UNIVERZA V LJUBLJANI

FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – praktična matematika (VSŠ)

Suzana Kodarin

Avtomatizacija testiranja programske opreme

Diplomska naloga

Ljubljana, 2011

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Zahvala

Najprej bi se zahvalila mentorju mag. Matiji Lokarju za mentorstvo pri diplomski nalogi. Zahvala gre tudi podjetju IN2, d. o. o., zaradi katerega je tema diplomske naloge sploh nastala. Posebna zahvala gre moji družini in prijateljem, zaradi katerih je bilo celotno študijsko obdobje manj stresno in bolj prijetno.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Program diplomske naloge

V diplomski nalogi predstavite osnovne pojme v povezavi s testiranjem programske opreme. Podrobneje predstavite možnosti avtomatizacije testiranja. Opišite uporabo orodja Oracle application testing suite.

Osnovna literatura

- Myers, *G. J., The Art Of Software Testing 2nd ed.*, New Yersey, John Wiley & Sons, Inc, 2004.
- Fewster, M., *Software Test Automation: Efective user of test execution tools,* Addison Wesley, 1999.

mentor

mag. Matija Lokar

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Povzetek

Diplomska naloga je razdeljena na tri dele. Prvi del zajema pet poglavij, v katerih so na kratko opisani splošni pojmi, metode in postopki testiranja programske opreme. V drugem delu je predstavljena metodologija ATLM in merjenje učinkovitosti investicije (ROI). Opisane so lastnosti orodij za avtomatsko funkcionalno testiranje. Tretji del pa opisuje orodje OpenScript in uporabo tega.

Abstract

The diploma is divided into three sections. The first section consists of five chapters which shortly describe general notions, methods and procedures of software testing. The second part presents ATLM methodology, measuring Return of Investment (ROI) with description of characteristics for automated functional test tools. The third part describes the OpenScript tool and its use.

Math. Subj. Class. (2011): 97F40.

Computing Review Class. System (1998): B.2.0.

Ključne besede: programska oprema, avtomatizacija testiranja, testni scenarij, modul, metode testiranja, verifikacija, validacija.

Keywords: software, automated testing, test scenario, module, testing methods, verification, validation.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Kazalo

1 UVOD

Danes se programska oprema pojavlja že skoraj povsod. Programsko opremo najdemo v mobilnem telefonu, avtu, pečici, v bolnici, na delovnem mestu itd. Zaradi pomembnosti programske opreme tako z ekonomskega kot socialnega vidika mora biti le ta zelo kakovostna. Manj kakovostna programska oprema, ki lahko povzroči izgubo podatkov ali celo življenj, ni več sprejemljiva.

Pri zagotavljanju kakovosti programske opreme ima pomembno vlogo testiranje. Ker je programska oprema vedno bolj kompleksna, je tudi testiranje te vedno bolj zahtevno. Zato podjetja težijo k novim metodologijam testiranja, ki vodijo k avtomatizaciji testiranja. Cilj je na čim hitrejši in učinkovit način zagotoviti zanesljivo delovanje in kakovost programske opreme. Dejstvo je, da z vnaprej pripravljenimi testi lahko delovanje programske opreme preverimo hitro in učinkovito.

2 DEFINICIJA IN OSNOVNI POJMI TESTIRANJA

Definicij, kaj dejansko pomeni postopek testiranja, je veliko. Tako pogosto zasledimo naslednje definicije testiranja:

- testiranje je proces, s katerim skušamo prikazati, da v programski opremi, ki jo gradimo, ni napak (Myers, 1979);
- testiranje je aktivnost ali proces, s katerim lahko prikažemo, da program deluje v pričakovanih okvirih (Myers, 1979);
- testiranje je izvajanje programa z namenom, da bi odkrili prisotnost napak (Myers, 1979);
- testiranje je aktivnost, s katero dokažemo in dosežemo zaupanje, da programska oprema dela, tako kot bi morala delati – v skladu z zahtevami, ki so jih specificirali uporabniki (Myers, 2004);
- testiranje je izvajanje ali vrednotenje programa (ali samo komponente) z namenom, da se preveri, ali program ustreza zahtevam oziroma, da se pokaže razlika med pričakovanimi in dejanskimi izhodnimi vrednostmi. Program lahko izvajamo z računalnikom ali na kakšen drugi način (Dogša, 1993).

Pri tem se je potrebno zavedati, da »Testiranje lahko prikaže samo prisotnost napak, nikdar pa ne dokaže njihove odsotnosti« (E. W. Dijkstra).

V diplomskem delu bomo izhajali iz opredelitve, da je osnovni namen testiranja dokazati prisotnost ene ali več napak.

Ko se ukvarjamo s testiranjem, srečamo številne pojme. Tu si bomo ogledali kratko definicijo nekaterih najpomembnejših.

Verifikacija je proces, pri katerem preverjamo, ali produkt tekoče faze ustreza zahtevam, postavljenim v predhodni fazi. Je sinonim za vse vrste preverjanj (Dogša, 1993).

Validacija je proces vrednotenja programske opreme na koncu njenega razvoja z namenom, da se ugotovi skladnost z zahtevami (Dogša, 1993).

Modul je najmanjša enota programske opreme, ki jo lahko urejamo, preverjamo, vstavljamo v knjižnico in testiramo. V bistvu je to datoteka, ki vsebuje vsaj eno programsko enoto, ki jo pozna prevajalnik (Dogša, 1993).

Testni primer oziroma scenarij je simulacija poslovnih dogodkov v poslovni praksi.

Produkt je končni izdelek razvoja programske opreme.

Razvijalec je avtor programske opreme. Do programske opreme je nekritičen.

Tester je oseba, ki testira programsko opremo. Usmerjen je h kakovosti programske opreme in je do nje kritičen.

Uporabniški vmesnik je sredstvo in način prek katerega in s katerim ljudje in računalniki komunicirajo med seboj. Predstavlja celoten sistem komunikacije med računalnikom in njegovim uporabnikom (človekom): »z ene strani posreduje uporabniške informacije, z druge pa prejema informacije od uporabnika« (Kragelj, 2002).

Aplikacija je program v računalništvu.

3 METODE TESTIRANJA

S testiranjem ugotavljamo pravilnost delovanja programske opreme. Na testiranje lahko gledamo z različnih vidikov. Eden od teh vidikov je, da ugotavljamo, ali gradimo programsko opremo na pravi način. Drugi vidik je, ali sploh gradimo pravo programsko opremo. Prvi vidik se imenuje verifikacija, drugi pa validacija (Solina, 1997).

Glede na to, ali testiramo posamezne dele programske opreme ali njene združene sklope, ločimo (Podgorelec, 2007):

- testiranje modulov

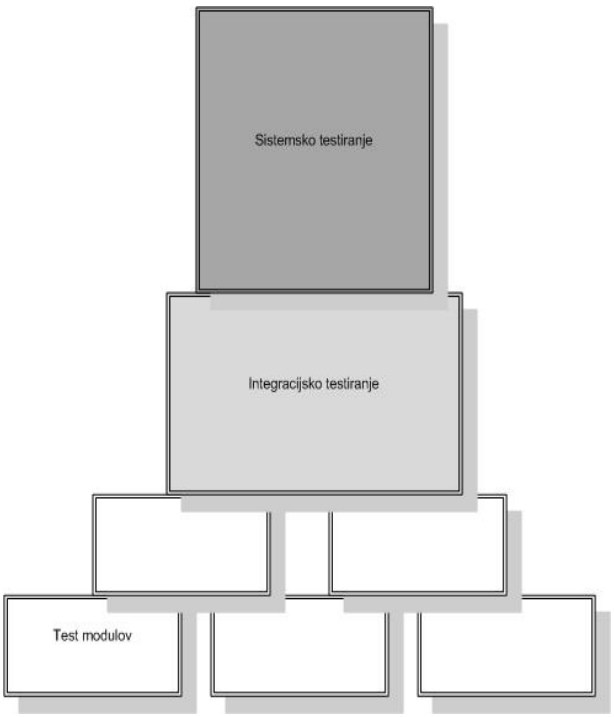
Pri testiranju modulov preverjamo, ali je posamezen modul pravilno implementiran. Pri tem testiramo posamezne module (angleško Unit testing). Testiranje izvede razvijalec, torej tisti, ki je napisal implementacijo modula.

- integracijsko testiranje

Pri integracijskem testiranju združujemo module v celoto. Testiramo povezave med posameznimi moduli in delovanje kot celoto. Testiranje izvede razvijalec.

- sistemsko testiranje

Pri sistemskem testiranju preverjamo, ali produkt izpolnjuje zahteve. Testiranje izvedeta razvijalec in naročnik produkta.



Slika 1: Nivoji testiranja [Vir slike: (Podgorelec, 2007)]

Več o nivojih testiranja je razloženo v poglavju POSTOPKI TESTIRANJA

Glede na način testiranja programske opreme ločimo:

- statične metode in
- dinamične metode testiranja.

Statične metode ne zahtevajo izvajanja programa. Pri statičnih metodah gre za branje dokumentacije in sledenje programski kodi. Pri tem ugotavljamo, ali je programska oprema grajena tako, kot navaja dokumentacija. Bolje kot je dokumentacija pripravljena, večja verjetnost je, da bodo razvijalci napisali dobro kodo.

Pri dinamičnih metodah gre za izvajanje programa in primerjanje rezultatov programa s pričakovanimi rezultati.

Poznamo dva osnovna pristopa k testiranju, ki sta si enakovredna in se dopolnjujeta. Uporabljamo ju na različnih ravneh testiranja:

- strukturno testiranje ali metoda bele skrinjice in
- funkcionalno testiranje ali metoda črne skrinjice.

Tako pri strukturnem kot pri funkcionalnem testiranju gre za isti cilj, in sicer za odkrivanje prisotnosti napak. Metodi se razlikujeta po načinu doseganja tega cilja.

Poleg strukturnega in funkcionalnega testiranja poznamo tudi preverjanje in sledenje kode ter ad-hoc testiranje.

3.1 Strukturno testiranje ali metoda bele skrinjice

Strukturno testiranje je testiranje programske opreme, ki potrebuje dostop do notranje strukture programa in algoritmov, vključno s kodo, ki le-te implementira. Za preverjanje delovanja kode izberemo vhodne podatke, ter določimo primerne izhodne podatke.

Testne primere naredimo na podlagi strukture programa in na podlagi znanja, ki jih ima tester. Testiramo posamezne zanke, pogoje, stavke, poti itd.

Pri strukturnem testiranju je naš namen zagotoviti, da se bodo vsi stavki in pogoji izvedli vsaj enkrat. Zato pripravimo takšne testne primere, ki bodo zajemali vse stavke in pogoje. Cilj strukturnega testiranja je s čim manj testnimi primeri pokriti čim več možnih načinov izvajanja programa. Pri tem nas zanima, v kolikšni meri testiranje sploh pokriva programsko kodo. Pokritost je merilo, ki kaže, kako temeljito je bila preverjena struktura programa. To pomeni, da preverimo, v kolikšni meri s testnimi primeri izvedemo stavke in pogoje.

Kodo programa najbolj obvlada avtor programa, zato je morda on najbolj poklican, da izvede to testiranje. Vendar je zaradi objektivnega testiranja bolje, da v vlogi testerja nastopa druga oseba.

Kot sem omenila, pri strukturnem testiranju preverjamo pokritost programske kode. Obstaja veliko različnih orodij, s katerimi si delo olajšamo. Poznamo naslednje metode pokrivanja kode (Dogša, 1993).

- Pokrivanje stavkov

Preverjamo, ali se ob testiranju vsi stavki kode izvedejo vsaj enkrat. Pri tem nam zelo pomaga to, da imajo nekateri prevajalniki možnost vklopiti nastavitev, ki omogoča pregled izvršenih in neizvršenih stavkov.

Primer:

```
stavek1;
če (pogoj) potem
        stavek2;
stavek3;
```

Koda bo pokrita, če imamo takšen testni primer, ki bo zagotovil, da je pogoj izpolnjen in se bo tako stavek2 izvedel.

- Pokrivanje zank

Testiramo tako, da določimo takšne testne primere, da se zanka ne izvede (izjema so zanke repeat, ki se vedno izvedejo vsaj enkrat), da se zanka izvede enkrat in da se zanka izvede več kot enkrat.

- Pokrivanje poti

Za testiranje poti določimo takšne testne vzorce, da se izvedejo vse poti v programu. Pot je zaporedje stavkov ali vejitvenih stavkov. Ker je že v majhnem programu veliko poti, se v praksi uporabljajo razne podmnožice preverjanja pokritosti poti, kot je na primer testiranje pokritosti stavkov in testiranje pokritosti odločitev.

- Pokrivanje pogojev

Za to testiranje je potrebno izbrati nabor takšnih testnih vzorcev, da vsak pogoj v odločitvenih stavkih zavzame vsaj enkrat stanje res in vsaj enkrat ni res.

- Pokrivanje odločitev

Testni vzorci morajo biti določeni tako, da se vse možne veje, ki izhajajo iz vejitvenih stavkov, izvedejo vsaj enkrat. Razen v enostavnih primerih je 100-odstotno pokritost težko doseči. V primerjavi s testiranjem stavkov je testiranje odločitev temeljitejše. Če se izvedejo vse odločitve, pri čemer s tem mislimo na to, da se z izvedbo testnih primerov izvedejo vse možne veje, sledi, da se izvedejo tudi vsi stavki. Obratno ni nujno res (na primer nestrukturirani programi). Tako se v primeru nestrukturiranega fortranskega programa

```
      IF (X>0) 200, 150, 150
100  X=X+A
   GO TO 100
150  X=X-A
200  X=X+X
```

ne glede na odločitev v stavku IF stavek z oznako 100 ne bo izvršil (Dogša, 1993).

3.2 Funkcionalno testiranje ali metoda črne skrinjice

Funkcionalno testiranje je najpogostejša oblika testiranja in tudi edina oblika testiranja, ki se ji ne moremo odreči. Značilnost funkcionalnega testiranja je, da obravnavamo program kot črno skrinjico, katere notranjosti ne poznamo. Preverjamo samo delovanje in funkcionalnost programske opreme, njena notranjost nas ne zanima. Usmerjeni smo v preverjanje obnašanja vhodov/izhodov. Če se za vsak podan vhod (za vsak primer testnih podatkov in akcij uporabnika) izhod (rezultat) ujema s predvidenim, potem je test uspešen. Ker je nemogoče preizkusiti vse možne vhode (testne primere), razdelimo vhodne pogoje v ekvivalenčne razrede. V vsak ekvivalenčni razred združujemo tiste vhodne podatke, pri katerih ne pričakujemo velikih razlik v obnašanju programske opreme. To pomeni, da so v istem razredu testni primeri, ki testirajo na primer isti postopek v programu. Testne primere potem izberemo iz vsakega ekvivalenčnega razreda.

Najpogostejše napake, ki jih odkrijemo pri funkcionalnem testiranju, so manjkajoče funkcije, napačno ali pomanjkljivo delovanje programske opreme, napake v uporabniških vmesnikih, napake v podatkovnih strukturah ali zunanjem dostopu do podatkovnih baz in podobno.

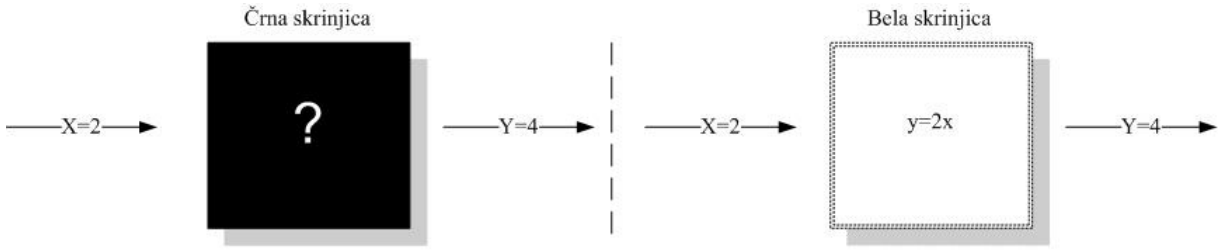
Funkcionalno testiranje je najbolj podprto z avtomatskimi testnimi orodji. Pri tej vrsti testiranja se odkrije največ napak (glej razdelek 7.1.1).

3.2.1 Primerjava metod bele in črne skrinjice

Obe metodi sta uspešni pri zagotavljanju pravilnega delovanja programske opreme. Da je proces izvedbe programa pravilen, pa lahko zagotovi samo metoda bele skrinjice. Pri pravilnem rezultatu pogosto menimo, da je pravilen tudi proces obdelave, vendar to ni nujno res. V nekaterih primerih je možno, da z napačnim procesom dobimo pravilen rezultat testa. Oglejmo si enostaven primer. Denimo, da imamo aplikacijo, ki računa potreben čas za izvedbo določene naloge. Ta vsebuje formulo y= 2x, kjer y pomeni potreben čas za izvedbo določene naloge in x težavnost naloge. Če imamo nalogo težavnosti 9, potrebujemo 18 ur za dokončanje naloge.

Lahko se zgodi, da pri določenih testnih primerih dobimo pravilen rezultat, čeprav je v program prenesena formula nepravilna. Vnesemo vrednost 2 in kot rezultat dobimo 4. Mislimo, da je formula pravilna. Če vnesemo vrednost 3, bo rezultat 9 in ne 6. To morda pomeni, da se je programer zmotil in vnesel formulo y= x^2. Če metoda črne skrinjice vsebuje le prvi testni primer, bomo prišli do napačnega zaključka glede pravilnosti delovanja aplikacije.

Za odkritje takšnih napak moramo uporabiti metodo bele skrinjice. Poznati moramo strukturo programske opreme in programski jezik. Pogledati moramo torej v notranjost aplikacije. Le tako se lahko izognemo zgoraj omenjeni napaki.



Slika 2: Primerjava metod testiranja [Vir slike: (Dustin, 2003)]

Testiranje z metodo črne ali bele skrinjice poveča odstotek odkritja napak za 40 odstotkov (v primerjavi s tem, če programske opreme sploh ne bi testirali). Če uporabljamo obe metodi, povečamo odstotek odkritja napak za 60 odstotkov (Craig, 2000).

3.3 Branje, preverjanje in sledenje kode

Branje kode je najbolj tradicionalna metoda za odkrivanje napak v programu. Čeprav avtor programa zagotovo kar nekajkrat prebere kodo, temu še ne moremo reči testiranje. Programer postane skoraj slep za nekatere pomanjkljivosti in napake, saj je osredotočen na to, kako naj program dela. Zato je bolje, če testiranje z branjem kode izvaja oseba, ki ni avtor programa. Ker ni njen avtor, je pri branju kode bolj kritičen.

Preverjanje kode je zelo učinkovit način za iskanje napak in pomanjkljivosti programske kode. Izvaja se v obliki formalnih sestankov. Namen sestankov je odkrivanje napak, ne pa njihovo popravljanje. Udeleženci sestanka so člani razvojne skupine. Vodja sestanka skrbi, da člani razvojne skupine mirno sodelujejo in, da se sestanek ne sprevrže v prerekanje. Poglejmo primer:

Nezgodno se lahko zavarujemo kot posameznik, skupina in družina. Programer napiše kodo in na sestanku predstavi, kaj počne njegov program. Pri tem udeleženci ugotovijo, da ne upošteva skupinskega zavarovanja, ampak le posamezno in družinsko. Na sestanku se ga na pomanjkljivost opozori in kasneje mora kodo tudi popraviti.

Sledenje kode predstavlja sprehajanje po programski kodi z uporabo testnih podatkov. Izvedbo sledenja kode naredi avtor. Predstavi jo drugim udeležencem postopka sledenja. Od branja kode in preverjanja kode se torej razlikuje v tem, da je pri sledenju osrednja oseba avtor programske kode, ki

jo z uporabo testnih podatkov predstavi drugim udeležencem postopka. Namen sledenja kode je tako iskanje problemov (napak) kot tudi izmenjava programerskega znanja.

3.4 Ad-hoc testiranje

Pri ad-hoc testiranju gre za ročno testiranje brez načrtovanja in dokumentacije. To pomeni, da je tester (oseba, ki testira) prepuščen improvizaciji in svojemu poznavanju programske opreme. Testiranje je odvisno od testerjevega znanja in izkušenj. Testni primeri naj bi se izvedli samo enkrat, razen če tester ugotovi napake v programski opremi. Tester z bogatimi izkušnjami pri testiranju in dobrim poznavanjem področja bo lahko odkril veliko ključnih pomanjkljivosti. Navadno se ad-hoc testiranje izvaja, kadar testerji za pripravo testnih primerov nimajo na voljo dovolj časa.

3.5 Testno okolje

Preden programsko opremo naročniku predamo v uporabo, oziroma preden jo namestimo v produkcijsko okolje, jo je treba testirati v okolju, ki bo čim bolj podobno okolju, v katerem se bo na koncu aplikacija izvajala.

Ločimo tri različna okolja:

- produkcijsko okolje: to je okolje, v katerem bo programska oprema delovala, ko bo predana naročniku programske opreme.
- testno okolje: okolje, ki naj bo čim bolj podobno produkcijskemu, zato naj vključuje enako strojno opremo, operacijski sistem, aplikacijski strežnik itd.
- razvojno okolje: je okolje, v katerem sistem razvijamo in se lahko razlikuje od produkcijskega okolja.

