UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

DIPLOMSKO DELO

MIHA JAKLIČ

UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

DIPLOMSKO DELO

TESTIRANJE PROGRAMSKIH REŠITEV

Ljubljana, marec 2010 MIHA JAKLIČ

IZJAVA

Študent/ka **MIHA JAKLIČ** izjavljam, da sem avtor/ica tega diplomskega dela, ki sem ga napisal/a pod mentorstvom **dr. ANDREJA KOVAČIČA**, in da dovolim njegovo objavo na fakultetnih spletnih straneh.

V Ljubljani, dne_____ Podpis: _____

KAZALO

UVOD

Podjetja so v današnjem času prisiljena za svoj obstoj biti boljša, hitrejša in uspešnejša od konkurentov. Svoje poslovne procese morajo hitro prilagajati razmeram na trgu, iskati morajo možnosti nižanja stroškov, izboljšanja kakovosti proizvodov ter skrbeti za čim boljši odnos s strankami.

Zaradi želje po ustvarjanju in ohranitvi dobrih odnosov med podjetjem in strankami so najnovejši trendi usmerjeni v pridobivanje in poenotenje informacij o strankah za vse poslovne funkcije v podjetju, ki imajo stik z njimi. Uvedba sistema za upravljanje odnosov s strankami (CRM) omogoča uporabniku vpogled v strankine potrebe in želje, kar pozitivno vpliva na odnos med njima ter dolgoročno tudi na poslovni izid podjetja.

Razvoj programske opreme se prav tako odvija hitro in je velikokrat omejen s časovnimi roki. Zaradi kratkih časovnih rokov in pa seveda človeškega faktorja zmotljivosti prihaja med razvojem programske opreme do številnih napak. Glede na to, da stroški odpravljanja napak logaritemsko naraščajo s časom odkritja napake, je za podjetje pomembno, da so le-te odkrite v zgodnejših fazah razvoja.

Proces testiranja se precej razlikuje po posameznih fazah razvoja programske rešitve. V zgodnjih fazah preverjamo pravilnost dokumentacije, ki definira lastnosti programske opreme ter kontroliramo, ali razvoj programske rešitve poteka v skladu z zahtevami. V naslednjih fazah pa poteka preverjanje delovanja same aplikacije, in sicer z izvajanjem logičnih testov, ki pokrijejo čim večje število možnih scenarijev. V zadnjih fazah razvoja se testira predvsem napake vmesnikov, odzivnost programa ter napake pri branju podatkov iz baze.

Septembra 2008 sem kot študent postal del projektne ekipe podjetja Marand d.o.o.. Naša ekipa se ukvarja z razvojem sistema za upravljanje odnosov s strankami in podporo informacijskim procesom Telekoma Slovenije. Ekipi sem se pridružil kot tester tega sistema. Namen diplomske naloge je, z analizo teorije s področja testiranja, utrditi praktično znanje, ki sem ga pridobil tekom testiranja v podjetju Marand d.o.o.. Poleg tega želim preučiti različne tehnike testiranja skozi vse faze razvoja informacijske rešitve, saj sem se projektni ekipi pridružil šele v fazi testiranja modulov in scenarijev ter tako zamudil začetne faze razvoja in testiranja.

Ob začetku pisanja diplomske naloge sem si glede na namen diplomske naloge postavil naslednje cilje:

- predstaviti nekaj definicij testiranja ter pojasniti, kaj je testiranje in kaj so napake,
- na kratko predstaviti glavne gradnike v razvoju programske opreme,
- predstaviti različne pristope in tehnike testiranja po posameznih fazah razvoja informacijske rešitve,

- opisati življenjski cikel odpravljanja napake,
- predstaviti ekonomske koristi in stroške izvajanja procesa testiranja in
- na praktičnem primeru predstaviti potek testiranja sistema za upravljanje odnosov s strankami.

Diplomska naloga je poleg uvoda in zaključka razdeljena na sedem poglavij.

V prvih dveh poglavjih bom predstavil definicije testiranja in napake ter na kratko pojasnil oba pojma.

V tretjem poglavju bom opisal ključne gradnike razvoja programske opreme. To temo sem v diplomsko nalogo vključil zato, ker je poznavanje procesa razvoja programske opreme in ključnih gradnikov v tem razvoju nujno potrebno za izvajanje testiranja.

Naslednje poglavje bo namenjeno predstavitvi različnih strategij in pristopov k testiranju skozi vse faze razvoja programske opreme.

V petem poglavju bom opisal življenjski cikel odpravljanja napake in pomen ustrezno napisanega poročila o napaki, v šestem poglavju pa bom analiziral in predstavil prednosti in slabosti, ki jih prinese vključitev testiranja v proces razvoja programske rešitve.

V okviru sedmega poglavja bo predstavil sistem za upravljanje odnosov s strankami, ki ga s sodelavci podjetja Marand Inženiring d.o.o. razvijamo za podjetje Telekom Slovenije d.d.. Predstavil bom glavne faze projekta CRM Telekom in opisal potek testiranja ter evidentiranja napak na tem projektu.

1 KAJ JE TESTIRANJE?

Testiranje je aktivnost, ki je tesno povezana z razvojem informacijskega sistema. Razvoj kvalitetne informacijske rešitve zahteva, da testiranje spremlja razvoj informacijskega sistema od začetka do konca.

V grobem lahko rečemo, da je testiranje proces iskanja, identifikacije, evidentiranja napak, pomanjkljivosti in nepravilnega delovanja informacijske rešitve. S testiranjem se ugotavlja morebitno odstopanje informacijske rešitve od zahtev stranke ter hkrati odkrije čimveč nepredvidljivih napak. Iz neskončne množice kombinacij je potrebno izluščiti najverjetnejše vzorce in le-te temeljito preveriti. Sicer pa je testiranje pomemben del razvoja informacijskega sistema, saj na splošno velja, da se za testiranje porabi več kot polovica časa, namenjenega razvoju informacijske rešitve (Križnik, 2002, str. 9).

Testiranje zahteva sodelovanje stranke, ki je naročnik projekta, razvijalcev in posebej za testiranje usposobljenih inženirjev. Testni inženirji najprej preučijo zahteve in želje bodočih uporabnikov informacijske rešitve in na osnovi tega pripravijo testne primere, ki

vsebujejo postopek in pričakovan rezultat. Testni primeri morajo zagotoviti zajem vseh pomembnejših scenarijev in predvideti čimveč možnih zapletov. Na koncu morajo napisati tudi poročilo izvedenih testov.

Skupino ljudi, ki izvaja razne funkcije na programski opremi in jih dokumentira ter poroča, lahko na kratko imenujemo testerji. Testerji prihajajo iz različnih družbenih okolij. Kljub temu da se testerji med seboj razlikujejo, pa se večina ljudi strinja, da testerji razmišljajo drugače ter da razmišljajo negativno. Radi se pritožujejo in obožujejo posredovanje slabih informacij. Vendar pa je to le splošno mnenje. Testerji se ne pritožujejo, temveč ponujajo dokaze, da morda ni vse tako popolno kot zgleda na prvi pogled (Kaner, Bach & Pettichord, 2001, str. 11).

Razlika med odličnim in povprečnim testiranjem je v načinu razmišljanja. Kvaliteta testiranja je odvisna od odločitev, sposobnosti razlage opaženega ter sposobnosti posredovanja jedrnate zgodbe o opaženem. Dober tester mora razmišljati kreativno, kritično, praktično ... (Kaner et al., 2001, str. 15).

Zavedati se moramo, da testiranje samo po sebi ne more odpraviti posledic slabega razvoja informacijskega sistema, vendar pa lahko zmanjša posledice. Kvalitetno, dobro načrtovano in pravilno izvedeno testiranje predstavlja dokaj visoke stroške testiranja, nekateri avtorji menijo, da celo več kot polovica celotnih stroškov, vendar pa je izognitev testiranju praviloma še veliko dražja. Končni rezultat po kvalitetnem testiranju je zanesljivo in nemoteno delovanje informacijske rešitve po prehodu v produkcijo, kar posledično znižuje stroške tehnične podpore in vzdrževanja informacijske rešitve (Knafeljc, Drnovšek, & Kožman, 2006, str. 2).

2 KAJ JE PROGRAMSKA NAPAKA?

Programska napaka (angl. *Bug*) je splošen izraz za napako v računalniškem programu ali sistemu, katere posledica je nepravilen oz. nepričakovan rezultat. Večina programskih napak je posledica človeške napake, in sicer nepravilnosti v programski kodi, ki ni bila ugotovljena s strani programerja.

V panogi razvijanja programske opreme na splošno velja, da je v programu prisotna napaka, če velja vsaj eno od naslednjih petih pravil (Patton, 2005, str. 16):

1. Program ne naredi nečesa, kar v produktni specifikaciji piše, da bi moral narediti.
2. Program naredi nekaj, za kar je v produktni specifikaciji navedeno, da se ne sme zgoditi.
3. Program naredi nekaj, kar v produktni specifikaciji ni omenjeno kot naloga programa.
4. Program ne naredi nečesa, kar sicer v produktni specifikaciji ni navedeno, vendar pa bi moralo biti.

5. Programska rešitev je kompleksna za razumevanje in uporabo, se odziva počasi oz. kako drugače ni primerna uporabi.

Za lažje razumevanje teh pravil bom navedel **primer možnih napak kalkulatorja**.

Specifikacija za kalkulator gotovo vsebuje navedbo, kaj se mora zgoditi pri kliku na posamezno tipko. V kolikor se ob kliku na + ne zgodi nič, je to primer napake iz prve točke. Če dobimo napačen rezultat, je to tudi napaka, ki spada pod točko ena.

V specifikaciji je verjetno prav tako navedeno, da kalkulator ne sme nikoli zmrzniti ali se zakleniti. Če se ob določeni kombinaciji tip kalkulator preneha odzivati, je to posledica napake, ki jo uvrščamo pod točko dve.

Recimo, da testiramo kalkulator, ki glede na produktno specifikacijo omogoča seštevanje, odštevanje, množenje in deljenje. Med testom je ugotovljeno, da kalkulator omogoča tudi izračun kvadratnega korena. Kljub temu da se to morda zdi koristna funkcija kalkulatorja, pa to uvrščamo med napake iz točke tri, saj lahko ta nepredvidena funkcija povzroči kake druge napake.

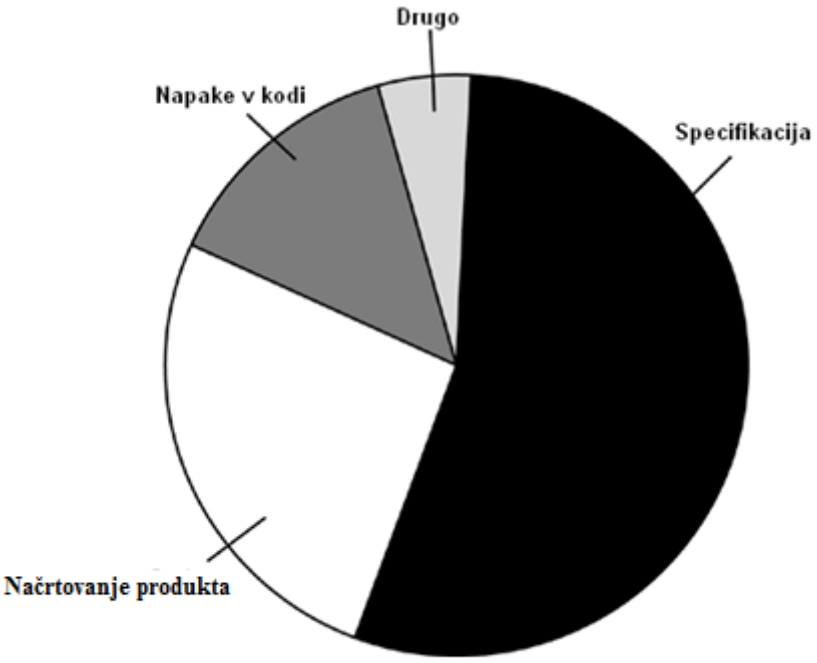
V okviru četrtega pravila so zajete funkcionalnosti programske rešitve, katerih se med izdelavo produktne specifikacije nismo spomnili. Med testiranjem kalkulatorja smo npr. ugotovili, da kalkulator, v kolikor je baterija skoraj izpraznjena, vrača napačen rezultat. Nikjer ni zapisano, kako naj se kalkulator obnaša v taki situaciji, saj je bila situacija praznjenja baterije med izdelavo produktne specifikacije spregledana. Ob upoštevanju pravila iz točke štiri je tudi to napaka.

V pravilu pet so zajete vse pomanjkljivosti programa. Ker smo kot testerji prvi uporabniki programske rešitve, se moramo postaviti v kožo končnega uporabnika. Če med testom naletimo na nekaj, kar se nam iz kakršnega koli razloga ne zdi prav, je to napaka. Na primeru kalkulatorja so to lahko premajhne tipke, neustrezen položaj tipk, neustrezen kontrast zaslona … Vse te napake so posledica upoštevanja pravila iz pete točke.

2.1 Zakaj nastanejo napake?

Na splošno velja prepričanje, da je neustreznost programske opreme praviloma posledica napak v programski kodi oz. posledica programiranja. Številne študije, ki so zajemale tako majhne kot tudi velike projekte, so ovrgle to prepričanje z ugotovitvijo, da je glavni vzrok za programske napake v specifikaciji. Na Sliki 1 so prikazani glavni povzročitelji napak v programski opremi.

Slika 1: Različni povzročitelji programske napake



Vir: R. Patton, Software Testing (2th ed.), 2005, str. 16.

Obstaja več razlogov, zakaj je specifikacija glavni povzročitelj napak. Pogosto je težava v tem, da specifikacija sploh ni napisana. Drugi razlogi so lahko še, da specifikacija ni dovolj natančna oz. da se neprestano spreminja ali pa ni pravilno predstavljena razvojni ekipi.

