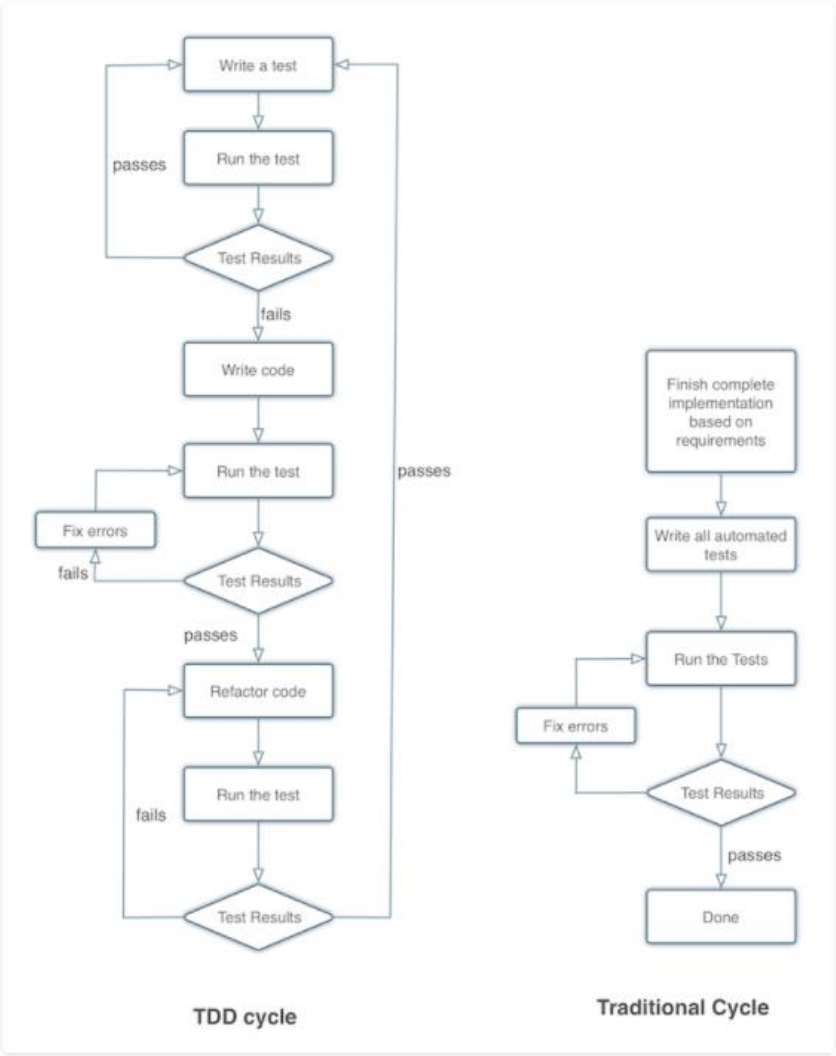
Kod tradicionalnog razvoja softvera projekt se izrađuje kroz nekoliko slijednih faza: prikupljanje zahtjeva, analiza, dizajn, kodiranje, testiranje, implementacija i održavanje. Ovakav način razvoja naziva se vodopadnim, jer se poput vodopada prelijeva s jedne razine razvoja na sljedeću, bez mogućnosti povratka. U skladu s time, sav posao vezan uz programiranje mora biti završen prije nego što testiranje može početi. [8]

Takav način razvoja ima mnoge mane. Ukoliko dođe do promjene zahtjeva usred razvoja projekta, ili shvatimo da smo napravili kritičnu grešku pri planiranju razvoja, može se

dogoditi da nemamo drugog izbora nego početi ispočetka. Upravo zbog toga vodopadni model razvoja često rezultira kašnjenjima i premašivanjem predviđenih financijskih sredstava. [8]

Razvoj softvera vođen testiranjem više je u skladu s agilnim metodama razvoja, koje su u posljednje vrijeme popularnije kod programera. Agilne metode više se temelje na fleksibilnosti, prilagodljivosti i zadovoljstvu klijenta, a manje na striktnim pravilima i podjelama razvoja na veće faze. Agilne metode zagovaraju razvoj u iteracijama, a pri svakoj iteraciji klijenti daju povratnu informaciju programerima. [8]

Kod razvoja softvera vođenog testiranjem postižemo potpunu pokrivenost testovima, što znači da je svaka linija koda testirana, što nije tako kod tradicionalnog načina razvoja i testiranja. TDD osigurava da napisani kod u potpunosti odgovara zahtjevima, što može pozitivno utjecati na samopouzdanje programera. [1]



Slika 1: Usporedba TDD-a i tradicionalnog razvoja

2.5. Vrste vođenog razvoja

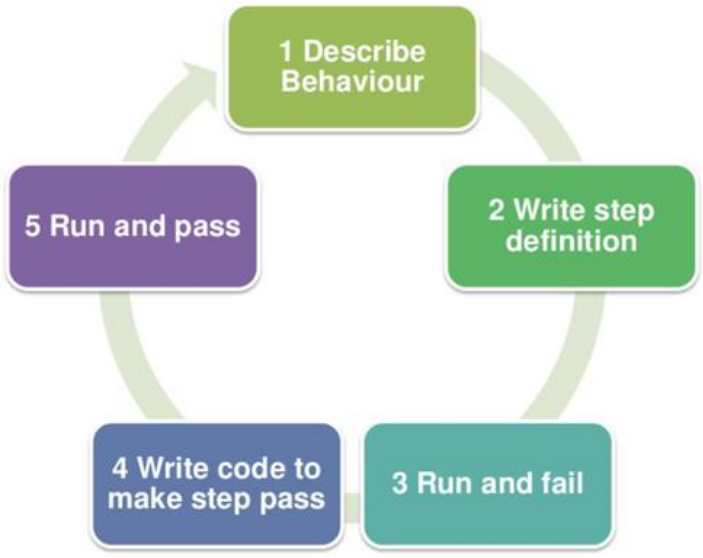
Pojam „vođeni razvoj“ označava metode razvoja softvera kod kojih se prati unaprijed isplanirani nacrt razvoja programa. Taj nacrt se očituje u testovima koji su napisani kako bi softver zadovoljio zahtjeve klijenata. Osim razvoja vođenog testiranjem (TDD), valja spomenuti razvoj softvera vođen ponašanjem (BDD) i razvoj softvera vođen testovima prihvaćanja (ATDD). [9]

2.5.1. Razvoj softvera vođen ponašanjem (BDD)

Razvoj softvera vođen ponašanjem (engl. *Behavior-Driven Development - BDD*) je metoda razvoja koja spaja principe TDD-a i razvoja softvera vođenog domenom (DDD), što je koncept kod kojeg je razvoj softvera uvjetovan dobrim poznavanjem procesa, pravila i strukture poslovne domene za koju je softver izrađen. BDD se može opisati kao aktivnost dizajniranja kod koje se dijelovi funkcionalnosti inkrementalno kreiraju u skladu s očekivanim ponašanjem softvera. Korisnici BDD-a rabe svoj materinski jezik u kombinaciji s jezikom DDD-a kako bi opisali svrhu i prednosti svog koda. Takav pristup pomaže ostalim stručnjacima koji rade na projektu da bolje razumiju implementaciju projekta jer ne moraju razumjeti programski kod. [9]

Ključne prednosti BDD-a su: [9]

- Pristupačnost široj publici zbog korištenja netehničkog jezika
- Pridaje pažnju funkcioniranju sustava iz klijentove i programerove perspektive
- Financijski isplativa metoda
- Smanjuje količinu rada potrebnu za saniranje grešaka koje se otkriju nakon isporuke softvera



Slika 2: Prikaz ciklusa BDD-a

2.5.2. Razvoj sofvera vođen testovima prihvaćanja (ATDD)

Razvoj softvera vođen testovima prihvaćanja (engl. *Acceptance Test-Driven Development - ATDD*) je metoda razvoja kod koje su klijenti uključeni u proces dizajniranja proizvoda i prije nego što je kodiranje softvera započelo. To je praksa u kojoj surađuju korisnici, testeri i programeri kako bi precizno definirali kriterije prihvatljivosti koje softver u konačnici mora zadovoljiti. ATDD osigurava da svi koji sudjeluju na određenom projektu znaju što točno treba biti učinjeno i implementirano. Svi testovi kod ovog načina razvoja softvera su specificirani prema poslovnoj domeni projekta, pa tako svaka funkcionalnost mora donositi stvarnu i mjerljivu poslovnu vrijednost. Testovi pružaju brzu povratnu informaciju o tome jesu li zahtjevi projekta zadovoljeni. Kada uzmemo sve u obzir, jasno je da je ATDD poprilično sličan BDD-u, no ATDD se više fokusira na jasno definiranje zahtjeva softvera dok je u fokusu BDD-a ponašanje softvera iz perspektive korisnika i programera. [9]

Prednosti razvoja projekta ovom metodom su: [9]

- Zahtjevi su jasno i nedvosmisleno analizirani
- Ovakav razvoj potiče suradnju različitih timova koji rade na projektu
- Testovi prihvaćanja služe kao vodič kroz cijeli proces razvoja softvera

2.6. Prednosti i nedostaci TDD-a

Razvoj softvera vođen testiranjem ima mnoge prednosti, ali kao i svaka druga metoda razvoja, ima i značajnih nedostataka, koji su naročito izraženi kod programera kojima je ovakva metoda razvoja nova. U ovom poglavlju obradit ću najbitnije prednosti i nedostatke razvoja vođenog testiranjem.

