UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Kuzem

# NAČRTOVANJE TESTIRANJA PRI RAZVOJU IS V MANJŠIH RAZVOJNIH SKUPINAH

DIPLOMSKO DELO NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

MENTOR: vis. pred. dr. Damjan Vavpotič

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00037/2010
Datum: 05.10.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK KUZEM**

Naslov: **NAČRTOVANJE TESTIRANJA PRI RAZVOJU IS V MANJŠIH RAZVOJNIH SKUPINAH**

**TEST PLANNING FOR IS DEVELOPMENT IN SMALL DEVELOPMENT TEAMS**

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

V okviru diplomske naloge proučite možnosti za uporabo sodobnih pristopov in orodij za potrebe testiranja pri razvoju IS v manjših razvojnih skupinah. Predstavite omenjene pristope in orodja ter prednosti in slabosti, ki jih prinaša uporaba le teh. Izdelajte tudi predlog postopka testiranja, ki bi manjšim razvojnim skupinam omogočil čim bolj učinkovito izrabo omenjenih pristopov in orodij, pri čemer upoštevajte specifične potrebe takšnih skupin.

Mentor:

viš. pred. dr. Damjan Vavpotič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

I Z J A V A  O  A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Rok Kuzem,

z vpisno številko 63040084,

sem avtor/-ica diplomskega dela z naslovom:

NAČRTOVANJE TESTIRANJA PRI RAZVOJU IS V MANJŠIH RAZVOJNIH SKUPINAH

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  vis. pred. dr. Damjan Vavpotič

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice:

_____

# Zahvala

Prisrčno se zahvaljujem:

Mentorju vis. pred. dr. Damjanu Vavpotiču, ki me je med izdelavo diplomske naloge vseskozi usmerjal in mi pomagal z nasveti ter pripombami.

Podjetju Gama System, d. o. o., kjer sem opravljal obvezno prakso ter se seznanil s področjem, o katerem pišem v diplomski nalogi.

Nini Leskovšek za pomoč pri prevajanju.

Lektorici Alenki Cizel, za lektoriranje.

Staršem za finančno in moralno podporo, ki so mi jo nudili tekom celotnega študija, ter ostalim ožjim družinskim članom.

Punci Lei Tuhtar, ki me je spodbujala pri pisanju, da je zadeva napredovala hitreje kot bi drugače.

# KAZALO VSEBINE

## KAZALO SLIK

# SEZNAM KRATIC

RFT —Rational Funcitonal Tester, to je orodje za testiranje

HTML — Hypertext Markup Language, osnovni jezik za pisanje spletnih strani

XML — Extensible Markup Language, soroden HTML-ju

SOAP — Simple Object Acess Protocol, je protokol, ki omogoča komunikacijo med ponudnikom storitev in odjemalcem

HTTP — Hypertext Transfer Protocol, je glavna metoda za prenos informacij na spletu

HTTPS — Hypertext Transfer Protocol Secure, je HTTP s kriptirano obliko komukacije

RUP — Rational Unified Process, je univerzalni proces, ki določa, kdo dela kaj, kdaj in kako za doseganje določenega cilja

# POVZETEK

Diplomske naloge sem se lotil povsem raziskovalno. Najprej sem preučil temo, ki mi jo je predlagal mentor na podlagi mojih preteklih izkušenj, nato pa sestavil osnovno zgradbo, na kateri sem gradil delo. Stvari, ki sem se jih naučil v teoriji, sem poskušal čim bolje sestaviti v eno zaključeno celoto, tako da bi moje delo bilo lahko uporabno v praktične namene. Pri tem sem uporabil tudi izkušnje, ki sem jih pridobil na področju testiranja v razvojnem podjetju, kjer sem opravljal obvezno prakso. V sklopu raziskovanja sem se seznanil z različnimi programskimi paketi in orodji, ki se uporabljajo na tem področju.

Rdeča nit diplomskega dela je proces testiranja programske opreme. Za kvalitetno načrtovanje je potrebno vedeti, kaj vse sploh testiranje obsega. Namen diplomskega dela je seznaniti bralca z vrstami, metodami in stopnjami testiranja. Predstaviti mu nekaj orodij, ki se uporabljajo za samo načrtovanje testiranja, ter nekaj orodij za izvrševanje testiranja. Za lažje razumevanje samega procesa testiranja ter za primerjavo bom podal že izdelan proces testiranja nekega podjetja. Predstavil bom tudi alternative testiranju, ki se pojavljajo v zadnjem času in poskušal odgovoriti na vprašanje, ki se pojavi vedno, ko načrtujemo testiranje tj. kdaj končati testiranje. Zaključek moje naloge je sestavljen iz nasvetov in usmeritev kako oblikovati proces testiranja pri nekem projektu, ter izdelati lasten proces testiranja. Rezultat diplomske naloge je predlog procesa testiranja za manjša podjetja, ki temelji na literaturi ter pridobljenih lastnih izkušnjah pri delu v takšnem podjetju.

## SUMMARY

The BA thesis is entirely based on research. I first studied the theme that my mentor suggested on the basis of my past experience. Further on, I formed the basic structure I worked further on. Everything I had learnt in theory, I tried to put together in one complete unit in the best way possible, so that my work could be useful in practice. I used my experience I had got while doing my practical work in a development company where I had worked in the field of testing. While researching, I was introduced to different software packages and tools that are used in this field.

A thread running through my BA thesis is the testing of software equipment. For the quality planning we need to know what the testing consists of. The purpose of the BA thesis is to introduce a reader to the sorts, methods and levels of testing, to introduce a reader to some tools that are used in the planning of testing and some tools for carrying out the testing. For better understanding of the process of testing, I will present a finished process of testing of a company in a way of comparison. I will present the alternatives to testing which are recently in use, and I will try to answer the question which appears every time we plan the testing that is when to terminate the testing. The conclusion of my BA thesis consists of advice and orientations of how to work on a project based on the process of testing and how to make your own process of testing. The result of the BA thesis is a suggestion of a process of testing for small companies. This process is based on the literature and my experience acquired while working in a company previously mentioned.

# 1. UVOD

Za začetek bi rad na kratko pojasnil, kaj testiranje programske opreme pomeni in kaj vključuje. Poznamo več definicij, kaj sploh testiranje programske opreme je. Ena izmed njih pravi, da je testiranje programske opreme v bistvu poganjanje programa z namenom odkrivanja napak v programski opremi[8]. Gre za sistematično odkrivanje različnih razredov napak, v vseh razvojnih fazah, v minimalnem času in z malo napora. Dober test je takšen test, ki ima visoko verjetnost odkrivanja še neodkritih napak. Uspešen test pa je tisti, ki dejansko odkrije do sedaj neodkrite napake. Prednost testiranja je dokaz, da programska oprema deluje, tako kot je zapisano v specifikacijah. Pomebno je tudi, da se zavedamo, da testiranje ne more pokazati odsotnost napak, temveč prisotnost le-teh. Če med testiranjem ne naletimo na napake, to še ne pomeni, da jih ni. Je pa seveda dober znak, če je po večkratnem testiranju vedno manj najdenih napak.

Testiranje, odvisno od uporabljenih metod, se lahko uporabi kadarkoli v razvojnem procesu. Vendar pa se še vedno največji odstotek nameni testiranju programov, ko so znane zahteve, ter je proces kodiranja zaključen. Tu gre za zelo ohlapno definicijo, ki opisuje predvsem namen testiranja v končni fazi razvoja programske opreme, ko že imamo delujoč produkt. Boljša definicija, ki zajema celoten postopek od začetka do konca razvoja progamske opreme, pa je takšna: »To je niz dejavnosti, ki se izvajajo z namenom odkrivanja napak v programski opremi.«

V začetku razvoja prve programske opreme se je izvajalo sprotno testiranje in odpravljanje napak, kasneje je ločitev teh dveh postopkov vpeljal Glenfor J. Myers, leta 1979. Njegova zamisel je sovpadala z željo skupnosti po razvoju programske opreme, ki je bila, da se ločijo osnovne razvojne aktivnosti, kot sta odpravljanje napak in testiranje. Usmerjenost razvoja in testiranja programske opreme se je skozi zgodovino dosti spreminjala. Dave Gelperin in William C. Hetzel sta leta 1988 klasificirala cilje, ki jih je imelo testiranje skozi različna obdobja po naslednjih fazah: do 1956 »(Debugging oriented) Faza razhroščevanja programske opreme skupaj s strojno«, 1957—1978 »(Demonstration oriented) Faza zaznavanja, lociranja, identificiranja in popravljanja napak«, 1979—1982 »(Destruction oriented) Faza poganjanja programske opreme z namenom odkrivanja napak«, 1983—1987 »(Evolution oriented) Faza odkrivanja napak skozi evolucijski življenski cikel programske opreme«, 1988—2000 »(Preventation oriented) Faza preprečevanja nastanka napak pri razvoju programske opreme«.[5, 15]

V tej diplomski nalogi sem se odločil predstaviti osnovne koncepte ter metode testiranja, ki bodo služili k lažjemu razumevanju samega bistva naloge. Predstavil bom stopnje (vrste testiranja), ki sestavljajo proces testiranja. Namen testiranja programske opreme je zagotavljanje kvalitetnih ter varnih programskih rešitev za podjetja in fizične osebe.

V mislih imam testiranje programskih rešitev že med samim razvojem (posameznih modulov ter sklopov programske kode) in potem celotne rešitve, preden se le-ta odda naročniku v

uporabo. Poiskal in predstavil bom nekaj orodij (programov, ki služijo testiranju), ki se lahko uporabijo med samim postopkom testiranja. Orodja se razlikujejo po namembnosti, zato jih bom razdelil po tem, v kateri stopnji testiranja se katero uporablja. Orodja služijo samo kot primer, njihova uporaba je odvisna od odločitve testerja, kajti na tržišču se vsak dan pojavi kakšno novo.

Temo za diplomsko nalogo sva določila skupaj z mentorjem, ki mi je na podlagi izkušenj predlagal primeren naslov. Nekaj izkušenj na področju testiranja programske opreme sem dobil v podjetju, kjer sem opravljal prakso ter kasneje ostal še nekaj mesecev. Opazil sem, da se testiranja tam lotevajo dokaj neorganizirano, brez kakšnega plana testiranja. Začetno testiranje opravlja kar razvijalec kode sam, nato sledi nekaj testov enot (ang. unit tests), potem pa se dejanskega testiranja lotijo, ko so moduli že sestavljeni v celoto. Težavo vidim predvsem v testiranju sestavljenih sklopov, ker nimajo razdelanega načrta, ki bi ločeval to testiranje po stopnjah. Začel sem razmišljati, kako bi se bolj sistematično lotil testiranja v tej fazi. Predlog: postopoma stestirati različne problemske vidike (funkcionalnost, zmogljivost, varnosti itd). Moj cilj je napisati nalogo, ki bo lahko služila kot ogrodje oziroma načrt, ki ga bo možno uporabiti pri razvoju programske opreme ter z njim pokriti celotno poglavje testiranja.

# 2. VRSTE IN NAMEN TESTIRANJA PROGRAMSKE OPREME

## 2.1 NAMEN

Osnovni namen testiranja je iskanje napak v programski opremi, tako da se pomanjkljivosti odkrijejo in odpravijo. Področje testiranja programske opreme pogosto vključuje tako pregled kode kot tudi izvajanje te kode v različnih okoljih in pogojih. Preveriti je treba da koda počne kar bi naj počela. Pogosto se zgodi, da je ekipa, ki se ukvarja s testiranjem, ločena od razvojne ekipe. Člani ekipe, zadolžene za testiranje, imajo med seboj različne vloge. Informacije, ki jih priskrbi testna ekipa, se lahko uporabijo za popravljanje postopka, po katerem se razvija programska oprema.

Zagotavljanje združljivosti programske opreme z drugimi aplikacijami ter operacijskimi sistemi prav tako spada med namene testiranja. Pomembna je tudi združljivost novejše programske različice z njeno predhodnico. Recimo, imamo nek urejevalnik besedila, ki z novo različico dobi novo obliko shranjevanja besedila. Pomembno pri tem je, da omogočamo v novi različici programa, tudi urejanje besedila, ki je bilo napisano v starejši različici. Seveda je ta združljivost s starejšimi programi smislena le do določene mere.

Zgodnje odkrivanje napak lahko zelo vpliva na same stroške razvoja. Znano je namreč, da prej kot so napake odkrite, cenejše je njihovo odpravljanje. Spodaj se nahaja Tabela 1[5], ki nam kaže stroške odpravljanja napak v odvisnosti od faze razvoja, v kateri je bila napaka odkrita.

| | | Faza v kateri je bila zaznana napaka | | | | |
|---|---|---|---|---|---|---|
| | | Analiza | Načrtovanje | Razvoj | Sistemsko testiranje | Po izdaji izdelka |
| Faza v kateri je do napake prišlo | Analiza | 1 x | 3 x | 5—10 x | 10 x | 10—100 x |
| | Načrtovanje | - | 1 x | 10 x | 15 x | 25—100 x |
| | Razvoj | - | - | 1 x | 10 x | 10—25 x |

**Tabela 1: Povečanje stroškov odprave napak pri razvoju programske opreme v odvisnosti od faze, v kateri je do napake prišlo in od faze, v kateri je bila napaka odkrita**

Pogosto se skupaj s pojmom testiranje programske opreme pojavljata še pojma verifikacija in validacija programske opreme (ang. Software verification and validation). Verifikacija je postopek ocenjevanja sistema ali komponente, ki določi, ali produkt v določeni fazi razvoja ustreza pogojem, določenim na začetku te faze. Validacija je proces vrednotenja sistema ali komponente med razvojem ali na koncu razvojnega procesa, ki pove, ali le-ta izpolnjuje določene zahteve

V povezavi s testiranjem programske opreme pogosto naletimo tudi na pojem zagotavljanje kvalitete programske opreme. Čeprav sta to dve področji, ki ju ločujemo med seboj, se pogosto dogaja, da v podjetjih, kjer razvijajo programsko opremo, en oddelek skrbi za oboje. Zagotavljanje kakovosti programske opreme se od testiranja loči po tem, da je njegova naloga preprečevanje napak. Se pravi, skrbi za to, da se napake v prvi vrsti ne pojavljajo. Medtem ko se pri testiranju napake iščejo.

