Sveučilište Jurja Dobrile u Puli

Fakultet informatike

ANTONELA BILOŠ

Razvoj programa vođen testiranjem

Završni rad

Pula, svibanj 2019.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

ANTONELA BILOŠ

Razvoj programa vođen testiranjem

Završni rad

JMBAG: 0303054110
Studijski smjer: Informatika

Predmet: Programsko inženjerstvo
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Tihomir Orehovački

Pula, svibanj 2019.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisana Antonela Biloš, ovime izjavljujem da je ovaj seminarski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio seminarskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

Pula, svibanj 2019.

IZJAVA

o korištenju autorskog djela

Ja, Antonela Biloš, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom *Razvoj programa vođen testiranjem* koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst, trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu sa Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli _____ (datum)

Potpis _____

Sadržaj

Sadržaj 5
Uvod 1
1. Testiranje 3
1.1 Počeci testiranja 3
2. Metode razvoja programa 5
2.1 Klasične metode razvoja programa 6
2.2 Agilna metoda razvoja programa 6
2.2.1 Ekstremno programiranje 7
3. Važnost testiranja 10
4. Prednosti razvoja programa vođenog testovima 14
5. Crvena, zelena i refaktor faza 17
6. Metode testiranja 19
6.1 Statičko testiranje 20
6.2 Dinamičko testiranje 20
6.2.1 Funkcionalno testiranje 20
6.2.2 Nefunkcionalno testiranje 21
6.3 Metoda testiranja Crna kutija 22
6.4 Metoda testiranja Bijela kutija 22
6.5 Testiranje komponenti, jedinično testiranje 24
6.6 Integracijsko testiranje 29
7. Uloge i organizacija kod testiranja 33
8. Testiranje i dokumentacija 36
Zaključak 38
Popis literature 39
Sažetak 40
Abstract 41
Popis slika 42

Uvod

Posljednjih godina u programskom inženjerstvu veliki fokus stavlja se na stabilnost koda i na njegovu iskoristivost. Da bi se izradio kvalitetan proizvod, lako održiv i lagan za korištenje, trebalo je provesti niz radnji koje bi to omogućile. S obzirom na to da su se tehnologije krenule ekstremnije razvijati, programski jezici postali su kvalitetniji, a razina sigurnosti veća. Došlo je do porasta cijena programskih proizvoda (eng. *Software*) i time do pojave želje za povećanjem stabilnosti koda na kojemu su ti isti proizvodi temeljeni. Kada programeri razvijaju neki programski proizvod, prije svega moraju imati očekivanja što će taj kod raditi i kako će se ponašati inače može doći do neočekivanih pogrešaka, nejasnih izlaznih jedinica (eng. *Output*) i najbitnije od svega, može doći do povećanja troškova.

U prošlosti se metode korištene za prevenciju i rješavanje problema takve vrste nisu pokazale najuspješnijima. Kao odgovor na manjkavost prethodno korištene kaskadne metode (eng. *Waterfall*) u upotrebu se polako uvodila nova metoda razvoja vođenog testovima. Kako bi se broj pogrešaka u kodu drastično smanjio i kako bi se došlo do krajnjeg željenog rezultata, poseže se za metodom razvoja programa vođenog testovima. Posljednji je rezultat upotrebe te metode povećanje kvalitete i brzine razvoja, a kao najvažnija stavka brzina je isporuke programa. Upotrebom takve metode lakše se pronalaze neispravnosti sustava nastale ljudskom greškom, odnosno unosom pogreške u kod. Što se kasnije greška otkrije, veći joj je trošak ispravka.

Budući da se žele provjeravati funkcionalnosti programa bitno je da se to radi često i da se cijeli proces bazira na povratnim informacijama koje se dobivaju iterativnim isporukama. U radu će biti detaljno razrađena agilna metodu programiranja, razvoj programa vođen testiranjem te će kroz primjere biti objašnjeno kako se ispravno testira i koristi u stvarnom poslovanju, koliko truda zahtjeva te koje pogodnosti donosi. Također, bit će prikazano koliko je stvarno primjena razvoja programa vođenog testiranjem zastupljena i koliko se ona ispravno provodi te koliko je njime pomaknuta kvaliteta programskih proizvoda zadnjih 10 godina. S obzirom na to da je proces testiranja programa jako širokoga spektra, biti će započeto kratkim povijesnim

razvojem, a zatim ulazak u srž teme. Kroz rad će se pokušati podržati hipoteza da je razvoj programa vođen testovima profitabilan i koristan svima koji se bave razvojem bilo kakvog programskog proizvoda.

1. Testiranje

Postoji više definicija pojma testiranje, ali neka općeprihvaćena bi bila (Manger, 2005:47):

''Testiranje je pokusno izvođenje programa ili njegovih dijelova na umjetno napravljenim rezultatima koji se izvodi uz pažljivo analiziranje rezultata.''

Testovi se mogu provoditi u bilo kojoj fazi razvoja programa, međutim uvijek moramo imati nekakvo očekivano ponašanje dijela ili kompleta programa koji testiramo. Kada Manger govori o pripravljenim rezultatima, misli na očekivano ponašanje koje testiramo, odnosno pripremu rezultata. Testovi provjeravaju radi li kod na očekivan način. Kada se testira, važno je znati da se može otkriti postoje li greške u napisanom kodu, ali se ne može dokazati da ih nema. Kako bi se pronašlo što više možebitnih pogrešaka, treba se koristiti što čudnije ulazne podatke. Testiranje treba služiti kao provjera postojanja predviđenih, odnosno očekivanih funkcionalnosti u programu i, naravno, da otkrije pogreške programa prije nego li taj program dođe u krajnju upotrebu. Što znači *očekivano* i kako se zna da su određeni izlazni podaci (eng. *Output*) očekivani ili ne? Očekivano ponašanje nekog dijela koda može se saznati iz očekivanja krajnjih korisnika (validacijsko testiranje) ili iz specifikacije (verifikacijsko testiranje).
Validacijskim testiranjem se provjerava odgovara li proizvod koji se gradi uvjetima zadanim u dokumentaciji dok se verifikacijskim testiranjem pokušava utvrditi odgovara li proizvod stvarnim potrebama krajnjih korisnika.

1.1 Počeci testiranja

Ozbiljni počeci testiranja programskih proizvoda bilježe se 80-ih godina prošloga stoljeća. Zapravo nije zabilježen točan datum kada je testiranje prvi put provedeno, ali testiranje kao pojam prvi put uvodi Glenford J. Myers 1979. godine. U knjizi *The art of Software Testing* razdvaja se pojam debugiranja (eng. *Debugging*) od pojma testiranja i sav fokus se stavlja na pronalaženje grešaka. Objašnjeno je da je

uspješan onaj test koji pronađe grešku i tim zaključkom kompletno se odvaja pronalazak grešaka od debugiranja, odnosno otklanjanja pronađene greške (Myers, Badgett i Sandler, 2012.). Testiranjem se ne može ustanoviti da proizvod funkcionira u svim situacijama, ali može se ustanoviti da ne radi ispravno u određenim situacijama. Kako bi programski proizvod što više i što bolje bio testiran, treba ga dobro poznavati. Time stavljamo u fokus ispitivanje i poznavanje koda što je jedna od najvažnijih stvari u cijelom procesu. Da bi se proizvod što bolje testirao, treba ga izložiti različitim uvjetima i okruženjima.

2. Metode razvoja programa

Od kada se programsko inženjerstvo pojavilo kao disciplina pojavile su se i brojne metode razvoja programskog proizvoda. Njih se svrstalo u dvije najopćenitije kategorije klasične i agilne metode. Klasične su se pojavile sedamdesetih godina, i u upotrebi su i danas, dok su agilne nastale malo kasnije kao modernije metode te postale izrazito popularne. Klasične su one koje podržavaju planiranje izrade proizvoda unaprijed s dokumentacijom svih detalja, s unaprijed pretpostavljenim vremenskim periodom i mnoštvom detalja za daljnju izradu. S druge strane, agilne su metode fleksibilnije s vrlo kratkim prethodnim planiranjima, mogućnostima akomodacije na nove ideje i okruženja.

Već godinama se unutar sektora informacijskih tehnologija raspravlja koju je metodu bolje koristiti, koja je profitabilnija te koja će uroditi boljim plodom kada bude izvršena. Međutim, jedna i druga imaju svoje prednosti i nedostatke kao i ljude koji se zalažu za njih i njihovo korištenje. Iako se agilna metoda pokazala kao metoda u široj upotrebi zbog svoje fleksibilnosti, klasične metode nisu napuštene i dalje se razvijaju.

Međunarodna IT savjetodavna tvrtka Standish Group, koja se bavi istraživanjima projekata javnog i privatnog područja rada, pratila je statistiku 5000 različitih projekata razvoja programa, od 2011. do 2015. godine, i provedena statistika govori da je samo 11% onih planiranih i izvedenih projekata klasičnim načinom uspješno završeno, odnosno dovedeno do kraja i isporučeno (Hastie i Wojewoda, 2015.). Suprotno tomu, postotak se diže na 39% uspješnosti za sve projekte na kojima korišten agilni pristup razvoju programa. Da je tvrdnja o efikasnosti i zastupljenosti agilne metodologije ispravna, govori i informacija da sve veće IT organizacije koje razvijaju neki programski proizvod koriste agilni način. Dobar su primjer Microsoft i Google.

2.1 Klasične metode razvoja programa

Pojavom računala, njihovom popularizacijom i komercijalizacijom, počelo se shvaćati da bi se ona, osim za osobnu upotrebu, mogla koristiti i za poslovanje. U počecima se koristilo za obične matematičke i evidencijske stvari, a kasnije se javila potražnja za aplikacijama i programskim proizvodima što je dovelo do enormnog rasta programskog inženjerstva kao struke.

Kako se klasične metode odlikuju s dugotrajnim procesom planiranja i dokumentiranja proizvoda, tako se i testiranje odvijalo dugotrajno, ali ne kao odlika te metode. Događalo se to da su osobe koje su bile zadužene za testiranje morale ručno unositi podatke i zapisivati ono što su dobili. Rezultati su se vodili kao testne skripte i koristili kao svojevrsni vodič, odnosno proces kojeg tester mora pratiti da bi odradio određenu akciju u sustavu i dobio uspješno neki rezultat. Cijeli proces testiranja trajao bi mjesecima, tjednima u najboljem slučaju, i to je bilo predugo. Bilo kakvom promjenom koda, tj. funkcionalnosti programskog proizvoda, trebali su se mijenjati i testovi te ponovno izvršavati tako da bi se sve moralo ponavljati i ponavljati što je počelo stvarati dodatno poteškoće.

