UNIVERZA V LJUBLJANI
FAKULTETEA ZA RAČUNALNIŠTVO IN INFORMATIKO

Kristina Jelnikar

AVTOMATSKO FUNKCIONALNO TESTIRANJE PROGRAMSKE OPREME

DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Mentor: doc. dr. Mojca Ciglarič

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

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

Št. naloge: 01505/2008
Datum: 01.09.2008

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

Kandidat: **KRISTINA JELNIKAR**

Naslov: **AVTOMATSKO FUNKCIONALNO TESTIRANJE PROGRAMSKE OPREME**

AUTOMATED FUNCTIONAL SOFTWARE TESTING

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Opišite področje razvoja in testiranja programske opreme s poudarkom na funkcionalnem testiranju in na možnostih avtomatizacije tega testiranja. Navedite tipe orodij, ki se lahko uporabljajo ter opišite življenjski cikel avtomatskega testiranja po metodologiji ATLM in komentirajte tehnične izpeljave. Analizirajte metrike za uspešnost in kakovost testiranja in jih uporabite na konkretni implementaciji testa. Za zaključek kritično ovrednotite prednosti in slabosti avtomatskega testiranja.

Mentor:

M. Ciglarič

doc. dr. Mojca Ciglarič

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Franc Solina

Zahvala

Zahvaljujem se mentorici doc. dr. Mojci Ciglarič za vodenje in prijazno pomoč pri izdelavi diplomske naloge.

Posebna zahvala gre moji družini ter teti Veri, saj so me ves čas študija spodbujali in mi stali ob strani.

Zahvaljujem se Simonu Sakelšku in razvojni ekipi Špice International d.o.o., od katerih sem se v zadnjih dveh letih naučila veliko in zaradi katerih je tudi nastala tema diplomske naloge.

Zahvaljujem se tudi sestrični Petri Šijanec za lektoriranje diplomske naloge.

Na koncu pa bi se zahvalila še kolegom iz fakultete, brez katerih bi bile študijske ure veliko daljše in težje.

Kazalo vsebine

Seznam uporabljenih kratic

- ATLM *(ang. Automated Test Lifecycle Methodology)* – metodologija življenjskega cikla avtomatskega testiranja
- AUP *(ang. Agile Unified Process)* – agilno združen proces
- DLL *(dynamic link library)* – izvršilni modul, ki vsebuje kodo ali vire za uporabo v drugih aplikacijah ali DLL-ih
- DUnit – testno okolje za avtomatsko testiranje modulov v Borland Delphi-ju
- MSSQL – strežnik podjetja Microsoft
- ODBC *(angl. Open DataBase Connectivity)* – odprta povezljivost z zbirkami podatkov
- RAD *(angl. rapid application development)* – hiter razvoj aplikacij
- RUP *(ang. Rational Unified Process)* – objektna metodologija razvoja programske opreme
- ROI *(ang. Return Of Investment)* – donosnost naložbe
- SCRUM – agilna razvojna metodologija za upravljanje projektov
- SDLC (Software development Life Cycle) – življenjski cikel razvoja programske opreme
- TDD *(ang. Test Driven Development)* – testno voden razvoj
- T&S (Time&Space) – produkt podjetja Špica International d.o.o. za zajem ter obdelavo registracij delovnega časa
- VMWare – virtualna programska oprema
- XP *(ang. Extreme Programming)* – ekstremno programiranje

Povzetek

V diplomskem delu je opisan pristop k avtomatizaciji funkcionalnega testiranja programske opreme.

V uvodnem delu je predstavljena problematika testiranja programske opreme v razvojnih podjetjih.

V drugem poglavju so opisane metode testiranja, kam spada testiranje v razvoj programske opreme, nekateri pristopi k razvoju programske opreme ter testno okolje.

V poglavju 3 je predstavljeno avtomatsko testiranje. Po kratki zgodovinski predstavitvi je naštetih nekaj dejstev, zakaj bi se avtomatskega testiranja bilo dobro lotiti. Nato so predstavljeni tipi orodij, ki jih lahko uporabimo pri avtomatskem testiranju ter metodologija ATLM. V poglavju 3.6 se prične predstavitev tehnične plati avtomatskega testiranja. Opisane so skriptne tehnike, metode avtomatske primerjave rezultatov, arhitektura testvera, priprava podatkov, pred in poprocesne procedure. Proti koncu je predstavljen režim, ki je zaželen pri avtomatskem testiranju. (Režim določa, kako upravljamo z avtomatskim testiranjem, kako se lotimo implementacije avtomatskega testiranja in kako je organiziran testver.) V poglavju 3.10 so podani napotki pri izbiri testnega orodja.

V poglavju 4 so predstavljene metrike, s katerimi merimo in nadzorujemo napredek ali potencialne nevarnosti v procesu avtomatskega testiranja.

5. poglavje je namenjeno predstavitvi implementacije v organizaciji. Predstavljen je pilotni projekt TsStartup.exe, ki smo ga preko ATLM metodologije pripeljali do avtomatizacije.

V zaključku so predstavljene sklepne ugotovitve in smernice za nadaljnji razvoj.

Ključne besede: testiranje programske opreme, avtomatsko testiranje programske opreme, testver, testno orodje, ATLM

Abstract

The following work describes an approach to software test automation of functional testing. In the introductory part we are introducing what testing problems development companies are facing.

The second chapter describes the test methods, what role does testing have in software development, some approaches to software development and the meaning of testing environment.

Chapter 3 is all about test automation. After a brief historical presentation, we are demonstrating through some facts why test automation is a good idea. We continue with types of tools, which can be used in automatic testing. We also introduce the reader with the ATLM methodology. In chapter 3.6 the technical aspects of test automation are emphasized. Scripting techniques, methods of automatic comparison, the testware architecture, data preparation, pre-and post-processing procedures are described. Toward the end a desirable automated testing regime is presented. (The regime sets out how to manage automated testing, how to address the implementation of automated testing and how testware should be organized.) In Chapter 3.10 guidance for selection of functional test tools is provided.

In chapter four we describe the metrics that measure and monitor the progress or potential danger in the process of automatic testing.

Section five is devoted to the presentation of implementation in the organization. The pilot project TsStartup.exe is presented. We have automated the project with ATLM.

At the end, the final conclusions and guidelines for further development are presented.

Keywords: software testing, software test automation, testware, test tools, ATLM

1 Uvod

Danes je programska oprema prisotna že v vsakem poslovnem sistemu. Uporabniki pričakujejo, da bo ta delovala pravilno, konsistentno in predvsem po njihovih pričakovanjih. Ta pričakovanja so razumljiva, saj vplivajo na pravilno, nemoteno in učinkovito delo uporabnikov.

V današnjem času morajo podjetja svoje poslovne procese hitro prilagajati razmeram na trgu. Z razvojem poslovnih procesov pa vedno znova prihaja tudi do novega razvoja programske opreme ter njenih izboljšav. Ta razvoj se odvija hitro in je velikokrat omejen s časovnimi roki. Zaradi teh kratkih časovnih rokov ter seveda človeškega faktorja zmotljivosti, pa se dogaja, da pride med razvojem do številnih napak, morda celo do napačne zasnove produkta, ki bi utegnila biti bolj ali manj usodna za uporabnikovo delovanje. Da ne bi bilo teh napak preveč, oziroma, da bi izločili vsaj tiste kritične, je za odkrivanje teh napak v procesu razvoja programske opreme zadolžena testna ekipa. Ta skozi celoten razvojni cikel programske opreme preverja ali jo gradimo pravilno in ali ustreza osnovnim zahtevam.

Časovni roki vplivajo obremenilno na razvojno ekipo, ki gradi produkt in tudi na testno ekipo, ki po navadi dobi v test produkt tik pred iztekom roka. Pod časovnim pritiskom se tako testna ekipa navadno sooča z velikim obsegom dela ter pomanjkanjem delovne sile.

Pravilnost programske opreme se v veliki večini primerov še preverja ročno. Se pravi, da mora testna ekipa ob vsaki spremembi programske opreme (popravek napake, sprememba, nova verzija, itd.) ročno preveriti njeno pravilno delovanje.

Kljub vsem naštetim oviram mora biti testiranje učinkovito pri iskanju napak, ki so v našem sistemu, istočasno pa naj bo izvedeno hitro in poceni.

1.1 *Motivacija in cilj diplomske naloge*

Kot del testne ekipe sem se v podjetju Špica International d.o.o. pobliže spoznala s področjem testiranja programske opreme. Delo testne ekipe je zahtevno, saj morajo člani testne ekipe obvladati množico različnih znanj, ki so potrebna za specifičnost tega dela. Tako usposobljene ljudi je težko dobiti, zaradi narave dela (ki je zahtevno, redundantno, sčasoma postane dolgočasno) pa še težje obdržati. V testni ekipi smo se odločili, da poizkusimo uvesti v testni proces avtomatsko testiranje, ki bi lahko spremenilo časovno zahtevno, dolgočasno in redundantno opravilo v dinamično opravilo s povečanim obsegom testiranja v omejenem času.

Cilj diplomske naloge je preko uvedbe avtomatskega testiranja dokazati izboljšanje procesa testiranja ter zmanjšanje stroškov testiranja. Preučili bomo, katere tehnike in metode avtomatskega testiranja so primerne, katero orodje bi bilo primerno za avtomatizacijo produkta T&S, ter spremljali koristnost uvedbe avtomatizacije preko metrik. Želimo si torej najti primerno rešitev, ki bo izboljšala proces testiranja funkcionalnih testov, bo efektivna in jo bo lahko vzdrževati.

2 Testiranje programske opreme

Uvodoma sem na kratko opisala problematiko testiranja programske opreme. Skozi diplomsko delo se bom tej tematiki še posvetila, vendar pa je prav, da še prej zapišem nekaj o osnovah testiranja.

V tem poglavju bom umestila testiranje v življenjski cikel razvoja programske opreme ter opisala najbolj zanimive metode ter postopke testiranja, predstavila bom tudi nekaj dejstev, ki nakazujejo, zakaj je dobro razmišljati v smeri avtomatizacije testiranja.

2.1 *Metode testiranja*

V zadnjih 25 letih so se razvili 4 različni pristopi k testiranju. Ti štirje pristopi vključujejo [1]*:*

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

Po načinu testiranja lahko ločimo metode testiranja na:

- statične metode (ne zahtevajo izvajanja programa) in
- dinamične metode

Pri **statičnih metodah** gre za branje dokumentov oziroma sledenje programski kodi, zato lahko govorimo tudi o ročnih metodah testiranja. Na pregledovanje in izpopolnjevanje dokumentacije se velikokrat gleda kot na nekaj, kar se ureja na koncu projekta, vendar pa bi morali razmišljati ravno obratno. Boljša kot bo dokumentacija in to še posebej v začetnih fazah (zahteve, načrt, specifikacije) večja je možnost, da bo razvojna ekipa napisala dobro kodo.

Pri **dinamičnih metodah** pa izvajamo program in njegove rezultate primerjamo s pričakovanimi rezultati.

Glede na vsebino testiranja ločimo vrste testiranja na:

- metode bele skrinjice ali strukturno testiranje in
- metode črne skrinjice ali funkcionalno testiranje

Metode bele skrinjice ali **strukturna analiza**. Metode v tej skupini pridejo v poštev, ko imamo na voljo izvorno ter izvršljivo kodo. Taka situacija je običajna za razvijalce, ki delajo na produktu (razvijajo nov produkt oz. nadgradnjo že obstoječega produkta). Po principu bele skrinjice testiramo predvsem posamezne module. Princip bele skrinjice pomeni, da imamo na vpogled notranjo strukturo modula. Na tej osnovi tudi načrtujemo testne primere. Med metode bele skrinjice sodita:

1. *Testiranje glavnih poti.* Osnovni namen metode je, da se vsi programski ukazi izvedejo vsaj enkrat.
2. *Testiranje zank.* Izkušnje kažejo, da pri kodiranju zank še posebej pogosto naredimo napake.

Metode črne skrinjice ali **funkcionalna analiza**. Pri teh metodah testiranja je notranja struktura kode zastrta. Izbiramo lahko različne vhodne podatke in dejanske izhodne podatke primerjamo s pričakovanimi. Testiranje po principu črne skrinjice uporabljamo v kasnejših fazah testiranja za preverjanje funkcionalnih zahtev. Zato funkcionalno testiranje ni nadomestilo za strukturno testiranje, saj odkriva druge vrste napak, npr.:

- napačne ali manjkajoče funkcije
- napake vmesnika
- nizko zmogljivost
- napake pri inicializaciji in na koncu procesiranja

Torej po vsebini testiranja na sistem gledamo z dveh različnih perspektiv. Oba pristopa sta enakovredna in se dopolnjujeta. Uporabljamo ju hkrati na različnih ravneh testiranja. Testiranje bele skrinjice zahteva dostop do izvorne kode aplikacije, ki pa ni vedno na voljo (npr. če v aplikacijo vključimo kupljene komponente). Ker se v praksi oba pristopa prepletata, nekateri predlagajo nov termin “siva skrinjica”, ki naj bi poudaril nujnost uporabe obeh pristopov za uspešno testiranje. Pri metodi sive skrinjice poznamo nekaj "notranjega" delovanja aplikacije, kar prinese boljše razumevanje aplikacije ter boljše rezultate testiranja. Tako metode črne in sive skrinjice kot tudi metode bele skrinjice lahko avtomatiziramo.

2.2 *Razvoj programske opreme ter postopek testiranja*

Vsako zrelo in konkurenčno podjetje, ki se ukvarja z razvojem programske opreme, se skozi celotni razvojni cikel programske opreme prizadeva narediti kvaliteten produkt. Kot bomo videli kasneje, se v vsaki fazi razvojnega cikla uporabljajo orodja, povezana s testiranjem, najprej pa si poglejmo, v katere faze je razvojni cikel razvoja programske opreme navadno razdeljen [2]:

A **Analiza**
V tej fazi se s stranko dogovorimo o sistemu, ki ga bomo gradili. Da bi bila specifikacija kar se da idealna, je potrebno definirati sistem čim bolj celovito ter nedvoumno. V realnosti je potrebno vnašati popravke tudi med razvojem programske opreme. Pregled te faze navadno naredi načrtovalna ekipa, da razreši konflikte in manjkajoče zahteve. S tem procesom lahko odkrijemo precejšnje število napak. Sprememba zahtev v kasnejših fazah razvoja lahko povzroči povečano gostoto napak.

B **Načrtovanje**
V tej fazi specificiramo sistem kot med seboj povezane enote, tako da je vsaka enota dobro definirana in jo lahko razvijemo in testiramo neodvisno od drugih. Načrt mora biti preverjen, da se izognemo napakam.

C **Kodiranje**
V fazi kodiranja se piše dejanski program za vsak posamezen modul. Navadno se piše v enem izmed višjih programskih jezikov (c, c++, java, c# …) Da ne bi prihajalo do napak, se izvajajo pregledi kode na skupinskih sestankih.

D **Testiranje**
Ta faza je kritični del za visoko kvaliteto in zanesljivost. Lahko nam vzame tudi 30–60 % celotnega razvojnega časa. Poteka na različnih ravneh, v različnih okoljih in s strani različnih vlog.

Večinoma testiranje delimo na posamične faze:

Testiranje modulov

V tej fazi se vsak modul testira posebej. S testiranjem modulov se ukvarjajo razvijalci, ki opravljajo test v razvojnem okolju. Ker je vsak modul relativno majhen in ga lahko testiramo neodvisno, ga lahko pregledamo veliko bolj podrobno kot celoten oz. velik program. Testiranje modulov je osredotočeno na ustreznost uporabniškega vmesnika in funkcionalnost programske enote glede na podane zahteve in standarde.

Integracijski test

Testiranje prevzame testna ekipa. Med integracijo se moduli postopoma sestavljajo v podsisteme in te delno sestavljene podsisteme je potrebno testirati. S tem ko sistematično dodajamo module v podsistem, lažje odkrijemo modul, ki je odgovoren za določeno napako. Pri testiranju naj s testno ekipo sodeluje ključni uporabnik, ki tudi potrdi ustreznost podsistema.

[3] Integracija lahko poteka od zgoraj navzdol ali od spodaj navzgor. Katero pot je smiselno ubrati, je odvisno od težavnosti testiranja. Med testiranjem moramo namreč module, ki še niso integrirani v sistem, nadomestiti z navideznimi moduli.

Integracija od zgoraj navzdol

Začnemo z glavnim ali kontrolnim modulom. Postopoma dodajamo podrejene module, v globino ali v širino. Če integriramo najprej v globino, lahko prej demonstriramo določeno funkcijo sistema. Potrebni so le začasni podrejeni moduli. Problemi lahko nastopijo le, če je simuliranje procesiranja v nižje ležečih modulih težavno.

Integracija od spodaj navzgor poteka tako, da module na nižjem nivoju združimo v skupine. Napisati moramo gonilnike, ki koordinirajo testiranje skupin modulov. Postopoma odstranjujemo gonilnike, jih nadomeščamo s pravimi moduli ter skupine združujemo. Da bi zmanjšali število potrebnih navideznih modulov, ali da bi najprej testirali najbolj kritične module, lahko kombiniramo oba načina sistemske integracije.

Sistemsko testiranje

Sistem kot celota se preizkuša med sistemskim testiranjem. Sistem naj bi že pravilno deloval, preveriti pa moramo, ali sistem res deluje tako, kot je predpisano v specifikaciji – to je *validacija* sistema. Pri izvedbi testa sodelujejo testna ekipa, ključni uporabniki in skrbnik aplikacije. Zanima nas hitrost programske opreme, zmožnost okrevanja sistema po izpadu sistema, maksimalna obremenitev sistema, varnost in občutljivost sistema, kaj se zgodi pri vnašanju napačnih podatkov in ukazov in podobno.

Pri *testiranju okrevanja sistema* programsko opremo na različne načine prisilimo, da izpade. Nato opazujemo, če sistem pravilno in hitro okreva, tako kot od njega zahtevamo, bodisi avtomatično bodisi z operatorjevim posredovanjem.

Pri *testiranju varnosti* moramo predvsem ugotoviti, če sistem preprečuje neavtoriziran dostop.

Pri *testiranju obremenitve* sistem izpostavimo nenormalnim obremenitvam (število uporabnikov, število vhodnih podatkov, poraba spomina, itd.) in opazujemo, kdaj sistem izpade ali postane neuporaben.

Pri *testiranju občutljivosti* skušamo odkriti, ali lahko določene kombinacije vhodnih podatkov, ki so posamezno obravnavani sicer pravilni, povzročijo nestabilnost sistema oziroma napačno procesiranje.

Potrditveni test
Namen te faze je preveriti sistemsko zanesljivost in zmogljivost v delovnem okolju. Programsko opremo, namenjeno širšemu krogu uporabnikov, pred prodajo pogosto damo v poskusno uporabo zainteresiranim posameznikom ali skupinam v obliki tako imenovanih verzij α in β.

Regresijski test
Ko verziji dodamo oziroma spremenimo neko funkcionalnost, se test izvede na novi dnevni izdaji (angl. *build*), da zagotovimo enako dobro oziroma nikakor ne slabše delovanje programa.

E **Produkt v uporabi**
Ko se razvojna ter testna ekipa strinjata, da je zanesljivost programske opreme dovolj dobra, izdajo končni produkt – programsko opremo. Vse napake, ki se pojavijo, se zapišejo, vendar pa se popravijo šele ob naslednji izdaji.