## 2.2 PRIMERJAVA FUNKCIONALNEGA IN NEFUNKCIONALNEGA TESTIRANJA

Namen funkcionalnega testiranja je preverjanje točno določenih funkcij, ki jih koda mora opravljati. Te funkcije so po navadi zapisane v dokumentaciji oziroma specifikaciji projekta. Nekatere razvojne metodologije pa temeljijo na primerih uporabe in izkušnjah uporabnika. Funkcionalni preizkusi poskusijo odgovoriti, ali lahko uporabnik stori tisto, čemur naj bi program bil namenjen, ali določene programske funkcije dejansko delujejo.

Nefunkcionalno testiranje se osredotoča na tisti del programske opreme, ki ni nujno povezan z določeno funkcionalnostjo, ki je namenjena uporabniku. Primerni temi za nefunkcionalno testiranje sta recimo varnost in prilagodljivost programa. Recimo pri informacijskih sistemih, ki povezujejo veliko število računalnikov in ostalih elektronskih naprav ter po njih potujejo pomembni podatki, morajo imeti dobro spisano kodo, ki omogoča pravilno delovanje sistema tudi ob večjih obremenitvah (veliko ljudi naenkrat prijavljenih v sistem).

## 2.3 PRIMERJAVA STATIČNEGA IN DINAMIČNEGA TESTIRANJA

Obstaja veliko pristopov k testiranju programske opreme. Statično testiranje preverja predvsem čistost kode, algoritmov ali dokumentacije. V prvi vrsti pomeni sintaktično preverjanje kode in/ali ročno pregledovanje kode ali dokumentov, da bi odkrili napake. Zanj

je značilno, da se programska oprema nad katero izvajamo to testiranje, ne poganja. Takšno vrsto testiranja lahko izvaja razvijalec, ki je napisal kodo, sam. V praksi se pogosto dogaja, da se statični preizkusi izpustijo, kar pa ni ravno dobra odločitev, kajti napake in hrošče, ki so odkriti v tej stopnji razvoja, je ceneje odpraviti zdaj, kot pa kasneje v razvojnem ciklu.

Pri dinamičnem testiranju preučujemo fizični odziv sistema na spremenljivke, ki niso konstantne in se spreminjajo s časom. Gre za dejansko prevajanje in poganjanje programske kode s podanimi testnimi primeri. Dinamično testiranje nastopi, ko se programska oprema prvič uporabi, tu se po navadi začne faza testiranja. Lahko se izvaja preden je program dokončan, z namenom testiranja določenih delov kode (modulov ali funkcij). Pri tem uporabimo nadomestke za manjkajoče komponente, ki jih testirana funkcija kliče. Ali pa izvajamo kodo v razhroščevalnem okolju.

# 3. METODE TESTIRANJA

Metoda testiranja je postopek, ki pripelje do rezultata testiranja. Običajno se delijo metode testiranja na pristop testiranja bele in črne škatle (ang. white- and black-box testing).

## 3.1 METODA BELE ŠKATLE ALI STRUKTURNO TESTIRANJE

Metoda bele škatle je testiranje programske opreme, ki potrebuje dostop do notranje strukture in algoritmov, vključno s kodo, ki le-te implementira. Vse to se uporablja za načrtovanje testnih primerov. Tester izbere vhodne podatke za preverjanje delovanja kode ter določi primerne izhodne podatke. To je podobno testiranju vozlišč v krožnem diagramu.

Testne primere naredimo na osnovi strukture programa. Pri identifikaciji dodatnih testnih primerov uporabimo naše poznavanje programa. Preverjamo posamezne stavke, na primer z izbiro operatorjev v izrazih. Preverjamo lahko tudi zanke, tako da:

- določimo pogoje tako, da se zanke ne izvede (izjema so REPEAT zanke)
- izvedemo zanko natančno enkrat
- izvedemo zanko več kot enkrat

Končno lahko preverjamo programske poti tako, da zagotovimo izvedbo vseh poti v programu. Pri tem moramo preveriti vejitve tako, da zagotovimo možnost izhoda iz pogoja testiranj vsaj enkrat.

## 3.2 METODA ČRNE ŠKATLE ALI FUNKCIONALNO TESTIRANJE

Služi za testiranje interne strukture oz. logike podsistema ali objekta. Ta metoda ravna s programsko opremo kot s črno škatlo, ne pozna notranje strukture sistema. Pod to metodo spada testiranje na podlagi specifikacije. Namen takega testiranja je ugotoviti ustreznost

programske opreme glede na zahteve zapisane v specifikaciji aplikacije. Usmerjeni smo v obnašanje vhodov/izhodov. Če se vsak podan vhod izhod ujema s predvidenim, potem enota opravi test. V večini primerov je nemogoče preizkusiti vse možne vhode (testne primere). Zato želimo zmanjšati število testnih primerov z ekvivalenčnim particioniranjem. Vhodne pogoje razdelimo v ekvivalenčne razrede. Testne primere izberemo iz vsakega ekvivalenčnega razreda. To metodo uporabljamo v kasnejših fazah testiranja za preverjanje funkcionalnih zahtev.

Takšna metoda ima svoje prednosti in slabosti. Ker testerji ne poznajo programske kode, se osredotočijo predvsem na to, da napake zelo verjetno obstajajo. Na preizkušnji so vsi možni primeri, ki si jih izmislijo, in na tak način odkrijejo napake, ki jih programerji ne. Slaba stran tega pa je, da včasih nastane več različnih testov, za en in isti primer, medtem ko nekatere zadeve ostanejo netestirane.

### 3.3 METODA SIVE ŠKATLE

To je metoda, ki se uveljavlja v zadnjem času kot nekakšen preplet metod bele in črne škatle. Temelji na poznavanju programske kode za namen izdelave testnih primerov. Upravljanje z vhodnimi podatki in formatiranje izhodnih ne spada pod to metodo. To razlikovanje je predvsem pomembno pri integracijskih testih dveh modulov, napisanih s strani dveh različnih razvijalcev, kjer se testiranje izvaja samo nad vmesniki. V to metodo spada tudi spreminjanje odlagališča podatkov, ker uporabniku po navadi ni omogočeno spreminjanje podatkov zunaj sistema, ki ga testira. Metoda sive škatle lahko vključuje tudi obratno inženirstvo (ang. Reverse engineering) za določanje mejnih vrednosti ali sporočil z napakami.

## 4. STOPNJE TESTIRANJA

Stopnje testiranja nam povedo, v kateri fazi razvoja programske opreme se uporabi kateri izmed testov oziroma kakšna je specifičnost določenega testa.

### 4.1 TESTIRANJE PROGRAMSKIH ENOT (ang. Unit testing)

Že samo ime pove, da se pri tem testiranju osredotočimo na določeno enoto oziroma določen del programske kode. Pravimo mu tudi testiranje komponent. V objektno usmerjenem okolju je to po navadi na nivoju razreda. Minimalni testi enot pa zajemajo konstruktorje in destruktorje. Takšni testi so po navadi napisani s strani razvijalcev, ko napišejo kodo. Z njimi poskušajo zagotoviti pravilno delovanje posameznih funkcij. Testiranje enot ne more zagotoviti pravilnega delovanja celotne programske opreme, temveč lahko le ugotovi, ali posamezni gradniki programske opreme delujejo neodvisno vsak zase.

## 4.2 INTEGRACIJSKO TESTIRANJE

Integracijsko testiranje je faza, ki sledi testiranju enot. Tu gre za združevanje posameznih programskih modulov (enot) v večje skupine, nad katerimi potem izvajamo testiranje. Kot rezultat dobimo integriran sistem, ki je pripravljen za sistemsko testiranje. Namen integracijskega testiranja je preveriti funkcionalnost, zmogljivost in zanesljivost glavnih gradnikov programske rešitve. Programske komponente so lahko integrirane iterativno ter se tako tudi izvaja testiranje (postopoma), lahko pa so integrirane vse hkrati in se nato nad njimi izvaja testiranje. Temu slednjemu pravimo "Veliki pok" (ang. Big Bang). Poznamo še nekaj načinov integracijskega testiranja. Eden izmed njih je "Od zgoraj navzdol" (ang. Top-Down). To je pristop, kjer začnemo s testiranjem modulov, ki so bili zadnji integrirani, ter gremo korak za korakom po njihovih vejah navzdol, vse do najnižjih modulov. Prednost pri takšnem pristopu je lažje odkrivanje manjkajočih vej. Drugi pristop je ravno obraten, imenovan "Od spodaj navzgor" (ang. Bottom-Up). Tu začnemo s testiranjem komponent na najnižjem nivoju, ki jih nato uporabimo za lažje testiranje komponent na višjem invoju. Proces testiranja se ponavlja dokler niso vse komponente preizkušene. Prednost takšnega testiranja je v lažjem odkrivanju hroščev (ang. Bugs). Pristop, ki združuje oba prej omenjena, pa se imenuje "Testiranje sendviča" (ang. Sandwich Testing).

## 4.3 SISTEMSKO TESTIRANJE

Sistemsko testiranje programske opreme je osredotočeno na celoten, integriran sistem (delujoča programska rešitev). Njegov namen je preveriti skladnost sistema z zahtevami, določenimi v specifikaciji oziroma dokumentaciji. V ta sklop testiranja spada prej opisana metoda črne škatle. Sistemsko testiranje je preiskovalna faza, katere fokus je ne samo testiranje zgradbe, temveč tudi obnašanje programa ter pričakovanje strank. Namen takšnega testiranja je tudi preizkusiti, kako se sistem obnaša, če uporabnikove zahteve presežejo zahteve, določene v specifikaciji.

## 4.4 SISTEMSKO INTEGRACIJSKO TESTIRANJE

To je proces testiranja programske opreme z ostalo programsko opremo, ki se nahaja na isti strojni opremi. Sistemsko integracijsko testiranje vzame pod drobnogled več integriranih sistemov, ki so prestali sistemsko testiranje, ter preveri, kako delujejo zahtevane interakcije med temi sistemi. Tej stopnji običajno sledi testiranje sprejemljivosti.

## 4.5 REGRESIJSKO TESTIRANJE

Kadar se programska koda dosti spreminja, se uporabi regresijsko testiranje. Namen takšnega testiranja je ugotavljanje, ali so se s spremembami pojavile tudi nove napake. Pogosto pa se išče tudi napake, ki so bile nekoč že odpravljene, vendar pa bi se lahko zaradi spremenjene kode zopet pojavile. Zato metode regresijskega testiranja pogosto vključujejo izvajanje že uporabljenih testov. Globina testiranja je odvisna od razvojne faze, v kateri se sprememba

pojavi ter od rizičnosti dodanih funkcij. Če so te spremembe v kodi storjene na pozni izdaji in dodane funkcije predstavljajo velik riziko, potem mora biti testiranje celovito. V primeru sprememb v zgodnji fazi razvoja pa je lahko testiranje bolj površinsko in temelji na pozitivnih testih posameznih funkcij.

Regresijsko testiranje je tudi sestavni del razvoja programske opreme, ki se odvija po metodi ekstremnega programiranja. V tej metodi je projektna dokumentacija zamenjana z obsežnim, ponavljajočim in avtomatiziranim testiranjem celotnega programskega paketa, v vsaki fazi življenjskega cikla programske opreme.

## 4.6 TESTIRANJE SPREJEMLJIVOSTI

To je zadnji test pred predajo programske opreme v uporabo naročniku oziroma na prodajne police. Programska oprema, ki prestane ta test, naj bi ustrezala vsem ali pa skoraj vsem kriterijem in zahtevam, ki jih je postavil naročnik programa, ali pa, če je namenjena za splošno prodajo, vsem zahtevam in kriterijem, ki so jih zastavili načrtovalci programa. Pri testiranju sprejemljivosti je pomembno, da se pogoji za testiranje čimbolj ujemajo z realnim okoljem, kjer se bo program uporabljal. Vsak testni primer, ki se izvede, mora čimbolj simulirati dejanske situacije, ki se bodo odvijale pri uporabi programa v praksi. Testiranje sprejemljivosti najpogosteje vključuje poganjanje serije testov nad zaključenim sistemom. Pomembno je tudi testiranje ekstremnih situacij, do katerih lahko pride v praksi. Pri vsakem testnem primeru je pomembno, da so natančno opisane aktivnosti, ki se izvajajo, ter tudi rezultati, ki jih pričakujemo.

Na podlagi podatkov, ki jih je priskrbel naročnik programa, se ustvari testno vhodne podatke ter pričakovane rezultate. Teste se nato izvaja s pripravljenimi vhodnimi podatki, nato pa dobljene rezultate testov primerjamo s pričakovanimi. Če se stvari ujemajo, pravimo, da je programska oprema uspešno prestala izvajani test. V primeru, da se rezultati ne ujemajo, se gre še enkrat skozi vhodne podatke ter se ugotovi ali so pričakovani rezultati realni ali ne. Lahko smo se ušteli pri predvidevanju rezultatov ali pa je test dejansko neuspešen. Cilj je zagotoviti sistem, ki izpolnjuje poslovne zahteve naročnikov in uporabnikov.

## 4.7 ALFA TESTIRANJE

Alfa testi so simulirani ali dejanski operativni testi, izvajani s strani potencialnih uporabnikov ali pa neodvisne testne skupine na strani razvijalcev. To so interni testi sprejemljivosti; če jih programska oprema uspešno prestane, gre v fazo beta testiranja.