2.2 Agilna metoda razvoja programa

Agile Manifesto uvodi pojam agilnosti u razvoj programskog proizvoda 2001. godine. Od tada se takav način razvoja programa i aplikacija koristi kao glavni i najproduktivniji.

Za početak je vrlo bitno objasniti što je to agilni razvoj i kakve on veze ima s testiranjima. Agilni je razvoj način razvoja programskog proizvoda koji je temeljen na iterativnim isporukama i baziran na povratnoj informaciji korisnika. Takav je način razvoja programa dosta prilagođen modernom programiranju i usmjeren na zadovoljstvo krajnjih korisnika. Zahtjevi korisnika često znaju biti jako konfuzni i nestabilni, a možda i vremenom promjenjivi, što bi značilo velike probleme za programere ako ne bi bilo iterativnih isporuka i traženja povratnih informacija od korisnika što omogućava i agilni razvoj. Naime, agil znači i prilagodljiv (eng.

Adaptable), iterativan i inkrementalan razvoj programa, što samo objašnjava gore navedeno. Kada postoje jako kompleksni i konfuzni zahtjevi korisnika, metoda agilnog razvoja kontrolira na neki način vrijeme u kojem se zahtjev korisnika može promijeniti. Kao jedan od većih benefita cijelog procesa je baš ta kontrola procesa prilagođena za razne promjenjive okolnosti.

Prvo se razvijaju funkcionalnosti s najvećom poslovnom vrijednošću, odnosno one funkcionalnosti od najvećih benefita za korisnika. Agilni razvoj podrazumijeva dobro organiziran tim ljudi koji su se spremni prilagoditi i brzo donositi bitne odluke. Agile potiče prilagodljivo planiranje, razvoj programskog proizvoda u vremenski ograničenim iteracijama, brz i fleksibilan odgovor na promjene i stalne povratne informacije od izdavatelja zahtjeva. Do povratne informacije dolazi se redovitim testiranjem i, kao što je navedeno, čestim funkcionalnim isporukama. Unutar agilnog razvoja programskog proizvoda nastale su i neke nove metode programiranja, najpoznatija i najraširenija je metoda ekstremnog programiranja koja u praksi potpuno podliježe razvoju programa vođenog testovima.

Agilne metode su danas u široj upotrebi, a najčešće se koriste:

- Ekstremno programiranje (eng. *Extreme Programming (XP)*)
- Scrum metoda
- Razvoj temeljen na poboljšanjima (eng. *Feature Driven Development*)
- Jasan slučaj (eng. *Clear Case*)
- Razvoj temeljen na prilagodljivosti (eng. *Adaptive Software Development*)

2.2.1 Ekstremno programiranje

Ekstremno programiranje (eng. *Extreme Programming, XP*) je, kao što je gore navedeno, daleko najpoznatija i najraširenija metoda agilnog razvoja programa. Ekstremno programiranje utemeljeno je na metodama koje za glavni zadatak imaju osigurati visoku kvalitetu proizvoda.

Kao glavni cilj XP-a bilo je smanjenje troškova promjena na programu koje dolaze zbog promjena zahtjeva. Ideja je bila da se cijeli proces rastavi na manje jedinice čiji se razvoj planira polako i zasebno. Samo se za sljedeću jedinicu planira što i kako. Cijeli proces je fleksibilan i kroz njega se mogu dodavati novi dijelovi ili promijeniti oni već implementirani.

XP je nastao devedesetih godina kao odgovor na sve veću neuspješnost projekata vođenih tada klasičnom kaskadnom metodom koja je određivala tijek razvoja programskog proizvoda. Tijek se sastojao od izrade specifikacije, dizajniranja projekta, implementacije koda, verifikacije, validacije i održavanja programskog proizvoda. Glavni problem kaskadne metode je u njenoj nefleksibilnosti i nemogućnosti prihvaćanja promjena tokom procesa. Međutim, ekstremno programiranje uzima mogućnost promjene i prilagodbe kao polaznu točku razvoja zbog čega je ovakav pristup nenadmašno efikasniji i zastupljeniji.

Kent Beck, poznati američki programski inženjer popularizirao je i uveo TDD u ekstremno programiranje. On je objasnio dva vrlo jednostavna pravila za razvoj programa vođenog testiranjem, jedno je da se treba napisat novi poslovni kod samo onda kada automatizirani testovi padnu, a drugo da je potrebno ukloniti sve duplikacije koje se pronađu u kodu (Beck, 2002.).

Naime, ta se dva pravila na prvi pogled čine jako zbunjujuća i možda nedovoljno definirana, ali u srži imaju smisla. Kada automatizirani testovi padnu, potrebno je promijeniti kod jer poslovna logika koja leži ispod je kriva ili možda nedovoljno pojednostavljena. S druge strane, kada se govori o duplikacijama, stvar je malo drugačija, možda čak i malo kompliciranija. Duplikacije mogu biti mrtvi kod, nepotrebno spremanje istih podataka u bazu ili implementacija iste poslovne logike na različita mjesta u kod. Postoje dva pravila koja definiraju timsko i pojedinačno ponašanje članova koji razvijaju neki proizvod (Beck, 2002.):

- Razvija se organski s kodom koji izvodi i omogućava povratne informacije između odluka.
- Svatko piše testove za sebe jer se ne može čekati 20 puta dnevno da ih netko napiše za vas.

- Razvija se okruženje koje mora biti sposobno brzo odgovoriti na male promjene (npr. brži *kompajler*).
- Dizajn se mora sastojati od visoko kohezivnih i labavo povezanih komponenti.

Navedene činjenice su jako bitne i tim ljudi koji razvijaju isti projekt ih se treba držati kao nekog obrazca (eng. *Pattern)*.

Najveća je promjena koju je donijela XP metoda pisanje programa, tj. koda vođenog testiranjem. U primjenu se uvelo pisanje koda nakon čega bi odmah uslijedilo i njegovo testiranje, odnosno provjera radi li sve napisano kako se željelo. Takvim pristupom osiguralo se da kod koji prođe testove radi i da nema bespotrebno implementiranih funkcionalnosti koje ničemu ne služe ili ne rade ništa - eliminacija mrtvog koda.

Naime jako je važno da programeri pišu efektivne testove. Kakvi su to dobri testovi?

- Da se izvode brzo (da se brzo prilagode, da imaju kratko vrijeme izvođenja i da padaju).
- Da se daju modificirati, odnosno prilagoditi.
- Da se koriste podaci koji ih čine jednostavnima i lako razumljivima.
- Da se koriste realni podaci kada je to potrebno.
- Da predstavljaju jedan korak prema generalnom cilju.

Zadnja činjenica zapravo objašnjava i jedan dio svrhe testiranja, predstavljanje jednog koraka prema cilju, odnosno ispunjavanje jednog od slučaja uporabe (eng. *Use-case*) navedenog u dokumentu sa zahtjevima korisnika, odnosno pokrivanje funkcionalnosti koje je ispunila takav slučaj (Beck, 2002.).

3. Važnost testiranja

Testira se zbog kvalitete i cijene. Kada se testiranje promatra iz perspektive kompanija koje se bave razvijanjem bilo kakvog programa, vidi se iz predstavljene šire slike IT[1] industrije da su većini prihodi jako veliki. Naime, ako se dogodi bilo kakav gubitak, on je drastično velik. Gubici su kompanija koje se bave programskim proizvodima milijunski.

Recimo da se prodaje nešto neopipljivo. Već se u startu zahtijeva velika opreznost. Ako ste isporučili neispravan proizvod, gubite klijenta, ako ne isporučite drugi kvalitetan proizvod, gubite posao. Potražnja je jako velika, ali i ponuda. Cifre su ogromne. Glavni razlog za testiranje su zapravo novci. Osobe koje se bave testiranjem kao glavnu zadaću trebaju imati otkrivanje velikog broja problema i to ne onih sitnih, dizajnerskih, već onih velikih, enormnih, čije bi posljedice bile katastrofalne s ekonomskog aspekta. Zbog pronalaska i otklanjanja što većeg broja rizičnih situacija poseže se za testiranjem. Ne treba se zavarati i misliti da testiranje pronalazi sve greške i osigurava ogromnu sigurnost i ekonomsku stabilnost, ali osigurava dovoljno dobru, ovisno u kojoj mjeri programski proizvod biva izložen testovima.

Bitno je razumjeti da je greške potrebno pronaći čim prije jer odmicanjem vremena troškovi otklanjanja grešaka postaju sve veći. Sada se govori i o gubitku vremena, ljudskih resursa, što je opet na kraju - novac. Troškovi otklanjanja grešaka prouzrokuju i nezadovoljstvo kupaca umjesto da se prethodno pronađu greške i otklone. Potrebno je preventivno djelovati na greške prije nego one završe u produkciji kako bi se izbjegla kašnjenja s isporukama i kako bi se izbjegli naknadno uvećani troškovi otklanjanja tih istih grešaka. Zadnjih godina razvijene su razne metode o kojima će se govoriti kasnije, a koje služe sprječavanju nastanaka grešaka programskog proizvoda.

[1] Informatička tehnologija (IT) je opće usvojen naziv za tehnologije i područja koja su usko vezana s informacijama i radom s njima.

Kod pisanja koda bitno je ići malim koracima. TDD[2] to značajno omogućava. Razvijanje programa u praksi je daleko produktivnije u malim koracima nego u pisanju ogromnog komada koda od jednom. Kada se napiše 100 linija funkcionalnog koda i onda krene testirati, velike su šanse da će nešto poći po krivu i da će testovi popadati zbog nedostataka iz novog koda. Mnogo je lakše pronaći i popraviti nedostatke, odnosno manjkavosti koda ako se testira svega par linija funkcionalnog koda.

Cijeli proces izgradnje programskog proizvoda može se zamisliti na dva, malo jasnija načina:

1. Ideja je proizvesti vozilo i kreće se s izrađivanjem motora. Paralelno s tim se izrađuje limarija, odnosno kostur auta te sjedala i oprema. Kompletan proizvod još ne postoji. U jednom trenu svi su pojedinačno proizvedeni dijelovi gotovi i trebaju se spojiti. Dogodi se da je motor premali za auto, da sjedala i oprema nisu taman kako bi trebala biti.

2. Ideja je proizvesti vozilo. Krećete se izrađivati motor i kad je on gotov kreće se s nadograđivanjem šasije na njega. Kada je sve to gotovo, napravi se kostur auta i postavi se motor u njega. Kada se vidi da motor radi u automobilu, kreće se s dodavanjem dodatne opreme.