2.6.1. Prednosti

Glavne prednosti TDD-a su sljedeće: [10]

- Zbog pisanja manjih testova postižemo veću modularnost koda, pa nam tako TDD pomaže da naučimo i shvatimo ključne principe dobrog modularnog dizajna
- Ovakav način razvoja rezultira dobrom arhitekturom sustava jer pisanjem testova prije razvoja koda se razni arhitekturalni problemi često prije otkriju
- TDD često bolje dokumentira kod od same dokumentacije projekta
- Programski kod je lakše održavati i refaktorirati
- Olakšava suradnju; članovi tima mogu jedni drugima mijenjati kod bez straha jer će ih testovi obavijestiti ako se kod ponaša neočekivano

- Pomaže kod prevencije grešaka i rano ukazuje na mane dizajna i zahtjeva programa, što olakšava njegov razvoj
- Pomaže programerima da bolje razumiju kod
- Štedi vrijeme potrebno za testiranje nakon implementacije koda jer je kod već temeljito testiran
- Potiče razvoj u manjim inkrementima i poboljšava dizajn softvera jer potiče uklanjanje nepotrebnih dijelova koda
- Pomaže pri definiranju zahtjeva softvera jer nas potiče da konkretno odredimo ulazne i izlazne podatke
- Jedinični testovi su vrlo korisni kod izmjena u kodu, bilo to zbog dodavanja novih funkcionalnosti ili popravljanja *bugova.* S obzirom da održavanje softvera u prosjeku čini između 60 i 90% životnog ciklusa softvera, jasno je koliko je bitno kreiranje kvalitetnih jediničnih testova
- Testiranje pri razvoju koda potiče programera da piše kod koji je dovoljno čist da se može testirati
- Mnoge „glupe“ greške su gotovo odmah otkrivene, što štedi vrijeme i trud

2.6.2. Nedostaci

Glavni nedostaci razvoja softvera vođenog testiranjem su: [10]

- Testovi se moraju održavati
- Testove je često teško pisati, pogotovo testove koji nisu jedinični
- Ovakav pristup programiranju u početku može rezultirati sporijim razvojem softvera
- Pisanje kvalitetnih jediničnih testova nije lako. Menadžeri se često više fokusiraju na pokazatelje poput pokrivenosti koda testovima, no oni nam ne govore ništa o kvaliteti tih testova
- Cijeli tim mora imati povjerenja u jedinične testove kako bi TDD dobro funkcionirao
- Naučiti razvijati sofver ovom metodom ispočetka može biti vrlo izazovno, pogotov ako učimo sami, bez podrške iskusnijih programera. Naučiti TDD zahtijeva discipliniranost te mnogo vježbe i upornosti
- Teško je primjenjivo na postojeći programski kod
- Postoji dosta zabluda vezanih za TDD koje sprječavaju programere da ga nauče i koriste
- Programerima koji godinama koriste druge pristupe razvoju može biti teško priviknuti se na ovaj pristup razvoju

- Često moramo koristiti oponašajuće objekte umjesto skupih ili kompliciranih resursa (baze podataka, datotečni sustavi i slično) što zna biti zamorno, ali dugoročno vrlo korisno
- Sa dodavanjem sve više testova raste potreba za refaktoriranjem testova kako bi se isti brže izvršavali te kako bi se suvišni testovi uklonili
- Kao sa svakom metodom razvoja, sa TDD-om je moguće pretjerati, jer najveće prednosti ima umjerena primjena ovakvog razvoja. Ukoliko prečesto refaktoriramo testove, postoji dobra šansa da na to trošimo previše vremena
- Ponekad nam razne funkcionalnosti *frameworka* za testiranje mogu odvratiti pažnju od bitnih stvari, stoga valja imati na umu da se jednostavni testovi najbrže pišu i najlakše održavaju
- Kreiranje testova za pogreške u kodu može biti zamorno, ali na kraju se obično isplati
- Refaktoriranje koda u početnim fazama razvoja zahtijeva testne klase za refaktoriranje
- Ukoliko svi u timu ne održavaju ispravno svoje testove kvaliteta softvera se brzo može narušiti

3. Uzorci razvoja softvera vođenog testiranjem

Kako bismo što kvalitetnije razvijali softver metodom TDD-a poželjno je pratiti dobre uzorke i prakse razvoja softvera vođenog testiranjem. Najbitnije od njih obradit ću u ovom poglavlju.

3.1. Općeniti uzorci razvoja

U ovom poglavlju obradit ću najbitnije općenite uzorke TDD-a, a to su oni koji se odnose na cjelokupni proces razvoja softvera, a ne samo na njegove točno određene faze.

Tablica 1: Općeniti uzorci razvoja

Naziv uzorka	**Opis uzorka**
Izolirani test (engl. *Isolated Test*)	Kod pisanja testova prije razvoja softvera, u cilju nam je da ti testovi međusobno budu potpuno neovisni, kako ne bi zbog jednog palog testa svi pali, već samo jedan. Jedna od pozitivnih strana izoliranih testova je i nepostojanje točno određenog redoslijeda izvršavanja testova. Pisanje izoliranih testova vrlo je korisno i zbog toga što nas potiče na kreiranje manjih, modularnih cijelina koda, što je poželjno jer takav kod je mnogo lakše izmijeniti ukoliko dođe do potrebe za time. [2]
Lista testova (engl. *Test List*)	Prije nego krenemo pisati testove, dobra je praksa napraviti popis svih testova za koje znamo da ćemo ih morati implementirati. Ukoliko napredak u pisanju testova ne bilježimo nego ga pokušavamo zapamtiti, to može imati negativne posljedice. Pisanje testova često rezultira idejama za nove testove, što može odvlačiti pozornost programera i negativno utjecati na kvalitetu testova i koda koje pišemo, ali i na količinu stresa pri kodiranju. S druge strane, ignoriranje potencijalnih testova koji nam padnu na pamet također nije poželjno jer ćemo gotovo zasigurno zaboraviti neki važan test što će utjecati na kvalitetu softvera. [2]
Dizajn za testiranje (engl. *Design for Testability*)	Pisanje testova prije pisanja koda rezultira programskim kodom koji je dizajniran tako da se može testirati, pa tako ne moramo

	razmišljati može li se naš kod testirati. Testiranje koda koji nije razvijen metodom TDD-a može biti značajan izazov za programere. [11]
Prvo *Assert* (engl. *Assert First*)	Određenu funkcionalnost softvera je najbolje započeti definiranjem testova koje ta funkcionalnost treba zadovoljiti. Pisanje testova je dobro započeti definiranjem *assert* dijela testa, a to je onaj dio testa u kojem provjeravamo odgovaraju li rezultati testa vrijednostima koje smo očekivali. Nakon toga programer piše ostatak testa redoslijedom koji želi, krećući od početka ili od kraja testa. [2]
Kod koji se ne može testirati (engl. *Minimize Untestable Code*)	Neki dijelovi koda teško se mogu testirati, primjerice komponente grafičkih sučelja, višedretveni kod i testne metode. Takav kod je teže refaktorirati ili nadograditi na siguran način, stoga trebamo težiti pisanju što manje koda koji se ne može testirati. [11]
Testni podaci (engl. *Test Data*)	Testni podaci bi uvijek trebali biti jednostavni za čitati i pratiti. Nije poželjno koristiti mnogo podataka u testovima, osim ako između tih podataka ne postoji značajna smislena razlika. Ipak, sustav koji očekuje više različitih ulaznih podataka treba testirati u skladu s time; s više različitih testnih podataka. Kod definiranja testnih podataka, ne savjetuje se korištenje iste konstante za više od jedne stvari. Alternativa izmišljanju testnih podataka je korištenje realnih podataka, tj. podataka iz stvarnog života. [2]
Evidentni podaci (engl. *Evident Data*)	Napisani testovi trebali bi biti razumljivi svakom programeru, a ne samo onome koji ih piše i računalu. Stoga je dobra praksa u kodu ostaviti što više tragova koji pojašnjavaju što svaka linija koda radi. Takav način pisanja koda može koristiti ne samo drugim programerima, nego i onome koji ga je pisao, ukoliko nakon mnogo vremena bude morao raditi izmjene u softveru. [2]
Proporcionalan napor (engl. *Commensurate Effort and Responsibility*)	Količina truda i napora potrebna za pisanje ili modificiranje testova ne bi trebala premašivati količinu truda potrebnu za implementiranje odgovarajuće funkcionalnosti. U skladu s time, alati potrebni za pisanje testova ne bi trebali biti zahtjevniji za korištenje od alata korištenih za implementiranje funkcionalnosti. [11]

