Automatizacija testiranja ogledne aplikacije

Džale, Doris

SVEUČILIŠTE U SPLITU

PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

Automatizacija testiranja ogledne aplikacije

Doris Džale

Split, rujan, 2021.

Temeljna dokumentacijska kartica

Diplomski rad

Sveučilište u Splitu

Prirodoslovno-matematički fakultet

Odjel za informatiku

Ruđera Boškovića 33, 21000 Split, Hrvatska

AUTOMATIZACIJA TESTIRANJA OGLEDNE APLIKACIJE

Doris Džale

SAŽETAK

Automatizacija testiranja je proces pisanja programskog koda koji će samostalno pokretati zadane testne scenarije te provjeravati ispravnost sustava. U uvodnom dijelu diplomskog rada definirani su pojmovi vezani uz temu, a to su računalni program, korektnost programa, vrste specifikacije programa, vrste pogrešaka. Zatim ovaj diplomski rad uvodi čitatelja u teoriju o testiranju sustava, te prikazuje značaj testiranja za razvoj aplikacija. Navedene su prednosti i nedostatci načina testiranja. Opisana je automatizacija testnih procesa, te način automatiziranja koristeći C# ugrađeni *unittest* okvir MSTest. Navedene su značajke automatizacije te alati za automatiziranje. Uz pomoć stečenih znanja u praktičnom dijelu rada prikazan je primjer automatiziranja testnih scenarija, napravljena je integracija jediničnih testova za aplikaciju Blackjack.

Ključne riječi: korektnost programa, verifikacija, programiranje, testiranje, jedinično testiranje, automatizacija.
Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: [64] stranice, [30] grafička prikaza i [3] tablice
Izvornik je na hrvatskom jeziku.

Mentor: **Dr.Sc. Ivica Boljat**, *izvanredni profesor Prirodoslovno matematičkog fakulteta, Sveučilišta u Splitu*
Voditelj: **Dr.Sc. Tonći Dadić**, *viši predavač Prirodoslovno matematičkog fakulteta, Sveučilišta u Splitu*
Ocjenjivači: **Dr.Sc. Ivica Boljat**, *izvanredni profesor Prirodoslovno matematičkog fakulteta, Sveučilišta u Splitu*
Dr.Sc. Tonći Dadić, *viši predavač Prirodoslovno matematičkog fakulteta, Sveučilišta u Splitu*
Dr.Sc. Saša Mladenović, *profesor Prirodoslovno matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen:

Basic documentation card

Graduation Thesis

University of Split

Faculty of Science

Department of Computer Science

Ruđera Boškovića 33, 21000 Split, Croatia

AUTOMATIZATION OF CHOSEN APPLICATION TESTING

Doris Džale

ABSTRACT

Testing automation is the process of writing program code that will independently run default test scenarios and check the correctness of the system. In the introductory part of the diploma thesis, the terms related to the topic are defined, namely computer program, correctness of the program, types of program specifications, types of errors. Then this thesis introduces the reader to the theory of system testing, and shows the importance of testing for application development. The advantages and disadvantages of the test method are listed. The automation of test processes is described, as well as the method of automation using the C # built-in *unittest* framework MSTest. Automation features and automation tools are listed. With the help of the acquired knowledge in the practical part of the paper, an example of automation of test scenarios is presented, the integration of unit tests for the Blackjack application is made.

Keywords: program correctness, verification, programming, testing, unit testing, automation.

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of: [64] pages, [51] figures, [2] tables and [23] references.

Original language: Croatian

Mentor: **Ivica Boljat, Ph.D.** *Associate Professor of Faculty of Science, University of Split*

Supervisor: **Tonći Dadić, Ph.D.** *Senior lecture of Faculty of Science, University of Split*

Reviewers: **Ivica Boljat, Ph.D.** *Associate Professor of Faculty of Science, University of Split*

Tonći Dadić, Ph.D. *Senior lecture of Faculty of Science, University*

Saša Mladenović, Ph.D. *Full Professor of Faculty of Science, University of Split*

Thesis accepted:

Sadržaj:

Uvod

Proces razvoja aplikacija nije nimalo jednostavan. Velika količina vremena potroši se na analiziranje potreba korisnika. Nakon analize i kreirane funkcionalne specifikacije implamentacija može početi. Nakon uspješne implementacije aplikacija se isporučuje korisniku. Testiranje aplikacija je proces koji se obavlja prije isporuke korisniku, jer je vrlo vjerojatno da se tokom implementacije pojave greške u sustavu. Zbog toga je potrebno provjeriti korektnost s obzirom na specifikaciju, a to je ispunjava li aplikacija zahtjeve specifikacije, a sve kako bi osigurali zadovoljstvo korisnika prije isporuke. Kako bi isporučili što kvalitetniji proizvod potrebno je testirati sustav na nekoliko razina. Ovisno o stanju sustava te mjestu u razvojnom ciklusu odredit će se vrsta tetiranja. Ako je implementacija tek započeta ili je pri kraju primjenjivati će se različiti načini testiranja. Kako je većina sustava vrlo opsežna i sadrži veliki broj funkcija, potrebno je uložiti puno vremena u proces testiranja. Jer ako se testiranje ne provede na detaljan i ispravan način, velika je vjerojatnost da će aplikacija imati bugove. Budući da se želi smanjiti potreba za velikim utrošenim vremenom potrošenim na ručni rad testera, moguće je automatizitati testiranje. Automatizacija testiranja je proces pisanja programskog koda koji će samostalno pokretati zadane testove te provjeravati ispravnost.

U ovom radu opisat ćemo automatsko testiranje ogledne aplikacije korištenjem korisničkog sučelja Microsoft Visual Studio u programskom jeziku C#. Primjenom stečenog teorijskog znanja izradit ćemo testove na primjeru projekta. Rad se sastoji od pet poglavlja. Prvi dio rada opisuje pojmove vezane uz temu, a to su računalni program, korektnost programa, vrste specifikacije programa, vrste pogrešaka. Drugi dio opisuje načine verifikacije programa: testiranje i formalna verifikacija, zatim vrste testiranja: po principu crne kutije, bijele kutije / inspekcija koda. Treći dio definira kategorizaciju vrsta testiranja. Četvrti dio opisuje automatsko testiranje pojam i svrhu, te prikazuje izabrani alat za jedinično testiranje. Zadnje poglavlje, odnosno praktični dio rada, opisuje izradu testova u C#, korištenjem ugrađenog *unittest* okvira MSTest za oglednu aplikaciju, igru Blackjack. Osnovna ideja je izraditi jedinice testiranja (*engl. testing units*), te pokazati provođenje automatskog testa. Poglavlje je podijeljeno na dva glavna dijela, a to su unit testovi i integracija.

1. Računalni program

Računalni program je skup, odnosno niz naredbi koje računalu određuju što treba činiti. Takav poredani slijed naredbi tvori detaljan plan ili postupak rješavanja određenog zadatka ili problema pomoću računala. Prema tome, možemo reći da je naredba (*engl. command*) temeljni element računalnog programa. Računalni programi se pohranjuju kao datoteke na tvrdom disku računala, ili u memoriji računala, i omogućuju računalu da izvrši niz naredbi uzastopce ili čak isprekidano. Kada korisnik pokrene računalni program, datoteku čita računalo, a procesor čita podatke u datoteci kao popis uputa. Računalo se može smatrati vrlo poslušnim „psom“, jer osluškuje svaku naredbu i čini sve što naredba kaže. Za računalne programe koji su pohranjeni na tvrdom disku računala i prilagođeni izvođenju na računalu kaže se da su instalirani.

1.1. Vrste računalnih programa

Ideju o interno pohranjenom programu uveo je krajem četredesetih godina prošlog stoljeća američki matematičar rođen u Mađarskoj Jhon von Neumann. Prvo digitalno računalo dizajnirano s internim kapacitetom programiranja bio je EDVAC (akronim za električno diskretno varijabilno automatsko računalo) izgrađen 1949. godine. [1]

Zbirka računalnih programa, biblioteka i srodnih podataka naziva se softverom (*engl. software*) odnosno programskom podrškom ili programskom potporom. Programska podrška se dijeli na sistemsku programsku podršku i aplikacijsku programsku podršku. Sistemsku programsku podršku čini skup računalnih programa pripravljenih za prihvaćanje i izvršenje korisničkih programa, dok aplikacijsku programsku podršku čine računalni programi za rješavanje određenog zadatka sastavljeni od korisnika računalnog sustava.

Sistemski softver obuhvaća:

- operacijski sustav (OS) – računalni program ili skup računalnih programa koji se pokreću odmah pri uključenju računala i objedinjuju sve dijelove računala u svrhovitu cjelinu, te je podloga svim ostalim programima koji se izvode na računalu. Popularniji operacijski sustavi:
  - MS-DOS,
  - Windows 3.1, 95, 98, Me, XP, NT, 2000, CE i dr.,

- UNIX,
- LINUX

- programe prevoditelje - programski sustavi koji računalne programe napisane u programskom jeziku visoke razine prevode u ekvivalentne programe programskog jezika niske razine.
- pomoćni (servisni, uslužni) programi - općenito se radi o računalnim programima koji predstavljaju sponu između sistemske i aplikacijske programske podrške.[2].

Računalne programe pišu programeri, a postupak kreiranja računalnog programa naziva se programiranje. Mentalni proces predstavljanja i implementacije određenog algoritma najprije u simboličnom obliku (npr. pseudokodu) a potom kodiranjem, pisanjem programskog koda u programskom jeziku osmišljenom za komunikaciju s računalom. Svaki programski jezik određuje ograničen skup riječi posebnog značenja (ključne riječi)(*Slika 1.*), te propisana pravila slaganja tih ključnih riječi u naredbe.

Slika 1. *Ključne riječi*

Računalni program se priprema tako da se prvo formulira zadatak, a zatim se izrazi u odgovarajućem programskom jeziku u obliku izvornog kôda *(engl. source code)*. Izvorni kôd (koji se također naziva izvor ili kôd) je bilo koji zbirni kôd napisan programskim jezikom visoke razine. To je verzija softvera koju je izvorno napisao (tj. tipkao u računalo) čovjek običnim tekstom (tj. ljudsko čitljivim alfanumeričkim znakovima) koji se koristi za pisanje uputa za

računalni program. Postoji mnogo programa koji se mogu koristiti za pisanje izvornog kôda na željenom programskom jeziku, u rasponu od jednostavnih uređivača teksta opće namjene (kao što je *Notepad* u Microsoft Windows-u) do integriranih razvojnih okruženja od kojih su neki od najpopularnijih programskih jezika C, C++, C#, Pascal, Java, PHP, Python i dr. Programski jezici su čitljivi i razumljivi za ljude ali ne i za računala. Prema tome kako bi računalo ili drugi mikroprocesorski proizvodi mogli izvršiti upute napisane na jednom od programskih jezika, izvorni kôd mora biti preveden kroz nekoliko koraka od strane računala u kodirani izvršni kôd, odnosno strojni kôd *(engl. machine code*) kojeg razumije računalo i samo na taj način može izvesti program [2]. *Strojni* kôd, koji se naziva i *strojni jezik* je strogo numerički jezik koji se želi izvoditi što je brže moguće i smatra se programskim jezikom niske razine. Sastoji se samo od nula i jedinica, tj. binarno kodiranih naredbi koje procesor (*engl. central processing unit (CPU)*) može izravno prihvatiti i izvršiti u bilo kojem trenutku kada mu je to naložio drugi program ili korisnik. Svaka naredba uzrokuje da CPU izvršava vrlo specifičan zadatak, kao što je učitavanje, spremanje, skok ili aritmetička logička jedinica (ALU) na jednoj ili više jedinica podataka u registrima ili memoriji CPU-a. Svaka vrsta CPU-a ima svoj strojni jezik, iako su oni u osnovi prilično slični [3].

Prijevod izvornog kôda može se izvršiti na dva načina. Jedan je način da računalo naredbe izvornog kôda prevodi u strojni oblik u trenutku izvođenja programa. Naredba se prevede pa izvrši. Nakon toga se prevede sljedeća naredba i izvrši i tako redom. Program ove vrste naziva se interpreter**.** Drugi način rukovanja prijevodom je da se sve naredbe izvornog kôda prevode i analiziraju odjednom, a rezultat rada je izvršni kôd. Program koji se koristi u ovoj shemi naziva se kompajler (*engl. compiler*). Za razliku od interpretera, kod kompajlera su izvorni kôd i izvršni kôd potpuno odvojeni i pri izvođenju neovisni [4].

1.2. Korektnost računalnog programa

Centralno pitanje u razvoju računalnog programa je ispitivanje njegove korektnosti (ispravnosti). s obzirom na specifikaciju programa. Specifikacijom se određuju preduvjet i postuvjet programa kao tvrdnje o vrijednostima programskih ulaza i izlaza. Program je korektan ako je ispunjen postuvjet uvijek kada je ispunjen preduvjet. Ukoliko neki program nije ispravan, tj. ukoliko u nekim situacijama on daje pogrešan rezultat, tada taj program ne samo da je nekoristan već može biti i štetan. Problem neispravnih programa je tijekom posljednih desetljeća postao jedno od ključnih pitanja u mnogim poslovnim granama. Računalni programi

se temelje na algoritmima (postupcima rješavanja problema) kojima su problemi riješeni matematičkim modelima. Nakon matematičkog rješavanja problema, postupkom programiranja te postupcima tumačenja i prevođenja u stojni jezik kreira se računalni program. Korektnost programa iz perspektive softverskog inženjeringa može se definirati kao pridržavanje specifikacija programa koje određuju kako korisnici mogu komunicirati sa softverom i kako se softver treba ponašati kada se pravilno koristi.

1.3. Vrste specifikacija računalnog programa

Specifikacija programa je definicija namjene i načina rada računalnog programa, nacrt ili plan koji programer treba izvesti, i važan je dio u razvoju softvera. Specifikacija može biti formalna ili neformalna.

U računalnoj znanosti, formalne specifikacije su matematički utemeljene tehnike čija je svrha pomoć u implementaciji sustava i softvera. Koriste se za opisivanje sustava, analizu njegovog ponašanja i pomoć u dizajnu provjerom ključnih svojstava. Dio je općenitije zbirke tehnika koja je poznata kao "formalne metode". Međutim, pojam formalne metode često se koristi u smislu da bi se označio niz matematički utemeljenih tehnika za 'sustav' specifikacija, tj. za opis strukturnih i bihevioralnih svojstava softverskih sustava. Opis se najčešće izražava cjelovitim formalnim jezikom specifikacije.

Formalne specifikacije omogućuju dizajnerima da koriste apstrakcije, smanjujući tako konceptualnu složenost sustava koji se projektira. Omogućuju preciznu, dosljednu i cjelovitu definiciju svojstava sustava i olakšavaju razgradnju sustava na module [5]. Precizne matematičke definicije omogućuju analizu stroja i manipulaciju. Pod analizom podrazumijevamo da se može provjeriti dosljednost i ispravnost specifikacija, otkrivajući tako određene projektne pogreške i nedosljednosti koje se mogu pronaći u kasnijim fazama; neki se slučajevi pogrešaka mogu potpuno eliminirati. Pod manipulacijom podrazumijevamo da se jedna specifikacija može mehanički transformirati u drugu, detaljniju ili složeniju od svoje prethodnice i (na kraju) u izvršni program.

Budući da se transformacija izvodi prema skupu dobro definiranih pravila (inače je ne bi bilo moguće mehanizirati), može se dokazati da je točna. Ono što je također podložno mehaničkoj provjeri je dosljednost, cjelovitost i ispravnost specifikacija. Kako većina specifikacija ima neku vrstu matematičke osnove, primjenjiva pravila zaključivanja mogu se

koristiti za otkrivanje proturječnosti, nedostajućih definicija i drugih nedostataka. Neke od ovih pogrešaka mogu imati značajan utjecaj na implementaciju, no klasičnim testiranjem i otklanjanjem pogrešaka može ih biti vrlo teško pronaći.

Iako se ponekad smatra da ih je teško konstruirati, potvrditi i koristiti, metode formalnih specifikacija počinju nalaziti svoje odgovarajuće mjesto u procesu dizajniranja softvera. Veličina i složenost suvremenih softverskih sustava bolno su jasno stavili do znanja da se softver ne može uspješno dizajnirati, implementirati i provjeriti bez čvrste, sustavne metodologije. Nedostatak takvih metoda rezultira onim što je općenito poznato kao softverska kriza, a upotreba odgovarajućih formalnih metoda za specifikacije sustava može pomoći dizajnerima da prevladaju barem neke od ovih problema [6].