2.3 *Pristopi k razvoju programske opreme*

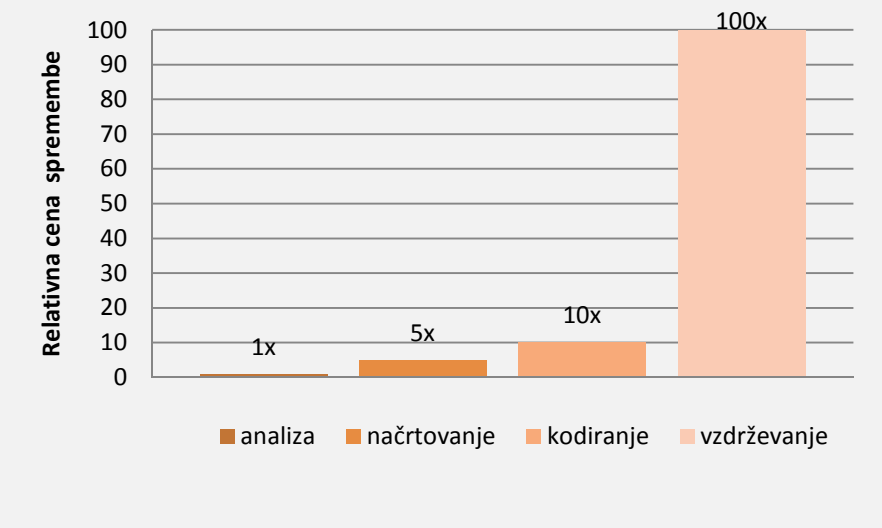
Kot večina razvojnih procesov sledi tudi razvoj programske opreme določenemu življenjskemu ciklu oziroma razvojnemu modelu, ki določa zaporedje faz razvoja. Življenjski model razvoja sistema (SDLC[1]) pove, v kakšnem sosledju in na kakšen način si v okviru razvoja sistema sledijo posamezne faze. Poznamo različne življenjske modele (zaporedni ali slapovni model, iterativni model, prototipni model, inkrementalni model …), v praksi pa se večinoma uporablja kombinacija različnih modelov.

2.3.1 Klasični zaporedni ali slapovni model

Eden prvih razvojnih modelov je bil *klasični zaporedni model*, ki se ga še danes uporablja v nekaterih primerih. Vse aktivnosti (analiza, načrtovanje, kodiranje, testiranje in implementacija) si sledijo zaporedno. In tako je tudi faza testiranja samostojna faza, ki sledi fazi kodiranja. Vračanje nazaj ni mogoče. Ta model ni fleksibilen in vsaka naknadna sprememba zahteva veliko dodatnega napora. V praksi je težko pričakovati, da bomo nek postopek v celoti zaključili, preden bomo začeli z naslednjim. Ne omogoča paralelnega izvajanja delov postopkov. Kljub vsemu pa nudi zelo čvrsto oporo sistematičnemu razvoju. Možno ga je uporabiti v kombinaciji z drugimi modeli.

Tak model je iz stališča cene spremembe oziroma napake skorajda nedopusten, saj je cena spremembe po fazi kodiranja že za 100 krat večja, kot bi bila cena spremembe v času analize (slika 2.1).

[1] SDLC – System Development Life Cycle



Slika 2.1 [3] Cena sprememb v programski opremi od faze analize do faze vzdrževanja strmo narašča.

Danes vemo, da je kvaliteta programske opreme odvisna od celotnega razvojnega procesa. Z nadzorom razvoja programske opreme je potrebno napake že vnaprej preprečevati in sproti odkrivati pomanjkljivosti. Kasneje, ko v toku razvoja programske opreme kaj spremenimo, dražja je sprememba. Ugotovitev, da nam manjka neka funkcija po fazi kodiranja nas lahko stane 100 kratno ceno iste ugotovitve v fazi analize zahtev. Testiranje je zato integralni del nadzora kvalitete celotnega razvojnega cikla, pri katerem moramo poskrbeti, da je rezultat vsake razvojne faze kvaliteten in v skladu z globalnimi cilji.

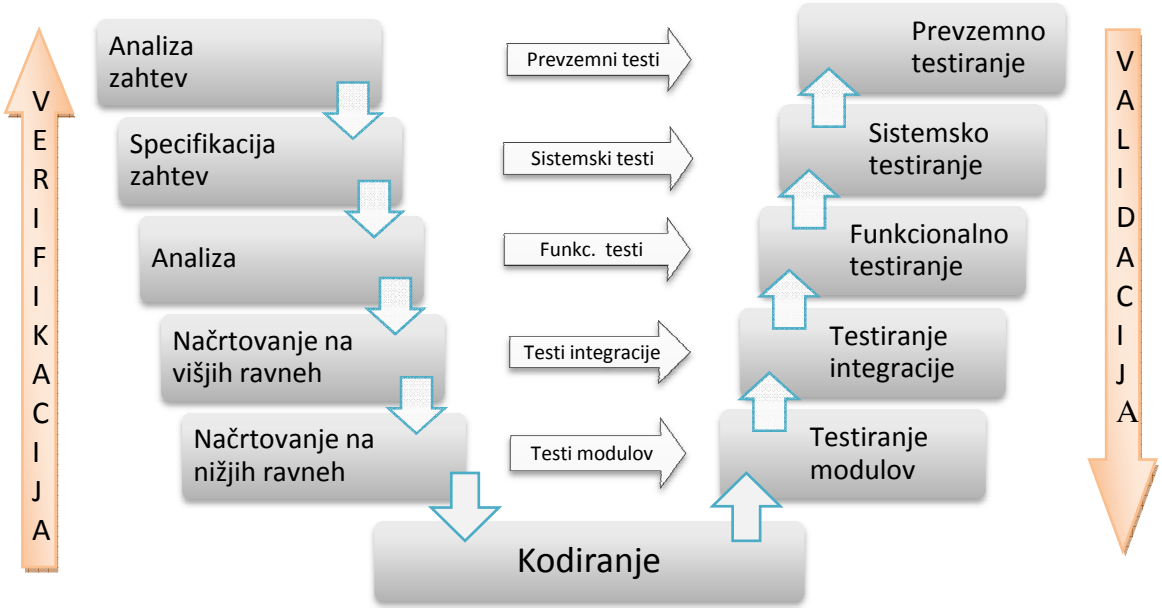
Uspešnost posameznih faz ugotavljamo z recenzijami, kjer preverjamo [3]:

1. Ali programsko opremo gradimo pravilno? Ta postopek imenujemo *verifikacija*. Verifikacija ugotavlja predvsem tehnično pravilnost zadnje faze razvoja.
2. Ali programska oprema ustreza osnovnim zahtevam? Ta postopek imenujemo *validacija*. Validacija preverja, če gradimo pravi produkt.

2.3.2 Testiranje življenjskega cikla ali V-testiranje

V-model je izboljšana različica klasičnega zaporednega modela. Cilj V-testiranja je ujeti napake, kar se da zgodaj. S tem se tudi zmanjšajo stroški popravila napak. Odkrivanje napak v zgodnjih fazah razvoja dosežemo s tem, da nenehno preverjamo sistem med vsemi fazami razvojnega procesa in ne omejujemo testa na zadnjo fazo.

Ko pričnemo z razvojem projekta, se mora pričeti tudi testiranje projekta. Razvojna ekipa prične z razvojem, testna ekipa pa prične s planiranjem testnega procesa. Obe ekipi naj bi začeli na isti točki z istimi informacijami. Ob določenih točkah med razvojnim procesom bo testna ekipa preverila razvojni proces z namenom, da odkrije morebitne napake.



Slika 2.2 V-model

Na testiranje dostikrat pomislimo kot na posamezno fazo, s katero se ukvarjamo šele po zaključnem kodiranju. Saj šele po fazi kodiranja dobimo produkt, katerega delovanje je treba preveriti in ne moremo testirati nečesa, kar še ne obstaja. Vendar pa ne smemo biti omejeni le na izvajanje testov. Potrebno je tudi preverjati načrte, dokumentacijo, itd. V-model razvojnega cikla programske opreme prikazuje, kdaj naj bi se izvajale testne aktivnosti. Vsaka razvojna aktivnost naj ima pripadajočo testno aktivnost. [4] **Ta princip velja za vsak razvojni model programske opreme.** V primeru hitrega razvoja aplikacij (Rapid Application Development - RAD) bi imeli serijo malih V-modelov .

2.3.3 Iterativni in inkrementalni razvoj

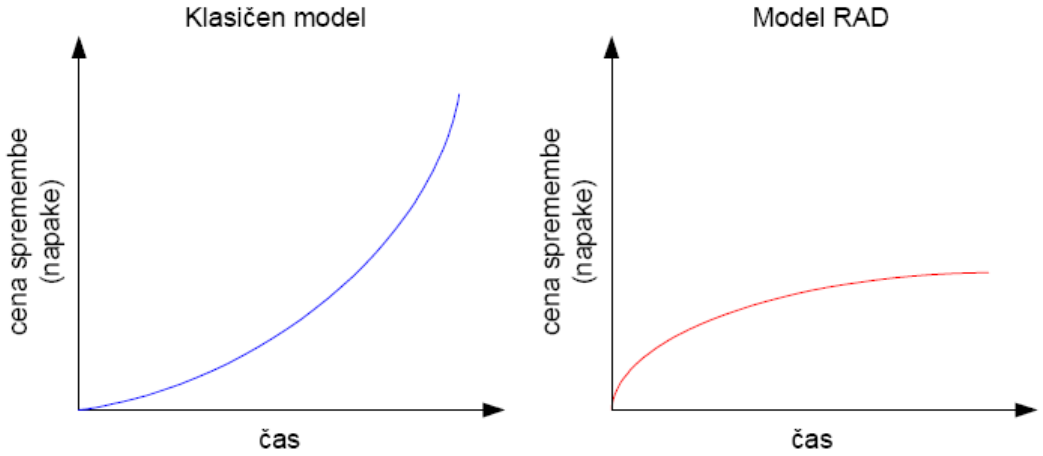
Iterativni in inkrementalni razvoj je cikličen proces, ki je bil razvit kot odgovor na slabosti zaporednega modela.

Pri *inkrementalnem* razvoju razdelimo sistem na več koščkov in jih razvijemo ob različnih časih ter v različnih deležih in jih integriramo, ko so končani. Testiranje poteka na vsakem koščku posebej ter seveda ob integraciji koščkov v sistem. Alternativa inkrementalnemu razvoju je razvoj celega sistema z integracijo celotnega sistema naenkrat.

Pri *iterativnem* razvoju se končni različici izdelka približujemo postopoma skozi več iteracij in ne z enim zamahom kot pri zaporednem modelu. Za razvoj manjših delov funkcionalnosti, ki jih rešujemo korak za korakom uporabimo zaporedni model. V vsaki iteraciji gremo čez vse faze zaporednega modela. Se pravi se testiranje izvede ob vsaki iteraciji, rezultate posamezne iteracije (ugotovljene napake, zahteve) pa se upoštevajo ob razvoju naslednje iteracije.

Z vedno hitrejšim spreminjanjem zahtev se je pojavila potreba po drugačnih metodologijah. Nastale so metodologije za hiter razvoj programske opreme (“Rapid Application Development” ali RAD), ki temeljijo predvsem na iterativnem življenjskem ciklu in prototipiranju. Metodologije RAD dajejo večji poudarek na implementaciji in testiranju,

manjši pa na analizi in načrtovanju. Poudarek na tem modelu so hitre iteracije čez cikel. Prototipi so načrtovani, razviti in pregledani z uporabniki, ki jih vklopimo v proces. Ta model je še posebej primeren za projekte v hitro spreminjajočem okolju, kjer se mora ekipa hitro prilagajati različnim situacijam.



Slika 2.3 [5] Naraščanje cene spremembe (odprave napake) s časom po klasičnem modelu in modelu RAD

Iterativni in inkrementalni razvoj je poglavitni del novih metodologij RUP, XP in na splošno agilnih metodologij razvoja programske opreme.

2.3.4 Nove metodologije razvoja programske opreme

V zadnjem času je naraslo število projektov, ki se opirajo na t.i. lahke ali agilne metodologije. Njihovi temeljni lastnosti sta učinkovitost in prilagodljivost.

Med znane metode agilnega razvoja programske opreme spadajo SCRUM, ekstremno programiranje (XP- Extreme programming), agilno enovit proces (AUP - Agile Unified Process), itd.

Testiranje se izvaja redno in ni posebne testne faze. Potrebno je testirati zgodaj in pogosto. Ko implementiramo neko funkcionalnost, jo je potrebno testirati. S pogostimi iteracijami, je potrebno pogosto tudi testirati. Potrebni so konstantni regresijski testi. Pri testiranju modulov se navadno uporablja testno voden razvoj (TDD - Test-driven development), kjer programerji napišejo testno kodo pred kodiranjem samega modula. Vendar pa testi modulov niso dovolj. Potrebno je izvesti tudi efektivne teste na uporabniškem nivoju. Potrebujemo nek nivo avtomatskih testov na uporabniškem nivoju.

Če vzamemo za primer ekstremno programiranje, se je potrebno osredotočiti na dve vrsti testiranja: testiranje modulov in prevzemno testiranje.

Teste modulov ali programerske teste pišejo programerji hkrati s programsko kodo ali pa preden začnejo programirati, če izvajamo testno usmerjeni razvoj. Namen testov je hitro preverjanje delovanja celotnega programskega sistema po vsakem posegu v sistem. Tega seveda ni možno učinkovito narediti z ročnim, ad-hoc testiranjem, temveč je za to treba narediti avtomatske programe. Po vsaki spremembi programske kode (npr. preoblikovanju

neke procedure) mora avtor spremembe pognati vse avtomatske teste in se tako prepričati, da programska koda deluje za primere, ki so bili predvideni.

Pri prevzemnem testiranju stranka izdela prevzemne teste, in sicer po definiranju primerov uporabe (ali uporabniških zgodb). Namen teh testov je preveriti ali sistem deluje tako, kot to zahteva stranka (validacija). Zaželeno je, da so tudi ti testi avtomatizirani. Prevzemno testiranje se izvaja bolj pogosto in veliko bolj zgodaj kot pri linearnih razvojnih modelih.

S prihodom agilnih metodologij (projekti so manjšega obsega in gostote), ki se hitro odzivajo na spremembe v zahtevah (okolju), sta postala razvoj in testiranje bolj povezana. Testna ekipa mora zgodaj testirati vsak inkrement v razvoju programske opreme. Njihova vloga pa je večja tudi pri planiranju, podpori in pregledu testov modulov ter testov na nivoju komponent. Razvojna ekipa pa naj bi se poleg testa modulov bolj aktivno vključevala v avtomatizacijo sistemskih testov.

Agilni procesi za glavni kontrolni mehanizem raje uporabijo povratno informacijo kot planiranje. Povratna informacija se dobi na podlagi rednih testov ter izdaj razvijajoče programske opreme.

Pristope k razvoju programske opreme sem opisala z vidika testiranja, več o pristopih k razvoju programske opreme si lahko preberete v [1, 3, 4, 5].

2.4 *Testno okolje*

Preden gre programska oprema v produkcijo, jo je potrebno testirati v okolju, ki bo čim bolj podobno produkcijskemu. Pripraviti je potrebno ločeno (računalniško) okolje, v katerem se bo testirana aplikacija obnašala tako kot v produkcijskem okolju. V takem okolju bomo lahko med izvajanjem programa opazovali rezultate izvajanja, do katerih bi prišel tudi uporabnik. Dobro testno okolje je tudi popolnoma pod nadzorom testne ekipe. Zato naj bo ločeno od razvojnega okolja in vseh drugih projektnih okolij, ki bi lahko vplivala na rezultate izvajanja. Bolj kot je podobno realnemu okolju, boljše rezultate testiranja lahko pričakujemo.

Na testno okolje moramo misliti zgodaj v razvoju, saj se mora testna ekipa pripraviti na test pravočasno. Poleg planiranja in načrtovanja je potrebno pridobiti tudi morebitno programsko ter strojno opremo in jo pripraviti še pred prvim načrtovanim testom.

Kot je omenjeno v poglavju 2.3, čas odkritja napake vpliva na ceno popravka napake. Če se ne posvetimo vzpostavitvi dobrega testnega okolja, imamo veliko večjo verjetnost, da bo stranka odkrila napako namesto nas. V tem primeru bodo naši stroški znatno večji, kot bi bili, če bi se posvetili testiranju v pravem okolju pravočasno.

3 Avtomatsko testiranje

3.1 *Zgodovina*

Orodja za testiranje programske opreme so se začela pojavljati v 70-ih letih prejšnjega stoletja. LINT, preverjalnik kode, se je pojavil kot del starega Unix sistema. Ena prvih naprav za preverjanje kode je bil JAVS. Razvil ga je Edward Miller leta 1974 za preverjanje strukturnega pokrivanja. V sredi 80-ih let je postalo računalništvo dostopno širši množici ljudi in se je dodobra razširilo. Nastale so močne programske tehnologije, kot so razvojna okolja, sistemi podatkovnih baz in tako omogočile razvoj orodij, ki lahko zajemajo in analizirajo veliko množico podatkov. Tako imamo danes na voljo že mnogo orodij z zmožnostmi, kot so posnemi in predvajaj, primerjavo, preverjanje pokritja kode, itd.

Nekaj orodij, ki se ukvarjajo s področji nadzora razvoja programske opreme je že skorajda nepogrešljivih v razvojnih okoljih. Orodja za odkrivanje puščanja pomnilnika so se pojavila leta 1992 in so se izkazala za nepogrešljiva v razvojnih organizacijah. Ko smo bili na prelomu tisočletja, so se razvila mnoga orodja za preverjanje Y2K problema. Danes pa je na tržišču že prek 300 orodij, ki ponujajo in obljubljajo avtomatizirano testiranje.

Mnoge organizacije se še vedno branijo uvedbi testnih orodij. Raziskave kažejo, da je razlog predvsem v težavni, strmi učni krivulji, s katero se mora spopasti testna ekipa. Ta se mora že tako spopadati s časovnimi roki in ne najde časa za odkrivanje novih pristopov. Nekatera orodja niso pokazala takih rezultatov, kot so jih obljubila, vendar pa se konstantno izboljšujejo, tako da bomo z večjim poznavanjem teh orodij lahko tudi pridobili na kvaliteti programske opreme. Danes si noben razvojnik strojne opreme ne predstavlja, da bi izdelal načrt brez simulacijskih orodij. In skorajda nihče več ne dela ročnih testov za strojno opremo. Verjetno se bo v naslednjih letih ta trend uveljavil tudi v programski opremi [2].

3.2 *Utemeljimo, zakaj naj bi se lotili avtomatskega testiranja*

V uvodu je zapisano, da so kratki časovni roki za dobavo programske opreme v precejšnji meri krivi za slabo kvaliteto izdane programske opreme.

Podkrepimo to izjavo:

- V tabeli 3.1 je prikazano, koliko časa je navadno posvečeno testiranju v razvoju programske opreme. Ker se kompleksnost in velikost programske opreme veča z vsako verzijo, dnevi posvečeni testiranju pa zaradi časovnih rokov in omejenih resursov ostajajo isti, se soočamo s slabšo kvaliteto programske opreme. Relativni čas, ki ga lahko posvetimo testiranju, se krajša.

