# Razvoj softvera vođen testiranjem

**Bedeković, Hrvoje**

**Master's thesis / Diplomski rad**

**2016**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:217:389505

*Rights / Prava:* In copyright / Zaštićeno autorskim pravom.

*Download date / Datum preuzimanja:* **2024-03-29**




*Repository / Repozitorij:*

Repository of the Faculty of Science - University of Zagreb









**SVEUČILIŠTE U ZAGREBU**

**PRIRODOSLOVNO – MATEMATIČKI FAKULTET**

**MATEMATIČKI ODSJEK**

Hrvoje Bedeković

# RAZVOJ SOFTVERA VOĐEN TESTIRANJEM

Diplomski rad

Voditelj rada:

Robert Manger

Zagreb, 2016

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. _____

*Zahvaljujem se svojim roditeljima na pruženoj podršci kroz cijeli studij i prof. dr. sc. Robertu Mangeru na pomoći i savjetima za izradu ovog diplomskog rada.*

# Sadržaj

PREDGOVOR ..... 1

1. METODE RAZVOJA SOFTVERA ..... 3

1.1. KLASIČNE METODE ..... 3

1.1.1. PRIMJER KLASIČNE METODE – *Unified Process* ..... 3

1.2. AGILNE METODE ..... 5

1.2.1. PRIMJER AGILNE METODE - *Extreme programming* ..... 6

2. OPĆENITO O RAZVOJU SOFTVERA VOĐENOG TESTIRANJEM (*eng. Test Driven Development - TDD*) ..... 7

2.1. UVOD U RAZVOJ SOFTVERA VOĐEN TESTIRANJEM ..... 7

2.2. OBLIKOVANJE SOFTVERA ..... 7

2.3. RAZVOJ SOFTVERA ..... 7

2.4. *RED-GREEN-REFACTOR* ..... 8

2.4.1. TRI FAZE SOFTVERA VOĐENOG TESTIRANJEM ..... 8

2.4.2. PRVA FAZA (eng. Red Phase) ..... 8

2.4.3. DRUGA FAZA (eng. Green Phase) ..... 9

2.4.4. TREĆA FAZA – REFAKTORIZACIJA ..... 10

2.5. PREDNOSTI RAZVOJA SOFTVERA VOĐENOG TESTIRANJEM ..... 10

2.6. JEDNOSTAVAN PRIMJER RAZVOJA SOFTVERA VOĐENOG TESTIRANJEM ..... 11

3. TESTIRANJE DIJELOVA (*eng. Unit Testing*) ..... 13

3.1. UVOD U TESTIRANJE DIJELOVA ..... 13

3.2. APLIKACIJSKI OKVIRI (*eng. Framework* ) ZA TESTIRANJE DIJELOVA.... 13

3.3. ASSERT.... 15

3.4. METODE KOJE SADRŽI *UNITTEST* APLIKACIJSKI OKVIR .... 15

3.5. IZNIMKE ( eng. *Exceptions* ).... 16

4. OBRASCI ZA RAZVOJ SOFTVERA VOĐENOG TESTIRANJEM .... 20

4.1. TEST.... 20

4.2. IZOLIRANI TEST.... 20

4.3. LISTA TESTOVA.... 20

4.4. PRVO TVRDNJE.... 21

4.5. PODATCI ZA TESTIRANJE .... 22

5. OBRASCI ZA PRVU FAZU (eng. Red Phase Patterns).... 23

5.1. IZBOR TESTA.... 23

5.2. POČETNI TEST.... 23

5.3. OBJAŠNJAVAJUĆI TESTOVI.... 24

5.4. TEST ZA UČENJE.... 24

5.5. TESTOVI POGREŠAKA.... 25

6. OBRASCI ZA TESTIRANJE .... 26

6.1. MANJI TEST.... 26

6.2. OPONAŠAJUĆI OBJEKT (eng. Mock Object).... 26

6.2.1. LUTKA.... 27

6.2.2. LAŽNI OBJEKT.... 28

6.2.3. OPONAŠAJUĆI OBJEKT ..... 28
6.3. ZAPISI U STRINGU ..... 30
6.4. TESTIRANJE PROBLEMATIČNOG KODA ..... 31
6.5. SAVJETI ZA PROGRAMERE ..... 32
7. OBRASCI ZA DRUGU FAZU ( eng. *Green Bar Patterns* ) ..... 33
7.1. LAŽIRANJE ..... 33
7.2. TRIANGULIRANJE (eng. *Triangulate*) ..... 34
7.3. OČITA IMPLEMENTACIJA ..... 35
7.4. KOLEKCIJA OBJEKATA ..... 36
8. OBRASCI ZA TREĆU FAZU – REFAKTORIZACIJA ..... 37
8.1. UJEDINJENJE KODA ..... 37
8.2. IZOLIRANA PROMJENA ..... 37
8.3. SMANJENJE METODA ..... 38
8.4. LINIJSKA METODA ..... 38
8.5. DODAVANJE SUČELJA *(eng. Interface)* ..... 39
8.6. PREMJEŠTANJE METODA NA PRAVO MJESTO ..... 39
8.7. PRETVARANJE METODA U OBJEKT ..... 40
8.8. DODAVANJE I PREMJEŠTANJE PARAMETARA ..... 40
9. STUDIJSKI PRIMJER ..... 42
ZAKLJUČAK ..... 55
LITERATURA ..... 56

SAŽETAK ..... 57

SUMMARY ..... 58

ŽIVOTOPIS ..... 59

# PREDGOVOR

Tokom zadnjih 50 godina metode razvoja softvera se stalno usavršavaju kako bi se mogao razviti što bolji i kvalitetniji softver u što kraćem roku. Danas se metode razvoja softvera mogu podijeliti u klasične i agilne metode. U ovome radu opisujemo metodu razvoja softvera vođenog testiranjem koja spada pod agilne metode.

Razvoj softvera vođen testiranjem je temeljen na principu da se prvo pišu testovi, a nakon toga se implementiraju metode koje će rješavati napisane testove. Svako pisanje koda se dijeli u tri faze: prva faza, druga faza i refaktorizacija.

Za izradu primjera se koristio programski jezik Python, a za programsko okruženje se koristio Python Notebook. Python se koristio zbog svoje jednostavnosti i mnogih implementiranih funkcija unutar njega. Svi primjeri koji su napravljeni u ovome radu mogu se implementirati u drugim programskim jezicima (npr. C++, Java …). Svaki programski jezik ima svoj aplikacijski okvir koji olakšava testiranje dijelova.

Ovaj diplomski rad podijeljen je u devet poglavlja. Prvo poglavlje opisuje klasične i agilne metode. Opisuju se karakteristike svake metode i navode primjeri.

Drugo poglavlje je uvod u razvoj softvera vođenog testiranjem. Opisuju se karakteristike razvoja softvera vođenog testiranjem i njegove faze. Kako bi se opisale kvalitete razvoja softvera vođenog testiranjem navedene su prednosti ovog načina razvoja softvera.

Treće poglavlje se bavi testiranjem dijelova i opisa mogučnosti koje pruža aplikacijski okvir *unittest* u programskom jeziku Python. U ovom poglavlju su opisane iznimke u Pythonu i objašnjeno je njihovo korištenje.

Kako se razvoj softvera vođen testiranjem koristi već dulje vrijeme uočili su se određeni obrasci koji se često pojavljuju, oni su opisani u četvrtom poglavlju.

U petom poglavlju krećemo detaljnije opisivati prvu fazu razvoja softvera vođenog testiranjem i opisujemo nastale obrasce za tu fazu. Obrasci koji se opisuju u ovoj i ostalim fazama pomažu jednako iskusnim programerima koji koriste razvoj softvera vođen testiranjem kao i početnicima.

Testiranje je temelj ovog načina razvoja softvera. U šestom poglavlju opisujemo obrasce za testiranje od kojih se većina koristi u svakom razvoju softvera, koji su lagani i logični za korištenje.

Sedmo i osmo poglavlje opisuju obrasce za drugu i treću fazu razvoja softvera.

Deveto poglavlje je studijski primjer koje objedinjuje sva poglavlja i prikazuje korištenje razvoja softvera vođenog testiranjem. Radi se o kalkulatoru koji osim klasičnih funkcija kao što su plus, puta i sl. ima funkcije izračunavanja opsega i površina nekih geometrijskih likova.

# 1. METODE RAZVOJA SOFTVERA

Danas se sve metode razvoja softvera mogu podijeliti u 2 skupine, klasične i agilne metode. Klasične metode sadrže planiranje cijelog projekta unaprijed, dokumentiranja svih detalja projekta i ograničenog vremenskog razdoblja, dok u agilnim metodama planiranja su kratkoročna ( par tjedana unaprijed ), smatra se da je kod dovoljna dokumentacija.

Ne postoji pravilo koja će metoda donijeti bolji rezultat na kraju i koju će biti jednostavnije primijeniti. Godinama se vode rasprave koja je metoda bolja, svaka ima svoje prednosti i mane, te svaka metoda ima svoje zagovornike. Tijekom godina pokazalo se da su za manje projekte bolje agilne metode zbog svoje fleksibilnosti.

U novijim tvrtkama se sve više kod planiranja razvoja softvera, razmatra koja će se metoda koristiti, kako bi se projekt mogao što kvalitetnije napraviti, te se smatra da se programeri mogu prilagoditi svakoj metodi.

## 1.1. KLASIČNE METODE

Pojavom aplikacije VisiCalc 1979. godine za računalo Apple II, uviđa se da računala nisu samo za osobnu razonodu, nego da se mogu koristiti u poslovanju. Kreće se javljati potreba za raznim poslovnim aplikacijama. Softver se krenuo razvijati kao i ostale inženjerske struke ( npr. brodogradnja ).

Testiranje aplikacija odvijalo se tako da testeri ručno moraju unositi podatke i zapisivati dobivene rezultate. Dobivale su se testne skripte koje su bile instrukcije kako tester mora obaviti određenu radnju u sustavu i što mora dobiti. Ovi procesi su bili spori, te je za testiranje bilo potrebno i nekoliko tjedana. Kada se sustav mijenjao mijenjali su se i testovi, te su svi testovi morali biti ponovno izvršeni.

### 1.1.1. PRIMJER KLASIČNE METODE – *Unified Process*

UP (*Unified Prosess*) se prvi puta spominje 1999. godine u knjizi The Unified Software Development Process. Ova metoda je iteraktivni i inkrementalni razvojni proces. Vrlo važna komponenta ovog modela su slučajevi uporabe (*eng. Use case*) koji se koriste kako bi se odredili funkcionalni zahtjevi i kako bi se definirao sadržaj pojedinih iteracija.

UP se sastoji od četiri faze koje se realiziraju u vremenskom slijedu jedna iza druge. Pojedina faza završava kada se dosegne njezin međaš (*eng. Milestone*), dakle kad se postignu njeni ciljevi. Svaka faza realizira se kroz simultano i iterativno odvijanje 5 temeljnih aktivnosti.[4]

Faze UP-a:

1. **Start** (*eng. Inception*) – Najkraća faza faza u projektu. Cilj ove faze je određivanje projektnog plana. Prema [7] po završetku ove faze moraju biti zadovoljeni sljedeći uvjeti:
    - isplativost za korisnike,
    - razumijevanje zahtjeva,
    - vjerodostojnost procjene troškova, rokova, prioriteta, rizika
    - složenost do tada razvijenih prototipova
    - uspostava osnovne odrednice prema kojoj će se uspoređivati ostvareni nasuprot planiranih troškova
2. **Elaboracija** (*eng. Elaboration*) – Ciljevi ove faze su ustanoviti faktore poznatih rizika, stvoriti jezgru arhitekture sustava, evidentirati slučajeve uporabe, sastaviti plan razvoja za cjelokupni projekt. Ako se nakon faze elaboracije ustanovi da je trenutni dizajn neisplativ, može se napraviti redizajn sustava ili odustati od projekta.
3. **Konstrukcija** *(eng. Construction)* – Najopsežnija faza projekta. Na temelju elaboracije se implementira sistemska funkcionalnost. Implementacija se odvija u iteracijama, nakon svake iteracije dobije se gotova verzija proizvoda. Pišu se potpuni slučajevi uporabe te svaki postaje početak nove iteracije. Cilj je razviti dizajnirani sustav. Na kraju ove faze se dobije prva verzija sustava.
4. **Tranzicija** (*eng. Transition* ) – Završna faza projekta. Gotov sustav se distribuira korisnicima. Uočene pogreške se popravljaju i daljnje verzije se usavršavaju.

Prema [4] opisujemo temeljne aktivnosti u UP:

1. **Zahtjevi** *(eng. Requirements)* – Utvrđuje se što sustav treba raditi.
2. **Analiza** *(eng. Analysis)* – Zahtjevi se analiziraju, profinjuju i strukturiraju.
3. **Oblikovanje** *(eng. Design)* – Predlaže se građa sustava koja će omogućiti da se zahtjevi realiziraju.
4. **Implementacija** *(eng. Implementation)* – Stvara se softver koji realizira predloženu građu.
5. **Testiranje** *(eng. Test)* – Provjera se da li stvoreni softver zaista radi onako kako bi trebao.