3.2. Uzorci za fazu Crveno

U ovom poglavlju bavit ću se dobrim uzorcima koji se koriste pri pisanju faze Crveno. U fazi Crveno zadatak programera je kreirati test za kod koji još ne postoji. S obzirom na to test neće proći, zbog čega je ta faza i dobila naziv Crveno.

Tablica 2: Uzorci za fazu Crveno

Naziv uzorka	**Opis uzorka**
Test u jednom koraku (engl. *One Step Test*)	Nakon što napišemo listu testova, sljedeći korak je odabir testa kojeg ćemo implementirati. Glavni kriterij kod odabira testa je da odaberemo test koji će nas nečemu naučiti i za koji smo sigurni da ćemo ga moći implementirati. Dakle, cilj je naći test koji nam predstavlja izazov s kojim ćemo se moći nositi. Svaki test na popisu bi trebao predstavljati jedan korak prema krajnjem cilju. Odabir testa kojeg ćemo sljedećeg implementirati je individualan za svakog programera. [2]
Početni test (engl. *Starter Test*)	Pri pisanju stvarnog testa neke funkcionalnosti softvera, moramo u istom trenutku razmišljati gdje taj test pripada i koji su ispravni ulazni i izlazni podaci testa. Takvim razvojem testa dugo nećemo doći do povratne informacije. Kako bismo to izbjegli, možemo započeti s testiranjem operacije koja je slična onoj koju želimo implementirati, ali zapravo ne radi ništa. Kod pisanja početnog testa vrijedi i pravilo testa u jednom koraku. Poželjno je izabrati početni test koji će nas naučiti nečemu, a da smo sigurni da ćemo ga moći brzo implementirati. [2]
Objašnjavajući test (engl. *Explanation Test*)	Osim što nam služe kako bi testirali napisani kod, testovi nam mogu služiti i kao objašnjenje programskog koda, pa tako programer može ponuditi ili zatražiti od drugih objašnjenje u obliku testa. To može biti vrlo korisno jer pomaže drugim programerima lakše shvatiti kod. Ovakva praksa može pomoći i pri većim razinama apstrakcije. Primjerice, ako nam kolega pokušava objasniti dijagram slijeda, možemo taj dijagram pretvoriti u testove ukoliko nam je tako predočen problem lakše shvatiti. [2]

Test za učenje (engl. *Learning Test*)	Kod korištenja softvera iz vanjskih izvora dobra je praksa da, prije njegovog korištenja, napišemo test koji potvrđuje da sučelje funkcionira na očekivani način. Svaku novu verziju takvog softvera je poželjno testirati prije korištenja, kako bi brže saznali je li s njim sve u redu. U slučaju da testovi prolaze, možemo sa sigurnosti ustvrditi da se softver ponaša kako bi trebao i možemo nastaviti s korištenjem tog softvera. [2]
Regresijski test (engl. *Regression Test*)	Promjene u kodu često mogu izazvati iznenađujuće probleme; primjerice uzrokovati *bugove* u metodama koje su naizgled nepovezane s novonastalom promjenom. Kreiranjem regresijskih testova osiguravamo ne samo da naša promjena u kodu čini ono što se očekuje, već i da nije uzrokovala probleme u metodama koje se prolazile prijašnje testove. [12] Kada uočimo pogrešku u kodu koju smo prethodno previdjeli napravit ćemo test kojim ćemo spriječiti da nam se takva greška ponovi. [3]
Početi ispočetka (engl. *Do Over*)	U slučaju da se osjećamo izgubljeno i ne vidimo načina da riješimo neki problem na način kojim smo započeli, možda je najbolje početi ispočetka. Situacija u kojoj, i nakon dužeg odmora, ne možemo naći način kako napisati određeni test, a na pamet nam pada još mnoštvo drugih testova, dobar je primjer toga. Iako nam instinkt sugerira da pokušamo problem riješiti kako god znamo i nastavimo u istom smjeru, često je najbolje rješenje izbrisati sve što smo do tada napravili i krenuti ispočetka. [2]

3.3. Uzorci za testiranje

Ponekad može biti teško pravilno testirati određene dijelove koda. Kako bi takve situacije riješili što uspješnije, dobro je pratiti uzorke za testiranje, koje ću obraditi u ovom poglavlju.

Tablica 3: Uzorci za testiranje

Naziv uzorka	**Opis uzorka**
Manji test (engl. *Child Test*)	U slučaju da smo napisali test koji je prevelik i zbog toga predstavlja problem, možemo napraviti manji test koji predstavlja dio većeg testa. Rješavanjem manjeg testa lakše će nam biti riješiti i veći test. [2]
Lažni objekt (engl. *Mock Object*)	Testiranje objekta koji se oslanja na skupe ili komplicirane resurse često nije praktično, stoga kreiramo lažnu verziju resursa koja oponaša originalni. Primjer skupog i kompliciranog resursa je baza podataka. Osim što pospješuju izvedbu koda, lažni objekti pozitivno utječu i na razumijevanje koda. Kada testiramo bazu s realnim podacima, ponekad je teže zaključiti jesu li podaci koje je upit vratio ispravni. [2]
Zapis u stringu (engl. *Log String*)	Kada provjeravamo jesu li sve metode pozvane i jesu li pozvane željenim redoslijedom, korisnim se mogu pokazati zapisi u stringu. Ova metoda manifestira se tako da svaka pozvana metoda dodaje u string svoj specifični dio. Nakon što se sve metode izvedu, dobiveni string usporedimo sa očekivanim stringom. Ukoliko su oni jednaki, možemo sa sigurnošću ustvrditi da su sve metode izvedene očekivanim redoslijedom. [2]
Lutka za testiranje sudara (engl. *Crash Test Dummy*)	Postoje dijelovi koda koji će iznimno rijetko biti korišteni, no ipak ih valja testirati. To je najbolje učiniti pomoću posebnog objekta koje će baciti iznimku. Uzmimo za primjer da želimo testirati što će se dogoditi s našim softverom u slučaju da je datotečni sustav pun. Mogli bismo stvoriti mnogo velikih datoteka i zaista napuniti sustav, no možemo i simulirati pun datotečni sustav. Osim što olakšava izvršavanje koda koji se rijetko poziva, ova metoda čini test jednostavnijim za čitanje. [2]

Test koji ne prolazi (engl. *Broken Test*)	Kada programiramo sami, ostaviti zadnji test nezadovoljenim prije završetka rada može biti dobra praksa. Kad se vratimo programiranju sljedeći dan ili tjedan, test koji ne prolazi može biti jak motiv za nastavak rada. Takvim pristupom osiguravamo da nam je očigledno gdje nastaviti, a lakše nam je i prisjetiti se o čemu smo razmišljali, što nam olakšava da brže nastavimo gdje smo stali. [2]
Programiranje u timu (engl*. Clean Check-In*)	Kada programiramo u timu, bitno je prije završetka rada pokrenuti sve testove. Isto valja učiniti i prije nego nastavimo s radom na softveru jer ne znamo sa sigurnošću u kojoj mjeri je kod izmijenjen od trenutka kada smo zadnji put bili u kontaktu s njim. Prolazak svih testova pokazati će da kod radi onako kako očekujemo, što će nam pružiti sigurnost i samopouzdanje potrebne za kvalitetan nastavak rada. [2]

3.4. Uzorci za fazu Zeleno

U ovom poglavlju obradit ću dobre uzorke koji se koriste kod pisanja faze Zeleno. U fazi Zeleno programer piše najjednostavniji mogući kod koji će kao rezultat imati prolazak testa napisanog u prethodnoj fazi Crveno. Upravo zbog tog prolaska testa ova faza TDD-a ima naziv Zeleno.