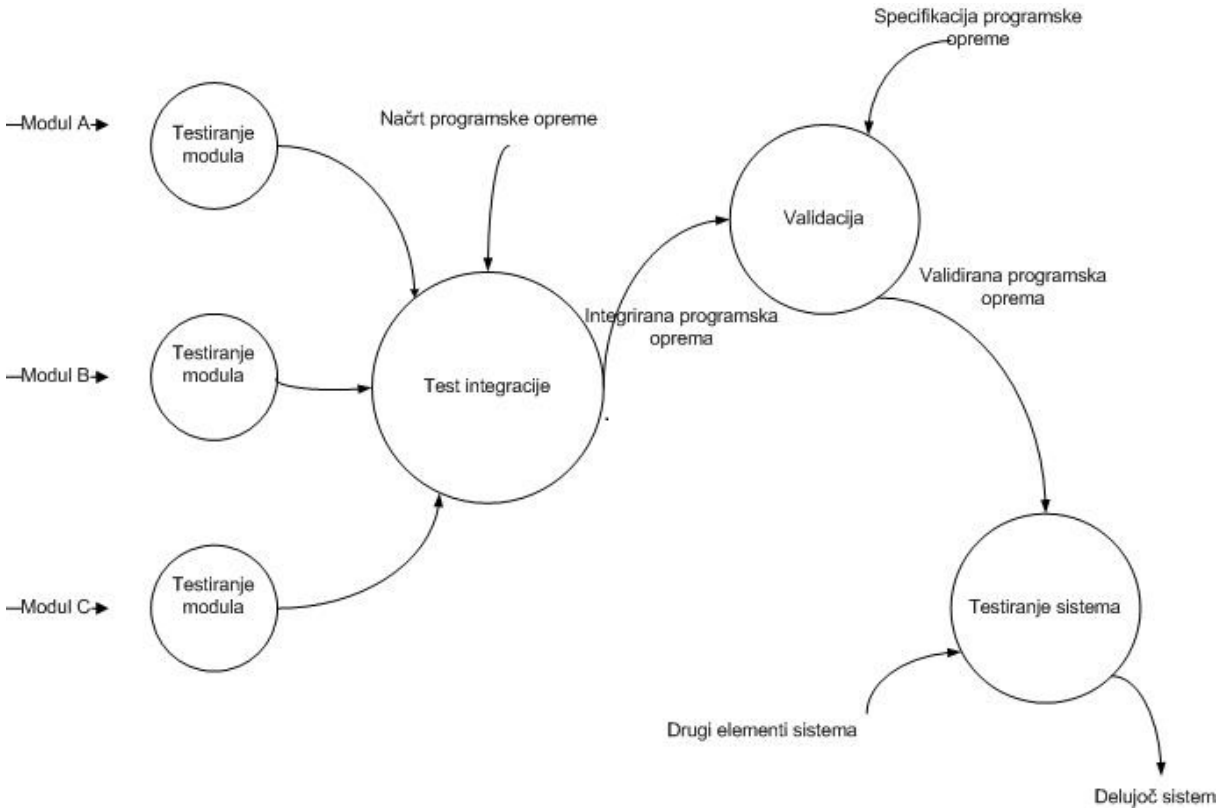
Testno okolje je pod nadzorom testne ekipe in je ločeno od razvojnega okolja. Bolj kot je testno okolje podobno produkcijskemu okolju, boljše rezultate testiranja lahko pričakujemo.

Spodnja tabela prikazuje, katere faze testiranja izvedemo v posameznem okolju.

	Razvojno okolje	Testno okolje	Produkcijsko okolje
Testiranje modulov	✓		
Integracijski testi	✓	✓	
Varnostni testi		✓	✓
Sistemski testi		✓	

4 POSTOPKI TESTIRANJA

Testiranje začnemo pri posameznih modulih, ki jih testiramo z metodo bele skrinjice. Testirane module postopoma integriramo v sistem. Ko je programska oprema sestavljena, sledi preverjanje zahtev, zapisanih v specifikaciji. Nazadnje opravimo še razna sistemska testiranja (Solina, 1997).



Slika 3: Testiranje programske opreme [Vir slike: (Solina, 1997)]

4.1 Testiranje modulov

Modul je enota kode, ki zaokroža določeno funkcionalnost. V praksi je to lahko funkcija, procedura oziroma v objektnem programskem jeziku metoda nekega razreda (Čebokli, 2006).

Ločimo tri tipe modulov (Dogša, 1993):

- navaden modul;
- krmilni modul – prenese izbrane testne vzorce v modul, ki ga testiramo, ter nato izpiše izhodne podatke;
- navidezni modul, ki simulira delovanje originalnega modula – sprejme in vrne podatke, ki jih zahteva klicani modul.

S testiranjem modulov razvijalec s pomočjo orodij za odkrivanje napak zagotavlja pravilnost delovanja posameznih modulov, ki morajo delovati v skladu z zahtevami. S takim postopkom testiranja se že v zgodnji fazi odkrijejo morebitne napake v programski opremi.

Testiranje modulov običajno izvedejo razvijalci sami. Za testiranje modulov praviloma uporabljamo metodo bele skrinjice. Osredotočeni smo na testiranje posameznih modulov. V ta namen postavimo posebno testno konfiguracijo, da bomo lahko vanjo vgradili modul. Testirani modul ne sme biti odvisen od drugih modulov. Zato potrebujemo krmilni modul, ki kliče testirani in navidezni modul, ki nadomešča podrejene module. Manj, ko je modul odvisen od drugih modulov, lažje ga je testirati (Dogša, 1993).

Pri izvedbi testiranja modulov se lahko odločimo za dva pristopa (Dogša, 1993):

- izolirano testiranje modulov – namen je ločeno preizkusiti posamezne module, preden jih integriramo v kompletni program;
- inkrementalno (postopno) testiranje modulov – najprej testiramo en modul. Ko je testiranje slednjega končano, v testno okolje dodamo nov, še ne testiran modul, nato naslednjega …

Testiranje modulov ne more zagotoviti pravilnega delovanja celotne programske opreme, ampak le ugotovi, ali posamezni moduli programske opreme delujejo pravilno, neodvisno drug od drugega.

4.2 Integracijsko testiranje

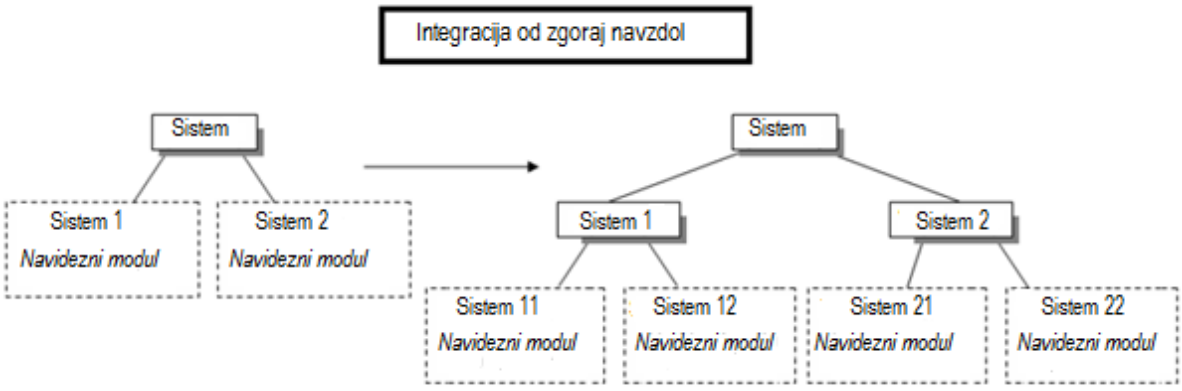
Integracijsko testiranje je vmesna faza med testiranjem modulov in sistemskim testiranjem. Pri integracijskem testiranju gre za povezovanje modulov v celoto. Testiramo medsebojno delovanje in ujemanje modulov. Kot rezultat dobimo integriran sistem. Čeprav so posamezni moduli že bili testirani, se napake lahko pojavijo. Do številnih napak lahko pride zaradi neusklajenih vmesnikov, skupnih podatkovnih struktur itd.

Integracijsko testiranje poteka po metodi črne skrinjice. To pomeni, da pripravimo vhodne podatke in nato dejanske izhodne podatke primerjamo s pričakovanimi vrednostmi. Integracija poteka postopoma. Vsakokrat, ko sistem razširimo za en modul, moramo ponoviti testiranje. Tako testiranje imenujemo regresijsko testiranje (opisano v razdelku 4.5). Z regresijskim testiranjem je vzrok morebitnih napak lažje ugotoviti, kot če bi testirali celoten sistem.

Integracija lahko poteka od zgoraj navzdol ali spodaj navzgor. Katero pot je smiselno ubrati, je odvisno od težavnosti testiranja. Med testiranjem nadomestimo module, ki še niso integrirani v sistem, z navideznimi moduli (Solina, 1997).

4.2.1 Integracija od zgoraj navzdol

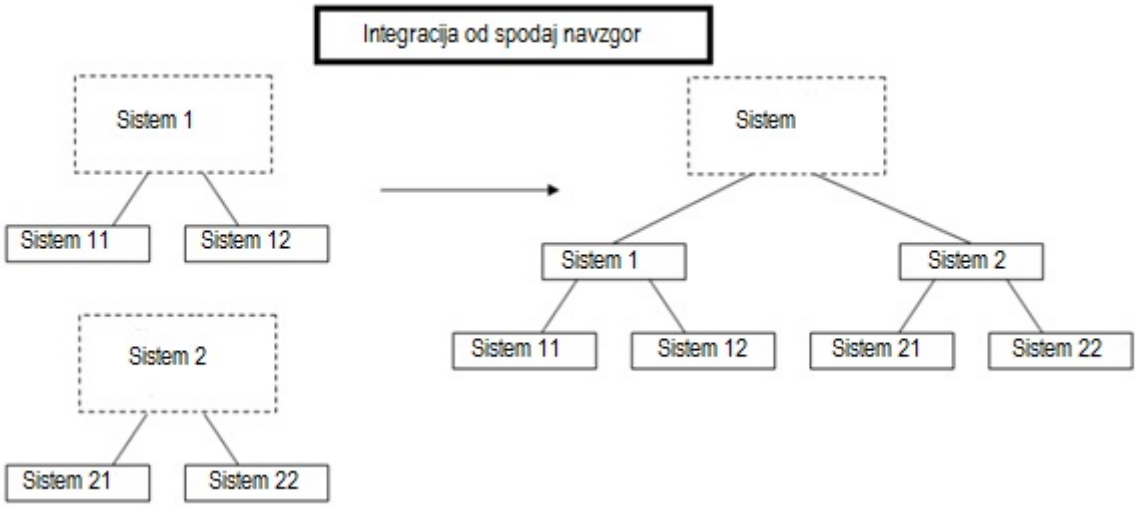
Pri integraciji od zgoraj navzdol začnemo z glavnim modulom, ki je na najvišjem nivoju. Postopoma mu dodajamo podrejene module. Manjkajoče module nadomestimo z navideznimi moduli.



Slika 4: Integracija od zgoraj navzdol [Vir slike: (Vetrih, 2009)]

4.2.2 Integracija od spodaj navzgor

Testiranje modulov začnemo na najnižji ravni. Nato vzamemo modul, ki je više in že vključuje testirane module. Manjkajoče module na višjih nivojih (module, ki še nismo testirali) kličemo s krmilnimi moduli. Postopek ponavljamo, dokler niso vsi moduli vključeni v testiranje.



Slika 5: Integracija od spodaj navzgor [Vir slike: (Vetrih, 2009)]

Da bi zmanjšali število potrebnih navideznih modulov, ali da bi najprej testirali najbolj kritične module, lahko kombiniramo oba načina integracije.

4.3 Sistemsko testiranje

Sistemsko testiranje je končna faza testiranja, ki nastopi, ko je sistem že integriran. Je najtežavnejši proces testiranja. Preveriti moramo, ali sistem res deluje tako, kot je predpisano v specifikaciji – to je validacija sistema. Po namestitvi programske opreme v delovno (produkcijsko) okolje je potrebno testirati še celoten sistem. Zanima nas hitrost delovanja programske opreme, obnašanje oziroma okrevanje sistema po izpadu sistema, maksimalna obremenitev sistema, varnost in občutljivost sistema, kaj se zgodi pri vnašanju napačnih podatkov in podobno.

Obstajajo različne definicije o tem, kaj vse sodi v sistemsko testiranje. Spodaj so opisane najpogostejše metode, ki jih lahko opravljamo v sklopu sistemskega testiranja (Čebokli, 2006):

- Testiranje varnosti sistema – preverjamo zaščito sistema pred nepooblaščenim dostopom.
- Testiranje shranjevanja in uporabe varnostnih kopij – preverimo pravilnost delovanja varnostnih kopij sistema – ali se ustvari taka varnostna kopija, da z njo lahko ob težavah programsko opremo vrnemo v delujoče stanje.
- Testiranje okrevanja sistema – programsko opremo na različne načine prisilimo, da preneha delovati. Opazujemo, če sistem okreva po naših pričakovanjih.
- Testiranje obremenitve – sistem obremenimo s tolikšno količino transakcij, kot naj bi jih bilo v produkciji. Sistem postopoma obremenjujemo, dokler ne neha delovati.
- Stresno testiranje – sistem namenoma prekomerno obremenimo s transakcijami in mu postopoma odvzemamo vire do točke zloma. Želimo, da sistem pade, ampak da ni nobene okvare podatkov.
- Testiranje skladnosti – preverjamo skladnost programske opreme z drugo programsko opremo, strojno opremo, mrežno opremo itd.
- Testiranje postopka namestitve in odstranitve aplikacije – preverjamo namestitev delne ali celotne aplikacije v ciljno okolje.
- Testiranje primerljivosti – s programsko opremo tekmecev primerjamo prednosti in slabosti programske opreme, ki jo testiramo.

4.3.1 Funkcionalno testiranje

V okviru sistemskega testiranja izvajamo funkcionalno testiranje, ki je namenjeno validaciji programske opreme. Funkcionalni testi so testi črne skrinjice in jih izvajajo testerji prek uporabniškega

vmesnika. S funkcionalnimi testi želimo preveriti, ali programska oprema ustreza uporabniškim zahtevam.

4.4 Prevzemno testiranje

Prevzemno testiranje je podobno funkcionalnemu testiranju. Tudi tu testiramo, ali programska oprema deluje v skladu z uporabniškimi zahtevami. Razlika je v tem, da ga izvajajo končni uporabniki v okolju, podobnemu produkcijskemu. Zaradi prekrivanja funkcionalnega in prevzemnega testiranja lahko uporabimo iste teste.

4.5 Regresijsko testiranje

Regresijsko testiranje se lahko izvaja na katerikoli ravni testiranja. Izvajamo ga vedno, ko se modul programske opreme spremeni. Ključna ideja regresijskega testiranja je v tem, da preverimo, ali so spremembe v programski opremi privedle do napak. Preverjamo tudi, ali so se morda znova pojavile napake, ki so že bile odpravljene. Tako testiranje nam omogoča obvladovanje sprememb programske opreme. Pri tem ne načrtujemo novih testov, ampak samo poganjamo stare teste, ki so v preteklosti bili že uspešno izvedeni.

Predvsem pri tej vrsti testiranja je smiselna uporaba avtomatiziranih orodij, s katerimi lahko pogosteje izvajamo večje množice testov. Avtomatsko izvajanje regresijskih testov je koristno tako v fazi razvoja kot v fazi vzdrževanja programske opreme.

5 NAPAKE V PROGRAMSKI OPREMI

Napaka je nepravilno delovanje programa. Je prisotna vedno, kadar program ne deluje tako, kot pričakujejo končni uporabniki. O napakah govorimo (Zorko, 2004), kadar:

- program ne opravlja nečesa, kar navaja specifikacija.
- program izvaja nekaj, kar specifikacija pravi, da ne bi smel.
- program počne nekaj, česar specifikacija ne omenja.
- je program težko uporabljati, je počasen, v očeh uporabnika ni viden kot ustrezen.

Programska oprema je podrobno opredeljena oziroma opisana v specifikaciji, ki jo preda naročnik programske opreme. Napaka je vsakršno odstopanje od specifikacije. Pomembno je, da je specifikacija pravilno zapisana. Če je pomanjkljiva, ne moremo pričakovati, da bo program pravilno deloval.

Vsaka napaka v programu lahko povzroči veliko škod, zato je testiranje programa pomembno tudi med razvojem programske opreme, ne le na koncu razvoja. Le tako bomo preprečili težave pri delovanju programa. V naslednjem razdelku je navedenih nekaj programerskih napak, ki so povzročile veliko denarno škodo.

5.1 Programerske napake

Oglejmo si nekaj znanih primerov programerskih napak. Spisek je povzet po (Alley, 2009).

- »Millenium bug« ali problem leta 2000 – Ob prehodu v leto 2000 se je veliko govorilo o tej napaki. Napaka naj bi bila posledica napačnega načrtovanja programske opreme. S 1. 1. 2000 naj bi računalniki napačno obračunavali datume in ure. Posledica je bila panika po vsem svetu, zato je prišlo do množičnega posodabljanja sistemov. Stroški so bili ocenjeni na 300 milijard dolarjev. Kasneje se je izkazalo, da so nekateri strahovi bili neutemeljeni, in da je večina programske opreme prehod v leto 2000 preživela brez težav.

- »North America blackout« – 14. 8. 2003 je v delih severovzhodne Amerike in Kanade prišlo do masovnega izpada električne energije. Napaka je bila v programski opremi, ki nadzira omrežje. Po ocenah je ostalo brez elektrike 50 milijonov ljudi, stroške pa so ocenili na 6 milijard dolarjev.

- Raketa Ariane 5, Polet 501 – 4. 6. 1996, 37 sekund po vzletu rakete Ariane 5, je zaradi napake v delovanju krmilnega sistema prišlo do eksplozije. Napaka je bila ena od najdražjih napak v zgodovini.

- Deljenje s plavajočo vejico pri procesorjih Intel Pentium – 30. 10. 1994 je profesor Thomas Nicely z univerze v Lynchburgu odkril napako v procesorjih Pentium, ko so ti izvajali deljenje s plavajočo vejico. Stroški zamenjave procesorjev so dosegli 475 milijonov dolarjev.

- Napaka DURS (12. 2. 2007) – Na državnem davčnem uradu so potrdili ugotovitve Računskega sodišča Republike Slovenije, da je zaradi napake v informacijskem sistemu Davčne uprave Republike Slovenije (DURS) država pri dohodnini za 2005 obračunala zavezancem okrog 74 milijonov tolarjev preveč, slabih 18 milijonov tolarjev pa premalo. Okoli 11 tisoč ljudi bo dobilo nove, nadomestne odločbe. Računsko sodišče za napako poleg davčne uprave krivi zunanjega izvajalca, podjetje RRC, kjer pa odgovornost zavračajo (24UR, 2007).

5.2 Programski hrošč

Hrošč (angleško bug) je izraz za programsko napako v računalniškem programu, ki jo programer med svojim delom ni odkril. To je lahko napaka v algoritmu, sintaktična napaka, aritmetična napaka itd.

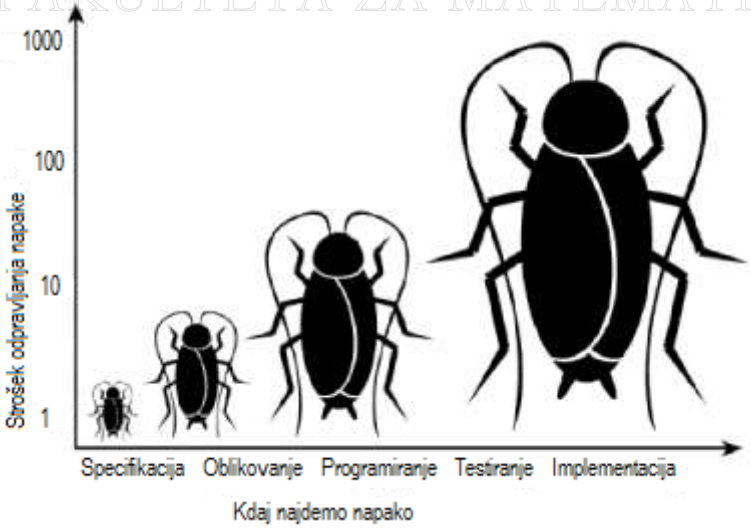
Nekatere napake povzročajo napačno oziroma nepredvidljivo delovanje programa, nekatere pa sesutje programa. Proces iskanja napak se imenuje razhroščevanje.

Na slovnične napake kode nas opozori prevajalnik. Preostale napake nastopijo v času izvajanja programa.

Kaner, Falk in Nguyen so napake v programski opremi razdelili v 13 kategorij (Ross, 1998):

- napake v uporabniškem vmesniku – programska oprema nima take grafične oblike, kot jo navaja specifikacija;
  Primer: V specifikaciji piše, da naj bo barva pisave temno siva, vendar je barva pisave rdeča.
- ravnanje z napakami – programer lahko na napačen način obravnava ugotovljene napake;
  Primer: V aplikaciji se pojavi napaka. Programer jo reši (namesti popravek), vendar se zaradi njegovega popravka pojavi nova napaka, ki je programer ni predvidel.
- kontrola nad verzijami in kodo – uporabljamo zastarele programe, ko že imamo na voljo njihove popravke, torej nove različice programa;
- napake pri obdelavi in interpretaciji podatkov – prenos podatkov med staro in novo verzijo programske opreme lahko povzroči napake;
- napake v povezavi z mejno vrednostjo – obravnavanje mejnih vrednosti je lahko nepravilno;
- nadzor nad potekom logičnih poti – napake zaradi napačnega vrstnega reda stavkov, napačen vrstni red klicanih funkcij;
- pogoji izbire – če naj se izvedeta dve nalogi, moramo določiti, katera ima prednost, ker se lahko zgodi, da se nalogi izvedeta v napačnem vrstnem redu in pride do nepravilnih rezultatov;
- obremenitvene zmožnosti – pri programski opremi, ki deluje na robu svojih možnosti, se začnejo pojavljati napake;
- računske napake – računski in logični izračuni so lahko napačni;
- začetne napake – funkcija se napačno obnaša le ob prvi izvedbi;
- strojne napake – komunikacija s strojno opremo pod določenimi pogoji ne deluje pravilno;
- dokumentacija – uporabniku ni na voljo, v priročniku opisan postopek;
- napake pri testiranju – testerji delajo napake pri procesu testiranja, saj mislijo, da je za nepravilno delovanje kriva programska oprema.

Pri odkrivanju programskih napak je zelo pomemben čas odkritja oziroma v kateri fazi razvoja programske opreme je napaka odkrita. V zgodnjih fazah razvoja programske opreme je strošek odkrite napake zanemarljiv, kar prikazuje slika (Slika 6). Napaka, odkrita v fazi programiranja ali testiranja, lahko povzroči strošek, ki je desetkrat ali stokrat večji od začetnega. Najvišji strošek pa nastane, ko napako odkrijejo končni uporabniki.



Slika 6: Strošek odpravljanja napake glede na časovno periodo [Vir slike: (Patton, 2005)]

5.3 Ravnanje z napakami

Obstaja mnogo različnih tipov napak in načinov, kako se z napakami spoprimemo (Slika 7). Zato bom omenila le tiste, s katerimi sem se srečala, oziroma se srečujem, in ki jih uporabljam.

- Izogibanje napakam

Napakam, ki se pojavijo, še preden je sistem nameščen v produkcijsko okolje, se lahko izogibamo, tako, da uporabimo primerno programersko metodologijo, ki zmanjša kompleksnost programske opreme. Izvajamo nadzor nad verzijami, da preprečimo nekonsistentnost programske opreme in uporabimo verifikacijo, da se izognemo algoritmičnim napakam oziroma hroščem.

- Odkrivanje napak

Programsko opremo testiramo na vseh nivojih (testiranje modulov, integracijsko in sistemsko testiranje). Iščemo napake.

Če se napaka pojavi, ko je sistem že izdan, jo moramo odpraviti (razhroščevanje). Razhroščevanje (angleško debugging) ali popravljanje napak ne smemo zamenjavati s testiranjem. Testiranje pokaže na napake, razhroščevanje pa mora poiskati njihove vzroke in jih odpraviti (Solina, 1997).