## 4.8 BETA TESTIRANJE

To testiranje sledi alfa testiranju. Izdaje programske opreme pod oznako *beta* so izdane omejenemu številu uporabnikov zunaj razvojne ekipe. Te skupine uporabnikov, ki prejmejo

beta različico, potem poročajo nazaj o hroščih, na katere so naleteli, kar omogoča razvojni ekipi hitrejše odpravljanje le-teh. Včasih so beta različice programov na voljo tudi širši javnosti, da je povratnih informacij za izboljšanje programske opreme čim več.

# 5. TESTIRANJE NEFUNKCIONALNIH LASTNOSTI PROGRAMSKE OPREME

V prejšnjih poglavjih sem se osredotočil predvsem na testiranje programske opreme z namenom odkrivanja napak. Tu pa bom opisal še nekaj vrst testiranja, katerih cilj ni odkrivanje samih napak v strukturi oziroma kodi programa ter preverjanje njegovih funkcij. Ta testiranja ciljajo na zmogljivost, stabilnost, varnost in uporabnost programske opreme.

## 5.1 TESTIRANJE ZMOGLJIVOSTI PROGRAMSKE OPREME

Ko omenimo zmogljivost programske opreme, imamo v mislih predvsem to, kako se obnese pri uporabi v realnem okolju. Večina sistemov je realnočasovnih, kar pomeni, da je takojšnja odzivnost velikega pomena. Če uporabnik začne izvajati neko operacijo v sistemu, pričakuje, da se bo sistem odzval ter opravil nalogo v realnem času. Zato je kritičnega pomena sprotno izvajanje operacij in nalog, ki jih sistem dobi od uporabnikov. Pri testiranju zmogljivosti lahko tudi preverjamo različne kvalitativne atribute, kot sta skalabilnost, zanesljivost, ter porabo virov (izkoriščanje strojne opreme). Osnovni test zmogljivosti se imenuje ***obremenitveni test***. Ta preverja, kako se sistem obnese pri največjem pričakovanem številu uporabnikov, ki bodo naenkrat uporabljali sistem. Vrne nam odzivne čase vseh pomembnih poslovnih kritičnih transkacij. Če v takem testu spremljamo tudi podatkovno bazo, aplikacijski strežnik in podobno, nam ta test lahko pokaže vse težave pri uporabi programske opreme. Drugi test, ki spada pod testiranje zmogljivosti, je ***stresni test***. Že samo ime pove, da tukaj preizkušamo obnašanje programske opreme v stresnih situacijah, kot so različne preobremenitve sistema s strani uporabnikov. Na podlagi tega testa se ugotovi, če se bo sistem učinkovito odzival, tudi če število uporabnikov ali pa število izvajanih operacij na sistemu preseže največjo pričakovano vrednost. Test nam pove dosti o stabilnosti naše programske opreme. Poleg prej omenjenih testov poznamo še ***test vzdržljivosti***, ki išče luknje v pomnilniku ali druge probleme, do katerih lahko pride pri daljšem izvajanju programa. Na tem mestu bi omenil še ***test uporabnosti***, ki je potreben za preverjanje uporabniškega vmesnika, odgovori nam na vprašanje, če je le-ta preprost za uporabo ter hitro razumljiv novemu uporabniku. Poleg vseh naštetih lahko v ta sklop vključimo še ***konfiguracijsko testiranje***, slednje nam zagotavlja, da razvita programska oprema pravilno deluje na vseh podprtih programskih in strojnih platformah.

## 5.2 TESTIRANJE VARNOSTI PROGRAMSKE OPREME

Glavni namen takšnega testiranja je ugotoviti, kako dobro sistem ščiti podatke ter pri tem ohranja predvidene funkcionalnosti. Sistemska varnost je ključnega pomena pri programski

opremi, ki obdeluje zaupne podatke, zato mora biti sistem dobro zaščiten pred vdori hekerjev. Pri testiranju varnosti programske opreme moramo pokriti šest varnostnih konceptov: zaupnost, integriteta, avtentikacija, avtorizacija, razpoložljivost in nezataljivost.

ZAUPNOST — varnostni ukrep, ki varuje pred razkritjem podatkov komu drugemu, kot prejemniku.
INTEGRITETA — ukrep, ki zagotavlja prejemniku, da določi, ali so informacije, ki jih poseduje, korektne.
AVTENTIKACIJA — proces oblikovanja uporabnikove identitete.
AVTORIZACIJA — proces odločanja o tem, da je prosilcu dovoljeno sprejeti storitev oziroma izvesti operacijo.
RAZPOLOŽLJIVOST — zagotoviti, da bodo informacijske in komunikacijske storitve na voljo, kadar se to pričakuje. Informacije morajo biti na voljo poblaščenim osebam, kadar jih potrebujejo.
NEZATALJIVOST — ukrep, ki skrbi, da iz sistema ne izginejo podatki o aktivnosti, ki se je zgodila v preteklosti.

# 6. ORODJA ZA TESTIRANJE

Orodja so osnova za vse vrste testiranja. Brez orodij bi bilo testiranje programske opreme zelo naporno opravilo za vsakega testerja. Orodja so prisotna v vseh fazah testiranja. Že za samo pripravo na testiranje ter odločanje o izvajanih testih si lahko delo olajšamo z različnimi orodji. Seveda pa je večji pomen na orodjih za samo izvajanje testiranja ter na orodjih za izdelovanje poročil o testiranju. Člani testne ekipe morajo biti seznanjeni ne samo z orodji za golo testiranje programske opreme, temveč tudi z orodji, ki pomagajo pri večji učinkovitosti samega procesa testiranja.

## 6.1 ORODJA ZA NAČRTOVANJE IN UPRAVLJANJE TESTIRANJA

Priprave na testiranje se začnejo lahko že pri samem zajemu zahtev, katerim mora ustrezati programska oprema, ki jo razvijamo. Zato je dobro, da se ekipa, zadolžena za testiranje, spozna tudi z orodji za zajem zahtev, kot so Rational Test Manager, HP Quality Center itd. Ta orodja lahko uporabimo za upravljanje s sredstvi testiranja, izdelavo plana in testnih primerov. Služijo lahko tudi za nadzorovanje izvajanja, poročanje ter poganjanje avtomatiziranih testov. V to skupino lahko pogojno štejemo tudi orodje Jira, ki v osnovi sicer ni namenjeno načrtovanju testiranj, vendar omogoča učinkovito upravljanje s spremembami in dodeljevanje opravil v zvezi z deli programske opreme, v kateri smo med testiranjem zaznali napake. Jira je zanimiva zlasti zaradi relativno široke sprejetosti med razvijalci. V nadaljevanju bom opisal dve orodji, ki jih lahko uporabimo v tej fazi.

### 6.1.1 HP Quality Center

[10]Je spletno orodje za upravljanje testov, ki omogoča nadzor testiranja nad celotnim projektom. Intuitivni vmesnik programa za načrtovanje in upravljanje opravil, kot so pokritje zahtev, načrtovanje testnih primerov, poročanje o izvedenih testih, upravljanje z odkritimi napakami in avtomatizacijo testiranja, olajša delo izvajalcem in načrtovalcem testiranja. Ker se orodje uporablja preko spleta, to pomeni, da je povsod dostopno, če le imamo dostop do internetnega omrežja.

HP Quality Center lahko razdelimo na dva dela. Eden je namenjen skrbniku orodja, drugi pa uporabniku. V angleščini se prvi imenuje Site Administration Bin (SABin), drugi pa Quality Center Bin (QCBin).

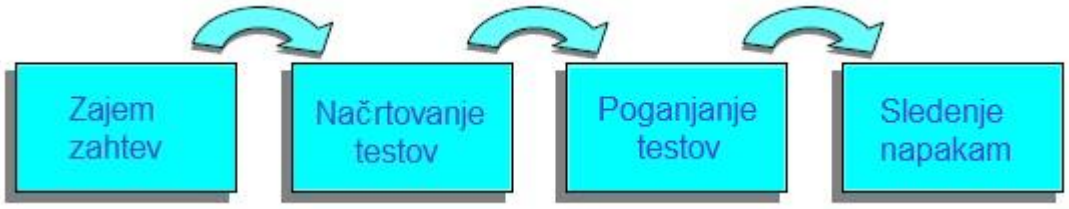
- SABin: Je pomemben del orodja in izhodiščna točka za uporabo HP Quality Center. Tu se izvajajo vse administrativne dejavnosti, kot so ustvarjanje projektov, dodeljevanje uporabnikov k projektu, ustvarjanje posebnih vlog itd. Ta del se uporabi tudi za konfiguracijo aplikacije z orodji. Na primer: če želite uporabiti Winrunner ali Quality Test Professional skripte iz orodja Quality Center, morate to določiti tukaj.

- QCBin: To je del, namenjen uporabnikovi interakciji z orodjem. V našem primeru imamo kot uporabnike v mislih člane testne ekipe. QCBin vmesnik ponuja funkcionalnosti, kot so ustvarjanje načrta testiranja, opredelitev zahtev, ustvarjanje testnih primerov, ustvarjanje testnega okolja, povezovanje zahtev z napakami, v glavnem vse, kar počne tester pri vsakodnevnih opravilih, razen samega izvajanja testov.

Quality Center je nameščen kot storitev v Windows okolju. Preden začnemo z delom, se moramo prepričati, da je Quality Center že pognan. Kot je že bilo omenjeno, lahko do njega dostopamo preko spleta na naslovu http://8080/sabin/SiteAdmin.htm, če seveda predpostavimo, da se nahaja na privzetih vratih. Ta stran je začetna točka, tu se nahaja prijavni zaslon, kjer vpišemo podatke o skrbniku (uporabniško ime in geslo), ki smo jih določili pri namestitvi centra. Ko ste prijavljeni v SABin, lahko izvajate vsa zgoraj omenjena skrbniška opravila.

Ko je končana faza definiranja projektov, uporabnikov itd, se lahko začne uporabljati QCBin. To je najbolj skupni vmesnik, ki ga uporabljajo stranke in uporabniki. Quality Center skrbi za pravilno dodelitev dovoljenj na podlagi vlog, ki jih ima določena oseba pri projektu. Na primer upravljalec testiranja lahko ustvarja projekte, vodja testiranja lahko pripravi načrt testiranja, tester lahko piše testne primere. Takšen pristop na principu vlog omogoča preprosto nadzorovanje dostopov do različnih delov projekta ter porazdeli odgovornosti med člani ekipe.

Naslednja slika prikazuje 4 faze, ki zajemajo proces testiranja v HP Quality Center.




Slika 1: Proces testiranja, razdeljen po fazah

### 6.1.2 JIRA – Sistem za sledenje napakam

Je zelo močno orodje, ki se lahko uporablja kot sistem za spremljanje napredovanja projekta, razporejanje dela, nadzor nad razreševanjem napak in novih zahtev. Jira ima veliko možnosti za povečanje produktivnosti, pomembno pa je, da znamo te možnosti izkoristiti. S prijavo v sistem Jira dobimo na voljo dve možnosti: Projekt in Kategorije projektov. Če izberemo kategorije projektov, lahko določimo, kako se projekti delijo po kategorijah. Recimo, lahko jih kategoriziramo po tem, katera ekipa izvaja določen projek. Na primer: če imamo ekipo A in ekipo B, vsaki ekipi določimo, za katere projekte je zadolžena. Ustvarjanje novih kategorij ali spreminjanje že narejenih je zelo enostavno. Pomembno je to, da se zavedamo, da lahko projekt pripada le eni kategoriji, medtem ko ima lahko ena kategorija več projektov.

Ko določimo kategorije, lahko začnemo raziskovati in ustvarjati različne vloge z uporabo brskalnika vlog. Mogoče za manjše ekipe to ni tako uporabno, vendar pa je pri večjih ekipah lahko to zelo učinkovito za proženje napak, ustvarjanje sistemov za obveščanje in tako naprej.

Še ena pomembna zadeva pri Jiri so dogodki (Events). Dogodki so zelo mogočni, in se obnašajo kot prožilci. Z dogodki določimo zanimive stvari, kot so, kako bo Jira prikazala določen dogodek, ki se je sprožil, katere operacije delovnega toka bodo na voljo ob določenem dogodku in kdo bo o tem dogodku obveščen. Ob tem je dobro pojasniti, kaj je delovni tok pri Jiri. Delovni tok je zelo pomembna funkcija, ki nam omogoča nastaviti, kaj se zgodi v vsakem koraku, kako pomanjkljivosti in težave prehajajo iz enega stanja v drugega ter katere možnosti so na voljo v vsakem prehodu. Vsi ti prehodi delujejo kot prožilci, k vsakemu prehodu lahko določimo pogoje, veljavnosti ter funkcije, ki jim sledijo.

V Jiri so operacije večina zastavljane kot sheme. Lahko imamo različne sheme za delovne tokove, obvestila in dovoljenja. Ustvariti moramo ločene sheme, ker bomo mogoče potrebovali različne sheme za različne projekte. Sheme se celo uporabljajo pri nazdoru look-n-feel (glej in občuti), za določanje, kateri deli bodo vidni pri vsakemu prehodu. To lahko dosežemo z uporabo zaslonskih shem(ang. Screen Schemes).
Ena izmed zanimivih funkcij pri Jiri je nastavljanje oglasnih desk (ang. Dashboards). Imamo lahko več oglasnih desk, in na vsaki lahko objavljamo poročila, uporabna za nas. To nam služi za sprotno informiranje o napredku projekta že na prvi strani Jire. Če želimo nastaviti določeno oglasno desko, moramo narediti poizvedbo z uporabo možnosti Find Issue (poišči težavo), ki na podlagi rezultata zgradi tabelo. Te tabele se nato lahko objavijo na oglasni

deski, ki se potem prikaže na domači strani vsakič, ko se prijavimo v Jiro, ter se tudi samodejno posodobijo.