1.4. Aplikacija

Aplikacija kao primjenjivi računalni program, ili app (*engl. Application software*) je računalni program dizajniran za pomoć korisnicima da bi izvršavali jedan ili više određenih zadataka [7]. Primjeri uključuju poslovne programe, računske programe, uredske programe, grafičke programe, medija izvođače, odnosno svaki segment u kojemu će ljudima pomoći izvršti aktivnost. Primjenjivi program se ovdje razlikuje od operacijskog sustava (programa koji pokreće računalo), pomoćnih programa (koji izvršavaju održavanje ili glavno-svrhovne poslove) i programskog jezika (s kojim se rade računalni programi). Ovisno o aktivnosti za koju je dizajnirana, aplikacija može manipulirati brojkama, tekstom, grafikom ili kombinacijom tih elemenata. Neki aplikacijski paketi nude zamjetnu računalnu moć koncentrirajući se na samo jedan zadatak kao što je obrada teksta, a drugi koji se zovu integrirani programi, su nešto manje moćni, ali sadrže nekoliko primjenjivih programa. Programi pisani od korisnika tjeraju sisteme da zadovolje njihove vlastite potrebe. Takoošer program pisan od korisnika sadrži predloške proračunskih tablica, makro naredbe obrađivača teksta, znanstvene simulacije, grafičke i animacijske skripte. Čak su i e-mail filteri vrsta korisničkog programa. Prepoznavanje sistemskog programa kao što je operacijski sustav i aplikacijskog programa, odnosno primjenjivog programa nije precizno i ponekad je u suprotnosti sa samim sobom. Na primjer, jedno od glavnih pitanja je bilo, da li je Microsoftov internetski pretraživač dio Windows operacijskog sustava ili odvojivi dio (primjenjivi program). U nekim tipovima ugradbenog sustava, primjenjivi program i program operacijskog sustava mogu biti neraspoznatljivi korisniku, kao u slučaju programa mikrovalne pećnice, programa koji se koristi

za kontroliranje DVD playera i sličnih elektroničkih uređaja [7]. Korisnici koji stvaraju te programe sami i često previde njihovu vrijednost, zahtjevnost, korektnost, te je upravo zbog toga potrebno izvršiti testiranje programa, što će biti detaljnije objašnjeno u sljedećem poglavlju.

2. Testiranje

Ljudi su skoro pa stalno u interakciji sa nekim softverom. Rabe ga svakodnevno u životu, koristeći se raznovrsnim aplikacijama. Koriste ih u bilo kojem segmentu života, bilo to obrazovanje, poslovni svijet, slobodno vrijeme za vlastite potrebe, i dr. S obzirom na učestalost korištenja čovjek se redovito susreće s nekim problemima. Problemi nastanu kada opcije aplikacije, neki dijelovi aplikacije ili sama apliakcija ne radi onako kako bi zapravo trebala raditi. Na primjer u bankarstvu se može dogoditi greška prilikom plaćanja karticom, zatim kod e-mail pošte se može pojaviti greška kada se ne može poslati pošta, ili je e-mail poruka zapela pri slanju ili greška kada ne pristiže nova pošta, itd. Velika većina tih problema nastane kod problema sa softverom.

Svi ti softveri nemaju istu važnost i iste greške, pa tako ni jednak utjecaj na svakog čovjeka. Pogreške su različite, postoje greške koje su bez posljedica i trivijalne, dok neke imaju posljedica, a to može rezultirati gubitkom novaca te reputacije, ili još gore, mogu rezultirati ozljedom ili smrću. Greška bez posljedice je na primjer kada tijekom prikaza podataka iz baze neka web stranica pogrešno prikazuje naziv ili opis tvrtke, tada je jedina posljedica da ta tvrtka izgleda neprofesionalno, ali se ništa loše ne može dogoditi, nema posljedice. Za razliku od pogreške prikaza podataka u aplikaciji koja se koristi za medicinske svrhe. Na primjer aplikacija koja mjeri razinu šećera u krvi, zatim predlaže i pokazuje dozu inzulina koju pacijent treba uzeti. Takva aplikacija je jako rizična i jako je bitno da ne dođe do problema sa softverom odnosno da se ne pojavi pogreška jer posljedica za pacijenta može biti jako velika te se mogu dogoditi loše stvari. Upravo zbog takvih slučajeva ali i onih ostalih potrebno je izvršiti testiranje softvera. Testiranje se provodi kako bi aplikacija radila bez pogrešaka za što više korisnika, a kako je moguće da uvijek nešto može poći po krivu svaki proizvod se treba provjeravati i testirati tijekom svog razvoja. Tako bi se greške što prije uočile, pa bi se s tim smanjila mogućnost pogrešaka kod završene, gotove aplikacije. Testirati se može sve u aplikaciji, od korisničkog sučelja do tijeka cijele aplikacije [8].

Testiranje se provodi i kako bi se zadovoljili zadani standardi ili kriteriji nekog regulatornog tijela. Na primjer u Sjedinjenim Američkim Državama regulatorno tijelo US Food and Drug Administration (FDA) kontrolira aplikacije za medicinu te postavlja različite zahtjeve koje aplikacija mora zadovoljiti kako bi bila odobrena od njihove strane. Pa ako se želi na tržištu poboljšati šansa za uspjeh proizvoda bitno je dobiti odobrenje od te regulatorne agencije FDA-a. To odobrenje je jako bitno za korisnike jer su šanse za mogućnost grešaka jako male [9].

Sama definicija testiranja prema International Software Testing Qualifications Bord (ISTQB) glasi: „testiranje je proces koji se sastoji od svih aktivnosti životnog ciklusa kako statičnih tako i dinamičnih koje se odnose na planiranje, pripremu i evaluacije softverskih proizvoda i srodnih proizvoda kako bi se utvrdilo zadovoljavaju li određene zahtjeve, da li su prikladni za svoju svrhu te da se otkriju nedostaci“ [10].

Prema tome možemo reći da je testiranje proces otkrivanja problema sa softverom i zadovoljavanje određenih zahtjeva. Provjerava se postoje li greške ili nedostatci u radu sa softverom, da li je softver spreman za isporuku te da li su svi korisnički kriteriji zadovoljeni. Također se testiranjem povećava osiguranje kvalitete proizvoda, jer pronalaskom pogrešaka prije nego što proizvod bude isporučen korisniku osigurava njegovo zadovoljstvo zbog toga što na kraju korisnik dobije softver bez pogrešaka.

Osiguranje kvalitete proizvoda jedan je od osnovnih pojmova u testiranju. S obzirom da je kvaliteta subjektivna jako je teško jednoznačno definirati osiguranje kvalitete proizvoda. Prema međunarodnom ISO/IEC standardu kvaliteta softvera ovisi o [11]:

- Funkcionalnosti proizvoda – osnovna svrha proizvoda, zadovoljava specifikacije i standarde. Odgovara na pitanje jesu li zahtijevane funkcije raspoložive.
- Pouzdanosti proizvoda – sposobnost sustava da kroz neki vremenski period održava pružanje usluga u zadanim uvjetima. Odgovara na pitanje koliko je programski proizvod pouzdan.
- Upotrebljivosti proizvoda – jednostavnost upotrebe određene funkcije sustava, privlačan korisnicima. Odgovara na pitanje je li softver jednostavno koristiti.
- Učinkovitosti proizvoda – označava sve resurse sustava koji se koriste kod funkcionalnosti (npr. prostor na disku, memorija, mreža, itd.). Odgovara na pitanje koiko je softver pouzdan.

- Održivosti proizvoda – odnosi se na sposobnost prepoznavanja i uklanjanja greški unutar softvera, održavanje, testiranje i analiziranje. Odgovara na pitanje koliko je jednostavno obavljati izmjene u softveru.
- Prenosivosti proizvoda – označava koliko je sustav prilagodljiv, softver mora moći imati promjene u zahtjevima i upotrebljavati u različitim okruženjima. Odgovara na pitanje koliko je jednostavno prebaciti softver u drugu okolinu.

2.1. Povijest testiranja

Sve do 19. stoljeća, prije doba modernog kapitalizma i industrijske revolucije, te prije nego se povećalo tržište kupci su bili prisiljeni kupovati proizvode bez osiguranja kvalitete. Takva kvaliteta na tržištu nije odgovarala traženoj cijeni, ali zbog malog tržišta i ponude kupci su ipak kupovali proizvode. U 19. stoljeću kada se tržište povećalo, dolazi do osiguranja kvalitete proizvoda. Prodavačev cilj je bio osigurati proizvode bez pogrešaka kako bi zadržali stare te privukli nove kupce.

Razvoj prvih računala počinje sredinom dvadesetog stoljeća, a s razvojem računala dolazi i do prvih programskih pogrešaka. Računalo Mark II jedno je od prvih računala, izrađen na sveučilištu Harvard, računalo na kojem su tehničari tijekom rada primjetili da je računalo prestalo raditi. Kada su pokušavali pronaći razlog prestanka rada računala, unutar računala pronašli su kukca, moljca koji je bio uzrok da računalo nije moglo normalno raditi. Tako je nastao naziv za pogreške (*engl. Bug*).

Nakon toga ubrzo je prepoznata potreba za testiranjem, pa tako dolazi do javljanja pojma osiguranja kvalitete proizvoda te se počinju formirati testni timovi. Prva knjiga isključivo za testiranje nastala je 1979. godine. Testiranje softvera s vremenom dobiva sve veće značenje, razvijaju se novi načini testiranja, razni modeli testiranja, zatim automatiziranje procesa testiranja. U samom početku testiranje programa obavljao je sam programer, dok u današnje vrijeme programer surađuje s testerom koji je slobodan pronaći najbolje rješenje za osiguranje kvalitete proizvoda [12].

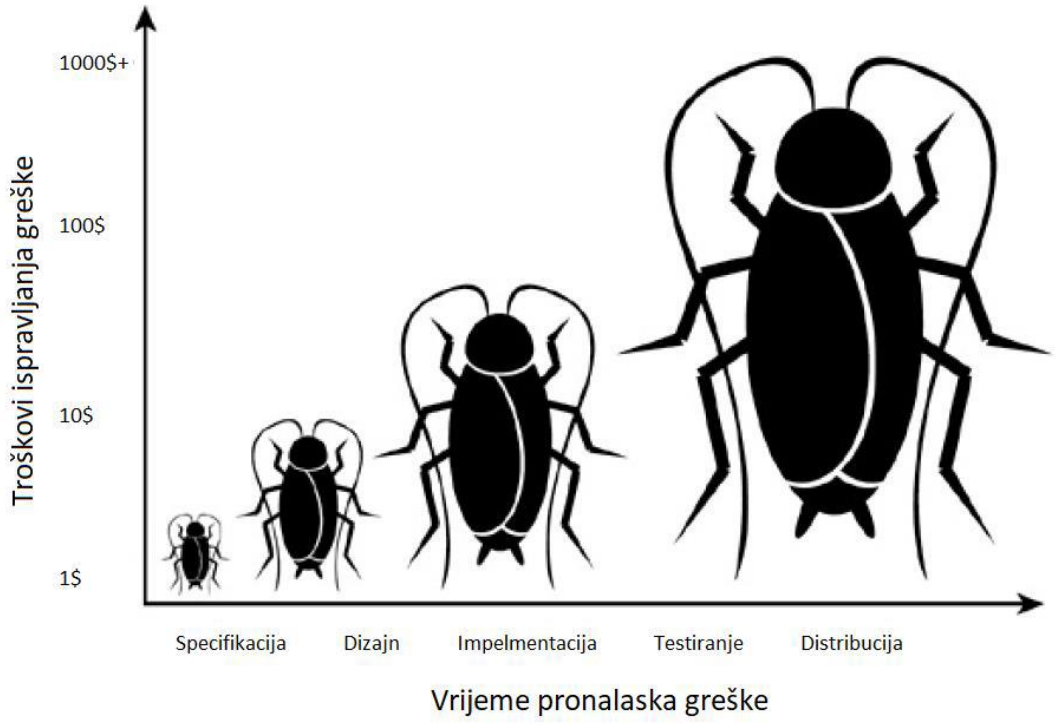
Osiguranje kvalitete proizvoda, odnosno testiranje programa danas ima bitnu ulogu, veliku važnost u izgradnji programskog proizvoda, pa je pretpostavka da će se u bližoj budućnosti održavati, ili povećavati ta uloga u razvoju programskog proizvoda. S obzirom na svakodnevne promjene koje se događaju, posao testera se značajno mijenja, a kako bi područje

testiranja zadržalo svoju poziciju ono mora biti spremno prilagoditi se tim promjenama. Na primjer pojavom novih tehnologija kao što su umjetna inteligencija ili roboti.

2.2. Svrha testiranja

Svrha testiranja može se najbolje pokazati nekim primjerom, jer postoje različita mišljenja u vezi testiranja. Netko smatra da zbog činjenica kako nikada nije moguće u potpunosti testirati softver, te činjenica da često sustav s pogreškama završi kod krajnjeg korisnika, mišljenje je da je testiranje softvera gubitak novaca i vremena za neko poduzeće. Smatra se da su programeri sami dovoljni testeri programskog proizvoda.

Da bi pokazali svrhu uzet ćemo na primjer nogometnog sudca. Iako sudac u nogometu veći dio vremena ne sudjeluje u utakmici ne smijemo ga u potpunosti ukloniti jer bi to prouzročilo puno većim brojem prekršaja i nepravilnosti ponašanja igrača. Takva ista analogija može se primjeniti i na proces životnog ciklusa softvera. Kada bi se iz procesa uklonilo testiranje, krajnji proizvod bi sadržavao puno više nepravilnosti i pogrešaka što bi naravno rezultiralo velikim nezadovoljstvom korisnika, pa je upravo zbog toga testiranje ima veliku svrhu u procesu razvoja softvera.



Slika 2. *Odnos vremena pronalaska greške i njezinih troškova ispravljanja*

Na slici 2. je prikazan značaj testiranja, te se vidi kako na troškove ispravljanja greške bitno utječe vrijeme njezina pronalaska. Ako je greška uočena u fazi specifikacije ili postoji greška u dizajnu tada su troškovi ispravljanja mali i te greške je najjednostavnije ispraviti. Ako je greška uočena u fazi implementacije trošak ispravljanja je malo veći nego u prethodne dvije faze ali je idalje prihvatljiv, a ispravak greške malo zahtjevniji s obzirom na prethone dvije faze. Ako je vrijeme pronalaska greške u fazi distribucije, posljednjoj fazi, ispravak je puno zahtjevniji, a trošak ispravljanja jako skup, može biti i do nekoliko milijuna.

Prema istraživanjima najviše pogrešaka nastane fazi implementacije, čak 50%, zatim 40% pogrešaka se dogodi u fazi dizajna, dok samo 10% u fazi specifikacije. S obzirom da se najviše pogrešaka dogodi u fazi implementacije vrlo važno je testirati sustav tijekom i nakon te faze. Na taj način se smanjuje mogućnost isporuke sustava sa greškama, pa samim tim visoki troškovi ispravljanja. Isto tako potrebna je provjera faza prije faze implemetacije kako bi što ranije uočili pogrešku i uklonili [12]. Kraće rečeno, svrha testiranja je osiguravanje isporuke korektnog sustava i smanjivanje troškova za ispravljanje pogrešaka.

2.3. Cilj testiranja

Možemo reći da je osnovni cilj testiranja pronaći greške u sustavu. Iako može postojati slučaj kada je u cilju testiranja samo potvrđivanje da izrađene funkcionalnosti rade, ali tako se onda propuste greške u sustavu. Ako takav sustav stigne do korisnika posljedice su vremenski i financijski značajnije. Prema tome što se prije pronađe greška to ju je jednostavnije i jeftinije popraviti. Sada možemo preoblikovati da je osnovni cilj testiranja što prije moguće pronaći greške u susatvu. Iz perspektive korisnika ni to neće biti dovoljno da bi programski proizvod bio isporučen bez grešaka. Jer iako grečke budu otkrivene u procesu testiranja može se dogoditi da neće biti ukolnjene do same isporuke korisniku, pa korisnik dobije proizvod s greškom ili greškama. S obzirom i na tu situaciju ponovno možemo preoblikovati da je osnovni cilj testiranja zapravo što prije moguće pronaći greške u sustavu i pobrinuti se da budu uklonjene. Taj cilj upotpunjuje zadatak testiranja i garantira korisniku da do njega stigne programski proizvod koji radi bez grešaka [13].

A kada pokušamo prikazati cilj testiranja programskog proizvoda na stručan način pojavljuju se dva nova pojma. To su verifikacija i validacija. Verifikacija kao koncept programskog proizvoda služi za procjenjivanje softvera na kraju razvojne faze. Kada su sve

faze završene provjerava se jesu li svi zahtjevi postavljeni na početku razvoja zadovoljeni, tj. ponaša li se program prema specifikacijama i regulacijama sustava. Za razliku od verifikacije, validacijom se vrši provjera zadovoljava li softver očekivanja kupca, provjera valjanosti programskog proizvoda, pa možemo reći da se odnosi na konačni programski proizvod koji testira kupac iz svoje perspektive. Validaciju je potrebno provjeravati u što ranijoj fazi razvoja softvera tako da ne može doći do velikih troškova, što bi bio slučaj kada bi se validacija provodila na završetku razvojnog procesa [14].