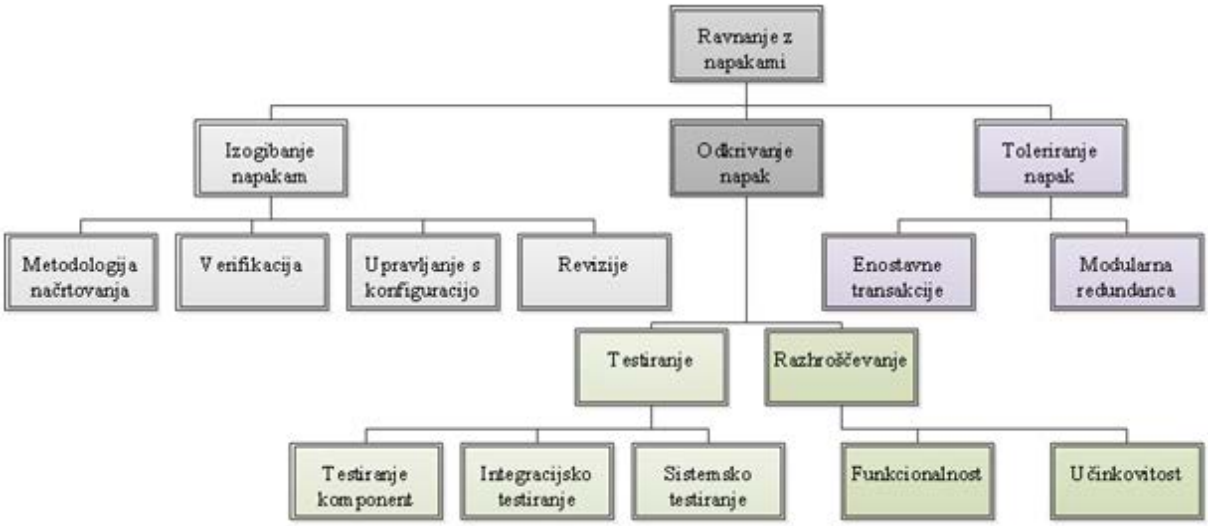
Odpravljanje napak zahteva veliko podatkov. Pridobiti moramo podatke o tem, kdaj, kje in kako se je napaka pojavila. Prvi korak pri odpravljanju napak je preverjanje ponovljivosti napake. Če napake ne uspemo ponoviti, bomo tudi vzrok za napako težko poiskali.

Ko ugotovimo vzrok napake, poiščemo potencialno rešitev. Le-to testiramo z različnimi testnimi scenariji in nadzorujemo, kako se programska oprema obnaša.

- Toleriranje napak

Če napake zaradi kateregakoli vzroka ne uspemo odpraviti, naredimo seznam znanih napak v programski opremi, ki ga priložimo uporabniškim navodilom. Po potrebi pripravimo scenarij za varno uporabo produkta.

Spodnja slika prikazuje razumno ravnanje z napakami.



Slika 7: Ravnanje z napakami [Vir slike: (Podgorelec, 2007)]

6 AVTOMATIZACIJA TESTIRANJA

V zadnjih nekaj letih želi veliko podjetij pri razvoju programske opreme doseči večjo kakovost, zato so začela v razvoj programske opreme uvajati metode za upravljanje projektov. Cilj je, da se upravljanje z razvojem programske opreme čim bolj avtomatizira. Pri tem je velika pozornost posvečena testiranju programske opreme. Slednje je še pomembnejše, ker so programi čedalje bolj kompleksni. Zato je možnost, da vsebujejo napake, toliko večja. Ker nasploh težimo k uporabi avtomatiziranih postopkov, želimo tudi pri testiranju uporabljati avtomatizirana orodja.

Testiranje lahko izvajamo ročno ali z uporabo orodij za testiranje. Ročno testiranje je izvajanje in preverjanje programske opreme brez uporabe orodij za avtomatizacijo testiranja. Orodje za testiranje je sredstvo, ki ga uporabljamo za izvajanje testnih primerov.

V zadnjih nekaj letih se na trgu pojavlja široka paleta testnih orodij, ki nam lahko pomagajo izboljšati učinkovitost testiranja. Še zmeraj pa je najbolj uporabljena tehnika ročno testiranje. Z njim se sreča vsak, ki dela na področju razvoja programske opreme. Tudi po končanem razvoju programske opreme je ročno testiranje skoraj neizogibno. Ročno testiramo tam, kjer (Poljšak, 2005):

- je razvoj programske opreme v začetni fazi;
- ni dovolj časa za avtomatizacijo;
- ni izkušenega kadra, ki bi obvladoval avtomatizacijo;
- avtomatizacija ni smiselna zaradi prevelikih stroškov in neponovljivosti;
- testiramo uporabniški vmesnik ali
- poskušamo napako ponoviti, da lahko določimo njen izvor.

Velikokrat, še posebej v zgodnjem razvoju programske opreme, se dogaja, da je potrebno teste večkrat ponavljati, kar zahteva veliko časa. Danes obstaja mnogo načinov, da se delo poenostavi in čas testiranja skrajša. Eden od takšnih načinov je avtomatizacija testiranja.

Prednosti avtomatizacije testiranja so (Kaner, 1998):

- hitrost – sodobna orodja omogočajo obdelavo ogromne količine podatkov. Na primer z ročnim testiranjem lahko vnesemo v aplikacijo le podatke o eni osebo, z orodjem pa lahko brez posebnega napora v istem času vnesemo podatke deset različnih oseb;
- učinkovitost – če testiramo ročno, v tem času ne moremo delati nič drugega. Z avtomatizacijo prihranimo čas, ki ga lahko porabimo na primer za načrtovanje testiranja;
- točnost in natančnost – ko izvedemo na stotine testov, človekova koncentracija pade, pri testiranju se začnejo pojavljati napake, medtem ko orodje isti test izvaja in preverja rezultate z vedno isto natančnostjo in točnostjo;
- vztrajnost – testno orodje se nikoli ne utrudi, teste lahko opravlja vedno znova.

Ker je na tržišču ogromno testnih orodij, je zelo pomembno, da se v primeru odločitve o avtomatizaciji testiranja pravilno odločimo glede izbire testnega orodja. Testna orodja se med seboj razlikujejo med drugim tudi po tem, kateremu delu razvoja programske opreme so prvenstveno namenjena. Zato moramo biti pozorni na to, da izberemo pravo orodje glede na fazo razvojnega procesa programske opreme, ki jo testiramo. Poleg tega nekatera orodja zahtevajo veliko programerskega znanja, druga pa so namenjena praktično vsem, ki imajo vsaj nekaj izkušenj s testiranjem. Vsekakor mora biti orodje prilagojeno testni skupini, njenemu znanju in izkušnjam.

Razen pozitivnih stvari, kot so vse prej naštete prednosti avtomatizacije testiranja, pa priprava na avtomatizacijo zahteva pazljivo načrtovanje in veliko časa. Veliko časa gre tudi za pripravo testnih primerov. Te moramo spreminjati in preverjati ob vsaki spremembi, ki jo razvijalci naredijo v programu. Razmisliti moramo, koliko časa bodo testni primeri primerni za testiranje, saj jih moramo ob spremembah oziroma po dodatnemu razvoju spet popraviti in prilagoditi.

Ne moremo trditi, da je ročno testiranje slabše od avtomatskega ali obratno. Za nobeno tehniko testiranja ne obstaja zagotovilo, da bomo z njo odkrili vse napake. Za katero možnost se bomo odločili, moramo presoditi sami. Pri tem upoštevamo razvojno fazo testirane programske opreme, testno okolje in še mnogo drugih dejavnikov. Kombinacija ročnega in avtomatskega testiranja je trenutno še zmeraj najboljša in tudi najugodnejša rešitev.

Avtomatsko testiranje delimo na dve večji področji:

- testiranje modulov in
- avtomatski funkcionalni sistemski testi.

Testiranje modulov je skoraj vedno avtomatizirano. Več o testiranju modulov smo že prikazali v četrtem poglavju (POSTOPKI TESTIRANJA).V nadaljevanju se bom osredotočila na avtomatsko funkcionalno sistemsko testiranje.

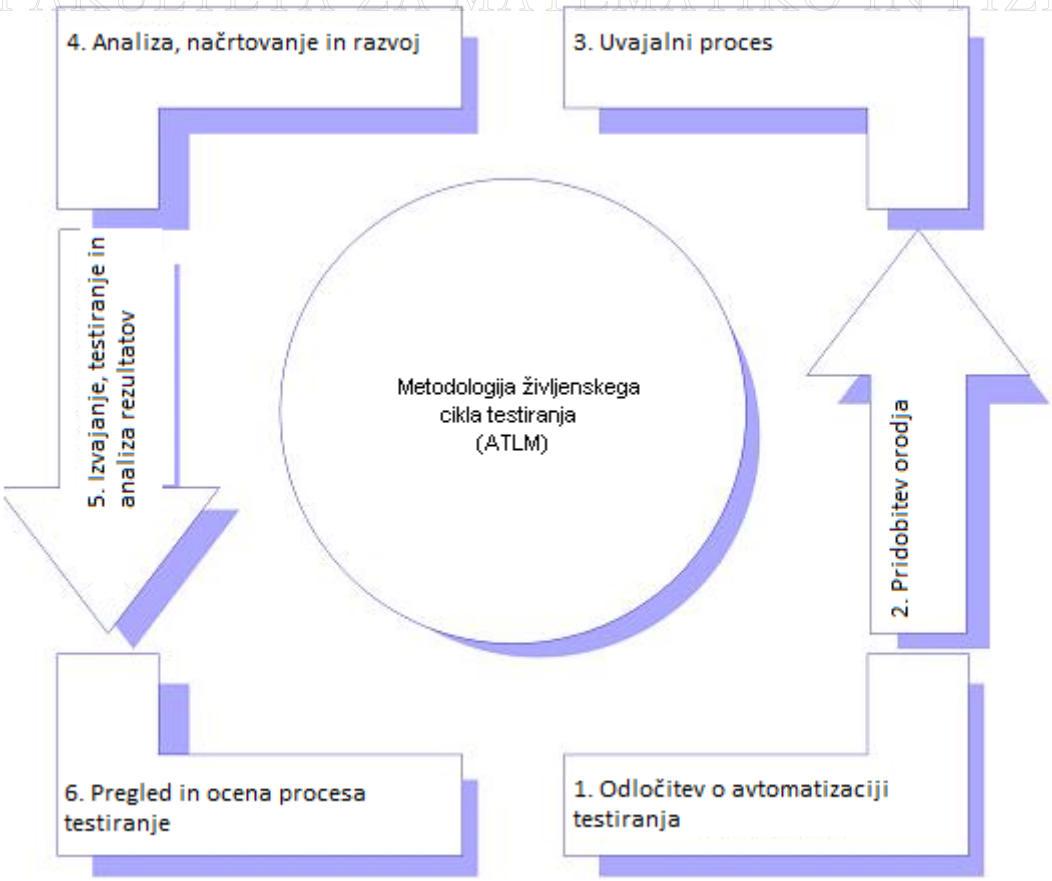
Avtomatizacija funkcionalnih testov temelji na posnemanju človekovega testiranja s testnim orodjem (glej razdelek Testna orodja za funkcionalno testiranje). Kot sem povedala že prej, so tovrstna orodja draga, zato je potrebno dobro pretehtati stroške in koristi avtomatizacije testiranja. Potrebno je izbrati pravo orodje, ki bo skladno z drugimi razvojnimi orodji, in se avtomatizacije lotiti na pravi način. Eden od načinov, kako se pravilno lotiti uvajanja avtomatizacije testiranja, je uporaba strukturirane metodologije ATLM (angleško Automated Test Lifecycle Methodology), ki nam je pri uvedbi avtomatizacije testiranja lahko v veliko pomoč.

6.1 Strukturirana metodologija ATLM

Prehod z ročnega na avtomatsko testiranje je uvedba sprememb v celotnem življenjskem ciklu programske opreme. Za izvedbo tega prehoda lahko uporabimo pristop po metodologiji ATLM. ATLM (angleško Automated Test Lifecycle Methodology) je namenjena procesu testiranja, ki teče vzporedno z razvojnim življenjskim ciklom programske opreme. V njej je zajet celovit pristop, od testiranja modulov do sistemskega testiranja. Poudarjena so tveganja in koristi avtomatizacije. Posebej so obravnavani načrtovanje, analiza, izvajanje in upravljanje procesa testiranja.

Metodologija ATLM se ne nanaša na konkretno testno orodje in ne priporoča uporabe določenega orodja. Je neodvisna od razvojne faze programske opreme in se jo lahko uporabi v katerikoli razvojni fazi.

Po tej metodologiji poteka prehod z ročnega na avtomatsko testiranje skozi šest faz in množico aktivnosti (Slika 8).



Slika 8: Metodologija ATLM [Vir slike: (Dustin, 1999)]

1. Odločitev o avtomatizaciji testiranja

Odločitev o avtomatizaciji testiranja predstavlja prvo fazo metodologije ATLM. V tej fazi moramo predvsem pridobiti soglasje vodstva za vpeljavo avtomatizacije v postopek testiranja programske opreme. Zelo pomembno je, da v tej fazi ekipa, ki pripravlja prehod na avtomatsko testiranje, skupaj z vodstvom pretehta pričakovanja o koristih, ki naj jih prinese avtomatizacija. Treba je oceniti stroške avtomatizacije, ki zajemajo tudi stroške nakupa orodja. Orodje mora ustrezati tako izkušnjam testerjev kot tudi programski opremi.

Zavedati se moramo, da prehod pomeni, da bo v začetku več stroškov kot koristi. Vendar je na odločitev potrebno gledati kot na dolgoročno naložbo. Poleg stroškov za nakup orodja so stroški povezani z morebitno zaposlitvijo novih ljudi na projektu in stroški izobraževanja za uporabo novih postopkov.

Večkrat se zgodi, da ima vodstvo nemogoča pričakovanja o avtomatizaciji testiranja, zato se mora zavedati, da:

- ni orodja, ki bi podprlo celoten proces testiranja;
- ne bodo vsi testi avtomatizirani;
- ne bo takojšnjega prihranka v času in pri zaposlenih;
- v začetku se avtomatski testi ne bodo izvajali ponoči oziroma zunaj delovnega časa;
- takoj ne bo vidnih še mnogo drugih koristi.

Šele po celoviti presoji se odločimo ali bomo avtomatizacijo uvedli ali bomo uvedbo opustili.

2. Izbor orodij

Ta faza se začne, ko imamo podporo vodstva za uvedbo avtomatizacije. Odločamo se, katero orodje je najprimernejše za naše potrebe.

Orodje mora biti prilagojeno razvojnemu orodju, ki ga razvijalci že uporabljajo. Na podlagi obstoječih razvojnih orodij določimo potencialna orodja za podporo testiranju. Določimo tudi merila izbora, ki bodo osnova za izbor orodja. Merila so na primer cena, združljivost, uporabnost na različnih projektih, zahtevnost uporabe itd. Za uspešen izbor orodja morajo biti merila izbora jasno določena.

Idealno bi bilo, da bi pred izbiro vsako orodje lahko nekaj časa preizkušali, kar žal ni možno pri vseh orodjih. Pogosto je tudi preizkusni čas zelo kratek. Če imamo možnost preizkusa, poskušamo čim bolj preveriti ustreznost orodja. To pomeni, da preverjamo elemente, ki smo jih določili v merilih izbora. Za vsako potencialno orodje določimo koristi in njegove slabosti. Potem izberemo tisto orodje, ki je najprimernejše.

3. Uvajalni proces

Ko smo v tej fazi, pomeni, da smo se že odločili, katero testno orodje bi bilo najbolj primerno za vpeljavo v podjetju. Dokončno izbiro orodja smo lahko tudi odložili do te faze. Zdaj začnemo z vpeljavo avtomatskega testiranja k obstoječemu ali novemu razvojnemu procesu. Udeleženci morajo dojeti, da gre dejansko za proces vpeljave orodja.

Najprej analiziramo obstoječi proces testiranja, če se seveda že izvaja. Če ročnega testiranja še nismo izvajali, začnemo na novo z novim procesom. Če obstoječi proces ni dobro definiran, se je potrebno najprej osredotočiti na definicijo tega. Dokumentacija naj bo razumljiva za vse udeležene. Definirati je potrebno strategijo, namene in cilje testiranja. Izbrati je potrebno primerno tehniko testiranja.

V tretji fazi dejansko preverimo ustreznost izbranih orodij. Preverimo, ali je orodje primerno glede na izkušnje testerjev, ali se orodje prilagaja drugim razvojnim orodjem, ali imamo dovolj časa na voljo za vpeljavo orodij, ali bo potreben dodaten kader itd.

Orodje preizkusimo na obstoječi testni aplikaciji ter tako preverimo združljivost in opazimo morebitne težave. Če ugotovimo, da je orodje primerno, nadaljujemo s četrto fazo. Če orodje ni primerno, nadaljujemo z iskanjem orodja. Če primernega orodja ne najdemo, iskanje opustimo in testiranje izvajamo ročno.

4. Analiza, načrtovanje in razvoj testov

Pred fazo načrtovanja testnih primerov je treba izvesti analizo zahtev testiranja in jih dokumentirati. Zahteve izhajajo iz zahtev testirane programske opreme. Analiza zahtev testiranja vključuje testne scenarije, teste združljivosti (testiramo aplikacijo na strojni opremi, na kateri bo tekla), izpostavo programske opreme nepredvidljivemu obnašanju uporabnika (lahko se zgodi, da uporabnik ne bo sledil navodilom za uporabo programske opreme).

Po definiranju zahtev testiranja se začne načrtovanje testnega načrta. Načrtovanje zajema obravnavo vseh aktivnosti, potrebnih za izvajanje testiranja in verifikacijo. Obravnava tudi organiziranost in učinkovito uporabo procesov, osebja, orodja in opreme. S prej zapisanega se oblikuje testni načrt, ki ga začnemo oblikovati že v predhodnih fazah in ga sproti dopolnjujemo. V testnem načrtu so zajete tudi:

- vloge in odgovornosti udeležencev projekta,
- doseg testnega procesa glede na omejitve v času in pri zaposlenih,
- zahteve za testno okolje,
- uporabljene metode testiranja,
- časovni načrt za razvoj in izvajanje testnih primerov itd.

Osnutek splošnega testnega načrta, kot ga priporoča standard ANSI/IEEE 829-1983 (Software Test Documentation).

TESTNI NAČRT

1. Identifikator testnega načrta
2. Uvod
3. Opis produkta
4. Karakteristike, ki bodo testirane.
5. Karakteristike, ki ne bodo testirane.
6. Uporabljene metode testiranja in terminalna kriterijska funkcija
7. Odpovedna kriterijska funkcija
8. Prekinitev in nadaljevanje testiranja
9. Identifikacija izročljivih dokumentov
10. Potrebne aktivnosti za izvedbo testiranja
11. Opis potrebnega testnega okolja
12. Identifikacija odgovornosti
13. Človeški viri in potrebno šolanje
14. Roki
15. Tveganja in posledice
16. Potrditev testnega načrta
17. Dodatki

Načrtovanje zajema tudi načrtovanje testnih primerov oziroma testnih scenarijev. Za vsak testni primer je treba določiti ali gre za ročni ali avtomatski test. Dokončno definiramo standarde kodiranja, ki jih uporabljamo za razvoj programske opreme, ter pristop pri izdelavi programske kode testov. Definicija testnega primera med drugim zajema osnovne, kot so ime, avtor, in druge podatke, kot so vhodni podatki, izhodni podatki, potrebne akcije itd. Podrobnost definicije je odvisna od zapletenosti testnega primera.

5. Izvajanje testiranja in analiza rezultatov

V fazi izvajanja testov se izvedejo predhodno pripravljeni testni primeri. Dobljeni rezultati se zbirajo in analizirajo. Testiranje izvajamo na vseh ravneh, od testiranja modulov do sistemskega testiranja. Po izvedenem testiranju se beležijo odkrite napake. Rezultati testov so lahko naslednji:

- Pozitivni rezultati – test se je izvedel uspešno oziroma, kot smo pričakovali. Dobili smo pričakovane rezultate. Kaže, da testirana komponenta oziroma programska oprema deluje pravilno.

- Negativni rezultati – test se je izvedel neuspešno. Rezultati niso takšni, kot smo predvideli (zahtevali). Testirana komponenta oziroma programska oprema torej ne deluje pravilno.

- Lažni pozitivni rezultati – test se je izvedel uspešno, čeprav testirana komponenta ne deluje pravilno. Razlogi so lahko v napačnem razpoznavanju objektov, napačnem branju rezultatov itd. Takšne teste moramo popraviti. Če je potrebno, moramo tudi dodelati kodo testnih primerov.

  Primer:
  Pri pripravi avtomatskega testiranja je predvideno, da se v določenem trenutku pojavi opozorilno okno, pri katerem uporabnik klikne na gumb *DA*. Pogosto testna orodja pri pripravi tega testnega scenarija ne posnamejo sporočila, ki se pojavi v opozorilnem oknu. Zato se lahko zgodi, da se pojavi opozorilno okno s sporočilom, ki ni enako posnetemu sporočilu. Če je posneto v scenariju, da uporabnik klikne *DA*, se bo to zgodilo tudi v tem primeru. Scenarij se bo sicer uspešno izvedel, a dejansko je bil potem napačen. Da se taki napaki izognemo, moramo test dodelati. To pomeni, da poskrbimo za to, da preberemo sporočilo v opozorilnem oknu (kodo dodelamo tako, da bo prebrala sporočilo) in ga primerjamo s pričakovanim sporočilom. Če je sporočilo enako predvidenemu, se test izvede do konca, sicer naj se test ustavi.

- Lažni negativni rezultati – test se je izvedel neuspešno, čeprav v resnici testirana komponenta deluje pravilno. Po izvedbi testiranja dobljeni rezultat primerjamo s pričakovanim. Če test ni uspešen, ne pomeni nujno, da testirana komponenta ne deluje pravilno. Razlog so lahko tudi napake v testnih primerih.

Ko začnemo testno orodje dejansko uporabljati, šele vidimo, kako se obnaša v različnih situacijah. Takrat mu moramo za vsako kritično točko povedati, kaj naj naredi. Lažnim rezultatom se bomo izognili samo ob spremljanju delovanja testnega orodja in sprotnemu odpravljanju kritičnih točk. Zelo je pomembno, da prvih nekaj izvajanj testne skripte nadzorujemo (to pomeni, da dobesedno gledamo, kako se testna skripta izvaja).