Naslednji velik izvor napak je načrtovanje. To je planiranje produkta s strani programerjev. To načrtovanje lahko primerjamo z načrti arhitektov pri gradnji zgradb. Razlogi za napake so tako kot pri specifikaciji: slaba komunikacija, hitenje, spreminjanje …

Napake v programski kodi so najpogosteje posledica kompleksnosti programa, slabe specifikacije, časovne stiske ali pa površnosti. Pomembno je poudariti, da so mnoge napake, ki na prvi pogled delujejo kot posledica napake v programski kodi, v bistvu posledica slabe dokumentacije in načrtovanja. Pogosto slišimo programerje reči: »Oh, tako naj bi to naredil. Če bi mi kdo to prej povedal, ne bi napisal kode na tak način.«

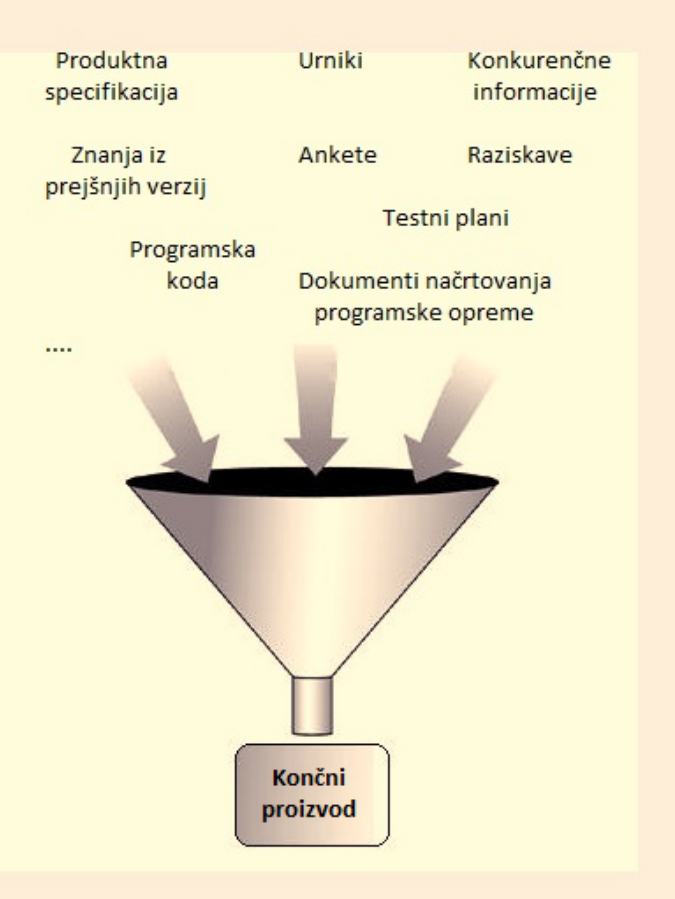
Ostale napake pa so lahko posledica napačne predstave o programu, za katere se izkaže, da niso napake. Obstajajo tudi napake, ki so posledica kake druge napake in bodo v okviru te tudi odpravljene. Te napake zajemajo majhen odstotek vseh prijavljenih napak (Patton, 2005).

3 KLJUČNI GRADNIKI RAZVOJA PROGRAMSKE OPREME

Za testiranje programske rešitve je nujno potrebno odlično poznavanje celotnega procesa razvoja programske opreme. Ustvarjanje nove programske rešitve lahko vključuje tudi tisoč ali več tisoč članov projektne ekipe, ki imajo med seboj razporejene naloge, vendar pa stremijo k skupnemu cilju. Kaj člani projekta delajo, kako sodelujejo med seboj in kako sprejemajo odločitve, vse to je del procesa razvoja programske opreme (Patton, 2005).

V nadaljevanju bom predstavil potrebne gradnike izgradnje programske rešitve (Slika 2) ter nekaj osnovnih pristopov razvoja, ki se uporabljajo danes.

Slika 2: Različni gradniki programske opreme



Vir: R. Patton, Software Testing (2th ed.), 2005, str. 24.

Posamezne komponente, ki so sestavni del gradnje programske rešitve, lahko strnimo v nekaj glavnih kategorij.

3.1 Zahteve stranke

Programsko opremo izdelujemo po naročilu strank. Seveda mora naš produkt ustrezati željam te stranke oz. skupine, ki je naročnik našega izdelka. Razvojna ekipa mora na začetku projekta zbrati informacije o željah stranke, in sicer v obliki anket bodočih uporabnikov, pogovorov z managementom, fokus skupin, izkušenj iz prejšnjih projektov, ocen raznih revij … Na podlagi študij in dodatnih razgovorov s stranko so ugotovljene osnovne značilnosti programske rešitve.

3.2 Specifikacije (tehnični podatki)

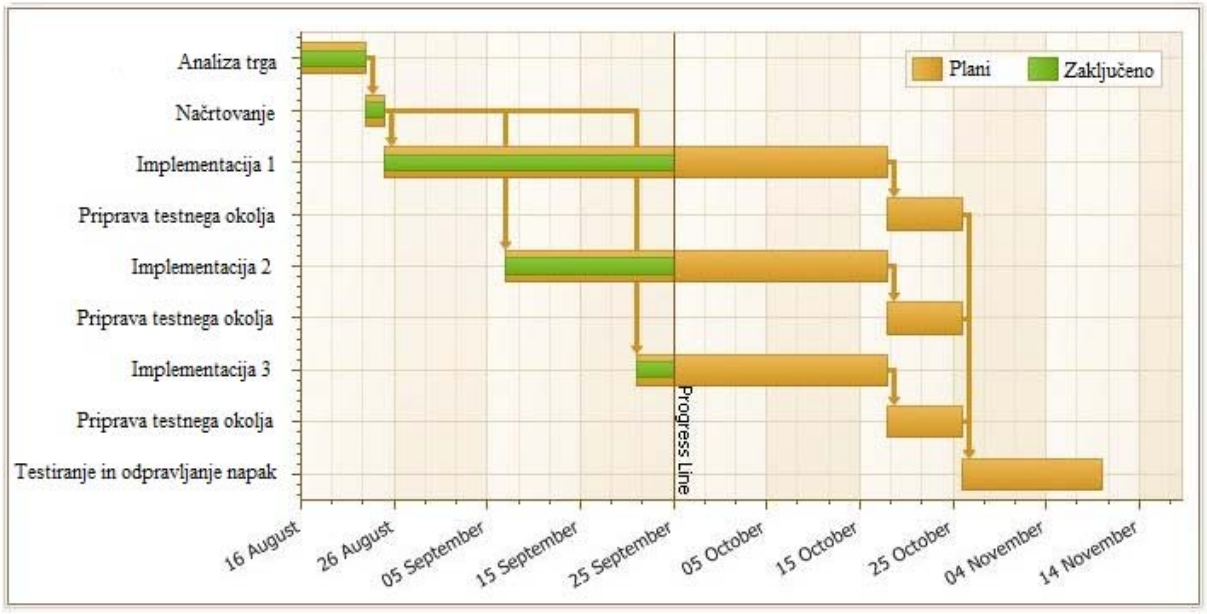
Rezultat analize zahtev strank so torej samo okvirni podatki o predstavi produkta v očeh stranke. Specifikacija pa zajema vse te informacije ter tudi podrobne lastnosti programske rešitve, njenih funkcij in izgleda.

Pomembnost specifikacije močno varira, glede na posamezno projektno skupino in glede na vrsto projekta. Nekatera razvojna podjetja, predvsem tista, ki razvijajo izdelke za vlado, letalske družbe, finančne in zdravstvene ustanove, kreirajo izredno natančno specifikacijo izdelka. Takšna specifikacija se med razvojem programske opreme naj ne bi spreminjala, oz. se spremeni le v ekstremnih pogojih. Vsak član razvojne ekipe mora poznati postavke specifikacije in se jih tudi držati. Obstajajo pa tudi razvojne ekipe, ki prisegajo na fleksibilnost in zato izdelajo specifikacijo bolj »ohlapno« ali pa je sploh ne izdelajo. Takšen razvoj je sicer bolj prilagodljiv, vendar pa se v takšnih pogojih pogosto dogaja, da so posamezni člani razvojne ekipe med seboj neenotni in neusklajeni.

3.3 Terminski načrt

Eden izmed ključnih delov pri ustvarjanju programske opreme je terminski načrt. Ko projekt pridobiva na velikosti in kompleksnosti ter je v razvoj programske rešitve vključenih veliko ljudi, postanejo nujno potrebni mehanizmi, s katerimi sledimo napredku projekta. Dober terminski načrt naj bi vseboval seznam zadolžitev po posameznih zaposlenih, tem zadolžitvam pa naj bi bil dodeljen približen čas izvedbe posamezne naloge. Namen terminskega načrta je vedeti, katere naloge so že bile opravljene ter koliko časa je še potrebnega za dokončanje posamezne naloge.

Slika 3: Prikaz projektne naloge s gantogramom



Vir: J. M. Bucknall, The seat of your Gantts, 2007.

3.4 Dokumenti načrtovanja programske opreme

Obstaja splošno mnenje, da se program naredi tako, da programer enostavno sede na stol in začne s pisanjem kode. Z izjemo kreiranja zelo majhnih in enostavnih programov je to mnenje popolnoma zgrešeno. Tako kot je pred začetkom gradnje stavbe nujno potreben gradbeni načrt, so tudi pred začetkom pisanja kode potrebni načrti.

Dokumenti, ki jih kreirajo programerji, se med seboj razlikujejo - odvisno od podjetja, narave projekta in projektne ekipe. Skupni namen teh dokumentov pa je organizacija kode, ki bo napisana.

Nekaj najbolj splošnih dokumentov načrtovanja programske opreme (Mallory, 2002):

- **Arhitektura programske opreme**: To je dokument, ki opisuje celoten načrt programske rešitve, vključno z opisom pomembnejših delov sistema ter povezavo med njimi.
- **Diagram toka podatkov**: Diagram prikazuje, kako se podatki prenašajo med različnimi deli sistema.
- **Diagrami poteka**: To je slikovni opis programske logike. Diagrami poteka dandanes niso popularni, vendar pa je programiranje s pomočjo teh diagramov veliko enostavnejše.

- **Komentirana koda**: V kolikor razvijalci beležijo komentarje v programsko kodo, je to zelo koristno za programerje, ki bodo vzdrževali oz. kakor koli drugače prišli v stik s kodo, saj bo tako lažje razumljivo, kaj in kako deluje koda.

3.5 Dokumenti testiranja

Tako kot programerji morajo tudi testerji načrtovati in dokumentirati svoje delo.

Najbolj pomembni dokumenti beleženja testnih rezultatov so (Wikipedia, 2009):

- **Plan testiranja**: Opisuje celoten potek preverjanja skladnosti programske opreme s produktno specifikacijo in zahtevami stranke. Vključuje »cilje kakovosti«, potrebe po virih, urnike, zadolžitve, metode itd.
- **Testni primeri**: So lista specifičnih scenarijev, ki bodo testirani. Tu je po korakih predstavljen celoten proces testiranja posameznega scenarija.
- **Poročilo o napaki**: V teh poročilih so prestavljeni problemi oz. napake, ki so bile odkrite med izvajanjem testnih primerov. Poročila so lahko pisana na papir, vendar pa so večinoma zabeležena v bazi podatkov.
- **Testno orodje**: Programi, ki so nam v pomoč pri testiranju, morajo prav tako biti zabeleženi in na kratko predstavljeni.
- **Testne statistike** in povzetki testov izražajo napredek tekom testiranja.

4 STRATEGIJE TESTIRANJA

V več kot 35 letih testiranja programskih rešitev so se uveljavili štirje različni pristopi uspešnega testiranja. Ti štirje pristopi so (Gao, Tsao & Wu, 2003, str. 87):

- statično testiranje,
- strukturno testiranje ali metoda bele skrinjice,
- funkcionalno testiranje ali metoda črne skrinjice,
- testiranje zmogljivosti.

4.1 Statično testiranje

Kot sem že navedel v enem od prejšnjih poglavij, približno 80% vseh napak izvira iz zgodnjih faz razvoja programske opreme. Vendar pa testiranje kode v začetnih fazah ni možno, saj je ta napisana kasneje. V zgodnjih fazah testiramo dokumentacijo. Razvoj programske rešitve se začne, nadaljuje in konča z dokumentacijo. Dokumentacija v

zgodnjih fazah definira lastnosti programske opreme, v kasnejših fazah pa je dokumentacija uporabniški priročnik.

Na pregledovanje in izpopolnjevanje dokumentacije se pogosto gleda kot na nekaj, kar se ureja na koncu projekta, vendar pa je to razmišljanje napačno. Več časa in napora, vloženega v razvoj dobre dokumentacije, pomeni veliko večjo možnost, da bo programer na podlagi le-te napisal dobro kodo.

Statično testiranje je v bistvu pregledovanje in preučevanje ustreznosti dokumentov in odkrivanje napak, preden se te dejansko zgodijo.

4.2 Metoda bele skrinjice

Za testiranje po metodi bele skrinjice potrebujemo izvorno in izvršljivo kodo. Metodo uporabljajo predvsem razvijalci, ki razvijajo nov sistem ali delajo nadgradnjo obstoječega, saj imajo tako dostop do izvorne kode (Beizer, 1990).

Najučinkovitejši test po metodi bele skrinjice je izveden s sodelovanjem programerjev in testerjev. Programer ima dostop do izvorne kode, poseduje znanje programske specifikacije, standarde programiranja ter logiko izvorne kode. Programer najbolj razume, kaj bo rezultat kode in kako jo izboljšati. Tester pa na drugi strani pozna tehnike, s katerimi odkrijemo, če lahko bodoči uporabnik povzroči napako v programu. Ravno zaradi kombiniranja vseh teh znanj je metoda bele skrinjice najbolj učinkovita metoda odkrivanja napak.