Tablica 4: Uzorci za fazu Zeleno

Naziv uzorka	**Opis uzorka**
Lažiranje (engl. *Fake It*)	Najjednostavniji način da napisani test prođe je da metoda koju test poziva vraća konstantu. Nakon što uz pomoć konstante test prođe, postupno možemo konstantu zamjenjivati varijablama kako bi test imao smisla. Ovakva metoda postizanja prolaska testa mnogima nema smisla jer konstantu koju prvotno napišemo u svakom slučaju moramo zamijeniti u fazi refaktoriranja. No ovakav pristup je poželjan jer je cilj TDD-a što brži prolazak testa. [2]
Trianguliranje (engl. *Triangulate*)	Prema metodi trianguliranja apstrakcija metode potrebna je tek kada imamo dva ili više primjera te metode. Prije toga metoda može vraćati konstantu koja odgovara rezultatu koji se očekuje ako koristimo

	argumente jedinog primjera. Trianguliranje može biti vrlo korisno kod implementiranja složenih funkcija za koje nam nije lako definirati odgovarajuću programsku logiku. [2]
Očita implementacija (engl. *Obvious Imlementation*)	Pri implementiranju funkcija koje su vrlo jednostavne nema potrebe za korištenjem metoda poput lažiranja ili trianguliranja, već možemo odmah implementirati funkciju. Na primjer, funkciju koja množi dva broja programer može implementirati na očiti način. U slučaju da se ipak dogodi da test koji provjerava tu funkciju ne prođe, onda se implementacija te funkcije dijeli na manje cjeline. [2]
Jedan prema više (engl. *One To Many*)	Funkcije koje koriste skupove objekata najbolje je prvo implementirati tako da rade samo sa jednim objektom, a zatim kroz refaktoriranje postići da funkcija radi i sa skupom objekata. Uzmimo za primjer funkciju koja množi sve članove polja cijelih brojeva. Za početak, umjesto računanja umnoška brojeva koji se nalaze u polju, koristimo konstantu. Refaktoriranjem tu konstantu postupno zamjenjujemo funkcijom koja zaista računa umnožak članova polja. [2]

3.5. Uzorci za fazu Refaktoriranja

Faza refaktoriranja je treća od tri osnovne faze TDD-a u kojoj se prethodno napisani kod prepravlja kako bi program uz što manje koda radio ono za što je predviđen. Kako bi refaktoriranje bilo što kvalitetnije poželjno je pratiti uzorke razvoja za fazu refaktoriranja, a oni su predmet proučavanja ovog poglavlja.

Tablica 5: Uzorci za fazu Refaktoriranja

Naziv uzorka	**Opis uzorka**
Uklanjanje razlika (engl. *Reconcile Differences*)	Više dijelova koda koji su međusobno slični možemo postupno refaktorirati s krajnjim ciljem potpunog uklanjanja razlika između njih. Takvo refaktoriranje često može biti vrlo zahtjevno i lako može doći do neželjene promjene u funkcioniranju softvera ukoliko nismo dovoljno oprezni pri implementiranju promjena. Refaktoriranjem u manjim koracima i s konkretnim povratnim informacijama može se smanjiti rizik od neželjenih posljedica pri mijenjanju koda. [2]

Izoliranje promjene (engl. *Isolate Change*)	Najbolji način da izmijenimo metodu ili objekt koji se sastoji od više dijelova je da izoliramo dio koji planiramo mijenjati. Na taj način naše izmjene neće utjecati na ostale dijelove metode ili objekta, što će nam omogućiti da se koncentriramo na kod koji želimo refaktorirati. Neki od načina izoliranja promjene koda su izdvajanje metode, izdvajanje objekta i kreiranje objekta od metode. [2]
Izdvajanje metode (engl. Extract Method)	Metode koje su sadržavaju mnogo koda mogu biti prekomplicirane i teške za čitati. Taj se problem može riješiti izdavanjem manjeg dijela te metode u zasebnu metodu, i njezinim pozivanjem na mjestu koda kojeg smo izdvojili. Time sprječavamo dupliciranje koda i olakšavamo njegovo čitanje i razumijevanje . Ipak, treba imati na umu da sa izdvajanjem metoda ne treba pretjerivati jer tako možemo postići efekt suprotan očekivanome. [2]
Smanjivanje pozivanih metoda (engl. *Inline Method*)	Ponekad u metodama pozivamo mnogo drugih metoda, što otežava razumijevanje koda osobi koja ga proučava. Ukoliko uočimo takvu „pretrpanu“ metodu, možemo ju učiniti razumljivijom tako da zamijenimo kod koji poziva drugu metodu kodom koji ta druga metoda sadrži. Nakon toga je potrebno obratiti pažnju na parametre i varijable metoda, i prepraviti ih tako da kod funkcionira kao i prije promjene. [2]
Kreiranje sučelja (engl. *Extract Interface*)	U slučaju da imamo više klasa koje su slične u smislu da imaju metode istog imena koje funkcioniraju na sličan način, možemo kreirati sučelje koje će te klase nasljeđivati. Sučelje će sadržavati metode zajedničke klasama koje će nasljeđivati sučelje, dok će specifične metode ostati u svojim klasama. [3]
Premještanje metoda (engl. *Move Method*)	Ukoliko uočimo da određena metoda ne pripada u klasu u kojoj se nalazi u tom trenutku, možemo ju premjestiti u drugu klasu i pozvati ju iz originalne. Cijeli taj proces obično je brz i bez rizika od prevelikih komplikacija, a glavna prednost mu je veća preglednost i mogućnost razumijevanja koda. [2]
Kreiranje objekta iz metode (engl. *Method Object*)	Složenu metodu koja koristi više parametara i lokalnih varijabli možemo pretvoriti u zasebni objekt. Ovaj način refaktoriranja koda koristan je kao priprema za dodavanje nove programske logike sustavu. Osim toga, kreiranje objekta iz metode izvrstan je način

	pojednostavljenja koda u slučajevima kada izdvajanje metode nije moguće. [2]
Dodavanje parametra metodi (engl. *Add Parameter*)	Ponekad shvatimo da metodi koju smo već kreirali moramo dodati još jedan parametar. Primjerice, prvi test metode smo prošli bez da smo trebali taj parametar, ali pod novim okolnostima moramo u obzir uzeti više podataka kako bi metoda funkcionirala kako želimo. [2]
Premještanje parametra u konstruktor (engl. *Method Parameter to Constructor Parameter*)	Ako isti parametar prosljeđujemo u više metoda istog objekta, kod možemo pojednostaviti tako da taj parametar proslijedimo konstruktoru objekta. U slučaju da se varijabla instance objekta koristi samo u jednoj metodi, možemo primijeniti i suprotno, to jest premještanje varijable u parametar metode. [2]

4. Primjer razvoja jednostavne aplikacije upotrebom TDD-a

Kroz ovaj jednostavni primjer prikazati ću kako izgleda razvoj softvera vođen testiranjem. Za primjer sam odabrao konzolnu aplikaciju koja pretvara upisani arapski u rimski broj. Problem pretvaranja arapskih u rimske brojeve jedna je od kata programiranja, a to je naziv za česte probleme koji se koriste za učenje programiranja.