Za potrebe analize rezultatov testiranja lahko uporabimo različne metrike. Z njimi beležimo kazalce o napredovanju testiranja in stopnji pokrivanja testov (koliko testnih scenarijev, zapisanih v specifikaciji, izvajamo). Ker je možnih metrik veliko in vsaka analiza zahteva čas, moramo smiselno izbrati tiste, ki nam dajo največ informacij. Primeri metrik so:

- pokrivanje testiranja – primerjamo število izvedenih testnih primerov s številom testnih scenarijev, ki so zapisani v specifikaciji;
- gostota napak – število vseh najdenih napak v primerjavi s številom izvedenih testnih primerov;
- trenutna stopnja kakovosti – število uspešnih testnih primerov v primerjavi s številom izvedenih testnih primerov.

6. Pregled in ocena procesa testiranja

Šesta faza se izvaja skozi celotni življenjski cikel testiranja. Naš cilj je nenehno iskanje izboljšav procesa. Med procesom in predvsem po izvajanju testnih primerov je potrebno upoštevati izbrane metrike ter analizirati rezultate. Pri tem metodologija ATLM predlaga naslednje korake:

- Po izvajanju testiranja mora testna skupina spremljati učinkovitost testnega orodja in preučiti, kje so možne in potrebne spremembe, da bi lahko proces izboljšali.

- Rezultati testiranja povedo, ali je aplikacija zrela za produkcijo.

- Vse zbrane izkušnje, tako negativne kot pozitivne, glede procesa testiranja, mora testna skupina dokumentirati.

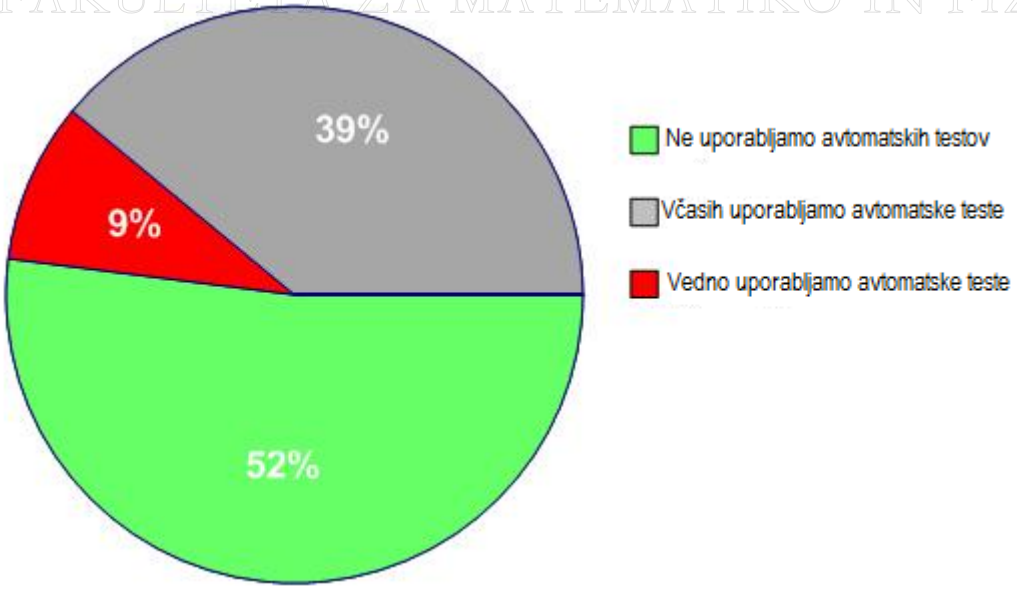
- Vsa dokumentacija skozi celotni življenjski cikel naj bo dokumentirana in shranjena v lahko dostopnem repozitoriju.

- Izračunati je treba koristi avtomatizacije. Podatke za ta izračun zbiramo že med procesom testiranja.

Metodologija ATLM daje precej poudarka na dokumentiranju in formalizaciji procesa testiranja. Je edina metodologija, ki obravnava celovit pristop pri uvedbi avtomatizacije testiranja. ATLM obravnava avtomatizacijo testiranja na vseh ravneh testiranja. Ker je testiranje modulov skoraj vedno avtomatizirano, daje ATLM poudarek predvsem avtomatizaciji funkcionalnega testiranja.

6.2 Stroški in koristi avtomatizacije

Vsaka vpeljava nove metodologije prinese v začetku več stroškov kot koristi. Enako velja za vpeljavo testnega orodja oziroma avtomatizacijo testiranja. Vendar je potrebno na nove metodologije gledati kot na dolgoročno naložbo. Ravno zaradi strahu pred visokimi stroški vpeljave avtomatizacije je še vedno najpogostejše ročno testiranje (Slika 9).



Slika 9: Rezultati ankete o uporabi avtomatskih testov [Vir slike: (Schwaber, 2006)]

Podjetje Forrester Research je naredilo raziskavo o uporabi avtomatskih testov. Raziskavo so izvajali v Severni Ameriki. Ugotovili so, da le 9 odstotkov podjetij, ki delujejo na področju razvijanja programske opreme, avtomatske teste uporablja vedno, 39 odstotkov jih uporablja občasno. Drugi avtomatskih testov ne uporabljajo.

Koristi vpeljave avtomatskega testiranja lahko merimo tudi v luči merila, ki mu pravimo povračilo naložbe. S povračilom naložbe (angleško Return of investment – v nadaljevanju ROI) merimo učinkovitost investicije. Vrednost koristi investicije delimo z vrednostjo, ki smo jo porabili za investicijo (Hoffman, 1999). Če je rezultat ena, to pomeni, da so dobljene koristi enake stroškom, torej je stanje ostalo nespremenjeno. Če je rezultat večji od ena, pomeni, da so se stroški testiranja znižali, saj smo pridobili večjo korist, kot smo imeli stroškov. Če pa je rezultat manjši od ena, so se stroški z uvedbo avtomatizacije povečali.

S primerjavo stroškov in koristi iz obdobja pred uvedbo avtomatizacije dobimo povračilo naložbe. Predpostavimo, da je ročno testiranje že vpeljano. Stroške vpeljave avtomatizacije lahko delimo na fiksne in spremenljive. Fiksni stroški so stroški nakupa testnega orodja, morebitne dodatne opreme, stroški usposabljanja itd. Spremenljivi stroški so odvisni od števila testov in pogostosti njihovih izvajanj. V spremenljive stroške uvrščamo tudi stroške načrtovanja, izdelave, izvajanja in analize testov. Mnogo stroškov je skupnih tako ročnemu kot avtomatskemu testiranju. Na primer izdelava načrtov testov je koristna tudi pri ročnem testiranju, zato ta strošek lahko izpustimo.

Za izračun določimo časovni okvir (**t**). Vzamemo na primer obdobje med dvema različicama programske opreme.

Upoštevati je treba amortizacijo pri dejavnikih, kjer so stroški fiksni. Če je na primer cena orodja 20.000 denarnih enot in pričakujemo, da bo orodje uporabno pet let, nas orodje v prvem letu stane 20.000/5, kar znese 4.000 denarnih enot. Če orodje uporabljamo samo prvo leto, gredo vsi stroški v prvo leto.

Najpomembnejša dejavnika izračuna sta pričakovano število izvajanj avtomatskih testov (**n1**) v primerjavi s pričakovanim številom izvajanj ročnih testov (**n2**). Avtomatske teste lahko večkrat izvajamo. Vendar je korist ponovnega zagona avtomatskih testov največja po spremembi testirane programske opreme.

Pomemben dejavnik je tudi vzdrževanje avtomatskih testov. Vprašanje je, kolikokrat lahko test poženemo, preden ga moramo popraviti (**N**). Izkaže se, da je pri uporabi pristopa posnemi in predvajaj (glej razdelek 6.3) ta dejavnik blizu ena. To nam pove, da je tovrstne testne skripte težavno vzdrževati.

Obrazec, ki podaja izračun povračila naložbe (ROI) na podlagi primerjav med ročnim in avtomatskim testiranjem (Hoffman, 1999).

$$ROI(t) = \frac{\Delta(Koristi\ avtomatizacije\ v\ primerjavi\ z\ ročnim\ testiranjem)}{\Delta(Stroški\ avtomatizacije\ v\ primerjavi\ z\ ročnim\ testiranjem)} = \frac{\Delta B_a}{\Delta C_a}$$

Vrednosti v obrazcu so naslednje:

ΔB_a (t) = $\sum(zmanjšani\ fiksni\ stroški\ zaradi\ avtomatizacije)$

+ $\sum(spremenljivi\ stroški\ poganjanja\ ročnih\ testov\ n_2\ med\ časom\ t)$

- $\sum(spremenljivi\ stroški\ poganjanja\ avtomatskih\ testov\ n_1\ med\ časom\ t)$

ΔC_a (t) = $\sum(povečani\ fiksni\ stroški\ zaradi\ avtomatizacije)$

+ $\sum(spremenljivi\ stroški\ izdelave\ avtomatskih\ testov)$

- $\sum(spremenljivi\ stroški\ izdelave\ ročnih\ testov)$

+ $\sum(spremenljivi\ stroški\ vzdrževanja\ avtomatskih\ testov)$

Oglejmo si primer takega izračuna. Navedimo podatke, potrebne za izračun:

- testiramo novo aplikacijo brez obstoječih testov;
- 5 let za razvoj ročnih testov;
- 15 let za razvoj avtomatskih testov;
- en človek za vzdrževanje avtomatskih testov po prvem letu;
- deset ljudi za izvajanje ročnih testov, en človek za poganjanje avtomatskih testov;
- 90.000 DE (denarnih enot) fiksnih stroškov za avtomatske teste z uporabno življenjsko dobo treh let;
- časovno obdobje (**t**) je 12 mesecev in 24 mesecev;
- cena zaposlenih 100.000 DE na leto = 400 DE na dan = 50 DE na uro.

Na podlagi zgornjega primera izračunamo stroške in koristi.

ΔB_a (v 12 mesecih) = 0 + (10 ljudi * 100.000 DE) – (1 človek * 100.000 DE) = 900.000 DE

ΔB_a (v 24 mesecih) = 0 + (10 ljudi * 200.000 DE) – (1 človek * 200.000 DE) = 1.800.000 DE

ΔC_a (v 12 mesecih) = (900.000 DE * (1/3)) + (15 ljudi * 100.000 DE) – (5 ljudi * 100.000 DE) + 0 DE = 30.000 DE + 1.500.000 DE – 500.000 DE = 1.030.000 DE

ΔC_a (v 24 mesecih) = (900.000 DE * (2/3)) + (15 ljudi * 100.000 DE) – (5 ljudi * 100.000 DE) + 100.000 DE = 30.000 DE + 1.500.000 DE – 500.000 DE = 1.160.000 DE

ROI (v 12 mesecih) = 900.000 DE/1.030.000 DE = 0.874 (majhna izguba)

ROI (v 24 mesecih) = 1.800.000 DE/1.160.000 DE = 1.552 (55-odstotno povračilo)

Navedeni primer ni resničen, vendar se rezultati ujemajo z izkušnjami v praksi. Vložek se praviloma povrne v daljšem časovnem obdobju.

6.3 Testna orodja za funkcionalno testiranje

Najbolj razširjena orodja za avtomatsko funkcionalno testiranje so orodja proizvajalcev HP, IBM, Borland, Compuware in Oracle.

Spomnimo se, da je funkcionalno testiranje dejansko način posnemanja uporabe določene aplikacije, zato imajo navadno vsa orodja za funkcionalno testiranje funkcijo posnemi in predvajaj. S klikom na gumb posnemi posname naš sprehod skozi aplikacijo. Ta sprehod predstavlja en testni primer. S klikom na gumb predvajaj lahko takšen test izvedemo – posneti sprehod se natančno ponovi. Skripte, ki jih posnamemo s takšnimi orodji, imajo nestrukturirano kodo z nespremenljivimi podatki. Občutljive so na najmanjše spremembe obnašanja testirane aplikacije (na primer gumb premaknjen bolj v levo, dodano vnosno polje itd.). Ena od prednosti takih orodij je hitrost izdelave testa.

Orodja, zasnovana na takšen način, so namenjena prav vsakemu testerju, tudi tistim, ki nimajo programerskega znanja. V zadnjih letih pa se pojavljajo testna orodja, ki nudijo mnogo več kot samo snemanje in izvajanje testnih skript. Eno od teh orodij je Oracle OpenScript, ki ga bom natančneje opisala v razdelku 7.1.

Novejša orodja za funkcionalno testiranje so objektno usmerjena. Z njimi posnamemo testni scenarij v obliki testne skripte. Testne podatke črpa iz vnaprej pripravljene podatkovne baze (glej razdelek 6.3.2). Ob snemanju skripte orodje posname vsako akcijo uporabnika (to je vnos besedila v vnosno polje, klik na gumb itd.), zato da lahko med predvajanjem testa natančno ponovi vsako akcijo. Lastnosti (podrobneje razložene v nadaljevanju) takih orodij so:

- razpoznavanje objektov,
- sinhronizacija testiranja,
- branje podatkov iz vnaprej določene podatkovne baze,
- preverjanje rezultatov,
- vizualizacija kode.

6.3.1 Razpoznavanje objektov in sinhronizacija testiranja

Med izvajanjem testa orodje prepozna elemente uporabniškega vmesnika, kot so gumbi, okna in podobno. Pri starejših orodjih je med izvajanjem lahko prišlo do težav s sinhronizacijo. Skripta je lahko prehitevala in izvedla akcijo (na primer pritisk na gumb), še preden se je ta pojavila na vnosni maski.

6.3.2 Branje podatkov iz vnaprej določene podatkovne baze

Nekatera orodja nam omogočajo, da testne podatke beremo iz vnaprej pripravljene podatkovne baze. Med izvajanjem testa se podatki, ki jih vnašamo v vnosno polje, nadomestijo s podatki iz podatkovne baze. Črpamo jih lahko zaporedno in naključno. S tem omogočimo hitro izvajanje testnih primerov z različnimi testnimi podatki.

6.3.3 Preverjanje rezultatov

Po vsakem testu, ki se izvede, želimo vedeti ali je bil test uspešen ali neuspešen. Običajno imajo testna orodja posebno okno, kjer lahko po končanem izvajanju testa vidimo poročilo testa. Kakšen bo prikaz rezultata testne skripte, je odvisno od načina prikaza rezultatov posameznega testnega orodja. Različna testna orodja na različne načine prikazujejo rezultate.

Priporočljive in zaželene informacije v poročilu so (Fewster, 1999):

- število izbranih testov;
- število testov, ki so bili dejansko izvedeni;
- število pozitivnih testov;

- število negativnih testov;
- število odkritih napak;
- število nedokončanih testov;
- podrobne informacije o negativnih in nedokončanih testih;
- omejitve orodja;
- časovni parametri (čas trajanja posameznega testa).

Če izvajamo avtomatske teste čez noč, želimo naslednje jutro porabiti čim manj časa za pridobitev rezultatov. Čeprav testno orodje kakovostno opravi proces testiranja, je zelo pomemben tudi način prikaza rezultatov. Dobro oblikovani grafični vmesniki osvežujejo podatke že med testiranjem. Tako lahko sproti spremljamo rezultate. Če opazimo veliko nezaželenih odstopanj, lahko test prekinemo in takoj posredujemo rezultate programerjem.

Idealno bi bilo, da bi se ob vsaki napaki v aplikaciji shranila slika okna, kjer se je napaka zgodila, ampak žal nimajo vsa testna orodja te možnosti.

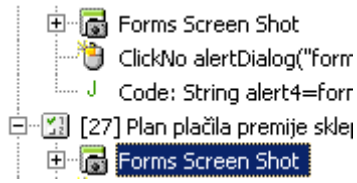
6.3.4 Vizualizacija kode

Veliko testnih orodij ima na voljo dva pregleda testne skripte. To je drevesni pregled in pregled kode. V pregledu kode vidimo kodo posnetega testnega scenarija. To kodo lahko spreminjamo oziroma dopolnjujemo. Edini pogoj je poznavanje programskega jezika, ki ga orodje podpira.

Ko se odločamo za testno orodje, se je smiselno pozanimati, katere programske jezike podpira orodje. Tako ne bomo tvegali, da bi na primer tester imel znanje programskega jezika Java, orodje pa bi podpiralo le programski jezik C++. Spreminjanje kode testnih skript je osnovni pogoj za doseganje ponovne uporabnosti testnih skript.

6.3.5 Dodatne funkcije

Nekatera orodja nudijo možnost pregleda slik (angleško Screenshot) vsake akcije, ki smo jo izvedli v aplikaciji. Tako je razvidno, kaj skripta izvede v vsaki akciji. Slike posameznih akcij nam tudi olajšajo popravljanje testnih skript.



Slika 10: Zaslonska slika

Ko uporabljamo aplikacijo, preberemo ponujene možnosti in razmišljamo o tem, kaj bomo izbrali. Pri avtomatskih testnih skriptah gre za navidezne uporabnike, kjer se odziv zgodi »takoj« (v skladu v scenariju predvidenim dogodkom). Zato je dobro, da imamo možnost nastavljanja časa zakasnitev (torej simulacijo premišljevanja »realnega uporabnika«). Možnost različnih hitrosti izvajanja testa je tudi ena od funkcij, ki jih imajo novejša orodja.

7 ORACLE APPLICATIONTESTING SUITE

Oracle je danes eden od največjih podjetij na področju poslovne programske opreme z več kot 320.000 strankami v več kot 145 državah sveta. Podjetje je na področju testiranja razvilo več avtomatiziranih orodij, ki pokrivajo upravljanje testov, funkcionalno testiranje in obremenitveno testiranje. Poleg tega obstajajo tudi orodja za maskiranje podatkov (Slika 17) in testiranje produkcijskih aplikacij. Orodja za testiranje delujejo v skladu z življenjskim razvojnim ciklom programske opreme ter pokrivajo vse faze od načrta, razvoja, testa in implementacije.

Leta 2009 je Oracle razvil prvo različico programske opreme za testiranje Application Testing Suite (v nadaljevanju OATS). OATS je integrirana in celotna rešitev za testiranje spletnih strani in aplikacij.

OATS vsebuje:

- Oracle OpenScript – javanska aplikacija, ki jo uporabljamo za ustvarjanje avtomatskih testnih skript v Javi.
- Oracle Load Testing – orodje, ki ga uporabljamo za stresno in obremenitveno testiranje programske opreme. Vsebuje tudi strežnik za spremljanje in poročanje med in po končanem izvajanju testa.
- Oracle Test Manager – orodje, ki omogoča organiziranje in upravljanje celotnega procesa testiranja.

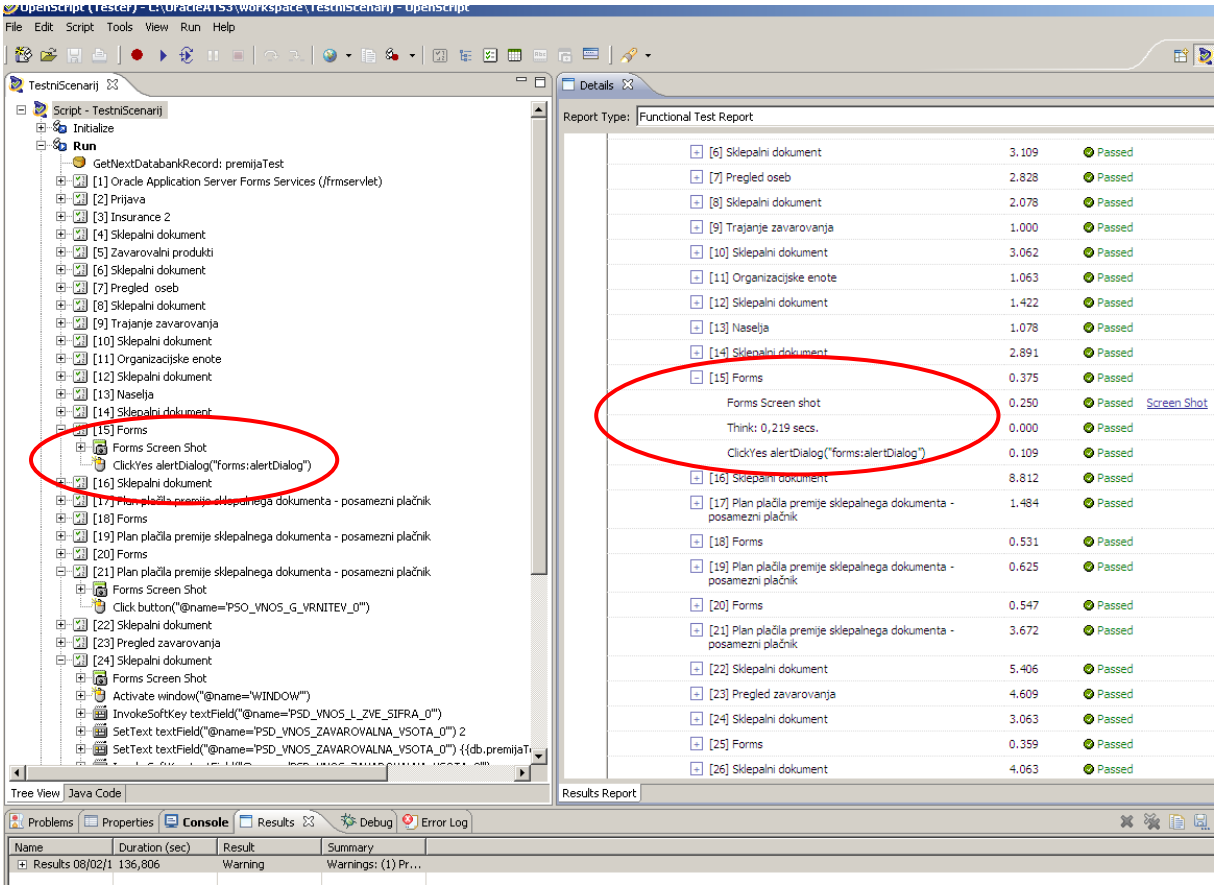
7.1 Oracle OpenScript

Slika 11: Oracle OpenScript

Oracle OpenScript je namizna javanska aplikacija za avtomatsko funkcionalno in obremenitveno testiranje spletnih aplikacij in spletnih strani. Nudi možnost dostopa do podatkov prek JDBC (Java Database Connectivity) ter testiranje aplikacij prek interneta. OpenScript lahko nadgrajujemo za svoje potrebe. Za prilagajanje in nadgradnjo je potrebno znanje programskega jezika Java. Je neodvisen od platforme, kar pomeni, da OpenScript lahko namestimo na različne operacijske sisteme. Podpira tudi HTML (Hyper Text Markup Language) in JavaScript.