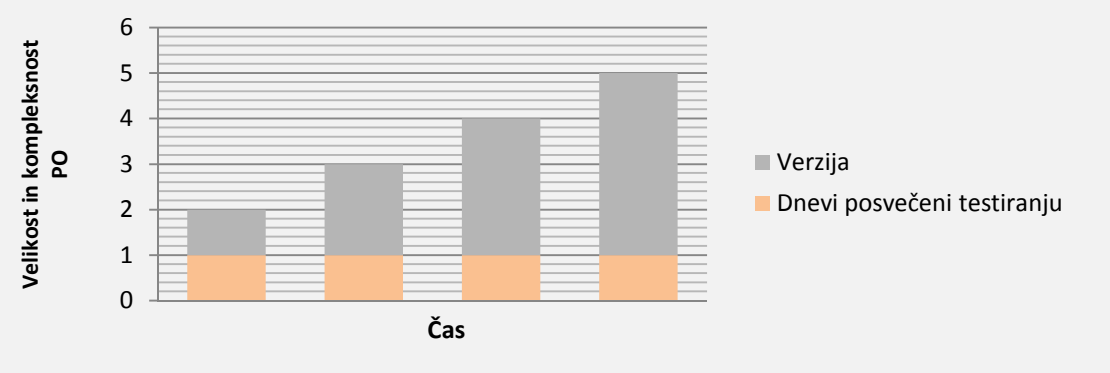


Tabela 3.1 Potencialna težava pri testiranju, če imamo premalo resursov. Kompleksnost in obseg programske opreme se navadno veča z vsako verzijo in če imamo premalo resursov, dnevi posvečeni testiranju ostajajo isti. Relativni čas, ki ga lahko posvetimo testiranju, se krajša.

Ogromno časa in denarja se posveča testiranju programske opreme.

- Navadno nas testiranje stane vsaj 50 % razvojnega proračuna.
  "Za razvoj delujočega programa se navadno porabi polovica stroškov na testnih aktivnostih." [6]
  "...razhroščevanje, testiranje in verifikacijske aktivnosti z lahkoto dosežejo 50 do 75 procentov celotnega razvojnega stroška. " [7]

- Testiranje, ki se izvaja, je potrebno izboljšati.
  Ameriški nacionalni Inštitut standardov in tehnologij (National Institute of Standards and Technology) (2002) poroča, da so napake v programski opremi ocenjene na $59.5 milijard letno in bi se jih z izboljšanjem testiranja dalo za tretjino zmanjšati.

- Innovative Defence technologies (IDT), je podjetje specializirano na področju načrtovanja, razvoja ter implementacije rešitev avtomatskega testiranja programske opreme. Naredili so anketo z več kot tristotimi raznovrstnimi podjetji, ki se ukvarjajo z razvojem programske opreme. 40 % teh podjetij je imelo 1–300 zaposlenih, 32 % 301–10000 zaposlenih in 24 % več kot 100000 zaposlenih. Se pravi so podjetja dovolj raznolika, da nam dajo vpogled na celotno sliko.

    - Na vprašanje. koliko časa porabijo za testiranje programske opreme, je 18 % podjetij odgovorilo s 30–50 % in kar 49 % podjetij je porabilo 50–75 % časa za testiranje programske opreme.
    - 72 % vprašanih je reklo, da je avtomatizacija uporabna in da se ravnateljstvo strinja s tem, da bi jo morali izboljšati, vendar je niso implementirali zaradi pomanjkanja časa in izkušenj.

Poglejmo še raziskavo Quality and Assurance(QA) Institute, ki govori v prid avtomatskemu testiranju. Na QA Institute so naredili študijo, kjer so primerjali ročno in avtomatsko testiranje. Ta študija je bila narejena leta 1995 in je vključevala 1,750 testnih primerov in 700 napak [qaq95]. Rezultati so prikazani spodaj v tabeli 3.2. Vidimo, da so orodja v začetni fazi potrebovala nekaj investicij, v skupnem seštevku pa so rezultati pokazali impresivne prihranke v času, ki so ga porabili pri testu.

	Ročno testiranje	Avtomatsko testiranje	Izboljšanje v procentih
Razvoj testnega plana	32	40	-25%
Razvoj testnih primerov	262	117	55%
Izvedba testiranja	466	23	95%
Analiza rezultatov testa	117	58	50%
Sledenje napakam	117	23	80%
Kreiranje poročil	96	16	83%
Celotne ure	1090	277	75%

Tabela 3.2 Ročno testiranje proti avtomatskem testiranju [8]

Kot vidimo, pride po tej raziskavi do 75% izboljšanja porabljenega časa, če uporabimo avtomatsko testiranje. In to je več kot dober vzvod, da raziščemo in si pogledamo to področje bolj od blizu.

3.3 *Delitev avtomatskega testiranja*

Avtomatsko testiranje lahko delimo na dve večji področji:

- Testiranje modulov
- Avtomatski funkcionalni sistemski testi

Testi modulov so danes že utečena praksa in skoraj vedno avtomatizirani. Tovrstno testiranje je verjetno najpomembnejši preboj, ki se je zgodil na področju testiranja v zadnjem desetletju.

V pričujočem delu pa se bom osredotočila na še ne tako utečeno in manj znano avtomatsko funkcionalno sistemsko testiranje.

Avtomatizacija funkcionalnih testov navadno temelji na oponašanju človekove interakcije s testirano aplikacijo. Orodja za tovrstno avtomatizacijo so draga (v smislu nakupa in vpeljave), zato je treba skrbno pretehtati stroške in koristi (poglavje 4.1.1), izbrati pravo orodje, ki bo skladno z ostalimi razvojnimi orodji in se avtomatizacije lotiti na pravi način. Eden možnih načinov za vpeljavo avtomatizacije testiranja je strukturna metodologija ATLM (Automated Test Lifecycle Methodology), ki je opisana v poglavju 3.5.

3.4 *Tipi orodij v razvojnem ciklu testiranja programske opreme*

Razvojni cikel testiranja programske opreme lahko med vsako posamezno fazo podpremo z vrsto orodij. V tabeli 3.3 so predstavljeni tipi orodij, ki podpirajo razvojni cikel testiranja programske opreme.

Faza življenjskega cikla	Tip orodja	Opis orodja
Faza analize	Poslovno modeliranje	Posname definicije uporabniških zahtev, avtomatska hitra konstrukcija
	Sledenje napakam	Upravljanje z napakami, ki se pojavijo med življenjskim ciklom
	Generatorji dokumentacije	Avtomatizacija generiranja dokumentov
	Upravljanje zahtev	Upravlja in organizira zahteve, načrtovanje testnih procedur ter poročanje o napredku testiranja
Faza definicije zahtev	Verifikatorji zahtev	Verifikacija sintakse, semantike, možnosti testiranja
	Generatorji primerov uporabe	Kreiranje primerov uporabe
	Načrtovanje podatkovne baze	Razvoj sistemov odjemalec-strežnik druge generacije
Analiza in načrtovanja	Strukturni diagrami, diagrami podatkovnih tokov, diagrami zaporedja	Modeliranje podatkov in procesov
	Generatorji testnih procedur	Generirajo testne procedure iz zahtev, načrta ali podatkov in objektnih modelov
	Preverjanje sintakse/ razhroščevalniki	Preverjajo pravilnost sintakse in imajo sposobnosti razhroščevanja. Navadno jih dobimo s prevajalnikom kode.

Faza življenjskega cikla	Tip orodja	Opis orodja
Kodiranje	Puščanje pomnilnika ter napake med izvajanjem	Zaznavanje napak med izvajanjem ter zaznavanje puščanja pomnilnika
	Testiranje izvorne kode	Verifikacija vzdrževanja, prenosljivosti, kompleksnosti in skladnosti s standardi
	Statični in dinamični analizatorji	Prikažejo kvaliteto in strukturo kode
Metrike	Analiza pokritja kode	Identificira netestirano kodo ter podpira dinamično testiranje
	Meritve uporabnosti	Testiranje uporabnosti
Druga podporna orodja	Orodja za prevedbo podatkov	Prevedba podatkov iz enega v drug format
	Generatorji testnih podatkov	Generiranje testnih podatkov
	Primerjalniki datotek	Iskanje razlik med datotekami, ki bi morale biti po vsebini enake
	Simulatorji	Simulacija aplikacij
	Upravljanje testov	Upravljanje testov
	Testiranje omrežja	Spremlja, meri, testira in postavlja diagnoze obremenitev čez celotno omrežje
	Testiranje uporabniškega vmesnika (Posnemi-Predvajaj)	Posnamejo interakcije uporabnika s sistemom, tako da jih lahko ponovno predvajamo avtomatsko
	Testiranje obremenitev/kapacitet	Izvaja testiranje (maksimalnih) obremenitev / kapacitet
	Testiranje varnosti	Izvaja testiranje varnosti in preverja občutljivost na ravni aplikacije ali mreže.

Tabela 3.3 Orodja, ki podpirajo razvojni cikel testiranja programske opreme

3.5 *ATLM – metodologija življenjskega cikla avtomatskega testiranja*

ATLM metodologija je proces testiranja in metoda, kjer avtomatizacija testiranja teče vzporedno z razvojnim življenjskim ciklom programske opreme.

Prehod z ročnega testiranja na avtomatsko testiranje ni samo nakup orodij in uporaba le-teh, ampak moramo ob tem prehodu uvesti spremembe v celotnem življenjskem razvoju produkta. Da bi uspešno uvedli ta proces testiranja, je potrebno k svari pristopiti na strukturiran način.

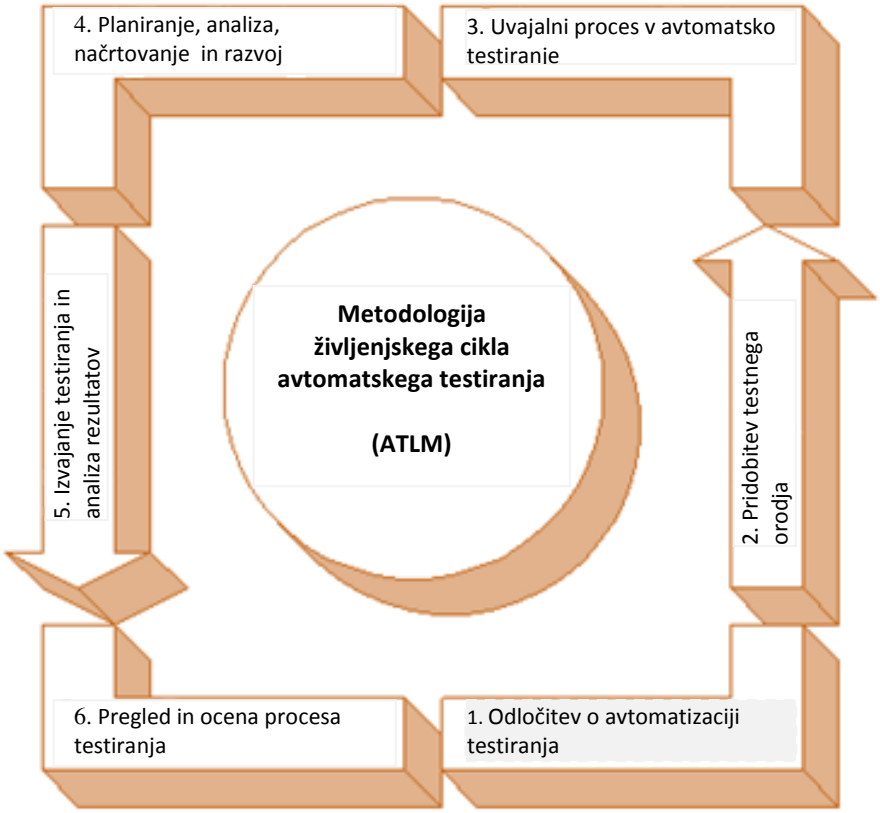
Večanje zmožnosti avtomatskega testiranja je predvsem posledica popularnosti iterativnega in inkrementalnega procesa razvoja programske opreme. Sicer ATLM lahko uporabimo ob različnih metodologijah razvoja programske opreme (zaporedni, agilni, SCRUM, itd.), je pa idealno, če se ATLM uporablja z metodologijo, ki se osredotoča na minimiziranje razvojnih urnikov ter pogoste dnevne izdaje. Cilj inkrementalnega in iterativnega razvoja je vključiti uporabnika in testno ekipo v zgodnje faze razvoja vsake verzije, tako da bo sistem odražal želje in potrebe uporabnika in se bodo naslovila najbolj tvegana vprašanja v zgodnjih verzijah.

V okolju nenehnih sprememb in dodatkov v programski opremi med verzijami, se testiranje programske opreme samo usmeri v iterativno naravo. Vsako novo verzijo spremlja nekaj novih testov ter spremenjenih oz. dodanih skript, prav tako kot je bilo spremenjenih nekaj modulov (kode). Z nenehnim dodajanjem in spreminjanjem kode, pa postane avtomatsko testiranje pomemben kontrolni mehanizem, ki zagotavlja pravilnost in stabilnost skozi vsako verzijo.

ATLM je večnivojski proces, ki si prizadeva vključevanje avtomatskih testnih orodij. Metodologija podpira med seboj povezane aktivnosti, ki so potrebne pri odločitvi izbire avtomatskega orodja in preverja ali naj avtomatsko orodje sploh uporabimo. Uvede nas v proces vpeljave in koristne uporabe takega orodja, pokrije razvoj testiranja in testnega načrtovanja, naslovi pa tudi izvajanje in vodenje testiranja. Metodologija podpira tudi razvoj ter upravljanje testnih podatkov in testnega okolja, se ukvarja z dokumentacijo ter testnimi poročili.

ATLM vključuje šest poglavitnih procesov oziroma komponent, ki jih lahko implementiramo kjerkoli v SDLC-ju [9]:

- Odločitev o avtomatizaciji testiranja
- Pridobitev testnega orodja
- Uvajalni proces v avtomatsko testiranje
- Planiranje, analiza, načrtovanje in razvoj
- Izvajanje testiranja in analiza rezultatov
- Pregled in ocena procesa testiranja



Slika 3.1 ATLM

- ATLM ni navezan na nobeno orodje
- ATLM se lahko uporablja neodvisno od razvojnega procesa
- ATLM se lahko izvaja paralelno z razvojnim ciklom programske opreme ali pa ga pričnemo izvajati v katerikoli fazi

3.5.1 Odločitev o avtomatizaciji testiranja

V tej prvi fazi je poudarjen pomen pridobitve popolne podpore ravnateljstva za vpeljavo avtomatizacije testiranja. Potrebno je razjasniti morebitna napačna pričakovanja o avtomatizaciji, prikazati resnične koristi in pasti uvedbe avtomatizacije, predstaviti orodja, ki so na trgu, ter ugotoviti donosnost naložbe.

Treba je zagotoviti, da bo na voljo dovolj sredstev, ne samo za nakup orodij, temveč med drugim tudi za plačilo dodatnih ljudi na projektu. Poleg testerjev so za uspeh avtomatizacije testiranja ključni izkušeni programerji. Upoštevati je treba, da bodo na začetku stroški avtomatizacije večji od koristi, možni so tudi negativni vplivi na doseganje rokov zaradi kratkoročno zmanjšane učinkovitosti testerjev. Testna skupina oziroma njeni odgovorni morajo dovolj zgodaj ugotoviti, ali se ravnateljstvo projekta zaveda vseh predvidenih stroškov avtomatizacije in ali jih je pripravljeno pokriti. Če temu ni tako, lahko skušajo vodstvu bolj nazorno predstaviti potencialne koristi s podrobno analizo stroškov in koristi. Če tudi to ne zaleže, potem je treba prilagoditi projekt avtomatizacije sredstvom, ki so na voljo, ali pa ga opustiti.

Najboljše prakse:

- Razumeti problem, ki ga hočemo rešiti – ali je to problem varnosti, uporabnosti, itd.
- Poznati orodja, ki so nam na voljo.
- Razjasniti pričakovanja.
- Razumeti resnične prednosti.
- Vedeti, kaj bomo avtomatizirali.

Nevarnosti, na katere moramo paziti:

- Potreba po daljšem času za spoznavanje z orodjem.
- Poznavanje orodja in programerska znanja so potrebna.
- Cena izobrazbe na orodju.
- Ne smemo vzeti testiranja kot stransko aktivnost - to je razvoj programske opreme.

3.5.2 Pridobitev testnega orodja

Ko imamo zagotovljeno podporo ravnateljstva, se je treba odločiti, katero orodje je primerno za naš primer in katera podporna orodja bomo morebiti potrebovali. Navadno je razvojno okolje že postavljeno in utečeno, zato se je s testnimi orodji treba temu prilagajati. Določiti je treba merila izbora orodja oz. množico kriterijev, ki bodo osnova za pripravo odločitvenega modela. Kriteriji so npr. cena, združljivost, uporabnost na različnih projektih, težavnost integracije z ostalimi orodji, itd.

Na podlagi določenih meril je potrebno izbrati orodja in jih dobiti na preizkus, kjer lahko preverimo njihovo delovanje na pilotnih projektih.

Najboljše prakse:

- Določiti ali je morda potrebno razvijati v hiši.
- Poznati okolje sistema, v katerem razvijamo.
- Izvesti ocenitev orodij.

Pri izboru orodij se ni dobro omejiti le na orodja proizvajalcev, saj je na voljo nekaj odličnih odprtokodnih testnih orodij, ki jih lahko uporabimo kot samostojna testna orodja ali kot izboljšavo testnih orodij proizvajalcev.

Nevarnosti, na katere moramo paziti:

- Avtomatska orodja za testiranje uporabniških vmesnikov imajo svoje omejitve.
- Avtomatizacija skozi uporabniški vmesnik je lahko preveč zamudna, lahko tudi nestabilna, ko se ukvarjamo z tisočimi testnimi primeri.
- Efektivna avtomatizacija testa se ukvarja s poglavitno funkcionalnostjo sistema.
- Veliko testnih orodij ne pokriva vseh vprašanj kodiranja in veliko jih kreira lažne pozitivne/ negativne rezultate.

3.5.3 Uvajalni proces v avtomatsko testiranje

Po izboru in preizkusu orodij se v tretji fazi začne proces vpeljave avtomatskega testiranja k obstoječemu ali novemu razvojnemu procesu. Udeleženci morajo dojeti, da gre dejansko za proces vpeljave in ne za dogodek. Izbrana orodja je treba prilagoditi procesu testiranja in ne obratno.

Analiza procesa testiranja
Tretja faza se prične z analizo obstoječega procesa testiranja v organizaciji. Navadno se testiranje tako ali drugače že izvaja. Če proces testiranja ni dobro definiran (ad hoc testiranje), je prvi korak, ki ga je treba storiti, definiranje procesa. Testni proces mora biti dokumentiran v takšni obliki, da ga lahko predstavimo vsem vpletenim. Če proces ni dokumentiran, ni ponovljiv, niti razumljiv vsem udeležencem. To posledično pomeni, da se proces ne bo izvajal. Poleg tega nedokumentiranega procesa ne moremo niti meriti niti izboljševati in še manj avtomatizirati. Definirati je treba strategijo, namen in cilje testiranja. Izbrati in dokumentirati je treba primerne tehnike testiranja in najboljšo prakso (bodisi že obstoječo v organizaciji bodisi iz zunanjih virov).

Preučitev testnih orodij
Preveriti je potrebno, ali so izbrana orodja dejansko ustrezna in ali jih lahko uporabimo v ciljnem projektu. Pri tem moramo upoštevati dejanske zahteve testiranja v ciljnem projektu, ljudi in opremo, ki so na voljo, ciljno okolje in značilnosti testirane aplikacije. Treba je uskladiti urnik projekta in preveriti, ali je na voljo dovolj časa za vpeljavo orodij. Treba je določiti vloge in odgovornosti na projektu glede na usposobljenost in znanje kadra v skupini za testiranje.