## 6.2 ORODJA ZA POGANJANJE TESTNIH PRIMEROV

Tu bom predstavil nekaj orodij za poganjanje testnih primerov, ki omogočajo avtomatizacijo testiranja. Večina orodij za avtomatizacijo testiranja podpira snemanje in predvajanje funkcionalnosti. Uporabljamo jih za avtomatizacijo in izvedbo avtomatiziranih testnih primerov. Med bolj znana orodja štejemo recimo IBM Rational Functional Tester, Rational Robot, QTP, WinRunner in SilkTest.

### 6.2.1 IBM Rational Functional Tester (RFT)

Je orodje za avtomatizirano testiranje, ki ga je razvilo podjetje Rational, kasneje pa je podjetje prišlo v last IBM-a. Kot skriptna jezika uporablja RFT Javo in VB.NET. RFT orodje je že integrirano v Microsoft Visual Studio in Eclipse, kar pomaga k hitrejšemu spoznavanju orodja, če nam delo v teh dveh okoljih ni tuje.

Orodje v prvi vrsti uporablja oddelek za zagotavljanje kakovosti programske opreme (ang. Software Quality Assurance). Testerji ustvarijo skripte s pomočjo snemalnika testov, ki ujame uporabnikove dejavnosti na aplikaciji, katero testirajo. Snemalni mehanizem naredi testne skripte na podlagi dejavnosti. Od verzije 8.1 so te skripte predstavljene kot zaslonske slike, temu pravimo testiranje na način »Storyboard«. Z RFT lahko te skripte poženemo, da preverimo funkcionalnosti aplikacije. Te skripte tipično poganjamo v serijskem načinu (ang. batch mode), kjer je več skript združenih skupaj in jih poženemo brez nadzora. Med fazo snemanja mora uporabnik uvesti točke preverjanja. Te točke predstavljajo pričakovano stanje sistema, kot je recimo posebna vrednost v polju, ali dana lastnost objekta, recimo omogočeno ali onemogočeno. Ko nastopi faza predvajanja, se vse razlike med posnetkom in dejanskimi rezultati, ujetimi med predvajanjem, shranijo v RFT dnevnik. Tester potem pregleda dnevnik in tako ugotovi, ali je bil dejansko odkrit kakšen hrošč.

Ključne tehnologije pri orodju RFT (op.p.: namerno so uporabljeni originalni angleški izrazi, zaradi boljše prepoznavnosti):

- Storyboard Testing (Ta tehnologija omogoča testerjem urediti testne skripte tako, da upravljajo z zaslonskimi slikami aplikacije.)
- Object Map (To je osnovna tehnologija, ki jo testerji uporabljajo za iskanje objektov v aplikaciji in za upravljanje z njimi. Snemalnik avtomatsko ustvari zemljevid objektov (ang. Object Map) in vsebuje seznam lastnosti objektov, ki jih potrebujemo za njihovo identifikacijo med predvajanjem.)
- ScriptAssure (Tehnologija, ki zagotavlja, da se razlike, ki so nastale nad objekti po snemanju, zanemarijo med predvajanjem. Kako velike so lahko te razlike, ki bodo zanemarjene, pa določi uporabnik sam.)

- Data Driven Testing (Zaradi večkratnega poganjanja regresijskih testov, z različnimi podatki, snemalnik avtomatsko parametrizira posnete podatke in jih združi ter shrani v preglednice. To mogoča testerjem, da dodajajo nove primere testnih podatkov, ne da bi spreminjali testno kodo. Takšna strategija poveča pokritost testiranja določene funkcionalnosti.)
- Object Proxy Mechanisem (Pomaga RFT-ju ravnati z danim objektom, na podlagi poznavanja objektnega vmesnika. To je predvsem pomembno pri objektih, ki so prilagojeni s strani razvijalca, da zagotovijo določene zahteve uporabe. V tem primeru lahko razvijalci ustvarijo proxy objekt, kjer definirajo kodo vmesnika za prilagojeni objekt.) [10]

### 6.2.2 QUnit

QUnit je zelo uporabno orodje za testiranje, ki temelji na JavaScript (skriptni programski jezik) kodi. Sposobno je testirati kakršnokoli generično kodo v JavaScript-u ter celo testirati kodo v tem skriptnem jeziku na strani strežnika. QUnit je posebej uporaben pri regresijskem testiranju. Kadarkoli je prijavljen kakšen hrošč, se napiše test, ki potrdi obstoj le-tega, nato se ga odpravi. Oboje se shrani. Nato se vsakič, ko nekdo spreminja kodo, znova požene ta test. Če se hrošč zopet pojavi, bo takoj odkrit in ga bo tudi lažje odpraviti, ker točno vemo, kje je bila koda spremenjena. Dobro pokriti testi enot nam omogočajo varno spreminjanje kode. Lahko jih poženemo po vsaki najmanjši spremembi ter tako vedno vemo, katera sprememba je pokvarila kakšno funkcijo.

Če želimo uporabiti QUnit, moramo vključiti datoteki *quint.js* in *qunit.css*, ter priskrbeti osnovno HTML strukturo za predstavitev razultatov. Za lažjo predstavo sem v nadaljevanju navedel primer uporabe in pripadajoči HTML.

Osnovni primer uporabe (Slika 2)[10]:

```
test("a basic test example", function() {
  ok( true, "this test is fine" );
  var value = "hello";
  equals( "hello", value, "We expect value to be hello" );
});

module("Module A");
test("first test within module", function() {
  ok( true, "all pass" );
});
test("second test within module", function() {
  ok( true, "all pass" );
});
module("Module B");
test("some other test", function() {
  expect(2);
  equals( true, false, "failing test" );
  equals( true, true, "passing test" );
});
```

**Slika 2: Primer uporabe QUnit**

Izvorna HTML koda (Slika 3)[10]:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <script src="http://code.jquery.com/jquery-latest.js"></script>
  <link                                                    rel="stylesheet"
href="http://github.com/jquery/qunit/raw/master/qunit/qunit.css"
type="text/css" media="screen" />
<script                                                 type="text/javascript"
src="http://github.com/jquery/qunit/raw/master/qunit/qunit.js"></script>

  <script>
  $(document).ready(function(){

test("a basic test example", function() {
  ok( true, "this test is fine" );
  var value = "hello";
  equals( "hello", value, "We expect value to be hello" );
});

module("Module A");

test("first test within module", function() {
  ok( true, "all pass" );
});

test("second test within module", function() {
  ok( true, "all pass" );
});

module("Module B");

test("some other test", function() {
  expect(2);
  equals( true, false, "failing test" );
  equals( true, true, "passing test" );
});

  });
  </script>

</head>
<body>
  <h1 id="qunit-header">Qunit example</h1>
 <h2 id="qunit-banner"></h2>
 <h2 id="qunit-userAgent"></h2>
 <ol id="qunit-tests"></ol>
 <div id="qunit-fixture">test markup, will be hidden</div>
</body>
</html>
```

**Slika 3: Izvorna koda v HTML za uporabo v QUnit**

Razlaga osnovnih gradnikov, ki naj jih vsebuje vsako telo (ang. Body) testnega HTML-ja:
Element **#qunit-header** naj vsebuje ime testa, ki ga QUnit ne spreminja.
Element **#qunit-banner** bo poskrbel, da se rezultat testa obarva rdeče, če je test spodletel, ter

zeleno, če je test uspel.
Element **#qunit-userAgent** je nastavljen, da prikaže lastnost navigator.userAgent.
Element **#qunit-tests** je uporabljen kot shramba za rezultate testiranja.
Element **#qunit-fixer** skrbi za označevanje testov in upravljanje z oznakami ter se bo avtomatsko ponastavil po vsakem testu. [14]

### 6.2.3 FitNesse

FitNesse je Wiki, to je strežniški program, ki uporabnikom omogoča prosto ustvarjanje in urejanje spletnih strani s spletnim brskalnikom. Wiki podpira hiperbesedilne povezave ter s preprosto skladnjo omogoča ustvarjanje novih strani in sprotne povezave med stranmi v sistemu Wiki, ki temelji na ogrodju Fit, ki se ga uporablja za avtomatizirano testiranje sprejemljivosti testnih primerov. Orodje omogoča uporabnikom, testerjem in programerjem ugotoviti, kaj naj bi programska oprema počela, hkrati avtomatsko naredi primerjavo, če to dejansko počne. Če povzamemo na enostaven način, primerja pričakovanja uporabnikov z dejanskimi rezultati.[5]

Namen FitNesse orodja je odpravljanje napak, ki so nastale v procesu zajema zahtev (analiza), z okrepitvijo sodelovanja, ter omogočiti deterministično izvršljive avtomatizirane teste. V primerjavi z različnimi testi enot, ki nam povejo, če neka napaka obstaja ali ne v kodi, nam testi, izvedeni z orodjem FitNesse, podajo informacijo o tem, ali je določena funkcija v programski opremi zgrajena na podlagi pričakovanj uporabnikov. Mehanizem poganjanja avtomatskih testov sprejemljivosti, ki ga uporablja to orodje, nam omogoči pridobivanje zgodnje povratnih informacij. Testi, napisani v FitNesse orodju, so deterministični, na primer so pozitvni ali negativni. Najbolj pomembno je, da so testni podatki razviti s strani uporabnikov ali pa ekipe, ki jim pomagajo predstavniki uporabnikov. To omogoči, da so rezultati, ki se od sistema pričakujejo, vsem dobro poznani.
Jedro orodja FitNesse predstavljajo tabele. Testi sprejemljivosti so tukaj definirani v tabelaričnem formatu. Ta deterministični tabelarični format predstavlja knjižnico Fit. Ti testi so opredeljeni s pomočjo vhodnih in pričakovanih izhodnih podatkov. Če želimo uporabiti te tabele za testiranje aplikacije s podatki iz tabel, moramo napisati Fixture Code (to je poseben razred v Javi ali katerem drugem podprtem jeziku). Fixture Code predstavlja nekakšen most med tabelo in testirano aplikacijo. Razume jezik tabele, ki jo nato uporabi za preizkus določene funkcionalnosti v aplikaciji.

Obstajajo različni slogi tabel, ki ustrezajo programski opremi, ki jo preizkušamo. Te tabele so ustvarjene na testnih straneh FitNesse. Če želite izvajati več kot eno testno stran hkrati, jih lahko združimo v skupino testov. Stran lahko preprosto nastavimo tako, da zajema skupino testov (več testnih strani v eni), s spreminjanjem nastavitev. Deluje tudi v obratni smeri, da stran razstavimo na posamezne testne strani. Oba načina sta prav tako implementirana kot tabele.
Teste, ki jih ustvarimo s tem orodjem, lahko poženemo tudi iz ukazne vrstice. To prinaša veliko prednosti. Na primer: lahko jih razhroščujemo, vključimo v svoje skripte, ni nam treba

poganjati spletnega strežnika za izvrševanje testov, lahko generiramo izhodne podatke v obliki HTML ali XML formata.

TestRunner nam ponuja tri različne izhodne formate. To so HTML, XML in navadni tekst ali format Result. Format Result je vmesna oblika shranjevanja podatkov, dokler test teče. Običajno se shranjujejo v začasno datoteko in se po izvršenem testu pošljejo v FitNesse, kjer se pretvorijo v eno izmed prej omenjenih oblik. [10]

### 6.2.3 Anteater

Anteater je ogrodje za testiranje zgrajeno, z orodjem Ant, ki so ga ustvarili pri Apache Jackarta Project. Zagotavlja nam preprost način za pisanje testov, ki preverjajo funkcionalnosti spletnih aplikacij ali XML spletnih storitev.

Vrste testov, ki jih lahko napišemo v Anteatru:

- Pošiljanje HTTP/HTTPS zahteve na spletni strežnik. Ko pride nazaj odziv, se preveri, če le-ta ustreza določenim kriterijem. Lahko preveri glavo HTTP in odzivno kodo, potrdi veljavnost telesa z regexp, Xpath, Relax NG ali s testi contetEquals. Poleg tega lahko preveri tudi nekaj dvojiških formatov. Nove teste je enostavno dodajati.
- Posluša na vhodu katerega URL-ja lokalnega računalnika, če bo prestregel kakšno vhodno HTTP zahtevo. Ko zahteva pride na URL, lahko preveri njene parametre in vsebino ter vrne ustrezni odziv.

Sposobnost sprejemanja in pošiljanja HTTP sporočil je nekaj posebnega pri orodju Anteater, kar je še posebej uporabno, ko delamo teste za aplikacije, ki uporabljajo visok nivo komuniciranja na podlagi SOAP, na primer ebXML ali BizTalk. Aplikacije, ki so napisane z uporabo teh protokolov, običajno prejemajo SOAP sporočila ter pošiljajo nesmiselne odzive. Šele nato poskusijo obvestiti klienta, s pomočjo HTTP zahteve, o rezultatih obdelave podatkov. To so tako imenovana asinhrona SOAP sporočila, ki predstavljajo jedro številnih visokonivojskih protokolov, temelječih na SOAP ali XML sporočilih. Na Sliki 4 je kratek primer, napisan z orodjem Anteater[9].