## 1.2. AGILNE METODE

Predlagatelji novih metoda kao što su bile Extreme Programming (XP), Scrum, Pragmatic Programming i ostali su 2001. godine napisali „Agile Manifesto“ u kojem opisuju nove metode razvoja softvera koje se zasnivaju na individualnošću i interakcijama u timu, softveru koju radi što je traženo od strane korisnika, kolaboraciju s korisnicima tijekom razvoja.

Prema [4] agilne metode imaju sljedeća svojstva:

- Razvoj softvera promatra se kao kontinuirani niz iteracija čiji broj nije moguće predvidjeti. Svaka iteracija usklađuje sustav s trenutnim zahtjevima. Zahtjevi se stalno mijenjaju, ne postoji cjelovita ili konačna specifikacija.
- Ne postoji upravljanje projektom u pravom smislu riječi. Aktivnosti se dogovaraju s korisnicima te se odvijaju uz njihovo aktivno sudjelovanje. Planiranje je kratkoročno, jedan ili dva tjedna unaprijed.
- Izbjegavaju se svi oblici dokumentacije osim onih koji se mogu automatski generirati iz programa. Smatra se da sam program u izvornom obliku predstavlja svoju najpouzdaniju dokumentaciju.

Agilne metode koje se koriste danas :

- Scrum
- Extreme Programming (XP)
- Feature Driven Development
- Clear Case
- Adaptive Softvare Development

Ove metode se razlikuju u njihovom implementiranju ali imaju neke zajedničke karakteristike:

- Važna je komunikacija unutar tima
- Može se računati na kolege unutar tima, odvajanje vlastitog vremena kako bi se pomoglo u rješavanju problematike kolega
- Nitko nije vlasnik dijela koda, nego su svi vlasnici cijelog koda i svatko je odgovoran za cjelokupnu kvalitetu koda
- Razvoj softvera je podijeljen u cikluse
- Programeri se moraju moć nositi s promjenama

### 1.2.1. PRIMJER AGILNE METODE - *Extreme programming*

Kako bi se bolje shvatio princip agilnih metoda opisat ćemo *Extreme programming* (*XP*) metodu.

Cilj *XP* je smanjiti cijenu promjena u zahtjevima tako da ima mnogo manjih razvojnih ciklusa. Planira se razvoj samo za sljedeći ciklus. Svaki ciklus može dodavati nove metode u postojeći program ili mijenjati postojeće.

Programeri su u stalnom kontaktu s korisnicima i s njima raspravljaju koje sve funkcionalnosti mora imati program. Moraju dobro razumjeti zahtjeve korisnika kako bi im mogli dati povratne informacije o tome kako se može taj zahtjev riješiti s tehničke strane (implementirati) ili ako se ne može riješiti, korisnicima se mora objasniti razlog.

Zahtjevi u *XP* prikazuju se kao skup kratkih kartica teksta, takozvanih *korisničkih priča*. Na početku ciklusa se biraju samo one priče koje će se implementirati u dotičnom izdanju. Ostale priče ostavlja se za ostale cikluse. [4]

Pristaše *XP* metode smatraju da je najbitniji dio razvoja sustava kod. Te njega smatraju dovoljnom dokumentacijom. Kod mora biti pregledan i pisan na način kako bi ga ostali programeri u timu mogli razumjeti.

*XP* uvodi programiranje u paru – dva programera sjede za jednim računalom i zajedno pišu isti programski kod [4]. Time se lakše uočavaju pogreške u kodu ( najteže je sam sebi otkriti pogreške kod programiranja ) i brže se osmišljavaju ideje kod rješavanja problema. Parovi se mijenjaju kod novih ciklusa ili zadataka kako bi svi doprinijeli razvoju cijelog sustava, a ne samo nekog njegovog dijela.

Najveća novost *XP* metode je pisanje koda vođeno testiranjem ( eng. *test-first development)* iz čega se kasnije razvila metoda razvoja softvera vođena testiranjem. Uobičajena praksa kod pisanja programa je napisati kod i nakon toga ga testirati da li radi. Kod razvoja softvera vođenog testiranjem pišu se prvo testovi i nakon toga se piše kod koji će proći napisane testove. Time se osigurava da sav kod koji je napisan radi (prolazi testove) i da će se implementirati samo potrebne metode ( neće biti nepotrebnih metoda).

*XP* metoda najpopularnija je kod manjih projekata ili kod projekata koji imaju promjenjive zahtjeve. U zadnje vrijeme kod sve više većih i složenijih projekata se odlučuju koristiti ovu metodu jer omogućuje izradu koda koji radi samo ono što je potrebno i traženo od korisnika (razvoj softvera vođen testiranjem).

# 2. OPĆENITO O RAZVOJU SOFTVERA VOĐENOG TESTIRANJEM (*eng. Test Driven Development - TDD*)

## 2.1. UVOD U RAZVOJ SOFTVERA VOĐEN TESTIRANJEM

Razvoj softvera vođen testiranjem spada pod agilne metode razvoja softverskog sustava. Nastao je 1999 godine u metodi *Extreme programming* . Danas je popularna metoda kod razvoja manjih softverkih projekata, iako se sve više složeniji softverski projekti rade njome. Odlika *TDD* je što kraći i pregledniji kod, pa je on sam dovoljna dokumentacija za projekt.

Razlika između *TDD* i *test-first developmenta (TFD)* je u tome što se s *TFD* podrazumijeva pisanje testova razvoja manjih dijelova programa kao što su klase, metode i sl. , a *TDD* je metoda za razvoj softvera i cjelokupnih programa. Iako ova dva pojma su slična u literaturi se često za *TFD* govori da je *TDD*, a u nekim literaturama se smatra da su ta dva pojma ista.

## 2.2. OBLIKOVANJE SOFTVERA

Nakon utvrđivanja zahtjeva napravi se grubi izgled i trenutna ideja softverskog sustava. Ideja cjelokupnog softvera je podložna promjenama iz razvojnog ciklusa u razvojni ciklus. U svakom novom razvojnom ciklusu može doći do dodavanja metoda koje nisu bile zamišljene ili izbacivanja metoda koje su bile u prvobitnom planu. Korisnici imaju veliku ulogu kod razvijanja softvera, nakon svakog razvojnog ciklusa dobe uvid u trenutni softver. Pisanje testova omogućava da završni produkt bude u skladu potreba i želja klijenata. Gotov softverski sustav sličan je prvobitno zamišljenom sustavu, ali ne mora biti isti.

## 2.3. RAZVOJ SOFTVERA

Razvoj programa vođen testovima, se svodi da se prvo napišu testovi, pa nakon toga krenu implementirati metode koje će proći testove. Pišu se jedan po jedan test, tek nakon što se implementira metoda koja prolazi prvi test, kreće se na pisanje i rješavanje drugog testa. U dogovoru s korisnicima i njihovim zahtjevima pišu se testovi, rješavanjem tih testova se osigurava željena funkcionalnost softvera.

Kada se napiše test, prva indikacija da test neće proći je ta što će test pokušati instancirati klasu ili funkciju koja još nije implementirana. Prvi korak prema rješavanju testa je

napraviti klasu i metodu koju test želi pozvati. U ovom slučaju test također neće proći jer klase i metode ne rade ništa. Nakon toga se kreće pisati što jednostavnija funkcionalnost metoda, ali da pritom test mora proći, tj. da nas se prevede program. Taj kod mora biti što jednostavniji i s ciljem da samo prođe test, ne smije se zamarati s time koje bi sljedeće metode se trebale implementirati. Poanta ovog koraka je da nam kod bude što jednostavniji za pregled i kasnije za održavanje. Ako bi se implementirale dodatne funkcionalnosti koje nam trenutno ne trebaju, lako moguće da kasnije neće biti potrebne za rad softvera, ali će povećati kod i time otežati njegovo održavanje.

Nakon što naši testovi prođu, napišemo negativne testove, to su testovi za koje znamo da naše implementirane metode neće proći.

Npr. ako kažemo da PDV ne može biti veći od 25% i izračunavamo ga na nekom iznosu, stavimo da PDV bude barem 26% da vidimo što će se dogoditi. U pravilu bi program trebao izbaciti iznimku. Kod takvih testova, program nam mora izbaciti željenu iznimku, u našem slučaju bi iznimka trebala napisati „PDV ne smije biti veći od 25%“, takve testove smatramo da su prošli.

Nakon što nam prođu testovi, refaktoriziramo do sada napisani kod, na način da maknemo duplikacije koda, prepravimo ga da izgled zadovoljava dobru programersku praksu, itd. (više o refaktorizaciji u osmom poglavlju).

## 2.4. *RED-GREEN-REFACTOR*

### 2.4.1. TRI FAZE SOFTVERA VOĐENOG TESTIRANJEM

*Red, green, refactor* odnosi se na tri faze koje se pridržavaju programeri koji koriste metodu razvoja softvera vođenog testiranjem. Ovaj poredak koraka osigurava da imamo testove za kod koji ćemo pisati i da pišemo samo onaj kod koji nam je potreban kako bi prošli testovi.

### 2.4.2. PRVA FAZA (eng. Red Phase)

Faza koja programerima koji su tek započeli s metodom razvoja softvera vođenog testiranje stvara najviše problema. U ovoj fazi se pišu testovi, za klase i metode koje još ne postoje. Testovi se neće kompilirati, što znači da neće proći.

Može se testirati i postojeća klasa sa nekom novom metodom ili metoda s novom funkcionalnošću, ali i ti testovi se neće kompilirati jer tražena svojstva još nisu

implementirana. Ako mijenjamo neke metode testovi se budu kompilirali ali nećemo dobiti tražene rezultate pa i dalje testovi nam neće proći.

Nakon što novo napisani testovi ne prođu, cilj nam je otići na drugu fazu (*eng. Green phase*).

### 2.4.3. DRUGA FAZA (eng. Green Phase)

Ako testovi ne prolaze znači da se nalazimo još u prvoj fazi. Kako bi se došlo do druge faze potrebno je napisati najmanje moguće koda da nam testovi prođu. Ponekad je teško odrediti kako napisati najmanje koda, pa je dobra praksa napisati što jednostavniji kod kako bi nam test prošao.

Npr. želimo napisati metodu koja ako joj se pošalje paran broj vraća 1.

```
def par_ili_nepar(broj):
    return 1
```

Slika 2.1

Sa slike 2.1 vidimo najmanje potreban kod kako bi nam se metoda prevela i kako bi prošao test koji šalje paran broj. Nakon što se kod prevede i prođe test, dodamo ostatak koda kako bi nam metoda radila što je bilo zamišljeno.

```
def par_ili_nepar(broj):
    if broj%2==0:
        return 1
    else:
        return 0
```

Slika 2.2

Mnogim programerima bi prvo palo na pamet napisati kod kao sa slike 2.2, poanta druge faze je da bude što manje koda, a da on prođe test. Kod ovakvih jednostavnijih testova nije pogreška odmah napisati cijelu metodu, ali kada se jave kompliciraniji testovi za koje metode neće imati par linija koda, treba pisati dovoljno koda da test prođe. Nakon što test prođe prelazi se na treću fazu – refaktorizaciju.

### 2.4.4. TREĆA FAZA – REFAKTORIZACIJA

Do ove faze se nije trebalo brinuti o kvaliteti koda, jednostavnosti održavanja i jednostavnosti čitanje koda. U ovoj fazi mijenjamo kod na temelju ova tri atributa, osim toga u ovoj fazi programeri poboljšavaju svoj kod. Testovi u ovoj fazi nam služe kako bi programeri vidjeli da li napravljene promjene utječu na njihovu prolaznost. Dok testovi prolaze znamo da je kod u redu. Nakon što je gotova faza refaktorizacije, krećemo ispočetka s prvom fazom i novim testom.

## 2.5. PREDNOSTI RAZVOJA SOFTVERA VOĐENOG TESTIRANJEM

Prema [1] Benderu i McWherteru prednosti razvoja softvera vođenog testiranjem su:

- Osigurava kvalitetan kod od početka. Piše se samo kod temeljem kojih će proći testovi i time će zadovoljiti zahtjeve korisnika. Manje koda znači manje pogrešaka u istom.
- Softver zadovoljava svim zahtjevima klijenata. Zahtjevi su pisani kao testovi, a testovi se rješavaju, to osigurava veliku sigurnost da će kod sadržavati sve zahtjeve klijenata.
- Rade se jednostavniji programi i korisnička sučelja ( *eng. Application Interface - API* ). Programer koji piše programe i radi sučelja je također prva osoba koja ih koristi, te na temelju toga odmah vidi da li imaju smisla ili im je potrebna promjena
- U krajnjem produktu ima jako malo ili uopće nema nekorištenog koda. Kod pisanja koda važno je samo implementirati metode pomoću kojih će proći test. Ne implementiraju se metode za koje nema testova.
- Testovi osiguravaju da kod buduće promjene koda ili dodavanja novih metoda se neće narušiti funkcionalnost. Ako testovi prođu nakon mijenjanja koda, znači da se nije ništa narušilo.
- Jako malen broj *bugova* ili ih nema. Kod pisanja testova i njihove implementacije, ako se uoči problematika ili *bug* kod se ispravlja i radi se test za tu određenu problematiku ili *bug*, koji osigurava da se neće više pojavljivati poteškoće.