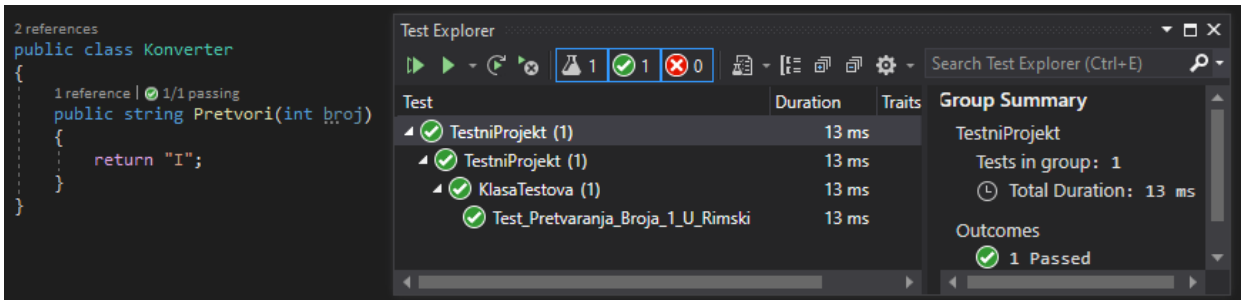
4.1. Prvi Test

Prije pisanja ikakvog koda aplikacije napisao sam test koji ima zadatak provjeriti hoće li metoda uspjeti pretvoriti broj „1“ u „I“. Takav test odgovara praksi „Testa u jednom koraku“ jer ću kroz njegovu implementaciju nešto naučiti, a mogu biti siguran da ću ga znati implementirati. Taj dio razvoja softvera spada pod prvu fazu „Crveno“ jer nakon što ga pokrenemo, vidjet ćemo da test ne prolazi, to jest, ne može se ni pokrenuti jer klasa i metoda koju test koristi u tom trenutku ne postoje.

```
[TestMethod]
0 references
public void Test_Pretvaranja_Broja_1_U_Rimski()
{
    Konverter konverter = new Konverter();
    string rimski = konverter.Pretvori(1);
    Assert.AreEqual(rimski, "I");
}
```

Slika 3: Test pretvaranja broja 1 u rimski broj

Stoga krećemo na sljedeću fazu u kojoj ćemo implementirati klasu *Konverter* i metodu *Pretvori()*. Ovdje sam se poslužio uzorkom za fazu „Zeleno“ koji se naziva „Lažiranje“, kod kojeg je karakteristično da testirana metoda vraća konstantu koja zadovoljava test.



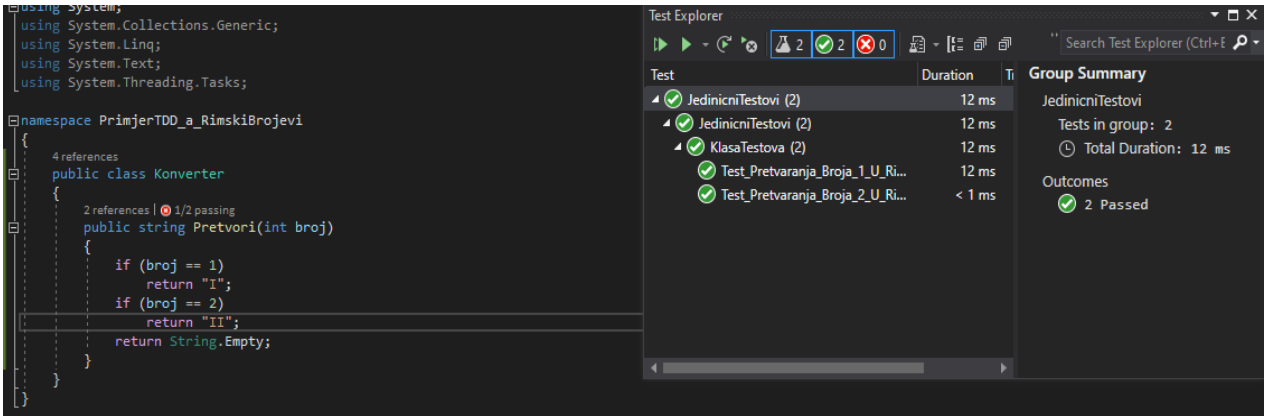
Slika 4: Implementacija klase i metode

U ovom slučaju to bi značilo da metoda vraća „I“. Kao što se može vidjeti na slici 4, rezultat toga je da napisani test prolazi, na temelju čega se da zaključiti da smo uspješno izvršili fazu „Zeleno“.

Uzevši u obzir da je kod metode *Pretvori()* trivijalan, posljednja faza „Refaktoriranje“ nije potrebna, stoga možemo krenuti sa sljedećim testom.

4.2. Drugi test

Nakon implementacije prvog testa krećemo sa sljedećim testom, čime opet dolazimo do faze „Crveno“. Novi test sličan je prošlom, ali je testni podatak 2, a ne 1 kao u prethodnom testu. Pokrenuvši testove, možemo vidjeti da novi test ne prolazi., što je karakteristika faze „Crveno“ Nakon toga krećemo na fazu „Zeleno“ u kojoj ćemo koristiti uzorak „Trianguliranje“ prema kojemu je apstrakcija metoda potrebna tek ako imamo dva primjera njezina korištenja, odnosno testiranja. Prolazak testova postići ćemo izmjenjivanjem metoda na način da provjerava je li proslijeđeni broj jednak broju 1, i ako jeste ispisati će „I“, dok će u slučaju proslijeđenog broja 2 ispisati „II“.



Slika 5: Implementacija drugog testa

Kao što je vidljivo na slici 5, nakon toga oba testa prolaze. Time završava faza „Zeleno“, što znači da možemo prijeći na sljedeću fazu; „Refaktoriranje“ To ću učiniti uz pomoć *for* petlje koja će ispisivati broj rimskih znamenki „I“ koji je u jednak broju prosljeđenom metodi.

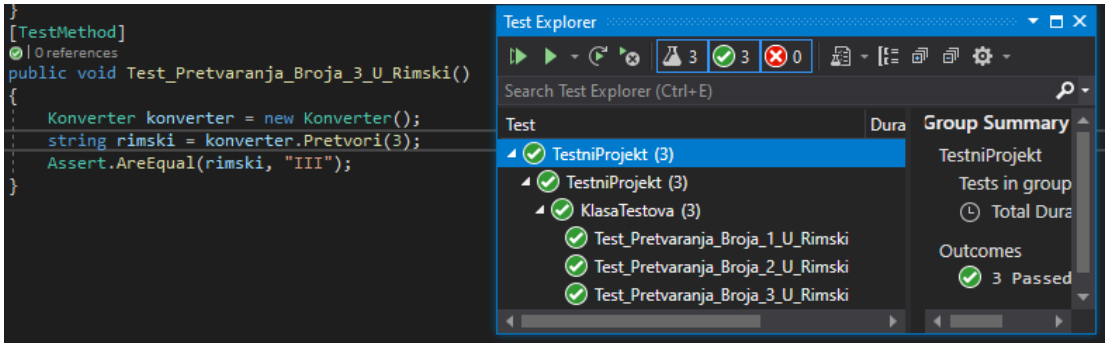
```
public string Pretvori(int broj)
{
    string rezultat = String.Empty;
    for (int i = broj; i > 0; i--)
    {
        rezultat += "I";
    }
    return rezultat;
}
```

Slika 6: Refaktoriranje metode

Pokretanjem testova možemo se uvjeriti da je refaktoriranje zadovoljavajuće, što znači da možemo krenuti sa sljedećim testom.

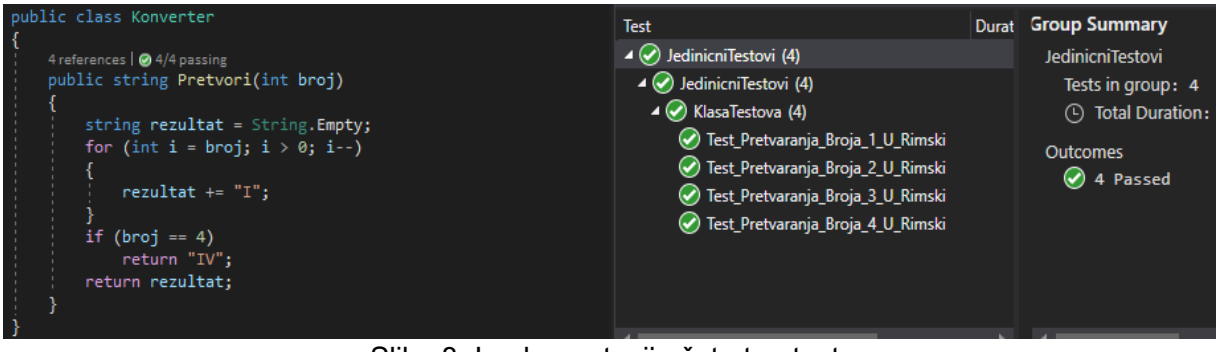
4.3. Treći i četvrti test

Nakon toga pišemo test koji provjerava hoće li prosljeđivanje broja 3 metodi rezultirati odgovarajućim rimskim brojem, a test prolazi pri prvom pokretanju, čime smo preskočili fazu „Crveno“ i stigli do faze „Zeleno“. S obzirom da je implementacija metoda poprilično jednostavna, a zadovoljava trenutne testove, fazu „Refaktoriranje“ možemo preskočiti.