```
<target name="simple">
  <soapRequest description="Post a simple SOAP request"
               href="http://services.xmethods.net:80/soap"
               content="test/requests/get-quote">
    <namespace                                                prefix="soap"
uri="http://schemas.xmlsoap.org/soap/envelope/"/>
    <namespace prefix="n" uri="urn:xmethods-delayed-quotes"/>
    <match>
      <responseCode value="200"/>
      <xpath select="/soap:Envelope/soap:Body/n:getQuoteResponse/Result"/>
    </match>
  </soapRequest>
</target>
```

**Slika 4: Primer SOAP sporočila**

### 6.2.5 Orodja za testiranje v sklopu Microsoft Visual Studio 2010

Gre za orodje s katerim imam tudi sam izkušnje. Prvič sem se z njim spoznal na fakulteti, nato pa sem ga uporabljal tudi pri delu v podjetju Gama System, kjer sem večinom opravljal delo testerja. Je zelo mogočno orodje, ki je v prvi vrsti namenjen predvsem razvijanju različne programske opreme, od spletnih aplikacij, spletnih servisov ter Windows aplikacij, ki temeljijo na Microsoftovi platformi Windows.

Lahko pa ga uporabljamo tudi kot orodje za testiranje. Sam imam nekaj izkušenj s pisanjem testov programskih enot (ang. Unit tests). Ima tudi že vgrajen razhroščevalnik (ang. Debugger), ki nam pomaga pri iskanju hroščev v kodi. V nadaljevanju sledi nekaj kode (Slika 5) kot primer preprostega testa programskih enot(ang. Unit Test), ki sem ga napisal sam med prakso v prej omenjenem podjetju. Teste sem pisal na podlagi specifikacije, ne da bi poznal izvorno kodo programa, ki sem ga testiral. Komentarje sem označil z zeleno zaradi boljšega pregleda.

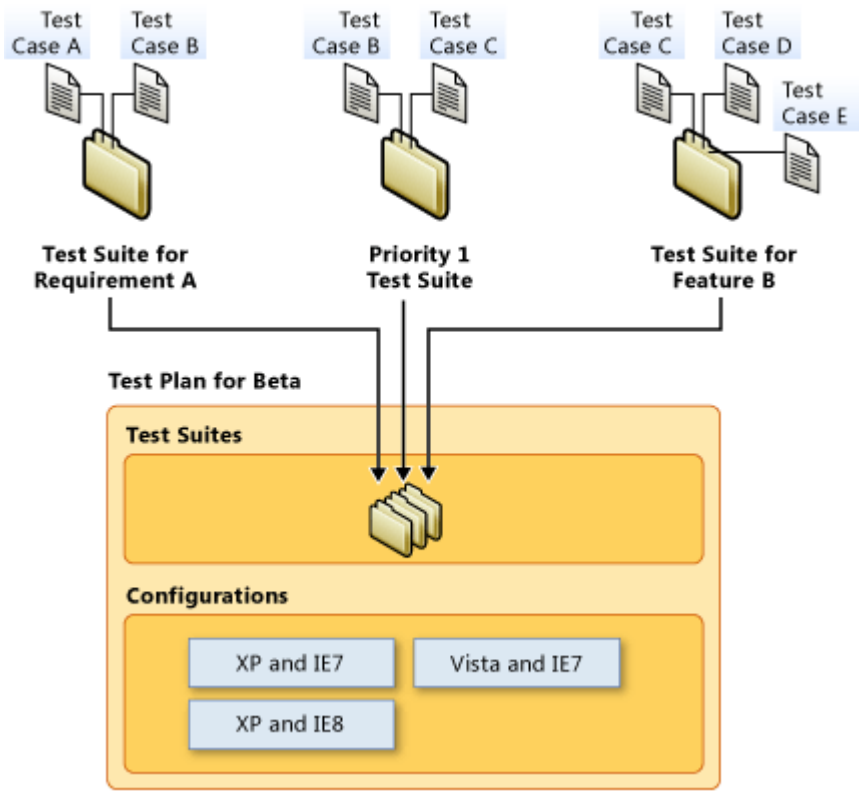
```
namespace GetUserTest
{
    /// <summary>
    /// Razred je namenjen testiranju metod oziroma funkcij, ki so bile zapisane v specifikaciji, imena metod so v angleščini, takšna so pravila lepega programiranja, ker je angleščina glavni jezik v računalništvu
    /// </summary>
    [TestClass()]
    public class GetUserTest : Input
    {

        #region Test using a wrong username
        /// <summary>
        /// Pri prvi testni metodi se uporabi neko izmišljeno uporabniško ime, ki ne more obstajati v bazi. Metoda preveri, če funkcija vrne pravilno izjemo(ang. exception), kakor je to zapisano v specifikaciji
        /// </summary>
        [TestMethod()]
        public void GetUserWrongUsername()
        {
            string username = "doesntexist";
            bool exceptionHappened = false;
            ServiceClient client = new ServiceClient("WSHttpBinding_IService");
            try {
                Service failName = new Service();
                failName.GetUser(username);
            }
            catch (FaultException<FaultExceptions.InvalidUserFaultContract>) { exceptionHappened = true; }
            catch { Assert.Fail("General error while calling GetUserBySigningCode method"); }
            Assert.IsTrue(exceptionHappened, "The InvalidUserFaultContract exception was recieved");
        }
```