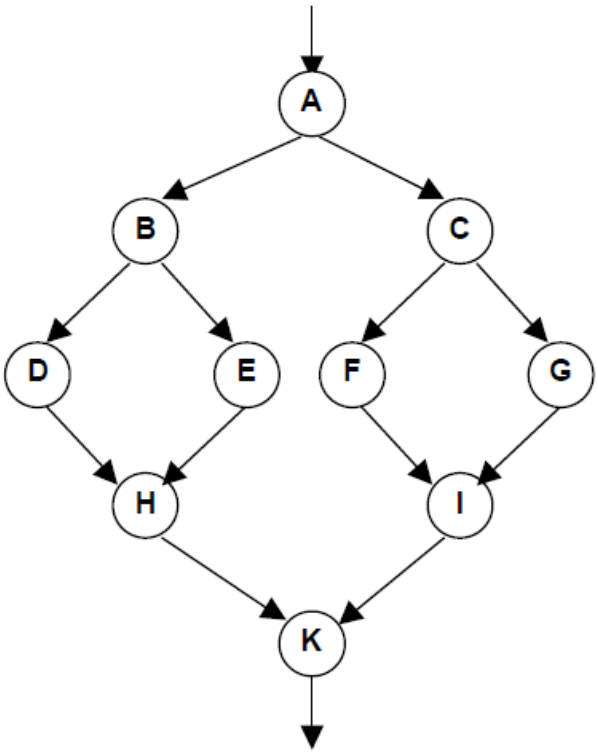
Prvi pogoj za testiranje po tej metodi je obstoj programskih zahtev, primerov uporabe, podatkov in izvorne kode. Programerji so običajno osredotočeni, da pišejo kodo tako, da bo delovala pravilno glede na aktivnosti primerov uporabe. Na ta način izvajajo tudi teste po vnaprej predvidenih scenarijih (selektivni testi). Testerji pa so pri testiranju bolj kreativni in izvajajo logične teste, ki pokrijejo več možnih scenarijev. Tak način testiranja omogoča odkritje večjega števila napak v zgodnejših fazah razvoja programske opreme, kar močno zmanjša stroške odpravljanja le-teh, kar je tudi poslovna motivacija testiranja po tej metodi.

Obstaja več metod bele skrinjice, ki nam omogočajo, da nam ni potrebno testirati vseh možnih poti. Glede na predpisano stopnjo pokritosti testiranja določijo testerji testne scenarije. Izvajanje vsakega testnega scenarija povzroči, da program izvede določen ukaz, kar se odraža v pokritosti specifične poti v grafu toka. S spreminjanjem pogojev lahko izbiramo različne poti.

Za testerja je bistveno, da pri pripravi testnih scenarijev upošteva logični tok aplikacije, ki pokaže, kateri del aplikacije se izvede ob določeni akciji. Pri tem je pomembno sodelovanje s programerjem, ki pozna logiko izvorne kode. Upoštevanje logičnega toka

omogoča določitev takšnih testnih scenarijev, ki pokrivajo različne poti, tako pa se doseže želena stopnja pokritosti testiranja s precej nižjim številom testnih scenarijev kot sicer. Logični tok aplikacije lahko predstavimo v grafu toka (Slika 4).

Slika 4: Graf toka



Vir: K. Ross, Practical Guide to Software System Testing, 1999, str. 94.

Graf toka sestavljajo (Ross, 1999, str. 94):

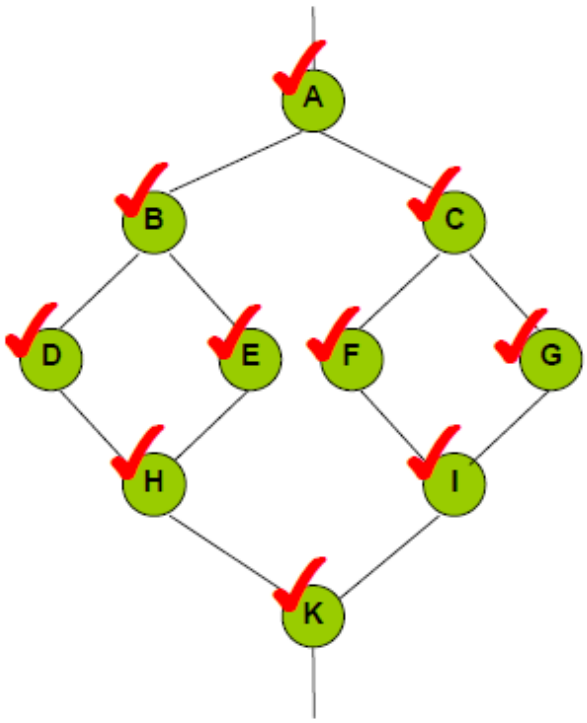
- **vozlišča** (trditve, ki so lahko zajete med izvajanjem programa),
- **robovi** (predstavljajo poti, po katerih logika dovoljuje aplikaciji prehod med vozlišči),
- **deljeni vozli** (vozlišča, ki prehajajo v več vozlišč),
- **robovi vej** (robovi, ki se nahajajo za vejami),
- **poti** (predstavljajo vse možne poti prehajanja med vozlišči; poti potekajo po robovih).

V nadaljevanju bom predstavil metode bele skrinjice, ki za izdelavo testnih scenarijev izkoriščajo logični tok aplikacije.

4.2.1 Pokritost trditev

S posameznim testnim scenarijem pokrijemo določeno število vozlišč in tako pokrijemo določen del trditev. 100% pokritost trditev bi dosegli, če bi bila po testiranju vsaka trditev iz grafa toka zajeta v vsaj enem testnem primeru.

Slika 5: Pokritost trditev



Vir: K. Ross, Practical Guide to Software System Testing, 1999, str. 95.

Če izberemo testni scenarij, katerega pot poteka med vozlišči A, B, D, H, K, smo s tem dosegli 50% pokritost trditev, saj smo s testom pokrili 5 od 10 vozlišč.

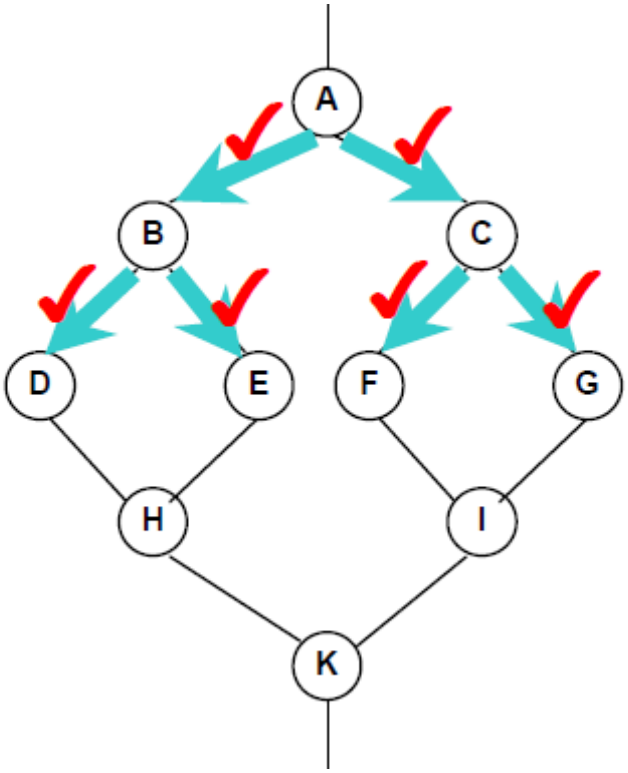
Na Sliki 5 vsakemu vozlišču pripada ena trditev, zato je pokritost trditev sorazmerna obisku vozlišč. Pokritost trditev pa se razlikuje od pokritosti vozlišč takrat, ko posamezno vozlišče združuje več trditev.

Hipoteza te tehnike pravi, da večja kot je pokritost izvedbe izvorne kode, manj napak bo odkritih v naslednjih fazah. V praksi pa velja, da so neizvedene vrstice izvorne kode programska časovna bomba, ki se bo nepričakovano sprožila v produkciji. Vprašanje ni, ali bo eksplodirala, temveč kdaj bo eksplodirala. (Everett et al., 2007, str. 108)

4.2.2 Pokritost vej

Pokritost vej je določena z deležem odločitvenih vej, ki so bile pokrite med testiranjem. Iz Slike 6 je razvidno, da v tem grafu toka obstaja šest odločitvenih vej. Če bi želeli doseči 100% pokritost vej, bi morali izvesti teste, kjer bi bila vsaka odločitvena veja vsebovana vsaj enkrat. Če se odločimo za pot A, B, D, H, K, smo s tem pokrili le dve od šestih odločitvenih vej in tako dosegli 33% pokritost.

Slika 6: Pokritost vej



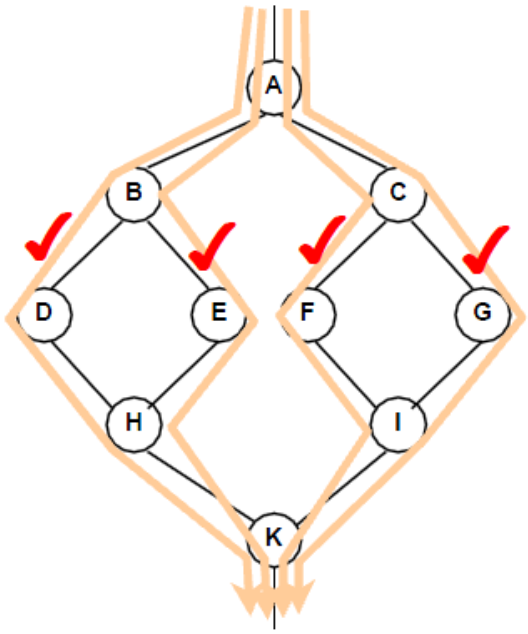
Vir: K. Ross, Practical Guide to Software System Testing, 1999, str. 95.

4.2.3 Pokritost poti

Pokritost poti si lahko predstavljamo kot delež poti, ki so bile zajete med testi. Za 100% pokritost poti moramo izvesti teste, v katerih bo vsaka pot zajeta vsaj enkrat.

V spodnjem grafu toka so možne 4 poti. Če s testnim scenarijem zajamemo pot ACFIK smo s tem obiskali eno od štirih poti ter tako dosegli 25% pokritost.

Slika 7: Pokritost poti



Vir: K. Ross, Practical Guide to Software System Testing, 1999, str. 96.

4.2.4 Tehnika sestavljenih pogojev

Tehnika sestavljenih pogojev razširja tehniko pokritosti vej z upoštevanjem različnih kombinacij pogojev, ki so zapisane z logičnimi operatorji IN, ALI in NE. Izziv je identificirati vse možne kombinacije testnih pogojev, katerih rezultat bo pravilno (*angl.* true) ali nepravilno (*angl.* false), za vsak pogoj posebej. Na podlagi rezultatov posamičnih pogojev, ki so povezani z logičnimi operaterji, dobimo še končni rezultat tipa pravilno/nepravilno. Vse možne kombinacije pogojev najdemo v pravilnostnih oziroma resničnostnih tabelah.

Primer dveh pogojev, ki sta povezana z logičnim operatorjem AND (IN):

(STAROST > 18 AND SPOL = M)

Da bi zagotovili 100% pokritost vseh možnih kombinacij med tema dvema pogojema, je kreirana pravilnostna tabela (Tabela 1).

Tabela 1: Pravilnostna tabela z dvema pogojema

STAROST	SPOL	Ustreznost pogoja AND
=18 false	=F false	=F false
=18 false	=M true	=F false
=19 true	=F false	=F false
=19 true	=M true	=T true

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 109.

Iz zgornje resničnostne tabele lahko razberemo, da enostavni test, ki vsebuje en pravilen in en nepravilen pogoj, ne zadošča sestavljenemu pogoju, zato je rezultat skupnega pogoja nepravilno (false).

Resničnostna tabela postane obsežnejša in kompleksnejša z uvedbo dodatnega pogoja, ki ga z logičnim operatorjem povežemo s prejšnjima pogojema (Tabela 2).

((STAROST > 18 AND SPOL = M) OR VIŠINA > 165 cm)

V tabeli 2 je prikazano, kako uvedba enega dodatnega pogoja poveča število možnih testnih scenarijev.

Tabela 2: Pravilnostna tabela s tremi pogoji

STAROST	SPOL	Ustreznost pogoja AND	VIŠINA	Ustreznost pogoja OR
=18 false	=F false	=F false	=165 false	=F false
=18 false	=M true	=F false	=165 false	=F false
=19 true	=F false	=F false	=165 false	=F false
=19 true	=M true	=T true	=165 false	=T true
=18 false	=F false	=F false	=185 true	=T true
=18 false	=M true	=F false	=185 true	=T true
=19 true	=F false	=F false	=185 true	=T true
=19 true	=M true	=T true	=185 true	=T true

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 109.

4.3 Metoda črne skrinjice

Testiranje po metodi črne skrinjice se uporablja v kasnejših fazah testiranja, in sicer se koncentriramo na iskanje okoliščin, v katerih se aplikacija ne obnaša v skladu s pričakovanji oz. specifikacijo. S pomočjo te metode odkrivamo predvsem napake vmesnikov, odzivnost programa, ustreznost zahtevanih funkcij ter napake pri branju podatkov iz baze (Jorgensen, 2008, str. 79).

Poznamo več različnih metod testiranja črne skrinjice (Everett et al., 2007, str. 113):

- določanje ekvivalentnih vhodnih skupin,
- analiza mejne vrednosti,
- analiza načrta,
- odločitvene tabele,
- diagrami stanj prehodov…

Za testiranje po tej metodi ne potrebujemo poznavanja same implementacije in interne logike programa, temveč je dovolj poznavanje funkcionalnosti. Na osnovi specifikacije se pripravijo testni primeri, in sicer se določijo vhodni in pravilni oz. pričakovani izhodni podatki.

4.3.1 Tehnika ekvivalentnih razredov

Bistvo te tehnike je identifikacija skupin vhodnih podatkov, ki bodo povzročili enako obnašanje sistema za vse vrednosti posamezne skupine.

Vzemimo za primer sistem za vnos pacientov, ki dovoljuje vnos pacienta med 1. In 99. letom starosti. Tester se lahko odloči, da bo testiral vnos vseh 99 vrednosti. Boljša rešitev pa je, da določi ekvivalentni razred starosti 1, 50 in 99 let ter sistem testira s 97% manj testnih vrednosti. Skupek vrednosti (1, 50 in 99) je torej ekvivalentni razred iz skupine vseh možnih vrednosti (1, 2, 3,…, 98, 99).

Tehnika testiranja po metodi črne skrinjice priporoča, da ekvivalentni razredi vsebujejo vrednost prve, srednje ter končne vrednosti izmed vseh vrednosti (Jorgensen, 2008, str. 97).

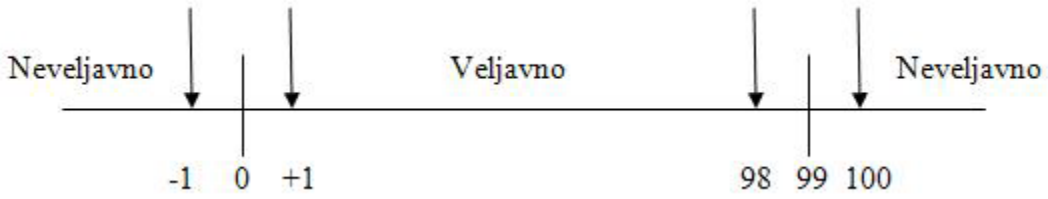
4.3.1.1 Tehnika analize mejnih vrednosti

Mejne vrednosti so za testiranje zanimive in zelo pomembne, saj se velik odstotek napak pojavi pri vnosu mejnih vrednosti oz. vrednosti okrog le-teh.