2.4. Greške

Greške (*engl. bugs*) su problemi koji se pojavljuju u razvoju programskog proizvoda, aplikacije. Problemi se događaju kada programski proizvod nije u skladu sa zadanim specifikacijama ili te specifikacije ne izvršavaju dobro određene funkcije. Svaki programski proizvod ima zadane specifikacije koje treba zadovoljiti, a one definiraju što će raditi, što neće raditi, te kako se treba ponašati. Greška nastaje u slučaju kad:

- se radi ono što specifikacije kažu da nebi trebalo raditi
- se ne radi ono što specifikacije kažu da treba raditi
- se radi nešto što nije navedeno u specifikacijama
- se ne radi nešto što je navedeno u specifikacijama
- je programski proizvod spor, nepregledan ili ga je teško koristiti

Greške se mogu podijeliti u dvije skupine, prema ozbiljnosti i prema prioritetu. Prema ozbiljnosti razlikujemo značajne greške, kritične greške, niže ili kozmetičke greške. Značajne greške su greške koje označavaju nerad neke funkcionalnosti, ali ta greška se može zaobići pa nije veliki prioritet za uklanjanje. Zatim kritične greške, greške koje onemogućuju rad programskog proizvoda u potpunosti pa veliki prioritet da se što prije uklone. Dok niže greške su one greške koje ne utječu na rad programskog proizvoda, a kozmetičke greške su kada dizajn odudara od zadanog dizajna [15].

2.5. Principi testiranja

Tijekom proteklih pola stoljeća testiranja nastali su principi testiranja, te se smatraju generalnim smjernicama prilikom svih vrsta testiranja [16]. Principi:

Nemoguće je testirati cijeli programski proizvod, aplikaciju – nije moguće testirati sve unose koji postoje, sve rezultate, te sva prethodna stanja. Onda umjesto sveobuhvatnog testiranja, provodi se testiranje svih bitnih dijelova aplikacije, te akcija koje se šešće pojavljuju i utječu na doživljaj korisnika.

Testiranje dokazuje prisutnost pogrešaka – dokazuje se prisutnost pogrešaka. Smanjuje se vjerojatnost da postoje skrivene grešake. Ako greške nisu pronađene, to ne mora značiti da ih nema uj programu

Rano testiranje – kako bi se što prije pronašle zatim i uklonile pogreške, potrebno je što prije započeti s testiranjem razvojnog procesa

Imunizacija programskog proizvoda, aplikacije – u slučaju stalnih izvršavanja istih testova, ti setovi neće biti u mogućnosti pronaći neke nove greške. Kako bi ss to izbjeglo testove je potrebno evaluirati i sukladno tome promjeniti, usmjeriti ih na druge dijelove programskog proizvoda, apliacije.

Testiranje ovisi o kotekstu – pristupa se ovisno o kontekstu, različite vrste programskih proizvoda, aplikacije se testiraju na drugačije načine.

Iluzija o nepostojanju grešaka – korisnici bez obzira na pronalaženje grešaka i njihovo uklanjanje ne znači da će prihvatiti taj programski proizvod, aplikaciju. U slučaju da je spora, te ne zadovoljava korisnikove zahtjeve, takva neće biti uspješna, jer sam broj grešaka nije najbitnija karakteristika programskog proizvoda, aplikacije.

Sve greške neće biti ispravljene – razlog tome je nedostatak vremena ili neispativosti ispravka, odnosno uklanjanja pogreške.

Što se više grešaka pronađe, to više grešaka postoji – u slučaju da se tokom testiranja proizvoda pronađu greške, velika je mogućnost da postoji još takvih grešaka ili postoje greške oko pronađene greške. Uočena greška lako može biti nagovještaj većih grešaka u pozadini.

2.6. Statičko i dinamičko testiranje

Statičko testiranje je vrsta softverskog testiranja u kojem se softverska aplikacija testira bez izvršenja kôda, odnosno programa. Ručni ili automatizirani pregledi programskog proizvoda sa zahtjevima i dizajnom rade se kako bi se pronašle greške. Osnovni cilj statičkog testiranja je poboljšati kvalitetu softverskih aplikacija pronalaženjem grešaka u ranim fazama procesa razvoja programskog proizvoda. Statičko ispitivanje uključuje ručni ili automatizirani pregled dokumenata. Naziva se i ispitivanjem neizvršenja ili provjerom provjere. Primjeri radnih dokumenata:

- Specifikacije zahtjeva
- Projektni dokument
- Izvorni kôd
- Ispitni planovi
- Ispitni slučajevi
- Test skripte
- Pomoć ili Korisnički dokument
- Sadržaj web stranice

Pod dinamičkim testiranjem izvršava se programski kôd. Provjerava funkcionalno ponašanje softverskog sustava, upotrebu memorije ili procesora i ukupne performanse sustava. Otuda i naziv „dinamičko“ (*engl. "Dynamic"*). Osnovni cilj ovog testiranja je potvrditi da softverski proizvod radi u skladu s poslovnim zahtjevima. Ovo se ispitivanje naziva i tehnikom izvršenja ili provjerom valjanosti. Dinamičko testiranje izvršava softver i potvrđuje izlaz s očekivanim ishodom. Dinamičko testiranje provodi se na svim razinama ispitivanja, a može biti testiranje u crnoj ili bijeloj kutiji. Statičko i dinamičko testiranje se razlikuje u mnogo čemu, pa u tablici 1. možemo vidjeti te razlike [17].

Slika 3. *Statičko i dinamičko testiranje*

Statičko ispitivanje	**Dinamičko ispitivanje**
Testiranje je obavljeno bez izvršavanja programa	Testiranje se vrši izvršavanjem programa
Statičko testiranje vrši postupak provjere	Dinamičko testiranje vrši postupak provjere valjanosti
Statičko testiranje odnosi se na sprečavanje grešaka	Dinamičko testiranje odnosi se na pronalaženje i otklanjanje grešaka
Statičko testiranje daje ocjenu kôda i dokumentacije	Dinamičko testiranje stvara programske greške
Statičko testiranje uključuje kontrolni popis i postupak kojeg treba slijediti	Dinamičko testiranje uključuje test slučajeva za izvršenje
Statičko testiranje obuhvaća testiranje pokrivenosti i testiranje strukturne pokrivenosti	Dinamičko testiranje obuhvaća analizu granica i podjelu ekvivalentnosti.
Troškovi pronalaska grešaka, nedostataka i popravljanja su manji	Troškovi pronalaska grešaka, nedostataka i popravljanja su visoki
Povrat ulaganja bit će visok jer je testiranje počelo u ranoj fazi	Povrat ulaganja bit će nizak jer je testiranje počelo nakon faze razvoja
Preporučuje se više komentara za dobru kvalitetu	Za dobru kvalitetu preporučuje se više nedostataka.
Zahtijeva gomilu sastanaka	Zahtijeva manje sastanaka

Tablica 1. *Razlike između statičkog i dinamičkog testiranja*

2.6.1. Testiranje po principu crne kutije

Testiranje crne kutije (*engl. Black box testing*) je dinamičko testiranje. To je tehnika softverskog testiranja koja ispituje funkcionalnost softvera bez zavirivanja u njegovu unutarnju strukturu ili kodiranje programskog proizvoda. Primarni izvor ispitivanja crnih kutija je specifikacija zahtjeva koju navodi kupac, odnosno korisnik. U ovoj metodi tester odabire funkciju i daje ulaznu vrijednost kako bi ispitao njezinu funkcionalnost te provjerava daje li ta funkcija očekivani izlaz ili ne. Ako funkcija daje točan izlaz, tada je proslijeđena u testiranje, inače nije uspjela. Nakon toga testni tim izvještava rezultat razvojnom timu, a zatim testira sljedeću funkciju. Nakon završetka testiranja svih funkcija ako postoje ozbiljni problemi, onda se to vraća razvojnom timu na ispravljanje [18].

Postupak testiranja crne kutije je postupak u kojem tester ima specifična znanja o radu softvera i razvija testne slučajeve kako bi provjerio točnost funkcionalnosti softvera. Ne zahtijeva poznavanje softvera za programiranje. Svi testni slučajevi su dizajnirani uzimajući u obzir ulaz i izlaz određene funkcije. Tester zna o određenom izlazu određenog ulaza, ali ne i o tome kako rezultat proizlazi. Postoje razne tehnike korištene u testiranju crne kutije za testiranje poput tehnike tablice odluka, tehnike analize granične vrijednosti, prijelazna stanja, tehnike grafikona uzroka, tehnike raspodjele ekvivalentnosti, tehnike pogađanja pogrešaka, tehnike slučaja [19].

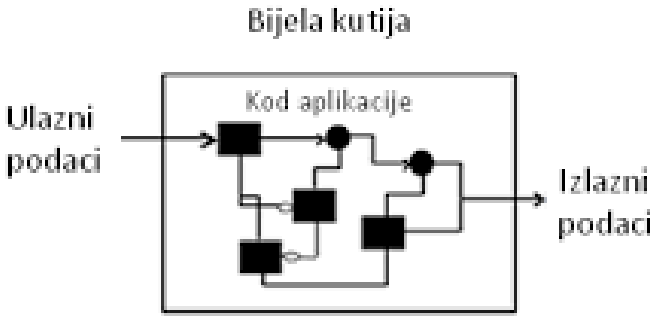
Slika 4. *Testiranje po principu crne kutije*

Opći koraci ispitivanja crne kutije [20]:

- Testiranje crne kutije temelji se na specifikaciji zahtjeva, pa se testira na početku razvoja programskog proizvoda.
- U drugom koraku tester stvara pozitivan i negativan scenarij testiranja odabirom ispravnih i neispravnih ulaznih vrijednosti kako bi provjerio obrađuje li ih softver točno ili netočno.
- U trećem koraku tester razvija razne slučajeve testiranja kao što su tablica odluka, ekvivalentna podjela, procjena grešaka, grafikon uzroka, itd.
- Četvrti korak uključuje izvršenje svih testnih slučajeva.
- U petom koraku tester uspoređuje očekivani izlaz sa stvarnim učinkom.
- U šestom, posljednjem koraku, ako postoji bilo kakva greška u softveru, ona se ukloni i ponovo se testira.

2.6.2. Testiranje po principu bijele kutije

Testiranje bijele kutije (*engl. white box testing*) koja je također poznata i kao testiranje staklene kutije, testiranje konstrukcija, testiranje prozirnih kutija, testiranje otvorenih kutija i testiranje prozirnih kutija. Ovaj postupak testiranja testira interno kodiranje i infrastrukturu softverskog fokusa na provjeru unaprijed definiranih ulaza u odnosu na očekivane i željene izlaze. Temelji se na unutarnjem radu programskog proizvoda, aplikacije, i vrti se oko unutarnjeg ispitivanja strukture. U ovoj vrsti testiranja potrebne su vještine programiranja za dizajniranje testnih slučajeva. Osnovni cilj testiranja bijele kutije je usredotočiti se na protok ulaza i izlaza kroz softver, te povećavanje sigurnosti softvera. Izraz "bijela kutija" koristi se zbog interne perspektive sustava, a označava sposobnost prolaska vanjske ovojnice softvera u njegov unutarnji rad.



Slika 5. *Testiranje po principu bijele kutije*

Programeri rade testiranje bijele kutije [21]. U ovom testiranju programeri će testirati svaki redak koda programa. Provode testiranje, a zatim šalju aplikaciju testnom timu, gdje će se izvršiti testiranje crne kutije i provjeriti programski proizvod, aplikaciju, zajedno sa zahtjevima te identificirati greške i poslati ih programeru. Programer uklanja greške i vrši ponovo jedan krug testiranja bijele kutije pa ga opet šalje testnom timu.

Testeri se neće uključiti u otklanjanje grešaka i nedostataka iz dva glavna razloga. Prvi razlog je zbog toga što bi njihovo ispravljanje greške moglo prekinuti neke druge značajke. Pa bi stoga tester uvijek trebao samo pronaći greške, a programeri ih ukloniti. Drugi razlog je da ako testeri provode većinu vremena otklanjajući greške i nedostatke, tada je vrlo vjerojatno da neće moći pronaći druge greške u programskom proizvodu, aplikaciji. Testiranje bijele kutije sadrži različite testove, a to su testiranje puta, testiranje petlje, testiranje stanja, zatim testiranje na temelju memorijske perspektive i testiranje izvedbe programskog proizvoda [22].

Opći koraci ispitivanja bijele kutije:

- Dizajniranje svih scenarija testiranja, testnih slučajeva, zatim davanje prioriteta prema broju visokog prioriteta.
- Ovaj korak uključuje proučavanje koda za vrijeme izvođenja kako bi se ispitalo korištenje resursa, nepristupačna područja koda, vrijeme potrebno raznim metodama i operacijama, itd.
- U trećem se koraku odvija testiranje unutarnjih potprograma. Internim potprogramima, poput ne javnih metoda, sučelja su u mogućnosti obrađivati sve vrste podataka na odgovarajući način.
- Četvrti korak se fokusira na testiranje kontrolnih naredbi poput petlji i uvjetnih naredbi kako bi se provjerila učinkovitost i točnost različitih unosa podataka.

- U zadnjem koraku testiranje bijele kutije uključuje sigurnosno testiranje kako bi se provjerile sve moguće sigurnosne rupe gledajući kako kod postupa sa sigurnošću.

Razlike između bijele i crne kutije i njihovog načina testiranja prikazuje sljedeća tablica, Tablica 2.

Testiranje po principu crne kutije	**Testiranje po principu bijele kutije**
metoda testiranja bez znanja o stvarnom kodu ili unutarnjoj strukturi aplikacije	metoda testiranja koja ima znanje o stvarnom kodu i unutarnjoj strukturi aplikacije
testiranje više razine, poput funkcionalnog tesiranja	testiranje se izvodi na nižoj razini testiranja, kao što su jedinično testiranje, integracijsko testiranje
koncentrirano je na funkcionalnost testiranog sustava	koncentrirano je na stvarni kodni program i njegovu sintaksu
za testiranje crne kutije potrebna je specifikacija zahtjeva	za testiranje bijele kutije potrebni su projektni dokumenti s dijagramima protoka podataka, dijagramima itd.
testiranje crne kutije rade testeri	Testiranje bijele kutije rade programeri ili testeri sa znanjem programiranja.

Tablica 2. *Razlike između načina testiranja bijele i crne kutije*

3. Vrste i metode testiranja

3.1. Vrste testiranja

Postoje različita shvaćanja u vezi vrsta testiranja. U literaturi vrste testiranja ovise o pogledu na proces testiranja programskog proizvoda, aplikacije, te o raznim faktorima, kao što su rok isporuke, budžet, raspoloživi ljudski resursi, itd.

Testiranje ispravne funkcionalnosti jedne komponente i testiranje same te komponente ne mora značiti da će kod svih testova ta komponenta biti bez grešaka. Pa je upravo zbog toga potrebno definirati vrstu testa. Vrste testova postoje kako bi se na određenoj razini mogli definirati ciljevi testiranja koji se žele ostvariti. Već prije je definirano da je osnovni cilj pojedinog testa testiranje funkcionalnosti komponenata, ali isto tako ponekad može biti u cilju

testirati i neku nefunkcionalnu komponentu. Pa prema tome testove općenito dijelimo na funkcijske i nefunkcijske, odnosno funkcionalno testiranje i nefunkcionalno testiranje [23].

3.1.1. Funkcionalno testiranje

Funkcionalno testiranje se odnosi na testove koji provjeravaju radi li programski proizvod ono što je zadano u specifikacijama, drugim riječima provjerava što aplikacija radi, te potvrđuju određenu radnju ili funkciju koda. Funkcionalni testovi odgovaraju na pitanja kao što su „Može li korisnik učiniti to?“ , te „funkcionira li točno određeno svojstvo?“. Testiranje se izvodi tako da se napravi neki unos u aplikaciji ili se dogodi neka akcija aplikacije, te se zatim provjerava je li dobiveni rezultat jednak očekivanom. Funkcionalno testiranje se može izvoditi po principu testiranja crne ili bijele kutije što je već objašnjeno u prethodnom poglavlju, zatim po principu testiranja jedinica, integracijkom testiranju, testiranju sustava, te po principu korisničkog testiranja prihvatljivosti. Ove metode testiranja biti će detaljnjije objašnjene u nastavku ovog poglavlja.

Funkcionalno testiranje se izvodi s krajnjim korisnicima imajući na umu kako bi točno korisnik upotrebljavao aplikaciju. Što znači da se testovi pišu gledajući na ponašanje korisnika i njegovo očekivanje od korištenja aplikacije. Funkcionalni testovi ne uključuju testiranje preformansi. Za to postoji druga vrsta testiranja, a to je nefunkcionalno testiranje [24].

3.1.2. Nefunkcionalno testiranje

Nefunkcionalno testiranje se ne odnosi na testiranje korektnosti aplikacije, već se odnosi na druge bitne aspekte aplikacije koji ne trebaju biti vezani za određene funkcije i korisnike. Neki od bitnijih aspekata su sigurnost aplikacije, upotrebljivost, prenosivost, integritet, itd. Nefunkcionalno testiranje bi trebalo povećati iskoristivost, učinkovitost, održivost i prenosivost proizvoda, aplikacije. Pomaže u smanjenju proizvodnog rizika i troškova povezanih s nefunkcionalnim aspektima proizvoda, optimizira način na koji se proizvod instalira, postavlja, izvršava, upravlja i nadgleda. Nefunkcionalno testiranje poboljšava znanje o ponašanju proizvoda, apikacije i tehnologija u upotrebi.