Najboljše prakse:
- Izberemo pilotni projekt ali pilotno opravilo in demonstriramo avtomatsko testiranje programske opreme.
- Izobrazimo razvijalce o testnem orodju in zahtevah testiranja, in kako bodo spremembe na kodi vplivale na implementacijo avtomatizacije testiranja (razvijalci lahko v podporo avtomatskemu testiranju uporabljajo beleženje za boljše razumevanje med testom, saj nam med testom pridejo prav sporočila o napakah).
- Preučimo spremembe procesa, ki so potrebne za uvedbo avtomatskega testiranja.
- Vedno se je treba učiti iz novih lekcij in jih dokumentirati.

3.5.4 Planiranje, analiza, načrtovanje in razvoj

Testno **planiranje** zajema pregled vseh aktivnosti, potrebnih za izvajanje testiranja in verifikacijo, da bodo procesi, metodologije, tehnike, osebje, orodja in oprema organizirani in uporabljeni na učinkovit način. Rezultat testnega planiranja je testni plan, v katerem je zbrana vsa potrebna dokumentacija o testiranju. Testni plan se prične oblikovati že v predhodnih fazah in se sproti dopolnjuje skozi vse nadaljnje faze projekta. Pripravi se testno okolje.

Analiza in načrtovanje testiranja
Pred fazo načrtovanja je treba izvesti analizo zahtev testiranja in jih dokumentirati. Zahteve testiranja izhajajo iz zahtev testiranega sistema. Pridobimo jih lahko z različnimi tehnikami, bodisi iz primerov uporabe bodisi iz drugih modelov iz analize testiranega sistema.

Po definiranju zahtev testiranja se prične načrtovanje testnih primerov. Za vsak testni primer je treba med drugim določiti, ali gre za ročni ali avtomatski test. Treba je dokončno definirati in dokumentirati standarde kodiranja ter pristop k izdelavi programske kode testov, ki bo v največji meri omogočal ponovno uporabnost, robustnost in čim bolj enostavno vzdrževanje kode.

V testnem planu so med drugim zajeti:
- vloge in odgovornosti udeležencev projekta
- doseg testnega procesa, glede na omejitve v času, zaposlenih in ostalih potrebnih virih

- strojna, programska oprema, omrežne zahteve za testno okolje, način upravljanja s konfiguracijo
- uporabljene tehnike testiranja po principu črne in bele skrinjice, pristop k načrtovanju testov, standardi kodiranja in poimenovanja testnih procedur
- časovni plan za razvoj in izvajanje testnih primerov
- zahteve po testnih podatkih in način njihove pridobitve oziroma generiranja
- razne povezovalne matrike, kot npr. matrika povezav med testnimi primeri in

Razvoj testnih primerov
Za analizo in načrtovanjem testiranja sledi kodiranje testnih primerov. Osnova za to aktivnost so priprave v prejšnjih fazah. Predhodno morajo biti definirani standardi kodiranja, navadno prilagojeni orodjem v uporabi. Tako večina orodij za avtomatsko testiranje prek uporabniškega vmesnika podpira neke vrste skriptni programski jezik, npr. različico Visual Basic-a.

Najboljše prakse:
- Oceniti kaj bomo avtomatizirali.
- Nočemo za vsako ceno avtomatizirati vsega.
- Postavimo si prioritete.
- Efektivno avtomatsko testiranje se izvaja tudi na nivoju sive škatlice, ne le na nivoju črne škatlice.

Kako in kdaj se lotimo avtomatizacije testiranja:
- Vse ni primerno za avtomatizacijo.
- Ne avtomatizirati vsega naenkrat, postaviti je treba prioritete.
- Ne sme se podvajati koda aplikacije, ki jo testiramo.
- Potencial za ponovno uporabo.
- Ponavljajoča se opravila.
- Podatkovno usmerjena opravila.
- Regresijski testi.
- Testiranje, kjer bi bilo ročno testiranje skorajda neizvedljivo: testiranje varnosti, puščanje pomnilnika, pokrivanje kode, testiranje obremenitev, maksimalnih obremenitev itd.

Nevarnosti, na katere moramo paziti:
- Se ne spuščamo v avtomatizacijo testov, brez da bi imeli pred seboj dokumentacijo ali načrtovanih testov.
- Ker avtomatsko testiranje omogoča, da naredimo lažje na tisoče testnih scenarijev, ni nujno, da je pokritje testov večje.
- Avtomatizirati je potrebno analizo testov, kjer lahko porabimo veliko časa (recimo, če imamo avtomatiziranih 1000 testnih primerov, lahko na njihovi analizi zapravimo preveč časa).
- Vprašati se je treba: ali test da dodano vrednost?
- Ali bomo s tem testom izboljšali pokritje testov?

3.5.5 Izvajanje testiranja in analiza rezultatov

V fazi izvajanja testiranja se izvedejo predhodno pripravljeni testni primeri. Hkrati se dobljeni rezultati testiranja zbirajo in analizirajo. Tu je mišljeno izvajanje testiranja na vseh ravneh, od testiranja modulov do sistemskega in prevzemnega testiranja. Po izvedenem testiranju se beležijo odkrite napake in naknadno izvedeni popravki, ustrezno se ažurira tudi testna

dokumentacija. Za potrebe analize rezultatov lahko uporabimo različne metrike. Z njimi beležimo kazalce o napredovanju testiranja in stopnji pokrivanja testov. Možnih metrik je veliko, ker pa vsaka analiza zahteva čas, moramo smiselno izbrati tiste, ki prinesejo največ informacij. Bolj podrobno se bomo metrikam posvetili še v poglavju 4.

Metrike, ki jim je dobro slediti:

- Primerjamo število testnih procedur, ki so se izvršile/uspešno izvršile, proti številu vseh testnih procedur.
- Število iteracij testnih procedur.
- Število napak, po prioritetah – metrika napak.
- Število zahtev, ki so bile testirane – sledenje.
- ROI
- Kolikšen procent testnih primerov se da avtomatizirati.
- Če imamo procent testnih primerov, ki se jih da avtomatizirati, koliko jih je dejansko avtomatiziranih?
- Kolikšno pokritje dosegamo z avtomatskimi testi?
- Vpliv avtomatizacije na kvaliteto – zanesljivost.

Najboljše prakse:

- Kontroliranje testnega okolja.
- Izoliranje različnih razlogov problemov.
- Ponovno testiranje po popravkih.
- Efektivno sledenje napakam.

3.5.6 Pregled in ocena procesa testiranja

Ta faza se izvaja skozi celoten življenjski cikel avtomatizacije testiranja s ciljem stalnih izboljšav procesa. Med procesom in predvsem po izvajanju testnih primerov je treba analizirati izbrane metrike ter rezultate.

Pri tem ATLM predlaga naslednje **Najboljše prakse:**

- Po izvajanju testiranja mora testna skupina spremljati učinkovitost testnega programa in preučiti, kje so možne in potrebne spremembe, da bi lahko pri naslednjem projektu proces izboljšali.
- Rezultati metrik povedo tudi, ali je testirana aplikacija zrela za produkcijo (npr. trend napak se zmanjšuje proti ničli). Končno oceno o tem mora podati skupina za testiranje.
- Testna skupina mora sprejeti stalni iterativni proces učenja kot del svoje kulture, tako na napakah kot na uspehih. Zbrane pozitivne in negativne izkušnje, izvedene izboljšave in popravljalne aktivnosti je treba dokumentirati.
- Rezultati metrik se dokumentirajo. Vsa dokumentacija skozi celotni življenjski cikel naj bo shranjena v lahko dostopnem repozitoriju.
- Treba je izračunati analizo povračila naložbe v avtomatizacijo, za kar moramo že med procesom testiranja zbirati ustrezne podatke.

ATLM poudarja ravnateljsko plat (angl. *managers view*) testiranja in manj izvedbeno. Glede na pristop je to sicer "težka" metodologija, ki daje precej poudarka dokumentiranju oziroma formalizaciji procesa. Kljub temu je to edina celovita metodologija za vpeljavo avtomatizacije testiranja. Obravnava pristop k avtomatizaciji na vseh ravneh od testiranja modulov do sistemskega testiranja, čeprav največ poudarka daje avtomatizaciji funkcionalnega oz. sistemskega testiranja.

3.6 *Tehnike pisanja skript*

S tem poglavjem odpiramo teme, ki obravnavajo tehnično plat avtomatskega testiranja. Opisane so tehnike pisanja skript, avtomatska primerjava rezultatov (poglavje 3.7), nasveti pri arhitekturi testvera (poglavje 3.8), zakaj in kako naj se lotimo avtomatizacije pred in po procesnih dejavnosti (poglavje 3.9), na koncu pa so predstavljeni kriteriji, na katere moramo biti pozorni pri izbiri orodja za avtomatsko testiranje (poglavje 3.10).

Začeli bomo s pregledom tehnik avtomatskega testiranja, ki so primerne za to, da bi dosegli avtomatizacijo testiranja, ki bo učinkovita in jo bomo lahko vzdrževali. Ocenili bomo testne tehnike, kot so pisanje skript ter avtomatska primerjava rezultatov.

Tehnike za pisanje skript so zelo podobne tehnikam programiranja. Te skripte morajo biti napisane tako, da jih bomo lahko vzdrževali skozi razvoj produkta.

Skripte vsebujejo podatke in navodila za testno orodje. Vključujejo primerjavo informacij, od kod naj se podatki berejo in kam naj se zapišejo, kontrolne informacije kot so if stavki in zanke. Bolj enostavna kot je skripta, bolj enostavno jo bo napisati, vendar navadno tudi toliko bolj težko vzdrževati. Da bi zmanjšali vzdrževalni del, je treba več vložiti pri kreiranju skripte. Dobro napisana skripta bo dobro strukturirana, podprta s komentarji, opravljala bo eno nalogo, bo razumljiva in jo bomo z veliko verjetnostjo ponovno uporabili.

Predstavljenih je pet skriptnih tehnik. Vsaka ima svoje prednosti in slabosti in vsako lahko uporabimo v primerni situaciji.

3.6.1 Linearna

Linearna skripta je tista, ki jo ustvarimo kot posnetek ročnega testa (posnemi-predvajaj). S to tehniko lahko začnemo hitro in ne potrebujemo nobenega programerskega znanja, saj nam orodje samo posname točno zaporedje dogodkov, ki smo jih naredili med snemanjem. Ta skripta je uporabna za ponavljajoče akcije in demonstracije. Ni pa primerna za dolgoročno in enostavno vzdrževanje. Te skripte se izkažejo za neefektivne, drage in ko jih hočemo spremeniti, zelo občutljive za manjše spremembe v programski opremi. Zato se tudi hitro zlomijo ob nepričakovanih dogodkih, ki se zgodijo med testom. Vhodni in izhodni podatki so trdo-žično vpeljani v skripto.

3.6.2 Strukturirana

Strukturirana tehnika uporablja kontrolne strukture. Če (ang. if) stavki povečajo robustnost, s tem ko omogočajo izvajanje različnih odzivov na dogodke. Z zankami si lahko pomagamo, kadar potrebujemo, da se določen dogodek ponavlja. S klicanjem procedur znotraj procedur dosežemo modularnost. Glavna prednost strukturiranega pisanja skript je robustnost, saj lahko na specifičnih delih programa ujamemo dogodke, ki bi lahko povzročili padec testa. S tem je skripto težje napisati, vendar se s takim načinom pisanja poveča moč skripte.

Linerarna skripta	Strukturirana
Izbira Opcije 'Datoteka/Odpri	Izbira Opcije 'Datoteka/Odpri
Fokus Na'Odpri'	Fokus Na'Odpri'
	če Sporočilo = Ali shranim obstoječi dokument?
	Klik Leve Miške 'Ne'
Napiši 'Test1.rtf'	Napiši 'Test1.rtf'
Klik Leve Miške 'Odpri'	Klik Leve Miške 'Odpri'

Primer 1: Ta del skripte vsebuje preverjanje stavka 'Ali shranim obstoječi dokument?' s če stavkom. Če se sporočilo pojavi, bomo nanj odgovorili s klikom na da gumb, drugače se bo skripta izvajala naprej.

3.6.3 Deljena

Deljene skripte uporabimo večkrat, se pravi, da so del več kot enega testnega primera. To omogoča, da so osnovne akcije zapisane enkrat na enem mestu. S tem je vzdrževanje veliko lažje obvladljivo. Deljene skripte naj bodo robustne, saj se bodo uporabljale na več koncih. Da bi imeli korist od deljenih skript, moramo biti dobro organizirani.

Linearna skripta	Deljena skripta	Uporaba deljene skripte
	BeležnicaOdpri(IMEDATOTEKE)	
Klik Leve Miške 'Beležnica'	Klik Leve Miške 'Beležnica'	
Fokus Na'Beležnica'	Fokus Na'Beležnica'	
Izbira Opcije'Datoteka/Odpri'	Izbira Opcije'Datoteka/Odpri'	
Fokus Na'Odpri'	Fokus Na'Odpri'	
Napiši 'Test1.rtf'	Napiši IMEDATOTEKE	Kliči BeležnicaOdpri
Klik Leve Miške 'Odpri'	Klik Leve Miške 'Odpri'	(Test1.rtf)

Primer 2: Vsak naslednji testni primer bomo implementirali lažje, saj lahko deljene skripte ponovno uporabimo.

3.6.4 Podatkovno usmerjena

Podatkovno usmerjene skripte shranjujejo vhodne in pričakovane izhodne podatke testov v datoteke, ki jih nato lahko preberemo iz splošnih kontrolnih skript. Ker moramo za dodajanje novih testov spreminjati le te datoteke, je dodajanje testov olajšano. Bolj razvita podatkovno usmerjena struktura uporablja stolpce v datoteki, ki se nanašajo na logične entitete v programski opremi, ki jo testiramo. S to tehniko lahko pridobimo lažje dodajanje testov, več testov, ki imajo podobno naravo je lahko izvedenih in skripte ni potrebno spreminjati, ko dodajamo nove teste. Tudi testerji niso obremenjeni s tehnično platjo in se lahko posvetijo testiranju. Slabša stran te tehnike pa je vzpostavitev tega stanja, ki zahteva dosti časa, za implementacijo so potrebni programerji in seveda mora biti dobro urejena. Skripte in tabele morajo biti konsistentne.

Podatkovno usmerjena skripta

```
Odpri Datoteko 'TestniPodatki1'
Za vsak zapis v TestniPodatki1
   Preberi VhodnaDatoteka
   Preberi Ime
   Preberi IzhodnaDatoteka
   Kliči BeležnicaOdpri (VhodnaDatoteka)
   Fokus Na   'Beležnica'
   Izbira Opcije  'Izberi/Dodaj element'
   Fokus Na  'Dodaj element'
   Napiši Ime
   Klik LeveMiške 'Vredu'
   Fokus Na  'Beležnica'
   Kliči BeležnicaZapri (IzhodnaDatoteka)
Konec
```

Primer 3: S podatkovno usmerjeno skripto enostavno spreminjamo testne podatke.

3.6.5 Skripta s ključnimi besedami (ang. Keyword- driven testing)

To tehniko lahko označimo tudi kot bolj sofisticirano podatkovno usmerjeno tehniko. Z njo prenesemo znanje iz skripte v podatkovno datoteko.. Podatkovnim datotekam v tem pristopu rečemo kar testne datoteke, saj opisujejo testne primere. Kontrolna skripta prebere vsako ključno besedo in pokliče pripadajočo podporno skripto. Kontrolna skripta ni več omejena na določeno aplikacijo ali sistem. Glavna razlika med skripto s ključnimi besedami in skriptami, ki so bile opisane pred njo, sta dva temeljna pristopa k implementaciji testnih primerov: predpisovalen ter opisen.

- Skripta s ključnimi besedami nam omogoča opisen pristop pri implementaciji avtomatskih testnih primerov, kjer moramo podati samo opise testnih primerov, kot bi jih dali članom testne ekipe. Ta pristop določa, kaj testni primer dela, ne pa, na kakšen način to dela. Kontrolna skripta prebere, kaj dela testni primer in pokliče pripadajoče podporne skripte.

- Druge skripte, ki sem jih opisala prej, uporabljajo predpisovalen pristop k implementaciji testnih primerov. Tako je znanje, kako bo skripta opravila nalogo, vključena v skripto samo.

Prednosti skripte, ki uporablja ključne besede
Prednosti, ki jih prinese ta pristop so velike. Ko so osnovne podporne skripte aplikacije napisane, je lahko implementiranih veliko več testov, brez da bi povečali število skript.
S tem zmanjšamo vzdrževanje skripte, zmanjšamo potreben čas za implementacijo ter omogočimo, da lahko te teste pišejo tudi testerji, ki nimajo programerskih izkušenj. S to tehniko lahko dosežemo, neodvisnost od orodja (ter platforme). Se pravi, da bomo morali ob zamenjavi testnega okolja zamenjati le podporne skripte. S tem ko omogočimo testni ekipi, da si izbere orodja po svoji meri, lahko pričakujemo hitreje in bolj varno narejene teste.

Slabosti skripte, ki uporablja ključne besede
Kompleksnost podatkovne datoteke se znatno poveča, če je ne obravnavamo z določeno mero pozornosti. Tako se lahko tudi vse prednosti, ki smo jih opisali, izničijo. Če orodje, ki ga uporabljamo, ne omogoča branja podatkov iz drugih virov, je potrebno razviti pristop, ki to omogoča. Včasih se nam splača razviti svoj pristop, tudi če to orodje omogoča, saj bomo tako neodvisni od kateregakoli orodja.

Kontrolna skripta	Testna datoteka	Podporna skripta
Za vsak TEST_ID	BeležnicaOdpri Test1	BeležnicaOdpri
OdpriDatoteko TEST_ID	DodajElement 20 20 20	…
Za vsak zapis v testni datoteki	ShranoKot Test1	…
Preberi ključno besedo		ShraniKot
Pokliči ključno besedo	BeležnicaOdpri Test2	…
konec	DodajElement 40 50 60	…
ZapriDatoteko TEST_ID	ShraniKot Test2	DodajElement
Konec		…
		…

Primer 4: Opisen pristop pri implementaciji avtomatskih testnih primerov, kjer moramo podati samo opise testnih primerov, kot bi jih dali članom testne ekipe.

3.7 *Avtomatska primerjava rezultatov*

Verifikacija je proces preverjanja, kjer določimo ali smo dobili prave rezultate. Med seboj je potrebno primerjati rezultate, ki jih dobimo, ter rezultate, ki jih pričakujemo.

Ob planiranju avtomatskih primerjav je potrebno določiti, katere in kolikšno količino informacij bomo primerjali med seboj, saj ta vpliva na sposobnost odkrivanja napak, stroške implementacije in stroške vzdrževanja.

Primerjavo lahko označimo za najbolj avtomatsko opravilo pri testiranju programske opreme in navadno se ta del tudi najbolj obrestuje, če ga avtomatiziramo. Primerjava list, številk in kakršnihkoli podatkov ljudem ne gre dobro od rok. Zelo lahko pride do napake, ker se nam zdi tako delo dolgočasno (ponavljajoče in podrobno). Računalniku pa so te naloge pisane na kožo.

Pri avtomatskem testiranju bomo dobili veliko rezultatov, ki jih bo potrebno primerjati. In brez avtomatske primerjave ne bomo imeli celostnega avtomatskega testiranja, čeprav bomo imeli avtomatsko izvajanje.

V primerjavo so lahko vključeni rezultati na zaslonu, podatki v podatkovni bazi, podatki, ki so bili poslani na tiskalnik, elektronska sporočila, itd.

Primerjalnik nam ne more povedati ali je test uspel ali ne, poišče pa razlike med dvema setoma podatkov. Pri avtomatizaciji testov so to navadno rezultati testa in pričakovanih rezultatov. Podatki primerjave se lahko prikažejo na zaslonu ali pa so shranjeni v podatkovni bazi, v obliki tekstovne datoteke, itd.