## 2.6. JEDNOSTAVAN PRIMJER RAZVOJA SOFTVERA VOĐENOG TESTIRANJEM

Kako bi se vidio način rada metode razvoja softvera vođenog testiranjem napravit ćemo kratki primjer programa koji izračunava opseg trokuta.

```
def test_trokut(a,b,c):
    trokut = Trokut(a,b,c)
    print(trokut.opseg())

if __name__ == '__main__':
    test_trokut(5,7,12)
```

Slika 2.3

Kao što vidimo na slici (2.3) prvo implementiramo test, u kojem navedemo sve tri strane trokuta, te nakon toga napravimo objekt trokut s tim stranicama i na kraju pozivamo metodu koja izračunava opseg trokuta. Nakon toga pokrenemo program kako bi vidjeli da test neće proći.

```
<ipython-input-1-b9ae841b6666> in test_trokut(a, b, c)
      1 def test_trokut(a,b,c):
----> 2     trokut = Trokut(a,b,c)
      3     print(trokut.opseg())
      4
      5 if __name__ == '__main__':

NameError: name 'Trokut' is not defined
```

Slika 2.4

Test nije prošao jer nemamo implementiranu klasu *Trokut* i u njoj metodu *opseg()*. To dvoje se sljedeće implementira.

```
class Trokut:
    def __init__(self, a, b, c):
        self.a=a
        self.b=b
        self.c=c
    def opseg(self):
        return self.a+self.b+self.c
```

Slika 2.5

Nakon pokretanja koda dobijemo broj 24, što je traženi opseg. Nakon što nam je test prošao, kreće refaktorizacija koda, ali s obzirom na to da smo od početka krenuli s dobrom programerskom praksom u ovome slučaju ona nije potrebna. U ovome primjeru bi se još mogli implementirati slučajevi ako se stavi da je neka od stranica truka negativan broj ili nula, ali to nećemo sada napraviti jer će se to kasnije obraditi u poglavlju 3.5.

Za pisanje softvera vođenog testiranjem danas se koriste već napravljeni aplikacijski okviri (*eng. Framework* ) koji olakšavaju pisanje testova i njihovo pokretanje. S time ćemo se baviti u sljedećem poglavlju.

# 3. TESTIRANJE DIJELOVA (*eng. Unit Testing*)

## 3.1. UVOD U TESTIRANJE DIJELOVA

Testiranje dijelova je test namijenjen kako bi testirao jedan dio jedne metode (*eng. One unit of one method*), može se testirati i cijela metoda. Ne zanima nas rad ostalog dijela sustava, osim ako nije direktno povezano sa dijelom metode koju želimo testirati. To olakšava testiranje, jer se testiraju mali dijelovi i lako se piše kod koji prolazi te testove. Ako neki test ne prolazi lako se uoči u kojem dijelu metode je problem. Mogu se testirati i cijele metode jednim testom, ali je preporučljivo da testovi testiraju što manji dio koda (dijelove metoda). Ako bi se jednim testom testiralo više metoda, teško bi se našla i ispravila pogreška u kodu.

*Unit* testovi najbitniji su testovi u razvoju softvera vođenog testiranjem, postoje još ostali tipovi testova koji su također bitni za različite dijelove sustava.

Ostali tipovi testova:

- Testiranje korisničkog sučelja
- Testovi integracije sustava – testira se na kraju kada su svi dijelovi sustava gotovi i kada se trebaju integrirati u cjelinu
- Testiranje opterećenja (*eng. Stress testing*) – testiranje opterećenja sustava kada više korisnika odjednom koristi sustav
- Test korisničkog prihvaćanja – davanje gotove aplikacije na uvid korisnicima, da se vidi da li zadovoljava njihove zahtjeve, te ako je sve u redu nakon ovog testa aplikacija se pušta u prodaju.

## 3.2. APLIKACIJSKI OKVIRI (*eng. Framework* ) ZA TESTIRANJE DIJELOVA

Sve većom popularnosti razvoja softvera vođenog testiranjem, kreću se razvijati aplikacijski okviri koji omogućuju lakše upravljanje testovima i njihovim izvršavanjem. Oni omogućuju programerima da ne trebaju sami stvarati testna okruženja. Pomoću njih svi testovi se mogu pokrenuti pomoću „gumba“. Danas za većinu programskih jezika postoji aplikacijski okvir za testiranje dijelova.

U Pythonu ima više različitih *unit* aplikacijskih okvira, a mi ćemo koristiti *unittest.* Dodaje se naredbom *import unittest,* klase i koje sadrže testove moraju imati ključnu riječ *test* na početku svoga naziva kako bi ih aplikacijski okvir mogao prepoznati i pokrenuti. Testovi se pokreću naredbom *unittest.main()* u *main* dijelu programa. Slijedeći test će prikazati primjer izračunavanja opsega trokuta kao u odlomku 2.5 ali ovdje ćemo

koristiti *unittest* kako bi se vidjela razlika kada se koristi aplikacijski okvir i kako izgleda bez njega. Klasa *trokut* sa slike 2.5 ostaje ista.

```
import unittest

class TestTokut(unittest.TestCase):

    def test_opseg(self):
        trokut=Trokut(5,7,12)
        self.assertEqual(trokut.opseg(),24)

if __name__ == '__main__':
    unittest.main(argv=['ignored', '-v'], exit=False)
```

Slika 3.1

Prvo smo definirali klasu *TestTrokut* u koju bi stavljali sve testove vezane za trokut, u našem slučaju samo ispitujemo da li u klasi *Trokut*, dobro radi funkcija *opseg()*. Klasa je naslijedila *unittest.TestCase* kako bi se znalo da ta klasa sadrži testne primjere, te će se moći kasnije pokrenuti sa *unittest.main()*. *Unittest.main()* ima argumente argv=['ignored', '-v'] i exit=False, oni samo govore da se ignorira *sys.argv* jer njega želi dohvatiti Python notebook ali ne može.

U klasi *TestTrokut* smo definirali funkciju *test_opseg* koja provjerava opseg trokuta. U njoj napravimo trokut sa stranicama 5,7 i 12. Sa *self.assertEqual()* provjeravamo da li funkcija *opseg()* iz klase *Trokut* dobro izračunava opseg trokuta.

Nakon što pokrenemo program dobijemo sljedeći ispis:

```
test_opseg (__main__.TestTokut) ... ok

----------------------------------------
Ran 1 test in 0.000s

OK
```

Slika 3.2

Na ispisu vidimo koji je test pokrenut ( test_opseg() ), koliko mu je bilo vrijeme izvršavanja (0.000s) i da li je test prošao (ok).

### 3.3. ASSERT

Najčešće metode koje se koriste za testiranje dijelova su *assert* metode, koriste se kako bi se usporedili dobiveni rezultati sa željenim rezultatima.

Neke od najčešćih *assert* metoda koje omugućava *unittest*:

- **assertEqual( *a,b* )** – testira da li su *a* i *b* jednaki, ako nisu test neće proći.
- **assertNotEqual( *a,b* )** – ako su *a* i *b* različiti test prolazi.
- **assertTrue( *x* )** – kako bi test prošao *x* mora biti istinita bool varijabla, ako *x* nije bool varijabla, pretvara se u nju.
- **assertFlase( *x* )** – provjerava se da li je *x* lažna bool varijabla
- **assertIs( *a,b* )** – testiranje da li *a* i *b* pokazuju na isti objekt
- **assertIsNot( *a,b* )** – test prolazi ako *a* i *b* ne pokazuju na isti objekt
- **assertIsNone( *x* )** – test prolazi ako je *x None* ( nema pridruženu vrijednost)
- **assertIsNotNone( *x* )** – testiranje da li je *x* različit od *None*
- **assertIn( *a,b* )** – testiranje da li je vriejdnost *a* sadržana u *b*
- **assertNotIn( *a,b*)** – testiranje da li vrijednost *a* nije sadržana u *b*
- **assertRaises( po*greška, fja, args*)** – testiranje da li će pokretanjem funkcije *fja,* s argumentima *args,* dati iznimku *pogreška*
- **assertGreater( *a,b* )** – testiranje da li je zadovoljeno $a > b$
- **assertGreaterEqual( *a,b* )** – testiranje dali je zadovoljeno $a >= b$
- **assertLess( *a,b* )** – testiranje da li je zadovoljeno $a < b$
- **assertLessEqual( *a,b* )** – testiranje da li je zadovoljeno $a <= b$

Ove i sve *assert* metode se mogu naći na stranici [3].

### 3.4. METODE KOJE SADRŽI *UNITTEST* APLIKACIJSKI OKVIR

*Unittest* sadrži mnoge funkcije koje omogućavaju lakše testiranje i pisanje testova. Opisat ćemo samo neke, koje se najčešće koriste, a cijela dokumentacija se može vidjeti na [3].

***setUp()*** – služi za postavljanje testnog okruženja, kao što je inicijalizacija varijabli, kreiranje objekata i sl. Sve iznimke koje se dogode u ovoj metode javljaju se kao pogreške, a ne kao pad testa.

**tearDown()** – metoda koja se poziva nakon što se pozvala testna metoda i spremio testni rezultat. Poziva se i ako se dogodila iznimka u testnoj metodi. Sve iznimke osim

*AssertionError* ili *SkipTest* koje se dogode unutar ove metode se smatraju kao pogreške u programu, a ne kao pad testa.

**run( *result* = *None* )** – pokreće se test, a rezultat se sprema u objekt *TestResult* koji se prosljeđuje kao *result*.

**skipTest()** – poziv ove metode tijekom testne metode ili *setUp()* metode preskače se trenutni test.

**debug()** – pokreće testove bez spremanja rezultata. Može se koristiti za podržavanje pokrenutih testova u *debuggeru.*

### 3.5. IZNIMKE ( eng. *Exceptions* )

Iznimke su pogreške koje se mogu uočiti tek tijekom izvršavanja programa( npr. umjesto broja smo unijeli slovo ). Iznimke nisu poseban dio *unittesta* nego su sastavni dio *Pythona*. One su neizostavan dio kod izrade testova i kod implementacije koda. Njihovo poznavanje i manipulacija njima programerima omogućava da se program neće urušiti kod korisničkih pogrešaka ili nekog sličnog odstupanja od zamišljenog upravljanja programom.

Osim testova koji provjeravaju funkcionalnost i točnost metoda, pišu se testovi koji provjeravaju ponašanja programa kod određenih iznimaka.

Iznimke se u većini slučaja hvataju unutar *try – except* blokova. U *try* bloku se piše kod koji se izvršava ako ne dođe do iznimke, a ako u nekom trenutku dođe do određene iznimke kreće se izvršavati *except* blok.

```
while True:
    try:
        x = int(input("Unesite broj: "))
        break
    except:
        print("Niste unijeli dobar broj, pokušajte opet...")
```

Slika 3.3

Na slici 3.3 je prikazan primjer u kojem ako korisnik ne unese dobar broj aktivirat će se iznimka *ValueError* i korisniku će se ispisati poruka *„Niste unijeli dobar broj, pokušajte opet...“* . U slučaju da je korisnik unio dobar broj, program neće izvršiti *except* blok i korisniku se neće prikazati poruka o krivom unosu.

U primjeru nismo definirali koja se iznimka može pojaviti, nego u slučaju bilo kakve iznimke program korisniku ispisuje poruku o krivom unosu.

Kod hvatanja određenih iznimki nakon ključne riječi *except* dolazi ime iznimke koju želimo uhvatiti.

```
except ValueError:
        print("Niste unijeli dobar broj, pokušajte opet...")
```

Slika 3.4

Kada bismo u primjeru sa slike 3.3 htjeli uhvatiti iznimku *ValueError* koja se aktivira ako se unese kriva znamenka, a kod nas su to svi znakovi osim brojeva. Program to prepoznaje jer ulazni znak pokušava pretvoriti u broj.

Nabrojat ćemo jedne od najčešćih iznimaka(popis svih iznimaka se može naći na [3]):

- **ValueError**
- **ZeroDivisionError**
- **SyntaxError**
- **OverflowError**
- **NameError**
- **KeyboardInterrupt**
- **EOFError**
- **AssertionError**
- **ImportError**
- **IndexError**

Ako želimo hvatati više od jedne iznimke odjednom, možemo nakon *try* bloka napisati više *except* blokova, svaki blok za jednu iznimku koju želimo uhvatiti. Kada hvatamo iznimke možemo nakon imena iznimke koju želimo uhvatiti stavi ključnu riječ *as* i nakon toga ime varijable u koju želimo spremiti iznimku koju je program napravio.

```
while True:
    try:
        x = int(input("Unesite broj: "))
        dio = 10/x
        break
    except ValueError:
        print("Niste unijeli dobar broj, pokušajte opet...")
    except ZeroDivisionError as greska:
        print(greska)
```

Slika 3.5

Naš primjer sa slike 3.3 smo proširili tako da broj 10 podijelimo s unesenim brojem. Na slici 3.5 se vidi kako želimo uhvatiti dvije vrste pogrešaka. Ako uhvatimo pogrešku *ZeroDivisionError* ( dijeljenje s nulom ) program će nam ispisati koju je pogrešku uhvatio.

```
Unesite broj: 0
division by zero
Unesite broj: a
Niste unijeli dobar broj, pokušajte opet...
Unesite broj: 1
```

Slika 3.6

Na slici 3.6 vidimo poruke koje nam program javlja ako unosimo simbole koji će generirati iznimke.