```
        /*vzpostavitev povezave s podatkovno bazo, tekst označenkjer je
rumena črta se drugače nahaja pot do pravilne podatkovne baze*/
         static       void       Database_SetConnection(object       sender,
                    .SetConnectionEventArgs e)
         {
             e.Connection = new ConnectionInfo()
             {
                 DatabaseName                                                =
ConfigurationSettings.AppSettings["DatabaseName"],
                 ServerName                                                  =
ConfigurationSettings.AppSettings["DatabaseServerName"],
                 TrustedConnection = true
             };
         }
     }
}
```

Slika 5: Primer testa programskih enot

Novi Visual Studio 2010 vključuje novo aplikacijo, imenovano Test Manager, ki nam pomaga zmanjšati napor pri testiranju s pomočjo načrtov testiranja. Z njim ustvarimo načrt testiranja, ki mu po želji dodamo skupine testov, testne primere ali nastavitve, ki jih potrebujemo, tako kot je prikazano na Sliki 6.[12]




Slika 6: Načrt testiranja v Test Manager-ju

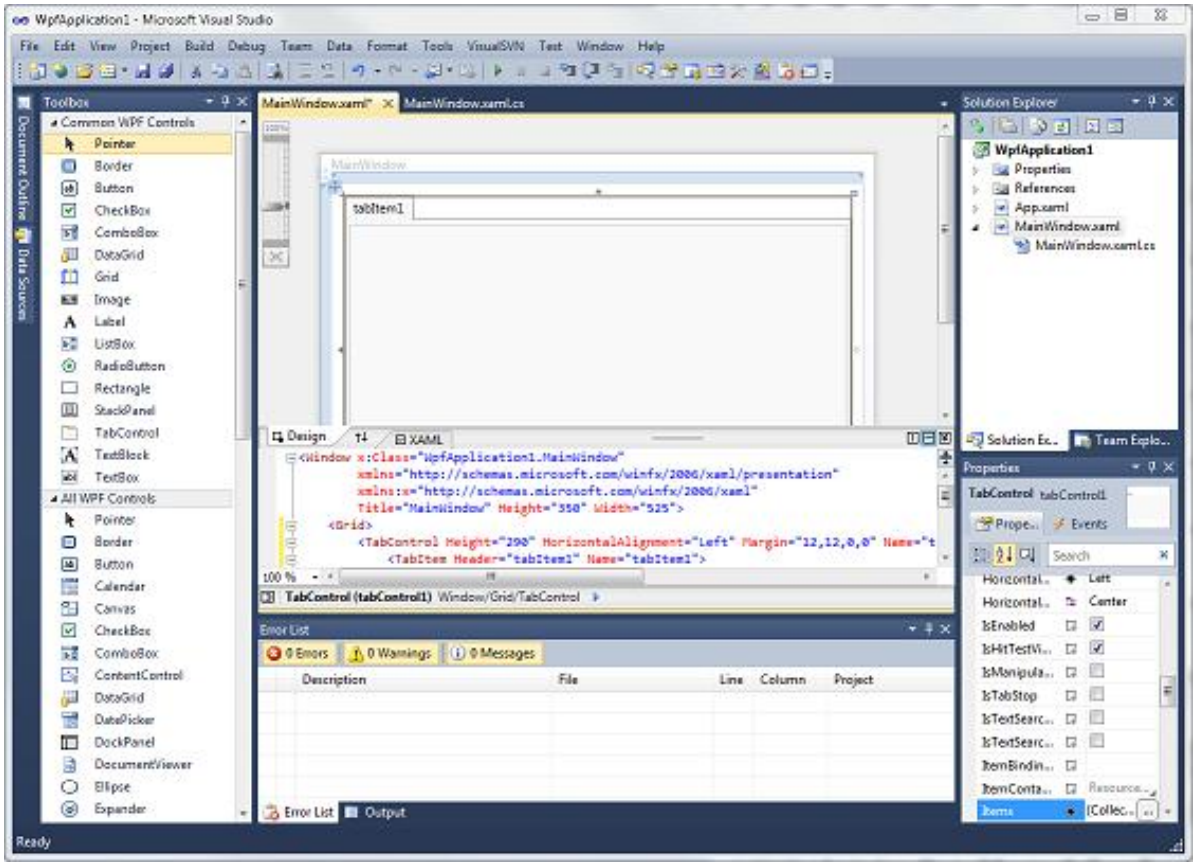
Po izdelanem načrtu smo pripravljeni na testiranje. Ko so zahteve, katerim naj aplikacija ustreza, oziroma funkcije, ki naj jih ima, pripravljene za testiranje, lahko začnemo s poganjanjem testov za vsako konfiguracijo, ki smo jo določili. Takšen načrt nam omogoča

opazovanje napredka pri izvajanju testov in nam sporoča, koliko testov nam je še ostalo za izvedbo.

Teste lahko izvajamo tudi ročno iz programa Microsoft Test Manager z uporabo orodja Microsoft Test Runner. Lahko poganjamo tudi avtomatizirane teste, če je avtomatizacija povezana s testnimi primeri. Rezultati teh testov se potem povežejo z načrtom testiranja. Poleg tega lahko tudi samostojno poganjamo teste iz Visual Studia, ki niso v načrtu testiranja. Te teste se lahko odločimo poganjati individualno kot del politike check-in ali na podlagi testnih kategorij. Lahko pa jih izvedemo kot del gradnika, ustvarjenega s programom Team Foundation Build, ali iz ukazne vrstice.

Zaradi integriranosti testnih orodij v Visual Studio lahko njihove rezultate shranimo v podatkovno bazo, generiramo trende in poročila o zgodovini, ter primerjamo različne vrste podatkov. Uporabimo lahko podatke, da vidimo, koliko ter kateri hrošči so bili odkriti s pomočjo testov.

Programski paket Visual Studio 2010, je tako obsežen, da poglabljanje vanj na tem mestu ni smiselno. Zaenkrat je dovolj, da dobimo osnovni občutek, kaj vse je mogoče početi s tem programom ter na kakšen način nam lahko olajša naše delo pri testiranju programske opreme.




**Slika 7: Slika vmesnika od MS Visual Studio 2010**

# 7. PRIMER PROCESA TESTIRANJA PODJETJA APPLABS

Sledeči proces testiranja sem povzel po skripti z naslovom »A Test Process for all Lifecycles« od podjetja AppLabs. To je veliko neodvisno podjetje, ki se ukvarja s testiranjem in zagotavljanjem kakovosti programske opreme. Z več kot desetletjem izkušenj je AppLabs postal partner več kot 600 podjetjem. Njihov proces sem vzel, ker je bil prosto dostopen na spletu, ter dobro in nazorno opisan. Služi naj kot primer, kako takšen proces poteka v podjetjih z večjimi razvojnimi skupinami. Pri večjih skupinah lahko za testiranje namenimo več ljudi, kar se vidi tudi v enem izmed diagramov spodaj, ki nam kaže hierarhijo testne ekipe ter koliko članov jo sestavlja. Zraven moramo upoštevati, da lahko naziv izvrševalec testov dobi več kot ena oseba.

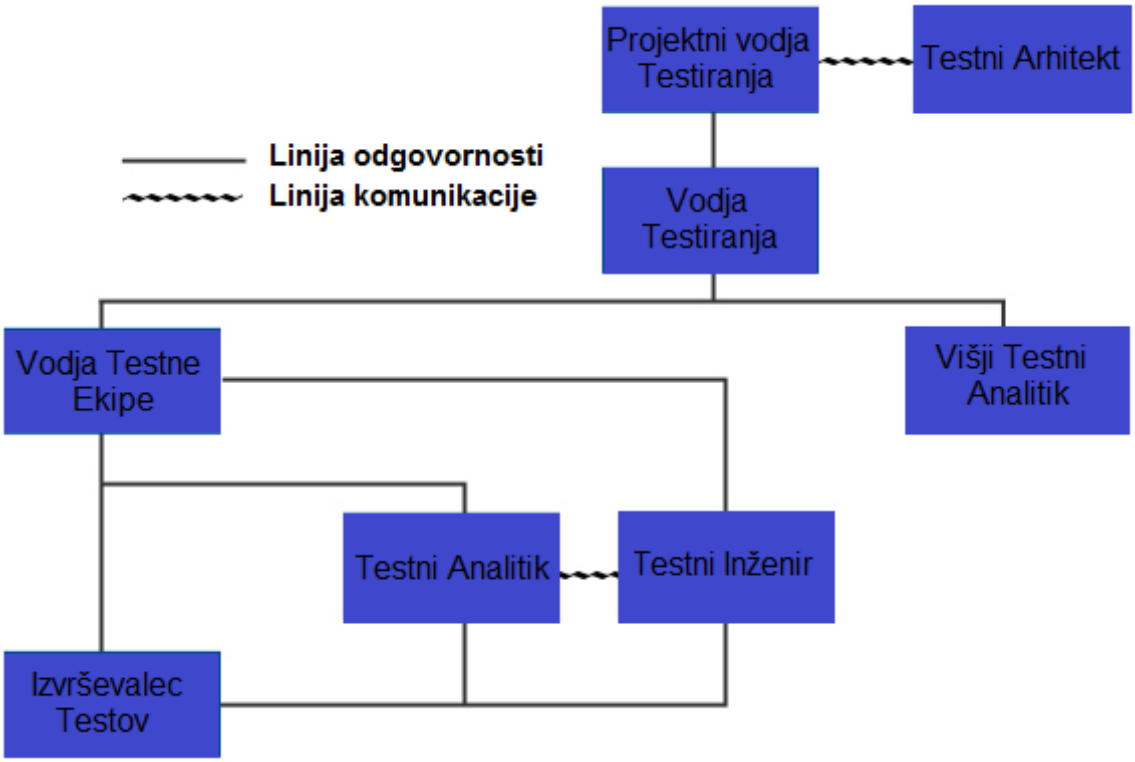
Pri procesu testiranja je pomembno, da ima podjetje, ki se ukvarja z razvojem programske opreme, izdelano strategijo in politiko testiranja. Se pravi, vedeti morajo, zakaj testirajo stvari, kaj je pomembno za podjetje (npr. stroški, kvaliteta, čas, obseg izdelave), kdaj pride testiranje na vrsto, kdo ga bo izvajal. Sam proces se nanaša na strategijo testiranja, ki jo moramo upoštevati pri načrtovanju testiranja, analizi in zasnovi testiranja, izgradnji testov in izvrševanju testov. Proces testiranja mora dati odgovore na naslednja vprašanja:

- Kaj bomo testirali in zakaj?
- Kako bomo testirali?
- Kdaj bomo testirali?
- Kdo bo testiral?
- Kaj je bilo odkrito?
- Kdaj bo testiranje končano?

Slika 8 prikazuje faze, ki jih zajema proces testiranja:


**Slika 8: Proces testiranja v AppLabs**

V nadaljevanju (Slika 9) lahko vidimo tipično hierarhijo vpletenih oseb, ki imajo dodeljeno vsak svojo vlogo in odgovornosti, potrebne za uspešno planiranje, analiziranje, izgradnjo in izvrševanje procesa testiranja.




Slika 9: Hierarhija v testni ekipi podjetja AppLabs

## 7.1 FAZA 1: NAČRTOVANJE TESTIRANJA

V tej fazi projektni vodja testiranja in/ali vodja testiranja naredita celotni testni načrt (ang. Master test plan) za cel program oziroma projekt, na katerem bo potekalo testiranje. Če je potrebno, se naredi plan tudi za podprojekte, faze testiranja itd., v skladu z velikostjo in zahtevami projekta. Ta načrt nam bo povedal, kaj bomo testirali in zakaj.

Med vsemi fazami bosta vodja testiranja in vodja testne ekipe redno dostavljala poročila o napredku, narejenem v posamezni fazi.

## 7.2 FAZA 2: ANALIZA IN ZASNOVA TESTIRANJA

V tej fazi sodelujejo višji testni analitik, testni analitik in vodja testne ekipe. Skupaj določijo, kako bo sistem testiran ter naredijo testno specifikacijo in zasnovo testiranja. Poleg tega bodo določili podatke, ki jih bodo potrebovali v podatkovni shrambi ter izbrali okolja, potrebna za določena testiranja. Z napredovanjem analize in zasnove testiranja se kreira začetni razpored testiranja, kjer se določi, kdaj bo kateri proces ali sistem testiran.

Testna specifikacija naj vsebuje vse podrobnosti opravljene analize, vključno s sestanki, delavnicami ter dokumenti, uporabljenimi pri analizi sistema, ki ga testiramo.

Zasnova testiranja določi testne primere, ki so potrebni za potrjevanje in preverjanje sistema, ki ga testiramo. Prav tako se pri zasnovi identificirajo, vendar ne tudi zgradijo, testne skripte, testni podatki in testna okolja, ki so potrebna za izpolnitev testnih primerov.

### 7.3 FAZA 3: GRADNJA TESTOV

Tudi tukaj so glavni sodelujoči višji testni analitik, testni analitik in vodja testne ekipe. Njihova naloga je ustvariti testne skripte, pridobiti ali narediti testne podatke ter zgraditi testna okolja, potrebna za določena testiranja. Poleg tega se bo, ko gradnja testov napreduje, razpored testiranja dopolnil tako, da bomo določili, kdaj se bodo testne skripte, za potrjevanje in preverjanje testiranega sistema, izvrševale.

### 7.1. FAZA 4: IZVRŠEVANJE TESTOV

V tej fazi preidemo na dejansko testiranje. Skupina ljudi, določenih za testiranje, dobi delovne pakete (skupine testnih skript, ustvarjenih s strani testnih analitikov) za izvrševanje v skladu z razporedom testiranja. Med tem izvrševanjem bodo testerji zabeležili vse odkrite napake z uporabo primernega orodja ter sproducirali zapise o izvršenih testih. Ti zapisi bodo povzeli, če so bile testne skripte uspešno ali neuspešno izvršene ter vključili vse odkrite napake.

Vodja testne ekipe konstantno posodablja in modificira razpored testiranja, v skladu z napredovanjem izvršenih testov. Spremembe dela na podlagi podatkov o tem, katere skripte so bile uspešno izvedene, katerih sploh ni mogoče pognati zaradi odvisnosti od drugih stvari, ki morajo biti ponovno testirane zaradi določene okvare ali napake.

Na koncu te faze vodja testne ekipe in vodja testiranja naredita uradno poročilo testiranja, ki vsebuje vse ugotovitve, ki so nastale v določeni fazi testiranja.

## 8. KDAJ KONČATI S TESTIRANJEM?

Testiranje je v teoriji proces, ki lahko traja v neskončnost. Nemogoče je testirati tako dolgo, da so odkrite in odpravljene vse napake. V neki točki moramo testiranje ustaviti ter dostaviti programsko opremo naročniku. Vprašanje tukaj je, kdaj to storiti.

Realno gledano, določimo obseg testiranja na podlagi kompromisa med proračunom, časom in kvaliteto. Temelji na modelih dobička. Najbolj pogost način je na žalost pesimistične narave, to pa je, da prenehamo s testiranjem, ko zmanjka katerega koli dodeljenega sredstva, pa naj bo to časa, denarja ali pa testnih primerov. Bolj optimističen način odločanja je, da zaključimo s testiranjem, ko zanesljivost programske opreme ustreza zahtevam ali pa ko ocenimo, da koristi nadaljnjega testiranja ne opravičujejo več stroškov testiranja. To pa po navadi zahteva uporabo zanesljivostnih modelov za ocenjevanje in napovedovanje zanesljivosti programske opreme, ki jo testiramo. Vsako vrednotenje zanesljivosti zahteva

ponavljanje naslednjega cikla: zbiranje podatkov o odkritih napakah – modeliranje – napovedovanje. Ta metoda ni najbolj primerna za sisteme, ki zahtevajo visoko zanesljivost, ker predolgo traja, da se zberejo vse realno možne napake.

Predviden zaključek testiranja se tako določi že v fazi analize, pri gradnji strategije testiranja. Takrat testni analitik oceni, na podlagi vseh podatkov, ki jih ima o projektu, ter preteklih izkušenj, kako dolgo bo trajalo testiranje ter kaj vse se bo stestiralo. Z napredovanjem projekta se večkrat analizira, kaj vse je že bilo storjeno ter se po potrebi prilagodi časovni okvir, namenjen testiranju.

Faktorji, ki se po navadi upoštevajo pri odločanju, kdaj končati s testiranjem: [11]

1. Proračun, namenjen testiranju. Ali ko stroški testiranja ne upravičijo stroškov projekta.
2. Razpoložljivi viri in njihove zmožnosti.
3. Rok zaključka projeta ter rok zaključka testiranja.
4. Uspešno prestani preizkusi kritičnih in ključnih testnih primerov.
5. Funkcionalna pokritost ter pokritost kode, ki se ujemata z zahtevami stranke do določene točke.
6. Pogostost napak pade pod neko določeno mejo in so odpravljene napake z visoko prioriteto.
7. Napredovanje projekta iz verzije alfa do beta in tako naprej.

## 9. ALTERNATIVE TESTIRANJU

Na testiranje programske opreme se vse bolj gleda kot na bolj problematično metodo za izboljšanje kakovosti. Predvsem zato, ker je uporaba testiranja za iskanje in odpravljanje napak lahko neskončen proces. Kot nakazuje ovira kompleksnosti: možnosti so, da testiranje in odpravljanje napak ne bo nujno izboljšalo kvalitete in zanesljivosti programske opreme. Včasih lahko pri odpravljanju enega problema povzročimo še večje in resnejše težave v sistemu. Primer takega dogodka je izpad telefonskih linij v Kaliforniji in na vzhodni obali leta 1991, kjer je prišlo do katastrofe, ko so spremenili 3 vrstice kode v sistemu signaliziranja.

V ožjem pogledu ima veliko tehnik testiranja pomanjkljivosti. Že leta 1979 je Myers predlagal, tako imenovano človeško testiranje, ki je le ena izmed alternativ tradicionalnega testiranja. Predlagana metoda vključuje pregledovanje, sprehod skozi kodo in ocenjevanje programske opreme. Hamlet (1994) zagovarja pregledovanje kot poceni alternativo testiranju enot. Rezultati poskusov kažejo, da je branje kode s postopno abstrakcijo vsaj toliko učinkovito kot funkcionalno in strukturno testiranje, v smislu števila odkritih napak in porabljenih sredstev za odkrivanje.