Analiza po tej tehniki se začne z določitvijo najmanjšega možnega prirastka, ki ga imenujemo epsilon. S pomočjo epsilona določimo +/- vrednost okrog začetne in končne vrednosti iz ekvivalentnega razreda (Everett et al., 2007).

Na Sliki 8 je prikazan primer analize mejnih vrednosti sistema za vnos pacientov, ki smo ga že uporabili v prejšnjem poglavju. Najmanjši prirastek (epsilon) je v tem primeru 1 leto, saj vnosno polje za nobeno starost ne dovoljuje vnosa nepolnih let. Pri testiranju tega sistema bi za spodnjo mejo izbrali vrednosti -1, 0 in +1, kjer sta -1 in +1 epsilon vrednosti okrog začetne vrednosti 0. Zgornja starostna meja je 99, njeni epsilon vrednosti pa 98 in 100.

Slika 8: Analiza mejnih vrednosti



Za zanesljivo testiranje po tej tehniki je priporočljivo, da se testira tudi srednjo vrednost ekvivalentnega razreda in njena epsilona, v našem primeru torej vrednosti 49, 50 in 51. V našem primeru ekvivalentni razred po tehniki mejnih vrednosti vsebuje vrednosti -1, 0, 1, 49, 50, 51, 98, 99 in 100.

4.3.1.2 Tehnika pričakovanih rezultatov

Medtem ko se obe tehniki črne skrinjice, ki sem jih predstavil do sedaj, nanašata na vhodne vrednosti, se tehnika pričakovanih rezultatov osredotoča na izhodne vrednosti. Prvi korak je, da v dokumentaciji zahtev poiščemo, kakšna so poslovna pravila oz. kakšni so pričakovani rezultati ali izhodne vrednosti. Na primeru bom pokazal, kakšna so poslovna pravila, ki so zapisana v dokumentaciji zahtev.

Vzemimo za primer aplikacijo, ki glede na starost pacienta ter znesek, ki ga zavarovanec mesečno plačuje za zdravniško oskrbo, izračuna nadomestilo za bolnišnični dan.

V dokumentaciji zahtev so poslovna pravila lahko zapisana v obliki tabele (Tabela 3).

Tabela 3: Poslovna pravila nadomestila za bolnišnični dan:

Vhodni podatek: **Starost pacienta**	**Vhodni podatek:** **Zavarovalnina**	**Izhodna vrednost/rezultat:** **Nadomestilo za bolnišnični dan**
0-6 let	50 €	50 €
7-17 let	75 €	100 €
18-35 let	100 €	150 €
36-49 let	125 €	300 €
50-74 let	200 €	350 €
75-99 let	250 €	400 €

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 115.

Ko smo seznanjeni s poslovnimi pravili in pričakovanimi rezultati, je potrebno sestaviti tabelo veljavnih kombinacij med vhodnimi podatki posameznih ekvivalentnih razredov in pričakovanimi rezultati.

V naslednjem koraku sestavimo tabelo, v katero poleg veljavnih vhodnih podatkov vnesemo tudi neveljavne. Če obstaja nedovoljena kombinacija med vhodnima podatkoma, mora sistem javiti napako.

Med testiranjem sistema tester v tabelo vpisuje tudi dejanske rezultate testov za posamezne kombinacije vhodnih podatkov. Če tester ugotovi neujemanje med pričakovanimi in dejanskimi rezultati, mora to analizirati ter ugotoviti vzrok neujemanja.

Tabela 4: Tabela pričakovanih in dejanskih rezultatov:

Vhodni podatek:	**Vhodni podatek:**	**Pričakovan rezultat:** **Nadomestilo za bol. dan**	**Dejanski rezultat:** **Nadomestilo za bol.**
0-6 let	50 €	50 €	
-1	50 €	Napaka - starost ni veljavna	
0	50 €	50 €	
1	50 €	50 €	
5	50 €	50 €	
6	50 €	50 €	
7	50 €	Napaka – znesek zavarovalnine ni veljaven	

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 116.

4.4 Testiranje zmogljivosti

S to vrsto testiranja lahko začnemo, ko je ugotovljeno, da programska rešitev deluje pravilno. Pri tej vrsti testiranja se tester ne osredotoča več na pravilnost delovanja, temveč na hitrost oz. odzivnost programske rešitve (Molyneaux, 2009).

Za planiranje in izvajanje testiranja zmogljivosti je najbolj primerna ekipa izkušenih testerjev, saj se tako združijo različne veščine testiranja, ki jih posedujejo posamezni testerji. Preden začnemo s testiranjem, je priporočljivo, da se testerji posvetujejo s poslovnimi strokovnjaki glede predvidene maksimalne obremenitve sistema v produkciji in se pripravi testno okolje, ki bo čim bolj podobno delovnemu okolju v produkciji. Testiranje začnemo na »praznem« sistemu, ki ga potem tekom testiranja pripeljemo do predvidene polne obremenitve. Med potekom testiranja beležimo odzivne čase in obnašanje sistema po posameznih fazah obremenitve sistema. Če test zmogljivosti pade, to je ko se aplikacija odziva počasi ali se celo neha odzivati, mora imeti tester pripravljene podatke, ki jih posreduje razvojni ekipi.

4.4.1 Planiranje obremenitev v produkciji

Planiranje obremenitev je prvi korak testiranja zmogljivosti. Za zagotovitev dobrega delovanja sistema v produkciji je zelo pomembno, da za test zmogljivosti karseda natančno ocenimo obremenitve sistema v produkciji ter te ugotovitve upoštevamo pri testiranju zmogljivosti. Nujno je namreč, da zagotovimo dobro odzivnost sistema za operacije, ki bodo najpogosteje izvajane v produkciji. Vendar pa je tudi pri enostavnih aplikacijah težavno določiti operacije, ki so prioritetne za test zmogljivosti.

Želja končnih uporabnikov je pogosto, da je vsaka transakcija oz. operacija izvedena v manj kot treh sekundah. Naloga testerja je tudi, da pojasni končnim uporabnikom, da imajo različne skupine operacij in transakcij različne zmogljivostne zahteve, in sicer glede na različne poslovne prioritete. Kot primer lahko vzamemo elektronsko zahtevo bančne stranke za prenos denarja na drug račun. Ker je politika večine bank, da se prenos denarja ne izvede prej kot čez noč, ima elektronska potrditev prenosa s strani stranke odzivni čas do 3 sekunde, dovoljen odzivni čas celotne izvedbe transakcije prenosa denarja na drug račun pa je lahko 12 ur oz. čez noč.

O tem, kako pomembna je odzivnost sistema za produktivnost uporabnikov, so bile v sedemdesetih in devetdesetih letih prejšnjega stoletja izvedene številne raziskave. Odkrili so, da sistem, katerega operacije imajo odzivni čas nad osem sekund, dramatično zmanjša produktivnost uporabnikov, saj le-ti pozabijo, kaj so in kaj bodo delali oz. svoje misli, ki bi bile sicer usmerjene na naslednjo operacijo, preusmerijo kam drugam. To ugotovitev so poimenovali »Pravilo osmice« (Everett et al., 2007).

4.4.1.1 Dokumentiranje odzivnosti sistema glede na zahteve

V specifikaciji zahtev so navedeni največji še dovoljeni odzivni časi za posamezne operacije aplikacije. Testerji zmogljivosti sistema s pomočjo poslovnih analitikov določijo maksimalne obremenitve sistema in čas teh obremenitev za posamezne akcije programske rešitve. Ko govorimo o maksimalni obremenitvi sistema, v splošnem mislimo na maksimalno število uporabnikov, ki bodo istočasno bremenili sistem v produkciji. Če je na primer povprečno število uporabnikov sistema 500, vendar pa se to število okrog 19ih povzpne do 2000, je maksimalna obremenitev sistema 2000 uporabnikov ter čas te obremenitve 19.00 (Tabela 5). Pri tem je potrebno upoštevati tudi to, da se teh 2000 uporabnikov razporedi po sistemu ter da jih le nekaj izvaja isto akcijo v istem trenutku (Everett et al., 2007, str. 132).

Tabela 5: Načrt polne obremenitve sistema:

Skupina akcij	***Zahtevan odzivni čas***	***Maksimalno število uporabnikov***	***Čas polne obremenitve sistema***
Pregled os. zaslona	Max 3 sek.	2000	Pon-pet: 12.00-13.00
Prijava/Odjava	Max 3 sek.	2000	Pon-pet: 12.00-13.00
Zaslon s podrobnostmi produkta	Max 4 sek.	2000	Pon-pet: 12.00-13.00
Koraki plačila	Max 7 sek.	500	Sob: 9.00-11.00
Ogled kataloga	Max 10 sek.	2000	Pon-pet: 12.00-13.00
Plačevanje s kreditno kartico	Max 30 sek.	500	Sob: 9.00-11.00

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 132.

V naslednjem koraku je potrebno ugotovljeno polno obremenitev zagotoviti na testnem okolju.

4.4.1.2 Prijave v sistem do polne obremenitve

Če je polna obremenitev 2000 uporabnikov, potem se mora za test zmogljivosti prijaviti v sistem v času kosila 2000 uporabnikov. Zelo pogosto test polne obremenitve pade v prvem poskusu prijave. Če hočemo izolirati točko napake, je dobro, da najprej preizkusimo s prijavo manjšega števila uporabnikov (5-10 uporabnikov). Nato postopno večamo število sočasnih uporabnikov sistema ter tako vidimo, na kateri točki obremenitve se sistem preneha odzivati ali se odziva neprimerno. Tako bomo odkrili različne vrste težav, kot so

npr. premajhen delovni spomin, premajhno število procesnih poti ali pa premalo datotečnega prostora (Everett et al., 2007, str. 134).

4.4.1.3 Množične odjave iz sistema

To je obraten proces prejšnjega testa. Test poteka tako, da se v roku 2-3, minut iz sistema odjavi vseh 2000 aktivnih uporabnikov. Če test ni uspešen, je potrebno za izoliranje problema stopnjevati število odjav do točke, kjer se pojavijo težave. Po vseh optimizacijah in izboljšavah, ki so bile izvedene po odkritih napakah, je potrebno na koncu zagotoviti tudi čas odjave v okviru zahtevanih odzivnih časov (Everett et al., 2007, str. 134).

4.4.1.4 Merjenje odzivnosti pod polno obremenitvijo sistema

Ko zagotovimo uspešno množično prijavo in odjavo maksimalnega števila uporabnikov v sistem, smo pripravljeni na merjenje odzivnih časov izvedbe posameznih operacij aplikacije. V tem koraku se osredotočimo na merjenje odzivnega časa posamezne operacije. To je čas, ki preteče od klika na gumb Enter v aplikaciji, do popolnega prikaza naslednjega zaslona z vsebovanimi pravilnimi rezultati ali prenosom podatkov, ki jih je zahteval uporabnik. Posamezna operacija aplikacije je torej izvedena v določenem odzivnem času. Ta odzivni čas pa je sestavljen iz posameznih časovnih točk. Če na primer traja celotni odzivni čas izvedbe določene operacije 4,5 sekund je ta odzivni čas sestavljen iz več časovnih točk:

- 0,5 sekunde klient – zaslon za vnos podatkov o plačilu,
- 0,2 sekunde omrežje – prenos podatkov o plačilu do strežnika,
- 2,4 sekunde strežnik – vnos podatkov o plačilu v bazo podatkov,
- 0,7 sekunde strežnik – ustvari zapis o potrditvi plačila,
- 0,1 sekunde omrežje – prenos zapisa o potrditvi ter podatkov do klienta,
- 0,6 sekunde klient – prikaz potrjenega zapisa,
  ========
- 4,5 sekund – skupni čas izvedbe te operacije ali celoten odzivni čas.

Seveda je potrebno izmeriti odzivne čase za vse ključne operacije programske rešitve. Meritve torej opravljamo pod različnimi obremenitvami sistema, in sicer do končne obremenitve, ki v našem primeru znaša 2000 uporabnikov. Operacije, katerih odzivni časi niso v okviru zahtev, je potrebno optimizirati. V spodnji tabeli so rdeče označeni trije odzivni časi. Za te tri operacije, ki so na Sliki 6 obarvane rdeče, so dejanski odzivni časi višji od zahtevanih, zato je te operacije potrebno še optimizirati.

Tabela 6: Ugotovitve testa pod maksimalno obremenitvijo sistema

Skupina akcij	*Zahtevan odzivni čas*	*Dejanski odzivni čas*	*Število uporabnikov*	*Čas polne obremenitve sistema*
Pregled os. zaslona	Max 3 sek.	4,5 sek.	2000	Pon-pet: 12.00-13.00
Prijava/Odjava	Max 3 sek.	2,0 sek.	2000	Pon-pet: 12.00-13.00
Zaslon s podrobnostmi produkta	Max 4 sek.	15,3 sek.	2000	Pon-pet: 12.00-13.00
Koraki plačila	Max 7 sek.	3,0 sek.	500	Sob: 9.00-11.00
Ogled kataloga	Max 10 sek.	1,6 sek.	2000	Pon-pet: 12.00-13.00
Plačevanje s kreditno kartico	Max 30 sek.	103 sek.	500	Sob: 9.00-11.00

Vir: G. D. Everett, R. McLeod, Testing Across the Entire Software Development Life Cycle, 2007, str. 137.

5 PRIJAVLJANJE UGOTOVITEV TESTIRANJA

Poročila o programskih napakah so primarni produkt večine testerjev. Ugled posameznega testerja se pridobi predvsem na osnovi teh poročil.