3.7.1 Dinamična primerjava

Dinamična primerjava je aktualna med izvajanjem testa. Testna orodja so v večini primerov opremljena za dinamično primerjavo. Ta se izvaja na zelo podoben način, kot ga uporablja človek. Rezultati na zaslonu se preverjajo med testom, tako da se lahko preverijo tudi, če jih kasneje neka akcija v testnem primeru prepiše. Tudi nekateri drugi rezultati, ki se ne pokažejo na zaslonu, se lahko preverijo, kot na primer atributi objektov.

3.7.2 Post-izvedbena primerjava

Post-izvedbena primerjava se izvede po zaključenem testnem primeru. Uporablja se za primerjavo rezultatov, ki se niso med izvajanjem testnega primera prikazovale na zaslonu, kot na primer ustvarjene datoteke, spremembe baze, itd. Ta primerjava nam omogoča bolj selektivno primerjanje. Lahko si izbiramo zaporedje primerjave, odvisnost nadaljevanja primerjave glede na prejšnje rezultate, itd.

3.7.3 Enostavna in kompleksna primerjava

Enostavna primerjava išče popolno pokrivanje med dejanskimi in pričakovanimi rezultati. Vsakršna razlika bo izstopala. Tovrstna primerjava se veliko uporablja, saj jo zlahka definiramo. Velikokrat pa se zgodi, da bi lahko zanemarili nekatere razlike. Kompleksna primerjava nam omogoča primerjavo med pričakovanimi in dejanskimi rezultati z znanimi razlikami (npr. datum).

3.7.4 Občutljivost testa

Če primerjamo vse informacije, ki so nam dane, kot na primer primerjava celega zaslona po vsakem koraku, bo vsaka razlika majhna ali velika, povzročila neskladje in test bo padel. Tak test lahko označimo kot občutljivi test. Če pa primerjamo le en del rezultatov (npr. zadnje sporočilo, ki je prikazano), pa se bo neskladje pokazalo le v primeru, da se pokaže napaka v tem delu. Temu pravimo robustni testni primer.

Na občutljivost testa pa ne vpliva samo količina informacij, ki jih preverjamo, ampak tudi način, s katerim jih preverjamo. Recimo primerjave, ki upoštevajo le vsebino sporočil, so bolj robustne kot tiste, ki preverjajo tudi njihov položaj na zaslonu.

Seveda pa moramo med robustnimi in občutljivimi testi najti vmesno pot. Če primerjamo premalo ali neprimerne rezultate, bo vrednost testnega primera zmanjšana, če ne izgubljena. Če primerjamo preveč informacij, bodo stroški implementacije in vzdrževanja prekosili koristi, ki jih pridobimo.

Tehniki avtomatskega testiranja, ki sta bili opisani v tem poglavju: tehnika pisanja skript ter avtomatska primerjava rezultatov, sta najbolj pogosti tehniki, ki nam olajšajo avtomatizacijo testiranja.

Nadaljujemo pa z arhitekturo testvera, saj nam tudi najboljše tehnike pisanja skript in primerjave rezultatov ne pomenijo veliko, če jih ne bomo organizirali v neko smiselno strukturo, iz katere bomo do njih dostopali na sistematičen in enostaven način.

3.8 *Arhitektura testvera*

Testver (ang. testware) je izraz, s katerim opisujemo vse izdelke, ki jih potrebujemo za testiranje: dokumentacija, skripte, podatki, pričakovani rezultati, dejanski rezultati, poročila o primerjavi, sumarna poročila, itd.

Arhitektura testvera je razporeditev teh dokumentov: kje so shranjeni, kje se uporabljajo, kako so grupirani ter kako se spreminjajo in vzdržujejo.

	Testver	
Testni material	Rezultati testiranja	
Vhodni podatki	Produkti	Stranski produkti
Skripte	Dejanski rezultati	Dnevnik, Status, Poročilo o razlikah
Podatki		
Dokumentacija		
Pričakovani rezultati		

Tabela 3.4 Razporeditev dokumentov testvera mora biti sistematična.

3.8.1 Ponovna uporaba

Da bi izkoristili prednosti avtomatskega testiranja, moramo upoštevati prednosti, ki nam jih prinese ponovna uporaba skript in podatkov. Glavni razlog za ponovno uporabo je izločiti podvojenost in s tem pospešiti implementacijo novih testov ter zmanjšati stroške vzdrževanja. Da bi dosegli ponovno uporabo skript, pa moramo vložiti delo v skripte in podatke, da bodo zopet uporabni ter jih shraniti na tako mesto, da jih bomo oziroma jih bodo tudi drugi lahko hitro našli. V primeru, da jih ne najdejo hitro oziroma da so slabo napisane, se lahko zgodi, da obupamo nad iskanjem in napišemo novo skripto, s čimer pridemo zopet do podvajanja. Zato je potrebno imeti knjižnico skript, v kateri bodo shranjene skripte, kjer hitro najdemo želeno skripto.

3.8.2 Verzioniranje

Prava vrednost avtomatskega nabora testov naj bi se pokazala, ko se testira nova verzija programske opreme. Preden se testira, so dodani popravki v testne primere. Ko so vsi popravki v teste vneseni, lahko poženemo test na novi verziji. Toda kaj se zgodi s staro verzijo? Staro verzijo testov je potrebno ohraniti, saj lahko pride do primerov popravkov prejšnjih verzij.

Torej, kam naj jih shranimo in kako dostopamo do njih? Lahko bi za vsako verzijo naredili kopijo celotnega testvera in bi bili sigurni, katera verzija testvera pripada kateri verziji programske opreme. Vendar pa tak pristop pride v poštev le pri majhnem številu avtomatskih testov in tudi s tem pristopom kmalu pridemo do težav. Bolj sofisticiran pristop k upravljanju več verzij bi bil tak, da bi imeli več kopij tistih datotek, ki so se spremenile. Tako sicer malo otežimo rekonstrukcijo, katere kopije spadajo kam, vendar pa tudi zmanjšamo velikansko podvojenost podatkov.

Predstavimo primer, kjer se uporabljajo različne kombinacije skript in podatkov, v treh različnih izdajah programske opreme.

	Izdaja 1	Izdaja 2	Izdaja3
Y skripta	I2.0	I2.1	I3.0
X skripta	I1.0	I1.0	I1.0
Podatkovne datoteke	I5.1	I5.1	I5.2

Tabela 3.5 Verzioniranje: Prikazana je možna situacija, v kateri so uporabljene različne kombinacije skript in podatkovnih datotek, v vsaki od treh izdaj programske opreme.

3.8.3 Platformna in okoljska neodvisnost

Programsko opremo je potrebno velikokrat testirati v različnih okoljih oziroma na različnih platformah. Idealno bi bilo, če bi bili testi platformno neodvisni. V praksi pa so nekatere razlike neizbežne. Če potrebujemo različne teste za različne platforme, naj testver arhitektura zagotovi potrebne kopije platformno odvisnih datotek, da se testver ne bi ponavljal.

3.8.4 Pristop k arhitekturi testvera

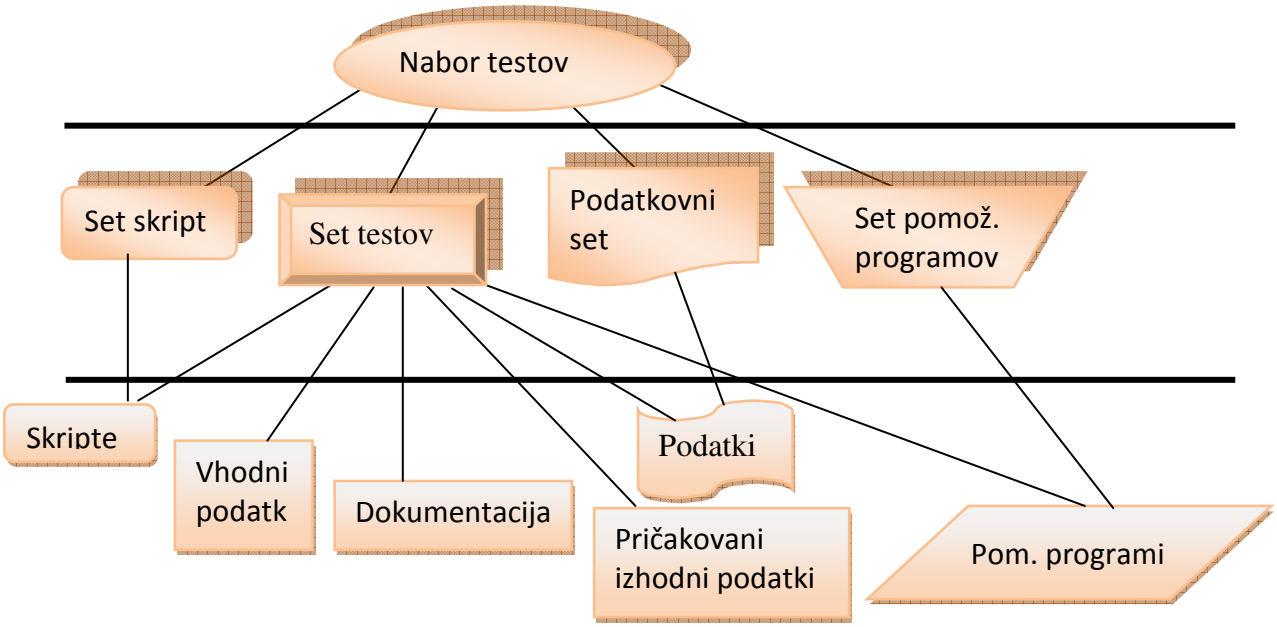
Arhitektura testvera je ureditev elementov, ki so potrebni za testiranje in avtomatsko testiranje. Vprašanja glede arhitekture testvera je potrebno nasloviti na samem začetku, saj se bomo v množici podatkov drugače kaj hitro izgubili. Pri implementaciji je priporočljivo uporabljati konsistenten pristop, saj ta pomeni boljši pregled ter večjo ponovno uporabo.

Seti testov(ang. Test Sets) – logično sestavljeni seti testnega materiala, ki vsebujejo enega ali več testnih primerov. Osnovni koncept testnega seta je, da vsebuje skripte, podatke, pričakovane rezultate in dokumentacijo.

Nabor testov (ang. TestSuite) – več setov testov skupaj tvori nabor testov. Sete testov združujemo v nabor po navadi takrat, ko imamo pred seboj nek zaključen cilj testiranja (npr. testiranje določene napake, regresijski test, itd.).

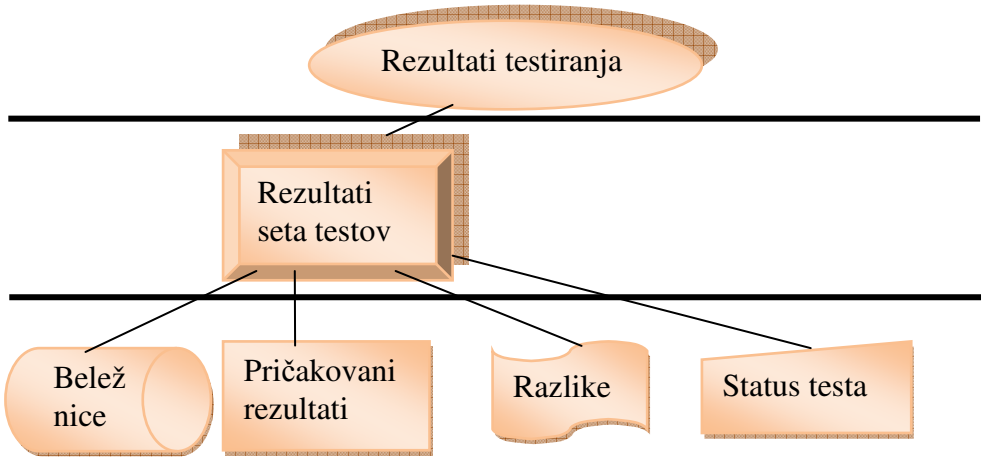
Testni podatki ali pa skripte, ki so razdeljene med testne primere, so shranjeni v ločene strukture, ki jim pravimo *seti podatkov* in *seti skript*. Podobno so tudi pomožni programi (gonilniki, posebni primerjalniki) shranjeni v sete pomožnih programov.

Glavne verzije teh setov pa so shranjene v testver knjižnici. Testver knjižnica je repozitorij glavnih verzij vseh testver setov. Do knjižnice naj bo omogočen enostaven dostop in vse spremembe, do katerih pride, naj bodo zabeležene.



Slika 3.2 Predstavitev razmerij v testveru

Testni Rezultati so shranjeni v ločeni strukturi, ki je paralelna strukturi nabora testov. Testni rezultati so generirani vsakič, ko poženemo test.



Slika 3.3 Hierarhična razmerja rezultatov v testveru

Testver arhitekturo implementiramo kot strukturo direktorijev. Vsak testver set ima svoj direktorij in ta vsebuje poddirektorij za vsak tip, ki obstaja v testver setu. Direktorij nabora testov odraža namen, ki ga ima nabor testov in vsebuje poddirektorije vseh pomembnih tesvare setov. Rezultati se hranijo v paralelnih direktorijih. Smiselno poimenovanje teh direktorijev je pomembno.

3.9 *Avtomatizacija pred- in poprocesnih dejavnosti*

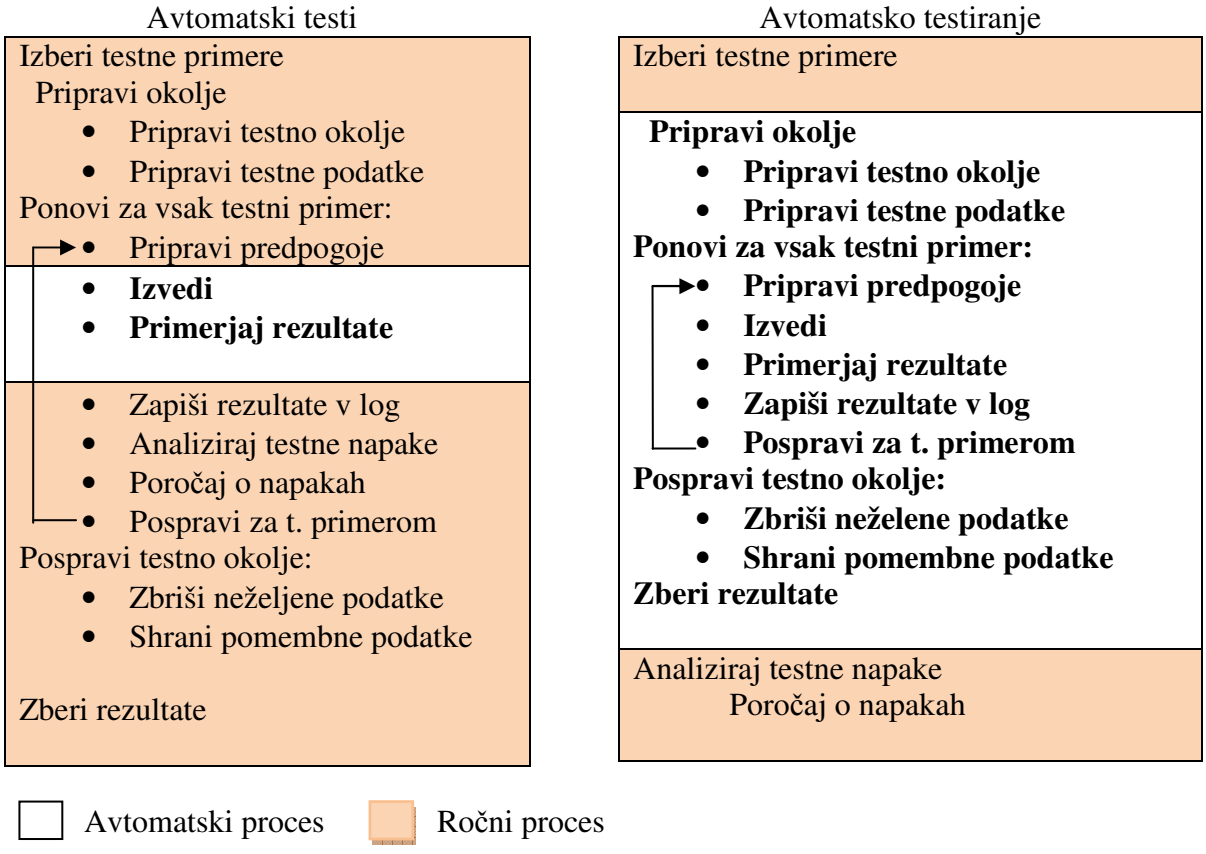
Katere so pred- in poprocesne dejavnosti?

Za večino primerov morajo biti izpolnjeni nekateri predpogoji, preden se lahko izvajanje testa prične. Ti morajo biti definirani pred vsakim testnim primerom. Nekatere predpogoje je potrebno nastaviti le enkrat, spet druge je potrebno ob vsakem zagonu testa ponovno nastaviti. Vsakemu opravilu, ki je povezano z vzpostavitvijo in ohranjanjem testnih predpogojev pravimo **predprocesna dejavnost**.

Takoj po končanem testu pa se ukvarjamo z raznimi produkti testa (rezultati, zapisniki …), ki so lahko razmetani na dolgo in široko. S temi datotekami bo potrebno nekaj narediti, če ne drugega, jih pospraviti. Nekateri od teh testov vsebujejo podatke, ki jih lahko takoj zbrišemo (poročilo o različnosti, ki pravi, da razlik ni), druge moramo ohraniti (datoteka z dejanskimi rezultati, ki so različni od pričakovanih). Datoteke, ki jih ohranimo, moramo morda prestaviti na neko skupno lokacijo, da bomo lažje naredili analizo, ali pa enostavno, da jo obvarujemo pred tem, da bi jo naslednji testi spremenili ali izbrisali. Tem primerom pravimo **poprocesne dejavnosti**.

Pred- in poprocesne dejavnosti kar kličejo po tem, da jih avtomatiziramo. Opravljanje teh del ročno je namreč časovno zamudno in hitro zmotljivo.

Za avtomatsko testiranje moramo imeti tudi avtomatizirane pred- in poprocesne dejavnosti, ki obdajajo izvajanje testov, drugače imamo avtomatizirano le izvajanje.



Slika 3.4 Razlika med avtomatskimi testi in avtomatskim testiranjem. Slika prikazuje zaporedje nalog, ki so potrebne pri avtomatskih testih in avtomatskem testiranju.

Predprocesne naloge

Kreiranje: Ustvarjanje predhodnih zahtev, kot so postavitev podatkovne baze, polnjenje baze s podatki potrebnimi za test, odstranitev neželenih podatkov iz baze, neželenih datotek.

Preverjanje: Včasih ni mogoče avtomatizirati vseh predprocesnih nalog, lahko pa jih preverimo. Primer: pregled ali določene datoteke obstajajo, da določene datoteke ne obstajajo, ali deluje lokalna mreža, ali je vstavljen disk, na katerega lahko zapisujemo podatke, itd.

Reorganizacija: Se ukvarja s kopiranjem ali prestavljanjem datotek. Primer: Ko naš test vključuje spremembo neke datoteke, jo skopiramo na svoj delovni direktorij, tako da ne bo original poškodovan.

Prevedba: Včasih je dokumente bolje shranjevati v drugačni obliki kot bo primerna za testiranje. Primer: neka velika datoteka je shranjena v stisnjeni obliki, ki jo za primere testa razširimo.