Osim što možemo hvatati iznimke i manipulirati što se događa kod njihovog nastajanja, možemo ih „ručno generirati“. Misli se da ih programer u svome kodu ručno implementira kada želi da se iznimka generira. Pomoču „ručno“ generiranih iznimaka možemo testirati program da se ponaša kako želimo kod nastajanja iznimaka. Iznimke se pozivaju s ključnom riječi *raise* i nakon toge ime iznimke koju želimo generirati. Najčešće se iznimke generiraju u *try – except* bloku kako bi se vidjelo da li program dobro prepoznaje iznimku.

```
try:
    raise ZeroDivisionError
except ZeroDivisionError:
    print("Generirali smo i uhvatili gresku")
```

Slika 3.7

Na slici 3.7 napisali smo jednostavan primjer u kojemu se generira iznimka *ZeroDivisionError* u *try* bloku, a u *except* bloku hvatamo tu istu iznimku i nakon što se uhvati ispiše se tekst „Generirali smo i uhvatili gresku“.

Nakon *try – except* blokova može se dodati blok s ključnom riječi *finally*, *finally* se može staviti i nakon *try* bloka čak i ako nemamo *except* blok. U *finally* stavljamo kod koji će se izvršiti neovisno o tome da li se generirala određena iznimka u *try* bloku.

```
def dijeli(x,y):
    try:
        x/y
    except ZeroDivisionError:
        print("Dijelite sa nulon")
    finally:
        print("Kraj funkcije")
```

Slika 3.8

Na slici 3.8 imamo primjer funkcije *dijeli()* koja prima dva argumenta i dijeli prvi sa drugim. Funkcija hvata iznimku ako smo kao drugi argument unijeli nulu i ima *finally* blok koji ispisuje „Kraj funkcije“ neovisno o iznimci.

```
Kraj funkcije
```

Slika 3.9

Na slici 3.9 vidimo ispis ako pozovemo funkciju *dijeli(2,1)*.

```
Dijelite sa nulon
Kraj funkcije
```

Slika 3.10

Na slici 3.10 vidimo ispis kod poziva funkcije *dijeli(2,0)*.

# 4. OBRASCI ZA RAZVOJ SOFTVERA VOĐENOG TESTIRANJEM

## 4.1. TEST

Svaka promjena koja se radi u programu treba imati adekvatni test koji će je testirati. Bez testova koji će evaluirati kvalitetu napisanog koda, mogu se dogoditi pogreške koje će se kasnije teško otkriti.

Kod testiranja bitno je da programer nije pod stresom, jer što je više pod stresom, to će manje testova raditi, a više pisati kod. To negativno utječe na daljnje pisanje jer što manje testira to će više biti pod stresom kako neće znati da li kod prolazi. To je začarani krug iz kojega se teško izvući. Zbog takvih problem su najbolji testovi koji testiraju samo dio metode ili klase. Što je test jednostavniji, to ga je lakše napisati i to će prije proći test, a što lakše kod prođe test to će programer manje osjećati stres.

Kod tek napisanih testova može se prevesti program kako bi se vidjelo da test neće proći. Ta metoda je najpoželjnija kod programera koji tek počinju koristiti metodu razvoja softvera vođenog testiranjem. Nakon što programer skupi dovoljno iskustva i bude se dovoljno dobro osjećao koristeći ovu metodu, nije potrebno da svaki puta kada napiše test prevede program, jer zna da test neće proći, a ako će pisati male testove, može uštedjeti na vremenu ne prevodeći program kod svakog testa.

## 4.2. IZOLIRANI TEST

Kada se pišu testovi mora se paziti na sljedeće stvari :

- Da ne utječu jedni na druge ( ako više testova ne prođe želimo da nam se za svaki javi pogreška, a ne ako jedan test ne prođe, da ne prođe još koji test )
- Ne dijele zajedničke varijable
- Imaju kratko vrijeme izvršavanja
- Mora biti neovisno u kojem redoslijedu se izvršavaju
- Kod dijeljena podataka (baza podataka i sl.) paziti da ako jedan test radi promjene na podatcima to ne utječe na druge testove

## 4.3. LISTA TESTOVA

Prije nego se krenu testovi kodirati trebali bi se svi testovi koje želimo implementirati napisati u nekom dokumentu kako bi se kasnije lakše znalo kojim redom

ih kodirati i kako se ne bi zaboravilo na neki test ( time se smanjuje i funkcionalnost krajnjeg softvera ).

Loša praksa je sve testove „imati u glavi“ ili ih implementirati kako ih se sjetimo. Ovaj način često dovodi da se zaboravi neki test i da se misli kako su neki testovi trivijalni i zato nepotrebni.

Pisanjem plana što se misli napraviti u slijedećih par sati, u tome danu, tog tjedna ili mjeseca, olakšava organizaciju radu i omogućava uvid u postotak napravljenog posla. Kod pojave novih metoda za napraviti ili dodatnog posla, lakše se može isplanirati kada će se raditi na njima, te do kada treba biti gotovo.

### 4.4. PRVO TVRDNJE

Prije nego što se krenu pisati testovi, treba proći željene specifikacije softvera na kojem radimo. Niti jedan test ne bi trebao biti napisan prije nego što se provjeri hoće li taj test biti koristan u gotovom sustavu. Testovi se trebaju pisati dio po dio, s time da se na početku kreće s time što test treba uspoređivati. Nakon što se početni dio napiše, ostatak testa se piše kako programer misli da će mu biti lakše. Može pisati od kraja prema početku (od toga što se uspoređuje, funkcije koje se koriste pa na kraju definiranje varijabli i objekata) ili će napisati od početka prema kraju. Na slijedećem primjeru ćemo ilustrirati preporučeno pisanje testa.

Napisat ćemo test koji provjerava da li kada opsegu trokuta oduzmemo broj 10 dobijemo broj 23.

Prvo definiramo test i napišemo što bi trebao uspoređivati. Tri točke označavaju mjesto gdje ćemo još dodati kod.

```
class test(unittest.TestCase):
    def test_opseg_broj(self):
        ...
        self.assertEqual(umanjen_opseg_trokuta,23)
```

Slika 4.1

Nakon što smo definirali što želimo uspoređivati, gledamo kojim ćemo smjerom ići, hoćemo li prvo definirati klasu *Trokut* pa od nje doći do opsega ili ćemo prvo napraviti smanjenje opsega za 10, pa od toga ići prema klasi *Trokut*. Mi ćemo izabrati drugi način.

```
class test(unittest.TestCase):
    def test_opseg_broj(self):
        ...
        umanjen_opseg_trokuta=opseg_trokuta-10
        self.assertEqual(umanjen_opseg_trokuta,23)
```

Slika 4.2

Još nam ostaje napraviti varijablu *opseg_trokuta* i definirati klasu *Trokut* sa zadanim stranicama.

```
class test(unittest.TestCase):
    def test_opseg_broj(self):
        trokut=Trokut(10,11,12)
        opseg_trokuta=trokut.opseg()
        umanjen_opseg_trokuta=opseg_trokuta-10
        self.assertEqual(umanjen_opseg_trokuta,23)
```

Slika 4.3

Nakon što smo napisali test moramo samo provjeriti da li su nam imena dobra, kako bi mogli razumjeti što se događa u testu samo na temelju koda.

## 4.5. PODATCI ZA TESTIRANJE

Podatci koji se koriste za testiranje moraju biti lagani za čitati. Ako ima razlike u podatcima tada ta razlika treba biti značajna (npr. isto je testirati da li je 3 manje od 4 i da li je 5 manje od 8). Ako metode mogu imati više različitih ulaznih podataka tada treba testirati sve mogućnosti ulaznih podataka.

Alternativi testnim podatcima su stvarni podatci, to su podatci koje koristimo iz stvarnoga svijeta. Stvarni podatci se samo mogu nabaviti ako radimo softver sličan nekome koji postoji ili nadograđujemo trenutni.

# 5. OBRASCI ZA PRVU FAZU (eng. Red Phase Patterns)

## 5.1. IZBOR TESTA

Nakon što smo završili kod za prolaz testa, postavlja se pitanje koji test slijedeće napisati. Odabire se onaj s kojim ćemo naučiti nešto novo ( testiranje nove metode ili ulaznih podataka koji će morati izbaciti iznimku ). Svaki test mora biti jedan korak prema sveukupnom cilju.

Nema točnog odgovora koji test je prije bolje implementirati, sve zavisi od osobe do osobe kojim će redom raditi. Bitno je da se osoba osjeća ugodno u tom trenutku implementirajući test i da ima ideju kako napisati metodu koja ga prolazi. Ponekad inspiracija za pisanje metode neće doći taj dan, pa ne treba siliti rješavanje testova koje u tom trenutku ne znamo.

## 5.2. POČETNI TEST

Za početni test bi bilo dobro odabrati neku inačicu operacije koju ćemo koristiti, a koja ne radi ništa. Treba znati gdje će operacija pripadati ( kojoj klasi ili funkciji ), kako bi je mogli dobro pozvati u testu. Test se treba implementirati tako da odgovori na samo jedno pitanje, a ne na nekoliko njih.

```
class testTrokut(unittest.TestCase):
    def test_prvi(self):
        trokut=Trokut()
        self.assertTrue(trokut)
```

Slika 5.1

Kada bi krenuli implementirati test za klasu *Trokut* prvo bi napravili test koji samo provjerava da li se može napraviti objekt *Trokut* ( kao na slici 5.1) .

Ako se radi test za aplikaciju s kakvom smo se već prije sreli, ne treba za početni test uzeti jednostavnu operaciju ili ispitivanje hoće li se napraviti objekt, može se napraviti test veće složenosti. Iako vođeni iskustvom u tim stvarima mora se paziti na pogreške i da samopouzdanje ne dovede do previše kompliciranog testa.

## 5.3. OBJAŠNJAVAJUĆI TESTOVI

Testovi osim što nam koriste kako bi isprobavali da li smo dobro iskodirali metode, mogu nam i služiti da pomoću njih objasnimo kod. Testovi koji objašnjavaju kod se implementiraju nakon što s prethodnim testovima provjerimo hoće li nam kod proći test. On služi kako bi drugi programeri mogli lakše razumjeli funkcionalnosti metoda.

```
class testTrokut(unittest.TestCase):
    def test_objasnjenja(self):
        strana_a=3
        strana_b=5
        strana_c=9
        trokut=Trokut(strana_a,strana_b,strana_c)
        opseg_trokuta=trokut.opseg()
        self.assertEqual(opseg_trokuta, 17)
```

Slika 5.2

Na slici 5.2 vidimo primjer testa koji objašnjava korištenje metode opseg iz klase *Trokut*, varijable su nazvane tako da se zna na što se koja odnosi.

Objašnjavajući test se može pisati odmah na početku, tako da objašnjava što se točno sve događa, ali oduzima više vremena, a nama je na početku bitno samo da možemo imati testove za metode koje ćemo implementirati. Ovi testovi se uglavnom pišu kada imamo implementiranu metodu i test koju ona prolazi.

## 5.4. TEST ZA UČENJE

Prije nego se krenu koristiti metode iz vanjskih datoteka, treba napraviti testove kako bi se bolje razumjelo kako te metode rade. Najveća prednost testiranja novih metoda je da ako te metode neće proći testove, onda nam neće niti raditi u programu jer smo nešto krivo napravili.

Sljedeći primjer je testiranje funkcije *power* u Pythonu iz matematičke biblioteke *NumPy*.

```
import numpy as np

class NPtest(unittest.TestCase):
    def test_power(self):
        x1=[0,1,2,3]
        exp=3
        rezultat=np.power(x1,exp)
        print("trazeni brojevi su:")
        print(rezultat)
```

Slika 5.3

Funkcija *power(x1,exp)*, prima dva argumenta, prvi argument *x1* je broj ili lista koju ćemo potencirati, a drugi argument *exp* je eksponent.

Kada pokrenemo test osim što će nam javiti dal imamo kakvu pogrešku u kodu, on će nam odmah i ispisati potenciranu listu kako bi se mogli uvjeriti da smo dobili željene rezultate.

## 5.5. TESTOVI POGREŠAKA

Ako se tijekom prevođenja uoči pogreška (*bug*) koji smo prethodno previdjeli osim što ćemo ga ispraviti u kodu, napravit ćemo test za tu pogrešku kako bi bili sigurni da nam se više ta pogreška neće dogoditi. S ovim pogreškama se ne misli na trivijalne pogreške kao što su zaboravljanje stavljanja zagrada i slično, nego na pogreške kao da nam *for* petlja ide izvan liste i slično.

Ti testovi se pišu kao da krećemo tek pisati test za metodu koja još nije implementirana.

# 6. OBRASCI ZA TESTIRANJE

Ovi obrsci su detaljnije tehnike pisanja testova.