Kot druga orodja za funkcionalno testiranje, ima tudi OpenScript funkcijo posnemi in predvajaj. S klikom na gumb posnemi posnamemo sprehod skozi aplikacijo in s klikom na gumb predvajaj posnet sprehod ponovimo. Paradigma posnemi in predvajaj ne zahteva nobenih programerskih izkušenj. Za uporabo bolj naprednih funkcij, ki so na voljo v orodju Oracle OpenScript, pa je potrebno znanje

programskega jezika Java. Za povezovanje z bazami podatkov in njihovo uporabo je potrebno poznavanje jezika SQL (Structured Query Language).



Slika 12: Grafični uporabniški vmesnik

Osnovne komponente orodja Oracle OpenScript

- Delovna površina (angleško WorkBench)

OpenScript vsebuje delovno površino (Slika 12), kjer lahko ustvarjamo in predvajamo avtomatizirane testne skripte. Na levi strani slike (Slika 12) vidimo drevesni pregled posnetega testnega scenarija. Vsaka akcija oziroma korak je označena s številko in z imenom. Prav tako vsaki akciji pripada zaslonska slika. Na desni strani je poročilo testne skripte. Zabeležena je vsaka akcija, ki se je izvedla in koliko časa je trajala. Primer:

Akcija 15 (obkrožena na sliki 12) je izvedla klik na gumb *DA* v opozorilnem oknu. Na desni strani vidimo, da se je ta akcija izvedla uspešno in je trajala 0,219 sekund.

Slika 13: Orodna vrstica

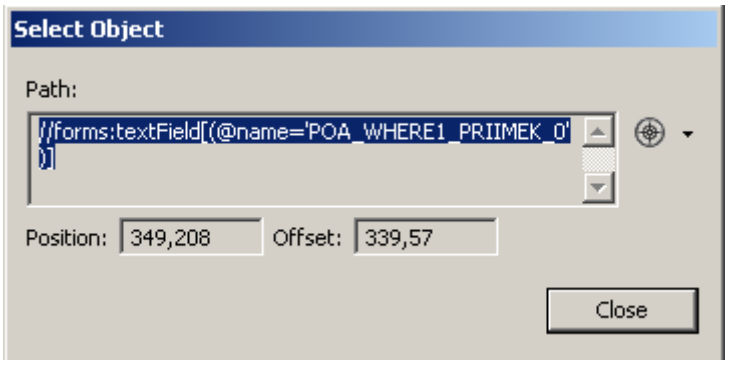
Orodna vrstica vsebuje gumbe posnemi, predvajaj, ponavljaj, pavza, stop, na desni strani pa še gumbe pregled poti (angleško Inspect path), objektni test (angleško Object test – opisan v nadaljevanju) itd.

Pregled poti je funkcija, ki nam za vsako polje v aplikaciji pove, kakšna je njegova pot, oziroma kako naj dostopamo do določenega polja. Ko posnamemo testno skripto, vidimo, da ima vsako polje, gumb itd. svoje ime oziroma svojo pot. Inspect path nam omogoča, da identificiramo objekt, kar nam še posebej koristi pri vstavljanju akcij v testno skripto.

Primer:

V testni scenarij bi želeli vstaviti akcijo, s katero v vnosno polje vpišemo ime in priimek. Kliknemo na gumb Inspect path in se z miško postavimo na obravnavano polje.

V oknu se prikaže opis polja.



Slika 14: Pregled poti

Zdaj lahko v testno skripto vstavimo akcijo, s katero vpišemo v vnosno polje priimek in ime.

```
forms.textField(1328,"//forms:textField[(@name='POA_WHERE1_PRIIMEK_0')]")
      .setText("KODARIN SUZANA");
```

V meniju pogledi lahko po želji izberemo prikaz zavihkov problemi, lastnosti, konzola, rezultati itd. Le-ti so prikazani v spodnjem delu delovne površine (Slika 12).

Skripte lahko urejamo prek drevesnega pregleda ali pregleda kode. Če želimo določene spremembe narediti neposredno v kodi, izberemo zavihek Java Code. S tem preklopimo na pogled v kodi in urejamo kodo v Javi.

- Vgrajeni testni moduli

OpenScript ima vgrajene različne testne module. Vsak modul ima implementirane določene funkcije (pakete), ki nam omogočajo testiranje točno določene vrste aplikacije.

Vgrajeni moduli so Adobe Flex (uporabljamo za testiranje aplikacij, razvitih s tehnologijo Flex), Oracle EBS/Forms (uporabljamo za testiranje aplikacij, razvitih z orodjem Form Builder), Oracle Fusion/ADF (uporabljamo za testiranje aplikacij, razvitih v orodju Oracle Fusion), Siebel (uporabljamo za testiranje aplikacij, nastalih z orodjem Siebel) in modul Web (uporabljamo za testiranje spletnih strani) (Slika 16).

Ker imam izkušnje le s testiranjem aplikacij Form, se bom omejila na opis uporabe modula EBS/Forms. Ostalih omenjenih aplikacij za zdaj še nisem testirala. Poleg že vgrajenih modulov pa za lastne potrebe testiranja lahko razvijemo dodatne module.

Modul izberemo, kadar ustvarimo novo skripto (Slika 16).

- Drevesni pregled

Grafični uporabniški vmesnik omogoča drevesni pregled testne skripte (Slika 18). V drevesnem pregledu lahko dodajamo oziroma brišemo akcije.

- Pregled kode v Javi

Ta pregled (Slika 19) omogoča ustvarjanje, urejanje in razhroščevanje kode testnih scenarijev. Katerokoli spremembo naredimo v tem pogledu, se avtomatsko posodobi tudi v drevesnem pregledu in obratno.

- Pregled rezultata in pregled lastnosti testa

V pregledu rezultata je za vsako izvedeno akcijo razvidno, če se je izvedla uspešno (Passed) ali neuspešno (Failed). Prav tako lahko za vsak korak oziroma akcijo v testu pogledamo lastnosti akcije in sliko akcije.

- Nastavitve OpenScript

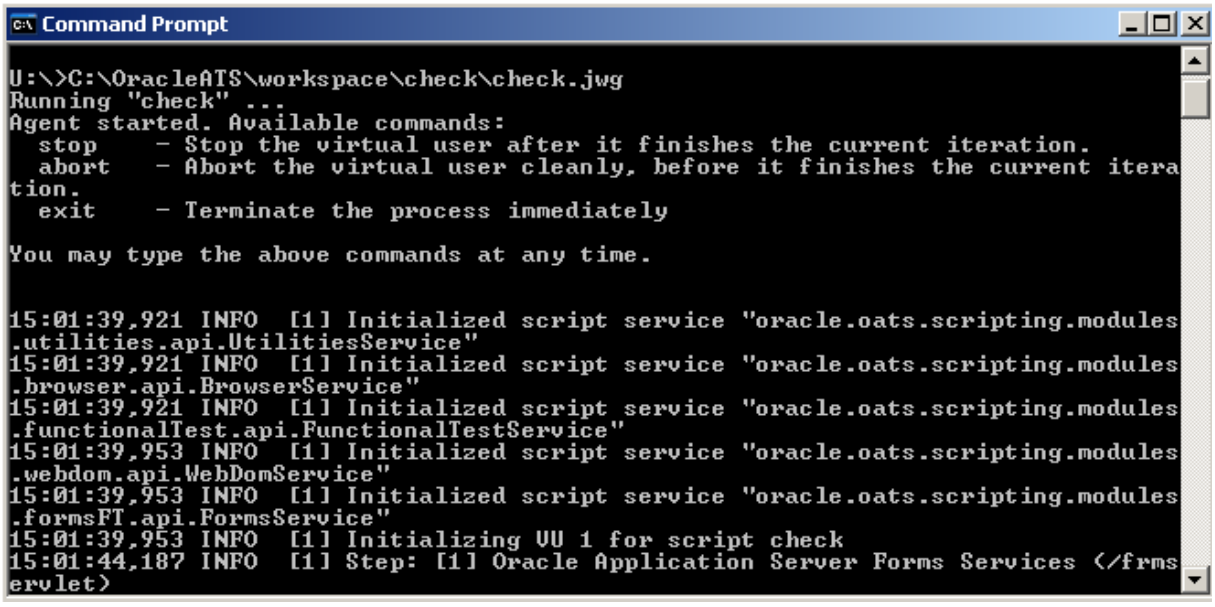
Tukaj določimo nastavitve za nadzor snemanja scenarija, predvajanja scenarija in splošne nastavitve.

7.1.1 Avtomatsko funkcionalno testiranje z orodjem OpenScript

Avtomatsko testiranje programske opreme poteka po metodi črne škatle, kot je opisano v razdelku 3.2. Razlika je v tem, da testiranje oziroma uporabo aplikacije posnamemo (več o snemanju testnih scenarijev je razloženo v nadaljevanju).

OpenScript omogoča izvajanje testnih skript na dva načina. Prvi način je poganjanje testov prek grafičnega uporabniškega vmesnika (angleško GUI mode), ki je najbolj pregleden, najlažji za uporabo in najlažji za analizo. Ta način se najpogosteje uporablja za pripravo testnih scenarijev.

Drugi pa je testiranje prek orodne vrstice (angleško Console mode). Prek orodne vrstice poženemo testno skripto s končnico jwg (**Slika 15**).



Slika 15: Izvajanje testne skripte prek orodne vrstice

Pri obeh načinih testiranja je rezultat poročilo v obliki HTML-ja.

7.1.2 Priprava testnih scenarijev

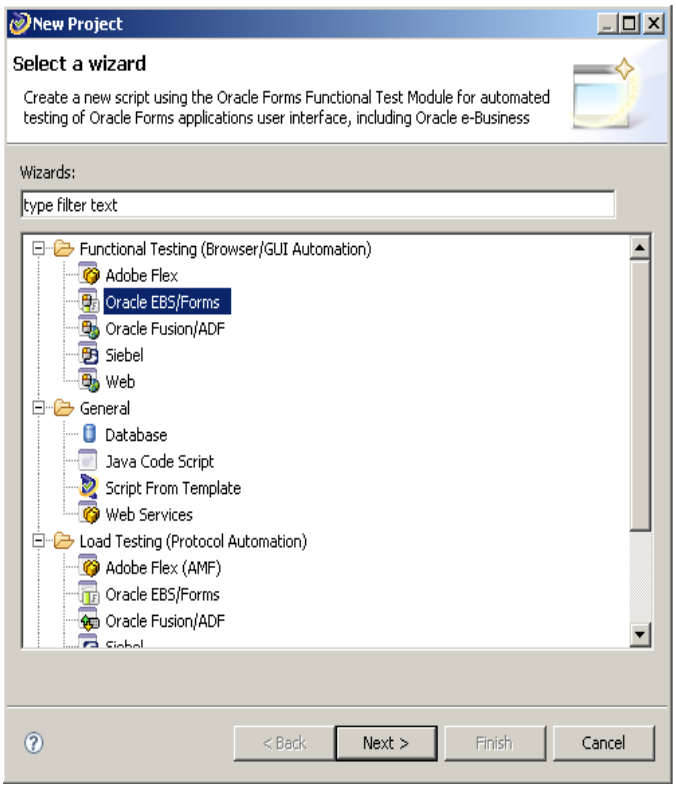
Testni scenarij je simulacija poslovnih dogodkov v poslovni praksi. Če bi za testiranje aplikacije imeli neomejeno količino časa, bi lahko pripravili poljubno veliko različnih testnih scenarijev. Žal je čas omejen in zato je tudi število testnih scenarijev omejeno.

Po navadi testne scenarije priloži naročnik programske opreme skupaj s specifikacijo. Tako za testerje kot tudi za razvijalce programske opreme so testni scenariji zelo pomembni. Testni scenariji morajo biti skrbno pripravljeni. Zajemati morajo čim več primerov, ki se dejansko dogajajo v praksi, saj bodo testerji testirali pravilnost izvajanja aplikacije na podlagi teh scenarijev.

7.1.3 Kreiranje projekta in potek snemanja testne skripte z orodjem OpenScript

Kot običajno v testnih, grafičnih ali katerih drugih orodjih se tudi v orodju OpenScript projekt kreira s klikom na zavihek »File« v levem zgornjem kotu in nato izbiro »New«. Izberemo modul, ki je najbolj primeren za vrsto aplikacije, ki jo testiramo (Slika 16). Za funkcionalno testiranje aplikacije Form uporabljamo modul EBS/Forms.

Po izbiri modula se skripti samodejno kreirajo trije vozli: initialize, run in finalize. Vozli se polnijo med snemanjem testnega scenarija. V vozlu initialize se posname zagon brskalnika in povezava z bazo podatkov. Vozel finalize vsebuje odjavo iz baze. Vse druge akcije se beležijo v vozlu run (Slika 18).



Slika 16: Moduli

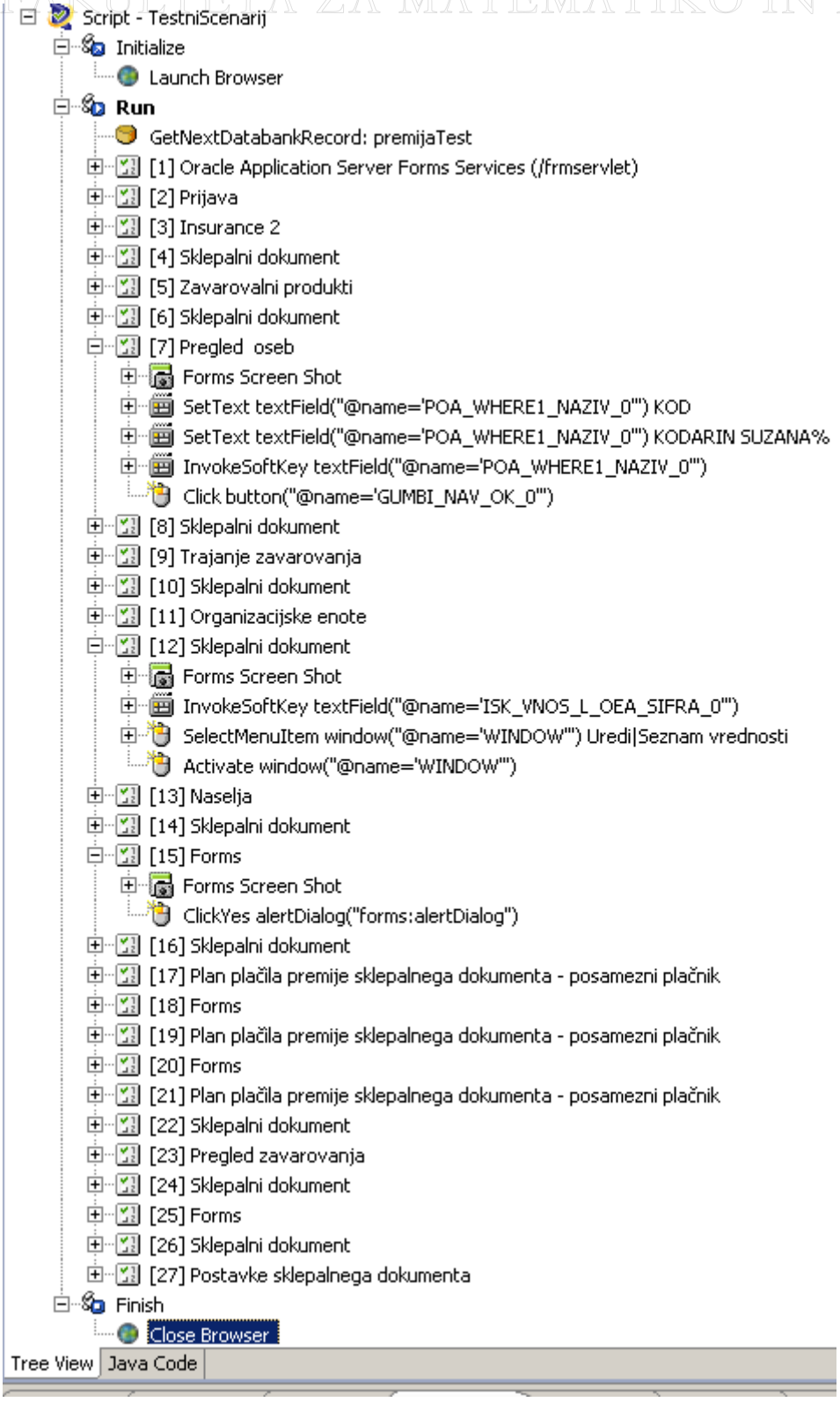
S klikom na gumb posnemi (angleško Record) začnemo snemanje testnega scenarija. Odpre se okno v brskalniku, kamor vpišemo url naslov spletne aplikacije.

Po navadi se je v aplikacijo treba prijaviti. Ker lahko skripto uporablja tudi več uporabnikov, ima OpenScript možnost maskiranja gesla. To pomeni, da tisti, ki bo uporabljal isto testno skripto, ne bo videl posnetega gesla (Slika 17).

Slika 17: Prijava v aplikacijo

OpenScript posname in shrani sliko stanja aplikacije po vsaki naši akciji v aplikaciji (kar vidimo na slikah Slika 12 in Slika 18) ter beleži čas med akcijami.

Snemanje zaključimo s klikom na rdeč gumb stop. Sliki 18 in 19 prikazujeta drevesni pregled in pregled kode testne skripte, ki smo jo posneli.



Slika 18: Drevesni pregled testne skripte

```
import oracle.oats.jagent.cntlproxy.databank.DBBuffer;

public class script extends IteratingVUserScript {
    @ScriptService oracle.oats.scripting.modules.utilities.api.UtilitiesService utilitie
    @ScriptService oracle.oats.scripting.modules.browser.api.BrowserService browser;
    @ScriptService oracle.oats.scripting.modules.functionalTest.api.FunctionalTestServic
    @ScriptService oracle.oats.scripting.modules.webdom.api.WebDomService web;
    @ScriptService oracle.oats.scripting.modules.formsFT.api.FormsService forms;
    private String premija;

    public void initialize() throws Exception {
        browser.launch();
    }

    /**
     * Add code to be executed each iteration for this virtual user.
     */
    public void run() throws Exception {
        getDatabank("premijaTest").getNextDatabankRecord();
        beginStep("[1] Oracle Application Server Forms Services (/frmservlet)",
            0);
        {
            web
            .window(188,
                "/web:window[@index='0' or @title='about:blank']")
                .navigate(
                "http://10.1.2.235/forms/frmservlet?form=zav0030f.fmx&config=TEST");
            web
            .window(189,
                "/web:window[@index='0' or @title='Oracle Application Server Forms Servi
                .waitForPage(null);
        }
        endStep();
        beginStep("[2] Prijava", 0);
        {
            forms.captureScreenshot(191);
            {
                think(15.031);
            }
            forms.logonDialog(192, "//forms:logonDialog")
            .setName("in2_kodarin");
            {
                think(0.0);
```

Slika 19: Pregled kode testne skripte

7.1.4 Predvajanje testne skripte

Predvajanje testne skripte je natančna ponovitev testnega scenarija ali del testnega scenarija, ki smo ga predhodno posneli z orodjem za avtomatsko testiranje programske opreme. Testna skripta se ustvari po končanem snemanju. Med predvajanjem se v oknu sproti prikazujejo vse akcije, ki jih je uporabnik izvedel med snemanjem.

Preden začnemo s predvajanjem testne skripte, lahko nastavimo spremembe delovanja testne skripte. Teh je veliko, zato bom omenila le nekatere.

- Želeni čas predvajanja

Ko posnamemo skripto, nam OpenScript beleži čas med posamezno akcijo. Posneti čas je beležen v sekundah in ga lahko poljubno spreminjamo tako v drevesnem kot v Java pregledu.

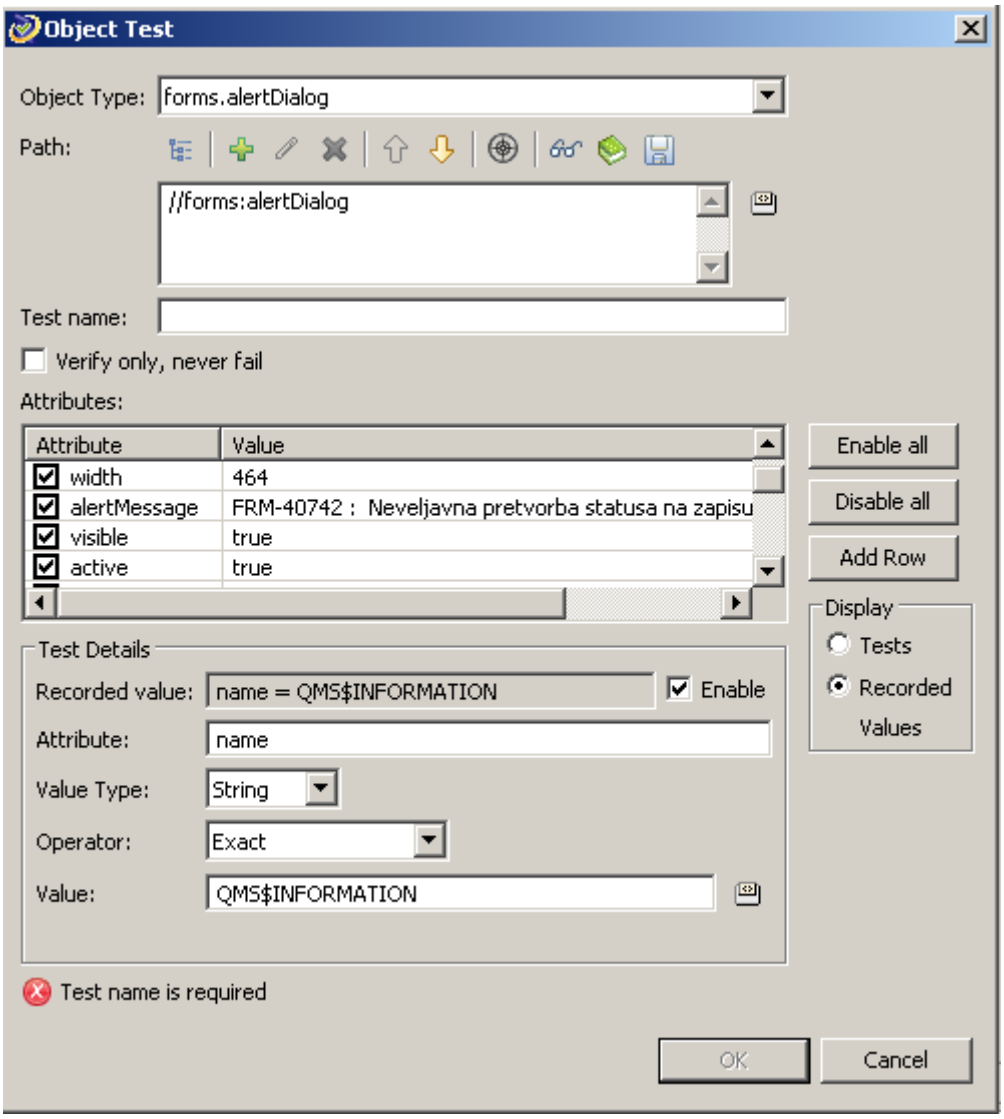
```
        {
            think(8.469);
        }
```

Lahko pa tudi nastavimo čas na »No delay«, kar pomeni, da bo med vsakim korakom oziroma akcijo enak čas ne glede na posneti čas.