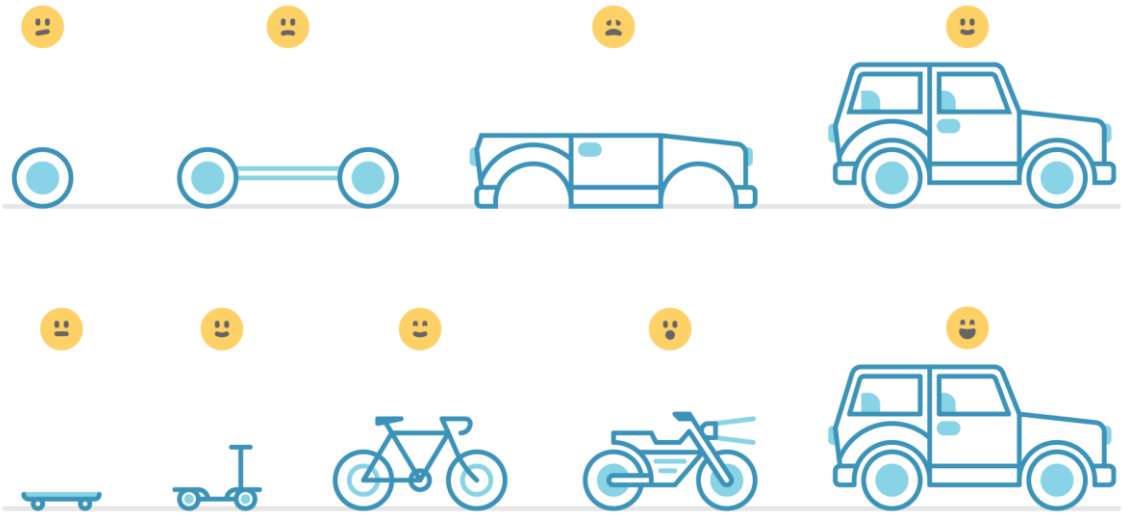
Kod primjera br.1 dijelovi automobila radili su se paralelno radi brže izrade te nisu testirani u iteracijama što je rezultiralo manjkavostima krajnjeg proizvoda. Zbog izostanka testiranja pojedine faze izrade automobila, došlo je do problema koji treba riješiti ako se želi funkcionalan automobil, ali onda proces zahtjeva i dodatne resurse i vraćanje u neke ranije faze što je u ovom slučaju gotovo nemoguće.

U primjeru br. 2 dijelovi automobila se nisu izrađivali paralelno, nego je napravljen jedan dio čija je ispravnost testirana i kada je to bilo gotovo nadogradio se neki novi dio automobila koji je kao i prethodni prvo testiran. Kada se govori u kontekstu

[2] Test Driven Development (TDD) proces je izrade programa ili aplikacije koji je baziran na testiranju i koji koristi vrlo specifične testne metode u svrhu poboljšanja proizvoda.

programa to bi bilo dodavanje neke nove funkcionalnosti. Kada bi dodavanje drugog dijela bilo uspješno završeno, na to bi se nadogradio neki sljedeći dio i tako bi se išlo prema pravom i kvalitetnom proizvodu koji bi imao neku svrhu u svakoj fazi izrade.

Ako je primjer konfuzan, na slici ispod je prikazan malo jednostavniji.



1. Proces stvaranja proizvoda

Bilo da govorimo o automobilu ili programu inicijalna je ideja ista - dobiti funkcionalan proizvod. Na gornjem dijelu slike se prikazuje kako je napravljen prvi dio – kotač. U fazi kada proizvođač ima samo jedan dio (kotač) još uvijek nema funkcionalan proizvod koji može na neki način upotrijebiti ili možda pokazati nekome. Sljedeće što proizvođač radi jest da spoji dva kotača, ali i dalje nema funkcionalan automobil koji može voziti. Zatim dolazi faza kada napravi i kostur automobila, ali i dalje nema automobil koji može voziti. Tek u finalnoj fazi ima konkretan proizvod koji obavlja željenu funkcionalnost.

Na donjem dijelu slike vidi se kako proizvođač gradi proizvod tako da ima male funkcionalne cjeline koje nadograđuje. U prvoj fazi ima *skateboard*, znači da ima proizvod sa malom funkcionalnošću koji ipak vozi. Zatim ga nadogradi i ima proizvod koji malo više nalikuje na željeni, ali još uvijek nedovoljno dobar, ali i u toj fazi ima

proizvod koji vozi. Svaka daljnja faza nadograđivanja dovodi proizvođača korak bliže proizvodu koji želi i u svakoj fazi ima proizvod koji obavlja željenu funkcionalnost.

Razvoj programskog proizvoda vođen testiranjem je upravo to. Izrada male funkcionalnosti koja se testira. Kada je uspješno testirana doda se nova i tako iteracijama izrade i testiranja nastane proizvod koji je stabilan, funkcionalan i koristan.

4. Prednosti razvoja programa vođenog testovima

Kako Bender i McWherter navode, neke od prednosti razvoja programa vođenog testiranjem sljedeće (Bender i McWherter, 2011:9-10):

- *"TDD osigurava kvalitetan kod od početka. Programere se motivira da pišu samo kod koji je potreban da testovi prođu i da ispune zahtjeve korisnika. Ako metoda ima imanje koda, jedino je logično da ima i manje šansi za imati grešku.*

- *Program 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 Programming Interface, API). Programer koji piše programe i radi sučelja je također prva osoba koja ih koristi te na temelju toga odmah vidi imaju li 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 se kod buduće promjene koda ili dodavanja novih metoda 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."

Dobri testovi osiguravaju kvalitetan i stabilan kod. Ideja da se piše što manje linija koda je sve zastupljenija u svijetu jer manje koda znači i manji broj grešaka. Ponekad korisnički zahtjevi nisu baš jednostavni i ne mogu se proizvesti kroz malo linija koda, ali ne treba se opteretiti ni ako količina koda nije minimalna jer se i dalje taj kod može testirati. Kod treba biti jasno i jednostavno napisan, a metode i klase ispravno imenovane kako bi se lakše testiralo i snalazilo po napisanom kodu. Da bi cijeli proces testiranja bio sasvim učinkovit, potrebno je testirati svaku napisanu metodu i imati kod bez ponavljajućih djelova.

Pokrivenost cijelog koda testovima znači da su postavljeni dobri temelji za dodavanje novih funkcionalnosti u budućnosti. Oni točno govore kako se sustav treba ponašat. Kada su testovi dobro napisani oni služe kao izvrsna projektna dokumentacija koja je uvijek usklađena s kodom. Programeri navode i jednu neočekivanu prednost pisanja testova. Kada se testovi napišu i kada prođu te se prikažu malene zelene kvačice koje znače da su svi testovi uspješno prošli, motivacija za daljnjim pisanjem postaje zarazna. Sa psihološkog aspekta, zbog takvih malih pozitivnih povratnih informacija o stabilnosti koda postaje se motiviraniji i zadovoljniji poslom koji se obavlja, što je danas jako rijetko.

Testiranje ima jako puno predno, još neke od njih su:

- Kvaliteta – Bitna je korisniku i pružatelju usluge. Za kvalitetan program, korisnik će platiti puno novaca a pružatelj usluge dobro zaraditi. Obje strane su zadovoljene. Kod razvijanja programa ili aplikacija potrebno je gledati dugoročno. Najbitnije od svega je graditi kvalitetan projekt te njime zaslužiti snažnu reputaciju i dobro ime za daljnje razvijanje aplikacija.

- Zadovoljni korisnici – Iako se kvaliteta i zadovoljstvo korisnika povezuje, zadovoljstvo korisnika ne dolazi samo od kvalitete programa. Centar bilo kojeg dobrog poslovanja bi trebali biti zadovoljni korisnici. Kada se nešto prodaje,

bitno je osigurati da to što se prodaje bude pravovremeno atrakcija, i da je postoji potražnja za tim. Facebook ne bi uspio kao danas da je napravljen 15 godina ranije, jer tada osobna računala nisu bila u većini kućanstava i ne bi imao ni milione korisnika kao danas ni zaradu. Korisniku ne znači ništa kupnja programa koji nije koristan. Testiranje tržišta i potražnje na tržištu također utječe na zadovoljstvo krajnjih korisnika.

- Profit – Uvijek i u svemu, točka na koju se uvijek treba vratiti je profit. Dobro proizveden proizvod ne treba jake marketinške reklame jer si ljudi međusobno preporučuju programe. Preporuke su najvrijednija reklama koju proizvođač može dobiti. Kada iznosi na tržište ozbiljan i strogo testiran program, dobiva povjerenje korisnika. To mu pomaže zadržati stare klijente i skupiti nove.

Iskustvo klijenta tokom korištenja programa je velika stavka za testirati. Program treba biti jednostavan, jasan i jednostavan za korištenje. Samo jako dobri i iskusni testeri to mogu osigurati. Njihovo iskustvo i zapažanje će osigurati da program koji se gradi bude logičan i intuitivan. Da bi se dobilo sjajno korisničko iskustvo, program treba biti bez ikakvih grešaka i smetnji. Odabi prave metode testiranja programa i profesionalnog tima osigurava se kvalitetan proizvod od početka i zadovoljstvo korisnika na kraju razvoja.

5. Crvena, zelena i refaktor faza

Cijeli proces razvoja programa koji je vođen testovima svodi se na par osnovnih koraka:

- Dodavanje testova
- Pokretanje testova te praćenje hoće li koji pasti
- Pisanje još koda
- Ponovno pokretanje testova i refaktor koda
- Sve ponoviti

Dodavanje testova znači pisanje istih. Prvo bi se trebao napisati test pa kod za njega. Slijedi pokretanje napisanih testova, i provjeravanje njihove prolaznosti. Prolazak testova osigurava prijelaz u sljedeću fazu, a pad znači da je potrebno još vremena i popravaka. Kada se napiše još novog koda, testovi se opet pokreću i provjerava se uspješnost. Međutim, kada se ide u dalju problematiku proširuju se ti osnovni koraci da bi se dobili malo precizniji i detaljniji procesi.

Crvena, zelena i refaktor faza tri su glavna procesa kojih se pridržavaju svi oni koji koriste metodu razvoja programa vođenog testiranjem. Tim redoslijedom osigurava se ono najbitnije, da postoje testovi za koje će se pisati kod te da se piše samo onoliko koda koliko nam je potrebno da bi prošli testovi.

Prva faza je crvena faza (eng. *Red phase*). Inicijalna faza tijekom koje se pišu testovi za funkcije i klase koje još nisu napravljene. Tijekom te faze očekivano je da testovi neće proći i da će testna konzola biti crvena zbog čega se ova faza i naziva crvenom fazom. Ona je jedna od najtežih za početnike.