Testiranjem sigurnosti aplikacije se provjerava zaštićenost korisničkih podataka. U to se ubrajaju provjere autorizacije i provjere autentifikacije. Autentifikacija je proces u kojemu

se od korisnika traži neki unos, najčešće korisničko ime i lozinka kako bi sam korisnik mogao pristupiti podacima [28]. Tada je bitno osigurati da nitko osim korisnika s pravom kombinacijom unosa ne može pristupiti određenim podacima. Neki od testova provjere autentičnosti uključuju test pravila kvalitete lozinke, test promjene lozinke, test zadanih prijava, test oporavka lozinke, test funkcionalnosti odjave, test sigurnosnog pitanja / odgovora itd. Autorizacija govori o pravima pristupa koje može imati bilo koji korisnik, te o njegovim privilegijama, pa je najbitnije osigurati da korisnik nema ovlasti nad podacima za koje nije autoriziran i da nema ovlasti nad samom aplikacijom. Neki od testova autorizacije uključuju test za obilaženje puta, test za nedostajuću autorizaciju, test za probleme s horizontalnom kontrolom pristupa itd. Dakle u osnovi se mora testirati "tko ste" i "što možete učiniti".

Testiranje prenosivosti je vrsta testiranja softvera koja se radi kako bi se utvrdio stupanj lakoće ili poteškoće po kojem se softver, aplikacija može učinkovito prenijeti s jednog okruženja na drugi. Stupanj se mjeri u usporedbi troškova prilagodbe softvera u novom okruženju s troškovima ponovnog razvoja softvera u novom okruženju. Provodi se tako da se promijeni okruženje aplikacije, te se onda nadgledaju njene preformanse, pa je zbog toga bitno imati na umu prenosivost tijekom razvoja životnog ciklusa. Pod promjenom okruženja uglavnom se smatra promjena uređaja, promjena internet preglednika ili promjena operativnog sustava. Bitno je da aplikacija radi u različitim okruženjima jer su svi korisnici različiti i pretežito koriste aplikaciju u drugačijim okruženjima od onih koje koriste programeri [25].

Testiranje upotrebljivosti je metoda koja se koristi za procjenu jednostavnosti korištenja aplikacije. Testovi se odvijaju sa stvarnim korisnicima kako bi se izmjerilo koliko je aplikacija "upotrebljiva" ili "intuitivna", te koliko je lako korisnicima postići svoje ciljeve. Testovi se izvode po principu testiranja crne kutije. Od korisnika se traži da izvrši zadatak, obično dok ih testeri promatraju, kako bi vidjeli gdje nailaze na probleme i gdje doživljavaju zbrku. U slučaju da se više korisnika susreće sa sličnim ili istim problemima, onda će se prema tome prije izdavanja proizvoda na tržište dati preporuke za prevladavanje tih problema upotrebljivosti, kako bi korisnici bili što više zadovoljni s gotovim proizvodom[26].

Testiranje prenosivosti je vrsta testiranja softvera koja se radi kako bi se utvrdio stupanj lakoće ili poteškoće po kojem se softver, aplikacija može učinkovito prenijeti s jednog okruženja na drugi. Stupanj se mjeri u usporedbi troškova prilagodbe softvera u novom okruženju s troškovima ponovnog razvoja softvera u novom okruženju. Provodi se tako da se promijeni okruženje aplikacije, te se onda nadgledaju njene preformanse, pa je zbog toga bitno

imati na umu prenosivost tijekom razvoja životnog ciklusa. Pod promjenom okruženja uglavnom se smatra promjena uređaja, promjena internet preglednika ili promjena operativnog sustava. Bitno je da aplikacija radi u različitim okruženjima jer su svi korisnici različiti i pretežito koriste aplikaciju u drugačijim okruženjima od onih koje koriste programeri.

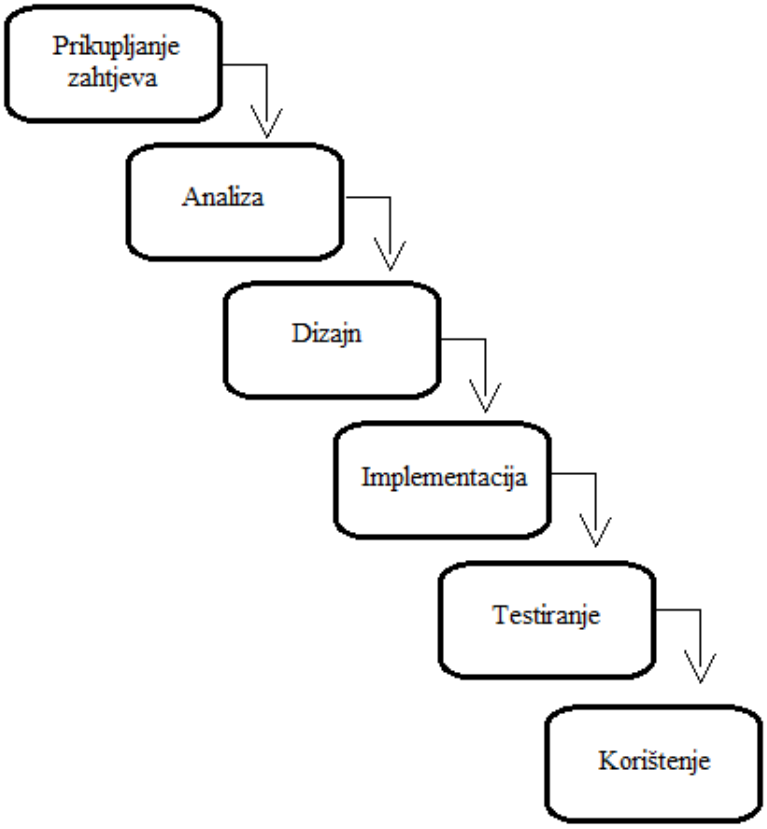
Za korisnika jedno od bitnijih nefunkcionalnih testiranja je testiranje integriteta korisničkih podataka. Pod integritetom podataka se smatra da su svi podaci cjeloviti i potpuni. Pa će se testiranjem integriteta provjeriti je li prijenos ili dohvaćanje, i spremanje podataka utjecalo na same podatke, te će se također provjeriti postoje li pogreške u softveru koje mogu dovesti do neovlaštenog pristupa podacima, što ide na štetu korisnika [27]. Postoje jos razna nefunkcionalna testiranja aplikacije kao što su testiranje lokalizacije gdje se testovima provjerava ispravnost lokacije, zatim testiranje internacionalizacije čiji testovi provjeravaju ispravnost jezika, testiranje izdržljivosti koje testira kako sustav reagira na velika opterećenja, testiranje kompatibilnosti, testiranje naprezanja, tetiranje glasnoće, testiranje oporavka od katastrofe, testiranje sukladnosti, testiranje održavanja, testiranje dokumentacije itd. Sve vrste testiranja se ne primjenjuju na sve projekte, vrsta testiranja ovisi o prirodi i opsegu projekta.

3.2. Metode testiranja

U procesu razvoja životnog ciklusa softvera postoji nekoliko metodologija, neke od njih su agilni razvoj, vodopadni razvoj, V-model, itd. Nastajanjem metodologija za razvoj pojavile su se metodologije za testiranje softvera. Kako bi korisnik bio zadovoljan, softver prije isporuke treba biti bez grešaka, pa zbog toga testiranje mora stalno biti uključeno u proces razvoja.

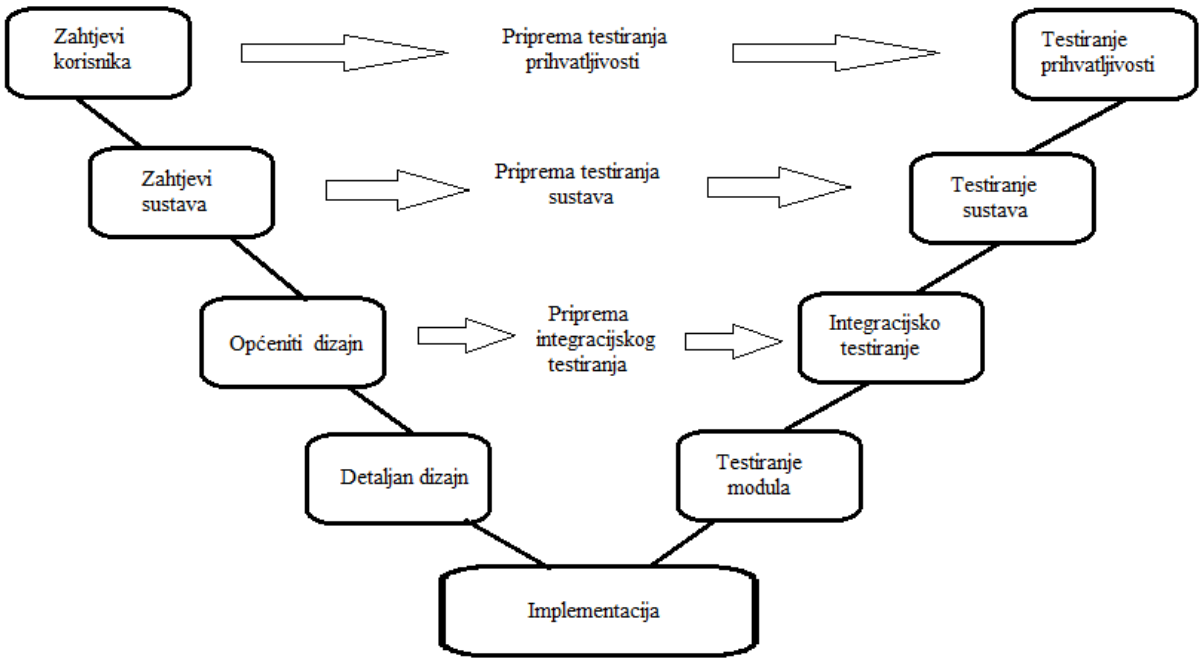
Metodologije testiranja softvera su razne strategije ili pristupi koji se koriste za testiranje aplikacije kako bi se osiguralo da se aplikacija ponaša i izgleda onako kako se očekuje. Izbor modela ovisi o oglednoj aplikaciji i njezinim ciljevima, jer se neke od metoda zasnivaju na zaradi i brzini, dok se druge zasivaju na kvaliteti i dokumentiranju. Prema tome svaka metoda osigurava drugačije mjesto testiranja.

Vodopadni model je bio prvi nastali model, jako je jednostavan, te najpoznatiji način provođenja životnog ciklusa. Ovaj model se sastoji od šest povezanih faza. Svaka iduća faza može započeti jedino kada je prethodna faza završena, s tim da je moguće vratiti se na prethodnu fazu. Sljedeća Slika 6. prikazuje vodopadni razvojni model i njegove faze [28].



Slika 6. *Vodopadni model i njegove faze*

Ovakav razvoj aplikacije je linearan sekvencijalan, te ne može doći do preklapanja faza. U prvoj početnoj fazi razvoja prikupljaju se zahtjevi. Kada su svi zahtjevi prikupljeni provodi se analiza tih zahtjeva kojom se skupljaju sve potrebne informacije o proizvodu, željama klijenta, te funkcionalnim mogućnostima. Nakon detaljne analize u trećoj fazi se osmišljava dizajn aplikacije, proizvoda. Zatim slijedi implementacija, razdoblje četvrte faze gdje se vrši izrada cijele aplikacije. Nakon izrade, u petoj, predzadnjoj fazi provodi se testiranje aplikacije. Nakon testiranja, dolazi zadnja faza u razvoju u kojoj se isporučuje te pušta na korištenje gotov proizvod. Ovakav način razvoja ima veliku manu, odnosno problem, a to je ne pravovremeno otkrivanje grešaka. Razlog tomu je to što se testiranje izvodi pri samom kraju razvojnog procesa, pa s time dolazi kasno otkrivanje pogrešaka. Kako bi ovaj problem bio riješen razvijen je poboljšani vodopadni model koji se naziva i V-model. Poboljšani model je usmjeren na to da testiranje počne u što ranijoj fazi razvojnog procesa, tj. osigurava da se testiranje odvija nakon svake faze, što se može vidjeti na sljedećoj slici, slika 7. V-model [29].



Slika 7.. *V-model razvoja životnog ciklusa*

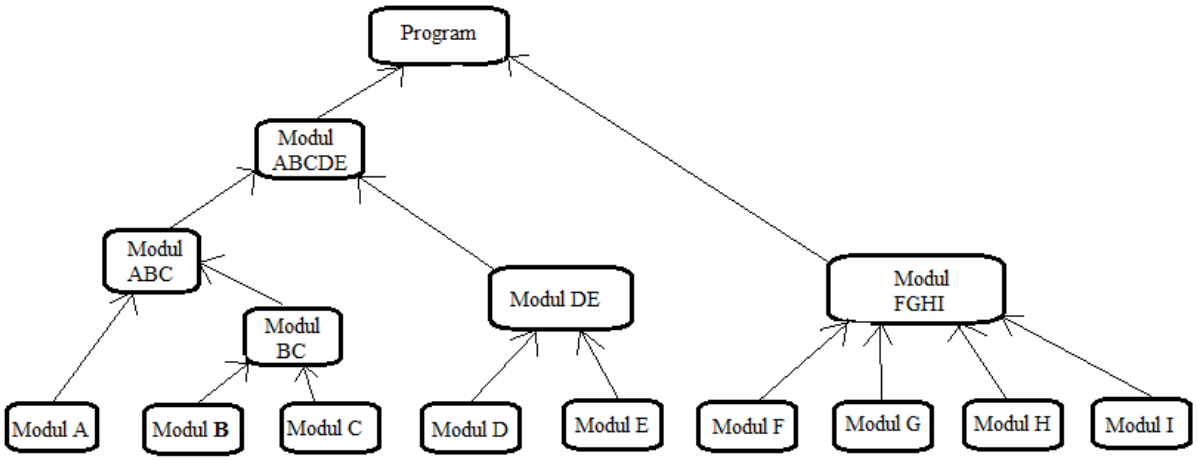
V-model je dobio naziv prema grafičkom prikazu koji ima oblik slova V. Lijeva strana modela prikazuje slijed aktivnosti koje programeri provode za razvoj, definiraju se zahtjevi, zatim u samom špicu se odvija implementacija definiranih zahtjeva, kako korisničkih tako i tehničkih. Dok se desna strana sastoji od niza aktivnosti koje provode testeri izvršavajući testiranje i integraciju pojedinih definiranih specifikacija. Korištenjem ovakvog načina rada greške se puno brže pronalaze i popravljaju, pa je time ovaj model puno uspješniji za razliku od prethodnog, vodopadnog modela [30].

Iz slike se vidi da u modelu postoje funkcionalne vrste testiranja. A to su redom testiranje modula (ili testiranje jedinica, jedinično testiranje, *unittesting*), zatim integracijsko testiranje, testiranje sustava, te testiranje prihvatljivosti. Ove četiri razine testiranja su malo detaljnije objašnjene u nastavku rada.

3.2.1. Testiranje jedinice

Svaka aplikacija se sastoji od puno malih komponenti ili modula koji zajedno tvore gotovu aplikaciju, pa samim tim i sve funkcionalnosti aplikacije. Moduli su najmanje jedinice

aplikacije, a to su na primjer metode, funkcije, objekti, klase, itd. Aplikacija se može simbolički prikazati na sljedeći način, Slika 8.



Slika 8. *Rastav programa (aplikacije) na module*

Softverske aplikacije su ogromne prirode i izazov je testirati cijeli sustav. To može dovesti do mnogih praznina u pokrivenosti testom. Stoga se prije prelaska na integracijsko testiranje ili funkcionalno testiranje preporučuje započeti s testiranjem modula. Ovakvo testiranje je svojevrsno testiranje bijele kutije. Postoji strategija ispitivanja i plan ispitivanja modula. Za svaki modul postoji testni scenarij koji će se dalje raščlaniti u testnim slučajevima.

Testiranje modula, naziva se još i jedinično testiranje (*engl. unit testing*), testiranje komponenti, programsko testiranje. Komponenta je isto što i modul, najniža jedinica bilo koje primjene. Dakle, testiranje komponenata, kao što i samo ime govori je tehnika testiranja najniže ili najmanje jedinice bilo koje aplikacije [31]. Ovisno o načinu implementiranja programskog koda jedinica može predstavljati klasu, metodu, postupak, objekt, funkciju, itd. Jednostavnije rečeno, programska jedinica je dio ili komad koda koji se poziva izvan jedinice i koji može pozvati druge programske jedinice. Prije nego što testiramo cijeli sustav, neophodno je da se temeljito ispita svaka komponenta aplikacije. U tom se slučaju moduli ili jedinice ispituju neovisno, bez doticaja sa drugim jedinicama. S ovakvim načinom testiranja možemo biti sigurni da se sve otkrivene greške odnose samo na tu testiranu jedinicu, pa ih je zbog toga jako jednostavno otkloniti, odmah se ispravljaju i nema nikakvih posljedica. Svaka jedinica prima ulaz, vrši neku obradu i generira izlaz. Izlaz se zatim potvrđuje prema očekivanoj značajci. Ovo

testiranje je od velike pomoći pri održavanju programskog koda, zbog toga što se osigurava da nastale promjene ne utječu na ostatak programskog koda [31].