## 6.1. MANJI TEST

Ako smo napisali preveliki test, koji se teško da cijeli implementirati i neki njegovi dijelovi nam stvaraju problem. U takvim slučajima možemo napraviti dvije stvari:

1. Podijelimo test u više manjih testova i krenemo rješavati manje testove, a s obzirom da će nam tada originalni veliki test biti višak, možemo ga obrisati, a i možemo ga ostaviti da vidimo da nismo nešto izostavili iz manjih testova (ako prođu sva tri mala testa, trebao bi i originalni test proći).
2. Napraviti jedan manji test koji će simulirati problematični dio u velikom testu. Krećemo prvo rješavati manji test i nakon što njega riješimo nastavljamo rješavati veći test. Riješeni mali test će nam osigurati da problematični dio u velikom testu prolazi.

Nakon što primijetimo da imamo preveliki test i prije nego što krenemo raditi manje testove, trebamo se zapitati zašto nam je test preveliki, da bi mogli iz toga izvući pouku, kako bi to mogli izbjeći u budućnosti.

## 6.2. OPONAŠAJUČI OBJEKT (eng. Mock Object)

Dobro napisani softver pokušava smanjiti i ograničiti ovisnosti. Ovisnosti unutar softvera se mogu smanjiti ali se ne mogu izbjeći. Te ovisnosti su uglavnom vezane na eksterne resurse koji su kompliciran ili skupi ( sporo spajanje, zauzimanja memorije i sl.). Takvi resursi su uglavnom baze podataka, datotečni sustavi, web servisi i slično.

Testiranje objekata koji su ovisni o takvim resursima može biti sporo i skupo, što nikako ne odgovara ideji da testovi budu brzi i efikasni. Sa oponašajućim objektima oponašamo skupe i kompliciran resurse, a da pritom ne koristimo originalne resurse. Takvi objekti omogućavaju da se ne povezujemo sa željenim resursima nego sami implementiramo podatke kakve bi dobili od tih resursa ( ovdje se podrazumijeva da znamo kako se ponašaju željeni resursi i da znamo kakve podatke ćemo dobiti ).

Oponašajući objekt kao generalni pojam se može podijeliti na 3 vrste objekata:

- Lutka (*eng. Dummy object*)
- Lažni objekt (*eng. Fake object*)
- Oponašajući objekt (*eng. Mock object*)

### 6.2.1. LUTKA

Najjednostavnija vrsta oponašajućih objekata. Oponašaju eksterne resurse na način da se kod poziva njihove metode vraća unaprijed definirana vrijednost. Ne mogu na temelju ulaznih podataka vraćati različite vrijednosti. Služe ako programerima ne treba veća funkcionalnost od povratne vrijednosti.

```
class TestTrokut(unittest.TestCase):
    def test_jednakostranican(self):
        trokut = Trokut(2,2,2)
        self.assertEqual(Lutka.jednakostranicni(trokut), 1)
```

Slika 6.1

Ako bi imali test kao na slici 6.1 u kojoj želimo testirati da li je trokut jednostraničan, a ne želimo pozivati neku eksternu metodu koja provjerava da li je trokut jednakostraničan ili nam nema potrebe da sami implementiramo cijelu takvu metodu. Napravimo objekt lutku koja nam samo vraća vrijednost 1 ( istinu ), da je trokut jednakostraničan.

```
class Lutka:
    @staticmethod
    def jednakostranicni(self, trokut=None):
        return 1
```

Slika 6.2

Slika 6.2 prikazuje klasu *Lutka* koja ima metodu *jednakostranicni* koja ako se pozove vraća vrijednost 1. Nama je takva metoda dovoljna jer želimo testirati samo jedan primjer u kojem je trokut jednakostranični.

### 6.2.2. LAŽNI OBJEKT

Objekti kojima metode vraćaju vrijednost na temelju ulaznog podatka, ali koje same po sebi nemaju kompliciranu funkcionalnost. Ne može pratiti koliko puta i kojim redoslijedom su pozvane metode.

```
class Lazni:
    @staticmethod
    def baza_podataka(index):
        if(index==1):
            return "Marko"
        if(index==2):
            return "Pero"
        if(index==3):
            return "Ivona"
```

Slika 6.3

Na slici 6.3 je prikazan primjer lažnog objekta koji glumi jednostavnu bazu podataka. Na temelju ulaznog podatka ( *indexa* ) vraća se osoba koja je pod tim *indexom*.

### 6.2.3. OPONAŠAJUĆI OBJEKT

Oponašajući objekti su slični lažnim objektima ali sadrže veću kompleksnost koda. Imaju implementirana pravila u kojem redoslijedu metode moraju biti pozvane i kako moraju reagirati na temelju ulaznog podatka. Mogu pratiti koliko puta je koja metoda pozvana. Na temelju svojeg sadržaja mogu reagirati drugačije u različitim situacijama. Oponašajući objekt zbog ovih razloga zahtjeva konkretno znanje kako se ponaša klasa koju oponaša.

```
class Oponasajuci:

    def __init__(self):
        self.prvi=0
        self.drugi=0
        self.kraj=0

    def vrati_broj(self,broj):
        if(broj%2==0):
            return "paran"
        else:
            return "neparan"

    def pozovi_prvo(self,broj):
        if(self.drugi>0 or self.prvi>0 ):
            raise Exception("prva metoda je vec bila pozvana")
```

```
        else:
            self.prvi=1
        return self.vrati_broj(broj)

    def pozovi_me_dva_puta(self,broj):
        if(self.prvi==0):
            raise Exception("Prije ove metode mora se pozvati prva
            metoda")
        if(self.kraj>1):
            raise Exception("Pozivate drugu metodu nakon zadnje
            metode")
        self.drugi+=1
        return self.vrati_broj(broj)

    def pozovi_za_kraj(self,broj):
        if(self.prvi<1):
            raise Exception("Niste pozvali prvu metodu")
        if(self.drugi<2):
            raise Exception("Niste dovoljno puta pozvali drugu
            metodu")
        self.kraj=1

        return self.vrati_broj(broj)
```

Slika 6.4

Na slici 6.4 je primjer oponašajućeg objekta. Oponaša objekt koji ima metodu koja vraća da li je dobiveni broj paran ili neparan. Ima pravilo pozivanja metoda, kako bi se metode morale pozvati u odgovarajućem redoslijedu, ako se ne pozovu u odgovarajućem redoslijedu prevoditelj će izbaciti odgovarajuću iznimku.

```
class Test_oponasajuci(unittest.TestCase):
    def test_metoda(self):
        op = Oponasajuci()
        print(op.pozovi_prvo(3))
        op.pozovi_me_dva_puta(2)
        op.pozovi_me_dva_puta(15)
        op.pozovi_za_kraj(7)
        print(op.vrati_broj(3))
```

Slika 6.5

Na slici 6.5 je jednostavan test kako bi se trebala koristiti klasa *Oponasajuci*. Prvo se jednom mora pozvati metoda *pozovi_prvo()*, nakon nje se dva puta mora pozvati metoda *pozovi_me_dva_puta()* i nakon toga se za kraj mora pozvati metoda *pozovi_za_kraj()*, s time da svaka metoda prima jedan broj, za koji će reći da li je paran ili neparan.

## 6.3. ZAPISI U STRINGU

Zapisi u stringu su najpopularnija metoda testiranja da li se metode pozivaju u željenom redoslijedu. Zapis se ostvaruje tako da svaka metoda dodaje u string svoj opis. Konkatenacijom zapisa metoda dobivamo rezultat koji na kraju možemo iščitati ili pomoću *assert()* funkcije provjeriti da li su metode pozvane u željenom redoslijedu.

```
class povecajUlaz:
    def __init__(self,ulaz):
        self.log = "konstruktor "
        self.ulaz = ulaz
    def povecajZaJedan(self):
        self.ulaz = self.ulaz + 1
        self.log = self.log + "povecanZaJedan "
    def pomnoziZaDva(self):
        self.log = self.log + "pomnoziZaDva"
```

Slika 6.6

Na slici 6.6 imamo jednostavnu klasu *povecajUlaz* koja kod inicijalizacija prima neki broj i ima metode *povecajZaJedan()* koja dani broj poveća za jedan i *pomnoziZaDva()* koja dani broj pomnoži za dva i nakon toga se sprema dobiveni broj. Svaka objekt ima *log* string u koji se sprema koja je metoda pozvana.

```
class testPovecanje(unittest.TestCase):
    def test_povecanja(self):
        test=povecajUlaz(3)
        test.povecajZaJedan()
        test.pomnoziZaDva()
        assert("konstruktor povecanZaJedan pomnoziZaDva" == test.log)
```

Slika 6.7

Na slici 6.7 testiramo da li su se željene metode pozvale u odgovarajućem redoslijedu. Mi testiramo da li je bio prvo pozvan konstruktor, te nakon njega da li se broj prvo povećao za 1 i nakon toga pomnožio sa dva.

Zapisi u stringu su korisni kod implementacije *Observer* objekta i ako želimo da nam notifikacije dolaze u određenom redoslijedu.

## 6.4. TESTIRANJE PROBLEMATIČNOG KODA

Pod problematičnim kodom najčešće se misli na iznimke koje se mogu dogoditi. Kod treba testirati, kako bi se utvrdilo da li dobro radi u slučaju iznimaka. Ako ne testiramo problematične dijelove tada ne možemo biti sigurni da će krajnji produkt raditi.

```
class dijeljenje:
    def __init__(self,brojnik,nazivnik):
        self.brojnik = brojnik
        self.nazivnik = nazivnik

    def dijeli(self):
        try:
            return self.brojnik/self.nazivnik

        except ZeroDivisionError as err:
            print('Nazivnik ne smije biti nula. Poruka greske:', err)
```

Slika 6.8

Na slici 6.8 imamo klasu *dijeljenje* koja prima dva broja i ima metodu *dijeli()* koja podijeli dane brojeve i može hvatati iznimku. Ako je nazivnik (djelitelj) jednak nuli program izbacuje iznimku.

```
class tetDijeljenje(unittest.TestCase):
    def test_dijeli(self):
        test = dijeljenje(7,0)
        test.dijeli()
```

Slika 6.9

Na slici 6.9 testiramo hoće li iznimka biti dobro uhvaćena ako dijelimo sa 0 ili će se dogoditi nešto neočekivano.

```
test_dijeli (__main__.tetDijeljenje) ...
Nazivnik ne smije biti nula. Poruka greske: division by zero
ok

----------------------------------------------------------------------
```

Slika 6.10

Na slici 6.10 je prikazan rezultat nakon što je pokrenut kod sa slike 6.9 . Vidimo da se metoda *dijeli()* u klasi *dijeljenje* dobro implementirala i da hvata iznimku kada bi htjeli dijeliti s nulom.

## 6.5. SAVJETI ZA PROGRAMERE

Savjeti Kent Becka u knjizi [2] za programere su sljedeći:

- Ako osoba sama programira, zadnji test prije nego što završi za taj dan treba ostaviti tako da ne prolazi ( ostaje na *red baru*). Ovo omogućava programeru da kada sljedeći puta kreće raditi na programu vraća si tok misli tamo gdje je stao i lakše će se sjetiti što je želio napraviti.
- Kod programiranja u timu prije završetka rada treba pustiti sve testove i trebali bi svi testovi proći. Ovo je malo u kontradikciji s prvim savjetom, ali ovdje se radi o timskom radu i programer bi morao imati kod koji radi. Pogotovo ako više ljudi radi na istome dijelu kodu, najbitnije je da kada se krene programirati da programeri imaju „čisti početak“ ( nastavljaju pisati kod, a dotadašnji kod prolazi sve testove)

# 7. OBRASCI ZA DRUGU FAZU ( eng. *Green Bar Patterns* )

Nakon što smo završili s prvom fazom (eng. *Red bar*) i imamo test koji ne prolazi želja nam je da što prije dođemo do druge faze u kojoj će nam testovi proći. Ovi obrasci pomažu kako bi se što prije došlo do druge faze, iako neki rezultati dolaska do druge faze neće biti „najljepši“ to se sve može popraviti u trećoj fazi koja je refaktorizacija.

## 7.1. LAŽIRANJE

Nakon što napišemo test koji ne prolazi, najlakši način da prođe je da se u metodi koju test poziva implementira da vraća konstantu. S konstantom dobivamo da nam test prolazi, a nakon toga postepeno mijenjamo konstantu .