5.1 Zakaj niso vedno odpravljene vse napake?

V praksi se dogaja, da niso vse ugotovljene napake takoj odpravljene. Nekatere so popolnoma prezrte, nekaj pa jih je odloženih za odpravo z naslednjimi verzijami programske rešitve. Za takšne odločitve je več razlogov, najpomembnejši med njimi pa so (Wikipedia, 2009):

- **Pomanjkanje časa**. Skoraj vsak projekt se sooča z velikim številom zahtev, pomanjkanjem programerjev, testerjev ter ostalih članov projekta. Ob boku tem težavam pa so še roki, ki so določeni za uvedbo programske rešitve v produkcijo oz. postavitev programske opreme na police.
- **Tole ni napaka**. "Tole ni napaka, tole je nova zahteva!" je stavek, ki ga je slišala že večina testerjev. Pogosto so med testiranjem ugotovljene določene pomanjkljivosti v funkcionalnosti programske rešitve. Če pa takšna funkcionalnost ni bila specificirana v dokumentaciji zahtev, pa to ni napaka aplikacije, ampak nova funkcionalnost ali

sprememba specifikacije. Če smo soočeni s pomanjkanjem časa in imamo določene roke, se takšne vrste "napak" preložijo v odpravljanje s kasnejšimi verzijami aplikacije.

- **Popravljanje je rizično**. Žal se tudi to dogaja, saj je programska oprema krhka, prepletena. Odprava določene napake lahko povzroči nastanek novih. Glede na natrpan urnik in pritiske bližajočih se rokov je spreminjanje logike programa pogosto prerizična odločitev. Verjetno je bolje pustiti znano napako in se tako izogniti neznanim napakam, ki bi se morda pojavile.
- **Reševanje preprosto ni potrebno**. Kljub temu da morda zveni kruto, pa je to realnost. Napake, ki se pojavljajo redko in napake, katerim se je mogoče izogniti po drugi poti (angl. workarounds), pogosto niso odpravljene. Take odločitve se sprejemajo glede na poslovna tveganja.

Na ta seznam spada še ena točka, ki je lahko vzrok vsem prejšnjim:

- **Neučinkovito poročanje o napaki.** Tester ni podal primernega testnega primera oz. njegovo poročilo o napaki ni vzbudilo potrebe po reševanju težave. Če je poročilo o napaki neustrezno napisano, si lahko programer napako razlaga napačno, napako lahko upošteva kot nepomembno za reševanje, si predstavlja reševanje težave kot preveč rizično ali pa smatra, da reševanje ni potrebno.

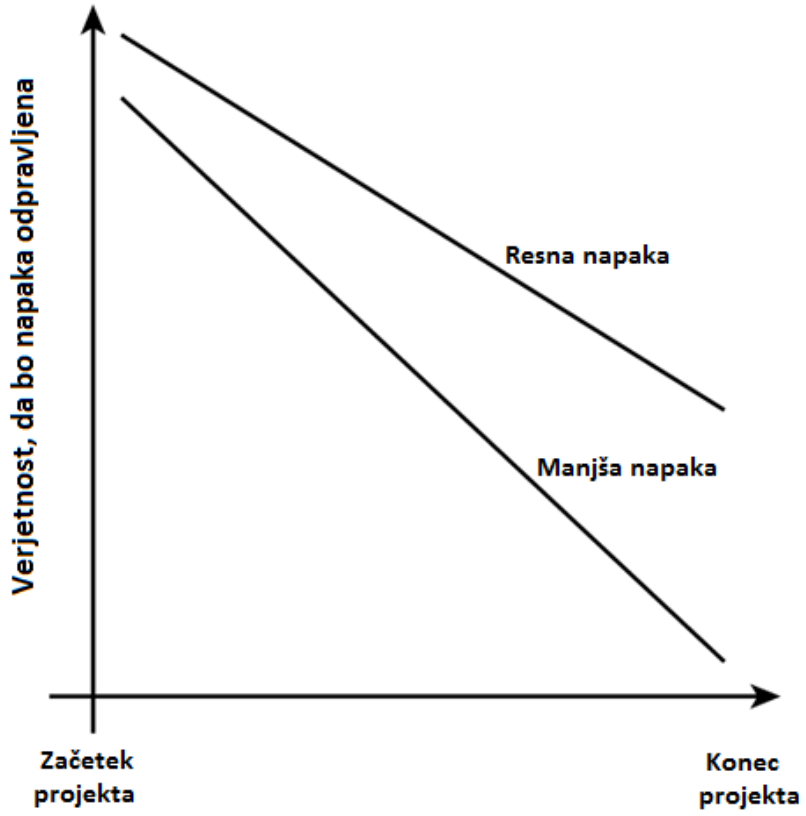
Zaradi različnih modelov razvoja programske opreme in dinamike projektne ekipe se upoštevanje poročil o napaki med posameznimi razvojnimi projekti razlikuje. V mnogih primerih je odločitev izključno na strani vodje projekta, v drugih primerih odloča programer, poslovni analitiki, testerji ali pa odločitve sprejemajo skupinsko.

Navkljub vsem tem dilemam o reševanju prijavljene napake pa je vseeno nujno, da tester svoje ugotovitve jasno in jedrnato predstavi ekipi, ki odloča o tem, ali je napako vredno odpraviti ter kdaj bodo pričeli z odpravljanjem le-te.

Za predstavitev dobrega argumenta o pomembnosti odpravljanja napake je nujno potreben predvsem zdrav razum ter komunikacijske sposobnosti. Tester naj bi se pri pisanju poročil držal temeljnih načel poročanja napak (Patton, 2005):

- **Napako je potrebno prijavi čim prej.** To je zelo pomembno, saj za reševanje napake, ki je bila odkrita prej, ostane več časa glede na to, da so roki za produkcijo še daleč. Denimo, da je odkrita neprijetna pravopisna napaka v Pomoč datoteki, in sicer nekaj mesecev pred prodajo programske opreme. Glede na to, da je rok za dokončanje projekta še daleč, obstaja velika verjetnost, da bo napaka odpravljena. Če najdete enako napako nekaj ur pred objavo, pa je verjetnost za odpravo napake skoraj nična. Na Sliki 9 je grafično prikazana povezava med časom in verjetnostjo odpravljanja napake.

Slika 9: Povezava med časom in verjetnostjo odpravljanja napake



Vir: R. Patton, Software Testing (2th ed.). 2005, str. 292.

- **Jasno in jedrnato opisovanje napake**. Če npr. programer dobi poročilo o napaki "Kadarkoli natipkam kup naključnih znakov v polje za prijavo v sistem, se začne aplikacija čudno obnašati", se zelo težko loti odpravljanja te napake, saj iz poročila ni razvidno, kakšni so ti naključni znaki, kako velik kup je ter kakšne vrste čudnih stvari se dogaja.
  Bodi kratek in jedrnat. V poročilu je potrebno navesti le dejstva in napake, ki so potrebne za prestavitev težave. Namesto navedbe "kup naključnih znakov" bi bilo veliko ustrezneje navesti natančno zaporedje znakov, ki vodijo k napaki.
  Eno poročilo, ena napaka. Pogosto je težko razlikovati med podobnimi napakami. Neizkušeni testerji zato podobne napake stisnejo v eno poročilo, kar pa lahko privede do tega, da je upoštevana in odpravljena le prva napaka iz poročila, ostale pa spregledane.
  Izoliranje napake. Če se napaka pojavi v aplikaciji med izvajanjem testa, ki vsebuje ogromno različnih akcij, je potrebno napako izolirati na čim manj klikov. Primer je npr. napaka, ki se je pojavila na avtomatiziranem testu, in sicer po šesturnem neprekinjenem izvajanju testa. V takšnem primeru je seveda potrebno napako izolirati, sicer obstaja

velika verjetnost, da ne bo rešena. Napako, ki se je pojavila po 10000 klikih, je npr. potrebno izolirati na ponovitev z 10 kliki.

Ponovljive napake. V poročilu o napaki je potrebno navesti zaporedje korakov, ki privedejo do ponovitve napake. Izziv dobremu testerju je napako, ki na prvi pogled zgleda kot posledica mnogih naključnih korakov, izolirati na čim manjše število specifično definiranih korakov.

- **Brez obsojanja v poročilu.** V poročilih ne smemo nikoli osebno napasti programerja oz. kateregakoli člana projektne ekipe. Poročilo kot je: »Tvoja koda za nadzor tiskalnika je grozna.« je neprimerno in ne prispeva k dobrim odnosom v ekipi. Poročila ne smejo vsebovati škodoželjnosti, obtožb, osebnih napadov …
- **Spremljajte svoja poročila o napakah.** Odgovornost vsakega testerja je, da napake, ki jih ugotovi in posreduje v reševanje, spremlja, dokler je napaka aktualna. Dober tester odkrije ter prijavi veliko napak, odličen tester pa te napake tudi spremlja skozi celoten proces odpravljanja napak.

5.2 Življenjski cikel odpravljanja napake

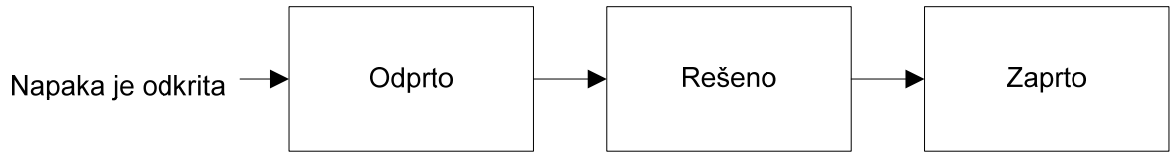
Stopnje življenjskega cikla napake (exforsys.com, 2006):

- **Novo**: Ko je napaka objavljena prvič, je njeno stanje »Novo«.
- **Odprto**: Ko tester napiše poročilo o napaki, to preveri vodja testiranja in če je napaka pristna, spremeni stanje v »Odprto«.
- **Dodeljeno**: Glede na vrsto napake je določen razvijalec ali skupina razvijalcev, ki bodo odpravljali napako. Poročilo se torej dodeli ustrezni osebi.
- **Testiranje**: Ko programer odpravi napako, preusmeri poročilo o napaki na testno ekipo v ponovno testiranje. Preden je popravek zajet v novih posodobitvah programske opreme, dobi stanje »Testiranje«. To stanje določa, da je bila napaka odpravljena ter da je zdaj v fazi testiranja.
- **Odloženo**: Napaka, katere stanje je bilo spremenjeno v »Odloženo«, bo najverjetneje odpravljena v eni od naslednjih verzij. Razlogov za spremembo poročila v to stanje je več. Najbolj pogost razlog je ta, da je pomembnost oz. prioriteta odpravljanja te napake nizka, saj se napaka bodisi pojavlja zelo redko ali pa popravek ne bo bistveno vplival na delovanje programske opreme.
- **Zavrnjeno**: V kolikor razvijalec meni, da napaka ni resnična, zavrne poročilo.
- **Podvojeno**: Če je napaka prijavljena dvakrat ali pa se v drugem poročilu omenja isti koncept napake, potem je stanje poročila spremenjeno v »Podvojeno«.
- **Rešeno**: Ko programer popravi kodo, preusmeri poročilo na testerja ter spremeni status poročila v »Rešeno«.
- **Zaprto**: Tester opravi ponovni test ter spremeni stanje v »Zaprto«, če ugotovi, da je popravek ustrezen.

- **Preverjeno**: Ko je napaka popravljena in je status poročila spremenjen v »Testiranje«, tester ponovi test. Če v programski opremi ni več omenjene napake odobri popravek ter spremeni status v »Preverjeno«
- **Ponovno odprto**: Če je na ponovnem testu ugotovljeno, da napaka še vedno obstaja tudi po tem, ko naj bi programer napako odpravil, je stanje spremenjeno v »Ponovno odprto«.

Slika 10 prikazuje najenostavnejši primer življenjskega cikla napake. Ko tester prvič naleti na napako, napiše poročilo in ga dodeli ustreznemu razvijalcu v popravilo. Stanje poročila je takrat »Odprto«. Ko programer odpravi napako, preusmeri poročilo na testerja ter spremeni stanje v »Rešeno«. Tester opravi ponovni test (test preverjanja) in če ugotovi, da je popravek ustrezen, spremeni stanje poročila v končno stanje, to je »Zaprto«.

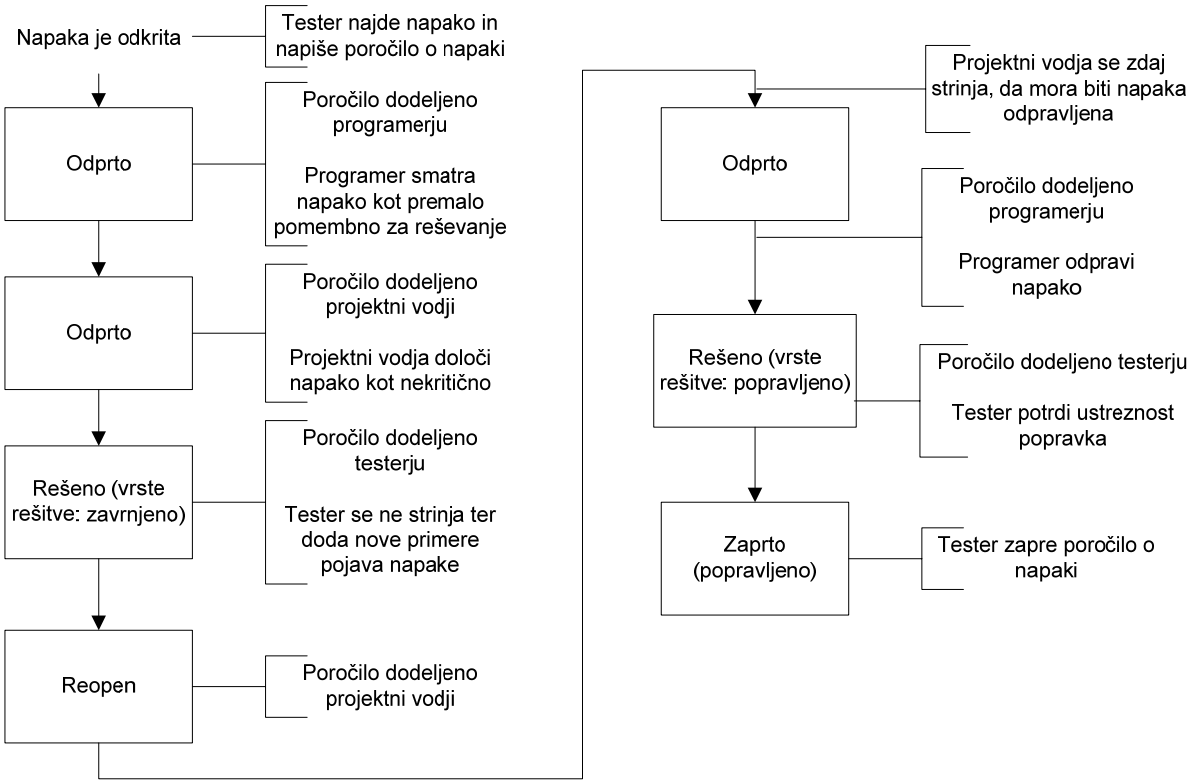
Slika 10: Življenjski cikel napake



Vir: R. Black, Managing the Testing Process (2th ed.), 2002, str. 136.