Prema tome može se reći da je glavni cilj testiranja modula provjeriti ulazno / izlazno ponašanje testnog objekta. Osigurava da funkcionalnost testnog objekta radi ispravno i potpuno u redu prema željenoj specifikaciji. Testove najčešće pišu i izvode sami programeri nakon završetka pisanja najmanje jedinice u programu [32].

3.2.2. Integracijsko testiranje

Nakon testiranja modula dolazi integracijsko testiranje. U stvarnom svijetu, kada se aplikacije razvijaju, raščlanjuju se na manje module, a pojedinačni programeri dobivaju jedan modul. Logika koju implementira jedan programer prilično se razlikuje od drugog programera, pa postaje važno provjeriti je li logika koju implementira programer u skladu s očekivanjima i daje li točnu vrijednost u skladu s propisanim standardima. Često se struktura podataka mijenja kad putuje od jednog do drugog modula, neke se vrijednosti dodaju ili uklanjaju, što uzrokuje probleme u kasnijim modulima. Moduli također komuniciraju s nekim alatima ili API-jevima treće strane koji također trebaju biti testirani da li su podaci koji su prihvaćeni tim API-jem alatom točni i da je generirani odgovor također očekivan. Vrlo čest problem u testiranju je česta promjena zahtjeva. Mnogo puta programer primjenjuje promjene bez da ih jedinica testira. Tada integracijsko testiranje postaje važno [33].

Integracija je proces kojim se povezivanjem nekoliko programskih jedinica formira veća jedinica ili podsustav softvera. Nakon procesa povezivanja iako je svaka programska jedinica testirana, potrebno je testirati da li ispravno surađuju. Prema tome integracijsko testiranje je proces testiranja povezanosti različitih jedinica i njihove međusobne interakcije [34].

Kako bi se integracijsko testiranje lakše izvršilo u cilju je grupirati programke jedinice u smislene cijeline. Za grupiranje postoje dva pristupa, pristup velikog praska (*engl. big bang*) te inkrementalni pristup.

Prvi, pristup velikog praska je pristup koji integrira sve programske jedinice odjednom te se izvršava testiranje. Ovakav način implementacije je ekstreman. Prednost mu je ta što su svi moduli gotovi u trenutku testiranja, dok mu je negativno to što se u jednom trenutku dogodi

dosta promjena pa se zahtijeva i veći broj testova na koje se troši puno vremena. S obzirom na to, puno je teže otkriti izvor nastalog problema.

Drugi pristup, inkrementalni, je pristup koji postepeno integrira programske jedinice jednu po jednu. Prednost ovog načina rada je što se jednostavno može otkriti uzrok problema jer nema puno novih jedinica, ali nedostatak je što u aplikaciji nisu napravljene sve promjene zbog čega se onda moraju pisati zamjenski kodovi. Ovakvom testiranju se može pristupiti na sljedeće načine: [34]

- od dolje prema gore - testiranje od dolje prema gore, kao što i samo ime govori, započinje od najnižeg ili najunutarnjeg modula aplikacije i postupno se kreće prema gore, gornjim modulima. Pristup se nastavlja sve dok se svi moduli ne integriraju i cijela aplikacija ne testira kao jedna cjelina.
- od gore prema dolje - ova tehnika započinje od gornjeg modula i postupno napreduje prema donjim modulima. Jedino se gornji modul testira u izolaciji. Nakon toga se donji moduli integriraju jedan po jedan. Postupak se ponavlja sve dok se svi moduli ne integriraju i ne ispitaju.

Nakon što smo uvjereni da se unit testovi uspješno izvršavaju, integracijsko testiranje uspješno izvršeno, te su greške uklonjene, može se prijeći na testiranje sustava.

3.2.3. Testiranje sustava

Testiranje sustava (*engl. system testing*) započinje nakon završetka integracijskog testiranja, te se izvršava pred samim krajem razvojnog procesa, netom prije objavljivanja aplikacije. Ovim testiranjem se provjerava radi li aplikacija prema očekvanjima ili ne, odnosno zadovoljava li integrirani sustav specificirane zahtjeve. Cijeli sustav je potrebno testirati i zbog toga što postoje neke funkcionalnosti koje mogu ispravno raditi tek ako su u cijelosti integrirane skupa sa ostatkom sustava. Testiranje se provodi uglavnom po principu testiranja crne kutije.

Testiranjem se ocjenjuje rad sustava s gledišta korisnika, ne zahtijeva nikakvo interno znanje o sustavima, poput dizajna ili strukture koda [35]. Upravo zbog toga testiranje provode testeri što povećava vjerojatnost pronalaženja grešaka. S obzirom na to da su testeri neovisni ne može doći do sukoba interesa razvoja i testiranja.

3.2.4. Korisnički test prihvatljivosti

Korisnički test prihvatljivosti provjerava ponašanje sustava u odnosu na zahtjeve naručitelja. Provodi se najčešće zajedno s timom naručitelja. Cilj testa je pokazati da sustav zadovoljava sve zahtjeve naručitelja, te da je spreman za uporabu. Prilikom ovakvog testiranja aplikacija se daje korisnicima na testiranje koji u kontroliranim uvjetima testiraju i isprobaju aplikaciju. Postoje dvije faze testiranja prihvatljivosti. Prva faza se naziva alfa testiranje koje se odvija u kontroliranim uvjetima gdje odabrani korisnici isprobavaju aplikaciju i bilježe se njihove reakcije i problemi s kojima se susreću. Druga faza se naziva beta testiranje. Faza u kojoj se aplikaciju učini dostupnom dijelu pravih korisnika koji koriste aplikaciju u nekontroliranim uvjetima. Zatim korisnici šalju svoje reakcije, mišljenja i probleme programerima koji onda ispravljaju pronađene greške [36].

Primjer objedinjenje testova integracije i primjena testiranja jedinica (modula) nalazi se u praktičnom dijelu rada, 5. poglavlje. Ali prije izrade upoznat ćemo se općenito s automatizacijom testiranja, testiranjem u programu C#, alatima za testiranje jedinica u C# i ponekim testnim razvojnim tehnikama.

3.3. Ručno testiranje

Ručno (*engl. manual*) testiranje najstarija je verzija testiranja. Testiranje u kojem sam tester sastavlja testne scenarije koji trebaju biti napisati tako da provjeravaju aplikaciju i njene rubne slučajeve. Tester je upućen u cijelu aplikaciju i zna sve funkcionalnosti koje ona pruža. Nakon što sastavi testne scenarije, tester izvršava te testove i označava ih jesu li prošli (dogodilo se što je očekivano) ili su pali (dogodilo se nešto što se nije trebalo dogoditi). Ako neki test nije prošao, tada je bitno imati zapis koje su sve akcije dovele do tog pada, te što se točno dogodilo. Pomoću tih zapisa programeri znaju ponoviti korake, zatim uvidjeti što je pošlo po krivu, te ispraviti grešku, nakn toga ponovno testiranje obavljaju testeri kako bi potvrdili da su greške ispravljene. Problem kod ručnog testiranja je što zahtijeva puno vremena i ljudskih resursa, a što aplikacija postaje veća i kompliciranija, problem raste. Ručno testiranje se ne smatra konzistentnim ni ponovljivim jer svaki ručni tester može pristupiti testnom scenariju ili testu drugačije. Razlike u brzini izvođenja nekih operacija ili mjestu klika mogu rezultirati drugačijim rezultatima [37]. Kako bi se riješili problemi ručnog testiranja, preporučuje se

automatsko testiranje gdje se pišu skripte koje izvršavaju zadane testove, što će biti detaljnije objašnjeno u sljedećem poglavlju.

4. Automatizacija testiranja

Kao što je već spomenuto, testiranje je jedan od ključnih koraka pri razvoju programskih proizvoda. Provedbom testiranja smanjuje se broj pogrešaka u sustavu te se osigurava isporuka kvalitetnijeg sustava korisniku. Osim što testiranje možemo provoditi ručno, kao što je opisano u prethodnim poglavljima, proces testiranja moguće je automatizirati. Zašto se odlučiti za automatizirano testiranje? Zato jer manualno testiranje troši puno vremena i novaca, teško je ručno testirati softver pisan na više jezika, još jedan od razloga je to što automatizacija ne zahtijeva ljudsku intervenciju (automatizirani test se može pokrenuti bez nadzora), zatim automatizacija pomaže povećati pokrivenost testiranja i povećava brzinu izvršavanja testa, dok je ručno testiranje zbog ponavljanja iste stvari ne rijetko sklono ljudskoj pogrešci.

Automatizirano testiranje podrazumijeva korištenje alata za automatizaciju koji izvršavaju testiranje softvera, a za to su potrebna znatna ulaganja novaca i resursa. Uzastopni razvojni ciklusi zahtijevat će ponavljanje istog skupa testova, pa je onda pomoću alata za automatizaciju testiranja moguće snimiti paket za testiranje, te ga reproducirati ponovo prema potrebi. Jednom kada je paket za testiranje automatiziran, nije potrebna nikakva ljudska intervencija [38]. Cilj automatizacije je smanjiti broj testnih slučajeva koji se ručno pokreću, a ne u potpunosti eliminirati ručno testiranje. Nadalje u ovom poglavlju je opisano automatsko testiranje programa.

4.1. Pojam automatskog testiranja

Automatsko testiranje pojavilo se krajem 80-tih godina 20.stoljeća razvitkom prvih sustava za automatizaciju [39]. Automatizacija testiranja je vještina veoma različita od samog testiranja. Da bi se dobila korist od automatizacije testiranja, testovi koji se automatiziraju moraju biti pažljivo odabrani i implementirani. Prema Paulu Ammannu i Jeffu Offuttu, automatsko testiranje je proces korištenja softvera za kontrolu izvršenja testova, usporedbu stvarnih s unaprijed predviđenim ishodima i za druge funkcije izvješćivanja o testiranju i kontroli [40]. Prema tvrdnjama, te vlastitom dosad stečenom znanju o procesu testiranja mogu

reći da je automatsko testiranje proces brze provjere i kontrole velikog skupa funkcionalnosti sustava te nam takvo testiranje pruža izvještaje o razlikama izlaznih i ulaznih parametara. Paul i Jeff smatraju da testiranje softvera može biti skup i intenzivan posao, te je zbog toga u cilju automatizirati što veći broj funkcionalnosti sustava [40].

Mnoge organizacije se začude kada vide da je skuplje automatizirati testiranje nego ga jednom izvršiti ručno. S tim se slažu Mark Fewster i Dorothy Graham koji razlog ovome navode potrebu za opreznim odabirom testova koji će se automatizirati. Smatraju da će test biti ekonomičniji za organizaciju, ako je ispravno automatiziran tako što je nužno napraviti analizu te odrediti koji će se dijelovi sustava automatizirati. Kada je test prvi puta automatiziran, on će biti manje ekonomičniji (budući da je potrebno mnogo truda da bi se automatizirao). Ali nakon što je automatizirani test pokrenut mnogo puta postat će mnogo isplativiji od testa izvedenog ručno. [41]. Kako bi se odredilo koj će se dijelovi automatizirati, potrebno je provjeriti koji su od napisanih testova najpodložniji automatizaciji, Za to se upotrebljava pojam ispitljivosti testa (*engl. testability*). Amman navodi da je ispitljivost stupanj do kojega sustav ili komponenta sustava olakšava određivanje testnih kriterija te provođenje testova kako bi se odredilo da li su zadovoljeni ti kriteriji [42]. U današnje vrijeme sve češća pojava je automatiziranje testiranja, skoro sve organizacije imaju bar neki dio funkcionalnosti koji se automatski testira. Razlog sve većem značaju automatiziranja testiranja može se vidjeti iz istraživanja u knjizi o budućnosti testiranja Mukesh Sharma. U knjizi Mukeh iznosi rezultate ispitivanja o povećanju automatizacije: 71 % ispitanika tvrdi da je otkrivanje pogrešaka poboljšano, 70 % ispitanika ističe bolju kontrolu i transparentnost testova, njih 39 % je prepoznalo cikluse testiranje koji traju kraće, te 66 % ispitanika svjedoči smanjenju troškova testiranja [43]. S obzirom na ovakve rezultate istraživanja, organizacijama je to bio dovoljan razlog da u svoj proces razvijanja sustava uvedu automatizaciju testiranja.

Automatizacija se može i treba primijeniti na različite načine i u različitim kontekstima. Još uvijek ne postoji jedna strategija koja će uvijek dati podjednako dobre rezultate.

4.2. Svrha automatskog testiranja

Iako je već rečeno da je osnovna svrha poboljšanje i ubrzavanje procesa testiranja, Fewster i Graham naveli su još nekoliko značaja automatskog testiranja [44]:

- Omogućeno je pokretanje postojećih (regresijskih) testova na novoj verziji programa. Jedan od najvažnijih zadataka, osobito u okruženju u kojem se softver često mijenja. Budući da testovi već postoje i provedeni su na ranijoj verziji programa, napor za obavljanje testova je minimalan.
- Često izvršavanje istog testa jedna je od glavnih mogućnosti automatizacije, odnosno mogućnost da se pokrene više testova u manje vremena, što omogućuje češće pokretanje i provedbu testova. Također je moguće postaviti automatsko pokretanje testova te na taj način se testovi sami pokreću u određeno (zadano) doba dana.
- Može izvršiti testove koje bi bilo teško ili nemoguće provesti ručno. Provesti test uživo na mreži od 300 korisnika može biti nemoguće, ali zato unos od 300 korisnika može biti u sistemu automatiziranog testa. Testovi korisničkih scenarija mogu se pokrenuti u bilo kojem trenutku i može ga pokrenuti i tehničko osoblje koje ne razumije zamršenost cjelokupnog testa.
- Automatskim testiranjem omogućuje se bolja raspodjela resursa tj. automatiziranje ručnih, jednostavnih i ponavljajućih zadataka, kao što je unos istih testnih ulaza. Poboljšava radnu sposobnost i s tim oslobađa kvalificirane testere obavljanja takvih zadataka. Tako testeri umjesto toga mogu uložiti više truda u osmišljavanje boljih testnih slučajeva.
- Omogućena je dosljednost i ponovljivost testiranja. Testovi koji se ponavljaju, svaki puta će se točno ponoviti, to daje razinu dosljednosti koju je teško postići ručnim testiranjem. Uvođenjem automatiziranog testiranja možemo osigurati dosljedne standarde kako u testiranju tako i u samom razvoju.
- Ponovno korištenje automatskih testova. Rad uložen u odlučivanje što će se sve testirati, zatim napor uložen u dizajniranje testova, te izgradnja testova se može smanjiti automatizacijom, zbog toga što će se isti test moći koristiti više puta.
- Automatskim testiranjem smanjuje se vrijeme potrebno za testiranje, pa će softver prije završiti na tržištu. Kada je test automatiziran, ponavljanja se obavljaju mnogo brže nego kad bi se testiranje obavljalo ručno.
- Povećano povjerenje u sigurnost sustava. Ako dobro definirani skup automatiziranih testova je uspješno pokrenut, te testiranje prođe bez grešaka, povećava se sigurnost sustava. Omogućuje korisnicima kada se softver pusti u prodaju da neće biti problema u sustavu, ondnosno neugodnih iznenađenja.

4.3. Microsoft visual studio

Microsoft visual studio je integrirano razvojno okruženje (IDE) u kojem programeri rade pri stvaranju programa na jednom od mnogih jezika. Podržava 36 različitih programskih jezika. Snažan je IDE koji osigurava kôd kvalitete tijekom cijelog životnog ciklusa aplikacije, od dizajna do implementacije. Neki se prozori koriste za pisanje koda, neki za projektiranje sučelja, a drugi za dobivanje općeg pregleda datoteka ili klasa u oglednoj aplikaciji. Microsoft Visual Studio uključuje i niz vizualnih dizajnera koji pomažu u razvoju različitih vrsta aplikacija. Koristi se za stvaranje konzola i aplikacija za grafičko korisničko sučelje (GUI) zajedno s aplikacijama Windows Forms ili WPF (Windows Presentation Foundation), web aplikacijama i web uslugama u oba izvorna koda zajedno s upravljanim kodom za sve platforme koje podržavaju Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework i Microsoft Silverlight.

Visual Studio nudi skup alata koji pomažu u pisanju i izmjeni koda programa, a također otkrivaju i ispravljaju pogreške u programima. Pruža fleksibilan i učinkovit način za pokretanje jediničnih testova i pregled njihovih rezultata u Visual Studiu. Instalira Microsoftove okvire za testiranje jedinica za upravljani i izvorni kôd [45].

U ovom radu bavit ćemo se izradom testova integracije koji testiraju upravo u ovom razvojnom okruženju, a za izradu ćemo korisiti preporučeni i najčešće korišten jezik, a to je C#.