- Objektni test (angleško Object test)

Objektni test naredimo, ko preverjamo, ali se določen objekt (gumb, opozorilno okno, vnosno polje itd.) pojavi v aplikaciji na točno določenem mestu. Na primer opozorilno okno z natanko določenimi vrednostmi se mora pojaviti v točno predvidenih okoliščinah. Če se med predvajanjem testa ne pojavi takrat, ko je predvideno, je objektni test neuspešen.

Spodnja slika prikazuje primer objektnega testa. Opozorilno okno s sporočilom *FRM-40742: Neveljavna pretvorba statusa na zapisu ...* mora obstajati točno takrat, ko se izvede del skripte, kjer smo objektni test vstavili. Če opozorilnega okna ne bo, je test neuspešen.



Slika 20: Objektni test

- Obravnavanje napak (angleško Error Recovery) bomo opisali v razdelku 7.1.6.
- Popravljanje in dodajanje akcij v kodo

Posnetemu scenariju lahko dodajamo akcije. Na primer klik na gumb, vnos drugačnega besedila v vnosno polje itd. Spremembe lahko izvajamo v obeh pregledih. Akcije dodajamo, kadar na primer

pozabimo preveriti pravilnost vsebine besedila v določenem polju, ali ko pride do manjše spremembe v aplikaciji, na primer nov gumb v aplikaciji.

- Databank

Open Script omogoča hranjenje neomejene količine podatkov v tekstovnih datotekah in podatkovnih bazah (Databank). Med predvajanjem testnega scenarija se parametri aplikacije napolnijo s podatki iz vnaprej pripravljene podatkovne baze. To pomeni, da lahko testni scenarij predvajamo z različnimi podatki. Ko želimo test večkrat predvajati, kliknemo na gumb Iterate (Slika 31), ki omogoča zaporedje uporabe testnih podatkov iz podatkovne baze.

Testne podatke lahko črpamo zaporedno ali naključno. Če črpamo podatke zaporedno, se bodo posneti podatki napolnili s podatki iz prve vrstice izbrane tabele, v drugi ponovitvi testnega primera pa s podatki v drugi vrstici tabele itd. Če jih črpamo naključno, pomeni, da bo vsaka ponovitev testa naključna vrstica izbrane tabele.

Zavarovalna vsota	PREMIJA
25000	42,50
27000	45,90

Slika 21: Tabela premijaTest

Na sliki (Slika 22) vidimo primer, ko vnos v tekstovno polje dobimo iz podatkovne baze.

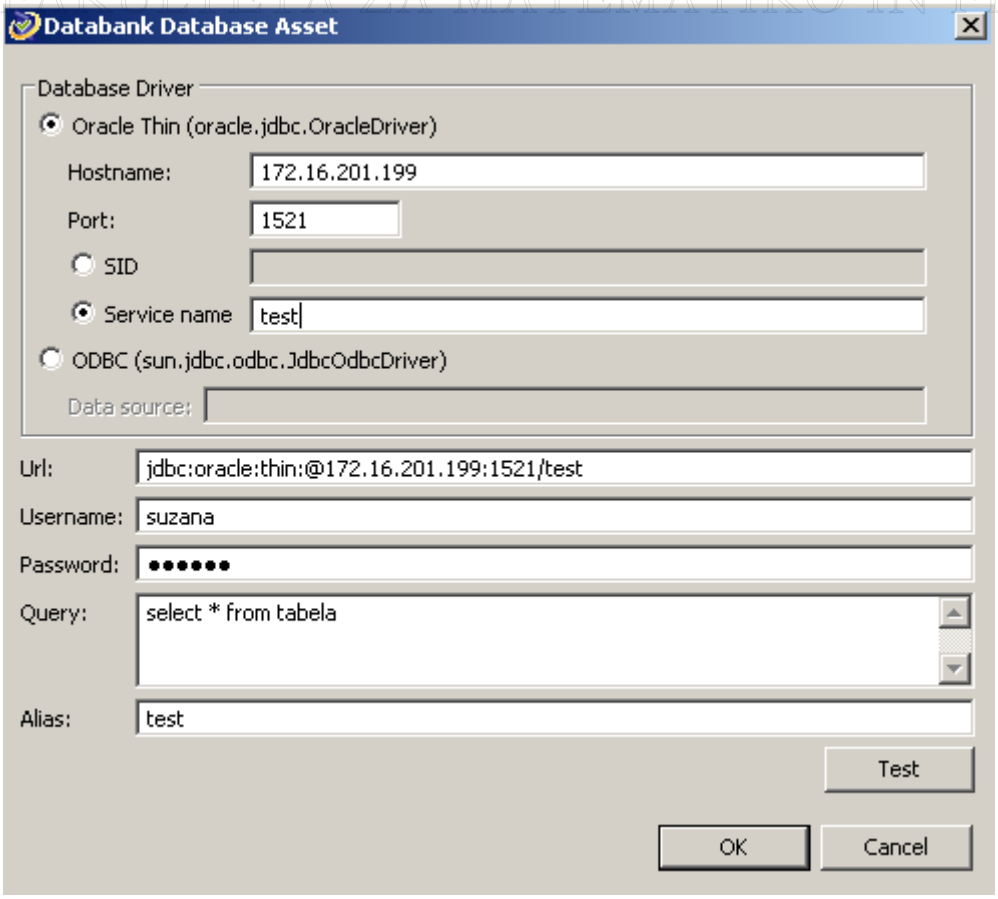
Slika 22: Branje iz podatkovne baze in vnos v tekstovno polje

Koda v Javi je videti tako.

```
forms.textField(306,"//forms:textField[(@name='PSD_VNOS_ZAVAROVALNA_VSOTA_0
')]").setText("{{db.premijaTest.Zavarovalna vsota}}");
```

Obkroženi del pomeni, da bomo ob izvajanju te kode iz tabele *premijaTest* in stolpca *Zavarovalna vsota* dobili tekoči podatek (pri zaporednem izvajanju bo to podatek, zapisan v prvi vrstici izbranega stolpca, pri naključnem izvajanju pa podatek iz naključne vrstice izbranega stolpca), s katerim bomo napolnili polje *Zavarovalna vsota* v aplikaciji (Slika 22).

Če podatke jemljemo iz podatkovne baze, se moramo, preden test predvajamo, z njo povezati (Slika 23). Izpolnimo podatke, ki jih obrazec zahteva. V polju *Query* s poizvedbo oblikujemo tabelo, s katere črpamo podatke. To pomeni, da lahko podatke pri različnih scenarijih različno filtriramo. Pri tem uporabimo jezik SQL.

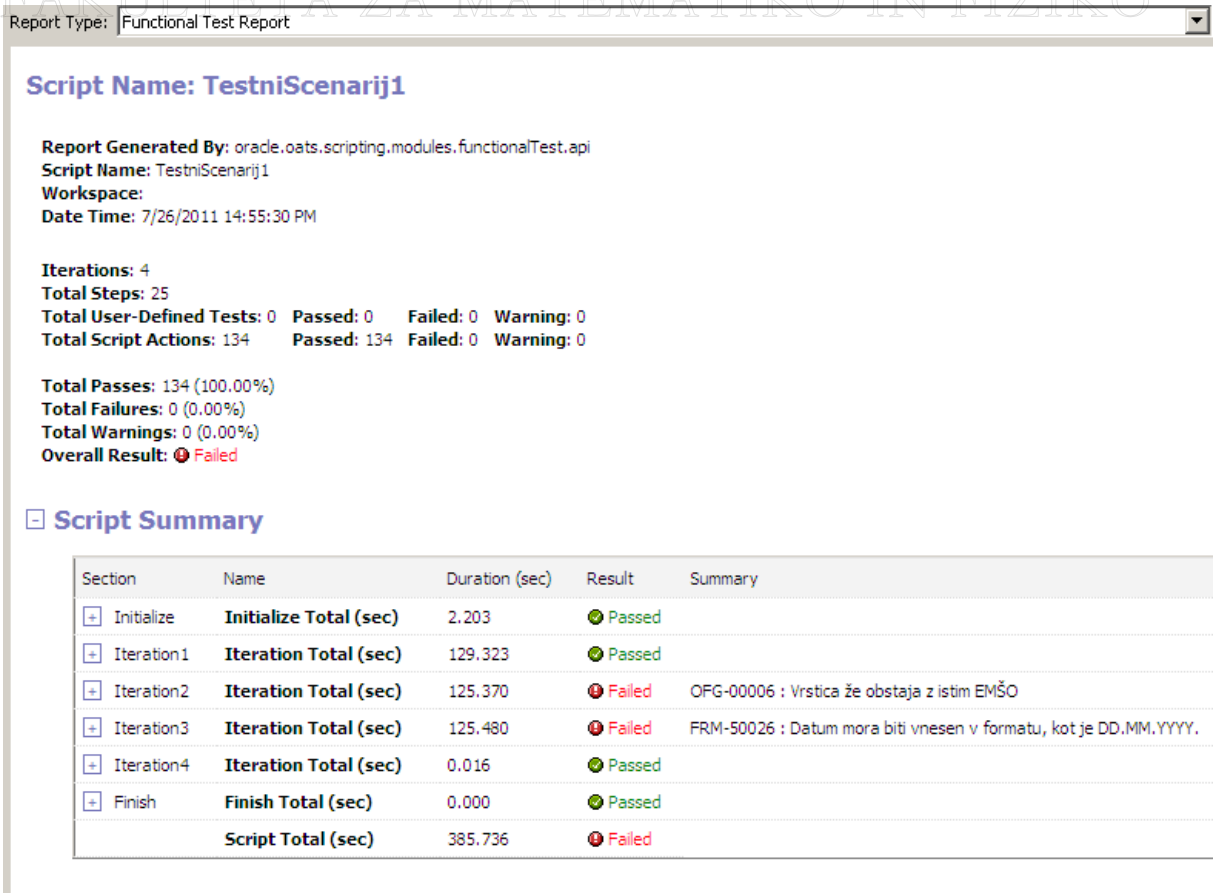


Slika 23: Povezava z bazo podatkov

7.1.5 Poročilo o dobljenih rezultatih

OpenScript prikaže poročilo v obliki spletne strani (Slika 24). Rezultate lahko tudi izvozimo v tekstovno datoteko (v formatu HTML). V poročilu je razvidna vsaka akcija, ki se je zgodila med predvajanjem testne skripte.

Po končanem predvajanju testa ima lahko test status Passed ali Failed. Passed pomeni, da je bil test uspešen, failed pa, da je bil test neuspešen (Slika 24).



Report Type: Functional Test Report

Script Name: TestniScenarij1

Report Generated By: oracle.oats.scripting.modules.functionalTest.api
Script Name: TestniScenarij1
Workspace:
Date Time: 7/26/2011 14:55:30 PM

Iterations: 4
Total Steps: 25
Total User-Defined Tests: 0 **Passed:** 0 **Failed:** 0 **Warning:** 0
Total Script Actions: 134 **Passed:** 134 **Failed:** 0 **Warning:** 0

Total Passes: 134 (100.00%)
Total Failures: 0 (0.00%)
Total Warnings: 0 (0.00%)
Overall Result: Failed

Script Summary

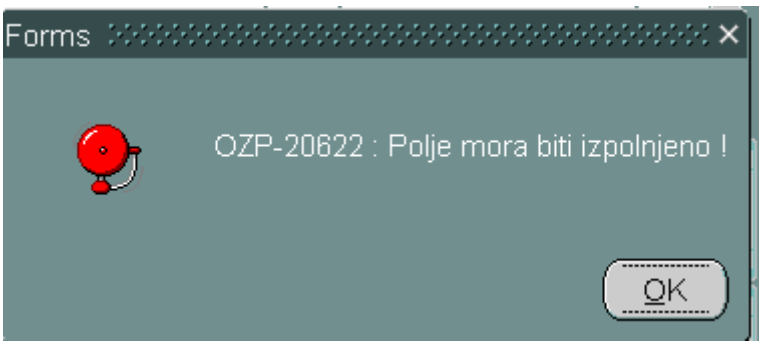
Section	Name	Duration (sec)	Result	Summary
Initialize	**Initialize Total (sec)**	2.203	Passed	
Iteration1	**Iteration Total (sec)**	129.323	Passed	
Iteration2	**Iteration Total (sec)**	125.370	Failed	OFG-00006 : Vrstica že obstaja z istim EMŠO
Iteration3	**Iteration Total (sec)**	125.480	Failed	FRM-50026 : Datum mora biti vnesen v formatu, kot je DD.MM.YYYY.
Iteration4	**Iteration Total (sec)**	0.016	Passed	
Finish	**Finish Total (sec)**	0.000	Passed	
	Script Total (sec)	385.736	Failed	

Slika 24: Primer poročila

7.1.6 Obravnavanje, odpravljanje napak in vrste napak

Ena od prednosti avtomatskega testiranja je enostavnost, s katero preverimo, ali smo napake odpravili. Potem ko smo napako predvidoma odpravili, ponovimo tisti test, ki je odkril napako. Na takšen način se preveri ustreznost popravka.

Zelo pomembno je, kako se obnaša testno orodje v primeru napake v aplikaciji. Po navadi, ko se v aplikaciji pojavi napaka, tej sledi sporočilo. Primer:



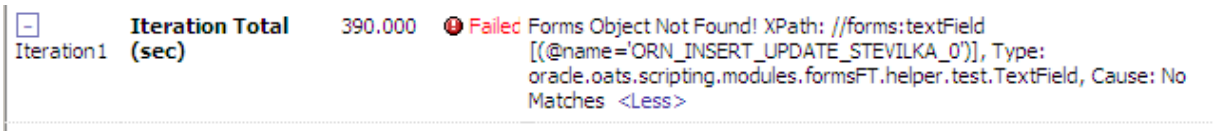
Slika 25: Primer napake v aplikaciji

Sporočilo je lahko informativne narave, vprašalne, opozorilne in sporočilo o napaki. Nekatera sporočila so predvidena s samo aplikacijo in jih ob pripravi scenarija tudi posnamemo. Napaka je, ko se pojavijo sporočila, ki niso posneta. Takrat je test neuspešen, saj v testni skripti ne najde objekta, ki bi se ujemal z resničnim dogajanjem v aplikaciji. V takih primerih želimo, da skripta javi, da se je test izvedel neuspešno. Prav tako bi takrat radi vedeli tudi točen vzrok neuspešnega izida. Tega bomo

verjetno lažje ugotovili, če bomo dobili sporočilo v opozorilnem oknu. Ker ga OpenScript samodejno ne prebere, moramo to dodati kodi testne skripte. Eden od načinov je:

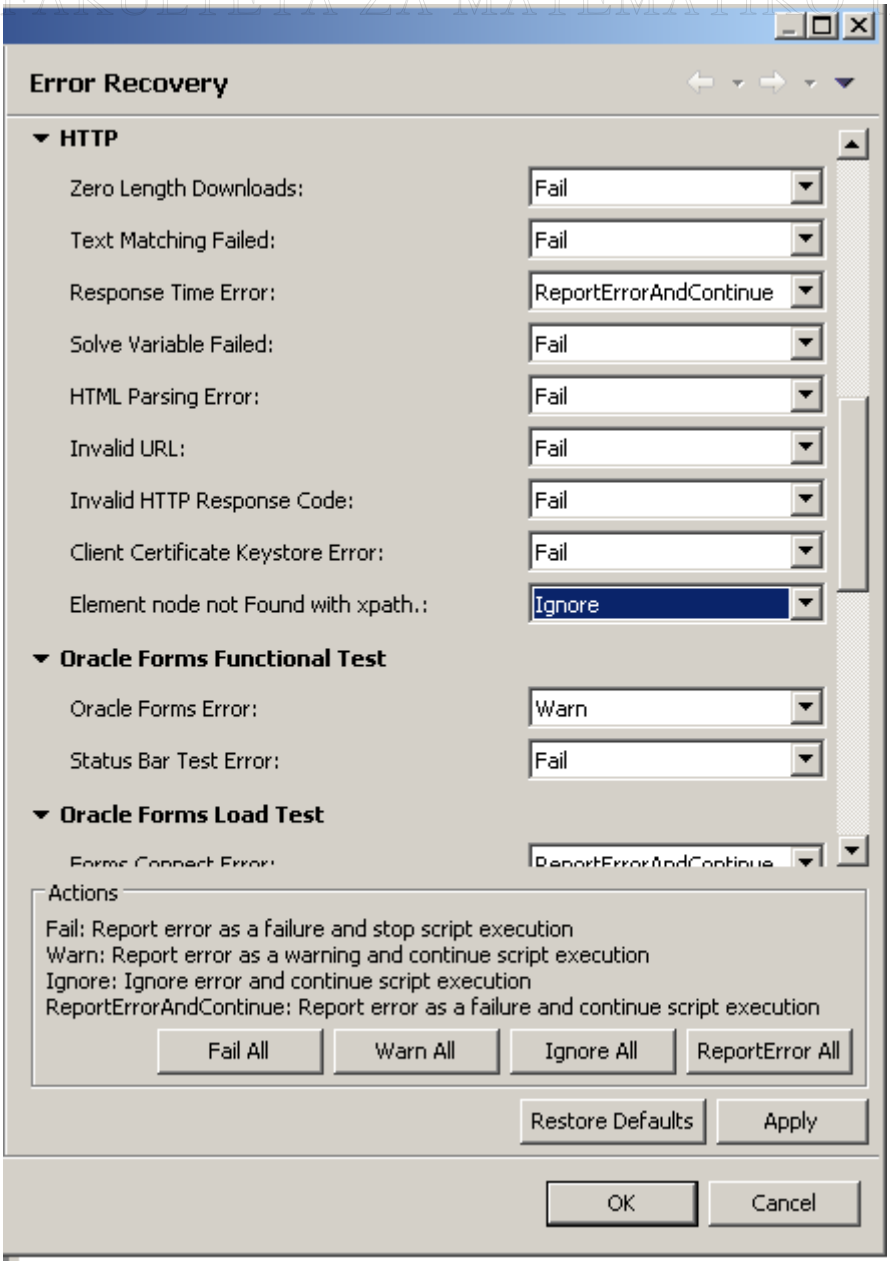
```
/*če se pojavi opozorilno okno*/
if(forms.alertDialog("//forms:alertDialog").exists()){
/*preberi sporočilo v opozorilnem oknu, zaključi test s statusom failed, in
izpiši vsebino sporočila*/
  fail(forms.alertDialog("//forms:alertDialog").getAlertMessage());
}
```

Testni scenarij se lahko tudi ustavi oziroma se izvede neuspešno, ker ne najde posnetega objekta. To pomeni, da je prišlo do spremembe v aplikaciji in OpenScript ne najde več na primer gumba. V tem primeru bomo v poročilu videli sporočilo *object not found*.



Slika 26: Primer napake

Aplikacija nas seveda na manjkajoči objekt ne opozori, ampak nas na to opozori OpenScript, kjer lahko nastavimo nastavitve za tovrstne napake (Slika 27).



Slika 27: Obravnavanje napak

Kot je razvidno iz zgornje slike, sami povemo, kaj naj se zgodi, če pride do napake. Test lahko napako ignorira, lahko nas samo opozori, da se je zgodila, lahko nas opozori in nadaljuje s testom, lahko pa se test takoj ustavi. To velja samo za zgoraj navedene napake.

Za uspešno obravnavanje napak moramo čim bolj poznati aplikacijo in testno orodje OpenScript. To pomeni, da moramo vedeti, katera sporočila in kje se lahko pojavijo.

7.1.7 Primer 1

Poglejmo si primer, kako bi z uporabo orodja OpenScript preverili delovanje dela določene zavarovalniške aplikacije.

Suzana Kodarin želi skleniti nezgodno zavarovanje za eno leto. Plačevala bo mesečno premijo prek položnice. Zavarovalna vsota je 27.000. Preverjali bomo, ali se bo polje *premija* pravilno zapolnilo. Prvi primer se bo izvedel z zavarovalno vsoto 27.000, drugi pa z zavarovalno vsoto 25.000.

Najprej potrebujemo testno skripto. To pomeni, da bomo posneli testni primer za sklenitev nezgodnega zavarovanja. Podatki, ki jih bomo vnašali med snemanjem, niso pomembni, saj jih bomo po končanem snemanju zamenjali s podatki iz podatkovne baze. Ker bomo naredili dva primera, bomo podatke o vrednosti zavarovalne vsote in premije hranili v bazi podatkov.

Zavarovalna vsota	PREMIJA
27000	45,9
25000	42,5

Slika 28: Tabela s podatki

Premijo izračunamo po naslednjem obrazcu:

*premija = zavarovalna vsota * premijska stopnja*

*25.000*0.0017=42.50*

*27.000*0,0017=45,9*

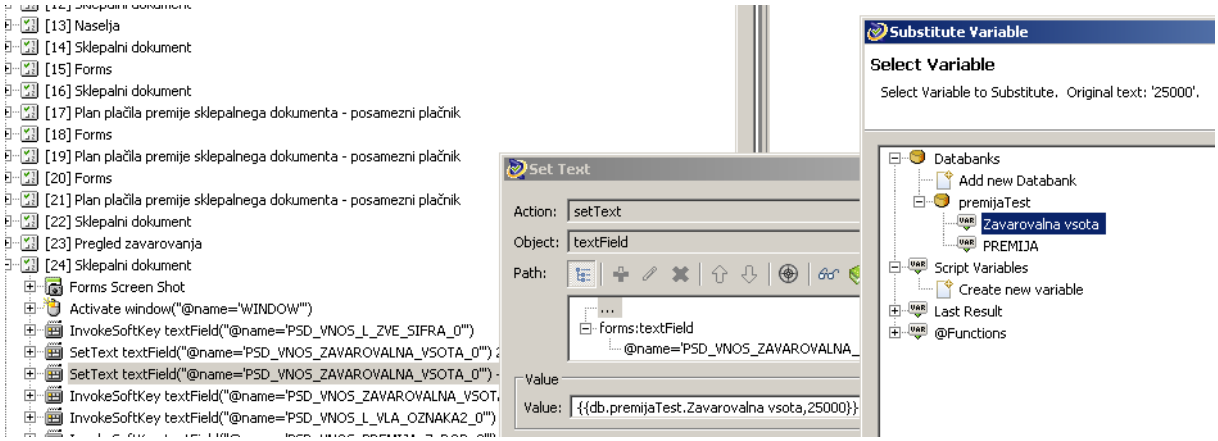
Polje *premija* se mora napolniti z vrednostjo, izračunano po zgornjem obrazcu (obrazec za izračun premije).