Druga faza je zelena faza (eng. *Green phase*). U nju se dolazi nakon što se ne prođu testovi, ali je ona upotrjebljena tek kada se napiše minimalni kod od svega par linija

koji osigurava prolaznost testova. Uglavnom se piše najjednostavniji kod radi prolaznosti testova. U toj fazi se ne treba opteretiti izgledom koda, niti njegovom urednosti. Kada se pokrenu testovi nakon ove faze, testna konzola će biti zelena.

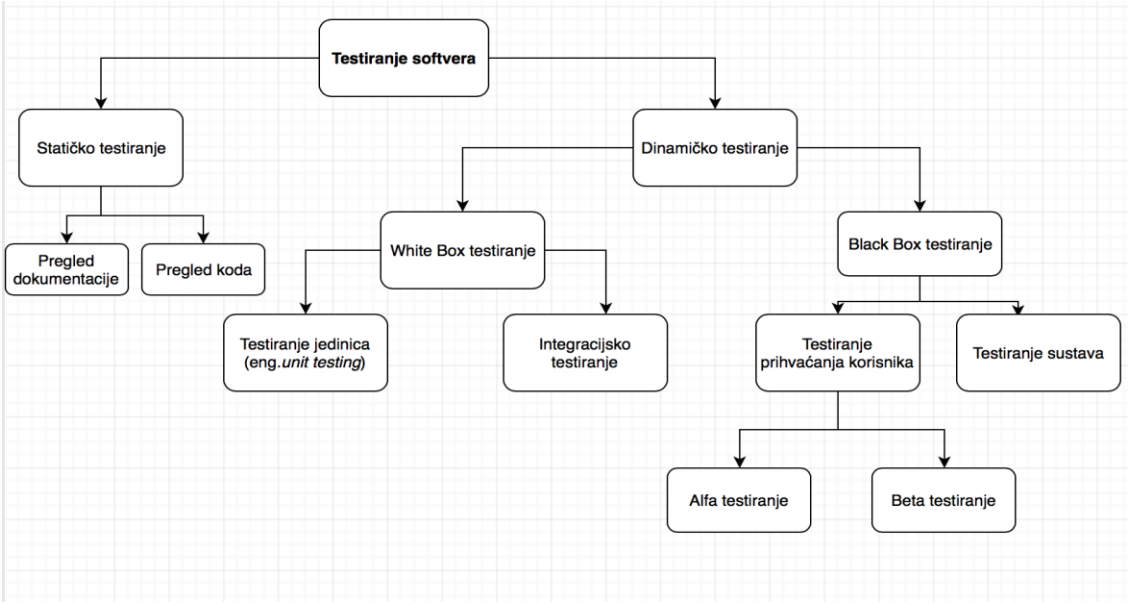
Treća faza je refaktor. U tu fazu dolazi se kada testovi uspješno prođu i kada se imaju željene funkcionalnosti. U toj fazi se kod unaprjeđuje, popravlja i prilagođava radi bolje čitljivosti, jednostavnosti i mnogih drugih razloga. Nekad se i piše određeni dio ispočetka ako se može kraće i jednostavnije. Fokus je na kvaliteti koda. Kada je ta faza gotova, ide se ispočetka na prvu fazu. Ponovno se ponavlja cijeli proces i nekoliko puta. Ponavljanje uglavnom ne bude dugotrajno.

Ovakav proces je učinkovit zbog jednog glavnog razloga. Uvijek se polazi od sitnih koraka i nekog testa kojeg se pokušava provjeriti. U svakom koraku u kojem se naleti na grešku, odmah ju se može alocirati i ispraviti. Kroz cijeli proces nije napisano puno linija koda pa je pogrešku lakše pronaći, što znači da u svakom koraku programer točno zna gdje je pogriješio i da zna točno kada je spreman za krenuti u daljnje testiranje. Praćenjem ovih metoda, programer postaje siguran u kod i stabilnost programa koji gradi. Ispravnim korištenjem crvene, zelene i refaktor faze, programer štedi vrijeme i novac, jer je pravovremeno otklonio greške na koje se neće trebati kasnije vraćati.

6. Metode testiranja

Dosta se puta može čuti kako je testiranje nekog programa sporedan i ne toliko važan proces. U nekim jako rijetkim slučajevima testiranje se može voditi kao neka manje bitna stavka prilikom izrade proizvoda. Ako program koji se izrađuje ne utječe direktno na poslovanje, nema veliki broj korisnika te obavlja neku sporednu i sasvim nebitnu ulogu, greške se mogu primijetiti i ukloniti kada se program počne koristiti. Takvi su slučajevi zaista rijetki jer se danas većina programskih proizvoda koristi za obavljanje neke poslovne aktivnosti koja je od velikog značaja. Kada se govori o nekom kvalitetnom pristupu izradi programskog proizvoda, tada se testovi uzimaju kao jedna od najbitnijih stavki. Testiranje dovodi do kvalitetnog programa, a isporučenje istog dovodi do zadovoljstva korisnika i programera koji ga isporučuju ili ljudi za koje ti programeri rade.

Testiranje je, pogotovo zadnjih godina, vrlo korišten proces. Postoje razne metode testiranja, međutim najčešće se dijeli na funkcionalno i nefunkcionalno, ovisno o tome što se želi testirati.



2. Dijagram podjele testiranja

Kako je prikazano na slici, postoji dosta podjela testiranja, ali su tu prikazane samo neke, a kasnije će u tekstu biti objašnjene i detaljnije podjele. Na višem nivou testiranje se dijeli na statičko i dinamičko. Testiranje programa, naravno, podrazumijeva uporabu obje metode. Nije pametno koristiti samo dinamičko ni samo statičko testiranje, zapravo jedno bez drugoga baš i ne može.

6.1 Statičko testiranje

Statičko je testiranje posebna metoda testiranja programa koja se u većini slučajeva prakticira neposredno prije dinamičkog testiranja. Kao metoda je vrlo bitna jer osigurava čisti start prilikom izrade programskog proizvoda tako što osigurava da arhitektura programa bude ispravna, da program odgovara dokumentu o specifikacijama. Najbitnija točka prepoznavanja statičkog testiranja je ta da se kod ne izvršava tijekom testiranja. Jedna je od vrsta testiranja statičkom metodom statička analiza koda koju se iskorištava tako da se provjerava složenost koda i sintaksne pogreške koje kod može sadržavati. Općenito se ono vrši na razne načine, uglavnom kroz neke preglede i revizije napisanog koda. Takve analize rezultiraju nekom dokumentacijom ili drugim kodom.

6.2 Dinamičko testiranje

Dinamičko testiranje druga je velika podjela testiranja programa. Od statičkog se razlikuje po tome što se prilikom prakticiranja ove metode kod izvršava. Nije bitno govori li se o dijelu koda ili kompletnom sustavu, ključna sintaksa je izvršavanje koda. Glavna podjela dinamičkog testiranja je na funkcionalno i nefunkcionalno testiranje.

6.2.1 Funkcionalno testiranje

Kod funkcionalnog testiranja fokus je da se ustanovi odgovara li program zahtjevima. Provjeravaju se ulazne i izlazne jedinice te stanje sustava na kraju.

Upotreba funkcionalnog testiranja je manje-više slična za sve sustave iako se izlazne i ulazne jedinice u svim sustavima razlikuju. Svaki poslovni sustav u nekoj mjeri izvršava pojedine funkcionalne testove, ali često bez plana i popratne dokumentacije koja odgovara sustavu. Funkcionalnim testiranjem provjeravaju se pojedine funkcionalnosti programa, a kao glavni cilj i zadaću uzima se dokazivanje kvalitete istog. Baza takve metode testiranja je test baziran na scenarijima (eng. *Test case*) koji su izrađeni na funkcionalnoj specifikaciji.
Postoje razne podjele i za takvu vrstu testiranja, ali glavne su:

- Ručno funkcionalno testiranje – testni scenariji su dokumentirani u nekom alatu za testiranje, npr. Microsoft Word.
- Automatizirano funkcionalno testiranje – testiranje se izvršava uz pomoć nekog alata koji automatizira funkcionalno testiranje.
- Regresijsko testiranje – svaka nova verzija programskog proizvoda donosi neke promjene koje mogu utjecati na ostatak programa i zbog dodavanja novih stvari koje mogu utjecati na sve potrebno je testirati koliko su ispravne stare funkcionalnosti.

6.2.2 Nefunkcionalno testiranje

Kod nefunkcionalnog testiranja programa fokus je stavljen na ispitivanje kvalitativnih aspekata programskog proizvoda. Neki od primjera nefunkcionalnog testiranja su:

- Testiranje sigurnosti programa – fokus je na sigurnosti implementiranih funkcionalnosti i podataka.
- Test pouzdanosti – fokus je na provjeravanju radi li sustav ispravno duži vremenski period.
- Test performansi – fokus je na provjeri brzine i vremena izvršavanja pojedinih zahtjeva.