Po procesne naloge

Brisanje: Brisanje datotek, ki so nastale med testom, brisanje podatkov iz baze, itd.

Preverjanje: Podobno kot pri predprocesnih nalogah: Preverjanje ali neka datoteka obstaja / ne obstaja, itd.

Reorganizacija: Podoben primer kot brisanje, vendar je tukaj poudarjeno predvsem kopiranje in premikanje datotek. Primer: kopiranje vseh testnih rezultatov na eno mesto, da bi lažje analizirali padec testa.

Prevedba: Kadar je format izhodnih podatkov neprimeren za primerjavo ali morda analizo podatkov, je potrebno izhodne podatke prevesti na primeren format.

Pred- in poprocesne naloge med različnimi stadiji

Pred- in poprocesne naloge se največkrat izvedejo vsakič pred in takoj za vsakim testnim primerom. Nekatere pred- in poprocese pa lahko štejemo kot del več testnih primerov skupaj. V opisani rešitvi arhitekture testvera, prideta v poštev še dve fazi, kjer se uporabljajo pred- in poprocesne dejavnosti. Ti dve fazi sta nabor testov ter set testov.

Za primer vzemimo uporabo baze med testiranjem. Recimo, da upravljamo s podatki uporabnikov v različnih testnih primerih. Kreirati bo potrebno bazo in jo napolniti s podatki uporabnikov, preden se lahko testi izvedejo. Kreiranje podatkovne baze je enkraten dogodek, ki je predprocesna dejavnost nabora testov. Restavracija baze pa je predprocesna dejavnost vsakega testnega primera posebej in se ponovi vsakič, ko se zažene testni primer.

3.10 *Ocenitev kriterijev orodja za avtomatsko testiranje*

Proces izbire orodja za avtomatsko testiranje (poglavje 3.5 – ATLM) oceni in izbere orodje, ki je primerno za določeno organizacijo. Še preden se lotimo izbire orodja, pa moramo definirati naše zahteve za testno orodje. Te zahteve so lahko povezane s problemi, ki jih želimo rešiti ali pa so del omejitev (cena, prilagajanje razvojnemu okolju …). Vsekakor pa je potrebno določiti merila izbora orodja oz. množico kriterijev, ki bodo vplivali na našo odločitev. V tem poglavju bomo predstavili nekaj kriterijev, ki so ključni pri izbiri ustreznega testnega orodja. Te kriterije je smiselno oceniti z neko lestvico (npr. 1–5), jim dodeliti uteži glede na pomembnost in jih predstaviti v matriki ali morda v odločitvenem modelu (dexi, hiview …). [10, 11]

1. Posnemi in predvajaj

   Ta kategorija nam pove, kako enostavno je snemanje in predvajanje testa.
   Vprašamo se:
   - Ali orodje podpira nizko raven snemanja (povleci miško, natančna lokacija zaslona) ali prepozna objekte?
   - Ali se posname pravilno, potem pa se pri predvajanju ne izvede pravilno?
   - Kako preprosto je branje in zapisovanje v skripto?

   Posnemi in predvajaj nam hitro lahko prikaže delovanje orodja, vendar pa je sčasoma vse manjši del avtomatizacije procesa, saj je običajno bolj robustno uporabljati funkcije za neposredno testiranje objektov, podatkovnih baz, itd.

2. Spletno testiranje

   Spletne aplikacije so del vsakdanjega življenja, zato je potrebno zagotoviti tudi preizkušanje funkcionalnosti spletnih aplikacij. Potrebno je pogledati podporo za HTML tabele, okvire, različne platforme za brskalnike, povezave, itd.
   Če se manjše težave ne upoštevajo, se spletno preizkušanje lahko hitro zaplete z različnih vidikov. Tukaj je nekaj primerov:

   - Ali obstajajo funkcije, ki povejo, kdaj se je dokončalo nalaganje strani?
   - Ali lahko preizkusno orodje počaka, dokler se slika ne prikaže?
   - Ali se da preizkusiti veljavnost povezav?
   - Ali obstajajo naprave, ki bodo načrtno iskale objekte določenega tipa na spletni strani ali našle določen objekt?
   - Ali lahko izpišemo podatke iz spletne strani same?

   Pri preizkušanju navadnih aplikacij so cilji navadno dobro opredeljeni, vemo, kateri operacijski sistem bomo imeli in tako naprej, ampak na spletu je stvar precej drugačna. Ista oseba je lahko povezana v ZDA ali Afriki, stranke so lahko invalidne osebe, ki uporabljajo različne brskalnike, tudi ločljivosti zaslona so lahko različne. Lahko govorijo različne jezike, imajo hitro ali počasno povezavo, se povežejo preko MAC, Linux ali Windows operacijskih sistemov, itd.

3. Testiranje podatkovne baze

   Večina aplikacij bo zagotovila možnost ohranitve podatkov zunaj njih samih. To se običajno doseže tako, da so podatki shranjeni v podatkovni bazi. Zaradi številnih podatkovnih baz, ki so nam na voljo (Oracle, DB2, SQLServer, Sybase, itd.) vse podpira univerzalni poizvedbeni jezik znan kot SQL in protokol za komunikacijo za

komuniciranje s temi podatkovnimi bazami ODBC (JDBC se lahko uporablja v java okolju).

- Pogledati je potrebno podporo orodja za SQL, ODBC in kako hranijo in vračajo podatke (npr. ali je to niz, spremenljivka, itd).
- Kako zna to orodje manipulirati vrnjene podatke?
- Ali lahko kliče shranjene procedure in priskrbi zahtevane vhodne spremenljivke?
- Kakšen razpon funkcij ponuja pri testiranju?

4. Podatkovno usmerjene funkcije

Kot smo že omenili, imajo v večini primerov aplikacije tudi pripadajoče baze podatkov. Iz te baze podatkov naj bi lahko med drugim ustvarili vhodne podatke za aplikacijo.

- Potrebno je pogledati, ali nam orodje omogoča, da določimo vrsto podatkov, ki jih želimo?
- Lahko samodejno ustvari podatke?
- Ali lahko uporabimo datoteke, preglednice, itd za kreiranje, izpis podatkov?
- Ali lahko dostopamo do teh podatkov po nekem naključnem postopku?
- Je ta dostop do podatkov resnično naključen?

Dodana korist te funkcije se lahko pokaže pri migraciji podatkov ali pa na primer pri nadgradnjah aplikacij. Te funkcije so tudi zelo pomembne, ko se preselite iz snemalno/predvajalne faze v podatkovno usmerjeno do celovitih testnih ogrodij. Podatkovno usmerjeni preizkusi so preizkusi, ki nadomeščajo trdo kodirana imena, naslove, številke, itd s spremenljivkami, ki so shranjene v nekem zunanjem viru, običajno datotekah, preglednicah ali podatkovni bazi. Testna ogrodja so navadno končni cilj v razvoju avtomatskih testnih orodij. Ogrodja priskrbijo vmesnik za vse aplikacije, ki so pod testom, podajo listo funkcij in s tem omogočajo tudi neizkušeni testni ekipi, da izvaja teste samo z izvajanjem/oskrbovanjem testnega ogrodja z znanimi ukazi/spremenljivkami. Kot smo že omenili, za postavitev ogrodja potrebujemo veliko časa, strokovnih virov in denar.

5. Mapiranje objektov

Če razvojna ekipa uporablja standardne objekte, se nam ni potrebno ukvarjati z mapiranjem objektov. Večina aplikacij je implementirana z uporabo standardnih objektov, ki so podprti s strani testnega orodja, seveda pa bo nekaj objektov tudi nestandardnih. Večina nestandardnih objektov se bo obnašala, podobno kot se obnašajo standardni objekti.

Naštejmo nekaj standardnih objektov, ki jih srečamo pri vsakdanjih aplikacijah:

- gumbi, na katere pritiskamo
- radijski gumbi, gumb s katerim izberemo eno od možnosti
- polja za urejanje
- kombinirana polja, ki združujejo besedilo in seznam

Če se srečamo z nestandardnim objektom, ki se obnaša kot eden izmed standardnih, ali jih bomo lahko mapirali (povedali testnemu orodju, da se nestandardni objekt obnaša tako kot standardni) Ali orodje podpira vse standardne metode gradnikov?

6. Testiranje slik

   Upajmo, da to ni pomemben del testiranja, vendar pa ga je potrebno občasno uporabiti za testiranje slik. Tudi kadar je aplikacija narisala kontrole, bomo morda potrebovali to funkcionalnost.

   - Ali orodje zagotavlja optično prepoznavanje znakov?
   - Ali lahko primerja eno sliko proti drugi?
   - Koliko časa traja primerjava?
   - Kako dolgo traja, če primerjava ne uspe?
   - Ali orodje omogoča, da se nekatera območja maskirajo med primerjavo?

7. Obnova po napakah v testu

   To je lahko eno izmed najtežjih področij pri avtomatizaciji, vendar če je avtomatizirana, zagotavlja temelje za izdelavo resnično zanesljivih naborov testov.

   - Denimo, da se aplikacija poruši med testiranjem, kaj lahko naredim?
   - Če funkcija ne sprejme pravilne informacije, kako lahko to obravnavam?
   - Če dobim sporočilo o napaki, kako naj se spoprimem s tem?
   - Če dostopam do spletne strani in dobim opozorilo, kaj naj naredim?
   - Ne morem vzpostaviti povezave s podatkovno bazo, kako naj preskočim te teste?
   - Kako preprosto lahko vgradim te funkcionalnosti v svojo kodo?

   Testno orodje bi moralo biti opremljeno za ravnanje z zgoraj zastavljenimi vprašanji. Gledati moramo na to koliko napak lahko orodje zajame, vrste napak, kako okreva po napakah, itd.

8. Mapiranje objektnih imen

   Testno orodje posname naše interakcije na objektih. Orodje mora zagotoviti sposobnost identifikacije vsakega objekta, s katerim je v interakciji. Za sklicevanje na objekte je dobro imeti mehanizem, ki omogoča enostavno posodobitev, če se aplikacija, ki jo testiramo spremeni. Najbolj zaželeno je, da orodje nudi centralni repozitorij za shranjevanje teh primerkov objektov. Lažje je spremeniti sklicevanje na enem mestu, namesto da bi šli skozi vsako od skript in ga tam nadomestili . Pri orodjih, ki ne podpirajo centralne sheme repozitorija, si olajšamo delo tako, da imamo reference oken in objektov na enem mestu (preko spremenljivk) in te spremenljivke potem uporabljamo v skripti.

9. Orodje za identiteto objekta

   Ko postanemo bolj usposobljeni za avtomatizacijo testiranja bo eden od glavnih sredstev za identifikacijo objektov preko "orodja za identiteto". Kot nekakšen vohun, ki se osredotoča na notranjost objekta, kot je objektno ime, ID in podobno.

   Orodje naj ve nekaj podrobnosti o lastnostih objekta, zlasti tistih, povezanih z enolično identifikacijo objekta ali okna.

10. Razteglјivi jezik

V primeru, da standardni testni jezik ne podpira neke funkcionalnosti, ali lahko naredim DLL, ali pa razširim jezik na nek način, da bo ta funkcionalnost podprta?
To je običajno napredna tema in ne naletimo nanjo dokler usposobljeni tester ne uporablja orodja najmanj 6–12 mesecev. Vendar, ko naletimo na ta problem, naj bo orodje podprto z razteglјivim jezikom.

11. Podpora okolja

Koliko okolij podpira orodje? Ali podpira najnovejšo Java izdajo, .Net, Oracle, itd. Nenazadnje je to najpomembnejši del avtomatizacije: podpora okolja. Če orodje ne podpira našega okolja/aplikacije, potem smo v težavah in v večini primerov bomo morali preklopiti na ročno testiranje aplikacije (več shelfwarea[2]).

12. Integracija

Kako dobro se da orodje integrirati z drugimi orodji? To vprašanje postaja vedno bolj pomembno. Ali lahko neposredno prikaže napake in ali ga lahko povežemo s testnimi informacijami zapisanimi v log-ih? Ali se ga da povezati s produkti kot so Word, Excel, orodji za upravljanje zahtev?

Pri upravljanju velikih testnih projektov (pri večjih podjetjih (npr. bankah)), govorimo pri preoblikovanju sistema o deset tisočih testnih primerov. Kako bomo to upravljali? Ali bomo naše orodje lahko povezali z orodjem za upravljanje testov? Če je integracija slabo podprta, pride do ločenih sistemov in do podvajanja podatkov.

13. Stroški

Ko na koncu potegnemo črto, je strošek eden najmanj pomembnih kriterijev. To pa zato, ker so si vsa orodja podobna v ceni. Cena se običajno giblje okoli par tisoč dolarjev (odvisno od količine, paketov, itd). Pravo vprašanje je, katero orodje bo opravilo svoje delo in ne katero orodje bo najcenejše. Običajno je potrebno računati tudi na dodatne stroške vzdrževanja, ki se gibljejo med 10 in 20 odstotki. Vendar pa je vse stvar ponudbe. Veliko dobrih orodij pa je tudi zastonj.

14. Preprostost uporabe

To poglavje je zelo subjektivno. Preprostost uporabe se s spoznavanjem orodja in izkušnjami spreminja. Ko testna ekipa postane bolj izkušena, se pojavijo vprašanja v zvezi z razširitvijo, skriptami, vzdrževanjem, podatkovno usmerjenimi testi, itd.

15. Podpora uporabnikom testnega orodja

Veliko gradiva se najde na internetu. Na forumih lahko najdemo večino odgovorov na naša vprašanja tako, da nam ni potrebno vedno klicati podpore. Pomaga tudi, če je na njihovi spletni strani veliko uporabnih uporabniških in prodajnih prispevkov. Upoštevati moramo tudi različne druge kriterije, kot so razpoložljivost kvalificiranih virov, spletnih virov, kvaliteta odgovorov iz pomoči uporabnikom, odzivnost podpore in podobno.

---

[2] Kupljena programska oprema, ki zaradi neuporabnosti ali pa pomanjkanja zainteresiranosti uporabe, konča na policah.

16. Testiranje objektov

- Kakšne so možnosti preizkušanja lastnosti objektov?
- Lahko preveri več lastnosti objektov naenkrat?
- Ali lahko preverja več objektov hkrati?
- Ali so lastnosti objekta pri zajemu stanja aplikacije nastavljive?

Pazljivost pri pregledu tovrstnih zmogljivosti orodij je potrebna, saj mora biti to večji del preverjanja, kar se tiče avtomatizacije procesa.

4 Metrike

4.1 *Zakaj merimo testiranje in avtomatizacijo testiranja?*

Testiranje in avtomatizacijo testiranja je potrebno tudi meriti. Preko meritev bomo lahko spremljali in nadzorovali avtomatizacijo testiranja in bomo lahko ocenili, kateri ukrepi so potrebni za optimizacijo avtomatskega testiranja. Pomembno je, da si izberemo merljive, ciljno usmerjene in uporabne meritve.

4.1.1 Donosnost naložbe

Preko izračuna donosnosti naložbe (angl. *Return of investment* – v nadaljevanju ROI) merimo učinkovitost naložbe. Dobimo jo tako, da vrednost koristi naložbe delimo z vrednostjo, ki smo jo porabili za naložbo [12].

Donosnost naložbe računamo iz dveh razlogov:

- preden investiramo v določeno stvar, je potrebna ocenitev, koliko naj bi z uvedbo določene spremembe pridobili (napoved)
- ter po tem, ko smo to stvar vpeljali, koliko smo dejansko pridobili (ocenitev)

$$\text{ROI} = \frac{\text{letni(čisti) dobiček kot posledica naložbe}}{\text{vrednost naložbe}} \text{x}100$$

V primeru, da imamo že vpeljano ročno testiranje, koristi avtomatizacije ne računamo v absolutnem znesku, ampak v primerjavi z ročnim testiranjem.

Naslednji obrazec podaja izračun povračila naložbe na podlagi primerjav med ročnim in avtomatskim testiranjem [12]:

$$\text{ROI}_{\text{avtomatizacije}} (\text{v času t}) = \frac{\Delta\, Koristi\ avtomatizacije\ napram\ ročnemu\ testiranju}{\Delta\, Stroški\ avtomatizacije\ napram\ ročnemu\ testiranju}$$

4.1.2 Izbire, primerjava alternativ, spremljanje napredka

Med testiranjem in avtomatizacijo testiranja moramo sprejeti veliko različnih odločitev. Katero tehniko naj uporabimo, kolikšna naj bo primerjava, da določimo, ali je bil test pravilen oziroma nepravilen, ali je ravnovesje med občutljivo/robustno primerjavo pristranski, itd.

Brez meritev bomo težko določili in podprli naše odločitve (ali so bile dobre ali slabe). Včasih se nam zdi, da je neka sprememba, ki smo jo uvedli, dobra, vendar brez objektivnih meritev bomo težko zagotovo vedeli ali nam je sprememba pomagala ali pa nam mogoče celo škoduje.

4.1.3 Zgodnje opozorilo na napake in napovedovanje

S tem ko spremljamo situacijo testiranja in avtomatskega testiranja, imamo večjo možnost, da bomo videli bližajoče se probleme, preden bi ti naredili preveliko škodo. Vzemimo za primer, da se procent najdenih napak zmanjša za neko večjo vrednost. V tem primeru je možno, da je

naša nova tehnika manj efektivna kot tista, ki smo jo uporabljali prej. Ali pa morda sedaj delo opravljajo manj izkušeni ljudje in testi niso tako dobro napisani.
S tem ko vemo, koliko napora je potrebno vložiti za vzdrževanje testiranja pri različnih spremembah, koliko avtomatskih testov bo vključenih v tako vzdrževanje, koliko sprememb bo v programski opremi v naslednjem letu, bomo lažje tudi planirali, koliko bomo morali vložiti v vzdrževanje testiranja in izvajanje testov.

4.2 *Kaj lahko merimo (primeri)?*

Programska oprema ima kar nekaj atributov, ki jih dokaj lahko merimo:
- velikost - število vrstic kode
- število funkcij
- število odločitev (if, while, case, itd.)
- cena razvoja (čas, trud vložen v pisanje programske opreme)
- število napak najdenih med testiranjem in uporabo
- število razvijalcev

Tudi testiranje ima kar nekaj atributov, ki jih lahko merimo na dokaj preprost način:
- število testov v testnem naboru
- število planiranih, izvedenih ter uspešnih testov
- cena (čas ali trud), ki ga porabimo na aktivnostih testiranja
- število napak najdenih med testiranjem in uporabo
- pokritost kode

Atributi, ki jih navadno merimo pri avtomatskem testiranju
- število skript avtomatizacije
- število avtomatiziranih testov
- čas, ki je potreben za izvajanje avtomatskih testov
- čas ali trud, ki je potreben za vzdrževanje testov
- število padlih testov, ki jih je povzročila napaka

4.2.1 Uporabne meritve

Uporabna meritev je tista, ki podpira efektivno analizo in odločitveno podporo in jo lahko pridobimo relativno lahko. Seveda pa je odločitev o tem, kaj bomo merili, odvisna tudi od ciljev, h katerim stremimo.

4.3 *Cilji testiranja in avtomatizacije testiranja*

Testiranje ima lahko veliko ciljev, ki določajo, kako bo potekal proces testiranja. Če želimo na primer odkriti čim več napak, bo testiranje usmerjeno v dele kode, ki so zelo kompleksne ali pa je bilo tam že prej veliko napak. Če je naš cilj, da uporabnikom zagotovimo stabilen program, ki bo deloval v vsakdanjih situacijah, se bomo posvetili delom kode, ki obravnavajo glavne poti skozi program med vsakdanjo uporabo. Različne organizacije imajo različne cilje, ti pa se čez čas lahko tudi spreminjajo. Testiranje pa mora biti prilagojeno primernim ciljem v vsakem času.