4.4. Jedinično testiranje u C#

C# je moderan objektno orijentirani programski jezik, te se može se reći da je C# među glavnim programskim jezicima u Microsoft Visual Studiu. Dizajniran je za izradu raznih aplikacija koje se izvode na .NET Framework–u. U kombinaciji s .NET Framework -om, C# omogućuje stvaranje Windows aplikacija, web usluga, alata za baze podataka, komponenti, kontrola i još mnogo toga. S obzirom na temu ovog rada bitno je i da C# osigurava definiranje i pokretanje automatskih jediničnih testova za održavanje ispravnosti koda, osigurava pokrivenost koda te pronalaženje pogrešaka i grešaka prije nego što to učine korisnici [46].

Svatko tko razmišlja o tome da učini nešto novo, a u ovom slučaju automatsko testiranje, trebao bi se razumno zapitati "što mi to znači?" Drugim riječima, prije nego što to učinite, trebali biste razumjeti njegovu vrijednost. Zašto jedinični test? Jedinično testiranje jer se

funkcionalnost oglednog programa raščlanjuje na diskretna ponašanja koja se mogu testirati i koja se mogu testirati kao pojedinačne jedinice. Jedinični testovi pružaju programerima i testerima brz način traženja logičkih pogrešaka u metodama klasa. Jedinstveno testiranje također ima najveći utjecaj na kvalitetu koda kada je sastavni dio procesa razvoja softvera. Čim se napiše funkcija ili neki drugi blok aplikacijskog koda, potrebno je izraditi jedinične testove koji provjeravaju ponašanje koda kao odgovor na standardne, granične i netočne slučajeve ulaznih podataka i koji provjeravaju sve eksplicitne ili implicitne pretpostavke koje je napravio kôd.

Jedna od najizazovnijih stvari koju treba učiniti prije svega je 'napraviti pravi izbor', a donošenje odluke postaje još kompliciranije kada je pred nama više mogućnosti. Potrebno je upotrijebiti okvir za jedinično testiranje za kreiranje jediničnih testova, njihovo pokretanje i izvještavanje o rezultatima testova. Trebali bi odabrati najprikladniji okvir za testiranje koji odgovara određenim projektnim zahtjevima zbog toga jer C# podržava tri glavna okvira za jedinično testiranje, a to su MSTest, NUnit i XUnit. Nadalje, u ovom radu ćemo pojedinačno proći kroz glavne značajke svakog okvira, te ćemo vidjeti i detaljnu usporedbu između okvira, kako bi lakše donijeli odluku o odabiru [47].

U sva tri programska okvira osnovna provjera ispravnosti rada programskog koda izvodi se najčešće putem assert funkcija. Assert u prijevodu znači tvrditi, te se iz toga može zaključiti kako se ovom funkcijom provjerava tvrdnja. Tvrdnja je povratna vrijednost funkcije. Putem assert-a može se vidjeti je li test vratio očekivanu povratnu vrijednost, tvrdnju ili nije. Posljednji korak kod pisanja testa je provjera izlaza sa znanim odgovorom. Postoje neki opći principi za pisanje tvrdnji:

- pobrinuti se da testovi budu ponovljivi i pokrenuti testove više puta kako bi dali isti rezultat svaki put
- potvrditi rezultate koji se odnose na ulazne podatke

Unittest dolazi sa mnogo metoda za potvrđivanje vrijednosti, tipova i postojanje varijabli. Evo neke od češće korištenih metoda [48]:

Metoda	**Testiranje**
Assert.AreEqual(objekt, objekt)	Testira jesu li navedeni objekti jednaki. Različiti numerički tipovi tretiraju

	se kao nejednaki čak i ako su logičke vrijednosti jednake. 42L nije jednako 42.
Assert.AreNotEqual(objekt, objekt)	Testira jesu li navedeni objekti nejednaki. Različiti numerički tipovi tretiraju se kao nejednaki čak i ako su logičke vrijednosti jednake. 42L nije jednako 42.
Assert.AreSame(objekt, objekt)	Testira odnose li se navedeni objekti (ulazi) na isti objekt.
Assert.AreNotSame(objekt, objekt)	Testira odnose li se navedeni objekti (ulazi) na različite objekte.
Assert.Equals(objekt, objekt)	Statički jednaki preopterećenja, koriste se za usporedbu instanci dva tipa za referentnu jednakost. Ova se metoda **ne** bi trebala koristiti za usporedbu dva slučaja za jednakost.
Assert.Fail()	Uvijek vraća klasu AssertFailedException. Klasa AssertFailedException koristi se za označavanje greške u testnom slučaju.
Assert.IsFalse()	Testira dali je navedeni uvjet lažan
Assert.IsTrue()	Testira dali je navedeni uvjet istinit
Assert.IsInstanceOfType(object, type)	Testira dali je navedeni objekt instance očekivanog tipa.
Assert.IsNotIstanceOfType(object, type)	Testira nije li navedeni objekt instanca pogrešnog tipa.
Assert.IsNotNull(objekt)	Testira dali je navedeni objekt ne-null.
Assert.IsNull(objekt)	Testira dali je navedeni objekt null.
Assert.ReplaceNullChars()	Zamjenjuje null znakove ('\ 0') s "\ 0".

Tablica 3. *Primjer unittest metoda*

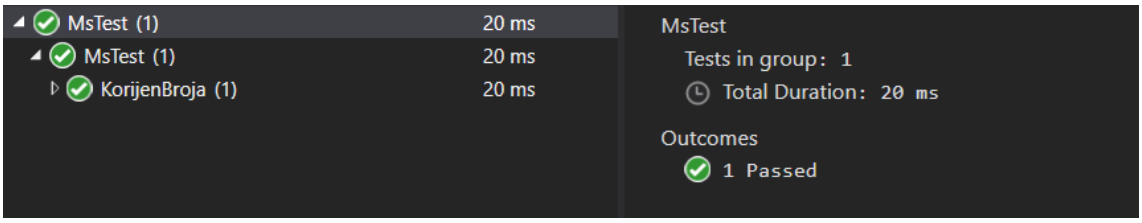
Slika 9. nam pokazuje primjer jediničnog testiranja u C#, jedan primjer među najjednostavnijim, a to je testiranje metode vađenja drugog korijena. Metode će biti testirane u okviru MSTesta koji će u sljedećem poglavlju biti detaljnije objašnjen.

Metoda *Math.Sqrt ()* je metoda klase Math koja se koristi za izračun drugog korijena ulaznog broja. Kvadrat je broj pomnožen sam sa sobom stoga predstavlja očekivanu vrijednost koja u konačnici mora biti ista kao vrijednost početnog broja da bi test bio uspješan.

```
[TestMethod]
[DataTestMethod]
[DataRow(36)]
[DataRow(79)]
0 references
public void ProvjeraKorijenaBroja(int broj)
{
    double korijen = Math.Sqrt(broj);

    double ocekivanBroj = korijen * korijen;

    Assert.AreEqual(ocekivanBroj, broj);
}
```

Slika 9.

Rezultat se nalazi na slici ispod (Slika 10.)



Slika 10.

4.4.1. MSTest

Okvir MSTest za automatsko testiranje zadani je okvir za testiranje koji dolazi uz Visual Studio. Ranijih dana počeo je kao alat za naredbeni redak za izvršavanje testova. MSTest postoji od Visual Studia 2015. Kada se prvi put pojavio, nije imao načina proslijediti parametre u vaše jedinične testove. Iz tog razloga, mnogi su se ljudi odlučili umjesto toga koristiti NUnit. Budući da V2 MSTest također podržava parametre, pa se razlika između okvira na svakodnevnoj osnovi znatno smanjila. Također se naziva i Visual Studio Unit Testing Framework, međutim, naziv MSTest više je sinonim za programere. MSTest pruža potrebne

alate za provjeru i provjeru izvornog koda. Okvir prepoznaje testove putem različitih atributa/napomena pod kojima je prisutan kôd testa. Najpopularniji atributi su:

- [TestMethod] - Označava metodu, tj. Stvarni test slučaj u ispitnoj klasi.
- [TestClass] - Označava klasu koja sadrži testove.
- [DataRow] - Omogućuje postavljanje vrijednosti parametara testa. U kodu ih može biti prisutno više od jednog.
- [DataTestMethod] - Nosi istu funkcionalnost kao atribut [TestMethod], osim što se koristi kada se koristi atribut [DataRow].

MSTest ima podršku za više platformi i može se proširiti pomoću prilagođenih atributa testa i prilagođenih tvrdnji. Izvođenje paralelnog testa podržava okvir MSTest, gdje je paralelizam moguć na razini metode ili na razini klase. Budući da okvir MSTest dolazi unaprijed u paketu s Visual Studiom, programeri koji za razvoj i testiranje koriste Visual Studio IDE preferiraju okvir MSTest u odnosu na druge okvire za testiranje poput NUnit, xUnit.Net itd. Međutim, izbor i preferencije ovisit će i o vrsti i složenosti projekta [50].

Istaknute značajke MSTest :

- Okvir MSTest predstavljen je prije nekoliko godina sa snažnim značajkama koje su idealne za korištenje okvira MSTest za unakrsno testiranje preglednika.
- MSTest je otvorenog koda, a projekt se nalazi na GitHubu. Javna spremišta MSTest su Microsoft/testfx i Microsoft/testfx-docs. Budući da je projekt otvorenog koda, dopušta doprinose zajednice.
- Podrška za više platformi - okvir MSTest je višeplatformska implementacija okvira pomoću koje programeri mogu pisati testove koji ciljaju .NET Framework, .NET Core i ASP.NET Core na platformama poput Linuxa, Mac i Windows.
- Proširivo - Poput ostalih okvira za testiranje, okvir MSTest sada se može proširiti prilagođenim atributima testa i prilagođenim tvrdnjama.
- Testiranje na temelju podataka - okvir omogućuje korisnicima da definiraju ponašanje svojih testova pružajući mogućnost da podaci upravljaju testovima. Postavljanjem testova na podatke, jedna se metoda može izvršiti više puta davanjem različitih ulaznih argumenata.
- Napomene - okvir MSTest omogućuje prilagođavanje izvođenja životnog ciklusa izvođenja testa putem napomena, npr. [TestClass], [TestMethod] itd.

- Izvođenje paralelnog testa - koristeći MSTest, testovi se mogu izvoditi paralelno, skraćujući vrijeme izvođenja testa.

4.4.2. NUnit

NUnit je otvoreni izvorni okvir za testiranje jedinica u C# koji je prenijet iz JUnit okvira za automatizirano testiranje. Najnovija verzija NUnit-a je NUnit3 koja je prepisana s mnogo novih značajki i ima podršku za širok raspon .NET platformi. Testiranje automatizacije u NUnit -u može se izvesti pomoću grafičkog sučelja Visual Studio i konzole NUnit (NUnit.ConsoleRunner). Pomoću okvira NUnit testovi se mogu izvoditi serijski i paralelno. Paralelno izvođenje testa moguće je na razini klase ili metode. Kad je omogućen paralelizam, prema zadanim postavkama mogu se paralelno izvesti četiri testa. Okvir NUnit koristi sustav stilova atributa/naredbi kao i drugi okviri za testiranje, a najpopularniji atributi NUnit koji se koriste za testiranje automatizacije su [51]:

- [Test] - Označava metodu, tj. Stvarni test slučaj u ispitnoj klasi.
- [TestFixture] - Označava klasu koja sadrži testove.
- [TestCase] - Omogućuje postavljanje vrijednosti parametara testa.

Istaknute značajke NUnit testiranja:

- Napomene korištene u NUnit-u pomažu u ubrzavanju razvoja i izvođenja testova jer se testovi mogu izvoditi s brojnim ulaznim vrijednostima.
- TDD (Test Driven Development) je prvenstveno koristan jer su jedinični testovi ključni u pronalaženju problema/grešaka tijekom ranih faza razvoja proizvoda.
- Omogućuje paralelno pokretanje testnih slučajeva.
- Pomoću NUnit-a mogu se izvoditi testni slučajevi iz konzole, bilo pomoću alata za testiranje automatizacije treće strane ili pomoću NUnit testnog adaptera unutar Visual Studia.

4.4.3. XUnit

XUnit.Net je okvir za testiranje otvorenog koda koji se temelji na .NET okviru. 'x' označava programski jezik, npr. JUnit, NUnit, itd. Kreatori NUnit-a stvorili su xUnit jer su

htjeli izgraditi bolji okvir umjesto dodavanja inkrementalnih značajki u okvir NUnit. Kako su kreatori okvira NUnit napisali xUnit, između ta dva okvira postoji mnogo sličnosti. Namjera stvaranja novog okvira za testiranje jedinica bila je izgradnja mnogo boljeg okvira od početka. Za razliku od drugih popularnih okvira za testiranje automatizacije, poput NUnit i MSTest, ovaj testni okvir slijedi jedinstveniji stil atributa/napomena.

Najpopularniji atributi/napomene korišteni u okviru XUnit:

- [Fact] - Označava metodu, tj. Stvarni test slučaj u ispitnoj klasi.
- [Trait] - Koristi se za postavljanje proizvoljnih metapodataka na testu
- [Theory] - Ovaj se atribut koristi kada se moraju izvesti testovi na temelju podataka. U takvim se slučajevima umjesto atributa [Fact] mora koristiti [Theory]
- [ClassData] - Ovaj se atribut koristi kada parametri koji se prosljeđuju testovima [Theory] nisu konstante. [Theory] [Podaci o klasi (vrsta (neki podaci))]

Istaknute značajke xUnit testiranja:

- Okvir xUnit ima manje atributa u usporedbi s okvirima NUnit i MSTest.
- Druga prednost korištenja xUnit-a je ta što okvir za automatizaciju testa stvara novu instancu testne klase za svaki test [52].

4.4.4 Temeljne razlike između MSTest, Nunit i Xunit okvira

Okviri xUnit NUnit i MSTest se razlikuju na više frontova. Pogledajmo osnovne razlike [53]:

- **Izolacija testova** - Okvir xUnit pruža mnogo bolju izolaciju testova u usporedbi s okvirima NUnit i MSTest. Za svaki testni slučaj ispitna se klasa instancira, izvršava i odbacuje nakon izvođenja. To osigurava da se testovi mogu izvoditi bilo kojim redoslijedom jer postoji smanjena/nikakva ovisnost između testova. Izvođenje svakog testa kao zasebne instance minimizira šanse da jedan test uzrokuje neuspjeh drugih testova!

- **Proširivost** - Kada gledamo NUnit vs. XUnit vs. MSTest, proširivost igra važnu ulogu u odabiru određenog okvira za testiranje. Izbor može ovisiti o potrebama projekta, ali u

nekim scenarijima proširivost može okrenuti tablice za određeni okvir za testiranje.U usporedbi s okvirima MSTest i NUnit, okvir xUnit je proširiviji.

- **Inicijalizacija i deinicijalizacija** - NUnit i MSTest koriste parove atributa za postavljanje aktivnosti vezanih uz inicijalizaciju i deinicijalizaciju testnog koda. S druge strane, xUnit koristi konstruktor klase za provedbu koraka koji se odnose na inicijalizaciju testa i sučelje IDisposable za implementaciju koraka koji se odnose na deinicijalizaciju. XUnit pokreće novu instancu po testu, dok se u okvirima NUnit & MSTest svi testovi izvode u istoj klasi.

- **Mehanizam tvrdnji** - Okvir xUnit koristi Assert.Throws umjesto [ExpectedException] koji se koristi u NUnit i MSTest. Nedostatak korištenja [ExpectedException] je taj što se pogreške možda neće prijaviti ako se pojave u pogrešnom dijelu koda. Na primjer, ako se za sigurnosnu iznimku mora pokrenuti potvrda, ali se dogodi iznimka provjere autentičnosti, [ExpectedException] neće pokrenuti potvrdu. S druge strane, Assert.Throws pokreće assert čak i ako je iznimka generička. To osigurava da se potvrda pokreće čak i nakon pokretanja iznimke.

- **Izvođenje paralelnog testa** - Sva tri okvira za testiranje C# jedinica podržavaju paralelno izvršavanje testova. **NUnit -** paralelnost je moguća na razini djece (dječji se testovi izvode paralelno s drugim testovima), učvršćenja (potomci testa do razine testnih uređaja mogu se izvoditi paralelno), samostalno (sam test može se izvesti paralelno s drugim testovima) testovi) i sve (test i njegovi potomci mogu se izvršavati paralelno s drugima na istoj razini). **XUnit -** paralelnost stavljanjem test klasa u jednu testnu zbirku ili paralelnim izvršavanjem 'n' puta. **MSTest -** paralelnost na razini metode, kao i na razini klase.