Bolje je lažirati neku vrijednost u početku i osigurati prolaz testa nego krenuti odmah sa kompleksnijim implementacijama. S lažiranjem osiguravamo da nam test odmah prolazi i postepenom implementacijom koda smanjujemo mogućnost pogreške.

```
class testDjelitelji(unittest.TestCase):
    def test_broj_djelitelj1(self):
        broj=Djelitelji(6)
        rezultat = broj.koliko()
        assert ( 4 == rezultat )
```

Slika 7.1

Najjednostavniji način kako bi se prošao test sa slike 7.1 je da metoda *koliko()* u klasi *Djelitelji* na početku vraća konstantu 4. Pod brojem djelitelja smatramo sve brojeve koji dijele zadani broj, uključujući njega samog i broj 1.

```
class Djelitelji:
    def __init__ (self, broj):
        self.broj = broj
    def koliko(self):
        return 4
```

Slika 7.2

Na slici 7.2 vidimo najlakši način implementacije klase *Djelitelji* koji prolazi test sa slike 7.1. Nakon što nam test prođe mijenjamo konstantu 4 u metodi *koliko()* s varijablom i

nakon toga implementiramo funkcionalnost da nam metoda *koliko()* vraća točan broj djelitelja.

```
class Djelitelji:
    def __init__ (self, broj):
        self.broj = broj
    def koliko(self):
        self.broj_djelitelja = 0

        for i in range (1,self.broj+1):
            if(self.broj%i is 0):
                self.broj_djelitelja +=1
        return self.broj_djelitelja
```

Slika 7.3

Slika 7.3 prikazuje kako će klasa *Djelitelji* izgledati na kraju.

## 7.2. TRIANGULIRANJE (eng. *Triangulate*)

Apstrakcije u klasi se implementiraju tek nakon što imamo 2 ili više testna primjera.

Trianguliranje se svodi na tri dijela :

1. Pisanje jednog testnog primjera i jednostavne metode koja će proći test
2. Dodavanje različitog testnog primjera (različiti brojevi i sl.)
3. Apstrakcija metoda da prolazi sve testove

U sljedećem primjeru želimo napraviti klasu koja prima 2 broja i vraća njihov umnožak.

```
class test_umnozak(unittest.TestCase):
    def testUmnozak(self):
        pomnozi = Mnozenje()
        self.assertEqual(6,pomnozi.puta(2,3))

class Mnozenje:
    def puta(self,a,b):
        return 6
```

Slika 7.4

Korištenjem prvog pravila trianguliranja prvo smo napravili jedan testni primjer koji treba pomnožiti dva broja ( u našem primjeru to su 2 i 3). I imamo klasu *Mnozenje* koja

ima jednostavnu metodu *puta()* koja mnozi dva broja i vraća rezultat. Na početku nam je bitno da je metoda sto jednostavnija pa ne stavljamo funkcionalnosti u nju nego smo samo napravili da vraća broj 6 ($2 * 3 = 6$).

```
class test_umnozak(unittest.TestCase):
    def testUmnozak(self):
        pomnozi = Mnozenje()
        self.assertEqual(6,pomnozi.puta(2,3))
        self.assertEqual(21,pomnozi.puta(3,7))
```

Slika 7.5

Na slici 7.5 se vidi korištenja drugog pravila trianguliranja i dodavanja još jednog primjera koji je različit od prvog. Kada bismo nakon dodavanja još jednog primjera probali kompilirati program, javila bi nam se pogreška. Metoda *puta()* nije apstraktna nego je implementiranja za samo dva konkretna broja.

```
class Mnozenje:
    def puta(self,a,b):
        return a*b
```

Slika 7.6

Na slici 7.6 se koristi treće pravilo trianguliranja u kojoj proširujemo zadanu metodu *puta()* kako bi je mogli koristiti za bilo koja dva broja.

## 7.3. OČITA IMPLEMENTACIJA

Kod implementacije jednostavnih metoda nije potrebno koristiti metode kao što su lažiranje i triangulacija. Metoda očite implementacija se svodi da programer samo implementira u kodu što misli da će raditi. Npr. kada bi se implementirala klasa *Mnozenje* s metodom *puta()* koja vraća umnožak dva broja, nije potrebno koristiti triangulaciju i sve njene korake nego se može odmah napisati kod kao na slici 7.6.

Prednosti ove metode su što ne zahtjeve nikakve dugačke korake i programer može brzo implementirati željenu metodu.

Negativne strane ove metode su da se često događaju pogreške i kod najjednostavnijih implementacija. Programeri mogu zapeti na trivijalnim stvarima koristeći ovu metodu i time izgubiti dosta vremena. Pogreške kod očite implementacije ako nisu odmah vidljive dovode do frustracije i nervoze programera.

## 7.4. KOLEKCIJA OBJEKATA

Kod implementacije funkcija (metoda) koje koriste skupove objekata, prvo se funkcija implementira bez skupa, a nakon toga se refaktorizira da radi sa skupom objekata. Ovo je slično metodi lažiranja jer kada implementiramo funkciju prvo lažiramo njenu povratnu vrijednost ili njenu funkcionalnost.

Metodu kolekcije objekata ćemo pokazati na primjeru funkcije koja prima skup brojeva i vraća njihovu sumu.

```
def suma_skupa(skup):
    suma = skup
    return suma
```

Slika 7.7

Na slici 7.7 se vidi da smo prvo implementirali funkciju *suma_skupa()* koja kao ulaz ima jednu varijablu *skup*. Funkcija *suma_skupa()* vraća sumu svih elemenata skupa kojeg prima, a kako u ovoj pojednostavljenoj verziji prima samo varijablu, tada i vraća samo vrijednost te varijable. Sljedeće refaktoriziramo funkciju kako bi radila ako je varijabla *skup* skup brojeva.

```
def suma_skupa(skup):
    suma=0
    duljina = len(skup)
    for i in range (0,duljina):
        suma+=skup[i]

    return suma
```

Slika 7.8

Funkcije *suma_skupa()* je prikazana na slici 7.8. Ova metoda omogućava da se malim koracima dođe do željene funkcionalnosti.

# 8. OBRASCI ZA TREĆU FAZU – REFAKTORIZACIJA

Kod ovih metoda nećemo mijenjati semantiku programa. Opisivati ćemo metode kako mijenjati oblikovanje sustava, koristeći male korake. Najbitnija stavka kod refaktorizacije je da nam neovisno o promjenama u kodu testovi prolaze.

Kod refaktorizacije koda ako uočimo manjak neke metode ili određenu problematiku neovisno o tome da li nam svi testovi prolaze, pišemo nove testove vezane za novou problematiku.

## 8.1. UJEDINJENJE KODA

Kod dijelova koda koji slično izgledaju ili ima sličnu funkcionalnost, može ih se probati ujediniti postepenim mijenjanjem. Mogu se ujediniti samo ako su identični i ako se njihovim ujedinjenjem ne narušavaju ostale funkcionalnosti. Ovakve promjene se ne preporučuju ako nije intuitivno kako bi se dijelovi koda mogli ujediniti (želimo male korake i brzo i jednostavno pisanje koda).

Dijelovi koda koji se ujedinjuju i što se dobije ako se mogu refaktorizirati da budu identični:

- Dvije slične petlje – mogu se ujediniti.
- Dvije slične grane kondicionala su slične – može se maknuti kondicional.
- Dvije metode su slične – može se maknuti jedna.
- Dvije klase su slične – može se maknuti jedna.

Ponekad je potrebno kod ujedinjenja koda razmišljati na drugačiji način. Ako želimo npr. više potklasa spojiti u jednu klasu, nećemo ih raditi sve identičnima i onda brisati sve osim jedne. Prvo ćemo napraviti jednu klasu u koju ćemo ih sve ujediniti i nakon toga ćemo tu klasu popunjavati sa željenim metodama. Kako stavljamo metode u klasu tako ih brišemo iz ostalih klasa. Jednom po jednom metodom popunimo novu klasu a ostale potklase praznimo i kada potklase budu prazne znamo da smo gotovi.

## 8.2. IZOLIRANA PROMJENA

Ako želimo promijeniti samo dio metode koja ima više dijelova, prvo što napravimo je izoliramo taj dio i nakon toga gledamo koji je najbolji način da se on promjeni.

Najčešći primjer izoliranih promjena jesu mijenjanje konstantne vrijednosti s nekom određenom koju vraća funkcija. Tada se može ostaviti isto ime varijable, da se ne mijenja u cijelom kodu, nego se samo njena vrijednost mijenja tamo gdje je potrebno ( pozivom funkcije ).

## 8.3. SMANJENJE METODA

Kako bi velike i komplicirane metode bile lakše čitljive jedan dio njihove funkcionalnosti se može prebaciti u posebnu metodu i tu metodu onda pozvati.

Ovdje ćemo opisati općeniti način uporabe ove metode:

1. Nađemo dio u metodi koji bi imalo smisla implementirati kao zasebnu metodu. Najbolji primjeri su petlje i grane kondicionala.
2. Moramo osigurati da varijable koje se koriste u dijelu koda od kojeg želimo napraviti metodu se ne koriste van njega.
3. Kopiramo kod iz stare metode u novu metodu i prevodimo ga.
4. Obrišemo kod iz stare metode i umjesto njega stavimo novu metodu.
5. Prevedemo kod.

Općenito se preporučuje imati što manje metode kako bi bile što čitljivije ali se mora paziti da se ne pretjera sa smanjenjima jer tako možemo imati previše metoda i opet će kod biti teže čitljiv.

## 8.4. LINIJSKA METODA

Metoda ( kao funkcionalnost ) može postati teška za čitanje i razumijevanje ako poziva previše drugih metoda. Mogu se pozivati metode koje imaju veću funkcionalnost nego što je nama potrebno. U ovakvim slučajevima se pozivi drugih metoda mijenjaju samo sa željenim kodom.

Način korištenja linijske metode:

1. Kopirati željenu metodu ili dio potrebne funkcionalnosti
2. Obrisati poziv metode i na to mjesto zalijepiti kopiranu metodu.
3. Zamijeniti potrebne varijable kako bi kod radio.

Dobra praksa kod linijske metode je korištenje lambda izraza, s njima se može još malo pojednostaviti kod, a da bude čitljiv.

## 8.5. DODAVANJE SUČELJA *(eng. Interface)*

Ako ćemo imati više sličnih klasa ( imaju metode istog imena i sličnog ponašanja) poželjno je dodavanje sučelja koje će klase naslijediti. Sučelje nam pomaže da se ne zaboravi implementirati neka metoda.

Sučelje se može dodati i naknadno ako nismo odmah znali da ćemo imati više sličnih klasa. Ako sučelje naknadno dodajemo koriste se sljedeći koraci:

1. Deklariranje sučelja. Ponekad će ime postojeće klase biti najbolje ime za sučelje pa se mijenja ime klase.
2. Postojeća klase nasljeđuje sučelje.
3. Dodavanje potrebnih metoda u sučelje.

Sučelja u Pythonu se razlikuju od sučelja u C++, Javi i sličnim programskim jezicima. Najveća razlika je u tome što kod Jave ako se ne implementiraju metode iz sučelja kod se neće kompilirati, dok će kod Pythona to proć. Tu se postavlja pitanje zašto onda koristiti sučelja kod Pythona? Glavni razlog je dokumentacija. U razvoju softvera vođenog testiranjem sam kod je dovoljna dokumentacija, sučelja osiguravaju preglednost koda i lakše snalaženje u njemu.

## 8.6. PREMJEŠTANJE METODA NA PRAVO MJESTO

Kod pisanja metoda nerijetki su slučajevi da se metoda stavila na krivo mjesto ili da se kasnije otkrilo kako bi određenoj metodi bilo bolje da je na nekom drugom mjestu. Najčešće se metode premještaju iz jedne klase u drugu.

Metode se premještaju na sljedeći način:

1. Kopira se željena metoda.
2. Zalijepi se na odgovarajuće mjesto ( u odgovarajuću klasu ), promijeni se njeno ime ako je to potrebno. Kompilira se.

Nije preporučljivo premještati metode ako koriste varijable klase u kojoj se nalaze ili ako će se njenim premještanjem utjecati na druge metode unutar klase.

## 8.7. PRETVARANJE METODA U OBJEKT

Metode koje koriste više parametara i lokalnih varijabli mogu se napisati u obliku objekta.

Pretvaranje metode u objekt:

1. Stvaramo objekt s istim parametrima kao i metoda.
2. Lokalne varijable koje koristi metoda stavljamo u objekt.
3. Stvaramo metodu *run(),* funkcionalnost te metode je ista kao i originalna metoda.
4. Umjesto korištenja originalne metode, stvaramo novi objekt i pokrećemo *run().*

Pretvaranje metoda u objekt dodaje novu logiku sustavu i pojednostavljuje kod. Lako se novome objektu dodaju metode koje proširuju sadašnju funkcionalnost. Tako olakšavamo razvoj daljnjeg koda.

## 8.8. DODAVANJE I PREMJEŠTANJE PARAMETARA

Dodavanje parametra u metodu:

1. Ako je metoda sučelje, dodati prvo parametar u sučelje.
2. Dodaj parametar.
3. Koristeći pogreške koje javlja kompajler, promijeniti kod kako bi radio.

Dodavanje parametara se koristi kada se uvidi da moramo proširiti metodu i dodati joj još neku funkcionalnost. Parametar se može premjestiti iz jedne metode u drugu. Kod takvog premještanja, prvo se doda parametar, tada se izbriše stari parametar na svim mjestima gdje se koristio i nakon toge se izbriše i on sam.

Premještanje parametra iz metode ili metoda u konstruktor:

1. Dodajemo parametar u konstruktor.
2. Dodajemo varijablu s istim imenom kao i parametar.
3. Stavljamo varijablu u konstruktor.
4. Jedan po jedan, preimenujemo reference od „parametar“ u „self.parametar“ (ovaj dio zavisi u kojem programskom jeziku radimo) u metodama.
5. Kada više ne bude bilo referenci na parametar, izbrišemo parametar iz metode.
6. Mičemo „self.“ Iz referenci.
7. Preimenujemo korektno varijable.