Slika 7: Treći test

Stoga možemo krenuti sa sljedećim testom koji provjerava vraća li testni podatak 4 rimski broj „IV“. Pokretanjem testova uočavamo da posljednji test ne prolazi, što nam sugerira da se nalazimo u fazi „Crveno“. Neprolazak testa riješio sam dodavanjem uvjeta u metodu koji provjerava je li argument proslijeđen metodi jednak broju 4 i zatim vraća odgovarajući rimski broj, što je još jedna verzija uzorka „Lažiranje“. Pokretanjem testova možemo vidjeti da svi prolaze što znači da se nalazimo u fazi „Zeleno“. Ovo rješenje nije previše praktično jer bi ovim načinom morali specificirati rezultat za svaki pojedini broj, no njime smo postigli brzi prolazak testa što je cilj TDD-a.



Slika 8: Implementacija četvrtog testa

Nakon prolaska testa možemo krenuti s refaktoriranjem koda. Uvođenjem drugih rimskih znamenki poput „V“ i „X“ problem postaje dosta složeniji nego što je bio do sada, što sam odlučio riješiti uvođenjem nove klase *RimskiBrojevi*, u kojoj se svaki objekt sastoji od

arapskog broja tipa *int* i odgovarajuće rimske znamenke tipa *string.* To se može vidjeti na slici 9.

```
public class RimskiBrojevi
{
    1 reference
    public int ArapskiBroj { get; set; }
    1 reference
    public string RimskiBroj { get; set; }

    0 references
    public RimskiBrojevi(int arapski, string rimski)
    {
        ArapskiBroj = arapski;
        RimskiBroj = rimski;
    }
}
```

Slika 9: Klasa RimskiBrojevi

U klasi *Konverter* metodi *Pretvori()* sam dodao listu tipa *RimskiBrojevi* kojoj dodajem parove arapskih i odgovarajućih rimskih znamenki. Metodu *Pretvori()* zatim sam dodatno refaktorirao tako da za svaki objekt te liste provjerava je li njegov arapski broj jednak proslijeđenom argumentu, te ako je da ispisuje odgovarajući rimski broj.

```
4 references | 4/4 passing
public string Pretvori(int broj)
{
    List<RimskiBrojevi> listaBrojeva = new List<RimskiBrojevi>();
    listaBrojeva.Add(new RimskiBrojevi(4, "IV"));
    listaBrojeva.Add(new RimskiBrojevi(3, "III"));
    listaBrojeva.Add(new RimskiBrojevi(2, "II"));
    listaBrojeva.Add(new RimskiBrojevi(1, "I"));

    foreach (var item in listaBrojeva)
    {
        if (item.ArapskiBroj == broj)
            return item.RimskiBroj;
    }
    return String.Empty;
}
```

Slika 10: Refaktoriranje metode Pretvori()

Pokretanje testova uočio sam da svi testovi ponovno prolaze što sugerira da je ova faza refaktoriranja bila uspješna. Ovakav algoritam još uvijek bi zahtijevao postojanje svakog pojedinog broja u listi, no jedno od temeljnih pravila TDD-a zahtijeva da se ciklus razvoja softvera od tri faze odvija bez dužih pauzi, stoga sam taj problem odlučio riješiti nakon dodavanja novog testa.

4.4. Peti test

Novi ciklus TDD-a započeo sam kreiranjem testa koji provjerava koji je rimski ekvivalent broju 6. Pokretanjem testa vidimo da novi test ne prolazi što nam očigledno govori da smo u fazi „Crveno“. Najbrži put do prolaska testa je dodavanje para arapskog i rimskog broja 6, čime smo došli do faze „Zeleno“, nakon čega možemo krenuti na refaktoriranje.

Kako bismo dobili algoritam koji točno određuje rimske brojeve, u već postojeću petlju koja prolazi kroz sve objekte liste dodat ću *for* petlju koja za zadatak ima naći prvu rimsku znamenku u listi koja je manja ili jednaka proslijeđenom argumentu. Tu znamenku će zapisati, a brojač *for* petlje umanjiti za arapski broj jednak upisanoj rimskoj znamenki. Postupak će se ponavljati dok god postoje brojevi u listi koji su manji ili jednaki od brojača petlje.

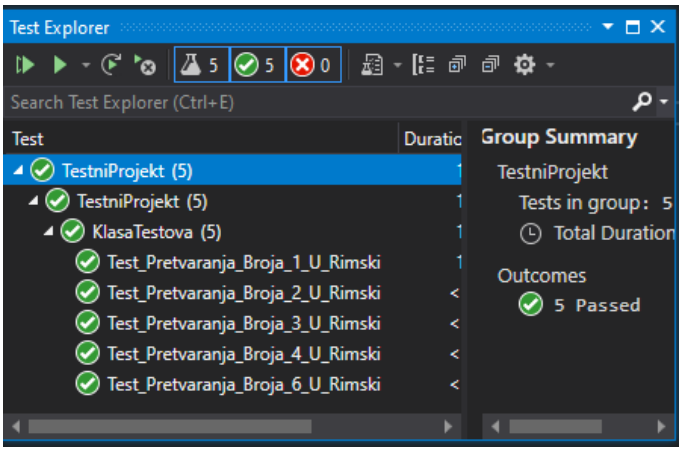
```
string rezultat = string.Empty;

foreach (var item in listaBrojeva)
{
    for (int i = broj; i >= item.ArapskiBroj; i -= item.ArapskiBroj)
    {
        rezultat += item.RimskiBroj;
        if (rezultat != "")
            broj -= item.ArapskiBroj;
    }
}
return rezultat;
```

Slika 11: Algoritam pretvaranja arapskih u rimske brojeve

Kako bi metoda funkcionirala kako je zamišljeno važno je da brojevi u listi budu sortirani silazno. S obzirom da posljednji test provjerava rimski zapis broja 6, na vrh liste moramo dodati broj 5. Brojeve 2 i 3 sam izbrisao iz liste jer su suvišni, no broj 4 sam ostavio zbog specifičnog načina računanja kod rimskih brojeva gdje manja znamenka ispred veće označava umanjivanje te veće znamenke za vrijednost manje znamenke, što bi algoritmu u metodi *Pretvori()* stvaralo velike probleme. Nakon što smo obavili ove izmjene možemo vidjeti da svi testovi prolaze, što znači da smo uspješno obavili ovo refaktoriranje.

```
List<RimskiBrojevi> listaBrojeva = new List<RimskiBrojevi>();
listaBrojeva.Add(new RimskiBrojevi(5, "V"));
listaBrojeva.Add(new RimskiBrojevi(4, "IV"));
listaBrojeva.Add(new RimskiBrojevi(1, "I"));
```

Slika 12: Ažurirana listaBrojeva



Slika 13: Status testova nakon refaktoriranja

4.5. Šesti test

Nakon implementacije testa broja 6 ponovno krećemo s ciklusom TDD-a pisajući test za broj 19..

```
[TestMethod]
0 references
public void Test_Pretvaranja_Broja_19_U_Rimski()
{
    Konverter konverter = new Konverter();
    string rimski = konverter.Pretvori(19);
    Assert.AreEqual(rimski, "XIX");
}
```

Slika 14: Test broja 19

Nakon što pokrenemo testove, nalazimo se u fazi „Crveno“ jer vidimo da tek dodani test ne prolazi, što je logično s obzirom da se sastoji od većih znamenaka koje lista brojeva ne sadrži. Kako bi test prošao dovoljno je ažurirati listu dodavanjem tih znamenaka.

```
List<RimskiBrojevi> listaBrojeva = new List<RimskiBrojevi>();
listaBrojeva.Add(new RimskiBrojevi(10, "X"));
listaBrojeva.Add(new RimskiBrojevi(9, "IX"));
listaBrojeva.Add(new RimskiBrojevi(5, "V"));
listaBrojeva.Add(new RimskiBrojevi(4, "IV"));
listaBrojeva.Add(new RimskiBrojevi(1, "I"));
```

Slika 15: Ažurirana listaBrojeva

Ažuriranjem liste postižemo prolazak testova što znači da smo u fazi „Zeleno“ i možemo krenuti na fazu refaktoriranja. S obzirom da listaBrojeva postaje sve veća, a time i metoda *Pretvori()* poželjno bi bilo kreiranje i popunjavanje liste izdvojiti u zasebnu metodu, što je u skladu sa uzorkom za fazu Refaktoriranja „Izdvajanje metode“. U skladu s time kreirao sam metodu imena *PopuniListu()*, a na mjestu gdje je lista prethodno bila kreirana i popunjena sam pozvao tu metodu. To se može vidjeti na slikama 16 i 17.