5. Primjer ogledne aplikacije s jediničnim testiranjem u okviru MSTest

5.1. O aplikaciji

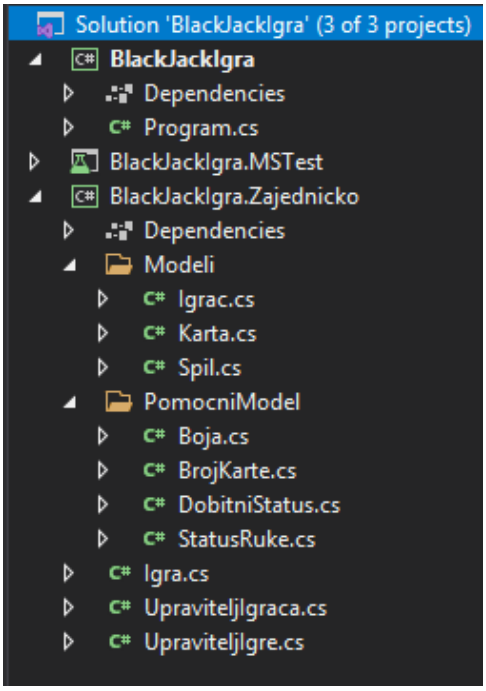
Praktični dio diplomskog rada uključuje izradu testove za igru Blackjack u C#. Primjenjeno je stečeno znanje o izradi testova koristeći ugrađeni *unittest*, odnosno MSTest. Cilj zadatka je testirati sve klase, metode i funkcije koje se koriste u aplikaciji.

Blackjack je jedna od najpopularnijih igara na sreću, svjetski je fenomen, ali ujedno i jedna od najstarijih kartaških igara. Blackjack se igra pomoću špila koji se sastoji od 52 karte u kojima se ne smije nalaziti joker, a sudionici igre su djelitelj karata (dealer) i igrači, u ovom primjeru jedan igrač čovjek i računalo kao djelitelj. Igrač igra protiv djelitelja, a igračev cilj je skupiti zbroj (vrijednost) karata koji je veći od vrijednosti karata djelitelja, ali tako da on ne iznosi više od 21. Zašto baš 21? Zato što je najveći zbroj dvije karte 21, tj. to je vrijednost asa i karte koja vrijedi 10. Ta se pobjednička kombinacija naziva blackjack. Ako vrijednost karata premašuje 21, igrač gubi igru. Vrednovanje karata je vrlo jednostavno: sve karte s brojevima vrijede isti taj broj, a sve karte s licima (dečko, dama i kralj) vrijede 10 bodova. As može vrijediti 1 bod ili 11 bodova, ovisno o tome koliki je zbroj ostalih karata. Ako zbrojene karte daju deset bodova ili manje as će vrijediti 11 bodova, a ako je zbroj veći od 11 bodova, onda as vrijedi 1. Može se dogoditi da se dobiju dva asa, ali u tom slučaju jedan mora vrijediti 11 (izuzev naravno ako ostale karte već nemaju više od 10 bodova). Na početku igre djelitelj će svakom igraču kao i sebi podijeliti dvije karte. Nakon što djelitelj podijeli karte, on će redom pitati svakog igrača želi li još jednu kartu (“hit” - tražiti još jednu kartu kako bi pokušao doći što bliže zbroju od 21 ili pogoditi točno 21) ili želi da stane na podijeljenim kartama, odnosno kartama u ruci (“stand” - ne tražiti još jednu kartu). Dakle, igrač može ostati na dvije karte koje su mu prvotno podijeljene ili može zatražiti djelitelja još karata, jednu po jednu, dok ne odluči stati (“stand”) na određenom zbroju karata (ako je zbroj 21 ili manje) ili dok ne pređe 21 („previše“ / “busted“). U ovom drugom slučaju, igrač gubi, a djelitelj pobjeđuje. Dakle, glavni cilj igre je nadmašiti djelitelja, i to tako da igrač postigne veći zbroj karata u ruci od djelitelja, a da je to broj manji od 21. Ako igračeve karte prijeđu zbroj od 21, automatski ispada iz igre i gubi.

5.2. Integracija

Korištena tehnologija za izradu ogledne aplikacije je Microsoft Visual Studio 2019., programski jezik C#, a svi testovi su programirani korištenjem ugrađenog okvira MSTest.

Aplikacija se sastoji od tri glavne klase, *Igra*, *UpraviteljIgraca* i *UpraviteljIgre*, tri modela (entiteti) *Igrac*, *Karta*, *Spil* i četiri pomoćna modela (enumeracija) *Boja*, *BrojKarte*, *DobitniStatus* i *StatusRuke*. Na slici (Slika 11.) nalazi se struktura direktorija u kojem se nalaze klase i testovi.



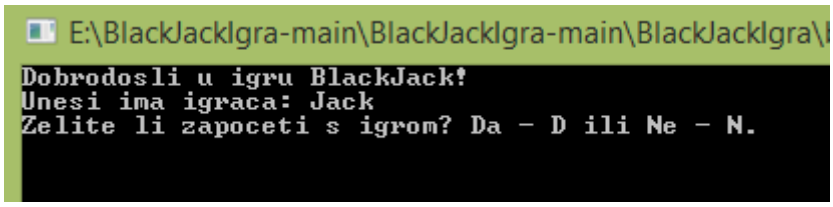
Slika 11.

Igra se pokreće pokretanjem metode u *mainu*, metodom *PokreniIgru* koja prima instancu klase *UpraviteljIgre* (Slika 12.).

```
using BlackJackIgra.Zajednicko;
using BlackJackIgra.Zajednicko.Modeli;
using BlackJackIgra.Zajednicko.PomocniModel;
using System;
using System.Collections.Generic;
using System.Linq;

namespace BlackJackIgra
{
    0 references
    class Program
    {

        0 references
        static void Main(string[] args)
        {
            UpraviteljIgre upraviteljIgre = new UpraviteljIgre();

            PokreniIgru(upraviteljIgre);

            Console.ReadLine();
        }
```

Slika 12.

Nakon pokretanja igre, od korisnika se traži unos imena igrača, te odabir želi li započeti s igrom, na slici možemo vidjeti izgled korisničkog sučelja (Slika 13.).

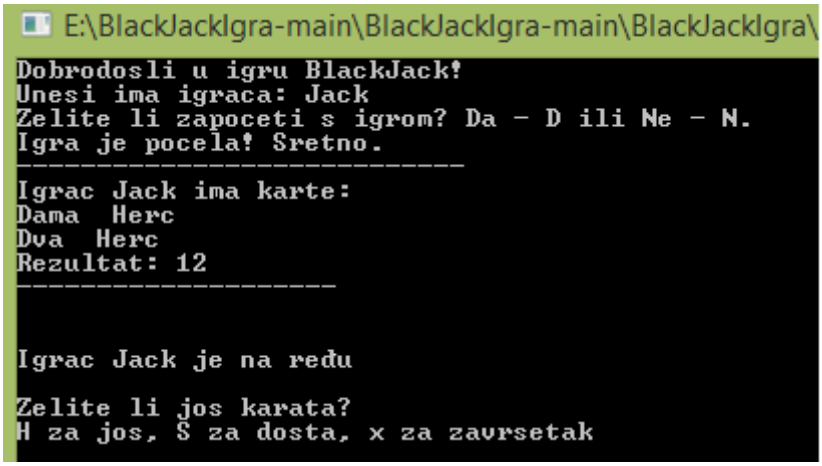


Slika 13.

Nakon šta korisnik (igrač) započne s igrom, u klasi *Igra* metodom *Dijeli* (Slika 14.) provjerava se pomoćni model *StatusRuke*, koji može biti *Open* (nastavak igre u slučaju da rezultat ne prelazi 21) ili *Done* (igra gotova, u slučaju kada je rezultat veći od 21). Kako se radi o početku igre *StatusRuke* je *Open*, te se poziva metoda *Reset* koja pomoću pomoćnih modela za enumeraciju *Boja* i *BrojKarte* pridjeljuje svakoj karti broj i boju, a zatim metodom *PromijesajKarte* svakoj karti pridjeljuje random mjesto u špilu, odnosno „miješa“ karte, te dodjeli igraču dvije karte.

```
public void Dijeli()
{
    if (StatusIgre == StatusRuke.Open)
        return;
    StatusIgre = StatusRuke.Open;
    igraci.RemoveAll(p => OdustaliIgraci.Contains(p.Ime));
    odustaliIgraci.Clear();
    spil.Reset();
    spil.PromijesajKarte();

    foreach (string name in NoviIgraci)
    {
        DodajIgraca(name);
    }
    noviIgraci.Clear();
    foreach (Igrac Igrac in Igraci)
    {
        Igrac.Ruka = new List<Karta> { spil.Dohvatikartu() };
        Igrac.DobitniStatus = DobitniStatus.Open;
        Igrac.StatusRuke = StatusRuke.Open;
    }
    Dealer.Ruka = new List<Karta> { spil.Dohvatikartu() };
    Dealer.DobitniStatus = DobitniStatus.Open;
    Dealer.StatusRuke = StatusRuke.Open;

    foreach (Igrac igrac in Igraci)
    {
        igrac.Ruka.Add(spil.Dohvatikartu());
    }

    Dealer.Ruka.Add(spil.Dohvatikartu());
    MoveActiveSlot();
    CheckIgracsForBlackjack();
}
```

Slika 14.



Slika 15.

Na slici (Slika 15.) se nalazi izgled korisničkog sučelja nakon što je igrač nastavio s igrom. Dealer (računalo) je iz „promješanog“ špila igraču podijelio dvije karte, te pomoću metode *IzracunajRezultatIgraca* (Slika 16.) i metode *DohvatiKartu* u klasi *Igra* i pomoću pomoćnih modela izračuna i ispisuje trenutni rezultat igrača (zbraja vrijednosti dobivenih karata).

```
public static void IzracunajRezultatIgraca(Igrac igrac)
{
    short rezultat = 0;

    if (igrac.Ruka == null)
        return;

    foreach (Karta nijeAs in igrac.Ruka.Where(c => c.BrojKarte != BrojKarte.As))
    {
        rezultat += DohvatiKartu(nijeAs);
    }

    foreach (Karta asKarta in igrac.Ruka.Where(c => c.BrojKarte == BrojKarte.As))
    {
        if ((rezultat + 11) <= 21)
        {
            rezultat += 11;
            continue;
        }
        rezultat += 1;
    }

    igrac.Rezultat = rezultat;
}
```

Slika 16. *Metoda IzracunajRezultatIgraca*

U sljedećem koraku igrač gleda na rezultat, te prema njemu odlučuje nastavak igre, odnosno pomoću metoda *Hit* (Slika 17) i *Stay* (Slika 18.) u klasi *Igra* igrač daje do znanja dealeru želi li da mu podijeli još jednu kartu (*Hit*) ili je zadovoljan rezultatom i ne želi još jednu kartu (*Stay*), ili treći slučaj u kojem igrač prekida igru.

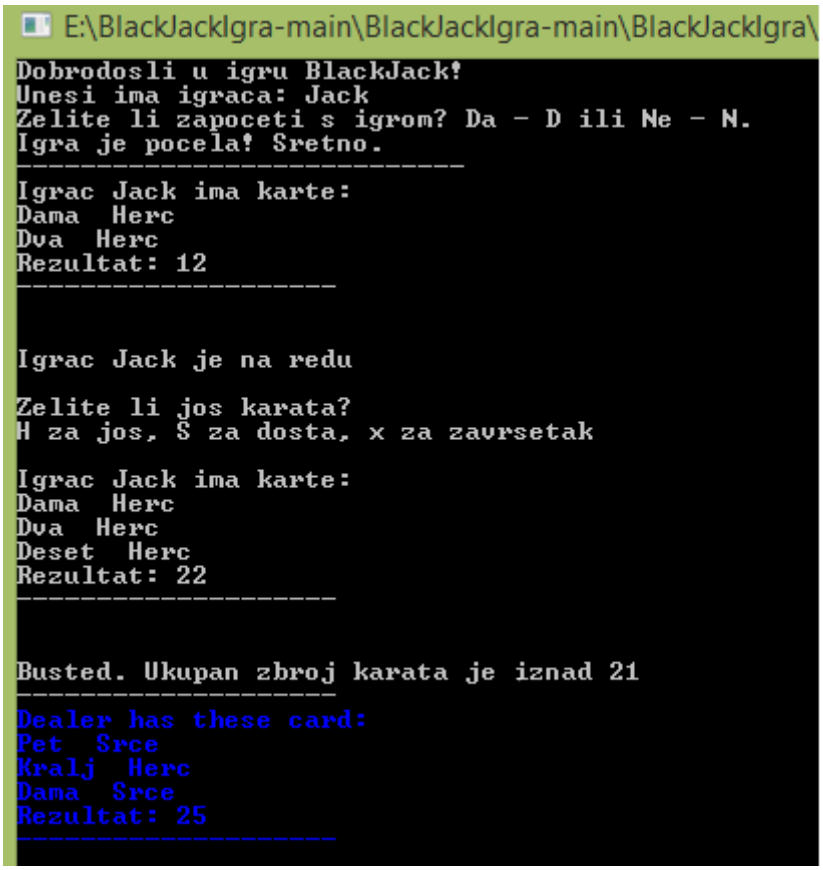
```
internal void Hit(Igrac Igrac)
{
    if (!Igrac.Potez && Igrac.Ime != "Dealer")
        return;

    Igrac.Ruka.Add(spil.Dohvatikartu());

    IzracunajRezultatIgraca(Igrac);

    if (Igrac.Rezultat > 21)
    {
        Igrac.DobitniStatus = DobitniStatus.Busted;
        Igrac.StatusRuke = StatusRuke.Done;
        if (Igrac.Ime != "Dealer")
            MoveActiveSlot();
    }
    else if (Igrac.Rezultat == 21)
    {
        Igrac.DobitniStatus = DobitniStatus.Blackjack;
        Igrac.StatusRuke = StatusRuke.Done;
        if (Igrac.Ime != "Dealer")
            MoveActiveSlot();
    }
}
```

Slika 17.

```
internal void Stay(Igrac Igrac)
{
    Igrac.StatusRuke = StatusRuke.Done;

    if (!Igrac.Potez)
        return;
    MoveActiveSlot();
}
```

Slika 18.

Ako igrač želi još jednu kartu, prvo se ponovno provjerava status ruke, a kako je u ovom slučaju rezultat manji od 21 status ruke je i dalje open, te dealer igraču dodjeljuje još jednu kartu i ponovo računa novi rezultat. Sada korisnično sučelje izgleda kao na slici (Slika 19.).



Slika 19.

U ovom primjeru novi rezultat prelazi 21, te je grač „uhvaćen“ - Busted, odnosno prešao je rezultat 21 i izgubio igru i igra završava. Nakon završetka igre u krisničkom sučelju će se prikazati karte koje je imao dealer.

5.3. Testovi jedinice MSTest

5.3.1. Test Špil

Klasa *SpilTest* sadrži sve testirane metode potrebne za rad sa špilom. Testnom metodom *DohvatiSpil* (Slika 20.) se provjerava da li špil sadrži 52 karte, zatim sadrži testnu metodu *DohvatiKartu* u kojoj se u slučaju da igrač odabere opciju Hit (uzme kartu) vrši testiranje da li se sada u špilu nalazi 51 karta te je li karta izvučena iz špila (Slika 20.). Testna metoda *Promiješaj* koja stvara novi „promiješani“ špil i prema podijeljenim kartama testira broj karata

u špilu. *SpilTest* također sadrži test s atributom [DataRow] koja provjerava dali je Broj karte očekivan, u ovom primjeru očekuje se karta deset dijamant. (Slika 21)

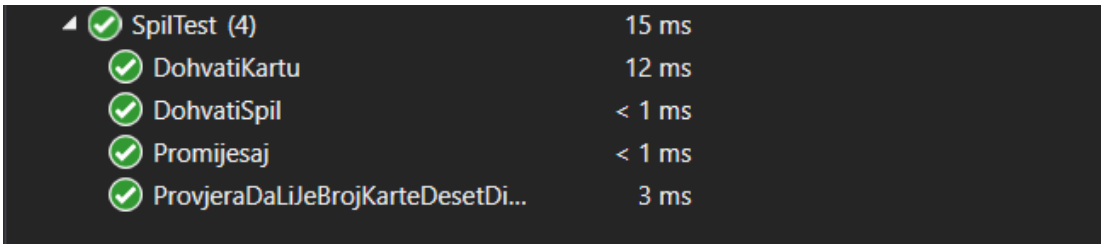
```
[TestClass]
0 references
public class SpilTest
{
    [TestMethod]
    | 0 references
    public void DohvatiSpil()
    {
        Spil spil = new Spil();

        Assert.AreEqual(52, spil.Karte.Count);
    }

    [TestMethod]
    | 0 references
    public void DohvatiKartu()
    {
        Spil spil = new Spil();

        Karta cardTaken = spil.Dohvatikartu();

        Assert.AreEqual(51, spil.Karte.Count);

        Karta missingCard = spil.Karte
            .FirstOrDefault(c => c.Boja == cardTaken.Boja && c.BrojKarte == cardTaken.BrojKarte);

        Assert.IsNull(missingCard);
    }
```

Slika 20.

```
[TestMethod]
[DataRow(10)]
| 0 references
public void ProvjeraDaLiJeBrojKarteDesetDijamant(int brKarte)
{
    Karta karte = new Karta
    {
        Boja = Boja.Dijamant,
        BrojKarte = BrojKarte.Deset
    };

    Assert.AreEqual(brKarte,Convert.ToInt32( karte.BrojKarte));
    Assert.AreEqual( "Dijamant", Boja.Dijamant.ToString());
}
```

Slika 21.