V nekaterih situacijah (glej Sliko 11) pa je življenjski cikel napake veliko bolj kompleksen. Tudi v naslednjem primeru se krog začne s tem, da tester naleti na napako ter odpre poročilo, ki ga dodeli v popravilo programerju. Vendar pa programer napake ne odpravi. Po njegovem mnenju napaka ni dovolj pomembna, zato preusmeri poročilo na projektno vodjo. Projektni vodja se strinja s programerjem, zato spremeni stanje poročila v rešeno, kot tip rešitve pa določi »Zavrnjeno«. Vendar pa se tester ne strinja z njuno odločitvijo, zato poišče bolj splošen in pogost scenarij, ki privede do napake ter spremeni stanje poročila v »Ponovno odprto«. Nova pojasnila ter testni primeri prepričajo projektnega vodjo v pomembnost odprave napake, zato preusmeri poročilo nazaj na programerja. Programer končno reši napako, status poročila pa spremeni v »Rešeno«. Ko je poročilo dodeljeno testerju, le-ta preveri ustreznost popravka ter spremeni status v »Zaprto«.

Slika 11: Kompleksen življenjski cikel napake



Vir: R. Black, Managing the Testing Process (2th ed.), 2002, str. 138.

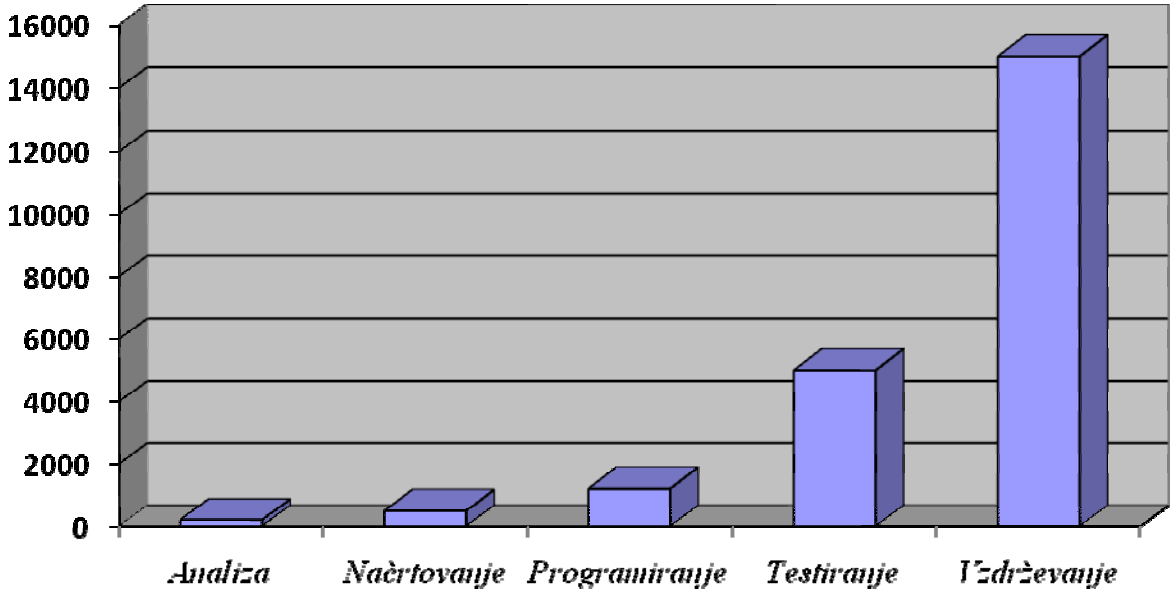
Ko tester pri testiranju aplikacije naleti na napako, se torej začne življenjski cikel odpravljanja napake, ki se praviloma konča, ko je napaka odpravljena in po testu potrjena s strani testerja. Življenjski krog napake se sicer razlikuje od podjetja do podjetja, vendar pa je v osnovi podoben. Večina projektnih skupin določi svoja pravila o tem, kdo lahko spreminja status poročila oz. jih preusmerja. Določijo lahko na primer, da odložitev napake določi le projektni vodja, poročilo pa lahko zapre le vodja testiranja. Za testerja je najpomembnejše, da preko celotnega cikla spremlja svoje poročilo ter zagotavlja potrebne podatke, dokler se krog ne zaključi.

6 EKONOMSKE KORISTI IN STROŠKI TESTIRANJA IR

Primarna naloga testerjev je čimprejšnje odkrivanje programskih napak. V zgodnjih fazah analize in oblikovanja specifikacije je namreč strošek odkrite napake zanemarljiv, z vsako nadaljnjo fazo v razvoju informacijskega sistema pa strošek odpravljanja napake močno narašča. Stroški napak, ki so odkrite s strani končnih uporabnikov oz. že v času produkcije, se lahko merijo v milijonih evrov.

Programske napake so torej lahko odkrite v različnih fazah razvoja informacijske rešitve, in sicer v fazi oblikovanja specifikacije, fazi modeliranja, fazi programiranja, fazi testiranja ali pa v fazi vzdrževanja oz. uporabe programske rešitve s strani končnega uporabnika.

Slika 12: Stroški glede na fazo v razvoju aplikacije



Vir: K. Ross, Practical Guide to Software System Testing, 1999, str. 14.

Kot vidimo na Sliki 12, stroški odkritih napak naraščajo logaritmično. Programska napaka, odkrita in odpravljena v zgodnji fazi oblikovanja specifikacije, stane do 200 €, ista napaka, odkrita v času testiranja, pa povzroči od 1000 € do 5000 € stroškov. Če pa napako odkrijejo končni uporabniki, pa se lahko stroški odpravljanja le-te zlahka povzpnejo na tisoče ali milijone evrov.

V splošnem velja, da lahko testiranje smatramo kot naložbo. Posamezna IT podjetja se odrečejo porabi denarja za nove projekte ter ga porabijo za financiranje tesne ekipe. V nadaljevanju bom predstavil analizo stroškov kakovosti, ki prikazuje, kakšna naložba zna biti investiranje v testiranje informacijske rešitve (Lewis, 2004).

Projektni vodje lahko vidijo testiranje kot »nujno zlo«, ki se pojavi ob koncu projekta. Glede na to, da taki vodje smatrajo, da testiranje traja predolgo, da je zanj porabljenega preveč denarja, povzroča konflikte med testno ekipo in razvijalci ter nenavsezadnje ne pomaga pri gradnji programske rešitve, je razumljivo, da na takih projektih le stežka pridemo do primernega proračuna, namenjenega testiranju.

Ostali vodje projektov pa so pripravljeni testiranju nameniti več sredstev. Zavedajo se namreč, da je testiranje naložba v kakovost, naložba ki prispeva k uspešnosti projekta (Lewis, 2004).

6.1 Koliko stane kakovost?

V knjigi Kakovost je brezplačna (*angl.* Quality Is Free) je Phil Crosby podal enačbo, ki prikazuje sestavo stroškov kakovosti (Black, 2002, str. 401).

$$\textbf{Stroški}_{\textbf{kakovosti}} = \textbf{stroški}_{\textbf{skladnosti}} + \textbf{stroški}_{\textbf{neskladnosti}}$$

6.1.1 Stroški skladnosti

Med stroške skladnosti vključujemo stroške preprečevanja in stroške ocenjevanja.

Stroški preprečevanja zajemajo denar, ki je porabljen za naloge zagotavljanja kakovosti, kot je usposabljanje testne ekipe in pregledovanje dokumentacije. Stroški ocenjevanja pa se nanašajo na stroške, ki izhajajo iz načrtovanja testnih aktivnosti, kreiranja testnih scenarijev in podatkov za potrebe testiranja in enkratne izvedbe pripravljenih testnih primerov (Black, 2002).

6.1.2 Stroški neskladnosti

Izvor stroškov neskladnosti pa so lahko notranje ali zunanje odpovedi.

Stroški notranje odpovedi vključujejo vse stroške, ki nastanejo, ko med prvim testom naletimo na napako. Tak strošek običajno naredi programer, ki prvi naleti na napako med testiranjem svoje kode. Strošek je povezan s odpravljanjem te napake. Stroški notranje odpovedi narastejo, ko pridemo do formalnega testiranja s strani testne ekipe. Če je napaka odkrita v tej fazi, so stroški razumljivo višji, saj proces odpravljanja napake ni več omejen le na programerja. V tem primeru tester med testiranjem aplikacije naleti na napako in napiše poročilo, programer napako reši, sistemski administrator doda popravek v aplikacijo, na koncu pa tester ponovno testira scenarij in tako preveri ustreznost popravka.

Stroške, ki nastanejo, ko namesto projektne ekipe na napako naletijo končni uporabniki oz. stranke, pa uvrščamo med **stroške zunanje odpovedi**. Ti stroški so višji od katerihkoli stroškov notranje odpovedi. V takšnih primerih se proces odpravljanja težave, ki je odkrita s strani testne ekipe, razširi še na ekipo tehnične podpore. Seveda pa je potrebno upoštevati

še neopredmetene stroške, kot so na primer jezni uporabniki, slabši ugled podjetja, izguba projektov in včasih celo tožbe.

Iz tega lahko povzamemo da se naložba v testiranje izplača, saj je preprečevanje napak cenejše kot odpravljanje le-teh, poleg tega pa so stroški notranje odpovedi nižji od stroškov zunanje odpovedi (Black, 2002).

6.1.3 Študija primera

Za boljšo predstavo o tem, da se investiranje v testiranje izplača, bom podal še praktični primer (Black, 2002). Kot primer vzemimo aplikacijo, ki ima en popravek vsake 3 mesece. V povprečju vsaka nova verzija vsebuje 1000 napak, ki so identificirane med testom te verzije ter morajo biti rešene v okviru naslednjega popravka.

Programerji med razvojem aplikacije najdejo in odpravijo 250 napak, medtem ko je preostanek napak, torej 750, odkrit s strani strank. Predpostavimo, da napaka, odkrita s strani programerja, stane IT podjetje 10 EUR, medtem ko napaka, ki jo odkrijejo stranke, povzroči podjetju stroške v višini 1.000 EUR.

V tabeli 7 lahko v stolpcu »Brez formalnega testiranja« vidimo, da stroški kvalitete v tem primeru znašajo 752.500 EUR. S tem denarjem ne pridobimo ničesar, temveč tvegamo nezadovoljne kupce, izgubo strank, tožbe …

Predpostavimo še, da so analize pokazale, da stroški napak, ki so jih odkrili testerji znašajo 100 EUR. V stolpcu »Ročno testiranje« vidimo, kako donosna je investicija v testiranje v primeru, da podjetje v testni proces vloži 70.000 EUR. Testna ekipa na primer pred izdajo novega popravka identificira 350 napak, kar število napak, odkritih s strani strank, skoraj razpolovi. To seveda pomeni večje zadovoljstvo strank, poleg tega pa tudi nižje stroške podjetja, ki v tem primeru znašajo 507.500 EUR. To pa pomeni, da so stroški z uvedbo testiranja padli za 245.000 EUR in da se je investicija v testiranje povrnila 350%.

Obstajajo pa še boljše rešitve. Denimo, da podjetje vloži 150.000 EUR v orodja za avtomatizirano testiranje. Uvedba avtomatiziranega testiranja bi omogočila identifikacijo 500 napak, kar pomeni, da bi stranke našle le še 250 napak. Učinkovitejša aplikacija bi seveda zadovoljila stranke, stroški kakovosti pa bi padli na 385.000 EUR, kar pomeni 445% povratek investicije.

Tabela 7: Analiza povrnitve investicije v testiranje:

	Brez formalnega testiranja	***Ročno testiranje***	***Avtomatizirano testiranje***
Testiranje			
Zaposleni	0 €	60.000 €	60.000 €
Infrastruktura	0 €	10.000 €	10.000 €
Orodja	0 €	0 €	12.500 €
Skupna investicija	**0 €**	**70.000 €**	**82.500 €**
Razvoj			
Št. odkritih napak	250	250	250
Stroški odpravljanja	2.500 €	2.500 €	2.500 €
Testiranje			
Št. odkritih napak	0	350	500
Stroški odpravljanja	0 €	35.000 €	50.000 €
Uporaba s strani strank			
Št. odkritih napak	750	400	250
Stroški odpravljanja	750.000 €	400.000 €	250.000 €
Stroški kvalitete			
Str. skladnosti	0 €	70.000 €	82.500 €
Str. neskladnosti	752.500 €	437.500 €	302.500 €
Celotni str. kakovosti	**752.500 €**	**507.500 €**	**385.000 €**
Povrnitev naložbe	**/**	**350 %**	**445 %**

Vir: R. Black, Managing the Testing Process (2th ed.), 2002, str. 406.

7 TESTIRANJE CRM V PODJETJU TELEKOM SLOVENIJE

Upravljanje odnosov s strankami (angl. Customer Relationship Management – CRM) je uskladitev poslovnih strategij, organizacijske strukture in kulture podjetja ter informacij o strankah in informacijske tehnologije tako, da je cilj vseh kontaktov s strankami zadovoljevati njihove potrebe ter dosegati poslovne koristi in dobiček (src.si, 2005).

Koncept upravljanja odnosov s strankami se je v preprostejši obliki pojavljal že v preteklosti, ko je bil poudarek predvsem na poenotenju prodaje s službo, ki se je ukvarjala s strankami. Zaradi želje po ustvarjanju in ohranitvi dobrih odnosov med podjetjem in strankami so najnovejši trendi usmerjeni v pridobivanje in poenotenje informacij o strankah. Te informacije lahko uporabljajo vse poslovne funkcije v podjetju, ki imajo stik s strankami (Dejak, 2007).