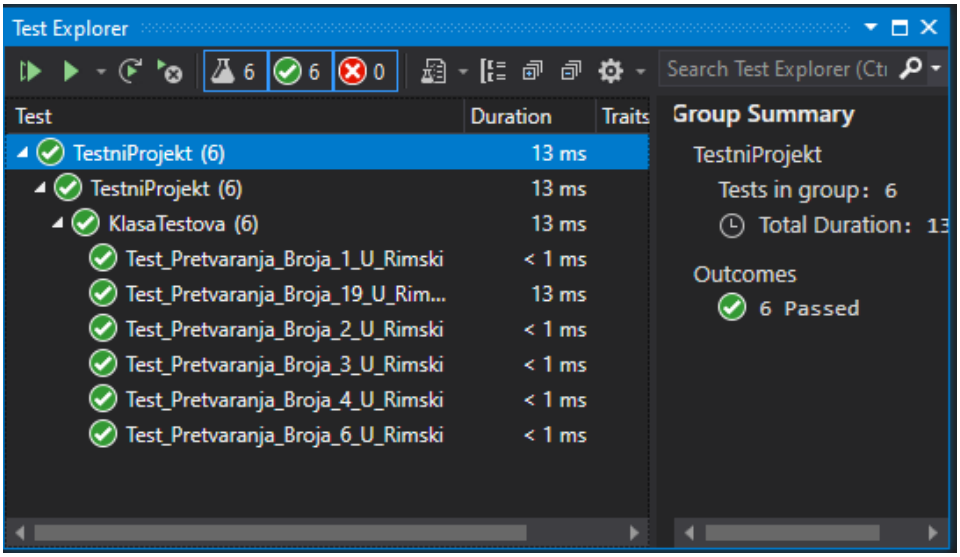
```
List<RimskiBrojevi> PopuniListu()
{
    List<RimskiBrojevi> listaBrojeva = new List<RimskiBrojevi>();

    listaBrojeva.Add(new RimskiBrojevi(10, "X"));
    listaBrojeva.Add(new RimskiBrojevi(9, "IX"));
    listaBrojeva.Add(new RimskiBrojevi(5, "V"));
    listaBrojeva.Add(new RimskiBrojevi(4, "IV"));
    listaBrojeva.Add(new RimskiBrojevi(1, "I"));

    return listaBrojeva;
}
```

Slika 16: Metoda *PopuniListu()*

```
public string Pretvori(int broj)
{
    List<RimskiBrojevi> listaBrojeva = PopuniListu();
    string rezultat = string.Empty;

    foreach (var item in listaBrojeva)
    {
        for (int i = broj; i >= item.ArapskiBroj; i -= item.ArapskiBroj)
        {
            rezultat += item.RimskiBroj;
            if (rezultat != "")
                broj -= item.ArapskiBroj;
        }
    }
    return rezultat;
}
```

Slika 17: Finalni izgled metode *Pretvori()*

Nakon napravljenih izmjena u fazi „Refaktoriranje“, ponovnim pokretanjem testova možemo se uvjeriti da nismo promijenili način funkcioniranja koda.



Slika 18: Status prvih 6 testova nakon refaktoriranja

4.6. Sedmi i osmi test

Kako bih se uvjerio da aplikacija radi kako je zamišljeno, dodao sam još nekoliko testova koji provjeravaju rimske verzije nešto većih brojeva. Počeo sam s testom broja 448, koji nije prošao nakon što sam ga napisao i pokrenuo, što ukazuje da smo u fazi „Crveno“. Kako bi proradio, dovoljno je u metodi PopuniListu() dodati nekoliko većih znamenki koje nismo prethodno koristili. Nakon toga test prolazi, što nam sugerira da smo u fazi „Zeleno“. Postupak se ponavlja i sa dodavanjem testa broja 1997. Treba napomenuti da kod implementiranja ovih testova, osim okruglih brojeva, u listu dodajemo i brojeve kao što su 40 („XL)“ i 90 („XC“) iz istih razloga zbog kojih smo dodali broj 4 („IV“).

```
[TestMethod]
0 references
public void Test_Pretvaranja_Broja_448_U_Rimski()
{
    Konverter konverter = new Konverter();
    string rimski = konverter.Pretvori(448);
    Assert.AreEqual(rimski, "CDXLVIII");
}

[TestMethod]
0 references
public void Test_Pretvaranja_Broja_1997_U_Rimski()
{
    Konverter konverter = new Konverter();
    string rimski = konverter.Pretvori(1997);
    Assert.AreEqual(rimski, "MCMXCVII");
}
```

Slika 19: Testovi brojeva 448 i 1997

```
List<RimskiBrojevi> PopuniListu()
{
    List<RimskiBrojevi> listaBrojeva = new List<RimskiBrojevi>();
    listaBrojeva.Add(new RimskiBrojevi(1000, "M"));
    listaBrojeva.Add(new RimskiBrojevi(900, "CM"));
    listaBrojeva.Add(new RimskiBrojevi(500, "D"));
    listaBrojeva.Add(new RimskiBrojevi(400, "CD"));
    listaBrojeva.Add(new RimskiBrojevi(100, "C"));
    listaBrojeva.Add(new RimskiBrojevi(90, "XC"));
    listaBrojeva.Add(new RimskiBrojevi(50, "L"));
    listaBrojeva.Add(new RimskiBrojevi(40, "XL"));
    listaBrojeva.Add(new RimskiBrojevi(10, "X"));
    listaBrojeva.Add(new RimskiBrojevi(9, "IX"));
    listaBrojeva.Add(new RimskiBrojevi(5, "V"));
    listaBrojeva.Add(new RimskiBrojevi(4, "IV"));
    listaBrojeva.Add(new RimskiBrojevi(1, "I"));

    return listaBrojeva;
}
```

Slika 20: Finalni oblik metode *PopuniListu()*

Slika 19 prikazuje posljednja dva napisana testa, a slika 20 ažuriranu metodu *PopuniListu()*. Implementacijom posljednjeg testa listu brojeva popunio sam svim znamenkama koje su Rimljani koristili. Mogao bih napisati još testova kako bih se još više uvjerio da nema broja kojeg ovaj algoritam ne može točno pretvoriti u rimski broj, no u skladu s općenitim uzorkom razvoja „Testni podaci“, koji sugerira da nije poželjno koristiti mnogo podataka u testovima osim ako razlika među njima nije smisleno značajna, to neću činiti.

4.7. Glavni dio programa

Ovakav, temeljito testiran kod, možemo implementirati u glavnom dijelu programa bez ikakvog straha, jer znamo da funkcionira onako kako smo zamislili. Ovaj dio programa spada pod kod koji se teško može testirati, što u skladu s dobrim uzorcima TDD-a treba izbjegavati. Ipak, uzevši u obzir jednostavnost ovog dijela programa, smatram da njegov razvoj bez korištena TDD-a neće predstavljati velik problem. U glavnom dijelu programa kreirat ćemo jednostavno sučelje u kojem korisnik može upisati broj, a aplikacija zatim ispisuje taj broj rimskim znamenkama.

```
namespace PrimjerTDD_a_RimskiBrojevi
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Unesite broj koji želite pretvoriti u rimski: ");
            int arapskiBroj = int.Parse(Console.ReadLine());
            Konverter konverter = new Konverter();
            string rimskiBroj = konverter.Pretvori(arapskiBroj);
            Console.WriteLine(rimskiBroj);
            Console.Read();
        }
    }
}
```

Slika 21: Glavni dio programa

```
Unesite broj koji zelite pretvoriti u rimski:
572
DLXXII
```