Ako prosljeđujemo isti parametar u više metoda istog objekta, tada možemo maknuti duplikaciju koda tako da parametar samo jednom proslijedimo.

# 9. STUDIJSKI PRIMJER

Za studijski primjer ćemo napraviti kalkulator. Krenut ćemo razvijati softver s jednostavnijim operacijama prema kompleksnijim.

Prva operacija koju želimo implementirat će biti zbrajanje dva broja. Prije nego se krenu pisati testovi moraju se osmisliti imena klasa i metoda koje ćemo koristiti. Metoda koja će zbrajati dva broja nazvat ćemo *plus()* a klasu u kojoj će biti će se zvati *aritmetika*.

```
class testAritmetika(unittest.TestCase):
    def test_plus(self):
        calc = aritmetika()
        self.assertEqual(calc.plus(2,3),5)
```

Slika 9.1

Prvo radimo jednostavni test u kojemu želimo koristiti metodu *plus()* i zbrojiti dva broja. Sada se nalazimo u prvoj fazi (*red phase*). Nakon što smo napisali test, prevodimo program kako bi se vidjelo da nam test neće proći.

```
test_plus (__main__.testAritmetika) ... ERROR

======================================================================
ERROR: test_plus (__main__.testAritmetika)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-4-9b10a1c64ea3>", line 3, in test_plus
    calc = aritmetika()
NameError: name 'aritmetika' is not defined

----------------------------------------------------------------------
Ran 1 test in 0.016s

FAILED (errors=1)
```

Slika 9.2

Prevoditelj nam javlja grešku da klasa aritmetika ne postoji, što je i očekivano jer je još nismo implementirali. Sljedeći korak nam je implementirati potrebnu klasu i metodu kako bi nam test prošao.

```
Class aritmetika:
    def plus(self,a,b):
        return 5
```

Slika 9.3

Na slici 9.3 smo definirali klasu *aritmetika*, s metodom *plus()* koja vraća broj 5, jer u testu zbrajamo 2 i 3. Nakon ove implementacije opet prevodimo program kako bi vidjeli da li nam test prolazi.

```
test_plus (__main__.testAritmetika) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

Slika 9.4

Na slici 9.4 vidimo da nam test prolazi i sada se nalazimo u drugoj fazi. Ovdje smo koristili metodu lažiranje, to se vidi po tome što metoda *plus()* zasada vraća konstantu 5. Nakon ove faze kreće treća faza – refaktorizacija. Želimo da nam metoda *plus()* umjesto konstante vraća zbroj argumenata.

```
class aritmetika:
    def plus(self,a,b):
        return a+b
```

Slika 9.5

Na slici 9.5 se vidi promjena u metodi *plus()* koja nam omogućuje da vraća zbroj ulaznih argumenata. Nakon ove promjene opet se program prevodi kako bi bili sigurni da nam test i dalje prolazi.

```
test_plus (__main__.testAritmetika) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Slika 9.6

Nakon ponovnog prevođenja programa kao što vidimo na slici 9.6 test nam prolazi. Možemo dodati još koji test koji provjerava zbrajanje ili se osloniti da će nam proći i ostale sumacije. Napisat ćemo još dva testa za *plus()* gdje ćemo sumirati dva negativna broja i negativan i pozitivan broj.

```
    def test_plus(self):
        calc = aritmetika()
        self.assertEqual(calc.plus(2,3),5)
        self.assertEqual(calc.plus(-4,-5),-9)
        self.assertEqual(calc.plus(-1,3),2)
```

Slika 9.7

Slika 9.7 prikazuje izgled funkcije za testiranje metode *plus()* nakon dodanih testova. Nakon što se prevodi program s ovim izmjenama dobije se rezultat isti kao na slici 9.6.

Sljedeća metoda koju ćemo implementirati je *minus()* koja će primati dva argumenata i vraćati njihovu razliku, od prvoga ćemo oduzimati drugi argument. Implementacija metode *minus()* je analogna implementaciji metodi *plus()*. Kako ne bi iste korake ponavljali budemo samo prikazali konačni izgled metode *minus()*, izgled testa i što se dogodi kada prevedemo program.

```
class aritmetika:
    def plus(self,a,b):
        return a+b

    def minus(self,a,b):
        return a-b
```

Slika 9.8

Na slici 9.8 je prikazana klasa *aritmetika* nakon dodavanja motode *minus()*.

```
    def test_minus(self):
        calc = aritmetika()
        self.assertEqual(calc.minus(2,3),-1)
        self.assertEqual(calc.minus(-4,-5),1)
        self.assertEqual(calc.minus(-1,3),-4)
```

Slika 9.9

Na slici 9.9 je prikazana testna metoda *test_minus()* koja se nalazi u klasi *testAritmetika*. Pomoću nje testiramo metodu *minus()*.

```
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

Slika 9.10

Na slici 9.10 je prikaz koji smo dobili nakon što smo preveli program.

Kao što se može vidjeti na slikama 9.7 i 9.9 kod testova u svakoj funkciji stvaramo objekt *aritmetika*. Kako se ne bi taj objekt inicijalizirao dva puta, a moguće da će se trebati i više puta inicijalizirati kako ćemo razvijati kod, budemo ga jednom inicijalizirali u metodi *SetUp()*.

```
class testAritmetika(unittest.TestCase):
    def setUp(self):
        self.calc = aritmetika()

    def test_plus(self):
        self.assertEqual(self.calc.plus(2,3),5)
        self.assertEqual(self.calc.plus(-4,-5),-9)
        self.assertEqual(self.calc.plus(-1,3),2)
    def test_minus(self):
        self.assertEqual(self.calc.minus(2,3),-1)
        self.assertEqual(self.calc.minus(-4,-5),1)
        self.assertEqual(self.calc.minus(-1,3),-4)
```

Slika 9.11

Na slici 9.11 je prikaz klase *testAritmetika* nakon refaktorizacije koda i uvođenja metode *setUp()*.

Sljedeće implementiramo metode *puta()* i *dijeli()*. Nećemo pisati jednu po jednu metodu jer su dovoljno jednostavne da se mogu zajedno napisati. Ovakva praksa inače nije dobra jer može lako dovesti do greške.

```
    def test_puta(self):
        self.assertEqual(self.calc.puta(2,3),6)
        self.assertEqual(self.calc.puta(-4,-5),20)
        self.assertEqual(self.calc.puta(-1,3),-3)
    def test_dijeli(self):
        self.assertEqual(self.calc.dijeli(6,3),2)
        self.assertEqual(self.calc.dijeli(-8,-4),2)
        self.assertEqual(self.calc.dijeli(-5,1),-5)
```

Slika 9.11 : Testovi za *puta()* i *dijeli()*

```
test_dijeli (__main__.testAritmetika) ... ERROR
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ... ERROR

======================================================================
ERROR: test_dijeli (__main__.testAritmetika)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-79-ca6ff2eadf12>", line 18, in test_dijeli
    self.assertEqual(self.calc.dijeli(6,3),2)
AttributeError: 'aritmetika' object has no attribute 'dijeli'

======================================================================
ERROR: test_puta (__main__.testAritmetika)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-79-ca6ff2eadf12>", line 14, in test_puta
    self.assertEqual(self.calc.puta(2,3),6)
AttributeError: 'aritmetika' object has no attribute 'puta'

----------------------------------------------------------------------
Ran 4 tests in 0.007s

FAILED (errors=2)
```

Slika 9.12 : Javljanje greške nakon prevođenja programa

Kao što se vidi na slici 9.12 prevoditelj javlja pogreške da ne postoje metode *puta()* i *dijeli()*. To je očekivano jer te metode još nisu implementirane.

```
    def puta(self,a,b):
        return a*b
    def dijeli(self,a,b):
        return a/b
```

Slika 9.13: Implementacija metoda *puta()* i *dijeli* u klasi *aritmetika*

```
test_dijeli (__main__.testAritmetika) ... ok
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK
```

Slika 9.14

Metode *puta()* i *dijeli()* su dobro implementirane pa nam svi testovi prolaze. Nakon što smo napravili ove testove trebamo dodati još jedan dodatni test za metodu *dijeli()*. Dodatni test će provjeravati dijeljenje sa nulom. Želimo da nam prevoditelj ispiše poruku „*Dijelite sa nulom*“ ako se dijeli sa nulom i da nam program dalje radi.

```
    def test_dijeli_nula(self):
        self.calc.dijeli(5,0)
```

Slika 9.15

Napisali smo jednostavan primjer dijeljenja sa nulom i nakon toga ćemo prevesti program.

```
test_dijeli (__main__.testAritmetika) ... ok
test_dijeli_nula (__main__.testAritmetika) ... ERROR
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ... ok

======================================================================
ERROR: test_dijeli_nula (__main__.testAritmetika)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-93-020fa93f431f>", line 22, in test_dijeli_nula
    self.calc.dijeli(5,0)
  File "<ipython-input-92-aeccf4e5de43>", line 10, in dijeli
    return a/b
ZeroDivisionError: division by zero

----------------------------------------------------------------------
Ran 5 tests in 0.006s

FAILED (errors=1)
```

Slika 9.16

Kao što se vidi na slici 9.16 prevoditelj nam izbacuje željenu pogrešku (dijeljenje sa nulom), ali nam napisani test ne prolazi i program se uruši kada dođe do dijela dijeljenja sa nulom.

Sljedeće idemo upravljati iznimkama da nam prevoditelj ne bi javljao grešku kao na slici 9.16.

```
def dijeli(self,a,b):
        try:
            return a/b
        except ZeroDivisionError:
            print("Dijelite sa nulom")
            return 0
```

Slika 9.17

Slika 9.17 prikazuje metodu *dijeli()* nakon što smo je promijenili da hvata iznimku kod dijeljenja sa nulom. Ako se dijeli s nulom stavili smo da se vraća nula kako se ne bi narušio kod ako se nekoj varijabli pridružuje povratna vrijednost metode *dijeli()*.

```
test_dijeli (__main__.testAritmetika) ... ok
test_dijeli_nula (__main__.testAritmetika) ... ok
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ...
Dijelite sa nulom
ok

----------------------------------------------------------------------
Ran 5 tests in 0.006s

OK
```

Slika 9.18

Nakon što smo preveli program vidimo da nam svi testovi prolaze i ispisuje nam se poruka da dijelimo sa nulom.

Kao što smo do sada vidjeli svaki puta kada smo pokretali testove, nismo pokretali jedan po jedan nego sve odjednom. I kada se dodao novi test i pokrenuo da vidimo da neće proći, pokrenuli su se i ostali kako bi se uvjerili da se samim dodavanjem novoga nije nešto narušilo u programu.

Nastavljajući sa ovom praksom ćemo implementirati klase pravokutnik i krug koje će imati metode da izračunavaju i vraćaju vrijednosti opsega i površine. Prvo ćemo implementirati klasu *pravokutnik*. Naravno sljedeći pravila razvoja softvera vođenog testiranjem prije implementacije klase i metoda pišemo test.

```
class testGeometrijskiLikovi(unittest.TestCase):
    def setUp(self):
        self.pravokut = pravokutnik(2,4)
    def testPravokutnik(self):
        self.assertEqual(self.pravokut.opseg(),12)
        self.assertEqual(self.pravokut.povrsina(),8)
```

Slika 9.19

Želimo da nam klasa *pravokutnik* ima konstruktor koji će primati dva broja (duljine stranica pravokutnika). Ovdje se može prvo pisati test za opseg pravokutnika, a nakon toga za površinu pravokutnika, ali nema potreba za time s obzirom na to da su metode *opseg()* i *povrsina()* jednostavne za implementirati. Znamo da ako sada pokrenemo test da on neće proći i da će prevoditelj javiti pogrešku da ne postoji klasa *pravokutnik*.

```
class pravokutnik:
    def __init__ (self,a,b):
        self.a=a
        self.b=b
    def opseg(self):
        return 2*self.a+2*self.b
    def povrsina(self):
        return self.a*self.b
```

Slika 9.20

Na slici 9.20 je implementacija klase *pravokutnik* kakvu smo željeli.

```
test_dijeli (__main__.testAritmetika) ... ok
test_dijeli_nula (__main__.testAritmetika) ... ok
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ... ok
testPravokutnik (__main__.testGeometrijskiLikovi) ...
Dijelite sa nulom
ok

----------------------------------------------------------------------
Ran 6 tests in 0.006s

OK
```

Slika 9.21

Nakon prevođenja programa vidimo na slici 9.21 da smo dobro implementirali klasu *pravokutnik*. Na slici 9.21 se može vidjeti kraj testnih metoda u kojoj se testnoj klasi nalaze.

Sljedeće ćemo implementirati klasu *krug* koja će također kao i klasa *pravokutnik* imati metode *opseg()* i *površina()*.

Pretpostavit ćemo da će se klasa *krug* spajati na bazu podataka kako bi iz nje dobila vrijednost $\pi$. Zbog lakše provjere vrijednost $\pi$ ćemo računati da je 3.14 . Klasu *krug* nećemo spajati na bazu podataka, nego ćemo napraviti objekt koji će oponašati željenu bazu podataka.

*Krug* će imati konstruktor koji će primati vrijednost polumjera.