6.3 Metoda testiranja Crna kutija

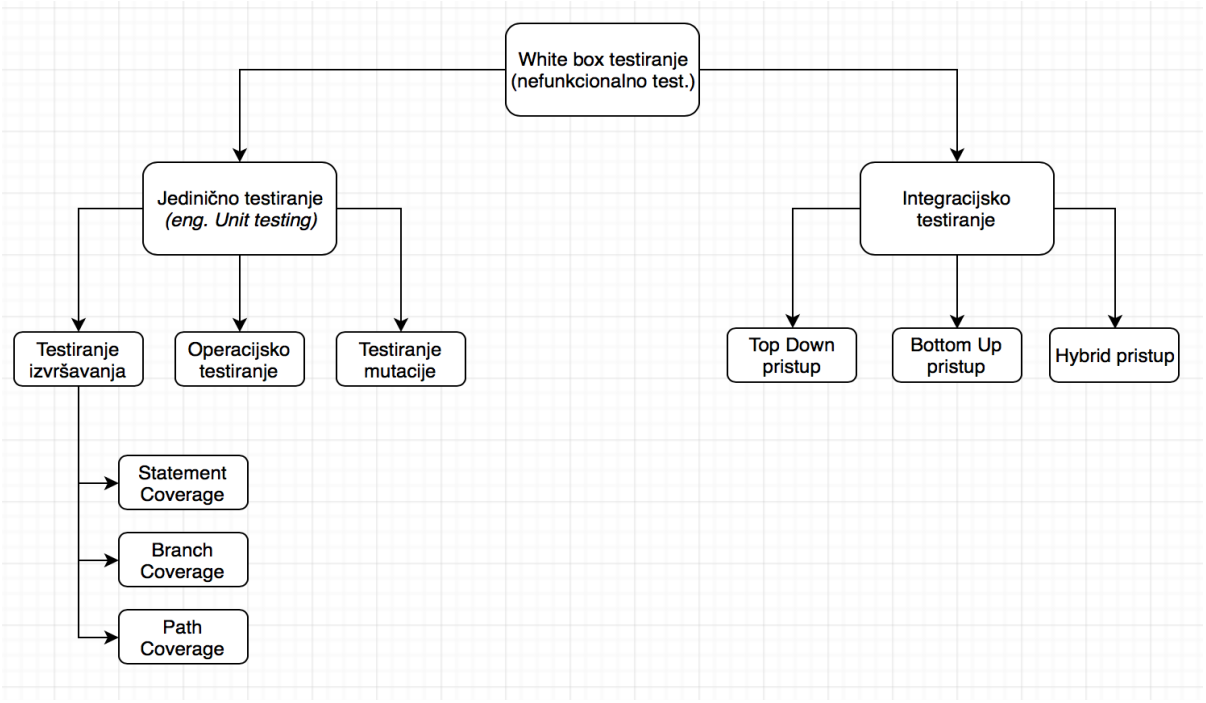
Crna kutija je metoda funkcionalnog testiranja (eng. *Black box*) koja se koristi ako se žele testirati funkcionalnosti programa ne dodirujući se previše direktno koda. Na osnovu prethodno definiranih koraka sistem uzima podatke i obrađuje ih, zatim se promatra slažu li se ulazne vrijednosti, odnosno ti podaci s očekivanjima. Naziva se i crna kutija jer se kod te metode sustav koji testiramo promatra kao izolirani sustav. Kod takve metode testiranja arhitektura samog sustava nije poznata osobi koja testira niti implementacija funkcionalnosti pa prema tome osobi koja testira nije ni potrebno znanje nekog programskog jezika. Fokus je na testiranju generalnog sustava i na testiranju korisničke prihvatljivosti. Pri planiranju crne kutije testiranja glavni značaj ima sama specifikacija programa. Iako je specifikacija početna točka cijelog procesa, ova metoda ne može garantirati da je kompletna specifikacija implementirana u sustav. Zapravo je ovakvo testiranje nazvano testiranjem protiv specifikacije jer se njime nastoje otkriti propusti specifikacije, odnosno dijelovi koji nisu popunjeni. Za razliku od bijele kutije (eng. *White Box*) testiranja koje provjerava implementaciju (testiranjem protiv implementacije), crna kutija puno je jeftinija metoda i prethodi metodi bijele kutije.

6.4 Metoda testiranja bijela kutija

Kod metode testiranja nazvane bijela kutija (eng. *White box*), kreće se od pretpostavke da tester dobro poznaje unutarnju strukturu programskog proizvoda. Bijela kutija metoda je nefunkcionalnog testiranja koja je još nazvana i provjerom protiv implementacije jer nastoji pronaći greške u implementaciji koda. Osim toga postoji više naziva za takvu metodu testiranja, npr. ispitivanje staklene kutije (eng. *Glass box testing*), ispitivanje otvorene kutije (eng. *Open box testing*) itd. Iz samih se naziva da naslutiti da su unutarnji mehanizmi sustava poznati testerima. Kod metode bijele kutije testiraju se samo interni komadi koda i infrastruktura programskog proizvoda. Privi korak procesa je provjeravanje koda prema postojećim specifikacijama. Jednom kada se osoba koja testira upozna sa strukturom koda, testovi se pokreću da bi se provjerilo je li program zadovoljio zahtjeve navedene u dokumentu o specifikaciji. Takvim ispitivanjem dolazi se do

povećane sigurnosti i korisnosti programa te se njime otkriva većina ranjivosti koje taj program može imati. Kod rane faze testiranja programa takva je metoda najbolja za otkrivanje pogrešaka.

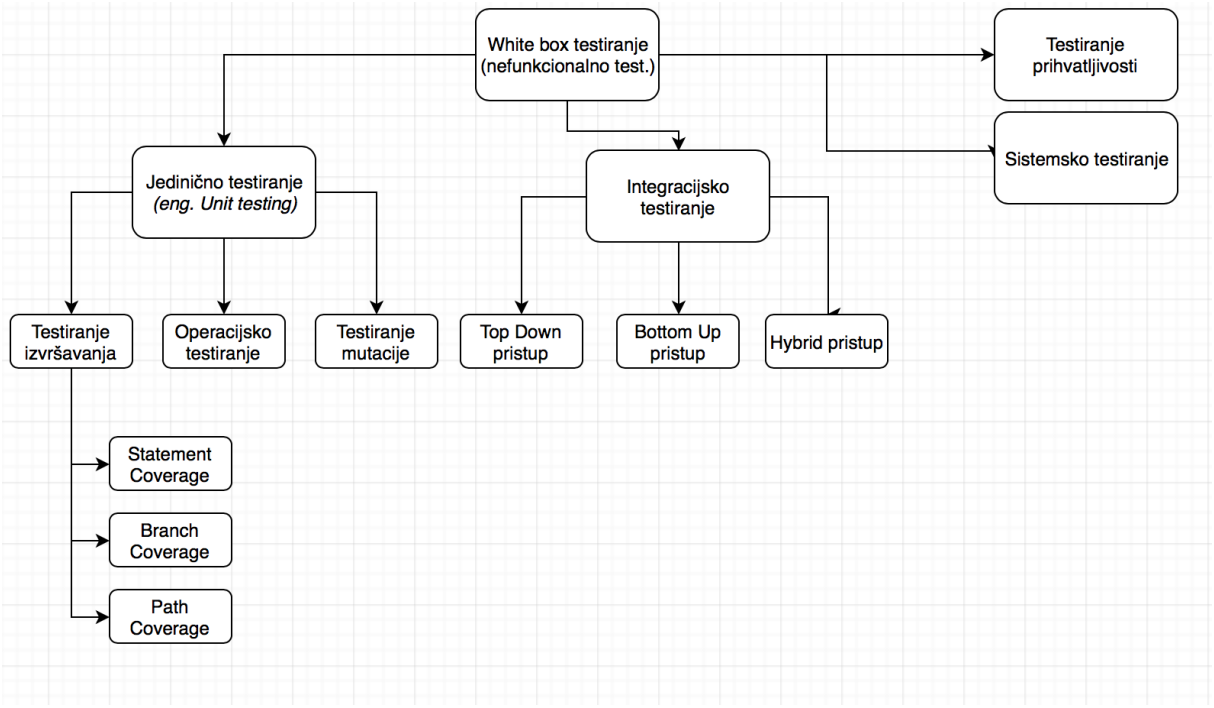
Ovakvom metodom testiranja ispituju se komponente, metode, objekti i aplikacije u cijelosti. Najvažnija stavka procesa testiranja je definiranje ponašanja koje se testira. Zbog toga programeri koji testiraju, odnosno ispitivači koda moraju imati dosta specifičnih znanja o kodu. Testeri primjenjuju testove na svakoj liniji koda i time otkrivaju sve možebitne anomalije. Da bi to bilo moguće, tester treba biti upoznat s time što njegov program treba raditi, odnosno kako se mora ponašati. Ako osoba koja testira zadovoljava sve navedene uvjete, tada se može govoriti o odstupanju programa od njegovog inicijalnog cilja.
Metoda bijele kutije ima više podjela:



3. Podjela metode Bijele kutije

Na slici iznad prikazana je podjela metode bijele kutije na dvije veće podjele, na jedinično i integracijsko testiranje. Strategija koju primjenjuje spomenuta metoda vrlo je zahtjevna i zahtijeva dosta vremena te se primjenjuje uglavnom na manje dijelove sustava generalno. Takva vrsta testiranja može se provoditi po nekim razinama.

Obično je veća podjela na jedinično testiranje, integracijsko, sistemsko i na testiranje prihvatljivosti kao što se može vidjeti na dijagramu ispod:



4. Nadopunjena podjela metode bijele kutije

Glavna zadaća postojanja ovoliko *levela* podjele metode bijele kutije jest da se cijeli sustav sagleda iz svih perspektiva te da se pronađu različite vrste grešaka koje sustav može sadržavati.

6.5 Testiranje komponenti, jedinično testiranje

Kao što i samo ime govori, jedinično testiranje (eng. *Unit testing*) vrsta je testiranja koja je fokusirana na ispitivanje pojedinih izoliranih komponenti sustava. Glavni uvjet za ostvarenje takvog ispitivanja mogućnost je promatranja komponente koju testiramo kao nezavisnu cjelinu koja se može odvojiti od kompletnog konteksta sustava i kao takva testirati. Komponente su dijelovi koda, njima se mogu smatrati objekti, klase, paketi itd. One mogu biti i dio korisničkog sučelja, kao i moduli za implementaciju složenih ili jednostavnih algoritama, moduli koji komuniciraju s ostalim dijelovima sustava itd.

```
package org.abilos;

public class Calculator {

    public long sum(long a, long b) {
        return a + b;
    }
}
```

5. Sum metoda

Na slici iznad prikazana je klasa *Calculator* s metodom *sum()* u programskom jeziku Java. Metoda prima dva parametra, a i b te vraća njihovu sumu. Vrlo jednostavna metoda i kao primjer ispod je napravljen test za nju:

```
package org.abilos

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {

    private Calculator calculator;

    @Before
    public void setUp() {
        calculator = new Calculator();
    }

    @Test
    public void testSum() {
        Assert.assertEquals("Sum is invalid", 2, calculator.sum(1, 1));
        Assert.assertEquals("Sum is invalid", 3, calculator.sum(1, 2));
        Assert.assertEquals("Sum is invalid", 4, calculator.sum(2, 2));
    }
}
```

6. Test za sum metodu

Testovi za metodu *sum()* napisani su u JUnitu, razvojnom programskom okruženju (eng. *Framework*) za programski jezik Java. Zanimljiva je činjenica da je na razvijanju JUnita radio programer koji je ujedno i jedan od najvećih zagovornika i začetnika razvoja programskog proizvoda vođenog testovima, Kent Beck. Metode koje se testiraju trebaju biti označene anotacijom *@Test* kao što se vidi i na primjeru iznad. Također se vidi i anotacija *@Before* koja je postavljena na funkciju *setUp()* koja se treba izvršiti prije testne funkcije *sum().* U JUnitu postoje još razne anotacije koje mogu definirati kada će se neka metoda odraditi, odnosno kojim redoslijedom. Na primjeru je prvo definirana metoda *setUp()* koja stvara *Calculator*, odnosno oponašani (eng. *Mock*) podatak koji je simulira kako bi se mogao iskoristiti u testu ispod. Da bi rezultati testa bili validni, potrebno je reproducirati stvarno ponašanje metode, i ono se, u ovom slučaju, dobije za početak kreiranjem testne metode *Calculator().*

Cijeli proces jediničnog testiranja izgleda tako da se određena komponenta izdvoji iz sustava i da se iskoristi u testnom kontekstu, odnosno u kontekstu koji je simulacija pravog sustava. Komponente koje se testiraju komuniciraju s drugim simuliranim komponentama i na temelju toga se dolazi do određenih rezultata. Postoje dvije vrste simuliranih komponenti ovisno o tome poziva li se njih ili one bivaju pozvane, a to su:

1. Driveri - Vrsta modula/komponenti koje simuliraju rad i ponašanje komponenti koje pozivaju druge komponente i shodno tome očekuju neki odgovor. Takve komponente uglavnom pokreću testove jer iniciraju pozive komponentama koje se idu testirati.