Nakon pokretanja testa *SpilTest* otvara se dijaloški okvir s podacima i statusu testova. Rezultati se nalaze na slici ispod (Slika 22.)



Slika 22. Rezultati testova *SpilTest*

Možemo primijetiti da su svi testovi prošli i možemo vidjeti koliko je vremena bilo potrebno za izvršavanje svakog testa pojedinačno.

5.3.2 Test Igrač

Igrac ima više mogućnosti kada mu dealer podijeli kartu As. Slučaj kada mu dealer podijeli dva As-a odjednom, tada jedan As poprima vrijednost 11, a drugi vrijednost 1, zatim u slučaju kada dealer podjeli jednog As-a, s obzirom na ostatak karata u ruci (nihovih vrijednosti, odnosno rezultat) As poprima određeu vrijednost. Ako rezultat s As-om u vrijednosti prelazi 21 onda As poprima vrijednost 1, dok u suprotnom slučaju, kada igraču dealer podijeli As-a, a rezultat ne prelazi 21, tada As ima vrijednost 11. Prema tome klasa IgracTest testira sve metode s mogućnostima za dodjelu vrijednosti karte As, a to su testne metode: *RezultatIgracaBezAsa (Slika 23.), RezultatIgracaSJednimAsomIznad21, RezultatIgracaSJednimAsomIspod21 (Slika 24.), RezultatIgracaSDvaAsaIspod21*. te metodu za još jednu jednu kartu „Hit“ (Slika 25.)

```
[TestMethod]
0 references
public void RezultatIgracaBezAsa()
{
    Igrac igrac = new Igrac
    {
        Ruka = new List<Karta>
        {
            new Karta {BrojKarte = BrojKarte.Dva},
            new Karta {BrojKarte = BrojKarte.Decko}
        }
    };
    Igra.IzracunajRezultatIgraca(igrac);
    Assert.AreEqual(12, igrac.Rezultat);
}
```

Slika 23.

```
[TestMethod]
0 references
public void RezultatIgracaSJednimAsomIznad21()
{
    Igrac igrac = new Igrac
    {
        Ruka = new List<Karta>
        {
            new Karta {BrojKarte = BrojKarte.Tri},
            new Karta {BrojKarte = BrojKarte.Dama},
            new Karta {BrojKarte = BrojKarte.As}
        }
    };
    Igra.IzracunajRezultatIgraca(igrac);
    Assert.AreEqual(14, igrac.Rezultat);
}
```

Slika 24.

```
[TestMethod]
0 references
public void Hit()
{
    Igra igra = GetGame();
    Igrac igrac = igra.Igraci.First();

    int karte = igrac.Ruka.Count;
    igrac.Hit();

    Assert.AreNotEqual(karte, igrac.Ruka.Count);
}
```

Slika 25.

Nakon pokretanja testa *IgracTest* otvara se dijaloški okvir s podacima i statusu testova. Rezultati se nalaze na slici ispod (Slika 26.)

IgracTest (5)	21 ms
Hit	20 ms
RezultatIgracaBezAsa	< 1 ms
RezultatIgracaSDvaAsaIspod21	< 1 ms
RezultatIgracaSJednimAsomIsp...	1 ms
RezultatIgracaSJednimAsomIzn...	< 1 ms

Slika 26.

5.3.3 Test Igre

Klasa *IgraTest* sadrži metode za provjeru kreiranja nove igre, stvaranje i brisanje igrača, te završetak igre. Metodom *KreirajIgruBlackJack* (Slika 27.) testira se stvaranje nove igre s dva igrača i dijeljenje početnih karata svakom igraču. Test stvara novu instancu Igre te dodaje nove igrače i dijeli početne dvije karte za svakog igrača.

```
[TestMethod]
0 references
public void KreirajIgruBlackJack()
{
    Igra igra = DohvatiIgru();
    Assert.AreEqual(2, igra.Igraci.Count());

    Assert.AreEqual("Dealer", igra.Dealer.Ime);
}

9 references | 9/9 passing
public static Igra DohvatiIgru()
{
    Igra igra = new Igra(new Spil());
    igra.DodajNovogIgraca("Ivan");
    igra.DodajNovogIgraca("Josip");
    igra.Dijeli();
    return igra;
}
```

Slika 27.

Metodom *DijeliDvijeKarteOdDealera (*Slika 28.*)* provjerava se ispravnost metode za dijeljenje početnih karata. Očekivan rezultat su dvije karte za igrača te dvije karte za djelitelja (Dealera).

```
[TestMethod]
0 references
public void DijeliDvijeKarteOdDealera()
{
    Igra igra = DohvatiIgru();

    foreach (Igrac p in igra.Igraci)
    {
        Assert.AreEqual(2, p.Ruka.Count);
    }

    Assert.AreEqual(2, igra.Dealer.Ruka.Count);
}
```

Slika 28.

Metodom *ZavrsiIgru (*Slika 29.*)* provjerava se ispravnost metode za završetak igre. Test stvara novu instancu Igre te dodaje igrače i dijeli početne dvije karte za svakog igrača. Svaki igrač ostaje u igri s opcijom „stand“ u tom slučaju karte se više ne dijele i statusi se promijene u „Done“. U ovom testu se korist tvrdnja AreNotEqual gdje očekivana vrijednost nije jednaka pronađenoj vrijednosti .

```
[TestMethod]
| 0 references
public void ZavrsiIgru()
{
    Igra igra = DohvatiIgru();

    foreach (Igrac igrac in igra.Igraci)
        igrac.OstaniUIgri();
    foreach (Igrac Igrac in igra.Igraci)
    {
        Assert.AreNotEqual(StatusRuke.Open, Igrac.DobitniStatus);
        Assert.AreNotEqual(StatusRuke.Open, Igrac.StatusRuke);
    }

    Assert.AreNotEqual(StatusRuke.Open, igra.Dealer.StatusRuke);
}
```

Slika 29.

Nakon pokretanja klase *IgraTest* otvara se dijaloški okvir s podacima i statusu testova. Rezultati se nalaze na slici ispod (Slika 30.).

IgraTest (12)	18 ms
DalilgracMozeOstatiUlgri	1 ms
DijeliDvijeKarteOdDealera	< 1 ms
DodavanjelgracaslmenomDealer	< 1 ms
DodavanjelgracaSlstimImenom	< 1 ms
DodavanjeViseOdPetlgraca	< 1 ms
KreirajlgruBlackJack	< 1 ms
KreirajVrijednostKarte	2 ms
NemogucnostDijelejnjeKarteUs...	14 ms
Obrisilgraca	1 ms
StvoriNovoglgraca	< 1 ms
UkupanRezultat_BlackJack_21	< 1 ms
Zavrsilgru	< 1 ms

Slika 30.

Zaključak

Početak razvoja programskih proizvoda počinje mnogo prije same implementacije, počinje detaljnom analizom potreba korisnika te specifikacijom ponašanja sustava. Veliki utrošak vremena i truda potroši se na analiziranje potreba korisnika. Osim procesa analize, u razvoju programskog proizvoda su bitni procesi testiranja proizvoda kako bi se osiguralo da sustav radi bez grešaka. Zbog toga je potrebno programski proizvod detaljno testirati kako bi korisniku bio isporučen sustav s minimalnim nedostacima. Ovisnost uspjeha traženog sustava ovisi i o fazi analize, tj. kako bi se isporučio ispravan proizvod potrebno je znati sva njegova ponašanja. Sam proces testiranja sastoji se od provjere rada sustava. Provjeravati, odnosno testirati se može rad cijelog sustava ili rad svakog pojedinačnog segmenta sustava. Osnovna ideja im je ista – pronaći greške, tj. nepravilnosti u sustavu. Procesi testiranja ovisno o načinu rada organizacije će se izvoditi na drugačiji način. Testiranje nije jednostavan zadatak. Tester koji obavlja testiranje treba biti upoznat s detaljima funkcionalne specifikacije, mora dobro razumijeti sustav, te biti sposoban uočiti razlike u ponašanju sustava od očekivanog ponašanja. Uz to tester blisko surađuje s razvojnim programerima te mu je zadatak pripomoći pri izradi kvalitetnog programskog proizvoda. Tester mora imati mogućnost sagledavanja sustava iz korisničkog pogleda, ali i visok stupanj kreativnosti kako bi se pronašli neuobičajeni slučajevi korištenja. Kako bi se olakšao posao testera, poboljšala sigurnost u ispravnost, testni scenariji se mogu automatizirati. Automatizacijom testiranja postiže se značajno smanjenje ručnog testiranja. Kako bi se osigurali da će uvođenjem automatizacije biti promjena potrebno je detaljno i kvalitetno specificirati testne scenarije. Iz osobnog stajališta mogu reći kako je automatizacija svakako korisna u procesu razvoja programskog proizvoda te da pomaže kod obavljanja zamarajućih i ponavljajućih zadataka. Kroz ovaj diplomski rad navela sam na koje sve načine je moguće testirati programske proizvode, zatim koje su moguće prednosti i nedostaci automatizacije testiranja, te na primjeru pokazala izradu automatiziranih jediničnih testova u razvojnom okruženju, Microsoft Visual Studiu Smatram da je testiranje iznimno važan korak pri izradi programskih proizvoda te da ukoliko bi se proces testiranja uklonio, isporučivali bi se proizvodi nezadovoljavajuće kvalitete što bi dovelo do gubitaka.

Literatura

[1] *Von Neumann, J. (1993). First draft of a report on the EDVAC. IEEE Annals of the History of Computing*

[2] https://www.britannica.com/technology/systems-programming

[3] https://www.digitaltrends.com/computing

[4] *Supporting Source Code Difference Analysis* Jonathan I. Maletic, Michael L. Collard Department of Computer Science

[5] *Formal Specification and Verification*, Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean

[6] *Formal Specification and Programming for SDN* relevant ID: draft-shin-sdn-formal-specification-01 Myung-Ki Shin, Ki-Hyuk Nam Miyoung Kang, Jin-Young Cho

[7] https://www.webopedia.com/definitions/application-software

[8] https://www.lifewire.com/what-are-apps-1616114

[9] https://www.enciklopedija.hr/natuknica.aspx?ID=3306

[10] https://www.fda.gov/about-fda/what-we-do

[11] Standard Glossary of Terms used in Software Testing Version 3.2 Foundation 2018 - Release Candidate Terms, https://astqb.org/assets/documents/ISTQB-Glossary-3.2-terms-used-in-FL2018.pdf , str. 27

[12] http://www.sqa.net/iso9126.html

[13] M. Sharma, Software Testing 2020: Preparing for New Roles, Boca Raton 2017., str 37

[14] R., Patton, Software Testing, Sams Publishing, Indiana, 2001

[15] S. Naik, P. Tripathy, Software testing and quality assurance: Theory and practice, New Jersey: John Wiley & Sons, Inc., 2008.str. 40

[16] Software testing revealed, Drugo izdanje, https://www.testinstitute.org/Software_Testing_Books_International_Software_Test_Institute.php

[17] https://www.merriam-webster.com

[18] https://www.guru99.com/static-dynamic-testing.html

[19] Potter, B., & McGraw, G. (2004). Software security testing. IEEE Security & Privacy Magazine, 2(5), 82–83.

[20] https://www.javatpoint.com/black-box-testing

[21] https://www.softwaretestinghelp.com/black-box-testing

[22] González, C. A., & Cabot, J. (2012). ATLTest: A White-Box Test Generation Approach for ATL Transformations. Lecture Notes in Computer Science

[23] https://www.javatpoint.com/white-box-testing

[24] A. Spillner, T. Linz i H.Schafer, Software testing foundations, Santa Barbara: Rocky Nook Inc., 2014., str. 35

[25] C. Tozzi, „Quality Assurance and Software Testing: A Brief History“, Available: https://saucelabs.com/blog/quality-assurance-and-softwaretesting-a-brief-history

[26] https://www.softwaretestinghelp.com/how-to-test-application-security-web-and-desktop-application-security-testing-techniques

[27] „Bringing Portability to the Software Process“ James D., Mooney West Virginia University, Dept. of Statistics and Computer Science

[28] https://www.experienceux.co.uk/faqs/what-is-usability-testing

[29] A. Spillner, T. Linz i H.Schafer, Software testing foundations, Santa Barbara: Rocky Nook Inc., 2014. https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

[30] D., Graham, E., van Veenendaal, I., Evans, R., Black, Foundations of software testing, ISTQB certification, International Thomson Business Press, London, 2006.

[31] International Journal of Information Technology and Business Management WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC S.Balaji Computer Science Dept., Gulf College Muscat, Sultanate of Oman. Dr.M.Sundararajan Murugaiyan Computer Science Dept., Government Arts College Chennai, TN, India. Str 27.

[32] International Journal of Information Technology and Business Management WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC S.Balaji Computer Science Dept., Gulf College Muscat, Sultanate of Oman. Dr.M.Sundararajan Murugaiyan Computer Science Dept., Government Arts College Chennai, TN, India. Str 28.

[33] Myers, G. J., Badgett, T., & Sandler, C. (Eds.). (2012). The Art of Software Testing. doi:10.1002/9781119202486, str 85

[34] https://www.softwaretestinghelp.com/what-is-component-testing-or-module-testing

[35] https://www.softwaretestinghelp.com/what-is-integration-testing

[36] A. Spillner, T. Linz i H.Schafer, Software testing foundations, Santa Barbara: Rocky Nook Inc., 2014., str. 65

[37] D., Graham, E., van Veenendaal, I., Evans, R., Black, Foundations of software testing, ISTQB certification, International Thomson Business Press, London, 2006.

[38] G., Myers, T., Badgett, C., Sandler, The art of software testing, Treće izdanje, John Wiley & Sons, Inc., Kanada, 2011.

[39] AUTOMATION TESTING Tutorial: What is, Process, Benefits & Tools. https://www.guru99.com/automation-testing.html

[40] P. Ammann, J. Offutt, Introduction to software testing, Cambridgeshire: Cambridge University Press, 2017. str. 69.

[41] M. Fewster, D. Graham, Software Test Automation: Effective use of test execution tools. London: ACM Press Books, 1994. str 22.

[42] P. Ammann, J. Offutt, Introduction to software testing, Cambridgeshire: Cambridge University Press, 2017. str 70.

[43] M. Sharma, Software Testing 2020: Preparing for New Roles, Boca Raton: CRC Press, 2017. str 113.

[44] M. Fewster, D. Graham, Software Test Automation: Effective use of test execution tools. London: ACM Press Books, 1994 str. 27

[45] https://docs.microsoft.com/hr-hr/visualstudio/test/unit-test-basics?view=vs-2019

[46] http://csharp.net-informations.com

[47] An Exploratory Study of Component Reliability Using Unit Testing R. Torkar, S. Mankefors, K. Hansson and A. Jonsson Dept. of Informatics and Mathematics University of Trollhättan/Uddevalla P.O.Box 957, SE-46129 Trollhättan, Sweden, str 2

[48] https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2019

[49] A. Manasyan, »7 Ways Learning Will Improve Your Software Testing,« Macadamian, https://www.macadamian.com/learn/7-ways-learning-will-improve-software-testing

[50] https://www.lambdatest.com/most-complete-mstest-framework-tutorial-using-net-core-2

[51] https://www.lambdatest.com/nunit-testing-tutorial-for-selenium-csharp

[52] https://www.lambdatest.com/xunit-testing-tutorial

[53] Comparison of selected tools to perform unit tests Piotr Strzelecki*, Maria Skublewska - Paszkowska Institute of Computer Science, Lublin University of Technology

Sažetak

Automatizacija testiranja je proces pisanja programskog koda koji samostalno pokreće zadane testne scenarije te provjerava ispravnost sustava. U uvodnom dijelu diplomskog rada definirani su pojmovi vezani uz temu, a to su računalni program, korektnost programa, vrste specifikacije programa i vrste pogrešaka. Ovaj diplomski rad uvodi čitatelja u teoriju testiranju sustava, te prikazuje značaj testiranja pri razvoju aplikacija. Navedene su prednosti i nedostatci pojedinih načina testiranja. Opisana je automatizacija testnih procesa, te način automatiziranja koristeći C# ugrađeni *unittest* okvir MSTest. Navedene su značajke automatizacije te alati za automatiziranje. Na temelju stečenih znanja u praktičnom je dijelu rada prikazan primjer automatiziranja testnih scenarija te je napravljena integracija jediničnih testova za oglednu aplikaciju Blackjack.

Summary

Testing automation is the process of writing program code that will independently run default test scenarios and check the correctness of the system. In the introductory part of the diploma thesis, the terms related to the topic are defined, namely computer program, correctness of the program, types of program specifications, types of errors. Then this thesis introduces the reader to the theory of system testing, and shows the importance of testing for application development. The advantages and disadvantages of the test method are listed. The automation of test processes is described, as well as the method of automation using the C # built-in *unittest* framework MSTest. Automation features and automation tools are listed. With the help of the acquired knowledge in the practical part of the paper, an example of automation of test scenarios is presented, the integration of unit tests for the Blackjack application is made.