Z uporabo CRM-ja ima uporabnik, ki preko različnih kanalov pride v neposreden stik s stranko, vpogled v strankine potrebe in želje. Tako lahko delo izvaja dosledno ne glede na kanal in stranka ima občutek, da se je uporabnik posvetil le njej, kar pozitivno vpliva na odnos med njima. Ne smemo namreč pozabiti, da so za podjetja na prvem mestu še vedno koristi in dobiček. Uvedba CRM sistema pomaga pridobiti in ohraniti stranke, ki so glavni vir prihodkov. Na tem mestu velja omeniti Paretovo načelo, ki pravi, da 20 % kupcev prinese podjetju 80 % dobička.

7.1 Potek projekta CRM (Telekom Slovenije)

Podjetje, ki se odloči za uvedbo sistema CRM, se mora zavedati, da je to obsežen projekt, ki zahteva veliko znanja, potrpežljivosti in predanosti, poleg tega pa ima tak projekt velik vpliv na nadaljnje poslovanje podjetja. Uspešna uvedba zahteva veliko mero pripravljenosti na spremembe s strani zaposlenih v podjetju, prav tako pa tudi 100 % podporo s strani vodstva podjetja.

Podjetje Telekom Slovenije se je pred uvedbo sistema CRM spopadalo s številnimi težavami, ki so bile neprijetne predvsem za kupce. Neenotni procesi so vplivali na odzivne čase od naročila storitve do vključitve storitve. Veliko je bilo ročnega dela, ki so ga želeli avtomatizirati. Podatki o strankah so bili razdrobljeni v različnih sistemih, zato si je bilo težko ustvariti celovito sliko o stranki, njenih naročilih, storitvah in zgodovini.

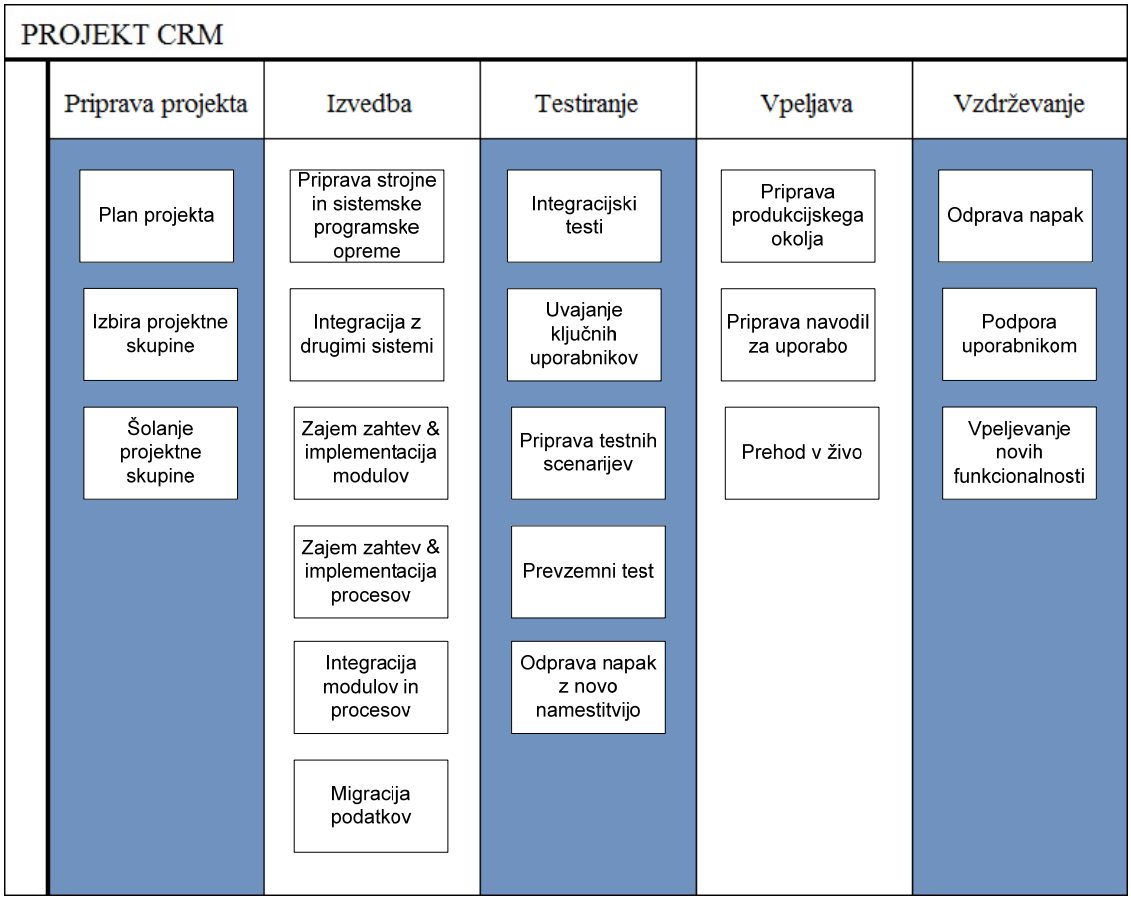
V projekt so vključeni uporabniki, oddelek informatike podjetja Telekom Slovenije in zunanji izvajalec Marand Inženiring d.o.o.. Procese so ločili od modulov, in sicer so Telekomovi informatiki definirali in poskrbeli za izvedbo procesov, zunanji izvajalec pa je bil zadolžen za realizacijo modulov, ki predstavljajo uporabniški vmesnik in bazo podatkov. Med celotno izvedbo projekta je sodelovanje vseh treh deležnikov nujno, saj se moduli in procesi prepletajo.

Okvirni časovni potek izvajanja projekta je bil narejen na začetku, vendar pa se tekom projekta prilagaja dejanskemu stanju. Glavne faze projekta so naslednje (Dejak, 2007, str. 22)**:**

- priprava strojne ter sistemske programske opreme,
- migracija podatkov,

- integracija,
- specifikacija in implementacija modulov in procesov,
- namestitev verzije modulov in integracija s procesom,
- testiranje modulov in scenarijev,
- integralno testiranje (prevzemni test),
- uvajanje ključnih uporabnikov,
- vpeljava v produkcijo,
- vzdrževanje in podpora uporabnikom.

Slika 13: Vpeljava projekta CRM



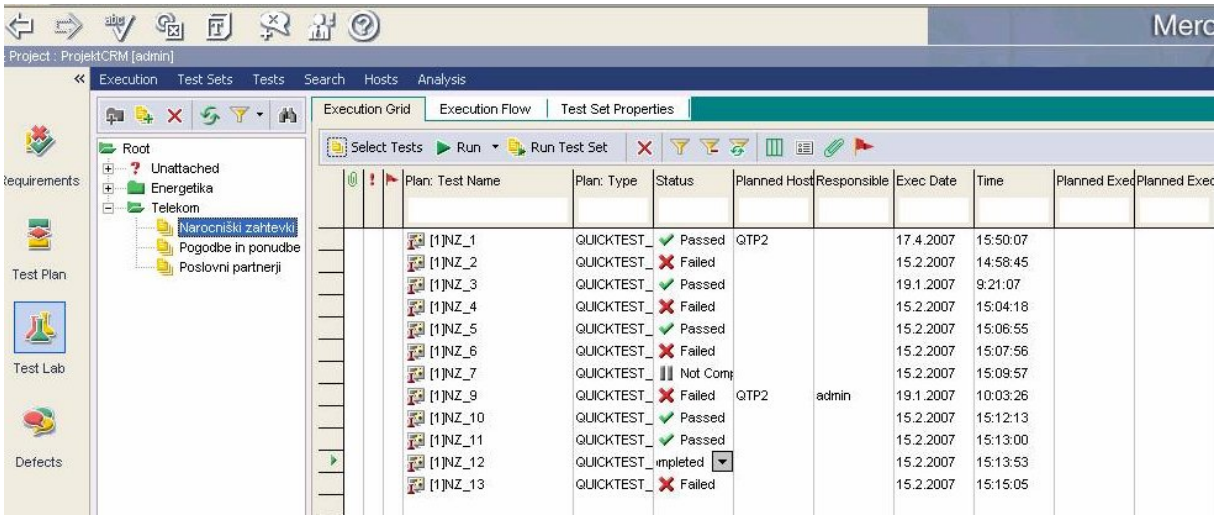
Vir: V. Dejak, Vpeljava CRM v telekomunikacijsko podjetje, 2007, str. 22.

7.2 Testiranje modulov in scenarijev

Testiranje modulov je potekalo na več nivojih. Sprva se je osnovne teste izvajalo s pomočjo orodja za avtomatsko testiranje. Prednost orodja Mercury TestDirector (Slika 14) je, da testerju v zelo kratkem času poda sliko o osnovnem delovanju sistema.

Osnova za izvajanje avtomatskih testov s pomočjo omenjenega orodja so posneti scenariji. Na tem mestu je bilo nujno potrebno sodelovanje testerja, poslovnih analitikov in bodočih ključnih uporabnikov, saj sistem CRM omogoča izvajanje velikega števila kompleksnih scenarijev, katerih izvajanje zahteva vsebinsko poznavanje posameznih storitev, ki jih ponuja podjetje Telekom Slovenije. Kmalu smo ugotovili, da avtomatsko testiranje v našem primeru ne bo prišlo v poštev. Ker je bil projekt v zgodnjih fazah, smo tekom testiranja tedensko ali dnevno prejemali nove zahteve s strani naše stranke, kar je posledično vplivalo na strukturo samih zaslonskih mask in s tem botrovalo vsakodnevnemu snemanju scenarijev. Orodje prav tako ne preverja, če se zahtevki, ki so bili izvedeni z aplikacijo, dejansko zapišejo v podatkovno bazo, ter kakšna je odzivnost sistema. Zaradi vseh naštetih težav se je vodstvo odločilo, da bomo CRM testirali ročno.

Slika 14: Mercury Quick Test Professional - orodje za avtomatsko testiranje



Vir: Mercury Quick Test Professional, 2009.

Nadaljevanje testiranja je torej potekalo ročno. Kot sem že omenil, so v projekt CRM vključeni uporabniki, oddelek informatike podjetja Telekom Slovenije in zunanji izvajalec Marand Inženiring d.o.o.. Zaradi boljše komunikacije med vsemi tremi deležniki se je naša ekipa preselila v prostore podjetja Telekom, in sicer v Stegne.

Po končani implementaciji procesov je bilo le-te treba povezati z moduli in testirati vse instance procesov tako, da je večji poudarek padel na testiranje integracije modulov s samim procesom. V ta namen smo od poslovnih uporabnikov zbrali več kot 200 različnih primerov iz realnega življenja ter jih razdelili na posamezne skupine testerjev. Skupine so bile kreirane glede na sklop storitev, ki jih bodo testirali. Posamezne skupine testerjev tako združujejo končne uporabnike, testerje in poslovne analitike. V začetku je bilo testiranje

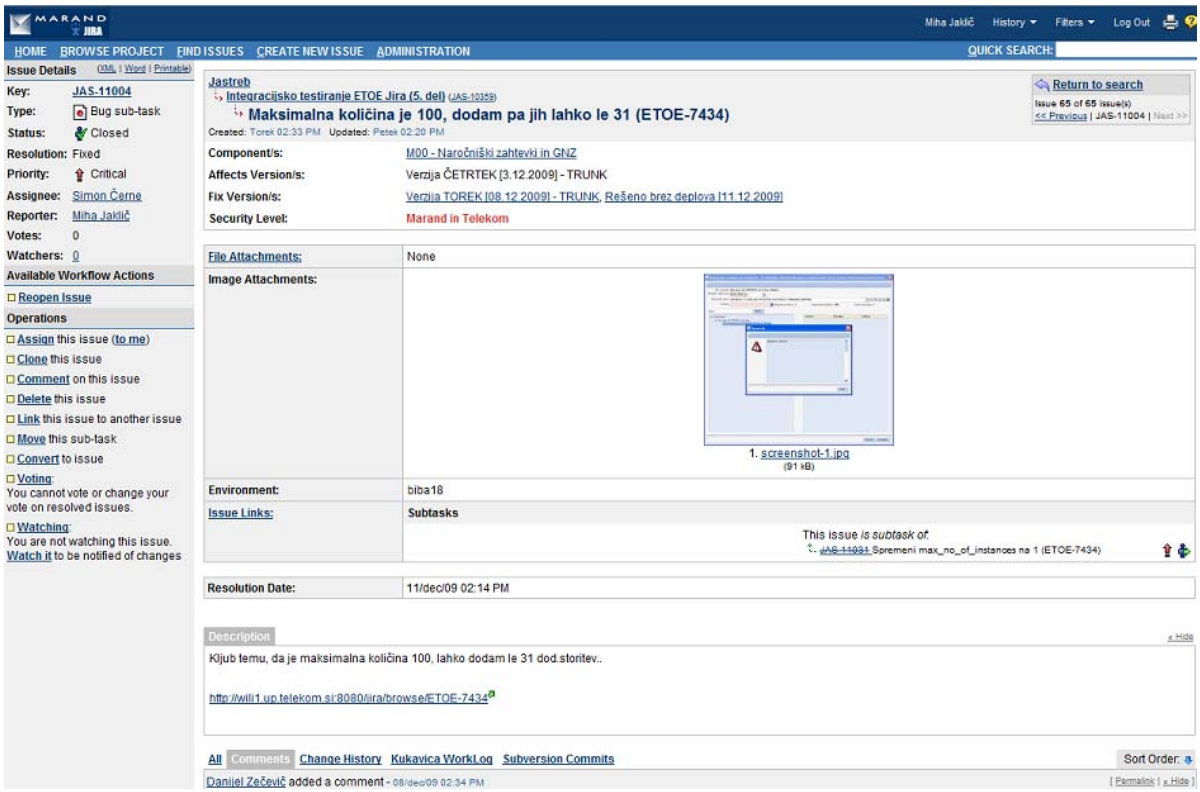
osredotočeno predvsem na pozitivne veje scenarijev, kasneje pa tudi na negativne scenarije (stornacije, spremembe naročil itd.).

7.2.1 Prijava napak – Atlassian Jira

Med testiranjem posameznih scenarijev si testerji zapisujejo svoja opažanja ter ugotovljene napake in želje za optimiziranje aplikacije prijavljajo v sistem za projektno vodenje.

Atlassian Jira je sistem za projektno vodenje, ki nam omogoča zelo premišljen postopek za evidentiranje programskih pomanjkljivosti, želje po novi funkcionalnosti, razdeljevanje nalog sodelujočim, pregled nad urnikom posameznikov in skupin. Omogoča vodenje pregleda napak, izboljšav, zahtev ali poljubnih zadev, prilagodljivo delovno okolje in statistike v realnem času. Jira je bila načrtovana kot takoj uporabno in prilagodljivo orodje s preprostim in zmogljivim uporabniškim vmesnikom, ki je lahko razumljiv tako poslovnim kot tehničnim uporabnikom. Zelo enostavno se integrira z drugimi sistemi, prav tako pa so celovito urejene pravice uporabnikov in skupin ter varnost. Sistem obveščanja uporabnikov o novih nalogah ali spremembah je popolnoma prilagodljiv (marand.si, 2009).

Slika 15: Marand jira - sistem za projektno vodenje



Vir: Marand d.o.o., 2009.

Orodje Jira je projektni skupini močno olajšalo nadzor nad aktivnostmi na projektu, testerjem pa prijavo napak ter omogočala enostavno spremljanje napredka le-teh. V takih poročilih se beležijo opravila (angl. *tasks*), datum kreiranja poročila, rok za odpravo napake, prioriteta reševanja, predvsem pa je pomembno, da sta programerju na razumljiv način opisana napaka ter postopek, ki je privedel do napake. Sistem prav tako omogoča hitro komunikacijo ter tako premošča časovne in krajevne ovire pri sporočanje informacij.

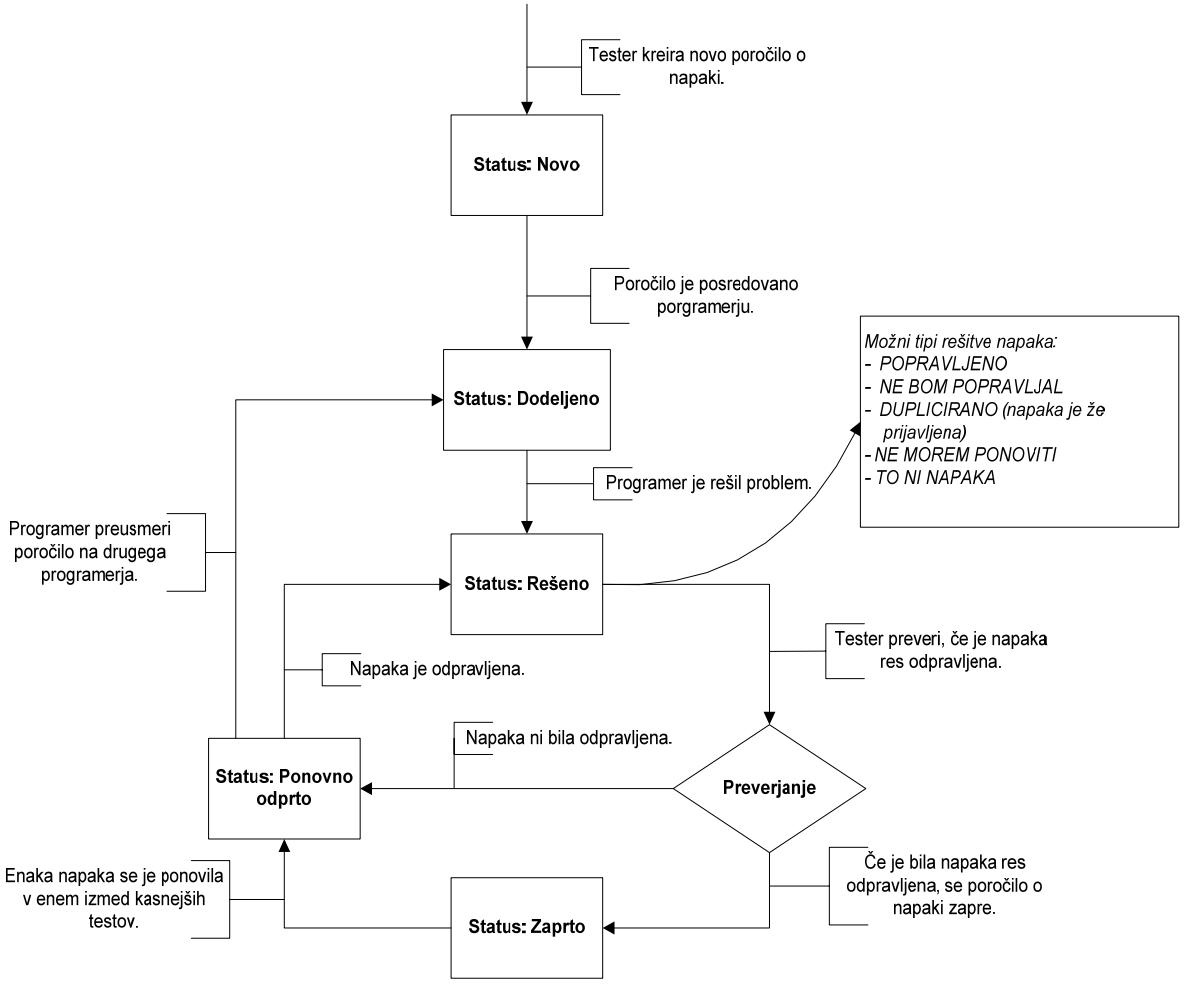
7.2.2 Analiza prijavljenih napak

Telekomova ekipa testerjev je ugotovljene napake prijavila v Telekomovo jiro ter poročilo o napaki, katere vir je bil na strani naše ekipe, usmerila na Marand CC. Te napake smo potem Marandovi analitiki ter testerji analizirali ter določili tip posamezne »napake«:

- **Napaka v kodi** (tako poročilo smo preusmerili na programerja, ki je napisal del kode z napako.)
- **Napaka na podatkih**. Napaka na podatkih je na našem projektu večinoma posledica migracije. Migracija podatkov je potekala tako, da je Marandov strokovnjak za bazo pripravil procedure, ki so kopirale podatke iz Telekomove baze strank na našo testno bazo, na kateri temelji tesno okolje. Če smo naleteli na napako na podatkih, je bila naloga analitikov, da smo ugotovili vir napake:
  - Napaka je v proceduri, ki posameznih podatkov bodisi ni kopirala oziroma jih je kopirala na napačen način. V tem primeru smo poročilo o napaki preusmerili na našega strokovnjaka za bazo.
  - Napaka je posledica napačnih podatkov v Telekomovi bazi strank. Kakovost podatkov je bila namreč predvsem v začetnih fazah projekta izredno nizka, saj so bili posamezni podatki podvojeni, veliko težav pa so povzročali tudi naslovi in uparjanje le-teh s formalnim registrom prostorskih enot. V takšnih primerih smo poročilo preusmerili na Telekomove strokovnjake, ki skrbijo za čiščenje podatkov.
- **Nepomembna napaka**. Glede na to, da smo bili stalno pod pritiskom rokov za produkcijo, smo poročilom o napakah, ki niso bistvene za delovanje aplikacije, znižali prioriteto ter jih odložili v reševanje po produkciji.
- **Nova zahteva oz. sprememba specifikacije**. Med testiranjem smo včasih naleteli na določene pomanjkljivosti v funkcionalnosti samega sistema. Če pa te funkcionalnosti niso bile navedene v specifikaciji zahtev, je o reševanju te problematike odločalo vodstvo projekta. Pomembnejše pomanjkljivosti smo pričeli odpravljati takoj, posamezne pomanjkljivosti pa so bile odložene v odpravljanje s kasnejšimi verzijami aplikacije.
- **Ni napaka**. Včasih smo ugotovili, da je vzrok določene napake napačno vnesen zahtevek s strani posameznega testerja. V tem primeru smo poročilo preusmerili nazaj na testerja ter kot tip rešitve poročila določili To ni napaka (angl. Not a bug).

Nadaljnji potek reševanja posamezne napake ter življenjskega cikla poročila sem že opisal v poglavju Življenjski cikel odpravljanja napake. Potek je nazorno prikazan tudi na Sliki 12.

Slika 12: Življenjski cikel poročila o napaki (Marand jira)



SKLEP

Cilj razvoja informacijske rešitve je njeno pravilno delovanje. Razvoj informacijske rešitve brez izjeme vsebuje napake, zato je nujno potrebno, da v proces razvoja informacijske rešitve vključimo tudi testiranje.

Do nedavnega je veljalo prepričanje, da je testiranje ena izmed manj pomembnih faz v razvoju programske rešitve, ki se izvaja ob zaključku razvojnega procesa. S testiranjem po razvoju programske rešitve se odkrije in odpravi večje napake, ostale napake pa bodo

ugotovljene s strani končnih uporabnikov in se bodo odpravljale v fazi vzdrževanja programske rešitve. Takšna miselnost je v današnjih časih neustrezna, saj tak način testiranja povzroča velike težave v kasnejših fazah razvoja in vodi v kreiranje neustrezne programske rešitve, ki ne bo zadovoljila uporabnikov in stranke.

Testiranje oz. preverjanje programske rešitve je potrebno vključiti v vse faze razvoja, saj nam to omogoči spremljanje skladnosti programske rešitve z uporabniškimi zahtevami skozi celoten proces razvoja. Na ta način so nepravilnosti v delovanju sistema odkrite pravočasno ter se ne prenašajo na naslednje razvojne faze, kar zniža čas in stroške odpravljanja napak.

Proces testiranja se izvaja z namenom zagotavljanja pravilnega razvoja programske opreme in skladnost z zahtevami uporabnikov. Testiranje skozi celoten razvoj programske opreme lahko smatramo kot pametno naložbo v kakovost, saj lahko odpravljanje napak, ki so bile odkrite s strani končnih uporabnikov, povzroči visoke stroške odpravljanja, nezadovoljstvo in izgubo strank ter posledično stečaj podjetja.

V diplomski nalogi sem izpolnil vse cilje, ki sem si jih postavil v uvodu. Med izdelavo diplomske naloge sem ugotovil, da se paktično testiranje v našem podjetju v večini sklada s teorijo tega področja, vendar pa se je proces testiranja na tem projektu resno začel šele po integraciji modulov in procesov. Ker statičnega testiranja nismo izvajali, smo imeli v kasnejših fazah obilico težav s pomanjkljivo produktno specifikacijo. Specifikacija, ki je bila pripravljena s strani Telekomove ekipe, je bila pripravljena nepopolno oz. je pomanjkljivo definirala produkte ter same lastnosti programske opreme. Posledica tega je bilo pogosto spreminjanje produktov ter zaslonskih mask, kar je onemogočalo avtomatsko testiranje in povišalo stroške testiranja.

Januarja 2010 so sistem CRM začeli uporabljati tudi končni uporabniki podjetja Telekom Slovenije, ki so z aplikacijo zelo zadovoljni. Glede na to, da s prehodom v produkcijo nismo imeli nikakršnih večjih težav, je bilo testiranje uspešno.

LITERATURA IN VIRI

1. Beizer, B. (1990). *Software testing techniques*. New York: Van Nostrand Reinhold.

2. Black, R. (2002). *Managing the Testing Process* (2th ed.). New Delhi: Wiley India.

3. Bucknall, J., M. (2007). *The seat of your Gantts*. Najdeno 28. novembra 2009 na spletnem naslovu http://community.devexpress.com/blogs/ctodx/archive/2007/07.aspx

4. Crosby, B., P. (1979). *Quality Is Free*. New York: Penguin Putnam Inc.

5. Dejak, V. (2007). *Vpeljava CRM v telekomunikacijsko podjetje* (diplomsko delo). Ljubljana: Fakulteta za računalništvo in informatiko.

6. Everett, G., D., McLeod, R., Jr. (2007). *Testing Across the Entire Software Development Life Cycle*. New Jersey: John Wiley & Sons, Inc.

7. Exforsys Inc (b.l.). *Bug Life Cycle & Guidelines*. Najdeno 3. septembra 2009 na spletnem naslovu http://www.exforsys.com/tutorials/testing/bug-life-cycle-guidelines.html

8. Gao, Z., J., Tsao, H., J., Wu, Y. (2003). *Testing and Quality Assurance for Component-Based Software*. Norwood: Artech House, Inc.

9. Jorgensen, C., P. (2008). *Software Testing: A Craftsman's Approach* (3th ed.). Boca Raton: CRC Press.

10. Kaner, C., Bach, J., Pettichord, B. (2001). *Lessons learned in software testing*. New York: John Wiley & Sons, Inc.

11. Knafeljc, B., Drnovšek, S., & Kožman, M. (2006). *Celovito obvladovanje kakovosti na projektih razvoja informacijskih rešitev*. Najdeno 18. avgusta 2009 na spletnem naslovu http://www.ipmit.si/IPMITstrani/ipmitslo.nsf/V/KF5C87FB83E0BD259C125716C0042840F/$file/04_Knafeljc_Drnovsek_Kozman_TQM.pdf

12. Lewis, E., W. (2004). *Software Testing and Continuous Quality Improvement* (2th ed.). Boca Raton: Auerbach Publication.

13. Mallory, R., S. (2002). *Software Development and Quality Assurance for the Healthcare Manufacturing* (3th ed.). Boca Raton: CRC Press.

14. Marand d.o.o. (2009). *Programska oprema - Mercury Quick Test Professional* (interno gradivo). Ljubljana: Marand d.o.o.

15. Marand d.o.o. (2009). *Sistem za projektno vodenje - Jira*. Najdeno 16. septembra 2009 na spletnem naslovu http://jira.marand.si/jira/browse/JAS-11004

16. Marand d.o.o. (2009). *Vse za vaše razvojno okolje*. Najdeno 22. decembra 2009 na spletnem naslovu http://www.marand.si/izdelki/programska-oprema/vse-za-vase-razvojno-okolje/

17. Molyneaux, I. (2009). *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. Sebastopol: O'Reilly Media, Inc.

18. Patton, R. (2005). *Software Testing* (2th ed.). USA: Sams Publishing.

19. Ross, K. (1999). *Practical Guide to Software System Testing*. Melbourne: K. J. Ross & Associates.

20. SRC d.o.o. (2005). *CRM - od poslovne strategije do uresničitve*. Najdeno 11. decembra 2009 na spletnem naslovu www.src.si/library/includes/file.asp?FileId=11. Ljubljana: SRC d.o.o.

21. Wikipedia. *Software bug*. Najdeno 23. avgusta 2009 na spletnem naslovu http://en.wikipedia.org/wiki/Software_bug

22. Wikipedia. *Software documentation*. Najdeno 18. avgusta 2009 na spletnem naslovu http://en.wikipedia.org/wiki/Software_documentation