2. Dubleri - Vrsta modula/komponenti koje simuliraju rad realnih, primaju pozive te vraćaju rezultate onakve kakve bi vratile i realne komponente, njih ima više vrsta ovisno o njihovoj složenosti:

```
import { expect } from 'chai';
import 'mocha';

describe('isEmpty', () => {
    it('Test treba vratiti undefined ako je input polje prazno', () => {
        const result = isEmpty();
        expect(result).to.be.undefined;
    });
});
```

7. Primjer Dubler testa

Na primjeru sa slike test je napisan u programskom jeziku JavaScript koji provjerava da li je *input* polje prazno. Test prolazi ako je spomenuto polje prazno. U primjeru su korištene biblioteke Mocka i Chai. Expect je metoda kojoj se preda očekivani rezultat (koji se želi dobiti testom) i koja vraća *true* ili *false*.

2.1 Dummy - Najjednostavnija vrsta *dubler* testa. On je prazni *doubler*. Njegova je primarna zadaća omogućavanje povezivanja programa pružajući očekivanu vrijednost na za to odgovarajuća mjesta.

```
import { expect } from 'chai';
import { describe, it } from 'mocha';

describe('Dummy test: ', () => {
    it('TRUE must be TRUE', () => expect(true).to.be.true);
});
```

8. Primjer Dummy testa

Na slici iznad prikazan je jednostavan *dummy* test koji provjerava je li *true* uvijek i, ako je *true* uvijek, test uspješno prođe. Testiranje s primjera je odrađeno uz pomoć biblioteka Mocha i Chai.

2.2 Stub – Pomoćna vrsta dublera koja dodaje pojednostavljenu logiku pružajući različite rezultate.

2.3 Spy - Vrsta dublera koji je takozvani špijun koji snima interne parametre i informacije o stanju modula koji testiramo što omogućava napredniju validaciju.

2.4 Mock - Značenje riječi *mock* je oponašanje, što govori puno o tome za što se *mock* i koristi - za oponašanje. To je napisana lažna komponenta definirana testom koja ima za zadaću provjeru ponašanja za pojedini test tako što provjerava redoslijed pozivanja komponenti i vrijednost parametara.

2.5 Simulator - Komponenta koja obuhvaća sve, ona osigurava točniju provjeru funkcionalnosti za koje pišemo *dubler*.

Driveri i *dubleri* se u dosta slučajeva pišu ručno, odnosno manualno programiraju, osim u nekim specifičnim slučajevima kada driver nije posebni tester koji sam manualno poziva modale iz korisničkog sučelja.

Za implementaciju *drajvera* se, ovisno o programskom jeziku u kojem je pisan dio koda, može koristiti JUnit(Java), NUnit(programski jezik C#), DBUnit (SQL). *Dubleri* se implementiraju jako jednostavno, zapravo se samo definiraju komponente koje će simulirati stvarne i u njima se definira ono što se vraća za pojedine dijelove ulaznih podataka. Postoje razne biblioteke koje olakšavaju kompletan proces definiranja takvih komponenti. Neke od njih su: Mocka, Chai, JMock, NMock, Rhino Mock itd.

Testiranje jedinica koda može biti i zahtjevan i ne tako ekonomičan proces kada gledamo na programski proizvod kratkoročno. Međutim, kada je komponenta koju testiramo ispravno dizajnirana i kada je sve od početka izrade išlo dobrim smjerom, tada broj testova može biti drastično smanjen, a time i implementacija lakša.

6.6 Integracijsko testiranje

Integracijsko testiranje vrsta je testiranja koja se primjenjuje neposredno nakon izvršavanja jediničnih testova. Međutim, često se postavlja pitanje zašto se nakon uspješno provedenog testiranja jediničnim testovima mora testirati sve to isto samo povezano u cjelinu? Sve to izgleda doista dobro na prvi pogled ali realnost nije baš najljepša. Problem je vrlo često u povezivanju komponenti. Neki se podaci mogu tijekom povezivanja jedne komponente s drugom izgubiti ili jedna komponenta može prepisati podatke drugoj. Neke funkcije mogu dovesti do neplaniranih akcija ili poremetiti glavnu funkciju. Neke male nepravilnosti koje su propuštene kroz prethodno testiranje spajanjem komponenti mogu dovesti do prilično kaotične situacije.

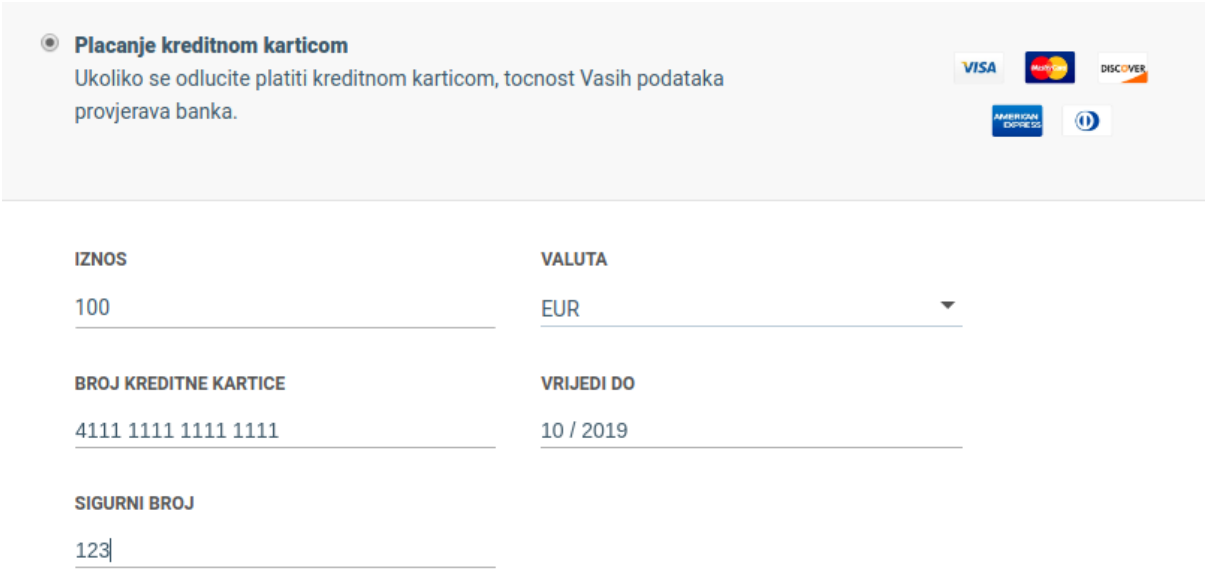
Kao dobar primjer važnosti upotrebe integracijskog testiranja može se uzeti NASA-in projekt *Mars orbiter* koji je propao zbog programske pogreške, odnosno zbog promaknuća integracijskog testiranja.
Mars Climate Orbiter bio je NASA-in projekt, odnosno sonda koju su poslali do Marsa kako bi evidentirala i bolje istražila stanje na Marsu, jer se pretpostavlja da je nekad cijeli Mars bio pod vodom, te odgovorila na pitanja zašto je voda nestala i gdje. Međutim, nakon 286 dana što je putovala do Marsa sonda je ispalila svoj motor kako bi se uspješno gurnula u orbitu, ali motor je krenuo pucati, a letjelica je došla na 60 km, 100 km bliže nego li su stručnjaci planirali, i 25 km ispod razine na kojoj bi stvari mogle funkcionirati ispravno. Što se dogodilo?
Naime, NASA je izgubila 125 milijuna dolara jer je projekt propao (Lloyd, 1999:1):

''NASA je izgubila je Marsov orbiter vrijedan 125 milijuna dolara jer je tim inženjera Lockheed Martin koristio engleske mjerne jedinice dok je tim iz agencije koristio konvencionalni metarski sustav za ključnu operaciju svemirske letjelice, navodi se u četvrtak objavljenom pregledu. Jedinica neusklađenosti spriječila je prijenos navigacijskih informacija između tima Mars Climate Orbiter u letjelici Lockheed Martin u Denveru i letačkog tima u NASA-inom laboratoriju Jet Propulsion u Pasadeni u Kaliforniji.''

To je jako poznat primjer kada nije bilo upotrjebljeno integracijsko testiranje, a bilo je vrlo potrebno iz više aspekata, što iz znanstvenog što iz ekonomskog.
U teoriji, integracijsko testiranje jedan je dio procesa testiranja kod kojeg se nakon spajanja komponenti one testiraju kao skup. Generalno, proces integracijskog testiranja može se složiti u par stavki:

- Grupiranje komponenata koje su prethodno prošle *unit* testove u jednu veću cjelinu.
- Testiranje koje je prethodno definirano i isplanirano.
- Uspješnim prolazom svih testova dobije se sustav koji je spreman za daljnje testiranje, testiranje sustava.