Nekaj ciljev pri avtomatizaciji testiranja:
- konsistentno ponavljajoče testiranje
- izvajanje testov brez nadzora

- iskanje regresijskih napak
- bolj pogosto izvajanje testov
- boljša kvaliteta programske opreme
- bolj podrobno testiranje
- povečanje zaupljivosti v programsko opremo
- večja pokritost testiranja programske opreme
- povečanje uporabniške zaupljivosti v programsko opremo
- merjenje zmogljivosti
- zmanjšanje cene testiranja
- testiranje na različnih OS
- pohitritev procesa testiranja
- izboljšanje morale med testerji
- testiranje na različnih bazah

Dober cilj je tisti, ki ga lahko dosežemo in predstavlja dobro investicijo. Če je zaželen cilj uvedba testov, pri katerih ne potrebujemo interakcije testerjev med izvajanem testov, bomo za dosego tega cilja potrebovali kar nekaj časa in truda. S tem ko usmerimo energijo v tako 'izpopolnjene' teste, pa lahko naredimo manj na kakšnem drugem področju, kot je razvoj skript, ki bi lahko bil bolj koristen. Zato si moramo izbrati prave cilje v okviru njihovih omejitev.

4.4 *Atributi avtomatizacije testiranja*

Katere atribute avtomatizacije naj merimo? Predstavila bom nekaj primerov merljivih atributov. Seveda pa je zopet treba vedeti, kakšni so naši cilji in kateri atributi (morda tukaj niti niso našteti) se nanašajo na te cilje. Ko se odločamo o atributih, ki jih bomo merili, je priporočljivo, da si izberemo tri ali štiri, ki nam bodo dali najbolj koristne informacije, jih nekaj časa opazujemo in se poskusimo nekaj naučiti iz njih.

Režim je dobro **vzdrževan**, če so testi v koraku z razvojem programske opreme.
Možne meritve vzdrževanja:

- Povprečen čas, ki je potreben za posodobitev testa.
- Pogostost sprememb programske opreme.

Učinkovitost je povezana s ceno. Možne meritve učinkovitosti:

- Čas, ki je potreben za opravljanje določenih opravil testiranja.
- Kolikokrat je nek testni element uporabljen.
- Procent testnih skript, ki so uporabljene vsaj x-krat.

Zanesljivost povezujemo s pravilnimi in ponovljivimi rezultati. Možne meritve zanesljivosti:

- Procent testov, ki padejo zaradi napak v testu samem.
- Število dodatnih iteracij oz. testnih ciklov zaradi napak v testih.
- Število napačnih negativnih/pozitivnih rezultatov, kjer se test spozna za napačnega/pravilnega, čeprav je pravilen/napačen.

Fleksibilnost povezujemo z obsegom uporabe različnih setov oz. podsetov testov. Možne meritve fleksibilnosti:

- Čas, potreben za testiranje nujnega popravka na stari verziji.
- Čas, potreben za določanje seta testnih primerov za določen namen, na primer vseh testov, ki so padli zadnjikrat.
- Potreben čas, ki je potreben za vzpostavitev testnega primera, ki je bil arhiviran.

Uporabnost je lahko različna glede na tip uporabnika. V primeru tehnično podkovanega uporabnika je režim drugačen kot v primeru tehnično nepodkovanega uporabnika. Zato moramo pri uporabnosti upoštevati tip uporabnika. Možne meritve uporabnosti:

- Čas, ki je potreben za dodajanje novih testnih primerov podobnega tipa v že obstoječi režim.
- Čas, ki ga potrebujemo za uvajanje uporabnikov v razvoj produktivne in sigurne avtomatizacije testiranja.
- Zadovoljstvo uporabnikov (kako enostavna se jim zdi uporaba).

Robustnost avtomatskega testnega režima predstavlja, kako uporabni so avtomatski testi v nestabilni ali hitro spreminjajoči programski opremi. Možne meritve robustnosti:

- Število testov, ki bo padlo zaradi ene napake.
- Pogostost izpada testa zaradi nepričakovanih dogodkov.
- Čas iskanja vzroka nepričakovanega dogodka, ki je povzročil izpad testa.

Prenosljivost avtomatskega testnega režima je povezan z zmožnostjo izvajanja v različnih okoljih. Možne meritve prenosljivosti:

- Potreben čas oz. trud za vzpostavitev avtomatskega testiranja v drugem okolju (npr. druga baza) ali na drugi platformi strojne opreme.
- Potreben čas oz. trud za izvajanje avtomatskih testov z drugim testnim orodjem.
- Število različnih okolij, v katerih se lahko izvajajo avtomatski testi.

4.5 *Kateri režim je najboljši?*

Možnih metrik je veliko, ker pa vsaka analiza zahteva čas, moramo smiselno izbrati tiste, ki prinesejo največ informacij. Tako je odvisno od primera do primera, katere metrike bomo spremljali.

Nekaj metrik avtomatskega testiranja, ki so se izkazale za najbolj uporabne:

1. Odstotek, ki ga lahko avtomatiziramo (Kolikšen procent testnih primerov se da avtomatizirati?).
2. Napredovanje (Če imamo procent testnih primerov, ki se jih da avtomatizirati, koliko jih je dejansko avtomatiziranih?)
3. Procent pokritja z avtomatskimi test.
4. Število napak, ki jih je našel avtomatski test in jih ročni ne bi našel.
5. Donosnost investicije v avtomatsko testiranje programske opreme.
6. Vpliv avtomatskega testiranja na kvaliteto/zanesljivost.
7. Število napak, po prioritetah – metrika napak.
8. itd.

5 Implementacija v organizaciji

5.1 *Ozadje*

Time&Space (T&S) sistem je produkt podjetja Špica International d.o.o. T&S je celovit sistem, sestavljen iz množice programov, ki podpirajo pristopno kontrolo in registracijo delovnega časa. V diplomskem delu bom opisala projekt, ki je potekal na aplikaciji Setup.exe. Ta aplikacija skrbi za namestitev sistema T&S. Pri izbiri aplikacije za pilotni projekt sem gledala na možnosti avtomatizacije. Setup.exe je v tem pogledu ugodna za avtomatizacijo, saj je potrebno narediti veliko prehodov skozi podobne akcije (ponavljajoča), uporabniški vmesnik se skorajda ne spreminja, njeno pravilno delovanje je zelo visokega pomena in še bi lahko naštevala.

5.2 *ATLM*

Pri vpeljavi avtomatskega testiranja se bom opirala na metodologijo ATLM, ki sem jo opisala v poglavju 3.5. ATLM metodologija nam ustreza predvsem zaradi narave razvoja programske opreme, ki je iterativno-inkrementalna.

5.2.1 Odločitev o avtomatizaciji testiranja

V začetni fazi smo pridobili podporo ravnateljstva, ki je odobrilo pilotni projekt. **Naš cilj je vpeljati regresijsko testiranje aplikacije Setup.exe na podlagi že napisanega dimnega testa**, ki se še izvaja ročno. Želimo doseči čim bolj popolno in neodvisno avtomatizacijo testiranja aplikacije Setup.

5.2.2 Pridobitev testnega orodja

Na trgu je ogromno število orodij. Nekatera so specializirana za avtomatsko testiranje spletnih aplikacij, druga za testiranje zmogljivosti, itd. Različna orodja tudi podpirajo različna okolja, platforme in tako je potrebno iz množice orodij, ki so na voljo, izbrati takega, ki bo ustrezal našim zahtevam. Naredili smo raziskavo možnih orodij, ki so na trgu za avtomatizacijo testiranja ter naredili ožji izbor.

Pri testiranju smo si pomagali z orodjem VMware, katerega že uporabljamo pri ročnem testiranju. Ta nam z virtualizacijo omogoča več delovnih okolij na eni delovni postaji. Pripravili smo si več virtualnih računalnikov, ki ustrezajo različnim testnim okoljem (MSSQL, Oracle, brez strežnika). Dobra stran dela z orodjem VMware je možnost kreiranja posnetka trenutnega stanja, v katerem se virtualni računalnik nahaja. Ta lastnost nam pride prav v primeru določanja napak v specifičnih okoliščinah, katere lahko posnamemo in se tudi vrnemo na to specifično mesto, kjer napako tudi zlahka ponovimo.

5.2.3 Uvajalni proces v avtomatsko testiranje

Analiza procesa testiranja

Stanje pred avtomatizacijo testov

Pristop k razvoju programske opreme je iterativno-inkrementalni in se nagiba proti agilnim metodologijam. Pred začetkom tega projekta se je del testiranja že izvajal avtomatsko. To so testi modulov. Ni pa še implementiranih avtomatskih funkcionalnih sistemskih testov. Zato se je celotno sistemsko testiranje izvajalo ročno. Pri iterativnem razvoju programske opreme se končni različici izdelka približujemo postopoma, skozi več iteracij. Pri testiranju T&S sistema tako uporabljamo različne metode skozi vsako iteracijo razvoja

- Statično testiranje
  - V začetnih fazah pregledamo dokumentacijo.
  - V fazi kodiranja na manjših sestankih pregledujemo kodo

- Strukturna analiza ali metoda bele skrinjice
  - Implementacija DUnit testov (testi modulov), s katerimi pokrijemo pomembnejša področja.

- Funkcionalna analiza ali metoda črne skrinjice
  - Ob popravkih kode se izvedejo funkcionalni testi posameznih področij ter pri novih izdajah verzij, kjer se izvede dimni test(ang. Smoke test).

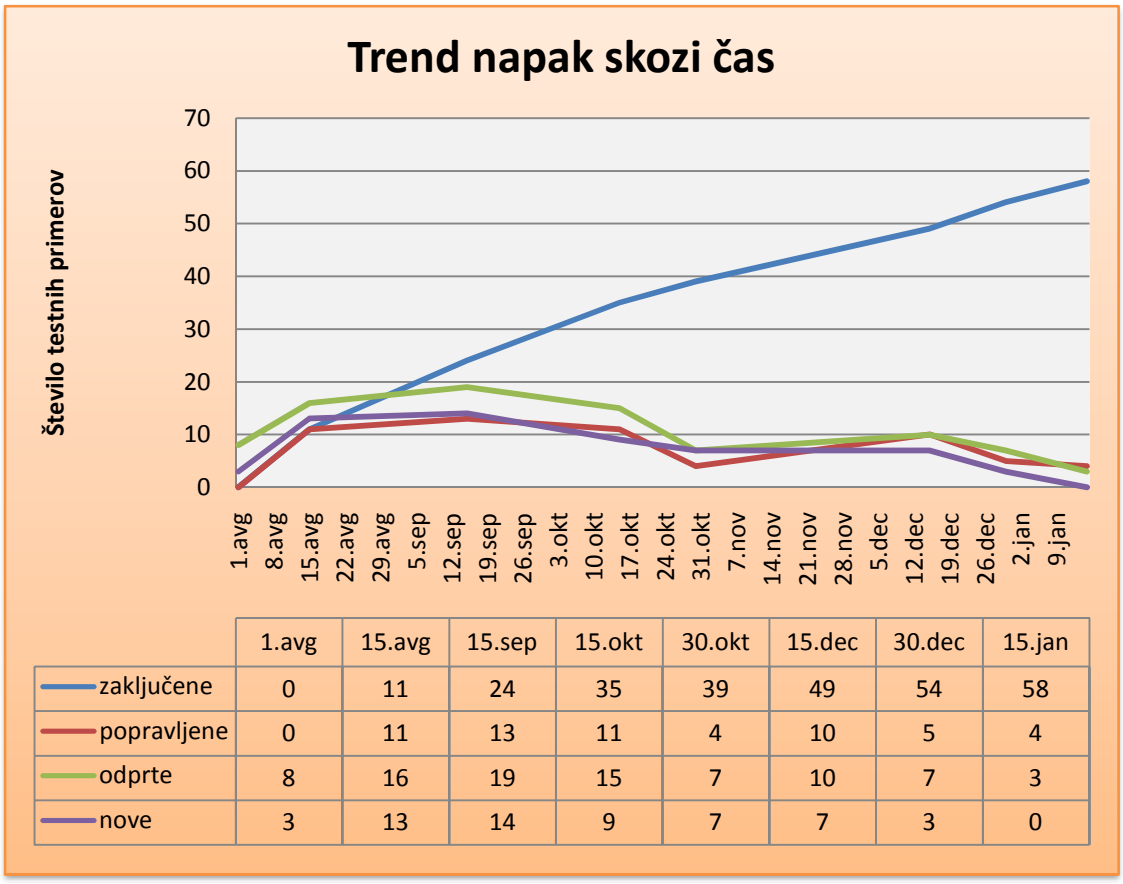
- Testiranje zmogljivosti
  - Testiranje strojne opreme ter programske opreme med sistemskim testiranjem.

Tehnične specifikacije:

- koda je napisana v programskem jeziku Delphi
- baza
  - Oracle
  - MSSQL(SQLExpress,2000,2005,2008)
- operacijski sistem: 2000/2005/XP/Vista

Ob manjših popravkih ali spremembah testna ekipa pregleda določene funkcije, katere so bile popravljene ali spremenjene. Pred vsako izdajo nove verzije pa opravimo tudi obsežnejši dimni test, ki pokriva vse ključne poti uporabnikov.

Graf 5.1 prikazuje, kako se je spreminjalo število testnih primerov skozi obdobje testiranja Setup aplikacije, ki je potekalo skozi 6 mesecev. Spremljamo štiri stanja testnih primerov: nove, odprte, popravljene ter zaključene. Pred začetkom testiranja so bili v čakalni listi trije testni primeri kot dediščina prejšnje verzije, vendar pa se je testiranje na novi verziji pričelo s 15. avgustom, kjer je zabeležena prva porast testnih primerov. Zanimivost tega grafa je, da nima oblike gaussove krivulje, kot naj bi jo imeli tovrstni grafi, ampak je dalj časa ostal raven. Ta pojav pripisujemo vedno novim fičerjem, ki so posledica iterativno-inkrementalnega razvoja z elementi agilnih metodologij.



	1.avg	15.avg	15.sep	15.okt	30.okt	15.dec	30.dec	15.jan
zaključene	0	11	24	35	39	49	54	58
popravljene	0	11	13	11	4	10	5	4
odprte	8	16	19	15	7	10	7	3
nove	3	13	14	9	7	7	3	0

Tabela 5.1. Status napak skozi čas . Ta graf nam pokaže skupno število napak po statusih skozi čas.

Naš namen z avtomatizacijo testiranja je zmanjšanje stroškov testiranja, testni ekipi dvigniti moralo (olajšati delo v smislu večje razgibanosti dela), avtomatizirati regresijske teste, jih pohitriti, jih izvajati brez prisotnosti testerjev ter seveda zagotoviti pravilno delovanje produkta in ga narediti čim bolj robustnega.

Med ožjim izborom orodij, smo se odločili za orodje TestComplete. Orodje ustreza bolj tehnično usmerjenim testnim ekipam. Ima širok izbor skriptnih jezikov, z možnostjo podatkovno ter objektno usmerjenih formatov testov. Tudi dostopen je po razumni ceni. V poglavju 3.5 je opisanih kar nekaj kriterijev, na katere moramo računati pri izbiri takega orodja. Orodje ustreza večini naših zahtev. Ne ustreza le pilotnemu projektu (Setup), vendar ga bomo lahko uporabili tudi na drugih projektih. Za pilotni projekt smo uporabili demo verzijo, ki je malenkostno okrnjena, vendar je njen nabor funkcionalnosti za enkrat zadosten:

- snemanje in simuliranje akcij uporabnika
- jezikovno neodvisne skripte
- pametno snemanje
- podpora za .Net aplikacije
- podpora za WPF(XAML) aplikacije
- dostop do notranjih objektov, metod in atributov
- podpora za testiranje modulov
- podpora ročnemu testiranju
- testiranje pod različnimi uporabniškimi računi

- brskalnik objektov (brskanje med objekti ter vpogled v njihove atribute, metode, polja, dogodke, itd.)
- test Log (sporočila različnih oblik: napaka, opozorila, slike, povezave na datoteke, itd. Sporočila lahko tudi filtriramo po času, tipu, prioriteti in jih avtomatsko formatiramo v poročila. Lahko naredimo tudi poročilo o napakah in ga avtomatsko post it v sistem za verzioniranje.)
- podpora verzioniranju
- integracija z Visual Studiem 2005
- integracija z source control sistemi, kot so Visual SourceSafe, CVS in ostali
- klicanje Win32 funkcij in funkcij DLL-ov
- itd.

5.2.4 Planiranje, analiza, načrtovanje in razvoj

Planiranje

Pripravili smo plan testiranja, kjer smo določili zadolžitve, cilje in vire, ki jih bomo uporabili, katero programsko opremo bomo potrebovali in časovni plan za razvoj in izvajanje testnih primerov. Določiti je bilo potrebno tudi standarde kodiranja, pristope k načrtovanju testov, načine pridobitve oziroma generiranja testnih podatkov. Določiti je bilo potrebno tudi, katere metrike bomo spremljali skozi vpeljavo avtomatskega testiranja.

Dimni test – Namestitev

Namestitvena verzija:_____
Tester: _____ Datum:_____
Predviden čas testiranja: 16 h
Porabljen čas testiranja:_____

1. Instalacija T&S sistema (vseh modulov) na »čist« računalnik:

Navadna
- Vključno z instalacijo SQLExpress-a
- SQL2000/2005/2008

Napredna
- Vključno z instalacijo SqlExpress-a
- SQL2000/2005/2008
- ORA

2. Preveritev zagona izbranih servisov po namestitvi:

- Vključno z instalacijo SQLExpress-a
- SQL2000/2005/2008
- ORA

3. Instalacija modulov in podatkovne baze v poljubne mape:

- Vključno z instalacijo SQLExpress-a
- ORA

4. Instalacija T&S sistema in produkcijske baze v ANG, CRO, SRB, BIH, MAC, SLO, ARA, GRE, RUS, GER jeziku; preveritev collation na bazi in Tslanguage v registru:

- SQLExpress/2000/2005/2008
- ORA

5. Postavitev nove produkcijske TSSPICA baze:

- Vključno z instalacijo SQLExpress-a
    1. Navadna
    2. Napredna
- S skripto
    1. Na SQL2000/2005/2008
    2. ORA

6. Postavitev demo baze:

- Vključno z instalacijo SQLExpress-a
    1. Navadna
    2. Napredna
- SQL2000/2005/2008

7. Instalacija revizijske sledi in preveritev delovanja v TSM:

- SQLExpress/2000/2005/2008
- ORA

8. Izdelava back-upa obstoječe baze med namestitvijo in preveritev v mapi, če obstaja varnostna kopija:

- MSDE/SQLExpress/2000/2005/2008

9. Preveri ustreznost SLO in ENG dokumentacije za vse module na T&S cd-ju (uporabniški priročnik in Kaj je novega):

- MSDE/SQLExpress/2000/2005/2008
- ORA

10. Delovanje F1 pomoči v vseh oknih namestitve in vsebinska ustreznost v SLO in ENG jeziku:

- MSDE/SQLExpress/2000/2005/2008

11. Preveri inštalacijo Timebox.dll v instalacijski direktorij in delovanje v TSM-ju:

- MSSQL
    1. Navadna
    2. Napredna
    3. Workstation namestitev
- ORA

12. Ostalo ...

(V poglavje Ostalo se zabeležijo kritične napake in postopki ponovitve, ki niso bili odkriti z zgoraj navedenimi postopki in se nanašajo na instalacijo T&S sistema.)