Uporaba formalnih metod za dokazovanje pravilnosti programske opreme je prav tako zanimiva smer v raziskovanju. Vendar tudi ta metoda ne premosti ovire kompleksnosti. Dobro

se obnese le pri relativno preprosti programski opremi. Pri večjih in kompleksnejših sistemih, ki so bolj nagnjeni k napakam, ni ravno zanesljiva.

Če pogledamo malo širše, lahko začnemo dvomiti o smislu testiranja programske opreme. Zakaj sploh potrebujemo bolj učinkovite metode testiranja, če pa sploh ni nujno, da odkrivanje in odpravljanje napak vodi do kvalitetnejše programske rešitve. Problem je podoben proizvodnji avtomobilov. V proizvodnji naredimo avto in odpravimo napake in težave. Vendar pa sta takšen način izdelave, zamenjala cevovodna proizvodnja in kvalitetno načrtovanje procesa proizvodnje, ki že med samo izdelavo preprečuje napake pri avtomobilu. To nakazuje, da je načrtovanje razvojnega procesa, ki nam proizvede produkt z manj napakam, lahko bolj učinkovito kot načrtovanje procesa testiranja. Testiranje pa se uporablja izključno za spremljanje in upravljanje kakovosti. To je korak za programsko opremo od izdelave do inženiringa.

# 10. PREDLOG PROCESA TESTIRANJA ZA MANJŠA PODJETJA

## 10.1 UVOD

Na podlagi modela RUP (ang. Rational Unified Process – univerzalni proces, ki določa, kdo dela kaj, kdaj in kako za doseganje določenega cilja[3]), procesa testiranja podjetja AppLabs ter lastnih izkušenj sem oblikoval predlog procesa testiranja, ki bi se lahko uporabljal v manjših podjetjih. Pri tem sem se osredotočil na oblikovanje načrta in scenarijev testiranja, ki so opisani v podpoglavjih 10.3 in 10.4 in temeljijo na nasvetih, podanih v 10.2. V zadnjem podpoglavju 10.5, pa sem podal prikaz celotnega procesa testiranja in njegovo primerjavo z RUP, pri čemer sem predstavil tudi ključne razlike med obema. Za boljšo predstavo in razumevanje sem besedilu dodal dva diagrama, prvi prikazuje moj predlog procesa testiranja, drugi pa je vzet iz RUP ter služi za primerjavo.

## 10.2 ŠEST NASVETOV ZA UČINKOVITO TESTIRANJE

1. Pregledati je potrebno zahteve ter specifikacijo, da izvemo, katere funkcionalnosti mora vsebovati programska oprema.
2. Sodelovati moramo z razvojnim oddelkom ter z analitiki za lažje testiranje funkcionalnosti. Na primer s poslovnim analitikom se pogovorimo glede testnih podatkov, ki jih bomo uporabili pri testiranju. On bo najbolje vedel, kakšni so primerni podatki.
3. Ugotoviti je treba, kaj je že bilo stestirano in kaj ne. Včasih se lahko pojavi nov vidik programske opreme, ki v izvirniku ni bil načrtovan, zato še ni stestiran. Le-tega je potrebno prav tako stestirati, da lahko zaključimo določeno transakcijo.
4. Dobro je ustvariti načrt, ki ga lahko ponovno uporabimo in nadgradimo pri drugih projektih. Smiselno bi bilo zgraditi podatkovno strukturo, ki se bo lahko uporabljala v

podjetju, ter jo z določenimi spremembami ter prilagoditvami prenesti na druge oddelke.

5. O spremembah, narejenih v strukturi sistema, podatkovni bazi ali pa sistemskem vmesniku, mora biti takoj obveščen odgovorni za načrtovanje scenarija testiranja. Razlog je v tem, da se morajo vse spremembe ujemati s funkcionalnimi zahtevami programske opreme. Takšne spremembe zelo vplivajo na scenarij testiranja, zato se porabi več časa za ugotavljanje, kaj je potrebno spremeniti v scenariju ter kako bodo te spremembe vplivale na ostale dele sistema.
6. Potrebno je temljito preizkusiti, ali delujejo vse zahtevane funkcionalnosti programske opreme, s tem zagotovimo, da le-ta služi svojemu namenu.[13]

## 10.3 NAVODILA ZA IZDELAVO NAČRTA TESTIRANJA

A. Izberemo primerno orodje za načrtovanje. Na primer HP Quality Center, ki sem ga opisal v podpoglavju 6.1.1. Testiranje si olajšamo s kakovostnim načrtovanjem programske opreme (ang. Software Quality Engineering). Veliko problemov, ki naj bi se kasneje odpravljali v fazi testiranja, lahko preprečimo že v fazi zajema zahtev ter načrtovanja razvoja. V nekaterih podjetjih imajo ti dve dejavnosti ločeni, ker pa je cilj testiranja ter načrtovanja kakovosti programske opreme enak, kar je zagotavljanje kakovostne, varne in stabilne programske opreme, se ju po navadi združi. Kot je bilo že v prejšnjih poglavjih omenjeno, stroški odpravljanja napak naraščajo z odvisnostjo, v kateri fazi so bile napake odkrite. Zato je potreben velik poudarek na tem, da že v začetnih fazah preprečimo, da bi do določenih napak sploh prišlo.

B. Določimo skupino, zadolženo za testiranje ter za podporo pri načrtovanju razvoja. Določi se odgovorne, ki skrbijo za komunikacijo z ostalimi člani projekta. Med samim načrtovanjem je pomembno sodelovanje z analitiki, ki so odgovorni za zajem zahtev, zato da je zagotavljanje kakovosti pokrito že v samem začetku, to pa je od analize oziroma zajema zahtev dalje. V fazi testiranja pa je pomembno sodelovanje z razvojno ekipo, ki skrbi za odpravljanje najdenih napak, ter seveda s končnimi uporabniki.

C. Raziščemo cilje testiranja. Ustvarimo strategijo testiranja, ki opisuje vse možne scenarije in funkcije, katere mora testiranje pokriti. Dokument ustvarimo na podlagi zahtev, ki so zapisane v specifikaciji za izdelavo sistema. To pomeni testiranje uporabniških funcionalnosti, tehničnih funkcionalnosti ter delovanje sistema z drugo programsko opremo.

D. Nato pride na vrsto pisanje scenarija testiranja. Scenarij vsebuje vsa možna opravila, ki jih bo skupina za testiranje izvajala pri testiranju programske opreme. Vključuje naj vhodne podatke, ki bodo uporabljeni med testiranjem. Scenarij testiranja naj bo čim bolj podroben, hkrati mora karseda slediti naravnemu toku funkcionalnega procesa. S tem zagotovimo testiranje realnih situacij, do katerih bo prišlo pri dejanski uporabi testirane programske opreme. Izberemo tudi orodja, ki jih bomo uporabljali za

testiranje programske opreme. Pomembno je, da čimbolj avomatiziramo testiranje (primer takega orodja je Rational Functional Tester) zaradi časa in stroškov, ki nastanejo v tej fazi. Nekaterih stvari se ne da avtomatizirati, vseeno pa si jih lahko olajšamo z nekaterimi orodji primernimi za pisanje testov programske opreme (Microsoft Test Manager) ter poganjanje testov (Microsoft Test Runner).

E. V scenariju moramo upoštevati tudi dokumentiranje odkritih napak ter delovanje samih funkcionalnosti sistema.

F. Vsakemu izmed testerjev določimo svoja opravila iz scenarija. Za vsako opravilo določimo in dokumentiramo datum zaključka. Pregledamo načrt skupaj z vsemi sodelujočimi, da se dogovorimo glede ciljev plana ter datuma zaključka testiranja.

G. V nadaljevanju je potrebno ustvariti okolje za testiranje. Pripraviti je potrebno primerne računalnike, na katerih je postavljen operacijski sistem, ki ga bo programska oprema potrebovala za poganjanje. Namestiti moramo primerna orodja, ki se bodo uporabila v določenih stopnjah testiranja. Strojna oprema, na kateri bomo izvajali testiranje zmogljivosti, naj bo čimbolj skladna z dejansko strojno opremo, na kateri bo tekel končni izdelek. Ujemati se mora v procesorju, kapaciteti spominskih modulov (RAM pomnilnik), kapaciteti trdih diskov, imeti mora vhodno-izhodne enote, ki jih podpira testirana programska oprema. To pomeni, naj bo zmogljivost računalnikov enaka zmogljivosti uporabnikovih računalnikov. Če pa gre za večji informacijski sistem se mora že pri analizi oziroma načrtovanju določiti strojno opremo, na kateri bo postavljen informacijski sistem.

## 10.4 SCENARIJ TESTIRANJA (KAKO SI SLEDIJO TESTIRANJA)

Vrste in stopnje testiranja sem opisal že v petem poglavju, zato jih bom tukaj samo naštel v logičnem zaporedju, po katerem poteka testiranje:

1. Testiranje programskih enot (poteka v fazi razvoja)
2. Testiranje funkcionalnega sklopa (poteka v fazi razvoja)
3. Integracijsko testiranje
4. Testiranje varnosti programske opreme
5. Testiranje zmogljivosti programske opreme (stresni testi, zanesljivosti, skalabilnosti …)
6. Sistemsko testiranje
7. Testiranje sprejemljivosti

Po potrebi se lahko doda še kakšna vrsta testiranja. Recimo v primeru, da pride zaradi odkritih napak ali pa dodatnih želja naročnika do velikih sprememb v kodi, potem dodamo k testiranju še regresijsko testiranje. V primeru, da bomo izdali nekakšno preizkusno verzijo (beta verzija)

programske opreme, bomo k vsem tem testom dodali še beta testiranje. Možnosti je ogromno, v rokah testnega analitika pa je, da predvidi, katero pot bo izbral pri določenem projektu.

## 10.5 KONČNI PROCES TESTIRANJA

Po preučitvi postopka testiranja po modelu RUP ter procesa testiranja podjetja AppLabs sem se lotil izdelave diagrama, ki prikazuje, kako naj teče proces testiranja od začetne do končne faze. Kot razvojni model sem vzel klasični slapovni pristop, kjer si sledijo faze v naslednjem zaporedju: analiza, načrtovanje, izvedba, testiranje, uvedba. Takšen pristop je primeren za projekte, katerih zahteve dobro razumemo in predvidevamo, da se med samim razvojem ne bodo bistveno spreminjale. Sam proces je zasnovan dokaj enostavno ter se ga prav zaradi tega lahko prenese tudi na ostale razvojne modele, kot so iterativni, inkrementalni in prototipni. To je pomembno zato, ker se v praksi dostikrat uporabljajo kombinacije teh modelov.

V modelu RUP je postopek razdeljen na sedem glavnih aktivnosti, katerim so pripisana opravila, znotraj opravila pa je določeno, kdo je zadolžen zanje, kaj predstavlja obvezne vhodne podatke, ter možne vhodne podatke ter na koncu tudi kaj naj bo rezultat opravila (izhodni dokument). Če pogledamo še malo širše, so znotraj opravil definirani tudi vsi možni koraki, ki se lahko izvajajo v sklopu enega opravila. Pri svojem načrtu nisem šel v takšne podrobnosti, saj že imamo RUP. Moj cilj je bil poenostavitev tega procesa z lažje razumljivim diagramom, ki pa je še vedno dovolj natančen za praktično uporabo. Zasnoval sem osnovni potek aktivnosti ter opravil znotraj njih. Aktivnostim sem pripisal tudi podatke oz. dokumente, ki vstopajo vanje, ter izhodne dokumente, ki vsebujejo podatke o rezultatih aktivnosti. Določil sem tudi glavne nosilce aktivnosti, ki sestavljajo celotno testno ekipo.

| AKTIVNOSTI | Analiza testiranja in izdelava strategije | Izdelava načrta testiranja | Namestitev okolja za testiranje | Testiranje v razvojnem okolju preizkušenih sklopov | Sistemsko testiranje | Testiranje sprejemljivosti |
|---|---|---|---|---|---|---|
| NOSILCI ODGOVORNOSTI | Testni analitik in načrtovalec | Testni analitik in načrtovalec | Vodja testiranja | Tester | Tester | Tester |