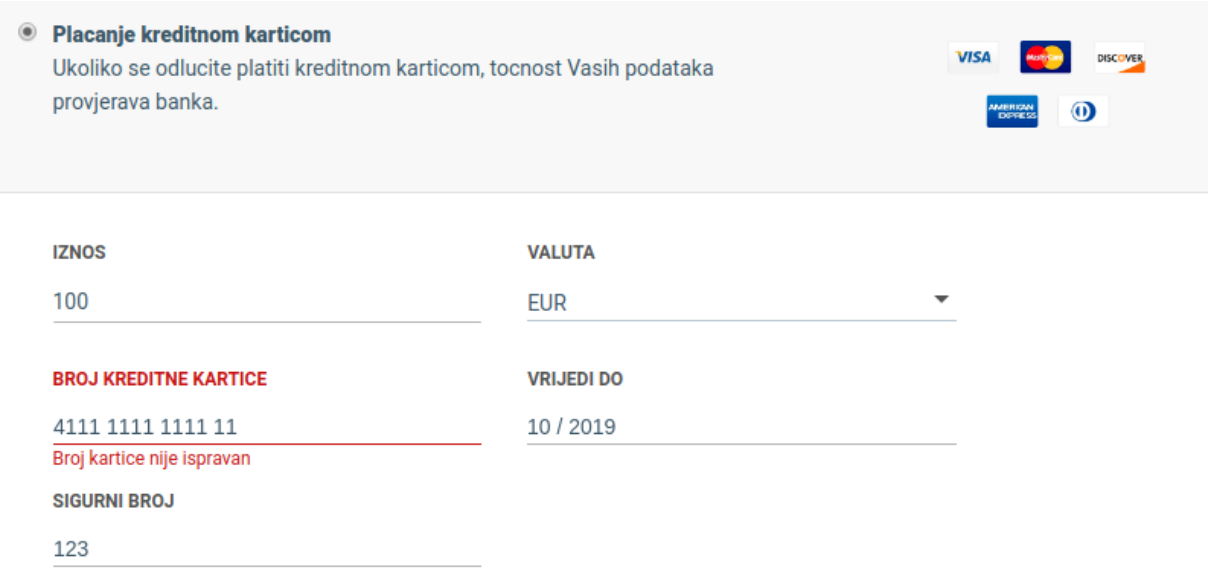
Kako bi se što bolje razjasnio pojam integracijskog testiranja, prikazuje se slučaj uporabe (eng. *Use case*) plaćanja kreditnim karticama. Korisnik kupuje tipkovnicu preko interneta. Na većini internet aplikacija koje pružaju neke usluge koje korisnik želi kupiti postoji forma koju korisnik treba popuniti s podatcima sa svoje kreditne kartice.



9. Forma za unos podataka

Kako bi pružatelj usluge koju korisnik kupuje bio siguran da su podatci o kreditnoj kartici ispravni, on mora komunicirati s korisnikovom bankom. Ta komunikacija se

odvija programskim putem. Banka očekuje da će dobiti objekt koji sadrži iznos koji treba biti tipa *float*, valutu koja treba biti tipa *string*, broj kreditne kartice koji treba biti tip *number*, vrijedi do treba biti tipa *string* i sigurni broj koji treba biti *number*. Programer koji je izradio formu trebao je osigurati da podaci koje korisnik unese budu točnih tipova podataka. Programer šalje objekt s podacima koje je korisnik unio banci odmah nakon što korisnik popuni posljednje polje. Komunikacija s bankom se događa nakon sto korisnik popuni pojedino polje. Ako je korisnik unio pogrešan podatak, banka javi grešku.



10. Forma za unos s greškom

Ovdje je riječ o integraciji između dva programa, forme za plaćanje od pružatelja usluge i banke. Integracijskim testiranjem provjerava se je li integracija ta dva servisa uspješna ili ne.

Postoji više vrsta integracijskog testiranja:

1. Integracijska metoda Veliki prasak (eng. *Big Bang*)

Takva metoda drži se principa da se integriraju sve postojeće komponente bez ikakvog prethodnog testiranja. Mnogo komponenti spoji se u cjelinu te se onda program testira. U dosta slučajeva takav pristup ima također kaotičan rezultat. Kada se spoji jako puno koda i onda se sve skupa testira, rezultat donosi jako puno

pogrešaka. Čim se jedna ispravi, odmah je na redu sljedeća i tako nebrojeno puno puta dok se ne riješe sve. To je proces koji traje i traje te nije optimalan.

2. Poboljšani Veliki prasak

Kako se metoda Velikog praska nije pokazala optimalnom, krenulo se s njezinom optimizacijom. Došlo se na ideju da se krene s integracijom komponenti nakon prethodno provedenog testiranja što je bio revolucionaran korak u odnosu na prvu metodu. Međutim, i dalje kada pogreške bivaju pronađene postoji problem alokacije problema u kodu.

3. Inkrementalni pristup

Kako i samo ime govori, inkrementalni pristup je pristup korak po korak. Implementira se dio po dio koda i zatim se testira također dio po dio. Jako mali koraci te primjena takve metode dovode do jednostavnijeg uočavanja i ispravljanja grešaka, potpunijeg testiranja korisničkog sučelja te bolje podloge za testiranje sustava generalno. Takav pristup je danas najzastupljeniji zbog svoje efektivnosti, a i zato što je njime riješen problem alokacije grešaka.

Inkrementalni pristup testiranju ima par načina primjene:

3.1 Primjena integracijskog testiranja odozgo prema dolje - Tijekom integracije testiranje se vrši prema dolje niz hijerarhiju, od glavnog programa naniže, razvija se kostur sustava i popunjava se komponentama. Moduli se uključuju ovisno o hijerarhiji.

3.2 Primjena integracijskog testiranja od dna k vrhu - Prvo se integriraju komponente koje sadrže najvažnije i najčešće funkcionalnosti te se one dodaju i testiraju, a zatim ostale.

3.3 Funkcijska integracija (eng. *Sandwich*) - Najčešće korištena metoda integracije u praksi jer kombinira pristup odozgo prema dolje i od vrha ka dnu. Integriraju se komponente u konzistentne skupove bez obzira na hijerarhijski slijed.

Prednosti takve integracije:

- Lako je napraviti testove.
- Komponente se mogu integrirati neposredno nakon implementacije.

Nedostaci:

- Kao i kod Big bang metode, greške se teško alociraju i izoliraju.

7. Uloge i organizacija kod testiranja

Kako se tada testiranje programskog proizvoda primijetilo kao jako bitan i neizostavan proces, tako je ostalo i danas. U današnje vrijeme organizacije koje se bave testiranjem proizvoda mogu biti kompletno odvojene od proizvođača, odnosno od razvojnog tima. Današnji alati i metode omogućavaju testiranje na osnovi samo poznavanja nekog programskog jezika. Članovi tima koji se bave testiranjem uglavnom imaju različite uloge za pojedine faze testiranja. Naime, neke firme imaju osim posebnih članova tima koji se bave testiranjem i posebne timove koje rade samo to. Budući da je to danas jako velik posao te je potrebno uzeti puno stvari u obzir prije i tijekom testiranja kako bi se osigurao programski proizvod koji zadovoljava ili nadmašuje očekivanja, takav posao se u praksi dodjeljuje cijelom timu ljudi. Cijeli proces testiranja mora se odvijati u svim fazama razvoja programa. Prije svega, taj proces se ne svodi samo na ispunjenje danih zahtjeva i definiranje nekih kriterija.

Klasična praksa je da programeri testiraju jesu li zadovoljeni zahtjevi korisnika. Ako pojedini korisnički zahtjev može biti uspješno odrađen i ako je postignut zadovoljavajući rezultat, testiranje se smatra uspješno završenim. Također, u praksi se jednom korisničkom zahtjevu dodjeljuje jedan komplet, to jest set testova i ako taj komplet uspješno prođe, korisnički zahtjev je ispunjen.

Upravljanje (eng. *Management*) u informacijskim tehnologijama, u drugu ruku, nije fokusiran samo na zahtjeve korisnika, zapravo nije uopće fokusiran na zahtjeve. Ljudi u tom segmentu posla su upoznati s informacijama kao što su: koje dobre klijente imaju, novčane transakcije koje se odrađuju, politikom poslovanja tvrtke i s vrstom

problema s kojima su se možda susretali nekada u prošlosti. Prema tome oni testiraju program na potpuno drugačiji način. Takva vrsta testiranje je bazirana striktno na iskustvu i veliki je pozitivni dodatak na način testiranja koji koriste samo programeri.
Iako se ta kombinacija već na prvu čini kao dovoljno dobra i možda potpuno sigurna, ona također ima neke manjkavosti.

Često se postavlja pitanje tko provjerava zahtjeve korisnika? U velikom se broju slučajeva i ne provjeri sukladnost postojeće specifikacije s postojećim prethodno specificiranim dizajnom. Takva vrsta problema dolazi zapravo od stvoritelja zahtjeva, odnosno od korisnika. Ne vizualiziraju uvijek ponašanje proizvoda koji treba biti napravljen prije same izrade. Može se također dogoditi i da završeni proizvod sadržava grešku koja je postojala u korisničkim zahtjevima koje nitko nije prethodno testirao. Vrlo se često dogodi da zahtjevi nisu kompletni ili su konfuzni. Oni koji daju takvu vrstu zahtjeva očekuju od programera da predvide ostatak zahtjeva ili misle da je to očigledno pa se neka funkcionalnost podrazumijeva. Osobe koje su inicijalno postavljale te zahtjeve mogu imati totalno drugačiju sliku funkcionalnosti i dizajna svega toga. Slijedi jedan običan i uobičajen primjer toga.

U zahtjevu korisnika postoji definiran jedan redoslijed obavljanja neke aktivnosti. Jedan od korisničkih zahtjeva može biti izrada programa za mobitel koji za jednu od zadaća ima da pritiskom bojeva 1234 može otključati uređaj i programer to napravi. Međutim, dogodi se da korisnik koji dobije na korištenje taj program unese brojeve krivim redoslijedom, npr. 3241 i ne uspije otključati uređaj, ne dobije nikakvu poruku o grešci niti dobije informaciju da postoji unaprijed određeni redoslijed koji dozvoljava obavljanje određene aktivnosti (u ovom primjeru otključavanja mobilnog uređaja). Što se dogodilo? Taj zahtjev nije postojao u specifikacijama dizajna. Takve slučajeve imamo i dan danas u svakom programskom proizvodu i oni se kao takvi pojavljuju u raznim oblicima. Oni programi koji su dovoljno dobri korisnicima će javiti grešku i poruku koja ih upućuje, odnosno informira o problemu, odnosno o toj grešci i informaciju kako si mogu pomoći.

Osobe koje se profesionalno bave testiranjem, odnosno čija je uloga prvenstveno testiranje postavljaju sebe u poziciju krajnjeg korisnika. To je glavno polazište. Takva

osoba može predvidjeti pogrešne poteze koje korisnik može napraviti i može optimizirati slučaj da korisnik bude zadovoljan. Dobro je postaviti se u poziciju djeteta od 12 godina koje se treba snaći u programu koji se izrađuje. Treba razmišljati koji se koraci mogu napraviti, što se potencijalno može pokvariti i može li se uopće kretati po programu. Osobe koje su u stanju to napraviti dobri su testeri jer oni svojim kreativnim razmišljanjem i iskustvom popunjavaju manjkavosti u zahtjevima korisnika.

Potrebno je definirati ciljeve koji će voditi do podizanja kvalitete i sigurnosti na jedan viši *level*. Za neku takvu aktivnost također se trebaju realizirati određeni propisi i djelatnosti koje će obavljati sudionici procesa. U najboljem slučaju cijela aktivnost testiranja bude dokumentirana ili barem glavni dijelovi:

- ciljevi provođenja testiranja
- raspored zaduživanja
- odgovornosti za planiranje testova, za izvođenje, za evaluaciju.