Kot vidimo v dimnem testu, je potrebno narediti vse te teste na več serverjih (SQL 2000/2005/2008, SQLExpress, ORACLE). Testni primeri se malenkostno razlikujejo med SQL verzijami, vendar pa je koda napisana deljeno, tako da jo bomo zlahka uporabili v vseh štirih primerih. V primeru ORACLE strežnika pa poteka proces namestitve bistveno drugače, zato bo potrebno izračunati, ali se avtomatizacija teh testov obrestuje ali bomo ostali pri ročnem testiranju. Omenimo še, da je aplikacija za namestitev T&S sistema edina aplikacija celotnega T&S sistema, kjer poteka proces drugače na Oracle in SQL strežnikih, zato bomo v primeru avtomatskega testiranja na drugih aplikacijah imeli olajšano delo, saj bomo lahko testne skripte uporabljali tako na aplikacijah, ki delujejo na Oracle kot na SQL strežnikih.

Načrtovanje

Pri načrtovanju testa aplikacije Setup smo upoštevali, da bo pri razvoju testov potrebno večkrat klicati posamezne korake, zato smo se odločili, da bomo pri pisanju skript uporabljali strukturirane in deljene tehnike pisanja skript. Za vsako točko v dimnem testu, je bilo treba napisati testni primer. Testni primeri so napisani v skriptnem jeziku DelphiScript.

Za vse razen dveh točk v smoke testu smo se odločili za avtomatizacijo testiranja. Ročno pa bomo testirali še Pomoč in dokumentacijo.

Predstavimo primer preverjanja točke 4: Instalacija T&S sistema in produkcijske baze v ANG, CRO, SRB, BIH, MAC, SLO, ARA, GRE, RUS, GER jeziku; preveritev collation na bazi in Tslanguage v registru. Za baze SQLExpress/2000/2005.

Pripraviti je bilo potrebno:

Predprocesne dejavnosti

- Ugasni vse T&S servise

Postprocesne dejavnosti

- Pobriši datoteke, ki jih je naredil Setup.exe.

Vhodne podatke:

- Licenčna datoteka: licenca.lic
- Seznam jezikov: Arabic, Bosnian, Croatian, English, German, Greek, Macedonian, Macedonian(cyrillic), Russian, Serbian, Slovenian, Other
- Možnosti namestitve (enostavna, napredna)

Izhodne podatke ter podatke primerjave

- Collation iz baze za vsak jezik
- TsLanguage v registru

Primerjave podatkov

Postizvedbena

- Instalacija T&S sistema in produkcijske baze v ANG, CRO, SRB, BIH, MAC, SLO, ARA, GRE, RUS, GER jeziku;
- preveritev collation na bazi in Tslanguage v registru

Razvoj

Spodaj opisana procedura TestNamestitveJezikov(vrsta namestitve, jeziki) predstavlja potek preverjanja točke 4 iz dimnega testa.

za i=0 do Seznam jezikov. Count	**11**	**11**
Ugasni T&S servise	**1**	**1**
Zaženi Setup.exe	1	1
Vpiši serijsko številko	1	1
Izberi jezik(Seznam Jezikov[i])	1	1
Sprejmi pogoje uporabe	1	1
Izberi vrsto namestitve (vrsta namestitve)	1	1
če vrsta namestitve = enostavna		
Izberi možnosti namestitve	1	
Izberi mesto namestitve	1	
Preveritev sistema ali so vsi predpogoji izpolnjeni	1	
Izberi namestitvene možnosti za podatke	1	
drugače če vrsta namestitve = napredna		
Izberi možnosti namestitve		1
Izberi mesto namestitve – napredno		1
Preveritev sistema ali so vsi predpogoji izpolnjeni		1
Izberi podatkovni strežnik		1
Izberi namestitvene možnosti za podatke – napredno		1
Končaj namestitev	1	1
Preveri collation	**1**	**1**
Preveri TsLanguage v registru	**1**	**1**
Pobriši ustvarjene datoteke	**1**	**1**
Skupno	14	15

Skripta je napisana strukturirano, deljeno ter je podatkovno usmerjena, da bi bila kar se da robustna in jo bo enostavno vzdrževati ob morebitnih spremembah funkcionalnosti.
Če bi hoteli preveriti (ročni ali avtomatski test) vse možne poti za točko 4, bi morali izvesti:

Vseh akcij = 11 (jezikov)*14 (akcij za enostavno namestitev)+ 11(jezikov)*15(akcij za napredno namestitev) = **154+165= 319 akcij.**

Vendar pri ročnem testiranju privzamemo, da je dovolj, če vsako vrsto namestitve (napredno, enostavno) izvedemo po enkrat in nato nadaljujemo s testi na eni vrsti namestitve.

Tako dobimo v primeru enostavne namestitve 11*14 = 154 akcij. Tem prištejemo še 15 akcij iz 1-kratne napredne namestitve in dobimo 169 akcij.

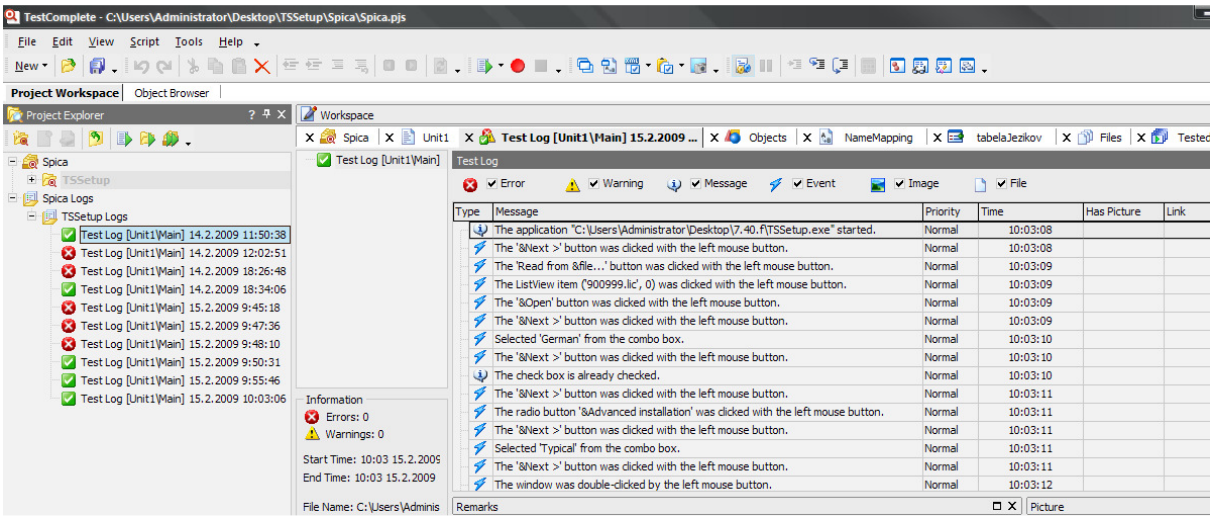
Pri avtomatskem testiranju smo pokrili celoten test z enostavno in napredno namestitvijo z enim testnim primerom, za izbiro namestitve pa uporabili atribut vrsta namestitve. V primeru popolnega ročnega testa, bi morali iti čez test kar 55-krat.

5.2.5 Izvajanje testiranja in analiza rezultatov

Setup test se izvaja vsakodnevno po vsaki izdaji dnevne verzije. Ta test je regresijski test in vsak dan preverja glavne poti. Rezultati našega testiranja se hranijo v beležnicah napak.

Naš pilotni test je našel 2 napaki. Prva napaka je bila v bistvu neveljaven rezultat testa, saj je bila napaka pravilna, vendar še ni bila zabeležena v testu. Ob vpisu v sistem T&S se je pojavilo opozorilo o neveljavni bazi, ki je bila posledica spremenjenih tabel v bazi.

Druga napaka je bila onemogočena opcija pri naprednem načinu, da lahko preskočimo kreiranje podatkovne baze.



Slika 5.1 Primer beleženja testiranja v programu TestComplete

5.2.6 Pregled in ocena procesa testiranja

Med procesom in predvsem po izvajanju testnih primerov je treba izvajati izbrane metrike ter analizirati rezultate. Podatke za izračune smo pridobili na podlagi predhodnih ročnih testov ter pilotnega projekta. Pri izračunih smo si dovolili 5 % odstopanje od točnih podatkov.

Procent pokritja z avtomatskimi testi:

Vseh testnih primerov = $139_{(MSSQL)} + 24_{(ORACLE)} = 163$ testnih primerov
Avtomatizirali smo 130 testnih primerov, 33 pa jih bomo še izvajali ročno.

Avtomatizirali smo $\frac{130}{163} \approx 80\%$ testov aplikacije Setup.

Donosnost investicije (ROI):
Pri računanju ROI si bomo pomagali s primerjavo z ročnim testiranjem, ki ga imamo že vpeljanega. V našem primeru stane ura testerja 10 €.

Ročno testiranje

V povprečju traja izvajanje enega testnega primera 8 minut. Tako bi trajal popoln ročni test 163x8 min = 1304 min ≈ 22 h.

Sicer pri ročnem testiranju nikoli ne gremo čez vse testne primere, saj jih lahko med seboj združujemo, ali pa predpostavljamo, da z enim testom lahko pokrijemo več testov. Vendar pa lahko s takšnim načinom testiranja hitro kakšen test tudi nenamerno izpustimo in spregledamo napako v sistemu. Zato bomo upoštevali v izračunih našo prvotno oceno izvajanja dimnega testa, ki obsega 22 ur.

Dimni test se izvaja pred vsako izdajo nove verzije. Izvede se v povprečju dvakrat, saj ga je ob odkritih napakah potrebno zopet izvesti. V povprečju se izdajo 4 verzije letno.
Tako pridemo do števila ur porabljenega za ročno testiranje v obdobju enega leta:

$$\text{Ročno testiranje}_{(\text{Setup})} = 22\text{h} \times 2_{\text{krat na verzijo}} \times 4_{\text{verzije}} = 176 \text{ h / leto}$$

V enem letu se v povprečju zamenjata 2 testerja. Čas potreben za uvajanje enega testerja je 10 delovnih dni. Tako porabimo 20 delovnih dni na leto za uvajanje testerjev.
Stroški ročnega testiranja letno = 176x10 € + 160x10 €= 1760 € +1600 € = 3360 €

Avtomatsko testiranje

Pri avtomatskem testiranju, je potrebno upoštevati še nekaj dodatnih stroškov, ki pridejo z vpeljavo testnega orodja. Cena orodja je 1500 €. Šolanje za uporabo orodja stane 1000 €. Za razvoj avtomatskih testov smo porabili 6 delovnih dni (48 ur).

Do sedaj smo govorili o enkratnih stroških vpeljave avtomatskega testiranja, dodati pa moramo še spremenljive stroške. Ob spremembah funkcionalnosti ali pa uporabniškega vmesnika programske opreme je potrebno teste posodobiti. Za ta opravek smo si rezervirali 1 uro na teden.

V prvem letu imamo tako stroške nakupa orodja, stroške šolanja, stroške posodobitev ter stroške izvajanja testov.
Stroški avtom. testiranja (1. leto) = 1500 € + 1000 € + 48x10 € + 52x1x10 €+8 (verzij x 2-krat na verzijo) x 1x10 € = 3580 €

V naslednjih letih pa imamo le še spremenljive stroške. Upoštevali bomo, da se zamenja en tester na leto, zato bomo dodali še eno izobraževanje.
Stroški avtom. testiranja (2. leto) = 52x1x10 €+8 (verzij x 2-krat na verzijo) x 1x10 € + 1000 € = 1600 €

Primerjava ročnega in avtomatskega testiranja

Ob podatkih, ki smo jih opisali posebej pri ročnem in avtomatskem testiranju, lahko izračunamo še razmerje med ročnim in avtomatskim testiranjem.

Razmerje (po 1. letu) = $\frac{3360€}{3580€}$ = 0.94 (imamo manjšo izgubo)

Razmerje (po 2. letu) = $\frac{2x3360€}{3580€+1600€}$ = 1.29 (29% povračilo)

Razmerje (po 3. letu) = $\frac{3x3360€}{3580€+1600€+1600€}$ = 1.49 (49% povračilo)

Računali smo, da bo orodje uporabno najmanj 3 leta. Vidimo, da imamo v 1. letu manjšo izgubo, vendar pa se ta v 2. in 3. letu spremeni v pozitivno povračilo.

Povedali smo, da se test izvaja kot regresijski test. Se pravi, se ne bo izvajal le pred izdajo verzij, ampak vsak dan po izdaji dnevne verzije. Tako se napake odkrivajo prej v procesu razvoja in s tem tudi manj stanejo. Zaradi omejenega časa tega pilotnega projekta teh meritev nismo vključili.

Poleg omenjenih atributov pa na rezultate uspešnosti vplivajo tudi atributi, ki niso merljivi. To je zadovoljstvo uporabnikov. Med uporabnike lahko štejemo testno ekipo, ki bo z avtomatskim testiranjem pridobila na raznolikosti dela, ki bo sedaj vsebovalo bolj tehnično zahtevno in manj redundantno delo. Druge vrste uporabniki pa so stranke, ki bodo dobile bolje delujoč produkt v krajšem času.

Pilotni projekt lahko označimo kot uspešen. Za oživitev avtomatskega testiranja pa bo v prihodnosti potrebno vložiti še veliko dela. Potrebno bo pokritje glavnih poti drugih aplikacij, ob vsaki verziji bo potrebno testne primere posodobiti, avtomatizirati bo potrebno analizo rezultatov, itn.

6 Zaključki

Cilj diplomske naloge je bil dokazati izboljšanje procesa testiranja ter zmanjšanje stroškov testiranja programske opreme preko vpeljave avtomatskih funkcionalnih testov. V diplomski nalogi sem predstavila konkretne napotke, kako lahko z uporabo določenih postopkov, učinkovito avtomatiziramo test in v petem poglavju tudi prikazala koristi, ki smo jih dosegli na pilotnem projektu. Našli smo primerno rešitev, ki je izboljšala proces testiranja funkcionalnih testov.

Prednosti, ki jih ima avtomatsko testiranje, so v večjem pokritju glavnih poti (čim večja avtomatizacija dimnih testov) in s tem zmanjšanje možnosti za napake. Z avtomatskim testiranjem povečamo obseg testiranja v omejenem času. S tem testna ekipa pridobi na času in ga nameni za testiranje novosti oz. vpeljavo novosti v avtomatizacijo. Testna ekipa je manj obremenjena z monotonimi in ponavljajočimi opravki, ki so pri testiranju večino časa prisotni in je bolj motivirana pri svojem delu. Velika prednost avtomatskega testiranja je izvajanje regresijskih testov. Ti se izvajajo avtomatsko vsak dan, in če je pokritje s testi dovolj dobro, bomo napako odkrili takoj, ko se pojavi.

Zavedati pa se je treba tudi omejitev oziroma ovir, ki ne govorijo v prid avtomatizaciji testiranja. Vpeljava avtomatizacije zahteva svoj čas in nova znanja. Orodja je treba izbrati/narediti, namestiti in jih spoznati, preden lahko pričnemo s testiranjem. Tudi planiranje in implementacija avtomatskih testov navadno vzame več časa, kot bi ga za to porabili pri ročnem testu. V času vpeljave avtomatizacije je potrebno računati na povečano zasedenost testne in morda tudi razvojne ekipe. Avtomatske teste je potrebno skupaj z razvojem programske opreme stalno vzdrževati. Zavedati se je treba, da vseh testov verjetno niti ne bomo mogli avtomatizirati, niti jih ne bomo želeli, saj bi bilo to nesmotrno.

Glede na zgoraj naštete ovire lahko ugotovimo, da avtomatizacija testiranja ni vedno primerna, stroškovno učinkovita ali pa celo potrebna. Preden se odločimo za naložbo v avtomatizacijo, si pri raziskovanju koristi naložbe lahko pomagamo z izračunom donosnosti naložbe. Če je ročno testiranje že vpeljano, bomo najboljše rezultate koristi naložbe izračunali s primerjavo stroškov in dobičkov, doseženih s pomočjo avtomatizacije testiranja proti ročnemu testiranju. Pri izračunih je potrebno tudi upoštevati, da se tovrstne investicije obrestujejo šele po nekaj mesecih (ali celo letih), oziroma po nekaj izvedenih projektih.

V prihodnosti pričakujem, da se bo testiranje programske opreme izboljšalo tako v količini pokritja testov kot v kvaliteti. S pojavom agilnih metodologij se je pomen avtomatskega funkcionalnega testiranja še poglobil. Testiranje in avtomatizacija testiranja imata pri agilnih pristopih ključno vlogo. S pojavom novih orodij pa je prišlo tudi do spremembe vlog, ki so prisotne pri testiranju. Večina teh orodij je napisana za razvijalce, tako da imajo testerji mnogo težav z njihovo uporabo. V nekaterih podjetjih so odgovornost za razvoj avtomatskih testov že prevzeli razvijalci. In morda bodo morali v naslednjih letih prav ljudje s poglobljenim programerskim znanjem prevzeti glavno vlogo pri razvoju avtomatskih testov.

Na koncu moram bralca še enkrat opomniti, da je avtomatizacija testiranja programske opreme zahtevna naloga, in se je ne da vpeljati čez noč. Če se je ne bomo lotili premišljeno in sistematično in je ne bomo vzdrževali, bomo tovrstno testiranje kaj kmalu opustili. Vendar ko je vpeljana, je navadno rezultat vpeljave izboljšan proces, zmanjšani stroški, bolj zadovoljna testna ekipa, hitrejši odzivni čas, itd.

Seznam slik

Seznam tabel

7 Literatura

[1.] **G. D. Everett, R. McLeod Jr.** *Software Testing.* IEEE Computer Society Press, 2007.

[2.] **Malaiya, Yashwant K.** *Automatic Test Software.* 2003.

[3.] **Solina, F.** *Projektno vodenje razvoja programske opreme. Založba FE in FRI,* 1997.

[4.] **Graham, M. Fewster & D.** *Software Test Automation.* New York : ACM Press, Addison-Wesley, cop. 1999

[5.] **M. Krisper, R. Rupnik, M. Bajec in ostali.** *Enotna metodologija razvoja informacijskih sistemov.* Vlada Republike Slovenije, Center Vlade RS za informatiko, Dec. 2003.

[6.] **Beizer, B.** *Software testing techniques.* New York : Van Nostrand Reinhold, 1990.

[7.] **B. Hailpern, P. Santhanam.** *Software debugging, testing, and verification.* IBM Systems Journal 41, 2002.

[8.] **Quest, QA.** *The newsletter of the Quality assurance Institute.* Nov. 1995.

[9.] **E. Dustin, J. Rashka, J. Paul.** *Automated Software Testing: Introduction, Management and Performance.* Addison-Wesley, 1999.

[10.] *MarketScope for Application Quality Management Solutions,* **Murphy, E.** Gartner RAS Core Research Note G00154031 , Mar. 2008.

[11.] **Robinson, R.** *Automation test tools*. http://www.vcaa.com/tools/Ray%20Robinson.pdf

[12.] *Cost Benefits Analysis of Test Automation Software Testing, Analysis and Review.* **Hoffman, D.** STAR conference, 1999.

Izjava o avtorstvu

Spodaj podpisana Kristina Jelnikar izjavljam, da sem diplomsko delo izdelala samostojno pod vodstvom mentorice doc. dr. Mojce Ciglarič, ter soglašam, da je diplomsko delo v elektronski obliki dostopno v zbirki »Dela FRI«.

V Žejah, marec 2009 Kristina Jelnikar