Slika 22: Prvi primjer

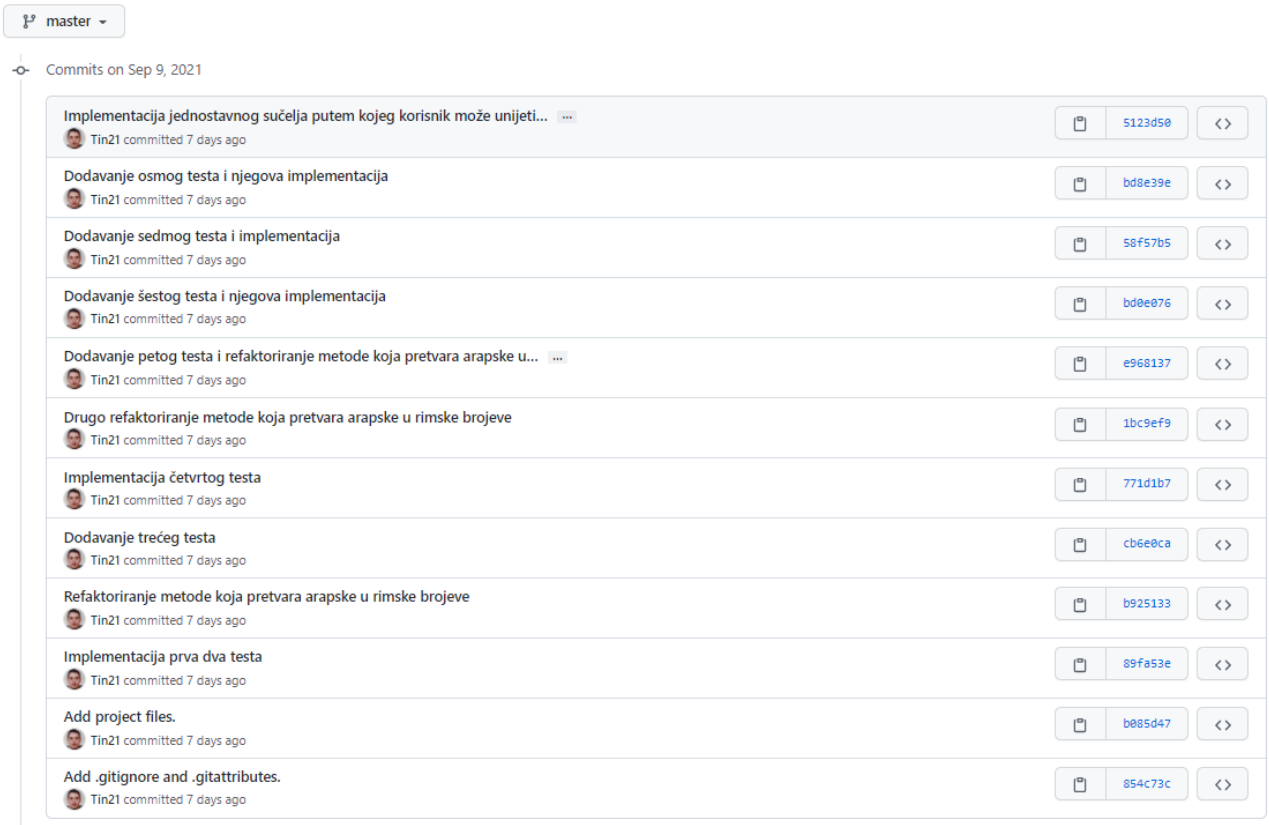
```
Unesite broj koji zelite pretvoriti u rimski:
2498
MMCDXCVIII
```

Slika 23: Drugi primjer

Na slikama 22 i 23 možemo vidjeti kako izgleda aplikacija. Postupak razvoja ove jednostavne aplikacije može se vidjeti u sustavu za verzioniranje GitHub, klikom na sljedeću poveznicu: https://github.com/Tin21/TDD_Primjer_Rimski_Brojevi

GitHub sam koristio tako da sam nakon svakog temeljnog ciklusa TDD-a, to jest svake implementacije određenog testa, ažurirao verziju softvera u sustavu za verzioniranje. Pri tome

sam svaki *commit* imenovao na način da bude što jasnije koje su promjene u njemu napravljene u usporedbi sa prethodnom verzijom, kao što se može vidjeti na slici 24.



Slika 24: Dinamika razvoja u GitHub-u

5. Zaključak

Razvoj softvera vođen testiranjem nastao je u sklopu ekstremnog programiranja, no s vremenom postaje sve popularnija zasebna metoda razvoja. Temeljna pretpostavka TDD-a je da prije pisanja ikakvog koda programa, napišemo test koji taj kod provjerava. Programeru koji se nikad nije susreo s takvim načinom razvoja to može zvučati pomalo radikalno, a prilagodba TDD-u može biti izazovna jer zahtijeva mnogo vježbe i upornosti.

Ipak, savladavanje TDD-a se često isplati jer rezultira sposobnošću programera da razvija vrlo kvalitetan kod. Neke od najznačajnijih kvaliteta su manji broj grešaka u softveru, veća modularnost koda, lakše izmjenjivanje dijelova koda i veće samopouzdanje programera.

Prije pisanja ovog rada i sam sam bio pomalo skeptičan prema ovoj metodi razvoja zbog činjenice da se prvo pišu testovi, a zatim odgovarajući kod koji ih zadovoljava. Pitao sam se je li zaista potrebno testirati svaki dio koda te nije li takav pristup previše dugotrajan. Ipak, nakon proučavanja literature te izrade teoretskog i praktičnog dijela rada shvatio sam koliko prednosti ovaj način razvoja koda ima. Potpuna ili gotovo potpuna pokrivenost koda testovima ima mnoge prednosti, što postaje sve jasnije kako količina koda raste. Pisanje velikog broja testova možda oduzima nešto više vremena, ali s druge strane može nam uštediti mnogo više vremena koje bi izgubili tražeći i ispravljajući greške u kodu koje bi kod TDD-a mnogo brže uočili.

Za potpuno svladavanje TDD-a, kao i kod svake druge vještine, vjerojatno su potrebne godine truda. Na tom putu od velike pomoći mogu biti uzorci i prakse dobrog razvoja. Njihovo poznavanje može znatno umanjiti vrijeme potrebno za razvoj određenog softvera kao i broj grešaka pri razvoju.

Smatram da je učenje ove metode razvoja i njezina primjena vrijedna truda, no to je nešto što svaki programer treba odlučiti za sebe. Jedan od razloga zbog kojeg to mislim je i činjenica da popularnost TDD-a s vremenom sve više raste. Zbog mnogih prednosti u usporedbi s tradicionalnim načinom razvoja softvera, TDD je sve zastupljeniji, ponajviše pri razvoju manjih projekata, ali raste mu popularnost i kod razvoja većih projekata.

Smatram da će TDD u budućnosti postati još više prihvaćen, što bi moglo rezultirati razvojem novih alata koji će dodatno olakšati razvoj softvera vođen testiranjem. Zbog svega navedenog TDD bi u budućnosti mogao postati metoda razvoja koju će preferirati velik dio programera.

Popis literature

[1] „What is Test Driven development (TDD)? Tutorial with example“, (bez dat.) [Na internetu]. Dostupno: https://www.guru99.com/test-driven-development.html [Pristupano 8.7.2021.]

[2] K. Beck, „Test-Driven Development By Example“: Three Rivers Institute, 2002.

[3] H. Bedeković, „Razvoj softvera vođen testiranjem“, Zagreb: Prirodoslovno-matematički fakultet, 2016. [Na internetu]. Dostupno: https://repozitorij.pmf.unizg.hr/islandora/object/pmf%3A77/datastream/PDF/view [Pristupano 30.8. 2021.]

[4] M. Bagić Babac, M Kušek, „Skripta: Testiranjem upravljano programiranje“, Zagreb: Fakultet elektrotehnike i računarstva, 2009. [Na internetu]. Dostupno: https://www.fer.unizg.hr/_download/repository/ILJ-skripta-testiranje[2].pdf [Pristupano 30.8.2021.]

[5] D. Barber, „Why Test-driven Development?“, 2012. [Na internetu]. Dostupno: http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/ [Pristupano 11.7.2021.]

[6] „What is unit testing?“ (bez dat.) [Na internetu]. Dostupno: https://smartbear.com/learn/automated-testing/what-is-unit-testing/ [Pristupano 12.7.2021.]

[7] „Testiranje programa“, Varaždin: Fakultet organizacije i informatike (bez dat.) [Na internetu]. Dostupno: https://elfarchive1920.foi.hr/pluginfile.php/108225/mod_resource/content/5/Testiranje.pdf [Pristupano 30.8.2021.]

[8] E. Miquelito, „What is Test-Driven Development? (And How To Get It Right)“, 2020. [Na internetu]. Dostupno: https://www.squash.io/what-is-test-driven-development-and-how-to-get-it-right/ [Pristupano 15.7.2021.]

[9] S. Ivičević, „TDD vs. BDD vs. ATDD“, 2021. [Na internetu]. Dostupno: https://serengetitech.com/tech/tdd-vs-bdd-vs-atdd/ [Pristupano 17.7.2021.]

[10] A. Ghahrai, „Pros and Cons of Test Driven Development“, 2017. [Na internetu]. Dostupno: https://devqa.io/pros-cons-test-driven-development/ [Pristupano 7.8.2021.]

[11] G. Meszaros, „XUnit Test Patterns: Refactoring Test Code“, Addison – Wesley Professional, 2007.

[12] „What is Regression Testing“ (bez dat.) [Na internetu]. Dostupno: https://smartbear.com/learn/automated-testing/what-is-regression-testing/ [Pristupano 24.8.2021.]

Popis slika

Popis tablica