8. Testiranje i dokumentacija

Zašto se povezuju pojmovi testiranje i dokumentacija? U modernom programiranju ta se dva pojma pronalaze često zajedno. Kod razvoja programa vođenog testiranjem dobra praksa bi bila da se testovi pišu prije bilo kakve implementacije koda. Da bi se testovi uspješno napisali, prvo trebaju pasti i prema tome se modificira kod, prilagodi se testovima. Osim koda potrebno je poznavati poslovnu logiku, odnosno poznavati slučajeve uporabe (eng. *Use Case*) koje trebamo pokriti napisanim kodom. Kada se sve te bitne stvari raspišu i projekt isplanira kreću se pisati testovi. Iz tipičnih jediničnih testova da se iščitati što određena metoda koja je testirana radi i koja su njezina željena ponašanja. Kvalitetan pristup testiranju programa nije ni malo lak posao, i zahtjeva veliku odgovornost.

Većina programera ne čita napisanu dokumentaciju o funkcionalnostima sustava, oni se više vole bazirati na kod. Realno gledajući, iz njihove perspektive tu ni nema ništa loše. Kako bi lakše razumjeli metode, objekte ili klase, većina programera će prvo pogledati kod. Sada se dolazi do dijela gdje se povezuje testiranje i dokumentacija. Dobro napisani jedinični testovi omogućavaju baš to, rad s dokumentacijom i specifikacijom funkcionalnosti u kodu. Naime, dobro napisani testovi sadržavaju i poslovnu kao i tehničku logiku. Iz njih se da iščitati što se očekuje od metode da radi i koji rezultat treba vratiti. Kao rezultat efektivno napisanih testova dobije se tehnička dokumentacija. Također, testovi mogu predstavljati vrlo bitan dio dokumentacije zahtjeva kupca programskog proizvoda. Oni mogu sadržavati odgovore na mnoga pitanja koja su možda najbitniji dio svake dokumentacije, kao što su npr:

- Kako sustav treba raditi?
- Što se od sustava generalno očekuje?

Od testova kao tehničke dokumentacije dobiva se zaista mnogo, međutim jesu li testovi sasvim dovoljna dokumentacija? Naravno da ne, ali čine vrlo bitan dio. I dalje je jako važno imati česte iteracije s korisnikom, imati sažeto napisanu poslovnu logiku koja objašnjava cijeli poslovni koncept. Dobra dokumentacija programskog proizvoda rezultat je kombiniranja napisane poslovne logike i dobrih testova.

Bitno je imati dobru dokumentaciju i to nije ni malo jednostavan zadatak. Kvalitetan pristup dokumentaciji je popraćen brojnim preprekama. Neke programeri smatraju da kod govori sve sam za sebe ne obazirući se na poslovnu logiku unutar koda, neki smatraju da je to pisanje dokumentacije sporedan proces koji se može i ne mora odraditi, a neki pak nikad ni ne napišu dokumentaciju. Kada se kvalitetno pristupi dokumentaciji, onda se u vidu ima jedna glavna stvar – dokumentacija može direktno utjecati na poslovni uspjeh. Kod nekih poslodavaca i IT firmi dolazi do česte promjene zaposlenih programera. Oni odlaze na druga radna mjesta i razvijaju nova iskustva na drugim mjestima. Kako bi firma što bolje i brže zamjenila te zaposlenike što su otišli, i kako bi ih što brže naučila stvarima koje su njihovi prethodnici znali, potrebna je kvalitetna dokumentacija. Dugotrajna obuka novih zaposlenika nije ni malo ekonomična ni za jednu firmu a nekvalitetna obuka može rezultirati kaotičnim situacijama. Ukoliko neki novi zaposlenik kroz nekvalitetno napisanu dokumentaciju krivo shvati svrhu koda može dobiti pogrešnu sliku kako sustav radi i kasnije ju prenijeti na krajnje korisnike. Ona mora biti jasna i pokrivati sve glavne točke i aspekte rada. Dobra dokumentacija je posljednji korak ka kvalitetnom programu.

Zaključak

Nakon pojave testiranja i početaka razvoja programa vođenog testovima te njegove šire primjene počeli su se razvijati stabilniji i kvalitetniji programski proizvodi. Iako je oduvijek fokus bio na kvaliteti, primjenom ovakvog pristupa razvoju programa velika važnost je stavljena i na njegovu stabilnost. Kako je novac najbitnija stavka cijelog procesa i kako se proizvod uglavnom radi najbolje moguće u što kraćem vremenu radi ekonomskih razloga, testiranje uvelike pridonosi efektivnosti. Naime, ako program od početka biva pokriven testovima i ako se od početka uloži malo vremena na testiranje i provjere, do kraja procesa doći će se puno prije nego metodom pokušaja i pogreški. Osim toga, jeftinije je ukloniti grešku na početku izrade nego u nekoj kasnijoj fazi ili u produkciji u najgorem slučaju.

Kako bi se to spriječilo postoji jako puno metoda testiranja, neke su automatizirane, za neke je potreban tester, a neke rade i programeri. Od početka izrade programa do kraja bitno je provoditi testiranje onim redoslijedom kako je to zamišljeno. Od provjere zahtjeva korisnika, specifikacije, dizajna itd. do krajnje provjere sustava generalno. Svako od testiranja bitno je provesti u određenoj fazi razvoja programskog proizvoda. Ako je tok ispravno popraćen, testiranje će biti uspješno provedeno, a proizvod stabilan i kvalitetan te spreman za krajnje korisnike.

Razvojem složenijih programskih proizvoda javlja se i potreba za boljim alatima za testiranje i testerima tako da nema dvojbe da je to jedna od grana tehnologije koja će rapidno nastaviti s razvojem. Nema puno protuargumenata protiv provođenja testiranja iako je jedan, koji programeri vole spomenuti, vrijeme. Neki put vrijeme predstavlja problem. Testiranje je uglavnom dosta brz proces, ali nekada se zna

razvući na duže periode i privremeno se može činiti kao neefikasna metoda, međutim dugoročno gledano itekako je isplativa. Ako se na početku projekt stavi na stabilne temelje, stabilniji će se dalje i razvijati. U svakom slučaju bolje je greške alocirati, prepoznati i rješavati tijekom razvoja, a ne na kraju kada one mogu biti drastične što po obujmu što po cijeni popravka koju nose sa sobom. Kvalitetnom i ispravnom primjernom *test-driven-developmenta* dobije se ispravan proizvod, zadovoljni poslodavci, ekonomičan proizvod te najbitnije od svega, zadovoljstvo krajnjih korisnika.

Popis literature

1. MYERS, G. J., BADGETT, T. i SANDLER, C. (2012.) *The art of Software Testing.* 3rd edition. New Jersey: John Wiley & Sons, Inc.

2. BECK, K. (2002.) *Test-Driven Development By Example.* Boston: Addison Wesley.

3. MANGER, R. (2013.) *Softversko Inžinjerstvo*, skripta, Zagreb. Dostupno na: http://ttl.masfak.ni.ac.rs/SUK/Softversko%20inzenjestvo.pdf (2019-4-25)

4. HASTIE, S. i WOJEWODA, S. (2015.) Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch, *InfoQ* Dostupno na: https://www.infoq.com/articles/standish-chaos-2015. (2019-4-7)

5. LLOYD, R. (1999.) Metrics mishap caused loss of NASA orbiter. Dostupno na: http://edition.cnn.com/TECH/space/9909/30/mars.metric.02/ (2019-5-1)

6. James Bender i Jeff McWherter (2011.) Test-Driven Development with C#, Developing real world applications with TDD. Dostupno na: https://www.pdfdrive.com/professional-test-driven-development-with-c-developing-real-world-applications-with-tdd-e50930836.html (2019-4-7)

Sažetak

Razvoj programskog proizvoda vođenog testovima zaista je bitna stavka koju treba sadržavati proces izrade programa. To je metodologija koja poboljšava stabilnost programa, izradom i pokretanjem automatiziranih testova prije stvarnog razvoja projekta ili aplikacije.

Razvoj programa vođenog testovima nije tako kompliciran proces za razumijevanje ili za implementiranje. Potrebno je znati tri glavna koraka: pisanje testa, prolazak testa i refaktoriranje istog. Takva praksa nazvana je Crveni, Zeleni i Refaktor proces. Postoji mnogo razloga zašto treba koristiti razvoj vođen testovima, jedan od njih je ekonomičnost. Korištenjem testiranja tijekom procesa dobije se stabilan, visoko kvalitetan i program bez grešaka koji zahtjeva manje popravaka, što znači manje novca koji će biti potrošen za održavanje. U današnjem svijetu programiranja sve više programera usvaja ili želi usvojiti način razvoja programa vođenog testovima. Također postoje i oni koji ne žele primijeniti praksu s testovima jer misle da je neisplativo i beskorisno te se može činiti kao dug i iscrpljujući proces.

Bilo kako bilo, uvijek će postojati dvije strane i dva različita mišljenja oko korištenja testova kroz razvoj projekta, ali sve više i više velikih tvrtki kao sto su Google ili Microsoft podržavaju metodologiju koja uključuje testiranje jer je zaista bitna i na kraju krajeva ekonomična metoda.

Ključne riječi: TDD, razvoj, programski proizvod, programiranje, testiranje, testovi, komponente, jedinice

Abstract

Test-driven development is a really important thing to have in the software building process. It is a methodology which improves software stability with developing and running automated test before the real development of the project or application.

Also, TDD is not so difficult to understand or implement. There are three main steps which are: write a test, make it pass and refactor it. This practice is also known as red, green and refactor. There is a lot of reason why TDD should be used, such as economic. With TDD you get stable, high-quality and error-free software project and that implies fewer repairs which means less money to spend. Nowadays, a lot of developers try to adopt TDD and use it appropriately. There are also people who don't want to adopt TDD practice because they think that it's ineffective and that spends a lot of time to implement it. In the beginning, it can seem like a durable process and a bit exhausting.

Anyhow, there will always be two sides and two kinds of opinions about using TDD, but more and more gigantic IT companies such as Google or Microsoft are huge fans of this methodology just because of its practicality and efficiency and the most important of all because of its economics.

Key words: TDD, development, software, implement, testing, tests, components, units

Popis slika

1. PROCES STVARANJA PROIZVODA 12
2. DIJAGRAM PODJELE TESTIRANJA 19
3. PODJELA METODE BIJELE KUTIJE 23
4. NADOPUNJENA PODJELA METODE BIJELE KUTIJE 24
5. SUM METODA 25
6. TEST ZA SUM METODU 25
7. PRIMJER DUBLER TESTA 27
8. PRIMJER DUMMY TESTA 27
9. FORMA ZA UNOS PODATAKA 30
10. FORMA ZA UNOS S GREŠKOM 31