```
    def testKrug(self):
        r = 3
        o = 2*r*3.14
        p = r*r*3.14
        self.Krug = krug(3)
        self.assertEqual(self.Krug.opseg(),o)
        self.assertEqual(self.Krug.povrsina(),p)
```

Slika 9.22

Na slici 9.22 nam je prikazan test kojim ćemo provjeravati točnost metoda *opseg()* i *povrsina()*. Varijabla *o* označava opseg, *p* označava površinu i *r* označava polumjer.

Nakon prevođenja dobijemo očekivanu poruku da klasa *krug* nije implementirana. Sljedeće ćemo prikazati izgled klase *krug*.

```
class BazaPodataka:
    def PI(self):
        return 3.14
```

Slika 9.23: Lažni objekt

Prije nego što krenemo implementirati klasu *krug* implementirali smo klasu *BazaPodataka* koja nam je lažni objekt. Ona će oponašati bazu podataka iz koje uzimamo vrijednost $\pi$.

```
class krug:
    def __init__(self,r):
        self.r = r
        baza = BazaPodataka()
        self.pi = baza.PI()

    def opseg(self):
        return 2*self.r*self.pi
    def povrsina(self):
        return self.r*self.r*self.pi
```

Slika 9.24: Implementacija klase *krug*

```
test_dijeli (__main__.testAritmetika) ... ok
test_dijeli_nula (__main__.testAritmetika) ... ok
test_minus (__main__.testAritmetika) ... ok
test_plus (__main__.testAritmetika) ... ok
test_puta (__main__.testAritmetika) ... ok
testKrug (__main__.testGeometrijskiLikovi) ... ok
testPravokutnik (__main__.testGeometrijskiLikovi) ...
Dijelite sa nulom
ok

----------------------------------------------------------------------
Ran 7 tests in 0.008s

OK
```

Slika 9.25

Nakon što smo pokrenuli prevoditelj, kao što se vidi na slici 9.25 dobro smo implementirali metode u klasi *krug*.

Kao što se može vidjeti klase *krug* i *pravokutnik* imaju iste metode pa bi bilo dobro kada bi svaka klasa naslijedila sučelje koje ima metode *opseg()* i *povrsina()*. Sučelje će se koristiti ako će se naknadno dodavati klase za koje ćemo htjeti da imaju navedene metode, te kako se one ne bi zaboravile implementirati. *Python* nema sučelja kao što imaju drugi programski jezici (npr. C++).

U *Pythonu* se može napraviti sučelje ali klasa koja nasljeđuje to sučelje nije obavezna implementirate metode koje se nalaze u sučelju. S time se gubi smisao raditi sučelje, jer lako zaboravimo implementirati neku metodu, a program će se prevesti bez da nam javi pogrešku.

Mi ćemo implementirati klasu *geometrijskiLik*. Ima jedan „trik“ kako će klasa imati željeno svojstvo da ako ga klasa naslijedi i ako ne implementira neku metodu da nam se javi pogreška.

```
class geometrijskiLik:
    def opseg(self):
        raise NotImplementedError
    def povrsina(self):
        raise NotImplementedError
```

Slika 9.26

Na slici 9.26 vidimo klasu *geometrijskiLik* koja će nam predstavljati sučelje. Klase *pravokutnik* i *krug* ćemo refaktorizirati tako da će naslijediti ovu klasu. Klasa ima metode *opseg()* i *povrsina()* koje izbacuju iznimku *NotImplementedError* ako se pozovu. Svaka klasa koja će naslijediti ovu klasu i ako neće implementirati metode *opseg()* i *povrsina()*, a budu se pozvale prevoditelj će nam izbaciti iznimku. S obzirom na to da prvo pišemo

testove znamo da će se željene metode pozvati, a ako se ne implementiraju da će se aktivirati iznimka.

```
import unittest

class aritmetika:
    def plus(self,a,b):
        return a+b
    def minus(self,a,b):
        return a-b
    def puta(self,a,b):
        return a*b
    def dijeli(self,a,b):
        try:
            return a/b
        except ZeroDivisionError:
            print("Dijelite sa nulom")
            return 0

class geometrijskiLik:
    def opseg(self):
        raise NotImplementedError
    def povrsina(self):
        raise NotImplementedError

class pravokutnik(geometrijskiLik):
    def __init__ (self,a,b):
        self.a=a
        self.b=b
    def opseg(self):
        return 2*self.a+2*self.b
    def povrsina(self):
        return self.a*self.b

class BazaPodataka:
    def PI(self):
        return 3.14

class krug(geometrijskiLik):
    def __init__(self,r):
        self.r = r
        baza = BazaPodataka()
        self.pi = baza.PI()

    def opseg(self):
        return 2*self.r*self.pi
    def povrsina(self):
        return self.r*self.r*self.pi
```

Slika 9.27

Na slici 9.27 je prikazan kod od svih klasa i njihovih metoda, kako bi se vidio cijeli kod programa na jednome mjestu.

```
class testAritmetika(unittest.TestCase):
    def setUp(self):
        self.calc = aritmetika()

    def test_plus(self):
        self.assertEqual(self.calc.plus(2,3),5)
        self.assertEqual(self.calc.plus(-4,-5),-9)
        self.assertEqual(self.calc.plus(-1,3),2)
    def test_minus(self):
        self.assertEqual(self.calc.minus(2,3),-1)
        self.assertEqual(self.calc.minus(-4,-5),1)
        self.assertEqual(self.calc.minus(-1,3),-4)
    def test_puta(self):
        self.assertEqual(self.calc.puta(2,3),6)
        self.assertEqual(self.calc.puta(-4,-5),20)
        self.assertEqual(self.calc.puta(-1,3),-3)
    def test_dijeli(self):
        self.assertEqual(self.calc.dijeli(6,3),2)
        self.assertEqual(self.calc.dijeli(-8,-4),2)
        self.assertEqual(self.calc.dijeli(-5,1),-5)
    def test_dijeli_nula(self):
        h=self.calc.dijeli(5,0)

class testGeometrijskiLikovi(unittest.TestCase):
    def setUp(self):
        self.pravokut = pravokutnik(2,4)
    def testPravokutnik(self):
        self.assertEqual(self.pravokut.opseg(),12)
        self.assertEqual(self.pravokut.povrsina(),8)

    def testKrug(self):
        r = 3
        o = 2*r*3.14
        p = r*r*3.14
        self.Krug = krug(3)
        self.assertEqual(self.Krug.opseg(),o)
        self.assertEqual(self.Krug.povrsina(),p)
```

Slika 9.28

Slika 9.28 prikazuje sve testove koje smo implementirali za naš program.

Nakon što prevedemo program na kraju dobit ćemo rezultat kao što je na slici 9.25.

U ovome studijskome primjeru smo koristili jednostavnije metode i neke primjere smo koristili ranije u ovome radu kako bi se približila određena metoda. Metode i klase su se ovdje ponovo koristile kako bi se moglo vidjeti kako će izgledati ako se koriste u većem projektu. Ovaj studijski primjer je zamišljen kako bi se pokazala ideja razvoja softvera vođenog testiranjem i kako bi se vidjelo kako izgleda takav način izrade softvera.

# ZAKLJUČAK

Cilj ovog diplomskog rada „Razvoj softvera vođen testiranjem“ je bio pobliže opisati istoimenu metodu razvoja softvera, opisati način korištenja aplikacijskih okvira za testiranje dijelova i navesti i opisati obrasce koji se koriste.

Većina programskih jezika ima svoje aplikacijske okvire koji omogućavaju jednostavnije testiranje dijelova. Prednost aplikacijskih okvira je što su jednostavni za korištenje i metode su slične, ako ne i iste u programskim jezicima koji ih imaju.

Programerima koji se nisu prije susreli s metodom razvoja softvera vođenog testiranjem, ova metoda će u početku stvarati probleme jer se teško naviknuti prvo pisati testove, a nakon toga kod. Obrasci koji postoje pomažu programerima koji su tek počeli koristiti ovu metodu kao i programerima koji ju koriste dulje vremena. Obrasci su jednostavni za korištenje i većina je logična, ali su nedostaci što ih ima puno i programer bi morao stalno razmišljati koja se metoda da iskoristiti ili ako se može odjednom više metoda iskoristiti treba znati kojim redoslijedom ih koristiti.

Razvoj softvera vođen testiranjem je dobra metoda ako je za dokumentaciju dovoljan kod. Najveći nedostatak ovoj metodi je što pisanje testova uzima skoro jednako vremena kao i pisanje metoda koje ih prolaze.

Metoda razvoja softvera vođenog testiranjem je gotovo postala standard za razvoj manjih softverskih projekata, ali razvojem aplikacijskih okvira i objavljivanjem sve više raznovrsne literature o toj metodi, ona se kreće više koristiti u većim projektima te bi u skoroj budućnosti mogla postati metoda koja se najviše koristi u izradi softvera.

# LITERATURA


[1]. J.Bender, J. McWherter, *Professional Test Driven Development with C#,* Wiley Publishing, Indianapolis, 2011

[2]. K.Beck, *Test-Driven Development By Example*, Addison Wesley, Boston, 2002

[3]. Python dokumentacija, dostupno na *docs.python.org*

[4]. Robert Manger, *Softversko Inžinjerstvo*, Element, Zagreb, 2016.

[5]. Harry Percival, *Test-Driven Web Development with Python*, O'Reilly Media, Sebastopol, 2014

[6]. David Beazley, Brian K. Jones, *Python Cookbook, Third Edition,* O'Reilly Media, Sebastopol, 2014

[7]. Tihomir Katić, Trendovi razvoja aplikacija, dostupno na: http://161.53.18.5/zpr/Portals/0/Predmeti/MIT/Trendovi%20razvoja%20aplikacija.pdf (listopad 2016)

# SAŽETAK

Razvoj softvera vođen testiranjem se krenuo koristiti radi želje programera da se u programu implementiraju samo potrebne metode i kako bi im testovi na vrijeme otkrili da li trenutna verzija programa radi ono što treba. Najčešće se ovaj način razvoja softvera koristi u manjim projektima, ali zbog sve veće popularnosti kreće se sve više koristiti u većim projektima.

Razvoj softvera vođen testiranjem se sastoji od tri faze. Prva faza se sastoji od pisanja testova za metode koje još nisu implementirane. U drugoj fazi pišemo metode koje će proći napisane testove. U trećoj fazi se brinemo o kvaliteti koda, jednostavnosti održavanja i jednostavnosti čitanje koda.

Kako se razvoj softvera vođen testiranjem krenuo sve više koristiti tako su se zamijetili određeni obrasci koji se najčešće pojavljuju. Obrasci se pojavljuju u sve tri faze razvoja softvera. Iako su neki očiti, preporučeno je poznavanje svih obrazaca prije početka korištenja ovog načina razvoja softvera. Njihovo poznavanje će umanjiti vrijeme potrebno za implementiranje metoda i smanjit će broj pogrešaka.

Kod razvoja softvera vođenog testiranjem bitno je koristiti programski jezik koji ima aplikacijski okvir za testiranje dijelova. To nije presudan uvjet ali on olakšava razvoj softvera. Aplikacijski okviri sadrže mnogo korisnih metoda koje će poslužiti programeru.

# SUMMARY

Usage of test-driven development was started because programmers wanted to implement only the necessary methods in their programmes and so the tests could reveal if the current version of the program is working as expected. This method is commonly used in smaller projects, but because of the increasing popularity it is getting more and more used in bigger projects.

Test-driven development has three phases. The first (red) phase consists of writing tests for methods that are not yet implemented. In the second (green) phase we write methods that will pass the written tests. In the third (refactor) phase we are concerned about code quality, code maintenance and code reading simplicity.

As test-driven development was more and more used, programmers started to notice patterns. Patterns occure in all three phases in software development. Although some are simple it is recommended to learn all of the patterns before using this type of software development. Pattern recognition will decrease time needed for method implementation al will decrease the number of errors.

In test-driven development it is important to use a programming language that has a unit test framework. This is not a crucial requirement but it simplifies software development. Frameworks have a lot of useful methods that will be at disposal to the programmer.

# ŽIVOTOPIS

Rođen sam 1.6.1990. godine u Zagrebu, gdje sam završio Prvu ekonomsku školu. 2009. godine sam upisao Prirodoslovno-matematički fakultet odsjek matematika u Zagrebu na kojemu sam 2014. godine stekao titulu sveučilišnog prvostupnika edukacije matematike.
Nakon završetka preddiplomskog studija na istoimenom fakultetu upisujem Diplomski sveučilišni studij Računarstvo i matematika. Tijekom studija sam sudjelovao na izradi različitih projekata i sudjelovao sam na natjecanju Mozgalo u *data miningu.*

Engleski jezik aktivno koristim u pisanju i pisanju ( razina C2 ), a njemački jezik na razini B2.

Aktivno se bavim stolnim tenisom, sudjelujem u amaterskoj ligi Sokaz.

U slobodno vrijeme volim čitati knjige, gledati filmove te se družiti s prijateljima.