Zavarovalna vsota	Premijska stopnja	Premija z dodatki
25.000,00	0,0017000000	42,50

Slika 29: Izračun premije

Pravilnost vrednosti premije, ki se bo zapolnila v polju *Premija z dodatki*, bomo primerjali z vnaprej izračunano premijo, ki bo zabeležena v bazi podatkov (Slika 28). Če bosta vrednosti enaki, bo status testa Passed, sicer pa Failed.

Preden začnemo s predvajanjem in urejanjem testnega scenarija, se moramo z bazo povezati (Slika 23). Nato posneti podatek (zavarovalno vsoto) zamenjamo s podatki v bazi. To naredimo tako, da se postavimo na podatek, ki ga bomo zamenjali, dvakrat kliknemo nanj in vpišemo bazo in ime stolpca, kjer so možne vrednosti, ki naj podatek zamenjajo. Spodnja slika prikazuje, kako zamenjamo podatek s podatkom iz stolpca *Zavarovalna vsota*, ki se nahaja v tabeli *PremijaTest*.



Slika 30: Zamenjava podatka

V pregledu kode pred zamenjavo podatka *zavarovalna vsota* vidimo:

```
forms.textField(306,"//forms:textField[(@name='PSD_VNOS_ZAVAROVALNA_VSOTA_0
')]").setText("25000");
```

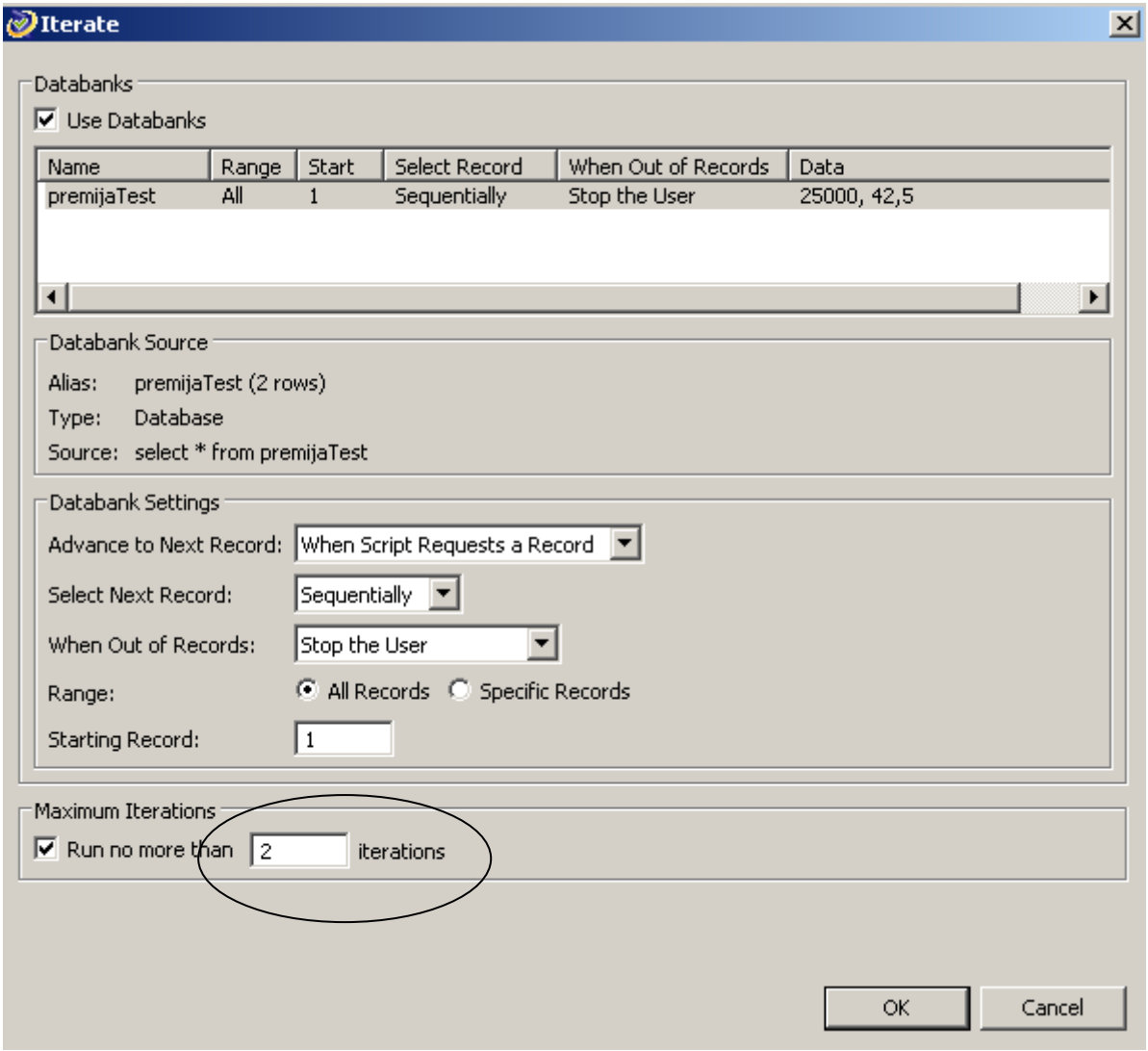
Po zamenjavi pa skriptna koda postane:

```
forms.textField(306,"//forms:textField[(@name='PSD_VNOS_ZAVAROVALNA_VSOTA_0
')]").setText("{{db.premijaTest.Zavarovalna vsota,25000}}");
```

Podatek, ki se bo pojavil v polju *Premija z dodatki*, primerjamo s podatkom v tabeli. Eden od načinov je:

```
String premija=forms.textField(309,
      "//forms:textField[(@name='PSD_VNOS_PREMIJA_Z_DOD_0')]").getText()
//premijo primerjamo s premijo v tabeli
      if(premija.equals(eval("{{db.premijaTest.premija}}"))){
            info("Premija je pravilna!");
      }
      else {
            warn("Premija ni pravilna!");
      }
```

Zdaj lahko predvajamo testno skripto. Kliknemo na gumb Iterate. Ker sta v bazi v tabeli dve vrstici, nam samodejno ponudi dve ponovitvi.



Slika 31: Število ponovitev

Na primer, če vrednost prve premije ni bila enaka vrednosti premije v tabeli, druga pa je bila enaka, je poročilo testne skripte naslednje:

Script Summary

Section	Name	Duration (sec)	Result	Summary
Initialize	**Initialize Total (sec)**	6.469	Passed	
Iteration1	**Iteration Total (sec)**	130.103	Warning	Warnings: (1) Premija ni pravilna!
Iteration2	**Iteration Total (sec)**	0.063	Passed	
	Script Total (sec)	136.806	Warning	

Slika 32: Poročilo testne skripte primera1

7.1.8 Primer2

V aplikacijo bomo vnesli štiri osebe z njihovimi osebnimi podatki (ime, priimek, naslov ...). Podatke bomo hranili v tabeli *Podatki*. Obvezni podatki za vnos fizične osebe v aplikacijo so: ime, priimek, datum rojstva, EMŠO in naslov.

S testnim scenarijem bomo preverili:

– ali so vsi obvezni podatki izpolnjeni;
– ali je datum rojstva vnešen v obliki, ki jo zahteva aplikacija (to je 27. 08. 1983, ne pa 27/8/1983);
– ali je EMŠO, ki ga vnašamo v aplikacijo, unikaten;
– ali se po kliku na gumb zaključi vrednost polja *zaključen ne* (glej sliko Slika 33)

Slika 33: Zaključena

spremeni v *da* (glej sliko Slika 34).

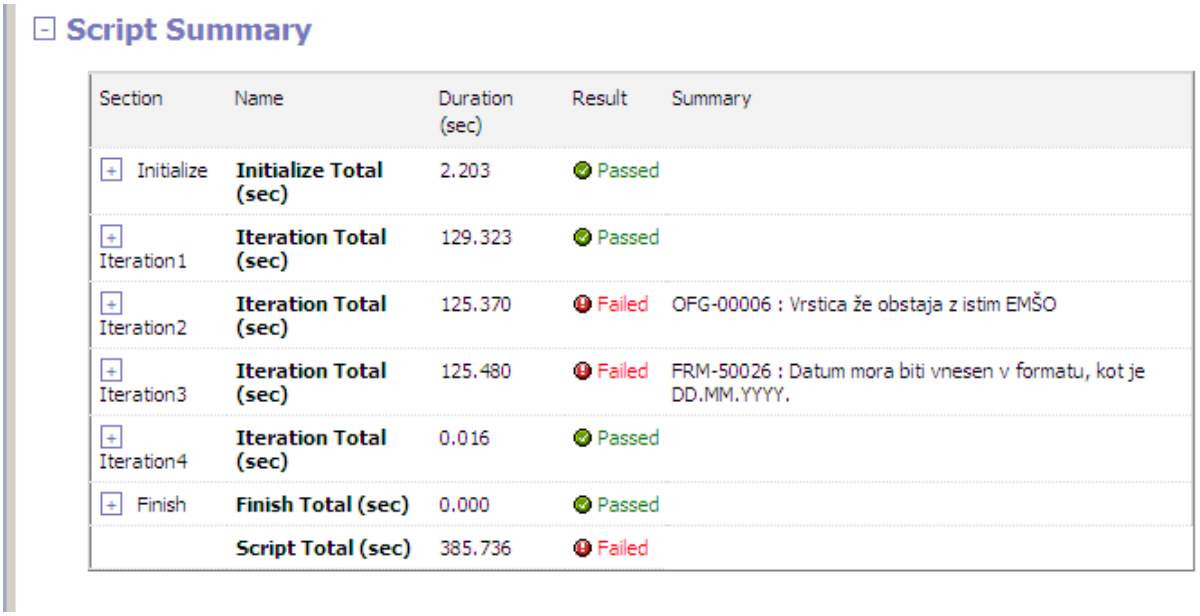
Slika 34: Nezaključena

Najprej moramo posneti testni scenarij, ki vnese fizično osebo v aplikacijo. Ko je skripta posneta, se povežemo z bazo, saj potrebujemo dostop do tabele *Podatki* (glej Slika 23). V bazo bomo vstavili podatke o štirih osebah. Prvo osebo vnesemo brez napak. Druga oseba naj ima EMŠO, ki je že uporabljen v aplikaciji. Aplikacija mora izpisati ustrezno opozorilo. Tretjo osebo vnesemo z napačnim formatom datuma, zato se mora pojaviti ustrezno opozorilo. Četrto osebo vnesemo brez napak. Pri vsakem testu bomo tudi preverjali, ali je oseba v aplikaciji zaključena. To bomo naredili tako, da bomo na koncu testa dodali:

```
if(forms.list(147,"//forms:list[(@name='OSA_INSERT_ZAKLJUCEN_0')]")
.getItemValue().equals("NE")){
      warn("Oseba ni zaključena!");
      }
else {
      info("Ok, oseba je zaključena.");
      }
```

Hkrati preverjamo tudi, ali so vsi obvezni podatki izpolnjeni. Če podatki niso izpolnjeni, se bo izpisalo ustrezno opozorilo.

Na sliki vidimo, kakšno je bilo poročilo testa pri enem od testiranj.



Script Summary

Section	Name	Duration (sec)	Result	Summary
Initialize	**Initialize Total (sec)**	2.203	Passed	
Iteration1	**Iteration Total (sec)**	129.323	Passed	
Iteration2	**Iteration Total (sec)**	125.370	Failed	OFG-00006 : Vrstica že obstaja z istim EMŠO
Iteration3	**Iteration Total (sec)**	125.480	Failed	FRM-50026 : Datum mora biti vnesen v formatu, kot je DD.MM.YYYY.
Iteration4	**Iteration Total (sec)**	0.016	Passed	
Finish	**Finish Total (sec)**	0.000	Passed	
	Script Total (sec)	385.736	Failed	

Slika 35: Poročilo testne skripte primera2

Iz poročila je razvidno, da sta se dve iteraciji izvedli uspešno, dve pa neuspešno. Zabeležen je tudi vzrok neuspešnih testov. Ne glede na statuse testov so se vsi testi izvedli uspešno, saj smo takšne statuse testov predvideli oziroma pričakovali.

7.1.9 Življenjska doba avtomatskega testa

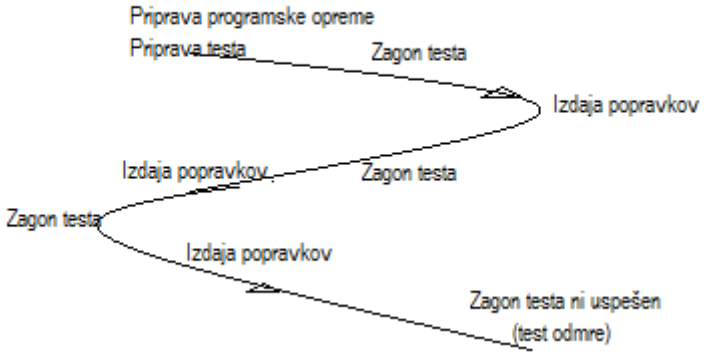
Kot smo že omenili, je zelo pomemben »rok trajanja« testnih scenarijev, s katerim mislimo na to, koliko časa lahko uporabljamo posnete testne scenarije. Ob spremembi programske opreme moramo testne scenarije praviloma spremeniti oziroma na novo posneti, kar ni zmeraj nujno. Če smo v programski opremi opravili le manjšo dodelavo, lahko v kodi samo dodamo akcijo. Na primer želimo dodati klik na gumb. To lahko naredimo tako, da najprej preklopimo v pregled na nivoju kode. V kodi se postavimo v tisti korak, kjer želimo, da se klik izvede. Pri vstavljanju akcije v kodo moramo biti zelo natančni. Vstaviti jo moramo točno na tisto mesto, kjer naj se akcija izvede.

Tako s

forms.button("//forms:button[(@name='GUMBI_NAV_OK_0')]").click;

v testno skripto vstavimo klik na gumb.

Po določenem času se lahko programerji odločijo za večjo dodelavo v programski opremi. V tem primeru ne moremo predvajati enakega testa, saj test ni prilagojen dodelani programski opremi. Življenjska doba testa se je iztekla. Test popravimo ali izumre.



Slika 36: Življenjska doba avtomatskega testa [Vir slike: (Marrick, 1997)]

Preden se odločimo za avtomatizacijo posameznega testnega scenarija, je smiselno oceniti življenjsko dobo testa. Nesmiselno bi bilo posneti testni scenarij, za katerega vemo, da nam že naslednji dan ne bo koristil.

7.2 Oracle Load Testing (OLT)

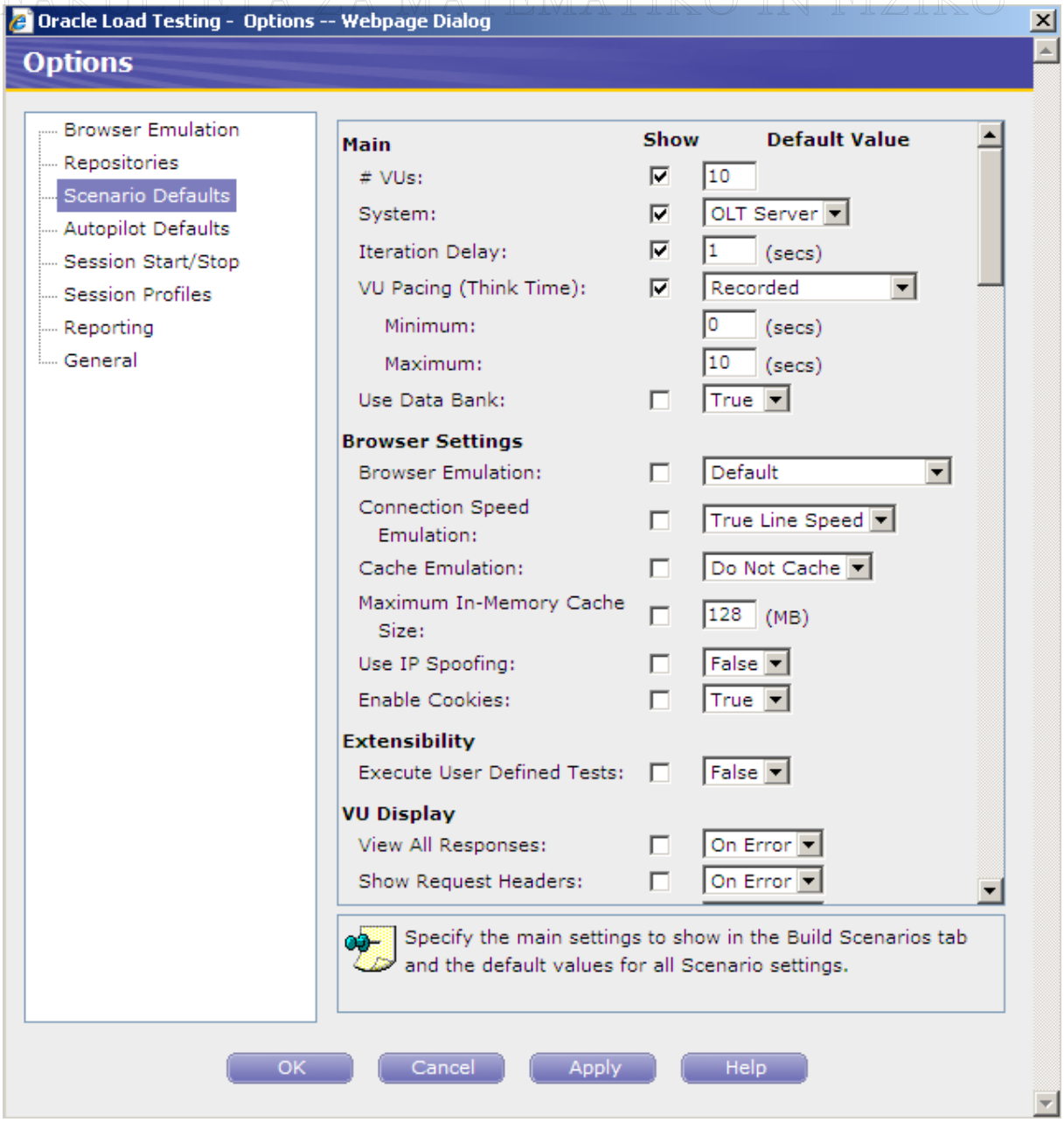
Po uspešni izvedbi funkcionalnega testiranja pogosto izvedemo tudi obremenitveno testiranje, s katerim merimo zmogljivost aplikacije. To pomeni, da preverjamo, kako se aplikacija obnaša pod večjim obsegom obremenitev. Tako je povečan obseg obremenitev lahko povečano število transakcij, povečano število uporabnikov, ki istočasno izvajajo enak tok dogodkov itd.

Orodje OLT omogoča dokaj enostavno in natančno izvedbo obremenitvenega testiranja spletnih strani in aplikacij. Lahko oponaša na tisoče virtualnih uporabnikov, ki hkrati vstopajo v spletno aplikacijo.

OLT ne zahteva programiranja. Testne skripte posnamemo v OpenScriptu in jih kasneje predvajamo v OLT (Slika 37). Ko ustvarimo novo skripto v OpenScriptu, izberemo Oracle EBS/Forms modul pod obremenitvenim testiranjem (angleško Load testing). Snemanje testne skripte poteka enako kot pri funkcionalnem testiranju.

Nastavitev za obremenitveno testiranje je veliko (Slika 37), zato bom omenila le nekatere:

- *VUs* – tukaj določimo število virtualnih uporabnikov,
- *VU Pacing* – nastavimo čas zakasnitve predvajanj med vsakim uporabnikom,
- *Error Recovery* – kako naj se obravnavajo napake,
- *Maximum Users per Process* – največje število virtualnih uporabnikov na sam proces (ponavadi je določeno neomejeno število uporabnikov).



Slika 37: Nastavitve OLT

Spodnji primer prikazuje poročilo obremenitvenega testiranja, kjer je vsako akcijo izvedlo pet uporabnikov, testni primer pa se je izvedel trikrat.

Session Performance Report - rezultat

Report Start Time:	8/23/2011 11:06:38	Session Start Time:	8/23/2011 11:06:38
Report End Time:	8/23/2011 11:12:11	Session End Time:	8/23/2011 11:12:11
Report Duration:	00:05:34 (334 sec)	Session Duration:	00:05:34 (334 sec)

Name	Min	Max	Avg
Active Virtual Users	0	5	4,682
Virtual Users with Errors	0	5	4,611
Transactions Per Second	0,063	3,007	1,435
Pages Per Second	0,267	6	3,242
Hits Per Second	1,4	23,867	12,688
Kilobytes Per Second	3,888	73,141	38,816

Totals	
Transactions	452
Transactions with Errors	447
Pages	924
Hits	3616
Kilobytes	11062

Performance by Profile and Timer *

Name	Min	Max	Avg	Pass	Fail	Std Dev	90th %
loadTest	0,682	25,274	4,919	5	447	6,638	13,223
Initialize: loadTest	0,009	0,009	0,009	5	0	0	0,009
Run: loadTest	49,643	49,643	49,643	5	0	0	49,643
[1] Oracle Application Server Forms Services	0,341	12,984	3	452	0	4,021	12,643
[2] WINDOW	0,335	3,025	0,72	452	0	0,699	1,822
[3] WINDOW_1	0,93	0,984	0,957	5	0	0,039	0,984
[4] WINDOW_1	0,344	0,677	0,51	5	0	0,236	0,677
[5] WINDOW_1	40,465	40,465	40,465	5	0	0	40,465
[6] WINDOW_1	0,044	0,044	0,044	5	0	0	0,044
Finish: loadTest	0	0	0	5	0	0	0

* **Profile timers include think times**

Oracle Load Testing Scenario Report

Profile Name: loadTest

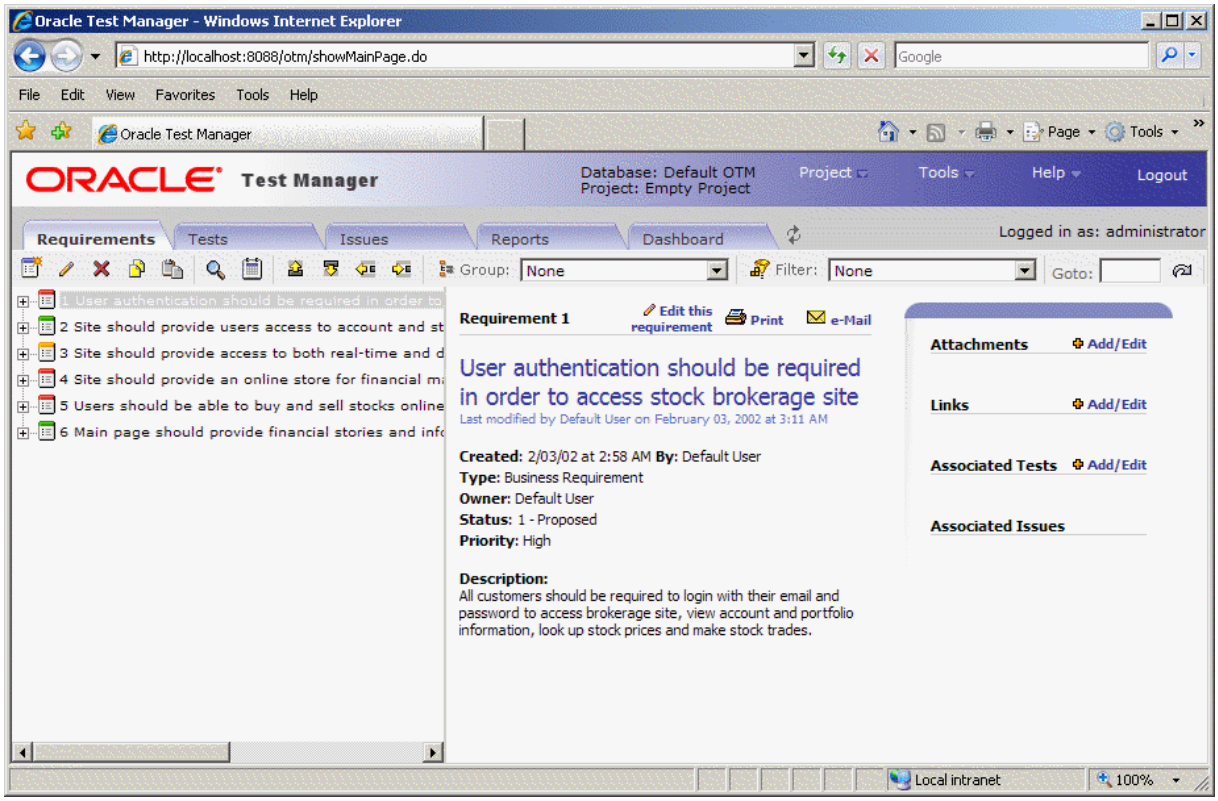
Main

Slika 38: Poročilo obremenitvenega testiranja

V prvem delu vidimo osnovne informacije, ki so, kdaj se je test izvedel in njegove nastavitve. Zabeleženo je število uspešnih in neuspešnih transakcij, ki jih je OpenScript izvedel. V spodnjem delu poročila je zabeležen minimalni, maksimalni in povprečni čas med iteracijami.