**Tabela 2: Nosilci odgovornosti v posameznih aktivnostih**

Vloge, ki sem jih izbral za opravljanje različnih aktivnosti ter opravil znotraj njih, sem v primerjavi z RUP-om skrčil na tri glavne predstavnike. To pa so testni analitik in načrtovalec (ena oseba), vodja testiranja ter tester (v tej vlogi lahko nastopa več oseb, odvisno od velikosti ekipe, ki je na razpolago). V RUP-u za manjše projekte sta vlogi analitika in načrtovalca ločeni. Pri testiranju sodelujejo tudi razvijalci, ki pa jih nisem posebej vključil v ekipo, zadolženo za potek testiranja. Razvijalci so nosilci dveh aktivnosti, to pa sta testiranje programskih enot in testiranje funkcionalnega sklopa. Razlog zakaj niso šteti kot člani ekipe je v tem, ker njihov primarni namen ni testiranje, temveč razvoj, ter za svoje delo odgovarjajo vodji projekta in ne vodji testiranja.

Testni analitik in načrtovalec prevzame prvi dve aktivnosti. Naredi temeljito analizo specifikacije sistema, da se spozna z zahtevami, ki jih zajema. Na podlagi teh podatkov izdela strategijo testiranja. V diagramu je razvidno, kakšne podatke naj strategija vsebuje. Po potrebi se strateški plan lahko razširi, odvisno od narave projekta. Nato začnemo z izvajanjem druge aktivnosti, to je izdelava načrta. Tukaj lahko testni analitik sodeluje tudi z vodjo testiranja, predvsem pri odločanju glede okolja testiranja ter razdelitve nalog med testerje.

Vodja testiranja je zadolžen za namestitev okolja za testiranje. Na podlagi načrta iz prejšnje aktivnosti je potrebno izpolniti tehnične zahteve za nemoten potek testiranja programske opreme. Ta aktivnost naj poteka v razvojni fazi projekta. Ko je okolje nameščeno ter je razvojni oddelek pripeljal programsko rešitev tako daleč, da je primerna za nadaljnje testiranje (vsi moduli so razviti ter je testiranje programskih enot in funkcionalnih sklopov končano), preidemo v naslednjo aktivnost.

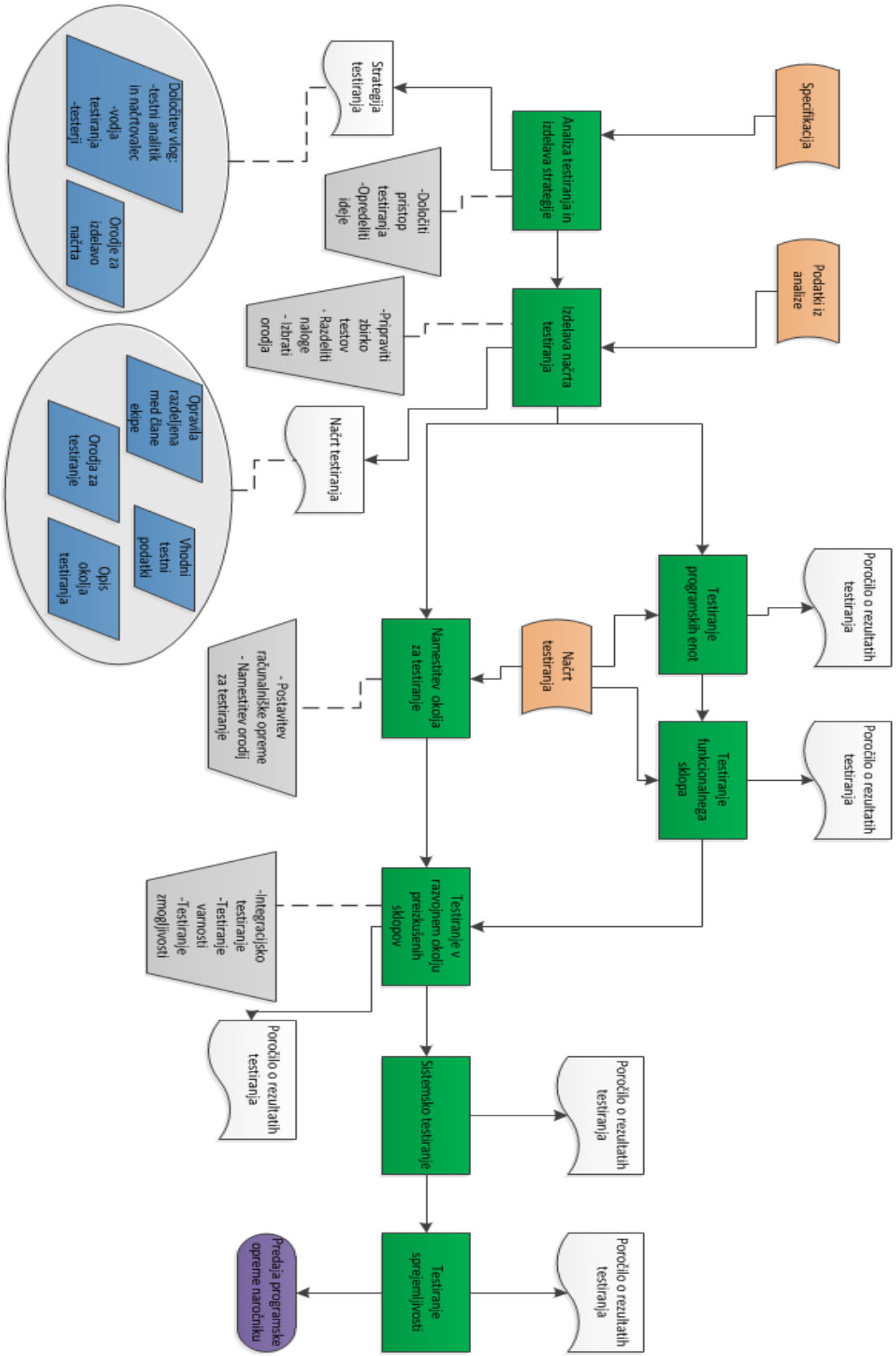
V nadaljevanju prevzamejo odgovornost testerji, začne se testiranje v razvojnem okolju preizkušenih sklopov. To nalogo lahko opravlja ena ali več oseb. Ne glede na to, koliko jih je, pa morajo skrbno beležiti rezultate testiranja, ki jih potem pregleda vodja testiranja ter jih posreduje razvojnemu oddelku, ki je zadolžen za odpravljanje odkritih napak.

Tako potekata tudi naslednji dve aktivnosti: sistemsko testiranje in testiranje sprejemljivosti. Ko programska oprema uspešno prestane test sprejemljivosti, jo lahko predamo v uporabo naročniku.

V primeru iterativnega pristopa se ta model uporablja tako, da po vsaki iteraciji pride na vrsto zopet testni analitik, ki preuči, kaj je bilo že preizkušeno ter zasnuje nove testne zbirke, nato pa si aktivnosti sledijo v enakem vrstem redu, z izjemo namestitve okolja, ki ostaja enako kot v prejšnji iteraciji. Lahko se edino doda kakšno novo orodje, če je to potrebno.

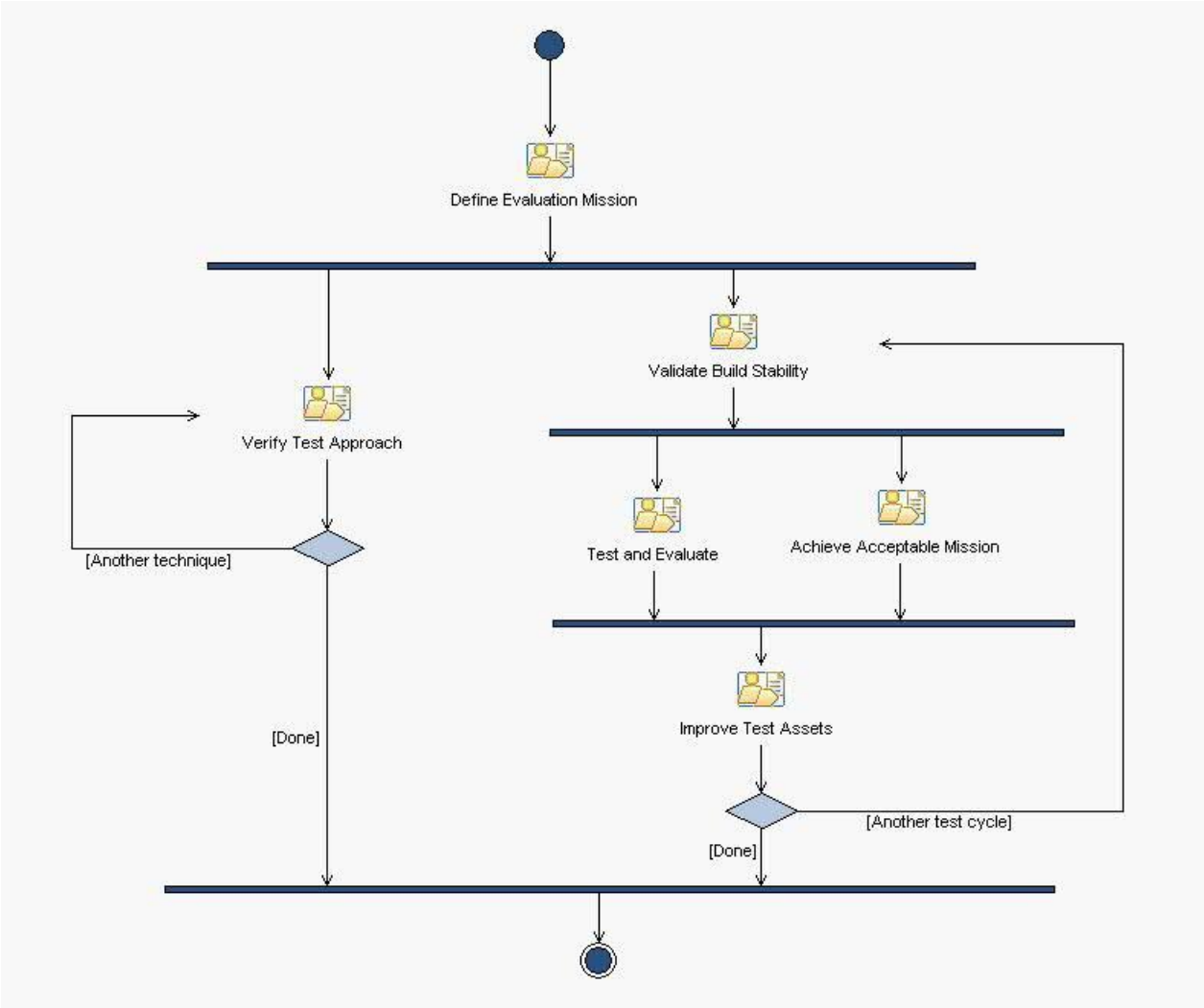
Za konec sledi diagram poteka aktivnosti pri procesu testiranja pri razvoju programske opreme v manjših razvojnih skupinah (Slika 11). Za lažje razumevanje pa še legenda (Slika 10), ki razloži pomen posameznih simbolov.


**Slika 10: Razlaga simbolov, uporabljenih v procesu testiranja**




**Slika 11: Diagram poteka procesa testiranja**

Za primerjavo podajam še diagram aktivnosti, kot ga lahko najdemo v RUP za manjše projekte (Slika 12).




**Slika 12: Diagram procesa testiranja iz RUP**

Za preprosto sliko se skriva dokaj kompleksen sistem opravil in korakov ter temeljita razdelitev nalog med nosilce testiranja. Poleg tega diagrama se s klikom na posamezno aktivnost pokažejo novi diagrami, ki prikazujejo vhodne dokumente, nosilce aktivnosti in opravila, ki so jim določena itd. Za lažje razumevanje strukture diagrama je spodaj še tabela z vsemi naštetimi aktivnostmi, pod vsako aktivnostjo so zapisana opravila, ki spadajo vanjo, znotraj opravila pa je določena odgovorna oseba, vhodni in izhodni podatki. S klikom na posamezni element znotraj opravil se nam odprejo podrobni podatki o tem elementu. Podrobnosti lahko zajemajo namen, razmerja, lastnosti, glavni opis ter ostale informacije. S svojim diagramom sem poskušal čimbolj strniti celoten model v en diagram ter z opisno obliko predstaviti, kako naj bi se ta model uporabljal.

# 11. ZAKLJUČEK

Skupaj s pojavom prvih računalnikov so nastali tudi prvi preprosti programi. Takrat, ko je bil nameščen in pognan prvi program, se je začelo tudi obdobje testiranja oziroma preizkušanja. Zaradi preprostih funkcij, ki so jih izvajali takratni programi, se testiranju ni namenjalo tako velike pozornosti. S konstantnim razvojem računalništva je tehnologija testiranja napredovala iz meseca v mesec. Računalniki so bili sposobni izvajanja vedno več operacij na sekundo ter hranjenja vedno več podatkov. Tako so nastale tudi večje potrebe po zmogljivi in zanesljivi programski opremi, ki bo omogočala lažje delo na različnih področjih (vojska, industrija, bančništvo, marketing itd.).

Z razvojem programske opreme so prišle tudi zahteve po večji kakovosti. Le-to pa lahko zagotovimo le z natančnim načrtovanjem ter intenzivnim testiranjem. S prihodom interneta je prišlo na dan vprašanje varnosti. Potrebno je bilo zagotoviti sisteme, ki so bili varni pred nezaželjenimi vdori, zaradi varovanja delikatnih podatkov. Lahko si predstavljamo, kakšno katastrofo bi pomenilo, če bi nekdo vdrl v bančni sistem ter si prenakazal denar iz tujih računov na svojega. Zaradi teh dejavnikov se je začelo testiranju programske opreme namenjati dosti več pozornosti. Pri načrtovanju razvoja programske opreme se je to začelo kazati z ločevanjem testiranja od razvojne faze. Večja podjetja so začela namenjati več sredstev v testiranje programske opreme. Nastala je potreba po kadru, ki je izučen za takšno dejavnost, zato so se v podjetjih oblikovale ekipe, posebej zadolžene za testiranje programske opreme. Ta dejavnost je z leti razvoja še pridobivala na pomenu, zato zdaj pri marsikaterem večjem projektu lahko stroški testiranja predstavljajo večino stroškov celotnega projekta.

Sam sem imel priložnost spoznati, kako poteka razvoj opreme v podjetju, kjer sem opravljal obvezno prakso ter kasneje delo preko študentskega servisa. Tam sem opazil pomanjkljivosti v procesu testiranja. Po mojem mnenju je bil razlog v premajhni količini sredstev, namenjenih temu procesu, ter pomanjkanju kadra, specializiranega za to področje. Sam sem se znašel v vlogi testerja, vendar pa sem imel včasih občutek, da moje delo ni tako učinkovito kot bi lahko bilo. Ko sem za potrebe diplomske naloge preučeval literaturo in druge strokovne vire s področja testiranja, sem se seznanil z mnogimi vidiki testiranja, ki mi prej kljub delu na tem področju niso bili znani. S preučevanjem različnih virov sem dobil boljši pregled nad tem kaj testiranje dejansko je, vendar kljub vsemu je to le vrh ledene gore, ki pod sabo skriva celo znanost. Verjamem, da bi lahko vsako podjetje hitreje in učinkoviteje razvijalo programsko opremo, če bi namenilo večjo pozornost temu področju.

Prvo, kar bi bilo potrebno spremeniti v takšnem podjetju, je, da zaposlijo primeren kader, ki bo sestavljal ekipo, zadolženo za testiranje. Temu kadru bi prav tako morali nameniti sredstva za dodatno izobraževanje na področju testiranja. V tej ekipi se jasno določi hiearhijo ter vloge, ki jih posamezni član prevzame. Delo te ekipe se olajša tudi s kvalitetnim načrtovanjem programske opreme, ki že preprečuje nastajanje nepotrebnih napak pri razvoju. Še pred tem je potrebno izdelati celoten proces testiranja, ki bo služil kot osnova za načrtovanje testiranja pri posameznih projektih. Ta proces sem tudi izdelal kot rezultat moje

diplomske naloge. Kot je bilo že prej omenjeno, je ta proces zasnovan za primer slapovnega ali zaporednega razvoja programske opreme. V prihodnosti bi bilo smiselno ta proces testiranja uskladiti z različnimi modeli, ki se uporabljajo pri razvoju programske opreme (iterativni model, inkrementalni model, prototipni model …), ga ustrezno dopolniti ter bolj podrobno razdelati elemente, ki ga sestavljajo. V takšni obliki, bi ga potem brez težav uporabili v praksi.

Zaradi velike konkurence podjetja vsakodnevno izgubljajo posle, v katerih se obrača mnogo denarja. V takem svetu bo »preživelo« samo podjetje, ki naredi zares kvalitetno programsko rešitev ter jo seveda primerno oglašuje. Včasih podjetja namenijo veliko več denarja oglaševanju izdelka kot pa kvalitetnemu preizkušanju le-tega. Vsi pa vemo, da je najboljša reklama tista reklama, ki jo naredijo zadovoljni uporabniki. Dostikrat pride zaradi pritiska naročnikov ali konkurence do izida pomanjkljive programske opreme, ki naredi podjetju več škode kot koristi (primer: Microsoftov operacijski sistem Windows Vista). Zato je pomembno da se podjetja učijo na svojih napakah ter jih poskušajo čimbolje odpraviti, tako v organizacijskem smislu kot v smislu razvoja programske opreme. Slednje je mogoče odkriti pravočasno ter potem tudi odpraviti, le s skrbno načrtovanim in izpeljanim testiranjem.

## 12. LITERATURA IN VIRI

[1] M. Fewster, D. Graham, »Software Test Automation,« *Effective use of test execution tools (str. 143—174)*, Great Britain, 1999

[2] J. Tian, »Software Quality Engineering«, *Testing, Quality Assurance, and Quantifiable Improvement (str. 67—101)*, New Jersy: John Willy & Sons, Inc., 2005

[3] D. Vavpotič, *Razvoj informacijskih sistemov*, Visokošolski strokovni študij, Študijsko gradivo, verzija 3.0, Univerza v Ljubljani, 2010

[4] AppLabs, *A test processfor for all Life Cycles*, Last updated april 2008. Dostopno na: http://www.applabs.com/html/TestProcessforallLifecycles_511.html

[5] Wikipedia, the free encyclopedia. Dostopno na: http://en.wikipedia.org/wiki/Software_testing, http://sl.wikipedia.org/wiki/Wiki

[6] Software Testing — Testing Tutorials, Testing Tools, Testing Softwares, Testing jobs, Testing Techniques. Dostopno na: http://www.onestoptesting.com/

[7] White Box Testing. Dostopno na: https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html

[8] Software Quality Assurance Glossary. Dostopno na: http://www.aptest.com/glossary.html

[9] Anteater. Dostpno na: http://aft.sourceforge.net/

[10] Software Testing Geek – Testing Tools. Dostopno na: http://www.testinggeek.com/index.php/testing-tools

[11] When to stop testing? Dostopno na: http://csqa.info/when_to_stop_testing

[12] Testing the aplication. Dostopno na: http://msdn.microsoft.com/en-us/library/ms182409.aspx

[13] Six software test planning tips for better QA testing ROI. Dostopno na: http://searchsoftwarequality.techtarget.com/news/column/0,294698,sid92_gci1351618,00.html

[14] QUnit – jQuery JavaScript Library. Dostopno na:
http://docs.jquery.com/QUnit

[15] ClearSpecs16V01_GrowthOfSoftwareTest.pdf. Dostopno na:
http://www.clearspecs.com/downloads/ClearSpecs16V01_GrowthOfSoftwareTest.pdf