7.3 Oracle Test Manager (OTM)

Oracle Test Manager (v nadaljevanju tudi OTM) omogoča organizacijo in upravljanje celotnega postopka testiranja. Zagotavlja poenoteno platformo za izmenjavo informacij med testerji. V OTM kreiramo projekte, v katerih izvajamo testne skripte. Zaradi preglednosti lahko hitreje najdemo testni primer in vse informacije, povezane z njim.

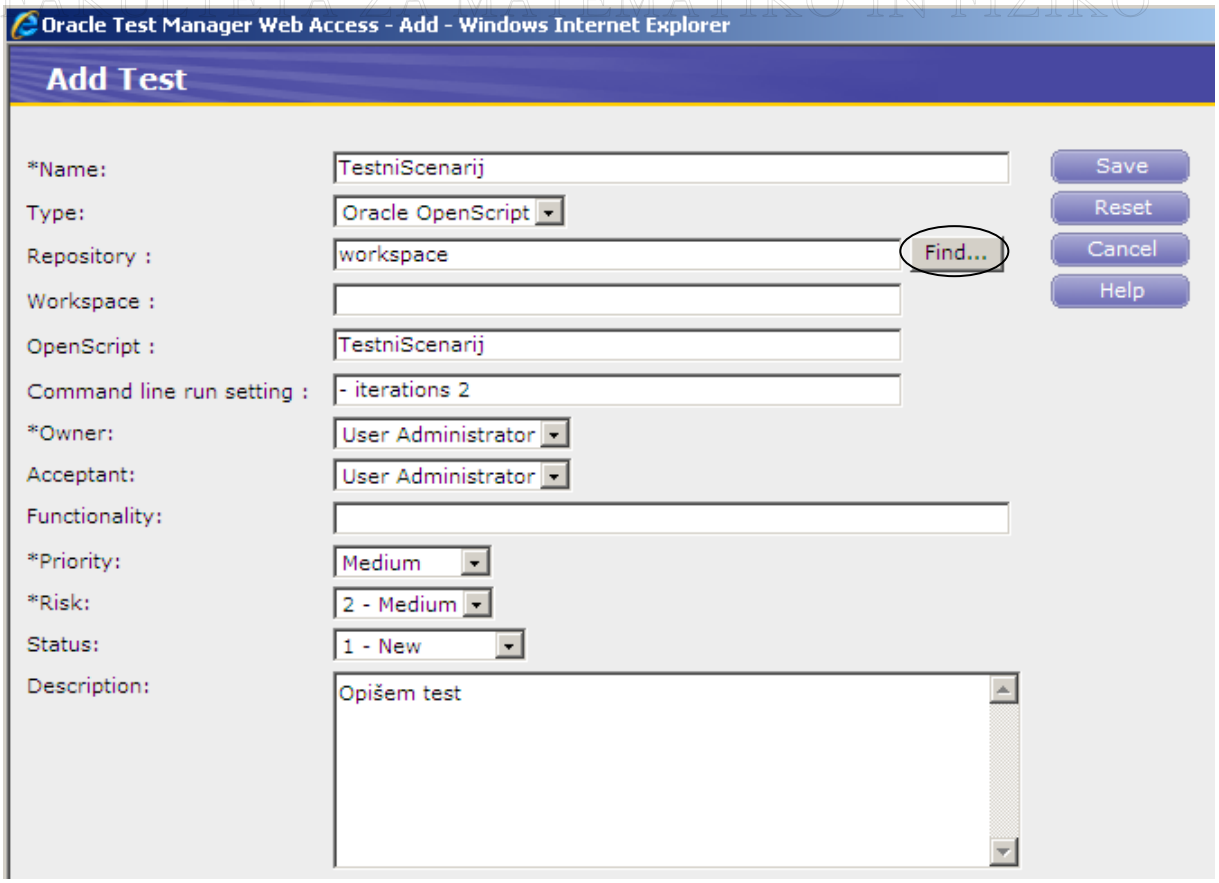


Slika 39: Oracle Test Manager

OTM omogoča naslednje:

- upravlja s testnim načrtom, ki vključuje ročno in avtomatsko testiranje;
- upravlja zahteve in beleži probleme za določen projekt;
- avtomatično sproži in izvrši skripte Oracle OpenScripta za funkcionalno in obremenitveno testiranje;
- pridobi in arhivira rezultate testnih skript;
- zloži testne skripte, rezultate testiranja, priloge, zahteve in probleme v skupno bazo;
- ustvari poročila za vodenje celotnega procesa testiranja.

Kot je prikazano na zgornji sliki (Slika 39), Test Manager vsebuje pet zavihkov: zahteve, teste, probleme, poročila in nadzorno ploščo. Za testerja sta najbolj pomembna zavihka Testi (Tests) in Poročila (Reports). V zavihek testi vnašamo testne primere, ki smo jih posneli v OpenScriptu (Slika 40).



Slika 40: Vnos testne skripte iz OpenScripta

Testno skripto vnesemo s klikom na gumb *Find*. Izpolnimo ime testa (angleško Name), izberemo lastnika testa (angleško Owner) in prioriteto testa (angleško Priority). V polju *Command line run settings* definiramo lastnosti testa. To so nastavitve predvajanja, čas izvajanja, število iteracij itd.

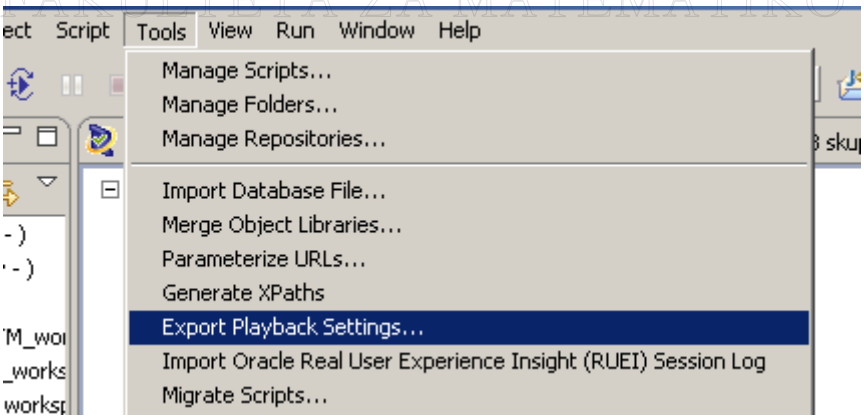
Na primer, naj se izvede pet ponovitev testne skripte:

– *iterations 5*

Naj se izvede pet ponovitev in naj bo med vsako iteracijo (izvedbo skripte) dve sekundi premora:

– *iterations 5 – iterationDelay 2*

Lahko pa tudi določimo, da naj bodo nastavitve predvajanja enake, kot so določene v OpenScriptu. Da lahko to storimo, moramo nastavitve najprej izvoziti iz OpenScripta (Slika 41). Testiranje se bo izvedlo z nastavitvami, določenimi v OpenScriptu. Posebej pa je treba v OTM določiti število ponovitev (na primer – *iterations 5*) testnega primera. Sicer se bo izvedla samo ena ponovitev s prvim podatkom, zapisanim v tabeli.



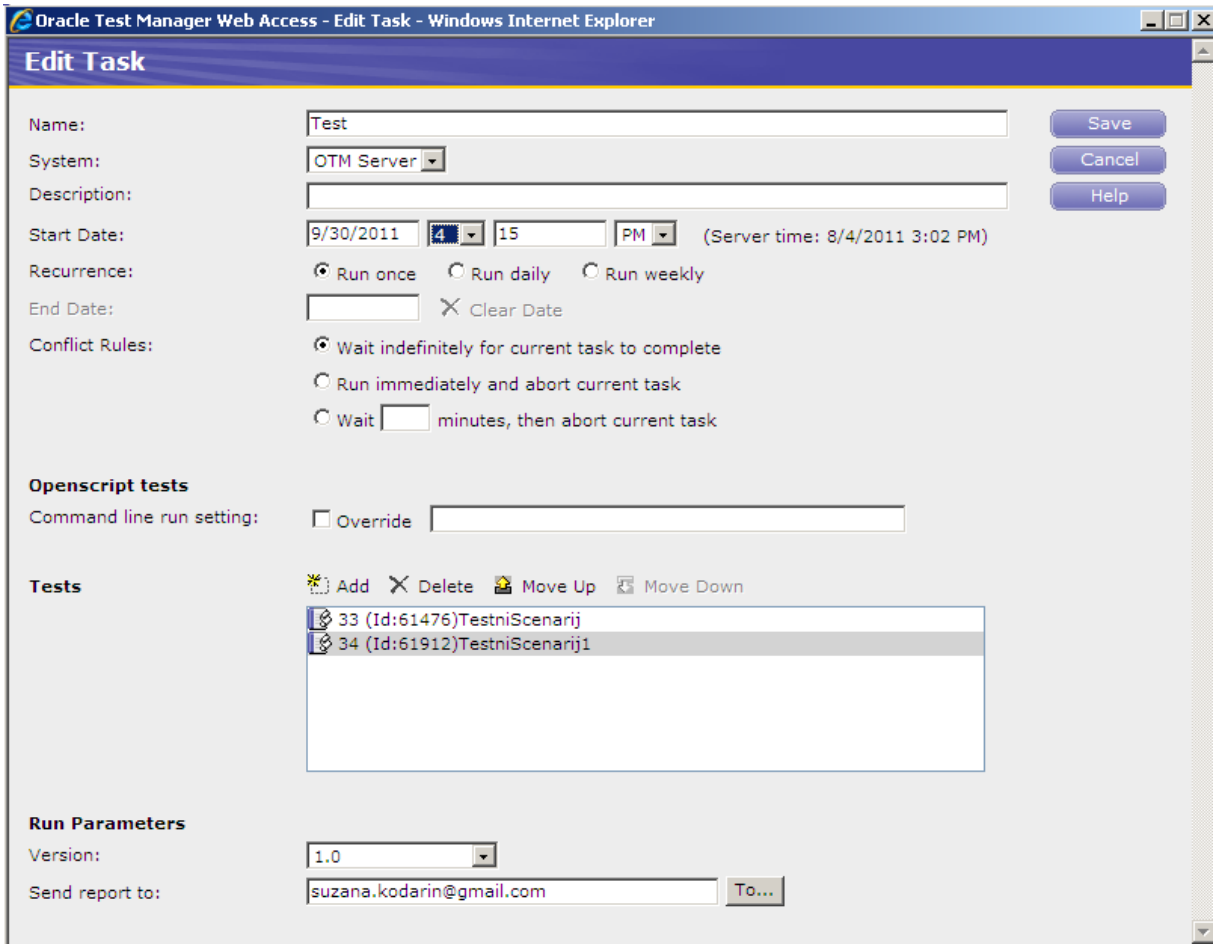
Slika 41: Izvoz nastavitev iz OpenScripta

Nato v polje *Command line run settings* vpišemo*:*

– propertiesPath "C:/additionalSettings.properties"

Posamezen test poženemo s klikom na gumb *"Run this Test"*.

OTM omogoča tudi avtomatično sprožitev izvedbe testiranja. Testiranje lahko izvedemo ponoči oziroma zunaj delovnega časa. S klikom na gumb *Schedule* (Urnik) se odpre okno, kjer izberemo testne primere, ki naj se izvedejo. Izpolnimo ime opravila, kdaj naj se izvede, katere testne primere naj izvede, in elektronski naslov, kamor se pošlje poročilo po opravljenem opravilu.



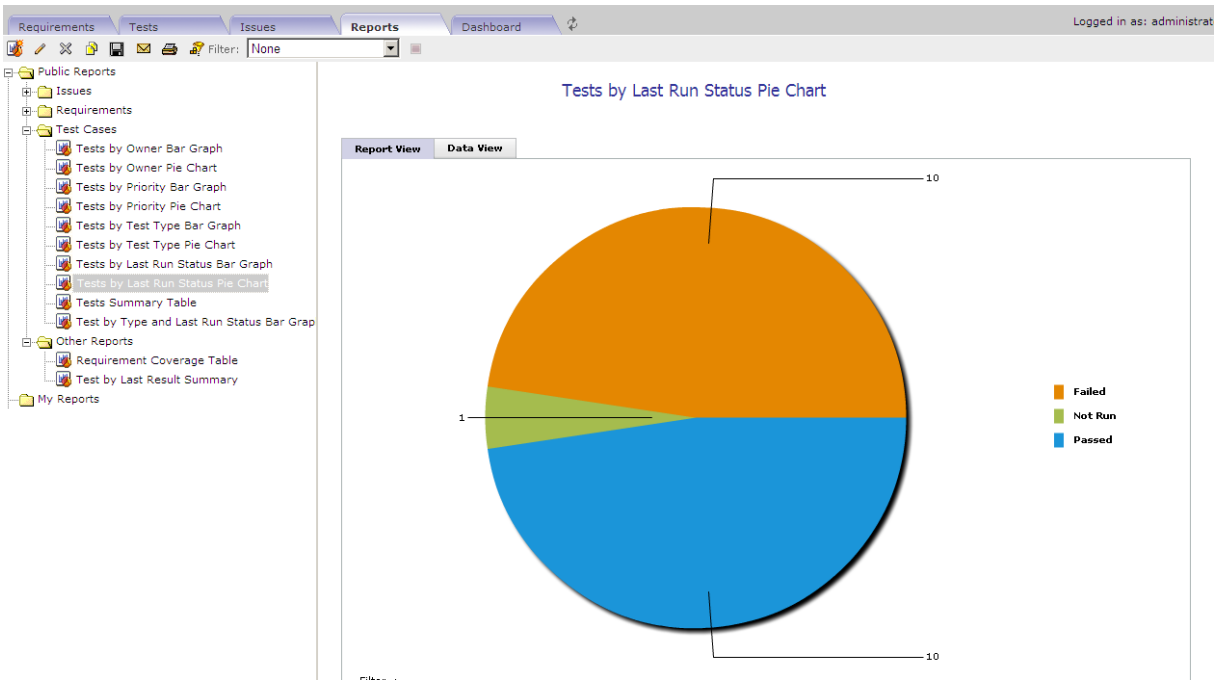
Slika 42: Urnik testov

Primer na sliki (Slika 42) se bo izvedel 30. 9. 2011 ob 16:15. Izvedla se bosta dva scenarija in poročilo testov bo poslano na suzana.kodarin@gmail.com.

Poročila lahko ustvarimo grafično ali podatkovno v formatu HTML. Poročila v formatu HTML imajo enako strukturo kot poročila v OpenScriptu (Slika 32).

V zavihku *Reports* lahko pripravimo grafična poročila izbranih opravil. Poročila lahko prikažejo rezultate glede na tip testa, status testa, lastnika testa, prioriteto testa in glede na podatke, kdo je nazadnje test spreminjal. Lahko pa prikažemo tudi razmerja med naštetim.

Slika prikazuje, koliko testov se je izvedlo uspešno, koliko neuspešno in koliko testov se ni izvedlo.



Slika 43: Grafični prikaz poročila

8 SKLEP

Temo o testiranju sem si kot temo diplomske naloge izbrala zato, ker sem želela bolje spoznati osnovne pojme testiranja in pomen avtomatizacije funkcionalnega testiranja. Pred tem si nisem predstavljala, kako zelo pomemben dejavnik za podjetja, ki delujejo na področju razvoja programske opreme, predstavlja testiranje. Za mnoga podjetja, ki vsak dan razvijajo, spreminjajo in popravljajo programsko opremo, so orodja za avtomatsko testiranje rešitev, ki omogoča njihovo večjo uspešnost.

Moj cilj je bil poleg spoznavanja osnov testiranja temeljito raziskati orodje Oracle OpenScript in spoznati čim več njegovih funkcij, zato da ga bom v prihodnosti čim bolje uporabljala.

Kazalo slik

9 Viri in literatura

24UR, Oddaja. 2007. 24ur. *24ur.* [Elektronski] 12. 2 2007. http://24ur.com/novice/slovenija/iz-24ur-pogovor-s-prvim-mozem-dursa.html.

Alley. 2009. Article Alley. *Article Alley.* [Elektronski] 2009. http://www.articlealley.com/article_915550_11.html.

Baša, Mateja. 2010. *Integracijsko testiranje v sistemih SOA.* Ljubljana : s.n., 2010.

Craig, D.R. 2000. *Systematic Software Testing.* Boston : Artech House, 2000.

Čebokli, Peter. 2006. *Avtomatizacija testiranja kot ključ agilnosti razvoja programske opreme.* Ljubljana : s.n., 2006.

Dogša, Tomaž. 1993. *Verifikacija in validacija programske opreme.* Maribor : Tehniška fakulteta, 1993.

Dustin, Elfriede. 1999. *Automated Software testing, Introduction, Management, and Performance.* USA : Addison-Wesley, 1999.

—. 2003. Effective software testing: 50 specific ways to improve your testing. [Elektronski] 2003. http://books.google.com/books?id=K0qWBUOAf6IC&lpg=PP1&hl=sl&pg=PP1%23v=onepage&q&f=false#v=onepage&q&f=false.

Fewster, Mark. 1999. *Software Test Automation: Efective user of test execution tools.* Great Britain : Addison Wesley, 1999.

Hoffman, Douglas. 1999. *"Cost Benefits Analysis of Test Automation", Software testing, Analysis and Review(STAR) conference.* 1999.

Jelnikar, Kristina. 2009. *Avtomatsko funkcionalno testiranje programske opreme.* Ljubljana : s.n., 2009.

Kaner, Cem. 1998. [Elektronski] 1998. http://www.kaner.com/pdfs/shelfwar.pdf.

Kočevar, Uroš. 2006. *Testiranje programske opreme in izdelava načrta za testiranje Time&Space sistema.* Ljubljana : s.n., 2006.

Kragelj, Boris. 2002. *Evalvacija spletnih predstavitev.* Ljubljana : Fakulteta za družbene vede, 2002.

Kristina, Jelnikar. 2009. *Avtomatsko funkcionalno testiranje programske opreme* . Ljubljana : s.n., 2009.

Kuzem, Rok. 2011. *Načrtovanje testiranja pri razvoju is v manjših razvojnih skupinah.* Ljubljana : s.n., 2011.

Marrick, Brian. 1997. *Classic Testing Mistakes.* 1997.

Mašinovič, Vesna. 2006. *Izboljšava procesa testiranja rešitev.* Kranj : s.n., 2006.

Myers, Glenford J. 1979. *The Art Of Software Testing.* New York : John Wiley & Sons, Inc, 1979.

—. 2004. *The Art of Software Testing 2nd ed.* New Yersey : John Wiley & Sons, Inc, 2004.

OpenScriptUsersGuide. [Elektronski] http://www.oracle.com/technetwork/oem/app-test/etest-101273.html.

Patton, Ron. 2005. *Software Testing.* Indianapolis : Sams Publisher, 2005.

Peter, Čebokli. 2006. *Avtomatizacija testiranja kot ključ agilnosti razvoja programske opreme.* Ljubljana : s.n., 2006.

Podgorelec, Vili. 2007. Informacijski sistemi. [Elektronski] 2007. http://mafalda.informatika.uni-mb.si/2007-08/isizr/predavanja/tisk/02-osnove.pdf..

Poljšak, Marija Goltez. 2005. *Testiranje informacijskega sistema v različnih razvojnih fazah.* Ljubljana : s.n., 2005.

Ross, Kelvin. 1998. *Practical Guide to Software Testing.* West Burleigh : K.J.Ross & Associates, 1998.

Schwaber, Carey. 2006. Teleconference Evaluating Functional Testing Tools. [Elektronski] 2006. http://www.forrester.com/Events/Content/0,5180,-1403,00.ppt.

Solina, Franc. 1997. *Projektno vodenjerazvoja programske opreme.* Ljubljana : Fakulteta za računalništvo in informatiko, 1997.

Stanojević, Dragana. 2009. *Organizacijsko informacijski model testiranja programske opreme.* Kranj : s.n., 2009.

Tomaž, Korelič. 2009. *Testiranje funkcionalnosti in zmogljivosti informacijskih sistemov.* Maribor : s.n., 2009.

Vetrih, Aleš. 2009. *Odprtokodno orodje za celovito izvajanje testiranja.* Ljubljana : s.n., 2009.

Zorko, Samo. 2004. *Lovci na žuželke.* s.l. : Moj Mikro, 2004.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO