UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA
Podiplomski študij
Magistrski program INFORMACIJSKO-UPRAVLJALSKE VEDE

# MAGISTRSKO DELO

Sistematično testiranje programske opreme v računalniških podjetjih

Ljubljana, september 2002 Katarina Križnik

# IZJAVA

Študentka Katarina Križnik izjavljam, da sem avtorica tega magistrskega dela, ki sem ga napisala pod mentorstvom prof. Andreja Kovačiča in skladno s 1. odstavkom 21.člena Zakona o avtorskih pravicah dovolim objavo magistrskega dela na fakultetnih spletnih straneh.

V Ljubljani, dne 24.9.2002 Podpis:

# Kazalo

# 1. Uvod

## *1.1 Problematika in namen magistrskega dela*

Iz prakse vemo, da imajo vsi programi več ali manj napak (hroščev), ki se odkrijejo kasneje z uporabo izdelane programske opreme (ang. Software). Poglejmo nekaj primerov težav zaradi napak na programski opremi, ki so se zgodile ne dolgo nazaj.

Motnje v SiOL-ovem (angl. Slovenia online) omrežju, ki so jih uporabniki občutili kot močno otežen dostop do interneta v torek 29. januarja, je povzročila napaka v operacijskem sistemu Ciscove opreme. (Cisco systems je vodilno ameriško podjetje v izdelavi omrežja in najrazličnejše programske opreme za omrežje.) Napako so strokovnjaki obeh podjetij v sodelovanju s SiOL-om ugotovili s simuliranjem Siolovega okolja v svojem testnem okolju. Do težav je namreč prišlo približno sedem ur po nadgradnji jedrnega dela omrežja. Nadgradnja omrežja na gigabitno zmogljivost se bo v prihodnosti nadaljevala, z novo različico Ciscove opreme, ki bo predhodno preizkušena v testnem okolju SiOL-a in po rezultatih testov SiOL-ovih strokovnjakov nebo vsebovala napak.

Januarja letos so odkrili računalniški črv Klez.E, ki se "prebudi" na šesti dan neparnega meseca in se širi po medmrežju. Črv, ki napada Microsoftove operacijske sisteme, uniči najbolj razširjene vrste datotek, zbriše pa ne samo druge računalniške viruse in t.i. črve, temveč lahko izbriše celo protivirusne programe, če ga ti ne zaznajo. Klez.E se širi prek datotek in elektronske pošte, njegova posebnost pa je, da ponaredi podatke o pošiljatelju. Črv izkorišča že zelo dobro dokumentirano varnostno luknjo v popularnih Microsoftovih programih Outlook in Outlook Express, za katero je že nekaj časa na voljo tudi popravek. Lahko okuži tudi vse mape in diskovne pogone v skupni rabi. Poleg tega se pod različnimi imeni kopira na uporabnikov disk ter prepiše določene datoteke in z ničlami zamenja vse znake v datotekah s končnicami .txt, .htm, .html, .wab, .doc, .xls, .jpg, .cpp, .c, .pas, .mpg, .mpeg, .bak ali mp3.

Naj omenim še eno izmed napak, ki je bila odkrita pred kratkim v podjetju AOL (angl. American online) lastnikom programa za izmenjavo neposrednih sporočil, ICQ (angl. I seek you), da jih čim prej

posodobijo na zadnjo različico. V prejšnjih različicah, starejših od različice ICQ 2001b, so odkrili hrošča, ki lahko ob določenih pogojih omogoči »hekerjem«, da pretihotapijo zahrbtno kodo (trojanskega konja in podobno) na želeni računalnik, ki je trenutno priključen na internet. Glede na to, da je trenutno registriranih 125 milijonov uporabnikov programa ICQ, je razumljivo, da želi AOL čim preje zapolniti varnostno vrzel. Uporabnikom, ki že imajo novo različico ICQ-ja, ni potrebno spreminjati ničesar. Zanimivo je, da je varnostno luknjo prvi odkril nek študent univerze v Pennsylvaniji in jo takoj posredoval na varnostni seznam, ki ga ima podjetje za pošiljanje e-pošte imenovan Bugtraq. AOL se je hitro odzval in med ostalim naredil nekaj sprememb na svojih strežnikih, ki naj bi zmanjšale možnost, da bo omenjeni hrošč prizadel uporabnike. V januarju je to že drugi hrošč, ki je bil odkrit v AOL-ovi programski opremi za izmenjavo takojšnjih sporočil.

To so le trije izmed res mnogih primerov, kjer je napaka v računalniški opremi povzročila hujše probleme v delovanju programa. Da bi bilo takšnih napak čim manj in se jim bi izognili v čim večjem številu, v računalniških podjetjih vpeljujejo testiranje računalniškega projekta oz. aplikacije že med njegovo izdelavo. To zahteva tudi standard ISO 9001.

Redki programerji so optimisti, ki upajo, da bodo kar takoj napisali program brez napak. Čeprav je programer še tako izkušen in natančen, obstaja verjetnost, da napravi napako. Pisanje programov je tako zahtevno, da se v programu prav hitro pojavi napaka. Možnost, da se pri pisanju programa takoj izognemo prav vsem napakam, je zelo majhna, zato je testiranje programov ena izmed pomembnih faz v razvoju programske opreme. In večkrat se zgodi. da je prav ta faza najbolj zamudna in utrujajoča, zato je zelo pomembno, da pišemo programe disciplinirano in da že med pisanjem računamo na to, da bo treba program tudi testirati. Le tako bo napisan program lahko testirati, pa tudi njegovo kasnejše delovanje bo zanesljivejše.

Sama se že dalj časa ukvarjam s testiranjem programske opreme, ki poteka vzporedno z izdelavo projekta. Mogoče v tem trenutku ni odveč povedati, da danes piše aplikacijo več programerjev hkrati. Vsak opravi svoj del, ki ga sam testira (angl. unit testing), integracijo vseh komponent pa testira testna ekipa.

## *1.2 Cilji dela*

Cilj te naloge je predstaviti, kako v podjetje vpeljati testiranje in zakaj ga sploh potrebujemo. Se nam finančno splača imeti zaposlene ljudi, ki se ukvarjajo z iskanjem napak in do kakšne mere je testiranje smiselno?

Testi morajo biti izvedeni sistematično. Opozarjam na to, da morajo biti izvedeni na koncu vsake faze razvoja projekta, ko so komponente projekta končane. V magistrskem delu opisujem še cikel testiranja in faze razvoja projekta.

Pomemben je tudi plan testiranja, ki je nekakšen vodnik skozi celotno testiranje nastanka projekta. Premisliti je potrebno, katere stvari bodo vključene v proces testiranja in katere ne, ter kdo je odgovoren za določeno komponento projekta.

V magistrskem delu ugotavljam kakšno je število napak, ki so še sprejemljive in o tem, da niso vse napake enako kritične. Predstavim posebnosti testov glede na vrsto aplikacije.

Zastavila sem si še en cilj. Ta je spoznati specifičnost testiranja mrežnih aplikacij, aplikacij za elektronsko poslovanje in aplikacij za mobilno telefonijo ter poiskati orodja, ki pomagajo pri testiranju mrežnih aplikacij.

## *1.3 Metode magistrskega dela*

V magistrskem delu bom uporabila predvsem analizo razmer v podjetjih za razvoj računalniške opreme. Opiram se na praktična in teoretična spoznanja različnih svetovno priznanih testnih inženirjev, avtorjev člankov v reviji STQE (The Software testing & quality engineer). Primerjam jih z lastnimi izkušnjami na tem področju v podjetju Hermes SoftLab in znanjem pridobljenim na podiplomskem študiju informacijsko upravljavskih ved. Pri delu pa mi je v veliko pomoč tudi literatura, navedena na koncu magistrskega dela.

## *1.4 Struktura poglavij*

Delo je sestavljeno iz petih sklopov. Prvi trije sklopi so predvsem teoretični, ostala dva pa vključujeta praktična spoznanja na tem področju.

Prvi sklop zajema predstavitev testiranja in opis tveganj na katera je potrebno paziti pri razvoju programske opreme. V tem delu predstavim ekonomski vidik testiranja in različne vrste testov, ki jih uporabljajo v praksi. V tem delu je predstavljen življenjski cikel razvoja celotnega projekta, življenjski cikel odkrite napake ter opis, kako naj bi v podjetju razvili metodologijo testiranja. Predstavljene so tudi metode, orodja in tehnike testiranja.

Prvi del drugega sklopa vključuje dokumentacijo, ki je potrebna oz. nastane pri testiranju. Drugi del drugega sklopa, pa obravnava specifičnost testiranja mrežnih aplikacij, aplikacij za elektronsko poslovanje in aplikacij za mobilno telefonijo.

# 2. Sistematično testiranje programske opreme

Kontrola kvalitete in testiranje je proces, ki zagotavlja, da bo razvijanje programske opreme in njeno spreminjanje zadostilo zahtevam funkcionalnosti programa in ostalim zahtevam, ter da bo vsak korak pri izgradnji aplikacije vodil v pravo smer. Testni inženir mora napisati, kako bo izgledalo testiranje napak in napisati testne primere, katerim mora izdelek zadoščati. Testni inženir si mora na začetku izmisliti vhodne podatke in na njih izvesti dejansko testiranje. Ponavadi mora za potrebe testiranja napisati računalniški program, ki ga bo pri testiranju uporabil ter mu bo pomagal analizirati napake in probleme. Pri delu uporablja tudi ostale programe znanih proizvajalcev, namenjene testiranju (npr. purify, quantify, glance, truss), da z njimi testira programsko kodo in zagotovi zahtevano kvaliteto programov. Nenazadnje pa mora napisati poročilo izvedenih testov.

Pri razvoju in uporabi programske opreme lahko govorimo o tveganjih, ki so specifična za računalniško okolje. Hkrati se nekatere nevarnosti v računalniškem okolju še povečajo, česar bi se programerji morali zavedati. Te nevarnosti se odražajo v problemih, povezanih z (Kaner, 1999):

1. neustrezno uporabo tehnologije,
2. kaskadnost napak,
3. nelogičnim procesiranjem,
4. nezmožnostjo prevesti potrebe uporabnika v tehnične zahteve,
5. nezmožnostjo nadziranja tehnologije,
6. repeticijo napak,
7. nepravilnim vnosom podatkov,
8. koncentracijo podatkov,
9. nezmožnostjo hitrega odziva,
10. nezmožnostjo dokazati procesiranje (sled),
11. koncentracijo odgovornosti.

Nekaterim težavam se lahko izognemo, če se lotimo sistematičnega testiranja aplikacije. Računalniška tehnologija omogoča sistemskim analitikom in programerjem mnogotere procesne kapacitete, le te pa morajo ustrezati potrebam uporabnikov v smislu optimizacije. Neskladje med tehnološkimi sposobnostmi in potrebami namreč povzroča nepotrebno tratenje organizacijskih sredstev.

Eno izmed najpogostejših neskladij povezanih s tehnologijo je vzpostavljanje nove tehnologije še preden so jasno izražene potrebe. Npr., mnogo podjetij vzpostavlja tehnologijo baz podatkov brez jasno izražene potrebe po tej tehnologiji (npr. »data warehouse«). Izkušnje so pokazale, da mnogi uporabniki novih tehnologij porabijo ogromno sredstev pri učenju, kako uporabiti novo tehnologijo.

Kaskadenje napak je domino efekt napak v aplikacijskem sistemu. Napaka v določenem delu programa ali aplikacije sproži naslednjo, še nepovezano napako v drugem delu aplikacijskega sistema. Druga napaka lahko sproži tretjo, in tako naprej. Nevarnost kaskadnosti napak je pogosto povezana s spreminjanjem sistema aplikacije. Sprememba je narejena in testirana v programu, kjer se sprememba pojavi. Vendar, spremenijo se nekateri pogoji kot posledica spremembe, kar povzroči napako drugje v aplikacijskem sistemu.

Kaskadenje napak se lahko pojavi med aplikacijami. Tveganje se povečuje z intenziviranjem integriranosti aplikacij. Za primer poglejmo sistem, ki sprejema naročila. Ta je povezan s serijo aplikacij, ki obdelujejo naročila. Če je aplikacija za naročila preobremenjena, se to odraža v aplikacijah, ki obdelujejo naročila. Nepomembna napaka v programu za vnos naročil ima lahko zelo hitro kaskaden vpliv v množici aplikacij, kar rezultira v hudi motnji programa za polnjenje inventarja (Kiger, 1997).

Vzroki, ki povzročijo kaskadnost napak, vključujejo:
1. Neustrezno testirane aplikacije.
2. Nesožitje časa in oblike sprememb.
3. Omejeno testiranje programskih sprememb, brez upoštevanja ostalih programov.

Nelogično procesiranje je pogosto pojav v avtomatiziranem okolju in je le malo verjeten v »ročnem« procesnem okolju. Računalniške aplikacije pač nimajo enakega-človeškega pregleda nad dogajanjem kot je to značilno za neavtomatizirano procesno okolje. Poleg tega pa ima le malo ljudi dovolj dobro razumevanje procesne logike računalniških aplikacij, hkrati je v nekaterih primerih nelogično procesiranje težko prepoznavno in ga je težko odkriti brez sistematičnega testiranja.

Pogoji, ki lahko vodijo do nelogičnega procesiranja, so (Kiger, 1997):
1. Nezmožnost pregleda nenormalno velikih količin izhodnih dokumentov.
2. Področja, ki so bodisi premajhna bodisi prevelika.
3. Nezmožnost kontrole izhodnih dokumentov.

Ena izmed poglavitnih težav procesiranja je nezmožnost prevesti uporabniške potrebe v tehnične zahteve, to je komunikacijska težava med uporabniki aplikacij in razvijalci aplikacij. V mnogih organizacijah imajo uporabniki težave pri ustreznem izražanju potreb v smislu priprave računalniških aplikacij. Podobno pa so tudi razvijalci včasih nezmožni pravilno razumeti želje in zahteve uporabnikov .

Tveganje zadovoljitve uporabniških potreb je razmeroma kompleksno. Tveganje pomeni nezmožnost implementacije potreb, ker uporabniki niso bili seznanjeni s tehničnimi možnostmi, neustrezno implementirane potrebe, ker tehnično osebje ni razumelo

uporabniških zahtev, uporabniški sprejem neustrezne implementacije, ker niso prepričani, da bodo znali specificirati spremembe in vpeljava odvečnega neavtomatiziranega sistema za kompenzacijo slabosti računalniških aplikacij.

Pogoji, ki vodijo k nesposobnosti prevajanja uporabniških potreb v tehnične zahteve, so naslednji (Kiger 1997):

1. uporabniki aplikacij brez tehničnih sposobnosti.
2. razvijalci brez zadovoljivega razumevanja uporabniških zahtev.
3. uporabniška nezmožnost specificirati zahteve na zadostno natančnost.
4. večuporabniški sistem brez individualnega skrbnika sistema.

Pri klasičnem – ročnem procesnem okolju se napake pojavljajo individualno. Recimo, človek lahko izvede nalogo pravilno, naredi napako pri naslednji in zatem 20 naslednjih nalog izvede pravilno, ko znova stori napako. V avtomatiziranem sistemu so pravila konsistentna. Tako je izvajanje nalog ali procesiranje vedno pravilno, če so pravila postavljena korektno. V primeru napake v pravilih pa bo procesiranje (izvajanje nalog) venomer nepravilno.
Napake so lahko posledica programskih težav aplikacije, strojne opreme ali kupljene programske opreme.

Vzroki, ki povzročijo ponavljanje napak, vključujejo:

1. Neustrezno preverjanje vnosa glavnih podatkov.
2. Nezadostno testiranje programov.
3. Neustrezno (nezadostno) zapisovanje rezultatov procesiranja.

Računalniške aplikacije koncentrirajo podatke v enostavno dostopni, formatni obliki. V fizičnih datotekah pa so podatki prostorsko precej zahtevnejši in shranjevani v več prostorih. Zato je neavtoriziran dostop težje izvedljiv, ker pač ni mogoče dolgo neopaženo brskati po policah.
Z uporabo digitalnih medijev pa lahko neavtorizirane osebe s pomočjo programov vdirajo v podatkovne baze. To pa je težko ugotovljivo brez uporabe primernih varnostnih mehanizmov. Poleg tega je mogoče hitro in enostavno kopirati velike količine podatkov brez vidne sledi ali poškodovanja originalov, kar pomeni, da se lastniki podatkov niti ne zavedajo pomembosti le teh. Tehnologija baz podatkov še povečuje to nevarnost, saj so na enem mestu shranjene ogromne količine podatkov.

Koncentracija podatkov povečuje probleme večje zanesljivosti posameznega podatka in posamezne datoteke. Če so vneseni podatki napačni, pomenijo tem večjo težavo, čim bolj je posamezna aplikacija odvisna od konkretnega podatka. In podobno, več aplikacij, ki uporabljajo koncentrirane podatke, večja je škoda ob težavah z dostopom do podatkov, ki so bodisi programske ali strojne narave. Vzroki, ki lahko povzročijo težave zaradi koncentriranosti podatkov, vključujejo (Nierstrasz, 1999):

1. Neprimerne vstopne kontrole, ki omogočajo neavtoriziranim osebam dostop do podatkov.
2. Neustrezni podatki in težave, ki jih povzroča večnamenskost teh podatkov.
3. Vpliv pomanjkljivosti programske ali strojne opreme, ki omogoči »razprodajo podatkov«.

Največja vrednost računalniških aplikacij je sposobnost hitro in učinkovito zadovoljiti potrebe uporabnika. Nekatere procesne naloge so regularne, klasične, spet druge se pojavljajo redko in zahtevajo poseben pristop. Če računalniška aplikacija ni sposobna učinkovito rešiti teh nalog, se ponavadi gradijo nepotrebni, preobsežni sistemi. Razloga, ki zavirata odzivno hitrost, sta:

1. struktura podatkovnih datotek je nekonsistentna z iskanimi informacijami,
2. strošek procesiranja presega vrednost iskane informacije.

Računalniške aplikacije naj bi bile sposobne pustiti sled procesiranja. Sled bi tako omogočila rekonstruirati procesiranje posamezne transakcije kot tudi kontrolne vsote. Aplikacije naj bi bile sposobne prikazati vse izvirne transakcije, ki podpirajo kontrolne vsote in potrditi, da ima vsak izvorni dokument rezultat v kontrolni vsoti. Aplikacijski sistemi potrebujejo procesno sled zaradi eventualnega popravljanja napak in dokaza pravilnosti procesiranja.

Razmere, ki lahko povzročijo izgubo procesne sledi, so:
1. dokazi niso posneti,
2. dokazi niso hranjeni dovolj dolgo,
3. stroški dokazljivega procesiranja presegajo koristi ali
4. dokazi vmesnega procesiranja niso pridobljeni.

## *2.1 Kaj je testiranje in kdo vse je vpleten vanj?*

Testiranje je aktivnost, ki je tesno povezana z vsem procesom razvoja programske opreme od samega starta do konca. Prisotno je pri samem definiranju ciljev, pri izvajanju planov preverjanju rezultatov in ukrepanju pri nepravilnih rezultatih. Naloga testiranja je predvsem to, da pokaže, da produkt ustreza zahtevam stranke, hkrati pa odkrije čimveč nepredvidljivih napak.

V začetku poglavja bomo pogledali možne težave, ki se lahko pojavijo ob neustreznem razvoju programske opreme. Eden izmed možnih razlogov za to je v vseh primerih ne dovolj učinkovito testiranje produkta. Učinkovito testiranje zahteva, da so vanj vpleteni tako stranke, uporabniki produkta, razvijalci in za testiranje posebej usposobljeni inženirji. Naloga naročnika projekta je, da se pogaja za programsko opremo, ki mu bo narejena. Uporabniki lahko spoznajo še dodatne zahteve, ki jih stranka ne pozna. Razvijalci zapišejo zahteve, dizajnirajo rešitve in kreirajo program, ki ga kasneje vzdržujejo.

Testni inženirji sestavljajo skupino ljudi, ki izvaja razne funkcije na programski opremi in jih dokumentira ter poroča. Biti morajo iznajdljivi, da raziščejo čimveč možnih zapletov, ki se v praksi (ob uporabi programske opreme) lahko zgodijo.

Testiranje je panoga, ki se ji ne more izogniti niti projektni vodja in vrhovno vodstvo, saj je kvaliteta eden izmed življenjskih ciljev informacijskega podjetja. Velika informacijska podjetja imajo zaposlene presojevalce, ti so nekakšni kontrolorji učinkovitosti in primernost kontrol na področju informacijske tehnologije. Gledano iz perspektive presojevalcev je testiranje ena izmed kontrol v razvoju.

Testni inženirji morajo preučiti zahteve in dizajn ter adekvatno k vsaki zahtevi zapisati testni primer, ki vsebuje opis zahteve in pričakovan rezultat. Med samim testnim procesom morajo imeti skrbno kontrolirano upravljanje s testnimi informacijami. Testni primeri, testni rezultati in poročila bi morali biti shranjeni v bazi. Pripraviti je treba razne testne programe, ki kontrolirajo del testnih primerov in pripraviti generatorje poročil.

Glavni cilj testiranja je zmanjšati tveganje v računalniških sistemih. Strategija testiranja mora zapisati možna tveganja, ugotoviti, katera izmed teh tveganj je možno testirati in pripraviti proces, kako jih zmanjšati.

Naj sedaj predstavim najpogostejša tveganja (splošne nevarnosti v računalniškem okolju).

***Vzroki tveganja pri razvoju in vzdrževanju procesnih sistemov:***

Tveganja, ki so prisotna pri vzdrževanju procesnih sistemov lahko razdelimo glede na (Asam, 1987), (Perry, 1981):

**Definicija podatkov**

- Podatki ne bodo konsistentno definirani.
- Odgovornost za podatke ne bo identificirana.
- Pravila izbora podatkov ne bodo definirana.
- Podatkovne strukture ne bodo definirane.

**Specifikacija sistema**

- Sistemski oblikovalec bo slabo interpretiral zahteve uporabnika.
- Sistemski oblikovalec ne bo optimiziral uporabe računalnika.
- Sistemski oblikovalec ne bo optimiziral uporabe ljudi.

**Programiranje**

- Če programer ne razume zahtev po programskih spremembah, se lahko pojavijo napake.
- Programer lahko povzroči napake, če izvaja spremembe direktno skozi strojno kodo.
- Programi lahko vsebujejo procedure, ki lahko onemogočijo ali se izognejo varnostnim mehanizmom. Npr., programer, ki pričakuje, da bo odpuščen, vključi proceduro v program, ki bo izbrisala vitalne sistemske datoteke, takoj ko se njegovo ime ne bo več znašlo na plačilni listi.
- Programerji izgubijo sled spremembe v programu, rezervne kopije niso izdelane ali ne uspejo formalizirati shranjevanih aktivnosti.
- Rezultat slabo oblikovanega programa se lahko izraža v podvajanju ključnih podatkov. Do napake lahko pride, ko je izvedena zahteva po spremembi podatkovne vrednosti in se ta sprememba izvede le na enem mestu.

- Programske spremembe niso zadovoljivo testiranje pred uporabo v proizvodnji.
- Programske spremembe se lahko odražajo v novih napakah zaradi nepričakovanih povezav med programi in moduli.
- Testi programske sprejemljivosti lahko spregledajo napake, ki se pojavljajo samo ob nenavadni kombinaciji vhodnih podatkov.
- Določene programske vsebine, ki naj bi bile varovane, niso identificirane in zaščitene.
- Koda, testni podatki s povezanimi rezultati, dokumentacija za certificirane programe niso shranjeni in na razpolago.

**Dokumentacija**

- Neprimerna dokumentacija se lahko odraža v napačni programski verziji programa, ki ga spreminjamo.
- Nezmožnost modificirati sisteme, zaradi neprimerne dokumentacije.
- Dokumentacija ni razumljiva.
- Potrebna dokumentacija ni shranjena.

**Konverzija**

- Programske aplikacije / sistemi niso sposobni za proizvodnjo, ko je to potrebno.
- Pri konverziji se pojavljajo napake v podatkih.
- Ko je neuspešna konverzija sprožena, je ni mogoče ustaviti in ponovno aktivirati prejšnjo verzijo.
- Datotečna integriteta je izgubljena med konverzijo.

***Vzroki tveganja pri podatkovno procesnih operacijah:***

Tveganja, ki so prisotna pri podatkovno procesnih aplikacijah lahko razdelimo glede na (Asam, 1987), (Perry, 1981):

**Računalniško strojno opremo**

- Strojno vzdrževanje poteka med procesiranjem in ni izolirano.
- Neprimerno upravljanje s tračnimi enotami glede na trenuten status.
- Brezskrbno ali namenoma nezavarovani hranilni mediji so izbrisani. V primeru, da obstojajo rezervne datoteke, je izbris opazen šele, ko se datoteke potrebujemo.
- Interne zaščite na hranilnih medijih niso preverjene.

- Datoteke z manjkajočimi ali neustreznimi datumi poteka so izbrisane.
- Nepravilno podatkovno procesiranje ali napačno osveževanje datotek je lahko posledica mehanskih poškodb hranilnih medijev.

**Aktivnost – programsko opremo**

- Zaradi vzporednega procesiranja iste datoteke, se lahko pojavijo nekonsistentnosti pri podatkih.
- Kontrolni mehanizmi, potrebni in dostopni s kupljeno programsko opremo, niso izkoriščeni.
- Kupljena programska oprema ima napake v kontrolah, česar rezultat je napačno procesiranje.
- Kupljena programska oprema ne more biti modificirana, ko so spremembe potrebne.

**Operacije v programski opremi**

- Nepazljiv ali nepravilen ponovni zagon po izključitvi lahko povzroči, da zadnje transakcije niso zapisane.
- Operater lahko vnese napačno informacijo na konzolo.
- Lahko se izvaja napačna programska verzija.
- Program je lahko izveden ob uporabi napačnih podatkov ali dvakrat za isto transakcijo.
- Operater se lahko izogne zahtevanim varnostnim kontrolam.
- Zaradi površno naučenih procedur lahko operater spremeni ali izbriše glavne datoteke.
- Ključne datoteke so lahko v uporabi, ne da bi bile zaščitene pred spremembami.

**Varnostno kopijo (ang. backup)**

- Operacijski sistem ne uspe zabeležiti, da so bile narejene številne kopije izhodnih podatkov.
- Operacijski sistem ne uspe obdržati neprekinjene revizorske sledi.
- Operacijski sistem ne uspe izbrisati vsega razdrobljenega prostora, dodeljenega operaciji, po normalnem ali nenormalnem zaključku operacije.
- Datoteke lahko dopuščajo branje in pisanje, ne da bi bile odprte.

- Operacijski sistem lahko kopijo informacije zavaruje šibkeje kot original.
- Med reorganizacijo baze podatkov ali sproščanjem prostora na disku lahko pride do uničenja datotek.
- Operaterji lahko ignorirajo operacijske procedure.
- Jezikovni parametri nadzora nad operacijo so lahko napačni.

**Varnost**

- Podatki ali programi so lahko odtujeni iz računalniških prostorov oz. Prostorov, v katerih so hranjeni
- Posamezniki niso zadovoljivo identificirani pred dovoljenjem za vstop v delovno področje.
- Oddaljeni terminali niso zadostno zaščiteni pred neavtorizirano uporabo. Nepooblaščena oseba lahko pridobi dostop do sistema skozi telefonsko linijo in z geslom pooblaščene osebe.
- Gesla so nehote razkrita nepooblaščenim osebam. Uporabnik si lahko geslo zabeleži na neprimerno mesto ali pa je geslo razvidno iz zavrženega tiskanega papirja ali pridobljeno z opazovanjem.
- Odpuščeni uslužbenec lahko obdrži dostop do sistema, ker sta njegovo ime in geslo izločena prepočasi iz avtorizacijskih tabel in kontrolnih seznamov.
- Nepooblaščena oseba lahko pridobi dostop iz osebnih razlogov (kraja računalniških storitev, podatkov ali programov, sprememba podatkov, sprememba programov, sabotaža, preprečitev storitev).
- Ponavljani poizkusi posameznika oz. terminala za pridobitev nepooblaščenega dostopa, so lahko neopaženi.
- Oblikovna ali implementacijska napaka operacijskega sistema lahko uporabniku dovoli izključitev revizijskih kontrol ali dostop k vsem informacijam sistema.
- Slabo definiran kriterij za pooblaščeni dostop se lahko odraža v zmedi glede dovoljenega dostopa. Oseba, odgovorna za varnost, ne uspe omejiti uporabniškega dostopa do procesov in podatkov, ki so potrebni za izpolnitev dodeljenih nalog.
- Občutljivi podatki so lahko površno obravnavani pri aplikacijskem osebju, z el. pošto, ali drugem osebju v organizaciji.
- Nepremišljena sprememba ali uničenje datoteke je lahko posledica pripravniškega dela s pravimi podatki.

- Ni sprožena primerna akcija, ko sporočilo o kršenju varnosti doseže osebje zadolženo za varnost; postopki sploh niso določeni.
- Uslužbenec lahko odtuji programsko opremo, ki jo uporablja, za osebno korist.
- Instalater se lahko izogne operacijskim kontrolam in pridobi informacije.
- Programerji lahko vključijo posebne programske omejitve za manipuliranje s podatki, ki se nanašajo na njih.
- Vzdrževalno osebje se lahko izogne varnostnim kontrolam pri vzdrževanju. V takšnih primerih je sistem izpostavljen napakam ali namernim kršitvam vzdrževalnega osebja ali kogarkoli, ki je v tistem trenutku priključen na sistem.
- Nepooblaščena oseba lahko prevzame računalniška komunikacijska vrata v trenutku, ko se pooblaščeni uporabnik odklopi. Mnogi sistemi spremembe ne zaznajo.
- Če se uporabljajo šifre, se lahko odtujijo šifrirni ključi.
- Neprava sporočila so lahko vnesena v sistem. Prava sporočila pa so lahko iz sistema izbrisana.
- Zlom operacijskega sistema lahko izpostavi dragocene informacije – gesla in avtorizacije.
- Podatki in programi so lahko ukradeni po telefonski liniji z oddaljenega terminala.
- Hranilni mediji, ki vsebujejo občutljive podatke, ne dobijo zadovoljive zaščite, ker operacijsko osebje ni obveščeno o naravi vsebovanih informacij.

**Komunikacije**

- Informacija je lahko po nesreči preusmerjena na napačen terminal.
- Komunikacijska stičišča lahko pustijo nezaščitene dela sporočil v spominu med nepričakovano prekinitvijo procesiranja.
- Komunikacijski protokol ne uspe pravilno identificirati oddajnika in sprejemnika sporočila.
- Neodkrite komunikacijske napake se lahko odražajo v napačnih ali spremenjenih podatkih.

**Baze podatkov**

- Podatki niso ločeni od programske kode, kar je vidno v težavah pri spremembah programa.

- Vzporedno vnašanje več uporabnikov lahko povzroči izgubo ene ali več osvežitev datotek.
- Kazalci lahko postanejo nefunkcionalni, kar lahko povzroči izgubo podatkov.

***Vzroki tveganja v podatkovno procesni aplikaciji so:***

Tveganja, ki so prisotna pri podatkovno procesnih aplikacijah lahko razdelimo glede na (Asam, 1987), (Perry, 1981):

**Začetek**

- Pomanjkanje nadzora nad dokumenti in ostalih kontrol nad virom podatkov transakcij lahko omogoči, da se nekateri podatki ali transakcije izgubijo brez sledi ali pa se dodajo nove.
- Nerazumni ali nekonsistentni izvorni podatki so lahko neodkriti.

**Avtorizacija**

- Pisna oblika mehanizma dodeljevanja dostopa ne obstoji ali se ji ne sledi.
- Zaščita s kartico – ključem nad bistvenim področjem ne obstoji ali se ne uporablja.
- Dostop do aplikacij ni omejen.
- Manjka zahteva, da si posameznik, ki je odgovoren za določeno sredstvo, vzpostavi nadzor nad dostopom do teh sredstev.
- Standardizirane procedure za avtorizacijo ne obstajajo.
- Ne obstaja organizacijska shema, ki ilustrira konkretno avtorizacijo in dostop.
- Predtransakcijski izvedbeni pregled ni izvršen.
- Formalna odobritev kapitala in neregularnih izdatkov ne obstoji.
- Formalna odobritev vseh povečanj plač ne obstoji.

**Vnos podatkov**

- Tipkarskih napak med vnašanjem ni možno ugotovit.
- Nepopolni ali slabo formatirani podatkovni zapisi so lahko sprejeti in obravnavani kot da gre za popolne podatkovne zapise.
- Podatki, ki so pridobljeni v zadnjih trenutkih, so lahko nepotrjeni pred procesiranjem.
- Zapisi, pri katerih so bile odkrite napake, so lahko popravljeni brez verifikacije celotnega zapisa.

**Procesiranje**

- Procesna pravila bodo nekorektno kodirana.
- Programske napake bodo povzročile napake pri procesirani podatkih
- Ugotovljive napake ne bodo ugotovljene.
- Nespecificirani podatki bodo procesirani na nepričakovan način.

**Hranjenje dodeljenih datotek**

- Dodeljene datoteke ne bodo primerno identificirane in bodo izgubljene.
- Sprejeta bo napačna datotečna verzija.

**Izhod (ang. output)**

- Izhod je lahko poslan napačnemu posamezniku ali terminalu.
- Neprimerno obravnavan izhod ali poprocesne enote povzročijo izgubo izhodnih podatkov.
- Preobsežni izhodni podatki bodo odstranjeni na neprimeren način.

Vsa zgoraj našteta tveganja veljajo za različna računalniška okolja in aplikacije. Strategija testiranja mora vsebovati možna tveganja za našo aplikacijo in pripraviti proces kako jih zmanjšati.

## *2.2 Kaj je napaka (hrošč) v programski opremi?*

Do kakšnih napak vse lahko pride pri uporabi programske opreme in da je ta množica zelo velika sem opisala v z veliko množico tveganj. Kaj pa so pravzaprav hrošči, ki jih testni inženirji že med razvojem programske opreme iščemo?

Vsa zgoraj našteta tveganja ne moremo vključiti v testiranje. Potreben je premislek, ki za dan primer pove, katera tveganja je koristno vključiti v strategijo testiranja. Tveganja, ki jih lahko kontroliramo, imenujemo testni faktorji in te vključimo v strategijo testiranja. Napaka (hrošč) v programski opremi je variacija produkta od predpisanih lastnosti. Testni inženirji iščejo te napake, razvijalci pa najdene napake popravijo, še preden gre programska oprema do uporabnika. Zato je zelo pomembno, da stranka dobro definira zahteve in da v komunikaciji s stranko ugotovimo možna tveganja.

Napake programske opreme lahko razdelimo glede na specifikacijo v dve kategoriji (Perry, 2000).

### *2.2.1. Funkcionalnost produkta varira glede na specifikacijo zahtev.*

Na samem začetku razvoja programske opreme se sestaneta programski vodja (ang. program manager) in stranka ter specificirata vse zahteve, ki jih zapišeta v dokument. Programerji se morajo teh zahtev držati, in če kakšno funkcionalnost implementirajo nepopolno, je to hrošč. Napako lahko imajo algoritmi. V to kategorijo bi pa uvrstila tudi performančne napake, napake glede uporabnosti in napake po varnosti. Skratka, tukaj so vse zahteve stranke, ki niso bile pravilno implementirane.

### *2.2.2. Neskladnost programske opreme s strankinimi pričakovanj, ki niso bile specificirane*

Pomembno je, da stranka resnično specificira in dokumentira vse zahteve. Nezadostnost specificiranja zahtev se kaže v nezadovoljivi funkcionalnosti, pomanjkljivi uporabnosti, predolgih odzivnih časih, itd.

Preverjanje vseh zahtev, ki spremljajo projekt, je ogromen in izredno pomemben projekt. Testni inženirji morajo preučiti dizajn in adekvatno k vsaki zahtevi zapisati testni primer, ki vsebuje opis zahteve in pričakovan rezultat.

## *2.3 Tipi testov in značilnosti*

Testiranje velikih programskih projektov vsebuje več korakov in tehnik. Konceptualno lahko ločimo vrste testov v tri večje sklope. Ti so testi enot (ang. »unit test«), integracijski testi in sistemski testi. Nobeden od njih ni bolj ali manj pomemben od drugega sklopa. Med sabo so povezani, se včasih delno pokrivajo in se malo razlikujejo glede na razvojni model projekta. Znotraj njih ločimo odobritvene teste, teste funkcionalnosti, instalacijske, konfiguracijske, alfa in beta

teste, teste zanesljivosti, regresivne teste, teste performance, obremenitvene teste in uporabnostne teste.

Testiranje enot potrdi funkcionalnost posameznih manjših delov programske kode, ki se jih da testirati ločeno in neodvisno od ostalih delov. Ti delčki so lahko posamezni programčki ali pa komponente, ki vsebujejo več tesno povezanih enot. Ti testi se lahko izvajajo s pomočjo orodij za odkrivanje napak (ang. debuging orodij) posameznih programskih okolij in jih lahko izvajajo programerji (pisci kode) sami.

Integracijsko testiranje se začne po končanem testiranju enot. To je proces, ki kontrolira komponente, ki so že bile posamično stestirane v sklopu testiranja enot. Pri integracijskem testiranju se dodaja več komponent skupaj, lahko po vrsti, lahko pa po logičnih sklopih. Večkrat pa se pod integracijskimi testi razume vse komponente in ne le nekaj skupaj. V integracijskih testih se kontrolira funkcionalnost projekta. Izvedejo se testni primeri, ki zadevajo funkcionalnost.

Sistemsko testiranje vključuje obnašanje celotnega sistema. Večina funkcionalnosti bi morala biti že preverjena v sklopu testiranja enot in integracijskih testov. Sistemsko testiranje preverja in primerja sistem z zahtevami kot so hitrost, zanesljivost, natančnost. V sistemske teste se šteje tudi kompatibilnost, preverjanje aplikacije v različnih operacijskih sistemih, različnih brskalnikih in različno strojno opremo. Nenazadnje pa v sistemske teste prištevamo tudi varnost.

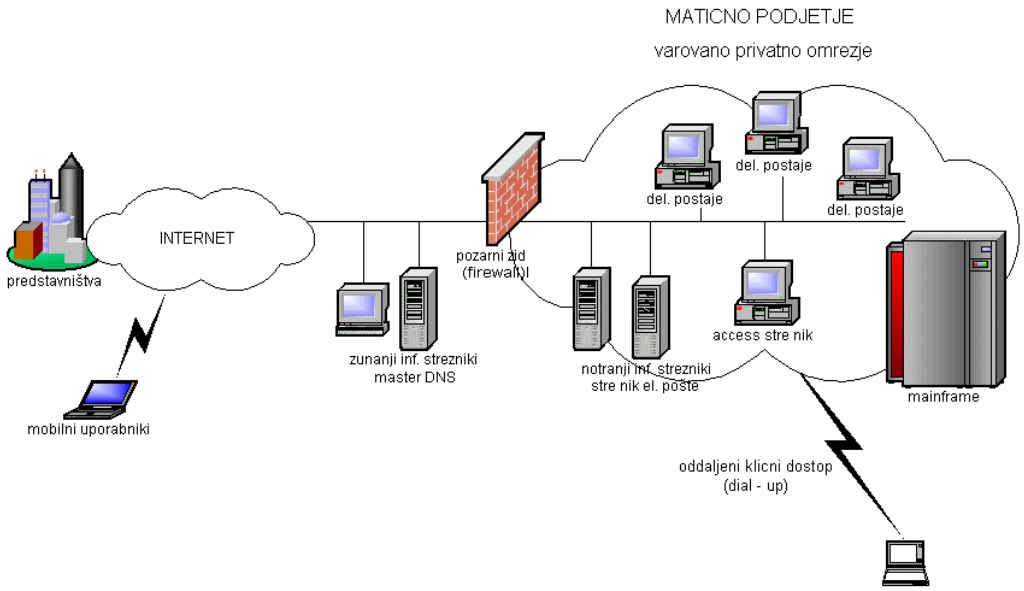
Čeprav bi nas mnogi proizvajalci radi prepričali v nasprotno, ne obstaja izdelek ali sistem, ki bi nam zagotovila popolno varnost. Nevarnosti nikoli ne moremo popolnoma odpraviti, lahko pa zmanjšamo verjetnost, da bo šlo kaj narobe, in poskrbimo, da bo neprijetnih posledic čim manj, ko se bo to zgodilo. Zato ponavadi govorimo o stopnji zaupanja, ki ga imamo v to, da se bo računalniški sistem obnašal v skladu s pričakovanji. Do tega lahko pridemo le s temeljito preučitvijo potencialnih nevarnosti, ocene razmerja med stroški in dobljenimi koristmi in preudarno izdelavo varnostnih pravil.

Pojmovanje računalniške varnosti v podjetjih je nekoliko drugačno od tistega doma. Medtem ko se doma poskušamo zavarovati predvsem pred lastnimi napakami in zlorabami v omrežju, je dojemanje računalniške varnosti v podjetjih precej celovitejše in zahtevnejše.

Čeprav so mnoge uporabljene tehnologije iste, se pogosto uporabljajo v druge namene.

Poglejmo si primer računalniškega omrežja podjetja (slika 2.1), s katerimi se danes srečujejo razvijalci aplikacij in navadni uporabniki.

Slika 2.1: Tipična shema sodobnega računalniškega omrežja




vir: http://grunf.hermes.si/QualityManual/

V terminologijo vrste testov uvrščajo tudi alfa in beta testiranje. Alfa testiranje je testiranje v lastnem testnem laboratoriju, lastnem okolju, medtem ko je beta testiranje, testiranje pri stranki. Dostikrat se ti dve vrsti testov ne vključi v testni plan in se ju nekontrolirano meša.

Poglejmo si sedaj podrobneje nekaj tipov testov, ki bi jih morali izvajati že v razvoju projektov. Seveda pa je količina posameznih testov odvisna od narave projekta, ki ga razvijamo.

### *2.3.1 Testi funkcionalnosti*

Najpomembnejša je funkcionalnost projekta, saj brez prave funkcionalnosti nima smisla testirati ostalih lastnosti, katerim naj bi program ustrezal.

Testi funkcionalnosti zagotavljajo, da je zahtevam stranke, ki so navedene v dokumentu, ustrezno zadovoljeno. Aplikacija mora

vsebovati vse zahtevane funkcije, ki morajo pravilno delovati. S testom funkcionalnosti moramo zagotoviti ustrezno število testnih primerov, da resnično pokrijemo vse zahteve po funkcionalnosti.

Pri testiranju funkcionalnosti nas ne zanima, kako sistem deluje, ampak le rezultat delovanja. Pri določenih vhodnih podatkih moramo dobiti določen odgovor funkcije. Ta vrsta testiranja je ena izmed lažjih, saj ni potrebno poznavanje logike programa. Pri teh testih pa je zelo pomembna natančnost, saj je število primerov, ki jih je potrebno preizkusiti, veliko večje, kot pri ostalih tipih testiranja.

### *2.3.2 Test uporabnosti*

Lahko se zgodi, da aplikacija po funkcionalnosti ustreza zahtevam stranke, ni pa uporabna zaradi nelogičnih korakov in odvečnih klikov, ki so potrebni, da pridemo do želenega rezultata, ki naj bi ga aplikacija dala. Funkcije, ki se večkrat uporabljajo, bi morale biti dosegljive že na prvem koraku in ne šele po množici korakov. Pri uporabnosti je pomemben tudi izgled, ki lahko avtomatično namiguje navigacijo programa.

Fino je, da se stranka zaveda pomembnosti uporabnosti in vse zahteve poda že v specifikaciji. Ti tipi testov obsegajo kontrolo enostavnosti uporabe aplikacije. Test uporabnosti lahko izvajamo na različne načine, kot so raziskave, opombe svetovalcev, testi v produkcijskem okolju. Glavni namen testa uporabnosti je zagotovi enostavno uporabo in logično navigiranje.

### *2.3.3 Testi kompatibilnosti*

Konkurenca na področju razvoja računalnikov povzroči, da uporabljamo različne operacijske sisteme in različne verzije le teh. Pomembno je tudi celotno okolje, v katerem aplikacija teče. Pri aplikacijah na omrežju naredijo veliko zmedo brskalniki, ki jih je ogromno, še več pa njihovih verzij.

Testi kompatibilnosti so lahko izvedeni v testnem laboratoriju, kjer so računalniki z različnimi verzijami operacijskih sistemov in različni brskalniki. Pri testiranju na različnih operacijskih sistemih pa

moramo vedno dodat še različne brskalnike. Najbolje je, da si naredimo matriko, ki kaže, katere kombinacije bomo testirali. Včasih pa se zadovoljimo s testiranjem na najnižji in najvišji verziji programske opreme in pričakujemo, da v vmesnih verzijah ne bo katastrofalnega odstopanja.

### *2.3.4 Preizkus delovanja, učinka*

Učinek aplikacije je zelo odvisen od hitrosti izvedbe določenih korakov v aplikaciji. Prepočasen odziv aplikacije lahko kljub popolni funkcionalnosti povzroči neuporabnost aplikacije. Ker se pomembnosti performance zavedajo tudi manadžerji, zapišejo zahteve že na samem začetku v dokument tako imenovani FURPS+ (»Functionality, Usability, Reliability, Performance, Supportability, Portability, Localizability, Additional objectives«, definicija funkcionalnosti in ostalih lastnosti produkta) .

Testiranje učinka, pomeni izvajanje testov, ki zagotavljajo, da sistem ustreza dogovorjeni performanci. V ta sklop prištevamo hitrost izvajanja statičnega in dinamičnega procesiranja, hitrost izvajanja transakcij...

### *2.3.5 Test zanesljivosti*

Zanesljivost je vrlina. In kaj je zanesljivost aplikacije? To ni samo zanesljivo delovanje funkcionalnosti. Pod zanesljivost štejemo tudi ponoven zagon aplikacije po ponovnem zagonu ( ang. restartu), vrnitev v stanje, ki je bilo pred začetkom procesiranja nedokončanih transakcij.

Test zanesljivosti lahko vsebuje izgubo kapacitete spomina, izgubo komunikacijske linije, poškodbe programske opreme in odpoved operacijskega sistema. Nemogoče je stestirati vse aspekte zanesljivosti. Nekaj značilnosti testa zanesljivosti (Nielsen, 1994):

- Sprožimo napako v sistemu med procesiranjem naše aplikacije in gledamo transakcije, ki lahko povzročijo nezaželeno obnašanje aplikacije.

- Naredimo rezervno kopijo podatkov, sprožimo nepredvideno napako. Po ponovni vzpostavitvi sistema primerjamo rešene podatke in podatke rezervne kopije.

Test zanesljivosti bi se moral izvajati vedno, kadar je kontinuiteta operacij nujna za normalno delovanje funkcionalnosti.

### *2.3.6 Test varnosti*

Razvoj tehnologije prinaša vedno nove in nove probleme, tako da je obravnava vseh praktično nemogoča. Za potrditev te trditve se ni treba ozirati daleč. Dovolj je, da se spomnimo na spremembe, ki jih prinaša internet, ki pa ni končna postaja, ampak prej začetek poti.

Najpomembneje se je zavedati, da obstajajo orodja, s katerimi lahko povečamo svojo varnost, vendar se tako kakor drugod v življenju ne moremo zanesti na to, da bo tehnologija rešila naše težave. Računalniška varnost ob nenehnem razvoju ni cilj, temveč pot, ki od nas zahteva veliko mero ozaveščenosti in udejanjanja .

Namen varovanja informacij je zagotavljanje neprekinjenega poslovanja in omejevanje poslovne škode na najmanjšo možno mero s preprečevanjem in zmanjševanjem učinkov varnostnih incidentov. Varovanje informacij omogoča dostop do skupnih informacij, s tem ko zagotavlja zaščito informacij in računalniške opreme.

Informacije nastopajo v različnih oblikah. Shranjene so na računalnikih, prenašajo se preko omrežja, so natisnjene, napisane na papirju, ali izgovorjene. Usmeritev k delovanju v omrežju zmanjšuje obseg osrednjega, strokovnega obvladovanja računalniških sistemov. Varnostni ukrepi so znatno cenejši in učinkovitejši, kadar so vgrajeni v sisteme in storitve informacijske tehnologije na stopnji specifikacije zahtev in zasnove programov. Čim prej ukrepamo v smeri varovanja računalniških sistemov, tem cenejše in uspešnejše bo to za podjetje na dolgi rok.

Da bi se izognili nesporazumom o posameznih odgovornostih, je bistvenega pomena, da jasno ugotovimo, za katera področja je odgovoren posamezen vodja, predvsem pa v naslednjih primerih (Smith, 2001):

- Ugotoviti in jasno opredeliti je potrebno vsa sredstva in varnostne procese, ki so povezani z vsakim posameznim sistemom.
- Vodja, ki je odgovoren za neko sredstvo ali varnostni postopek, naj bo uradno potrjen in njegova odgovornost pisno dokumentirana.
- Ravni pooblastila morajo biti jasno opredeljene in dokumentirane.

Zadostnost in kvaliteto procedur, ki skrbijo za varnost aplikacije, je potrebno testirati (MacIntosh, 2000). Poskušati je potrebno spremeniti podatke z neavtoriziranim dostopom do aplikacij. Uporabiti je potrebno vse svoje znanje in domišljijo. Če je v aplikaciji več takšnih mest, kjer lahko varnost odpove, je smiselno vplesti v testiranje strokovnjake s tega področja.

### *2.3.7 Obremenitveni testi*

Običajno sistem deluje, ko z njim dela določeno število ljudi in ga do neke mere obremeni. Za primer obremenitve bi lahko navedli primer iz mobilne telefonije. Vsi lahko z mobilnim telefonom normalno telefoniramo. Običajno je na določeni lokaciji določeno število ljudi. Ob nepredvidljivi situaciji, ko je več ljudi na istem mestu (večji dogodki, prireditve npr. skoki v Planici) pa mobilno telefoniranje odpove. Z obremenitveni testi ugotovimo, kako se aplikacija obnaša in če lahko sistem normalno deluje tudi pod večjim obsegom obremenitev, kot so običajne. Obremenitve se lahko pojavijo pri povečanem številu transakcij, pomanjkanju prostora na disku, komunikaciji med računalniki ...

Če se bo aplikacija pod obremenitvenimi testi obnašala primerno, lahko pričakujemo, da se bo tudi pri normalnih pogojih obnašala točno. Objektivnost obremenitvenih testov dosežemo s simulacijo produkcijskega okolja, v katerem lahko simuliramo običajno število transakcij z možnostjo procesiranja velikega števila podatkov. Imeti moramo dovolj sredstev, da simuliramo komunikacijsko linijo in možne časovne zakasnitve. Kdaj sploh delati stresne teste? Vedno, kadar obstaja dvom o količini podatkov, ki jih aplikacija lahko obravnava brez napake. Obremenitveni testi poskušajo zlomiti sistem

z preobremenitvijo, velikim številom transakcij. Slaba stran obremenitvenih testov pa je čas za pripravo testov in sredstva, porabljena za izvajanje le-teh. Pretehtati je potrebno ceno testiranja z rizikom neodkritja napak pred postavitvijo projekta v produkcijo.

# 3.Testiranje z uporabo metodologije življenjskega cikla

Cikel razvoja projekta vsebuje neprestano testiranje projekta skozi razvojni proces. Na določenih točkah razvojnega procesa je potrebna kontrola, da zagotovimo pravilnost implementacije in odkrivanje napak v zgodnjih fazah razvoja. Sam proces testiranja se ne more začeti, dokler ne dopolnimo razvojnih faz projekta.

Testiranje je odvisno od samega procesa razvoja in dokumentacije ter drugih izdelkov. Če odklanjamo tak razvoj projektov, lahko postane testiranje brez učinka. Takšen projekt zaradi neprestanih sprememb poveča stroške.

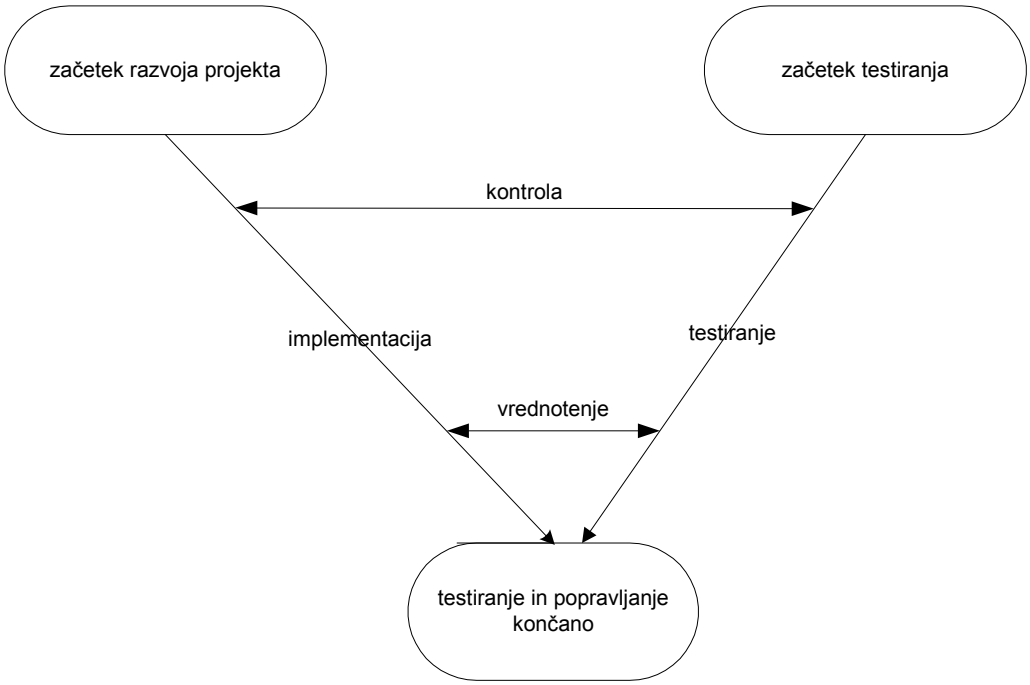
Testni proces oziroma metodologija testiranja bo najboljša, če jo formira testna ekipa skupaj z razvijalci projektov. Obvezno pa je, da tako razvijalci kot testni inženirji upoštevajo metodologijo testiranja. Nikakor se ne sme zgodit, da razvijalci razvijajo na testnem okolju in testni inženirji testirajo na razvojnem, saj v tem primeru ne vemo, katera verzija se testira, ne znamo ponovit napake. Testni proces moramo opravljati z enako stopnjo resnosti in odgovornosti kot sam razvoj.

Dobro je, da se vloge med testnimi inženirji in razvijalci od projekta do projekta menjajo. Vsak testni inženir mora biti tudi razvijalec in obratno. Nikakor ne sme razvijalec testirat projekta, ki ga je razvijal, saj so izkušnje pokazale, da so razvijalci »slepi« za svoje napake.

Testiranje ima učinek, če se drži metodologije in ga ne izvajajo avtorji kode.

Na sliki 3.1 je prikazan koncept življenjskega cikla razvoja projekta. Ta pove, da se istočasno z razvojem projekta začne tudi testiranje projekta.

Slika 3.1: Prikaz sočasnosti testiranja projekta in razvoja projekta v V-diagramu.




Vir: Perry, 2000

Priprava testne dokumentacije mora biti končana hkrati z dokončanjem aplikacije. Začne se testiranje in popravljanje napak v lastnem okolju, to je tako imenovano alfa testiranje (ang. alfa testing). Ko pa aplikacija zadošča zahtevam (kritičnih napak ni več), jo lahko inštaliramo v produkcijsko okolje (okolje stranke), kjer se aplikacija ponovno testira, začne se tako imenovano beta testiranje (ang. beta testing).

Takemu testiranju pravimo V-testiranje (ang.V-testing). Skupina testnih inženirjev in razvijalcev že od samega začetka tesno sodelujeta. To zmanjša tveganje projekta in poveča verjetnost pravočasnega končanja aplikacije.
Mogoče ni odveč omeniti, da morajo biti tako komponente projekta kot vsi predmeti testiranja: testni podatki, rezultati testov enot (ang. »unit test«), testni plani, testne procedure in testni podatki) pod kontrolo

verzij. Shranjene morajo biti vse prejšnje verzije, da se lahko ob odkritju napak izpostavi stara verzija.

## *3.1 Življenjski cikel razvoja projekta*

Samo pomislimo lahko, kako bi razvili kompleksen računalniški program brez predhodnega načrtovanja, brez načrta kako bo potekalo programiranje. Verjetno podobno kot izdelava hiše brez predhodnega načrta oziroma bodo posledice še hujše. Vendar v resnici mnogi posvečajo premalo pozornosti analizi in načrtovanju, čeprav poudarjam, da v našem podjetju Hermes Softlab ni tako. Ena od rešitev za kakovostno izdelavo projektov je ustaljen cikel razvoja projekta. Tega zahtevajo tudi mednarodne organizacije za standardizacijo, kot je ISO (International organization for stardandization) in IEEE (Institute of Electrical and Electronics Engineers).

V nadaljevanju prikazujem življenjski cikel razvoja projekta, v katerem imajo razvijalci stalen kontakt s stranko in nimajo ogromno dodatnih zahtev med razvojem projekta. V tem življenjskem ciklu je 5 faz, ki se v splošnem med sabo ne pokrivajo in si sledijo ena za drugo. Naslednja faza razvoja projekta se ne more začet, dokler se ne konča prejšnja.

Življenjski cikel ima v vsakem podjetju nekaj svojih specifičnosti, in je odvisen od metodologije razvoja projekta ter od specifičnosti projekta. V grobem pa je enak opisanemu v tem poglavju.

Splošne točke oz. koraki, ki so predpogoj za uspešno izvajanje vsakega testnega procesa ter koraki, po katerih se izvajajo testni procesi so:

1) FURPS+ (Functionality, Usability, Reliability, Performance, Supportability, Portability, Localizability, Additional objectives definicija funkcionalnosti in ostalih lastnosti produkta), HLA (High Level Architecture) sta dokumenta, ki podrobno opisujeta zahteve po funkcionalnosti, uporabnosti, zanesljivosti ter arhitekturo programa. Testna ekipa ta dokumenta prebere z namenom spoznati se z zahtevami in predmetom testiranja. Lahko komentira in zahteva spremembe.

2) Sledi priprava načrta testiranja. Načrt pripravi projektni vodja ali pa vodja testne ekipe. Pregleda in odobri ga projektni vodja.

3) Naslednji korak je priprava testne specifikacije za teste enot (ang. unit test). Testi enot zagotavljajo pravilno delovanje posameznih delčkov kode. Test enot je lahko tudi analiza in pregled kode, da se preveri, če so bile polovljene izjeme, če je pokrita vsa funkcionalnost, pa tudi sama struktura kode.

   Za pripravo specifikacije testov enot in za njihovo izvedbo v njej specificiranih testov ter za pripravo poročil(a) o rezultatih testiranja so zadolženi razvojni inženirji. V tem koraku programerji pridno programirajo, testni inženirji pa pripravljajo pogoje za testiranje. Programerji izvedejo test enot.

   Testi enot morajo biti uspešno izvedeni preden se začne integracijsko (ang. integration testing) in sistemsko (ang. system testing) testiranje. Projektni vodja lahko odloči, da poročil o testih enot (testnih specifikacij in rezultatov) ni potrebno hraniti.

4) Priprava testne specifikacije za sistemske in integracijske teste .
   - testiranje delovanja in obnašanja produkta v izjemnih situacijah (ang. stress test),
   - testiranje performanc (ali se akcije izvedejo v pričakovanem času),
   - testiranje, ali različni podmoduli skupaj delujejo kot celota,

   Za pripravo testne specifikacije na osnovi FURPS+, HLA in ostalih dokumentov, v katerih so definirane zahteve in lastnosti, ki jim mora ustrezati produkt je zadolžena testna ekipa. Projektni vodja preveri in odobri pripravljeno specifikacijo.

   Za izvedbo (v testni specifikaciji) definiranih testnih primerov je zadolžena testna ekipa, ki mora zabeležiti rezultat testiranja - informacijo o (ne)uspešnosti testa, opis odkritih napak. Zapise (testne specifikacije in rezultate) je potrebno hraniti in morajo biti vedno dostopni vsem članom testne in razvojne ekipe.

5) Sledi priprava testnega okolja. Testno okolje je odvisno od arhitekture in od tehničnih ter funkcionalnih zahtev projekta. Projektni vodja določi, kdo je odgovoren za postavitev testnega okolja. Pri postavitvi testnega okolja (strojna in programska oprema) so dolžni sodelovati (pomagati) vsi člani projektne skupine. Ko je testno okolje postavljeno, je zanj odgovorna testna ekipa.

6) Nato sledi objava rezultatov o izvedbi testov enot. Razvojni inženirji objavijo, da so na projektu (ali na njegovem določenem delu – modulu) končali z testi enot. Podajo informacijo o dodatni oziroma morebiti spremenjeni funkcionalnosti in dajo skupaj komponente programa (ang. build) za instalacijo v testno okolje.

7) Potem pride na vrsto določitev testnih primerov. Testna ekipa izbere, določi oz. po potrebi dopolni testne primere, ki bodo izvedeni. Dejansko pripravi načrt izvedbe testiranja (poleg testnih primerov, izbere še izvajalce – testerje, določijo roke, …).

8) Sledi instalacija. Testna ekipa (po navodilih za instalacijo) v testno okolje instalira pripravljeno programsko opremo. Če navodila za instalacijo niso predmet testiranja, potem programsko opremo (ali njene dele), lahko v testno okolje naložijo tudi razvojni inženirji.

9) Nato testni inženirji izberejo ali naredijo pomožna orodja. Razvojni inženirji lahko pripravijo oziroma pomagajo pri pripravi pomožnih orodij (SQL procedure, poročila, …), ki jih testni inženirji potrebujejo pri testiranju.

10) Sledi Izvedba testnih primerov. Testna ekipa prične s sistematičnim izvajanjem testnih primerov. Sproti beleži rezultate in zabeleži vse najdene napake.

11) Obveščanje o napakah. Pred prvo predajo produkta naročniku testni inženir o najdeni napaki obvesti razvojnega inženirja, odgovornega za (pod)modul, v katerem je bila napaka najdena.

Pred in po predaji naročniku se vsaka napaka zabeleži s pomočjo orodja, ki omogoča nadzor in upravljanje nad zahtevami za spremembo programske opreme (zaradi odkritih napak ali novo definirane funkcionalnosti). V tem primeru je projektni vodja (ali vodilni inženir) tisti, ki določi, kateri razvojni inženir bo odpravljal« napako.

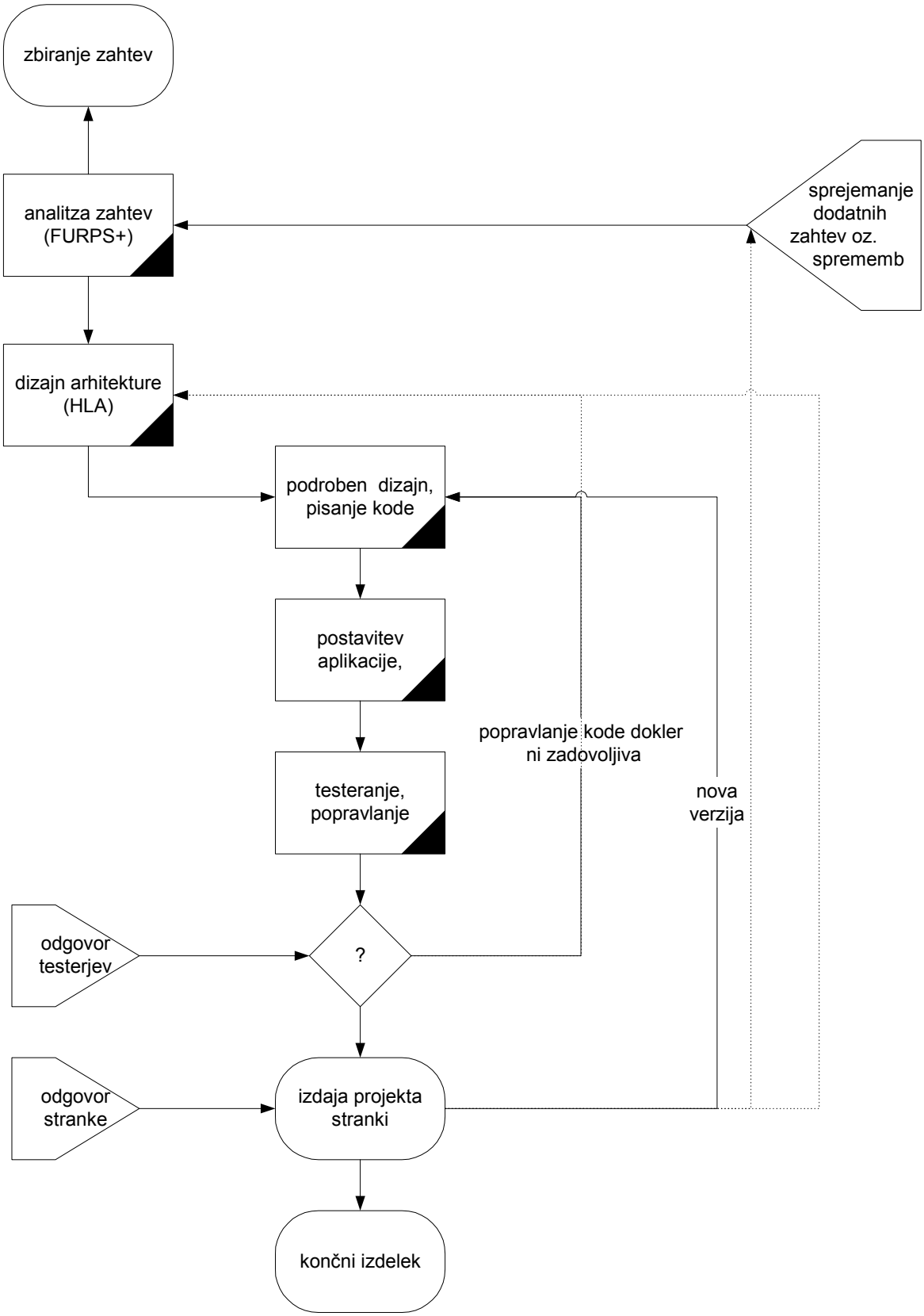
12) Testna ekipa pripravi poročilo o rezultatih testiranja. V njem ugotovi, ali so rezultati v mejah, ki določajo sprejemljivost produkta.

13) Če produkt ustreza zahtevam sledi predaja naročniku.Projektni vodja potrdi poročilo in odloči, ali se produkt oz. modul izdelek posreduje naročniku ali ne, če produkt ustreza zahtevam sledi predaja naročniku.

14) Če je med testiranjem ugotovljeno preveliko število napak (torej niso izpolnjeni pogoji za sprejemljivost produkta), se testni cikel (potem ko razvijalci odpravijo napake) ponovi od vključno točke 6) naprej. Poleg ponovitve testov, ki so se v prejšnjem ciklu končali neuspešno, je potrebno ponoviti tudi teste, ki so se končali uspešno oziroma izvesti ustrezne avtomatske teste. Cikel testiranja se ponavlja toliko časa, dokler število napak (glede na posamezne kategorije napak) ni v naprej določenih mejah.

Diagram poteka aktivnosti v razvoju projekta poglejmo na sliki 3.2.

Slika 3.2: Diagram poteka aktivnosti skozi razvoj projekta.




Vir: URL: http://grunf.hermes.si/QualityManual/release2/

### *3.1.1. Faza zahtev* (faza 0 )

V tej fazi (slika 3.3) se zbirajo predvsem zahteve, ki jih postavlja stranka. Zapišejo se zahteve v dokumente, ki jih mora stranka odobriti. To so predvsem zahteve po funkcionalnosti, uporabnosti in performanci programa. Testni inženir napravi koncept kako bo projekt testiran. Menedžer določi projektnega vodjo in razdeli vloge v projektu. Testni inženirji izvedejo korak1, ki je bil opisan v uvodu poglavja.

Slika 3.3:Diagram poteka aktivnosti v fazi zahtev.


Vir:URL:http://grunf.hermes.si/QualityManual/release2/

### *3.1.2 Faza definicij (faza 1)*

V tej fazi (slika 3.5) se za vsako zahtevo, za katero sta se stranka in podjetje dogovorila, najde metoda, kako jo sprogramirati. Zapišejo se orodja in tehnike programiranja, ki bodo uporabljene ter arhitektura programa. Po dogovoru se ločijo večje in manjše zahteve, to so tiste, ki jim programska oprema obvezno mora zadoščati, da program sploh deluje in tiste, ki niso glavnega pomena. Zato se izdela kriterij (ocena) za velikost napake. Testni inženirji izvedejo koraka 2 in 3, opisana v uvodu poglavja.

Definicija vsakega testnega primera vsebuje opis postopka in pričakovanih rezultatov. Testiranje ne uspe, če je rezultat izvedenega testnega primera različen od pričakovanih rezultatov.

Neuspešne teste ločimo glede na resnost napake na 5 kategorij. Poglejmo jih v tabeli 3.4.

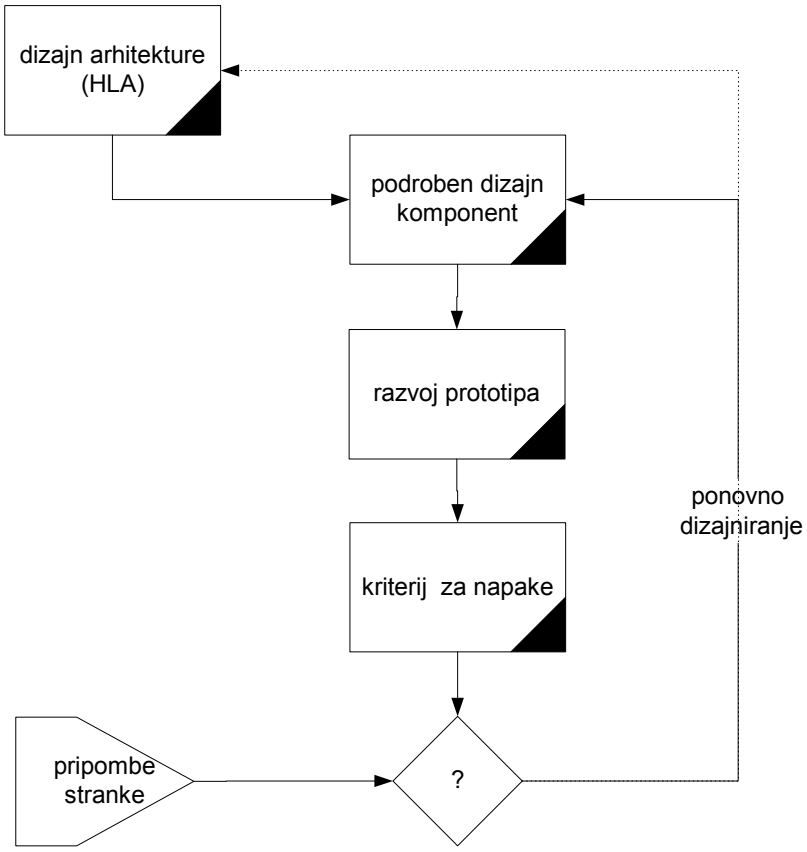
Tabela 3.4: Tabela stopenj kritičnosti napak.

| **Kategorija** | **Opis napake / problema** |
|---|---|
| 1 | Kritična napaka: produkt je povsem neprimeren za uporabo |
| 2 | Resna napaka: pravi obhod napake (ang. workaround) dejansko ne obstaja |
| 3 | Manjša napaka: obhod napake obstaja |
| 4 | Opozorilo (ang. warning) |
| 5 | Ni napaka, le predlog za izboljšavo |

Vir: URL: http://grunf.hermes.si/QualityManual/release2/

Kaj dejansko pomeni vsaka posamezna kategorija je za konkretni predmet testiranja definirano v testni specifikaciji, ki se nanaša na ta predmet testiranja. Opis napak pa se malce razlikuje glede na vrsto aplikacije, ki jo testiramo.

Slika 3.5 Diagram poteka aktivnosti v fazi definicij.

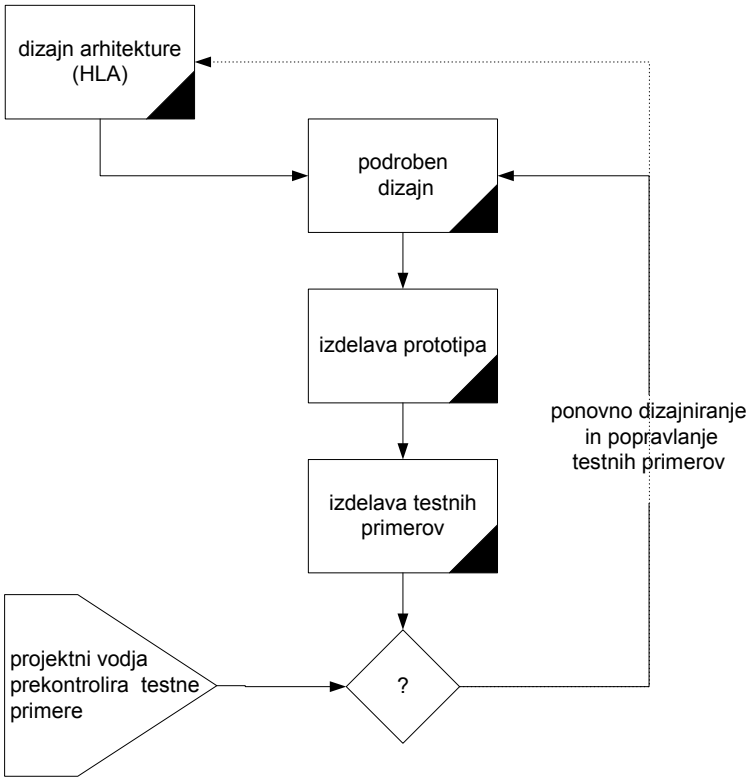



Vir: URL: http://grunf.hermes.si/QualityManual/release2/

### *3.1.2 Specifikacija (faza 2)*

V tej fazi nadaljujemo vse začete aktivnosti. Vse plane testiranja se specificira do te mere, da jih stranka lahko potrdi. Za vsako zahtevo se zapiše testni primer. Revidira se vso napisano dokumentacijo. Izvede se korak 4 in 5.

Slika 3.6: Diagram poteka aktivnosti v fazi specifikacij.




Vir: URL: http://grunf.hermes.si/QualityManual/release2/

Preden izdamo novo verzijo produkta morajo biti, ne glede na predmet testiranja, vse napake kategorije 1-3 odpravljene. Ob izdaji produkta je dopustnih največ 20% neuspešnih testnih primerov 4. kategorije ali največ 30% neuspešnih testnih primerov 5. kategorije.

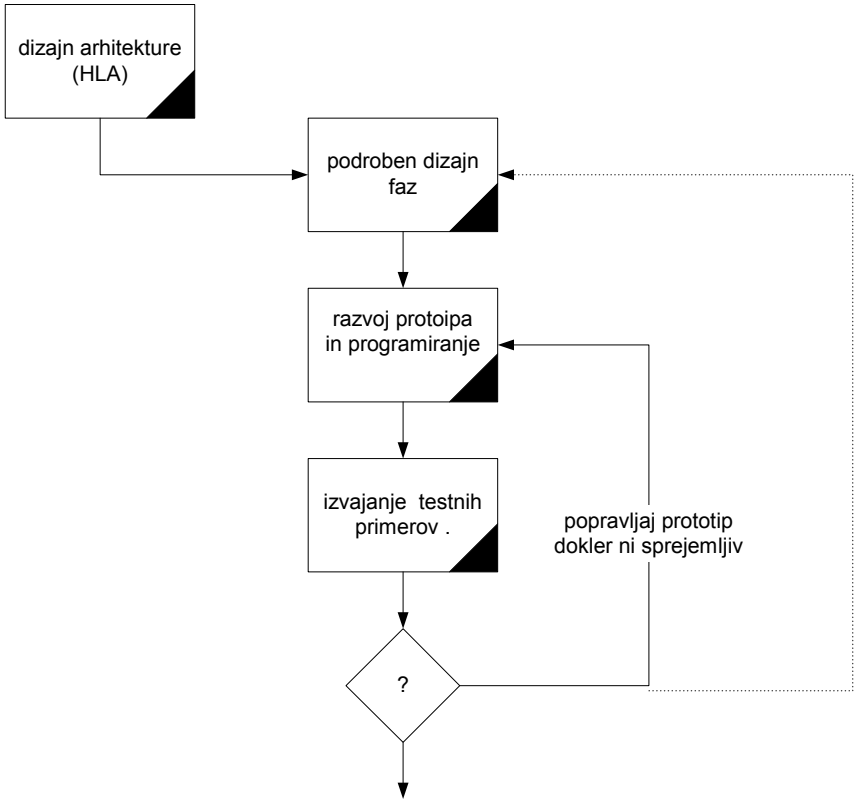
### *3.1.3 Faza implementacije (faza3)*

Na sliki 3.7 vidimo aktivnosti v fazi implementacije. Vse plane testiranja razvijejo še do višjega nivoja. Razvijalci programirajo, testira se vsaka posamezna enota programa, ki je pripravljena za integracijo v končni program. Vsak modul je stestiran posebej, da se zagotovi stabilnost in preveri planirana funkcionalnost. V tej fazi opravimo tudi

test celotnega produkta, da ugotovimo, če produkt res zadošča zahtevam pogodbe. V tej fazi testni inženirji izvedejo korake 6-11 iz uvoda poglavja.

V primeru neuspešnega testnega primera v splošnem nadaljujemo z izvajanjem naslednjega testnega primera. V primeru napak 1. in 2. kategorije pa navadno izvajanje naslednjih testnih primerov sploh ni možno. Po odpravi take napake je potrebno ponoviti vse do tedaj uspešne testne primere (bodisi ročno, bodisi izvesti ustrezne avtomatske teste).

Slika 3.7: Diagram poteka aktivnosti v fazi implementacije.



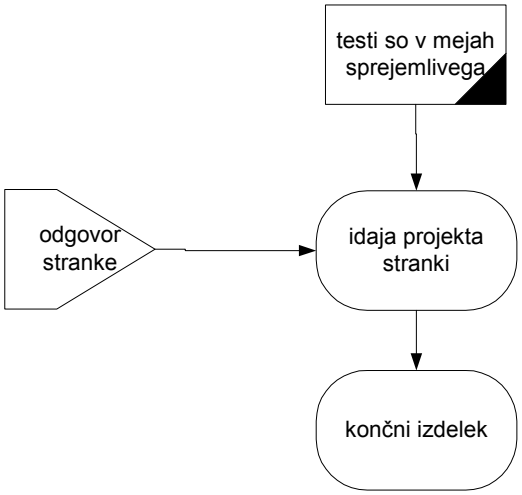

Vir: URL: http://grunf.hermes.si/QualityManual/release2/

### *3.1.4 Faza odobritve (faza 4)*

Faza odobritve je faza, ko stranka sprejme produkt. Na sliki 3.8 je prikazano zaporedje aktivnosti v tej fazi. V njej se zberejo vsa poročila iz katerih ugotovimo, ali programska oprema res deluje, kot je bilo zahtevano (ang. acceptance test). Če gledamo s področja testiranja, v

tej fazi izvedemo korake 12-14 navedene v uvodu poglavja. Pripravimo tudi testna poročila za stranko.

Slika :3.8 Diagram poteka aktivnosti v fazi odobritve.




Vir:URL:http://grunf.hermes.si/QualityManual/release2/

## *3.2 Življenjski cikel hrošča*

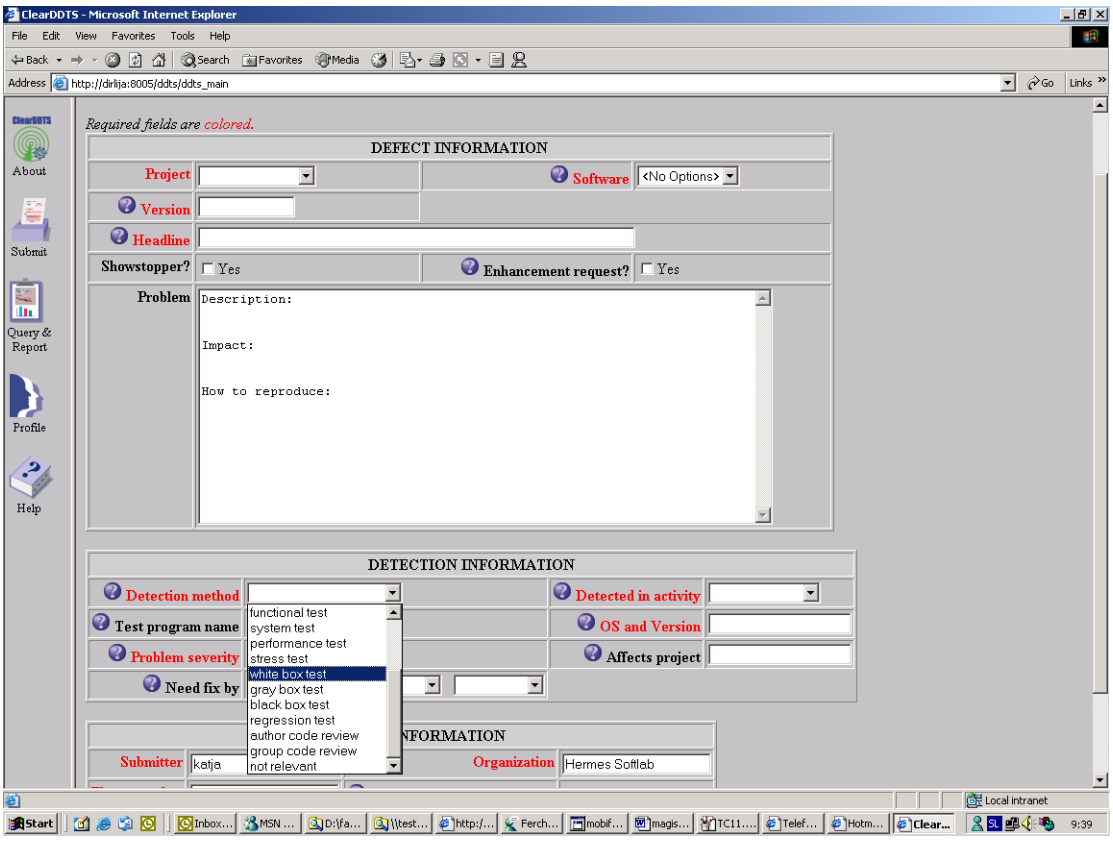
Življenjski cikel hrošča se začne nekako v fazi implementacije. Pred to fazo so testni inženirji že pripravili testno dokumentacijo v skladu z zahtevo stranke. Testni inženirji začnejo po pripravljenih testnih primerih izvajati testiranje. Pomembna je kvaliteta teh testnih primerov, vodenje njihovega izvajanja ter zapisovanje njihovih rezultatov.

Pomembno je zapisovanje napak za posamezno verzijo izdelane programske opreme in njihovo arhiviranje. Seveda pa je enako pomembno tudi arhiviranje in označevanje verzij produkta.

Nujno je, da najdeno napako vpišemo v orodje za sledenje napakam. To orodje mora omogočati arhiviranje napak in ne sme pustiti zbrisati napake iz svoje baze. V vsakem trenutku mora imeti podatek kakšen je status napake in kdo se ukvarja z njo. Takšno orodje je na primer DDTS (ang. distributed defect tracking system), programska oprema podjetja *Rational Software* za zasledovanje hroščev.

Ko je napaka odkrita jo testni inženir vpiše v orodje za sledenje napak (glej sliko 3.9). V tem trenutku je potrebno določiti inženirja, ki bo zadolžen za odpravo te napake. Ponavadi ga določi testni inženir ali pa vodja projekta. Napaka spremeni status iz »nov« (ang. new) v status »zadolžen« (ang. assigned). Kdo je ta programer, ki je odgovoren za napako, je pomembno zapisati, saj se lahko skozi več verzij programa ta podatek pozabi ali pa izgubi, kar ima lahko za posledico nerešeno napako.

Slika 3.9: Vpis odkrite napake v orodje DDTS.



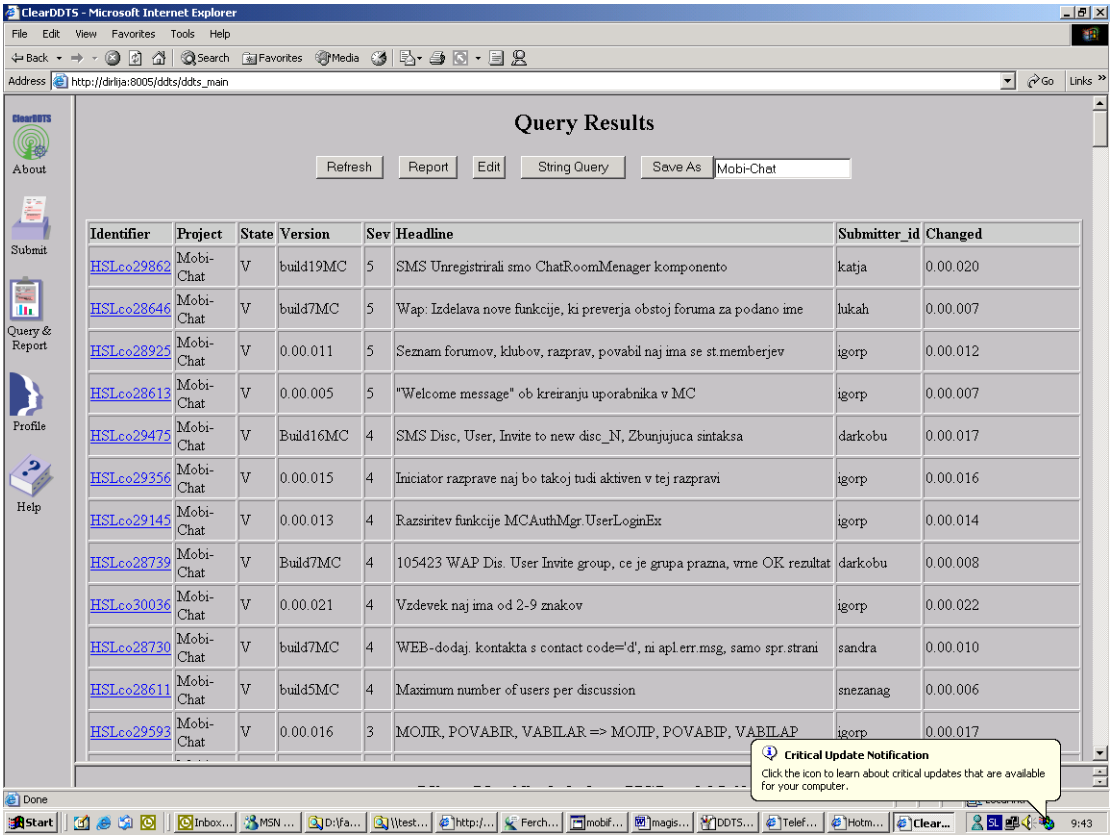

Vir: URL: http://dirlija.hermes.si:8005/ddts/ddts_main

Razvijalec nato pregleda napako in oceni situacijo. Lahko sprejme napako in jo začne reševati ali pa jo zaradi različnih razlogov ovrže.

Če razvijalec potrdi napako, jo mora odpraviti in javiti testnemu inženirju, ki je našel napako, ali je bila napaka res odpravljena. Napaka dobi status »rešena« (ang. resolved), ko pa testni inženir potrdi, da je napaka odpravljena dobi status »preverjeno« (ang. verified). Nato testni inženir zaključi krog napake.

Tudi razvijalec uporablja orodje DDTS za sledenje napak. Menedžer lahko v vsakem trenutku pogleda, kakšno je stanje napak in njihova kritičnost v določenem projektu. Glej sliko 3.10.

Slika 3.10: Rezultati vseh vpisanih napak v orodje DDTS.

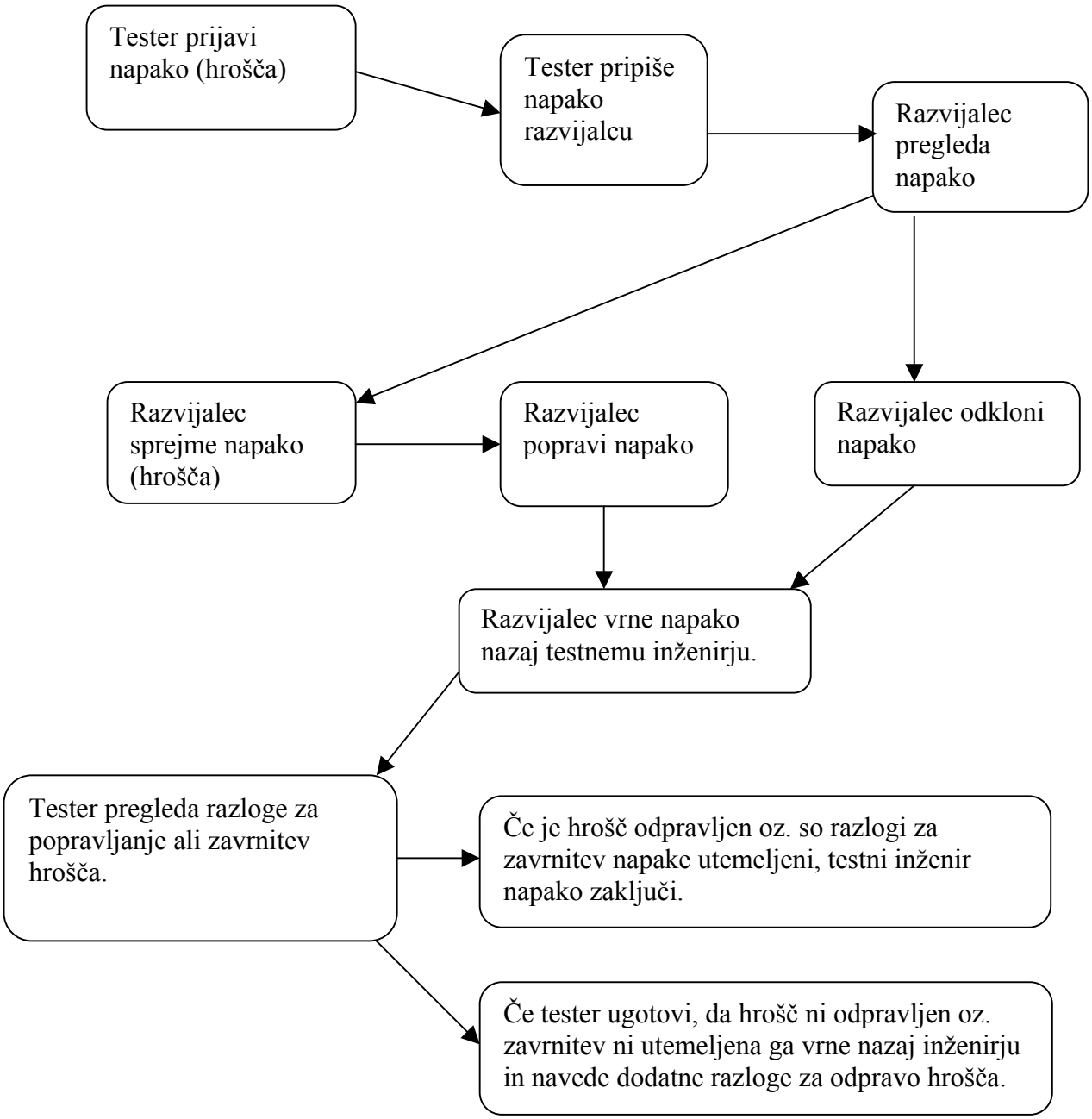



Vir: URL: http://dirlija.hermes.si:8005/ddts/ddts_main

Če se razvijalec odloči, da prijavljena napaka ni v njegovi domeni oz. da to sploh ni napaka ga da drugemu inženirju oziroma ga vrne testnemu inženirju z navedenim vzrokom zavrnitve. Ti so lahko: to ni problem dizajna, napaka se ne da reproducirati. Napaka v tem primeru dobi status »zaprta« (ang. closed). Napaka je lahko dvojnik kakšne druge napake. V tem primeru dobi status podvojena (ang. duplicated).

Nikoli pa se ne da nobene napake izbrisati in vedno lahko preverimo v kakšnem stanju je napaka ter v kateri verziji je bila odpravljena. Pogledamo lahko tudi celotno zgodovino napake.
Zaporedje aktivnosti v življenjskem ciklu hrošča predstavljamo na sliki 3.11.

Slika 3.11: Diagram aktivnosti v življenjskem ciklu hrošča.




Vir: HERMES SoftLab Business Manual, 2000

# 4. Metodologija testiranja, orodja in tehnike

V tem poglavju predstavljamo pomembnost metodologije in na kaj moramo biti pozorni pri razvoju le-te. Opisujem tehnike metode in orodja potrebna za testiranje. Metodologija testiranja je sredstvo, s katerim zagotovimo, da je testna strategija dosežena. Skupina testnih inženirjev, ki razvija metodologijo, definira teste in metode ter njihove pričakovane rezultate, ki zagotovijo odpravo rizika definiranega v testni strategiji.

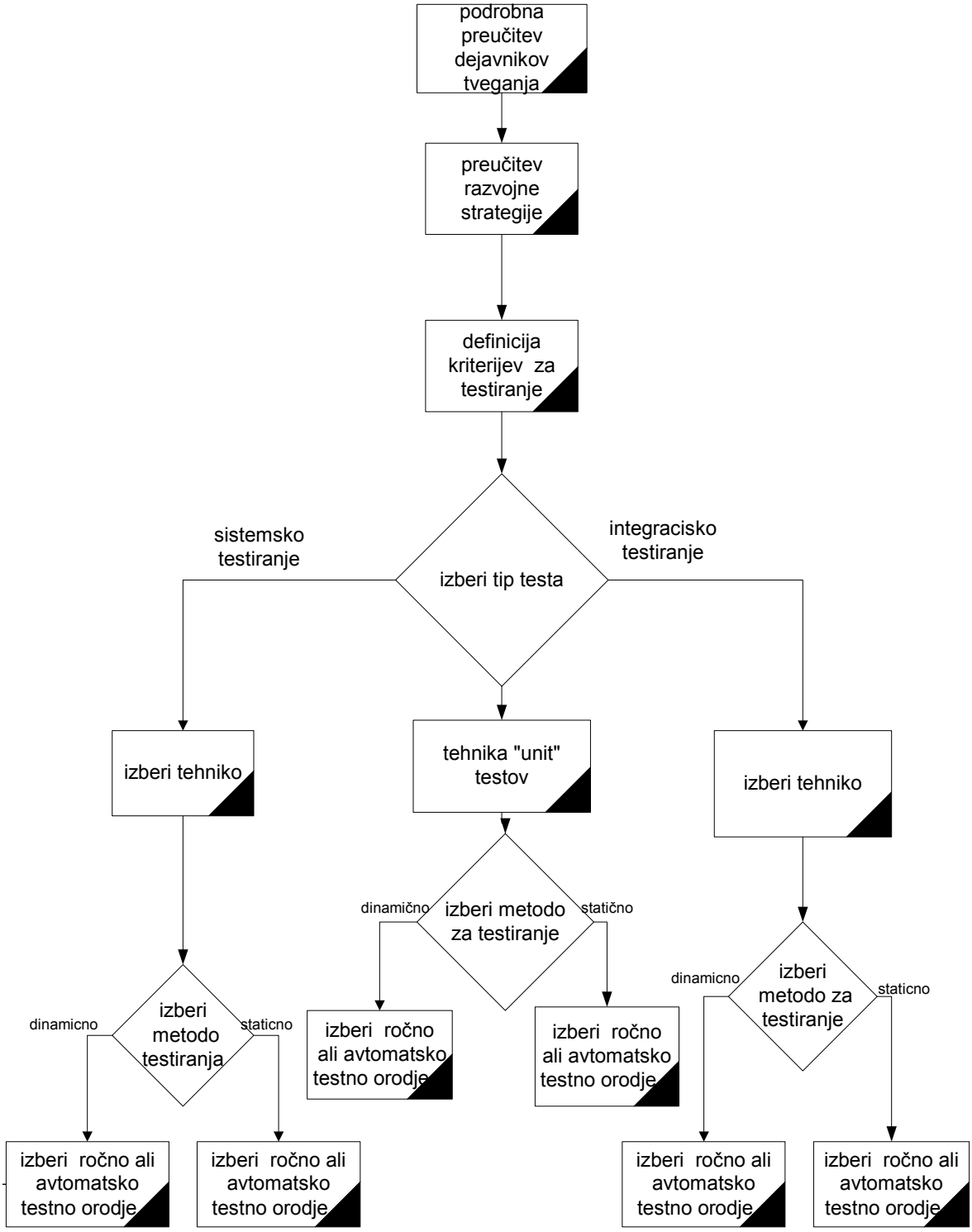
V zadnjih dvajsetih letih je bilo na razvojnem procesu vloženega veliko truda. Testiranje je v večini organizacij postalo neizogibno. Razvite so bile nove in nove metode za povečanje produktivnosti in efektivnosti testov.

Kakšna je torej razlika med testnim orodjem in tehniko? Orodje je sredstvo, s katerim izvajamo testni proces, toda samo po sebi brez tehnike nezadostno. Enako velja za kladivo, ki služi zabijanju žebljev. Kladivo je orodje, vendar brez tehnike kako ga držati in kaj delati z njim je brez vrednosti. Tehnika je proces, ki zagotavlja, da je določen pogled na aplikacijo ali njeno enoto primeren.

Metode testiranja lahko razdelimo v dinamične in statične. Dinamična metoda je tradicionalna in zahteva, da program teče med tem ko se na njem izvajajo določeni testi, katerih rezultat je ugotovitev, ali aplikacija dela pričakovano. Statična metoda pa vsebuje tehnike, ki ne zahtevajo izvajanje samega programa, ki je predmet testiranja ampak imajo nalogo testiranje sintakse. Uporablja se predvsem v fazi zahtev in fazi dizajna.

Testnih tehnik je malo, orodij za izvajanje teh tehnik pa več. Za testnega inženirja je pomembno, da najprej razume tehniko, nato pa izbere in spozna primerno orodje. Orodje za testiranje bi moralo biti izbrano glede na njegove sposobnosti, kako ustrezajo testnim tehnikam in metodam. Poglejmo si (slika 4.1) kako naj bi izbirali testno orodje glede na poznano tehniko in metode. Graf je neodvisen od življenjskega cikla razvoja projekta.

Slika 4.1 Grafični prikaz časovnega poteka izbiranja testne metode tehnike in testnega orodja.




Vir: Perry, 2000

Na tem mestu opozarjamo na kaj je potrebno biti pozoren pri prenosu testne strategije v taktiko testiranja in razvoj testnega plana, ki mu sledi dnevno testiranje in izgradnja okolja za razvoj testnih taktik. Nekaj metod oz. taktik na katere moramo biti pozorni pri razvoju metodologije testiranja, je (Perry, 2000):

1. Dobiti moramo testno strategijo in jo preučiti. Testno strategijo ponavadi razvije skupina, ki je seznanjena z tveganjem, povezanim z programsko opremo, taktike pa razvija testna ekipa. Na tej točki bi se morali vprašati, kateri je pomembnost določenega faktorja testiranja, kateri rizik je najbolj pomemben ter kakšne so posledice, če se program ne obnaša pravilno in če ni končan v roku.

2. Ugotoviti je potrebno tip razvojnega projekta. To pomeni okolje in metodologijo v kateri bo programska oprema razvita. Tako kot so različni razvojni projekti, mora biti različen tudi pristop testnih inženirjev.

3. Ugotoviti je potrebno tip računalniškega sistema z vsemi njegovimi značilnostmi.

4. Spoznati je potrebno obseg projekta. S tem mislimo aktivnosti, vključene v projekt, zahteve sistema in razumevanje vseh specifikacij. Če gre le za spremembo obstoječega sistema, moramo spoznati to spremembo. Obseg projekta naredi običajno zasnovo za obseg testiranja. Pod to točko spada tudi ugotovitev, kako bo nova aplikacija vplivala na že obstoječ sistem. Ga bo samo izboljšala?

5. Ugotoviti je potrebno vsa možna tveganja. O tveganju smo precej govorili že v prvem poglavju. Pretehtati je potrebno kakšne so možnosti, da se določeno tveganje zgodi, in oceniti naše sposobnosti, da določeno tveganje stestiramo.

6. Določiti moramo začetek testiranja, kdaj v kateri fazi razvoja projekta se bo zgodila kakšna izmed testnih aktivnosti. En primer testnih aktivnosti skozi razvoj projekta smo navedli že v poglavju 3.1. Naj navedemo še enkrat vse aktivnosti, ki so vpletene v dobro metodologijo:

A. Aktivnosti v fazi zahtev:
    - razvoj testne strategije,
    - ugotovitev zadostnosti zahtev v specifikaciji,
    - ustvariti pogoje za test funkcionalnosti.

B. Aktivnosti v fazi definicij:
    - določiti konsistenco dizajna z zahtevami,
    - ugotoviti zadostnost dizajna,
    - ustvariti pogoje za ostale sistemske teste

C. Aktivnosti v fazi implementacije:
    - Ugotovitev konsistence z dizajnom,
    - generirati testne pogoje za posamezne dele projekta.

D. Aktivnosti faze odobritve:
    - ugotovitev adekvatnosti testnega plana,
    - testiranje aplikacije

E. Instalacijske aktivnosti:
    - Postavitev testnega sistema v produkcijo.

F. Aktivnosti v vzdrževalni fazi:
    - popravljanje in dodajanje testnih primerov
    - ponovno testiranje

7. Narediti je potrebno sistematični testni plan. Taktičen plan testiranja mora vsebovati opis kdaj in kje bo testiranje izvedeno. Vsebovati mora objektivno oceno tveganj in podroben opis testov (testnih primerov), ki bodo izvedeni. Testni plan je kot zemljevid, ki mu v testnih ciklih sledimo. Po izvedbi teh testnih primerov se rezultati zberejo in naredi se poročilo testiranja.

8. Skozi interni dizajn je sistem razdeljen v več komponent ali enot, ki skupaj predstavljajo celotni program. Vsaka od teh enot bi morala imeti svoj lastni testni plan. Ta je lahko enostaven ali pa kompleksen, odvisno od organizacije in njenega vrednotenja kvalitete. Pomembno je, da v testnem planu določimo, kdaj je ta končan, da se lahko začne integracijsko testiranje. Nima smisla, da v celoto sestavljamo enote, ki vsebujejo napake, saj se potem pojavi ogromno napak in za odkritje v kateri komponenti je napaka je

potreben čas. Z dobrimi testi enot lahko zelo zmanjšamo stroške testiranja.

Ko smo spoznali metode, potrebne za testiranje, jih lahko združimo in s pridom uporabimo za razvoj testnega plana. Čas, porabljen za planiranje se odraža v bolj kvalitetnem testiranju. Ponavadi se 1/3 vsega časa, namenjenega testiranju, porabi za planiranje.

## *4.1 Metode testiranja programske opreme*

Po teoriji lahko metode testiranja klasificiramo na 2 načina. Pri prvi klasifikaciji je pomembno, kako so testi generirani, pri drugi klasifikaciji pa je pomembno poznavanje projekta.

### *4.1.1 Klasifikacija metod, osnovana na načinu generiranja testov*

Razno razni faktorji vplivajo na to kako so testi generirani. Ti faktorji so intuicija, predhodno znanje, specifikacija, struktura kode, narava aplikacije, predhodno odkriti problemi. Glede na to ločimo (Perry, 2000):

- Metodo »Ad hoc« (lat. iz viška), ki se zanaša samo na intuicijo in sposobnosti testnega inženirja, ter izkušnje s podobnimi programi. Uporabnost te metode lahko zelo varira. Učinek je lahko velik, če je testni inženir resnično strokovnjak ali pa zanemarljiv, če mu primanjkuje izkušenj.
- Metoda pretehtavanja vseh možnosti. Testne primere generiramo na osnovi vseh možnih kombinacij logičnih relacij med pogoji in funkcijami.
- Metoda mejnih vrednosti. Testni primeri se generirajo na osnovi mejnih vrednosti podatkov.
- Naključno testiranje. S to metodo preizkušamo delovanje funkcij z naključnimi vrednostmi.

### *4.1.2 Klasifikacija metod testiranja na osnovi razumevanja implementacije projekta*

Glede na potrebnost razumevanja same implementacije projekta ločimo (Kung,1998):

- Metodo črne škatle (ang. black-box), ki ne zahteva poznavanje same implementacije in interne logike programa. Zahteve le poznavanje potrebne funkcionalnosti. Na osnovi tega znanja se generirajo testni primeri. Definirajo se vhodni podatki, in pričakovani rezultati. Vmesno dogajanje pa je črna škatla. Ta metoda vključuje testiranje mejnih vrednosti in testiranje funkcionalnosti.
- Metodo bele škatle (ang. white-box) ponekod imenovane tudi steklene škatle (ang. glass-boxs) pa zahteva poznavanje interne logike programa, na osnovi katre lahko razvijemo hipotetične testne primere. Sem lahko uvrščamo metodo ugibanja napak, naključno testiranje in še kaj.
- Tretja možnost je kombinacija zgornjih dveh.

## *4.2 Tehnike*

V tem trenutku je potrebno opozoriti, da se za izraz »tehnike testiranja« uporablja več različnih stvari. Mnogi mešajo in za izraz »tehnika« uporabljajo spodaj navedena orodja. Orodja za testiranja lahko pokrivajo zelo veliko množico aktivnosti in so primerna za najrazličnejše tehnike in jih je res ogromno. V literaturi je lahko zaslediti menjavo pojmov testne metode in testne tehnike. Mogoče včasih tudi opravičeno. Mi bomo za izraz tehnike testiranja uporabii tehnike opisane že v drugem poglavju. Te razdelim v tri sklope (1) sistemske strukturne testne tehnike, (2) sistemske integracijske ( funkcionalne) testne tehnike, (3) tehnike testiranja enot (ang. unit test tehnique). Vsaka izmed testnih tehnik pokriva en ali več testni faktorjev (tveganje, ki se mu lahko z testiranjem izognemo) in ime tehnike izhaja iz tveganja, ki ga pokriva. Tako tehnika testiranja varnosti pokriva tveganje oziroma rizik varnosti.

### *4.2.1 Tehnika testiranja funkcionalnosti*

Testi funkcionalnosti zagotavljajo, da je zahtevam stranke, ki so navedene v dokumentu ustrezno zadovoljeno. Pri testiranju funkcionalnosti nas ne zanima kako sistem deluje, ampak le rezultat delovanja. Pri določenih vhodnih podatkih moramo dobiti določen odgovor funkcije. Ta vrsta testiranja je ena izmed lažjih, saj ni potrebno poznavanje logike programa. Je pa pomembno, da pred uporabo ostalih tehnik zagotovimo, da je program operativen. Zagotovljeno mora biti politiki organizacije in nekaterim splošno znanim pravilom.

Ta tehnika zahteva primarno kreiranje testnih pogojev in listo zahtev funkcionalnosti. Testne primere generiramo, ko so zahteve dovolj dobro specificirane.

Če pripravimo testne primere za testiranje funkcionalnosti direktno iz uporabniških zahtev, je lahko to testiranje bolj produktivno, saj se v testnih primerih, pripravljenih po uporabniški dokumentaciji lahko skrijejo napake, ki so bile narejene že ob pisanju dokumentacije in jih ob izvajanju testov ne bomo odkrili.

### *4.2.2 Tehnika testiranja uporabnosti*

Dogaja se, da ob brskanju po spletu pridemo na strani, razno razne portale, ki imajo nelogično navigacijo. Tako ne moremo nazaj na osnovno stran, po vpisu določenih podatkov pa pridemo na kakšno drugo stran, ki si jo ne želimo. Skratka, s klikanjem lahko sicer pregledamo strani in jih uporabimo, vendar bi do določenih strani želeli priti z veliko manj navigiranja.

Ti tipi testov obsegajo kontrolo enostavnosti uporabe aplikacije. Test uporabnosti se lahko izvaja na različne načine kot so raziskave, opombe svetovalcev in testi v produkcijskem okolju. Glavni namen testa uporabnosti je zagotoviti enostavno uporabo in logično navigiranje. Ta tehnika vsebuje zapisovanje in izvajanje testnih primerov, ki zagotavljajo uporabnost programa.

### *4.2.3 Tehnika testiranja kompatibilnosti*

Izkušnje kažejo, da uporabljamo aplikacije na različnih operacijskih sistemih in različnih verzijah le-teh. Pomembno je tudi celotno okolje v katerem aplikacija teče. Veliko zmedo naredijo tudi brskalniki, ki jih je ogromno, še več pa njihovih verzij.

Testi kompatibilnosti so lahko izvedeni v testnem laboratoriju, kjer so računalniki z različnimi verzijami operacijskih sistemov in različni brskalniki.

Pri testiranju na različnih operacijskih sistemih moramo paziti na različne brskalnike. Pri mobilnih aplikacijah moramo obravnavati različne telefone, saj imajo vsi svoje določene lastnosti. Pri drugih vrstah aplikacij pa so spet druge lastnosti, na katere moramo biti pozorni. Ta tehnika vsebuje zapis testnih primerov, ki kažejo, katere kombinacije programske opreme smo pokrili; z njimi pokažemo, da sistem deluje v vseh zahtevanih okoljih. Včasih pa se zadovoljimo s testiranjem na najnižji in najvišji verziji programske opreme in pričakujemo, da v vmesnih verzijah ne bo večjega odstopanja.

### *4.2.4 Tehnika preizkusa učinka*

Prepočasen odziv aplikacije lahko kljub popolni funkcionalnosti povzroči neuporabnost aplikacije (Ireson, 1988).

Tehnika testiranja učinka pomeni zapisovanje in izvajanje testov, ki zagotavljajo, da sistem ustreza dogovorjeni performanci. V tem sklopu so šteti testi hitrosti izvajanja statičnega in dinamičnega procesiranja, hitrost izvajanja transakcij…

### *4.2.5 Tehnika testiranje zanesljivosti*

Pod zanesljivost štejemo ponoven zagon aplikacije (ang. restart), vrnitev v stanje, ki je bilo pred začetkom procesiranja nedokončanih transakcij.

Test zanesljivosti lahko vsebuje izgubo kapacitete spomina, izgubo komunikacijske linije, poškodbe strojne opreme in odpoved

operacijskega sistema. Nemogoče je testirati vse aspekte zanesljivosti. Tehnika vsebuje zapis in izvajanje testnih primerov glede zanesljivosti aplikacije.

Test zanesljivosti bi se moral izvajati vedno kadar je kontinuiteta operacij nujna za normalno delovanje programa.

### *4.2.6 tehnika testiranja varnosti*

Razvoj tehnologije prinaša vedno nove in nove probleme, tako da je zajeti vse praktično nemogoče.

Razmisliti moramo, kako zagotoviti varnost in z »ad hoc« metodami poskusiti doseči neavtenticirano uporabo aplikacije. Zapisati pa je potrebno čim več testnih primerov, ki zagotavljajo varnost aplikacije. Uporabiti je potrebno vso svoje znanje in domišljijo. Če je v aplikaciji več takih mest, kjer lahko varnost odpove, je smiselno vplesti v testiranje strokovnjake s tega področja.

### *4.2.7 Obremenitvena tehnika*

Obremenitve se lahko pojavijo pri povečanem številu transakcij, pomanjkanju prostora na disku, komunikaciji med računalniki ... Če se bo aplikacija pod obremenitvenimi testi obnašala primerno, lahko pričakujemo, da se bo tudi ob normalnih pogojih obnašala točno. Premislimo, kje v naši aplikaciji bo prišlo do obremenitev, in zapišimo ter izvedimo teste, ki simulirajo obremenitve.

Objektivnost obremenitvenih testov dosežemo s simulacijo produkcijskega okolja, v katerem lahko simuliramo običajno število transakcij z možnostjo procesiranja velikega števila podatkov. Imeti moramo dovolj sredstev, da simuliramo komunikacijsko linijo in možne časovne zakasnitve.

To tehniko uporabimo vedno, kadar obstaja dvom o količini podatkov, ki jih aplikacija lahko obravnava brez napake. Obremenitveni testi poskušajo zlomiti sistem z preobremenitvijo, velikim številom transakcij. Slaba stran obremenitvenih testov pa je čas za pripravo testov in sredstva, porabljena za izvajanje le-teh.

## *4.3 Orodja za testiranje*

V zadnjih dvajsetih letih je bilo na področju razvoja procesa narejenega veliko in tako je v večini organizacij postalo testiranje neizogibno. Hitrost izvajanja testov pa lahko Razvite so bile nove in nove metode za povečanje produktivnosti in efektivnosti testov.

Izbira pravega orodja za testiranje je pomemben korak v testnem procesu. Orodje za testiranje bi moralo biti izbrano glede na njegove sposobnosti, kako ustrezajo testnim tehnikam in metodam. Poglejmo, kako naj bi izbirali testno orodje glede na poznano tehniko in metode (slika 4.1). Poznati moramo tehnike, metode in šele na podlagi tega se odločimo za orodje.

Orodja za testiranja lahko pokrivajo zelo veliko množico aktivnosti in so primerna za najrazličnejše tehnike. Za uporabo testnih orodij pa so potrebne različne izkušnje, pa tudi cena uporabe določenih orodij lahko zelo varira. Včasih so potrebne izkušnje, ki zahtevajo visoko tehnično znanje in znanje programiranja, spet druga pa so zelo splošna in so uporabna za vsakega, ki bi rad testiral aplikacijo.

Naj naštejem nekaj orodij, da spoznamo repertoar možnih orodij (Perry, 2000, str.135-185):

1. **Kriterij sprejemnega testa** razvije standard, ki mu mora sistem ustrezati, še preden ga lahko damo v produkcijo.

2. **Analiza mejnih vrednosti.** Z metodo ločevanja aplikacije v segmente preverja mejne v vrednosti posameznih in jih sestavlja v celoto.

3. **Grafični prikaz efekta.** Grafično prikažemo efekt vsake funkcije z namenom, da bi zmanjšali število testnih pogojev.

4. **Kontrolni seznam** je vnaprej narejen seznam vprašanj, narejenih za preverjanje različnih funkcij in grafov.

5. **Primerjava kode** odkriva razlike med dvema različnima verzijama programa. Lahko ga uporabimo za objekte ali pa samo kodo.

6. **Prevajalnikove analize.** Izkorišča diagnostiko prevajalnika. Temu doda razne rutine, da lahko identificira napake že med samim prevajanjem kode.

7. **Test kompleksnosti** uporablja statistiko, da razvije relacije, ki napovejo kompleksnost procesa in kdaj bo testiranje lahko končano.

8. **Preverjanje/potrditev.** Preveri pravilnost večih pogledov na sistem in preveri obstoj dokumentov.

9. **Analiza kontrole poteka** Zahteva razvoj grafične predstavitve programa z namenom analizirati logične probleme.

10. **Dokazovanje popravkov** vsebuje množico stavkov in hipotez, ki definirajo popravke procesiranju. Te hipoteze potem preverimo, da se ugotovimo, če ustrezajo popravkom.

11. **Podatkovni slovar** vsebuje podatke in generira teste, ki verificirajo delovanje programa glede na te podatke.

12. **Analiza toka podatkov** odkrije definirane podatke, ki bi morali biti uporabljeni v programu in uporabljene podatke, ki niso definirani.

13. **Testiranje funkcionalnosti na osnovi dizajna** ovrednoti atribute funkcij glede na zahteve dizajna.

14. **Analiza programske logike** Po končanem programiranju analiziramo ustreznost programske logike.

15. **Test razdejanja** simulira napake operacijskega sistema, da ugotovi, če se bo po napaki sistem pravilno obnašal.

16. **Ugibanje napak** Zanaša se na izkušnje testnih inženirjev in predhodne probleme organizacije. Preizkusi transakcije, ki so večkrat povzročile velike napake.

17. **Izčrpavanje vseh možnosti**. Za vsak pogoj in vsako pot v programu poskušamo narediti testno transakcijo.

18. **Modeliranje okolja** simulira različna okolja in ugotovi, kako se v njih obnaša sistem.

19. **Združljivost operacij** preveri, če stare in nove verzije programa dajejo enake oziroma združljive rezultate.

To je le majhen delček vseh orodij. Večina naštetih orodij se ne uporablja na široko. Razlog je v njihovi specifičnosti in velikih stroških uporabe. Pomembno je, da se testiranje izvede skozi življenjski cikel razvoja projekta. Eden izmed razlogov za uspeh je disciplinirano ročno testiranje zahtev dizajna in kode (Nguyen, 2001). Ta orodja lahko uporabimo brez velikih finančnih sredstev. Previdno planiranje, dobra definicija začetka testa in organizirano shranjevanje zapisov je kritično za uspešno testiranje.

Pomembna je zbira orodja, boljša ko je izbira bolj učinkovit je testni proces. Naj omenim, da za mrežne aplikacije še ni bilo razvitega dobrega avtomatskega orodja. Še vedno je ročno klikanje po straneh najbolj zanesljivo in hitro. Narejeno je bilo avtomatsko orodje za odkrivanje napak na HTML (ang. Hyper Text Markup Language) straneh, vendar, kot se je izkazalo, nima prave vrednosti.

Tip testiranja se razlikuje glede na fazo v življenjskem ciklu razvoja projekta, zato je nujno izbrati orodje, primerno fazi življenjskega cikla v katerem se bo test izvajal. Bolj, ko gre življenjski cikel proti koncu bolj prehajajo testna orodja od ročnih k avtomatskim. Najbolj produktivno je ravno testiranje v začetnih fazah.

Vsak posameznik mora izbrati orodje primerno njegovim ali njenim sposobnostim. Neprimerna bi bila izbira orodja, ki zahteva programiranje od testnega inženirja, ki tega ne obvlada. Seveda pa lahko za uporabo orodij naredimo trening.

Uporabniške sposobnosti zahtevajo od posameznika poznavanje aplikacije, poznavanje okolja v katerem bo aplikacija uporabljena, sposobnost odkrivanja in ukvarjanja z uporabniškimi problemi.

Programerske sposobnosti zahtevajo razumevanje računalniških konceptov, programiranje v jezikih, ki jih organizacija uporablja, odkrivanje napak ( ang. debuging) in dokumentiranje programov.

Sistemske sposobnosti vsebujejo zmožnosti prevesti uporabniške zahteve v specifikacijo dizajna. Pod te sposobnosti štejemo analizo

problemov, dizajniranje metodologije, računalniške operacije in nekaj splošnih poslovnih sposobnosti.

# 5. Vrednotenje rezultatov in poročila narejenih testov

V tem poglavju opozarjamo na pomen dokumentacije v testnem procesu. Testni proces je prevelik in preveč kompleksen, da bi se lahko izognili njegovemu dokumentiranju. Dokumentirati je potrebno vse teste, ki jih izvedemo in njihove rezultate, ki jih ovrednotimo. Narejen mora biti testni plan, vsa testna dokumentacija pa se mora neprestano obnavljati. Tudi dokumentacija mora ustrezati dokumentacijskim standardom, kot tudi vsa projektna dokumentacija. Bolj kot je ta strukturirana lažje jo je popravljati.

Testna dokumentacija, bi morala biti del aplikacijske dokumentacije. Predpisana bi morala biti njena dolžina glede na potrebe in uporabnost. Dokumentiran bi moral biti celoten testni proces.

Dokumentacijo lahko razdelimo v dve kategoriji. Te sta testni plan in poročila testiranja.

## *5.1 Testni plan*

Testni plan bi moral biti plan za testiranje aplikacije vključno z vsemi opisi testih primerov. Testni primer vsebuje opis testa, okolje v katerem bo teklo, orodje, ki ga bomo uporabili in pričakovani rezultat. Oblika testnega plana naj bi izgledala tako, kot v spodnji tabeli.

Tabela 5.1: Vsebina testnega plana

**TESTNI PLAN**

1. SPLOŠNE INFORMACIJE

1.1 Povzetek funkcij programske opreme, ki bodo testirane.

1.2 Okolje in zgodovina projekta.

1.3 Reference (opis vseh testnih politik, standardov, ostalih dokumentov, vezanih na ta projekt).

2.PLAN

2.1 Opis programske opreme (na kratko se opišemo vse funkcije in programsko opremo, ki bo testirana).

2.2 Časovni plan.

2.3 Zahteve.

2.4 Testni material (dokumentacija, primeri vhodnih in izhodnih podatkov).

2.5 Treningi potrebni za testiranje.

3. SPECIFIKACIJA

3.1 Zahteve ( seznam osnovne funkcionalnosti za prvo testiranje).

3.2 Funkcije programske opreme in njene relacije.

3.3 Metodologija.

3.4 Pogoji.

4. OPIS TESTOV

4.1 Metode za kontrolo testnega procesa. Ročne ali avtomatske metode za vpis podatkov, operacij in zapis rezultatov.

4.2 Opis vhodnih podatkov in komand.

4.3 Opis rezultatov in možnih vmesnih sporočil.

4.4 Podroben opis po korakih vseh testnih procedur.

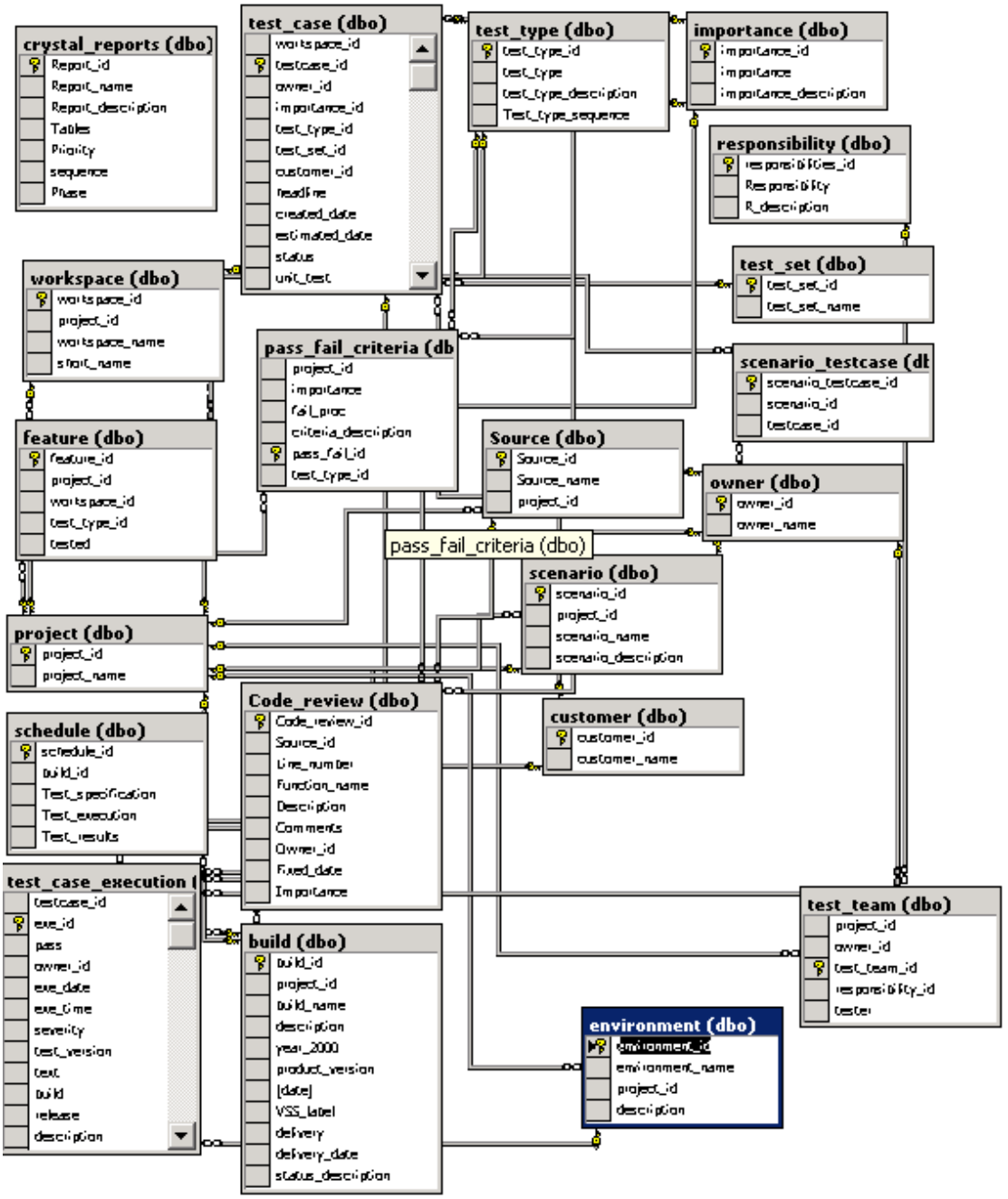
Vir: Perry, 2000

Za podroben opis testov in vodenje korakov, opis potrebnih orodij, definicijo testnega okolja ter verzij aplikacije je smiselno zgraditi bazo podatkov, v katero vpisujemo vse našteto. Testni plan lahko namesto opisov testnih primerov vsebuje povezave z dokumenti, kjer se le ti nahajajo. Hkrati pa lahko v to bazo vpisujemo tudi rezultate izvajanja testov. To orodje je zelo splošno in ustreza mnogim tehnikam. Zelo pospeši celotno izvajanje testiranja, ga sistematizira. Nenazadnje pa

lahko iz baze s pomočjo programa Seagate Crystal Reports iz baze vedno naredimo poročila s poljubno vsebino in obliko.

Izgled tabel v bazi in njihove povezave si poglejmo na sliki5.2.

Slika 5.2 Diagram tabel in povezav v bazi Test Assistent.

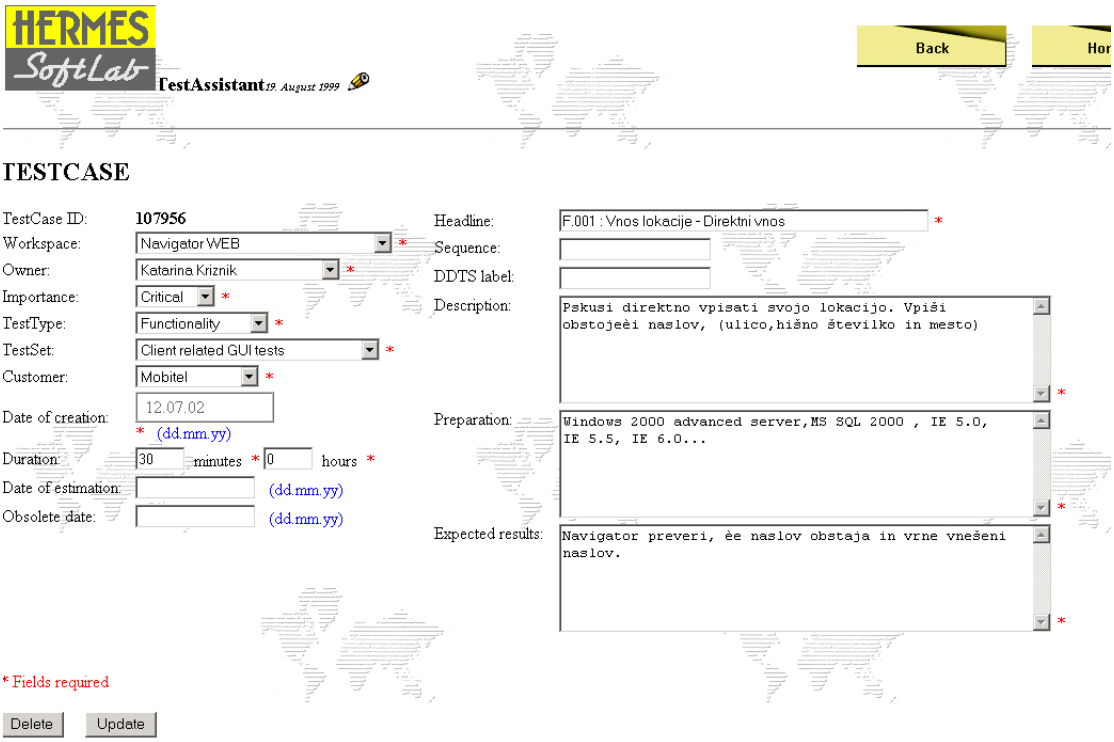



Vir: URL: http://jadran.hermes.si/testassistant/

Če pogledamo zgornji diagram, vidimo kompleksnost celotnega testiranja. Praktično nemogoče bi bilo vse te podatke voditi v množici excelovih tabel in hkrati skrbeti za njihove rezervne kopije. Na osnovi

SQL-stavkov lahko vedno dobimo želene podatke. Lahko pa si za lažje vpisovanje in izpisovanje naredimo ASP strani, ki dostopajo do te baze. Na sliki 5.3 poglejmo kako izgleda ASP stran za vpisovanje posameznega testnega primera v bazo.

Slika 5.3: Vnos podatkov v bazo Test Assistant.

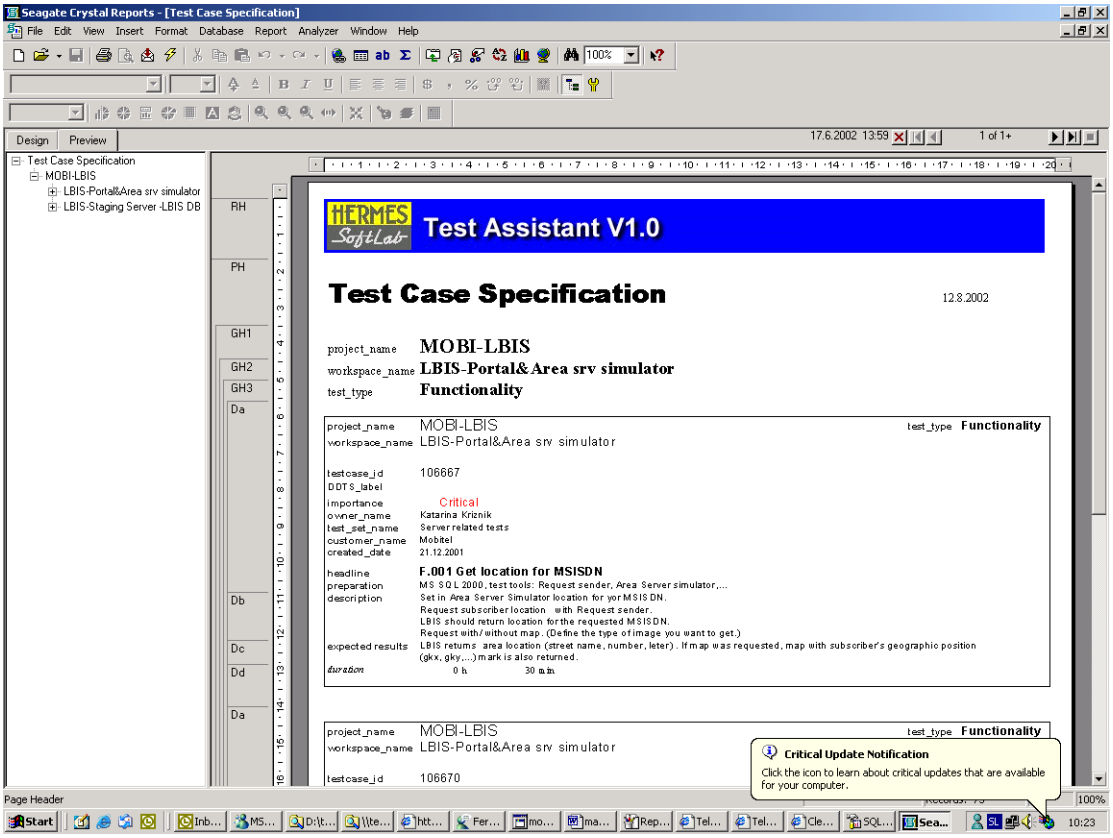



Vir: URL:http://jadran.hermes.si/testassistant/TestCaseMaintain.asp

Na sliki 5.2 smo videli, kaj vse je potrebno vpisati v bazo, da definiramo posamezen testni primer. Za dobro testiranje je pomemben govoreč naslov testnega primera (ang. headline) in številka zahteve, ki je vezana na specifikacijo stranke. Podoben izgled ima stran za vpis izvedenih testov. Ti se vpisujejo v tabelo »Test_case_execusion«.

Za izpis vseh testnih primerov za posamezni projekt, podprojekt, verzijo ter scenarij lahko uporabimo že omenjeni Seagate Crystal Reports. Dizajn izgleda poročil naredimo sami. Glede na to, da je testna dokumentacija del dokumentacije, ki se preda stranki, je smiselno dizajn skozi projekte standardizirati.

Slika 5.4 Specifikacija testnih primerov, narejena s pomočjo Seagate Crystal Reports




Vir: URL: http://dirlija.hermes.si:8005/ddts/ddts_main

## *5.2 Poročila testiranja in analiza rezultatov*

Poročila narejenih testov dokumentirajo rezultate testov. Zanimivi so za stranko programerje aplikacije in tudi za menedžerje. Analiza rezultatov testiranja ovrednoti tudi kvaliteto testiranja, pokaže, kateri procesi so efektivni in kateri ne. S pomočjo teh rezultatov lahko ocenimo in spremenimo metodologijo.

Vsebina poročila testiranja naj bi vsebovala štiri dele, kjer se nahajajo splošne informacije o testu, dokumentacija rezultatov in njihova odkritja, dokumentacija aplikacije ter povzetek analize in mnenja testnih inženirjev.

V tabeli 5.5 predstavljamo priporočljivo vsebino testnega poročila.

Tabela 5.5: Vsebina poročila testiranja.

POROČILO TESTIRANJA

1. SPLOŠNE INFORMACIJE

1.1 Povzetek glavnih funkcij programske opreme in testov, ki so bili narejeni

1.2 Zapiši okolje, v katerem bo aplikacija inštalirana, in kako bi se odražala na aplikacijo in njeno testiranje razlika v okolju.

1.3 Seznam vseh referenc, to je ostale dokumentacije vezane na projekt in testni plan.

2.REZULTATI TESTIRANJA

2.1 Zapiši rezultate dinamičnega testiranja in jih primerjaj s pričakovanimi rezultati.

2.2 Zapiši rezultate statičnega testiranja in jih primerjaj s pričakovanimi rezultati.

3. REZULTATI IN ODKRITJA ZA POSAMEZNO FUNKCIJO

3.1 Ime funkcije

3.1.1 Kratek opis funkcije. Kratek opis programske opreme potrebne za izvajanje funkcije. Opis rezultatov, ki jih je pokazal eden ali več testov.

3.1.2 Opis robnih pogojev in mejnih vrednosti. Opis odstopanj od predvidenega, ugotovljenega med testiranjem.

4. POVZETEK ANALIZE

4.1 Opis zmožnosti aplikacije, kot jih je pokazal test. Pripravi ugotovitve, zapiši primerjave med zmožnostmi in zahtevami. Opiši kako se odražajo razlike v testnem okolju od produkcijskega okolja. Rezultati naj bodo osredotočeni na performanco.

4.2 Priporočila in ocene. Za vsako razliko od zahtevanega navedi oceno časa in napora potrebnega za popravilo, in predloge kot so :

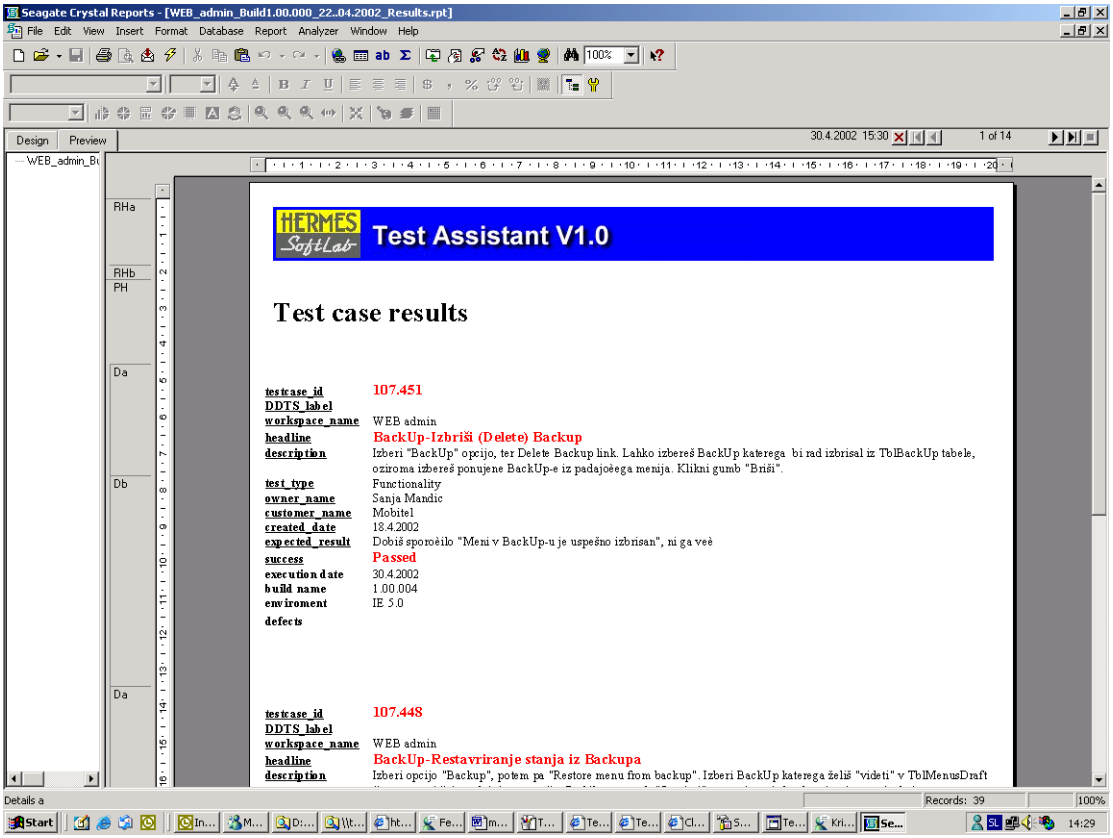
a) Nujnost vsake spremembe
b) Kdo je odgovoren za spremembe
c) Kako naj bo sprememba narejena

4.3 Podaj mnenje o pripravljenosti aplikacije za instalacijo v produkcijo ali pa zahtevaj popravke in novo verzijo aplikacije.

Vir: Perry, 2000

Za opis vseh opravljenih testov lahko spet uporabimo Seagate Crystal Reports ( glej sliko 5.6), ki nam bo iz baze v želeni obliki izpisal vse rezultate izvedenih testov.

Slika 5.6: Prikaz rezultatov izvedenih testov s pomočjo Seagate Crystal Reports.




V končno poročilo je zaželeno vključiti tudi poročilo o številu vseh napak skozi projekt. Zanimiv je podatek o številu odpravljenih in neodpravljenih napak. Če smo za vpis napak uporabljali orodje DDTS ali kaj podobnega nam ni potrebno preštevati napak, ker nam takšne podatke omogoči orodje samo. Glej sliko 5.7

Slika 5.7: Poročilo o številu napak po stopnji kritičnosti generiran z orodjem DDTS.

ClearDDTS MANAGEMENT SUMMARY
of
PROBLEMS by PROJECT by SEVERITY
RESOLVED & UNRESOLVED PROBLEMS
(Mon Aug 12 09:45:07 MEST 2002)

| Project | Sev1 | Sev2 | Sev3 | Sev4 | Sev5 | Total |
|---|---|---|---|---|---|---|
| Mobi-LBIS | 11 | 12 | 55 | 10 | 5 | 93 |
| TOTAL | 11 | 12 | 55 | 10 | 5 | 93 |

Youngest Problem Date => 20020729

Oldest Problem Date => 20020124

Software Versions => 2.00.003 2.00.007 2.00.005
2.00.004 2.00.002 2.00.006
2.0.002 2.00.001 0.00.001
0.00.005 0.00.009 0.00.008
0.00.007 0.00.004 2.00.007+

O.S. Versions => win 2000 Pro RequestSender RequestResponse
windows 2000 RewuestSender Requestsender
TestTool SB Sleeping Beauty TestTool
N/A REquestSender windows XP
Other Test tool SB Test Tool
SB Test Tool RequestSender

ClearDDTS MANAGEMENT SUMMARY
of
PROBLEMS by PROJECT by SEVERITY
UNRESOLVED PROBLEMS ONLY
(Mon Aug 12 09:45:07 MEST 2002)

| Project | Sev1 | Sev2 | Sev3 | Sev4 | Sev5 | Total |
|---|---|---|---|---|---|---|
| Mobi-LBIS | 0 | 0 | 2 | 0 | 0 | 2 |
| TOTAL | 0 | 0 | 2 | 0 | 0 | 2 |

Youngest Problem Date => 20020729

Oldest Problem Date => 20020124

Software Versions => 2.00.003 2.00.007 2.00.005

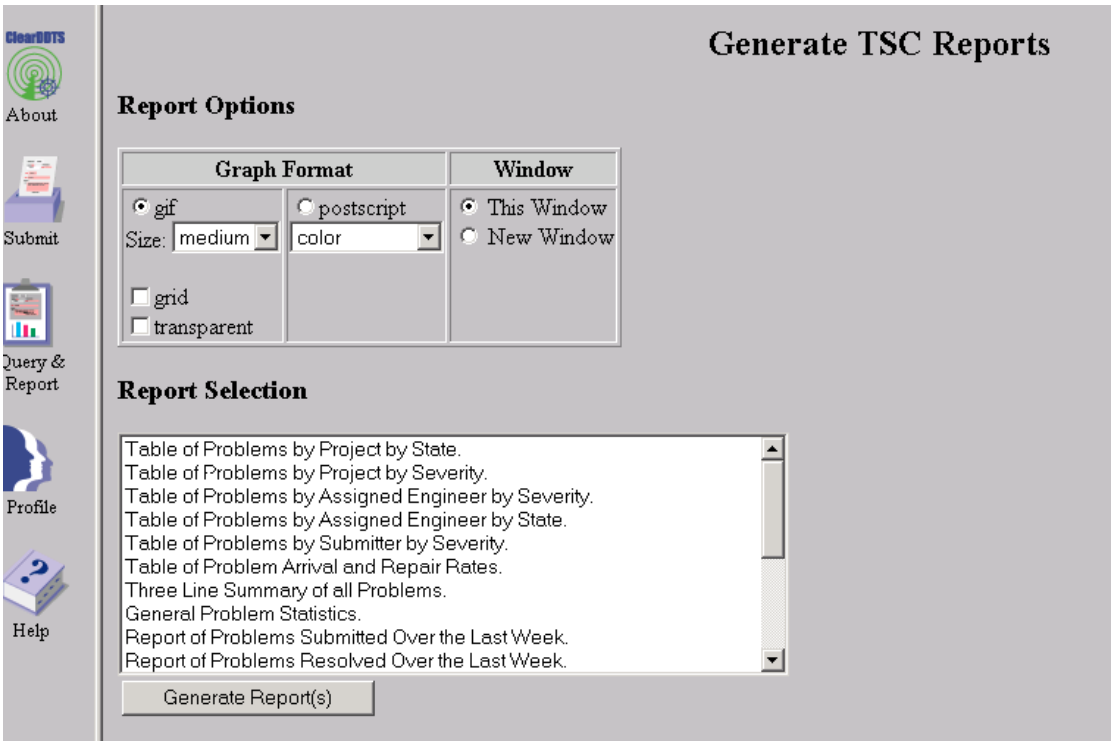
2.00.004 2.00.002 2.00.006
2.0.002 2.00.001 0.00.001
0.00.005 0.00.009 0.00.008
0.00.007 0.00.004 2.00.007+

O.S. Versions => win 2000 Pro RequestSender RequestResponse
windows 2000 RewuestSender Requestsender
TestTool SB Sleeping Beauty TestTool
N/A REquestSender windows XP
Other Test tool SB Test Tool
SB Test Tool RequestSender

Vir: http: //dirlija.hermes.si:8005/ddts/ddts_main

Glede na želje stranke lahko s tem orodjem generiramo še poljubne tabele in grafe, ki nam dajejo pregled in različne poglede na rezultate. Katere možnosti nam da orodje DDTS, lahko vidimo na sliki 5.8

Slika 5.8: Izbira vseh možnih poročil generiranih z DDTS.




Vir: URL: http://dirlija.hermes.si:8005/ddts/ddts_main

Zanimivo je tudi pogledati grafični prikaz števila in stopnje napak glede na verzijo produkta. Število napak vseh stopenj naj bi se iz verzije v verzijo zmanjševalo in na koncu v zadnji verziji bilo blizu nič. Vsaj kritičnih in večjih napak ne sme biti več. Kakšen je dovoljen

odstotek manjših napak, mora biti definirano v zahtevah stranke, zato je v končno poročilo pomembno dati tudi odstotek nerešenih napak v zadnji verziji.

# 6. Testiranje mrežnih aplikacij, aplikacij za elektronsko poslovanje ter mobilnih aplikacij

Zjutraj, ko grem od doma sem vedno pozna. Pošljem hitro e-pošto s svojim telefonom, da naj še zame skuhajo kavo. Ko se peljem z avtom opazim, da mi zmanjkuje bencina. S telefonom poiščem najbližjo odprto bencinsko črpalko.

Ko končno prispem v službo, mi ob kavi kolega pove, da imajo pri bližnjemu prodajalcu avtomobilov razprodajo. Že dolgo kupujem nov avto. Spet mi tehnologija omogoča hiter pregled ponudbe.

Ob kosilu se s kolegom odločiva, da bova zvečer obiskala kino. Hitro dobim informacije o kinu s svojim telefonom in rezerviram karto za večerno predstavo. Popoldne, ko se vozim v avtu mi zapiska moj žepni PC. Mama ima rojstni dan. Izberem darilo, ki ga ponujajo na daljavo aktivni servisi (ang. online servisi) in plačam z elektronskim denarjem.

Vse to mi omogočajo mobilne aplikacije in omrežne aplikacije. Omrežne aplikacije so osnovane na internetu, intranetu in ekstranetu. Prav tako je tudi elektronsko poslovanje osnovano na internetu. V tem poglavju se osredotočimo na specifičnost testiranja takšnih aplikacij. Pri testiranju teh lahko uporabimo enak življenjski cikel, kot je opisano v poglavju 3, s poudarkom na testiranju uporabnosti in varnosti.

Testni plan mora vsebovati tveganja vezana na internet. Izbrati je potrebno primerna orodja za izvajanje testov, povezanih z mrežnimi aplikacijami.

## *6.1 Testiranje mobilnih aplikacij je drugačno od običajnega*

Število brezžičnih aplikacij različnih podjetij narašča dnevno. Za vsako takšno aplikacijo je veliko dejavnikov, ki jih je potrebno premisliti in testirati, da zadovoljimo funkcionalnosti. Vsak dober testni inženir ve, da mora posvetiti pozornost osnovam testiranja. Te so (Courtney, 1997):

- preverjanje osnovne funkcionalnosti in značilnosti aplikacije;
- preverjanje dizajna in rešitev že v ciklu razvoja projekta;
- Testi kompatibilnosti. Test aplikacije na vseh možnih variacijah programske in strojne opreme, na katerih bo aplikacija tekla;
- izpostava celega sistema oz. aplikacije nepredvidljivim dogodkom, nepredvidljivemu obnašanju uporabnika;
- izpostava programske opreme velikemu številu robnih podatkov, strestnih primerov, vse z namenom ugotoviti resnični učinek oz. meje performance;
- uporabnost in tudi zahteve, ki niso posebej določene s strani stranke.

Dober plan testiranja bo vključeval zgoraj naštete zahteve. Ko pa gre za mobilne aplikacije (ang. wireless enabled applications), mora biti testna strategija spremenjena, z namenom zagotovitve, da strojna in programska oprema testirane aplikacija ustreza vsem ciljnim okoljem in uporabnikom (Greenberg,1998).

Strateški načrt kako se lotiti testiranja mobilne aplikacije vsebuje številne karakteristike, ki so specifične za posamezen mobilni aparat. Več pozornosti velja posvetiti :

- kompleksnosti testov na številnih novih na dan prihajajočih žepnih računalnikih in telefonih,
- varnosti aplikacij in
- obremenitvenim testom.

### *6.1.1 Povečanje kompleksnosti testov na področju žepnih računalnikov*

Najbrž ni nobenega drugega področja v mobilnem računalništvu, ki se bi razvijalo tako hitro kot prav prezentacijske naprave (žepni računalniki). Prvi žepni računalnik je bil Apple Newton narejen leta 1993. Od takrat je nastalo na tisoče vrst različne strojne opreme, programske opreme in brezžičnih omrežij. Dejansko je osnova žepnih računalnikov veliko bolj kompleksna kot je bila včasih vsa arhitektura mrežne aplikacije. Brezžična telefonija je pri žepnih računalnikih izzvala ogromno novih specifičnih karakteristik (Cabiory, 1997).

Današnji žepni računalniki imajo že vsak svoj operacijski sistem, brskalnik, aplikacije v realnem času (ang. runtime application). Testiranje kompatibilnosti med njimi in sam brezžični mrežni sistem je eden izmed pomembnejših pri razvoju mobilnih servisov.

Podpora različnim vmesnikom in prikazovalnim standardom, aplikacije razvite v HDML (ang. handheld device markup language), WML (ang. Wireless markup language), WAP 1.2 (ang. Wireless Application Protocol), WAP 2.0, cHTML in x HTML (ang. Hyper Text Markup Language), bodo še naprej potrebovale testiranje na različnih napravah.

Pod kompatibilnost štejemo tudi podporo različnim vmesnikom. Predstavljajmo si aplikacije, ki so bile kompatibilne z operacijskimi sistemi, kot so Windows 3.1 Win95, win98, win2K, Windows ME, Windows XP ter Macintosh in so izziv tako za razvijalce, kot testne inženirje.

Nov primer je aplikacija na platformi uporabnika. Včasih je bil operacijski sistem relativno preprost, vse se je procesiralo na WAP strežniku in potem se je informacija v obliki teksta počasi prenašala do uporabnika. S prihodom novih žepnih računalnikov pa ima uporabnik svojo lastno bazo in bogato uporabniško okolje, kar spet povzroči večjo kompleksnost testiranja (Gerrard, 2000a), (Gerrard, 2000b).

Z možnostjo naložitve in izvajanja kode preko omrežja, nastopi skrb za varnost, tveganje povezano z avtentikacijo uporabnika in varnostjo

transakcij, virusov ter druge zahrbtne kode. Vse to poveča potrebo po testiranju in verifikaciji varnostnih mehanizmov.

V naslednjih petih letih bo v vsaki večji regiji na svetu mrežna evolucija, ki bo nadgradila drugo generacijo mobilne tehnologije v tretjo. Omrežje sedaj ponuja prenos podatkov z hitrostjo 9,6 KBps, v novi generaciji z UMTS (ang. Universal Mobile Telecommunications System) pa bo omrežje omogočalo prenos informacij z hitrostjo 2 Mbps. Vse te spremembe bodo zahtevale prilagoditev in globalni premik. Verifikacija in testiranje servisov bosta neobhodna tudi za tiste, ki so bili doslej najbolj brezbrižni.

### *6.1.2 Testiranje mobilnih aparatov*

Dvom o tem, da razvitje servisov naslednje generacije mobilnih telefonov zahteva razumen test in inženirje, ki zagotavljajo kvaliteto, naj razjasnim z nekaj dokazi.

- Leta 2001 so uporabniki Nokia, Samsung, Sony in Matsushita telefonov naleteli na napake v programski opremi, ki so povzročile 95 miljonov dolarjev škode.
- Na demonstraciji GPRS (ang. General Packet Radio Service) opreme je morala Nokia uporabiti Motoroline mobilne aparate, ker so v zadnjem trenutku odkrili napako v njihovi programski opremi.
- Tretja generacija mobilnih telefonov je na Japonskem že v uporabi. Hkrati pa izvajajo teste, ki so pokazali, da čeprav signali dosežejo telefone, kompleksne I-mode strani in aplikacije niso preveč zanesljive.

Seveda se pričakuje, da bo med testiranjem programske opreme prišlo do odkritja napak, toda proizvajalec jih mora najti in popraviti pravočasno. V odkrivanje napak mora vložiti trud, da ostane lojalen strankam in opravi s konkurenco.

Testiranje mobilnih aplikacij mora vsebovati naslednje ugotovitve ustrezne različnim telefonskim aparatom (Hayes, 1996):

- Testni inženirji morajo posvetiti več pozornosti uporabnosti in obliki kot običajnim aplikacijam. Manjši zasloni, počasnejši

procesorji, manj prostora za podatke in počasnejše povezave za prenos podatkov zahtevajo testiranje z najrazličnejšimi metodami. Testni inženir mora poznati specifične metode, ki jih razvijalec uporablja pri razvoju ter najti strategijo in specifične metode, kako testirati te faktorje.

- Testni inženirji morajo pridobiti strokovno znanje o različnih modelih telefonskih aparatov in njihovi programski opremi. Raznolikost strojne opreme, operacijskih sistemov, mikro-brskalnikov zahteva od strokovnjakov, da preizkusijo kako kompatibilnost vpliva na funkcionalnost in performanco mobilnih aplikacij.
- Testni inženirji morajo biti pripravljeni prilagoditi teste za obravnavo telefonskega aparata kot računalniškega sistema in ne le kot preprost odjemalec. Poudarek je potrebno dati na interakcijo med mobilno aplikacijo in kompleksnostjo procesiranja podatkov.

### *6.1.3 Testiranje brezžične infrastrukture*

Testi kažejo, da se precejšen del ampak na mobilnih aplikacijah pojavi zaradi specifičnosti strojne in programske opreme ter vmesnikov, manj pa na samem brezžičnem omrežju. Spremenilo pa se bo s prihajajočo tretjo generacijo mobilnega omrežja in z njo prihajajočimi servisi, protokoli in brezžičnimi aplikacijami. Kvaliteta storitev bo eden izmed najpomembnejših faktorjev za zadovoljstvo stranke.
Navedimo nekaj nasvetov kako izboljšati testiranje brezžične infrastrukture (Sabourin, 2001), (Weyukler, 2000), (Wilson, 2001):

- Testni inženirji morajo razviti strokovno znanje za vso mrežno konfiguracijo, ki se uporablja. Poznati morajo vse omrežne vmesnike, protokole, požarne zidove in tehnične rešitve.
- Koristno vpeljati teste v ciljno aplikacijo. Za kompleksne aplikacije je to še posebej pomembno. Testirati je bolje v pravem omrežju kot na prototipu, saj s tem zagotovimo pravilno testiranje varnosti (avrentikacija, gostovanje, WTLS/SSL interpretacija), performance, mrežnih obremenitev aplikacij vrste odjemalec/strežnik .
- Previdno je potrebno izbirati orodja za brezžične aplikacije. Čeprav obstaja kar nekaj dobrih testnih orodij, lahko njihova

uporabnost zelo niha glede na komplekstnost aplikacije in standarde kodiranja ( WML, HDML, xHTML).

- Aplikacije, ki vsebujejo komercialne transakcije in avtomatsko zaračunavanje, zahtevajo poudarek na testiranju v realnem času. Lokacijsko osnovane aplikacije pa zahtevajo spet svoje teste.

### *6.1.4 Testiranje mobilnih servisov in aplikacij*

Številni tipi brezžičnih aplikacij, ki so opisani na začetku tega poglavja napovedujejo pestrost uporabe mobilnega omrežja v prihodnosti. Pri verifikaciji funkcionalnosti morajo testni inženirji upoštevati (Courtney, 1999):

- Verifikacija servisov bo zahtevala tudi testiranje v realnem okolju. Testiranje z emulatorji nam v testnem okolju ne zagotovi enako kvaliteto, kot bi jo testiranje v realnem okolju. Emulatorji in simulatorji so uporabni za verifikacijo funkcionalnosti in kompatibilnosti. Te lahko v testnem okolju kontroliramo. Torej naj bo testno okolje za testiranje po metodi bele škatle. Ostalo pa je potrebno preveriti v pravem okolju. Na sliki 6.1 si poglejmo izgled SMS emulatorja.
- Poseben poudarek pri testiranju bodo zahtevale brezžične WAN/PAN (ang. wireless networks, personal area networks) aplikacije. Zagotoviti je potrebno neokrnjenost servisa skozi omrežje.
- Mobilna aplikacija lahko teče na različnih napravah (telefonu, dlančniku, računalniku). Iste informacije lahko dobimo preko različnih uporabniških vmesnikov (ang. Interface) WAP-a WEB-a ali SMS-a. Temu pa morajo biti prilagojeni tudi testi.

Slika 6.1: SMS emulator


vir: URL: http://grunf.hermes.si/QualityManual

## *6.2 Testiranje mrežnih aplikacij*

Dobra internetna stran je nujno potrebna za uspeh podjetja, posebno s pojavom e-poslovanja in trendom aplikacij na omrežju. Če si vpleten v katerikoli korak razvoja omrežnih strani, si odgovoren tudi za prispevek h kvaliteti strani.

Da bi obiskovalcu zagotovili visoko kvaliteto strani in ponovni obisk strani, potrebujemo dobro planirano in premišljeno testiranje, ki vključuje različne brskalnike, različne tehnologije in variante interneta.

Uporabniki zahtevajo večjo kvaliteto kot kdaj koli in samo en klik jih lahko prepriča, da ne uporabljajo več tvoje strani in gredo h konkurentu. Kaj storiti, da se to ne bo zgodilo?

### *Kompatibilnost brskalnikov*

Kompatibilnost brskalnikov je bistvenega pomena. Veliko je različnih brskalnikov, še več pa verzij vsakega brskalnika. Trg je najbolj preplavljen z tremi: Netscape Navigator, Microsoft Internet Explorer (IE) in America OnLine (AOL).

Brskalniki ne sledijo vedno istim standardom, zato tehnologije in funkcije, ki delajo v enem brskalniku, ne delujejo nujno tudi v drugem. To pomeni, da je potrebno aplikacijo zelo previdno stestirati z uporabo najrazličnejših brskalnikov.

Kako se brskalniki razlikujejo med sabo? Nove verzije brskalnikov podpirajo nove tehnologije. Netscape in IE ne implementirata iste tehnologije na enak način, zato zaradi teh razlik WEB stran lahko izgleda različno in se obnaša različno če jo gledamo z različnimi brskalniki. Večkrat je WEB stran razvita tako, da je razvijalec upošteval samo en, brskalnik ostale pa zanemaril. Testiranje skozi različne brskalnike bo razkrilo napake, ki se jih razvijalec ob pisanju kode ne zaveda. Kot primer naj navedem WEB stran, ki vsebuje gumb »Pošlji naročeno!«. Tak gumb v IE izgleda čisto normalno, v Netscapu pa je lahko ta gumb premajhen. Posledica je, da se ne vidi celoten tekst. IE podpira VB Script in Java Script, medtem ko Netscape podpira samo Java Script. Pa še veliko drugih razlik je med njima, kot na primer način kako podpirata DHTML in ozadja (ang. Cascading Style Sheets).

Naslednja razlika je tehnologija ActiveX. Netscape za njihovo predstavitev potrebuje instalacijo dodatnega programa (plug-in za ActiveX), zato je pri testiranju potrebno uporabi računalnik, ki še nima instaliranih nobenih dodatnih programov. Zato nasvet za testiranje- vedno imejte še kakšen testni primer, ki vključuje na novo inštaliran brskalnik, brez dodatnih komponent (Samaroo, 1999).

S katero verzijo brskalnika naj bi pravzaprav testirali? Veliko je uporabnikov, ki uporabljajo najrazličnejše verzije brskalnikov. Niso vsi pripravljeni instalirati nove verzije programa. Spet drugi pa instalirajo novo verzijo takoj, ko je to le mogoče. Pri testiranju moramo zaradi teh razlogov uporabiti najrazličnejše verzije brskalnikov 3.x, 4.x, 5.x in 6.x Netscape, IE in AOL. Lahko pa se odločimo za drugo strategijo, da testiramo le z najnižjo in najvišjo verzijo.

### *Kompatibilnost Operacijskih sistemov*

Testiranje mora bit opravljeno na različnih operacijskih sistemih, saj tudi operacijski sistem vpliva na to kako se omrežna stran prikaže. Omrežna stran bi morala biti stestirana na naslednjih platformah.

- Windows 95 Gold,
- Windows 95 OSR 2.0 s,
- Windows 95 OSR 2.1,
- Windows 95 OSR 2.5,
- Windows 98,
- Windows 98 Second Edition,
- NT 4.0,
- NT 4.0 w/ SP1,
- NT 4.0 w/ SP2,
- NT 4.0 w/ SP3,
- NT 4.0 w/ SP4,
- NT 4.0 w/ SP5,
- Windows 2000,
- Windows XP,
- Mac 7.5.5 ,
- Mac 8.6

Seveda pa moramo poznati ciljno publiko. Kakšnega iz zgornjega seznama lahko odvzamemo ali pa dodamo. Če bodo znanstveniki gledali te strani moramo na zgornji seznam dodati še Unix in Linux.
Pri testiranju na različnih operacijskih sistemih moramo pa vedno dodat še različne brskalnike. Najbolje je, da si naredimo matriko, ki kaže, katere kombinacije smo pokrili.

### *Funkcionalnost*

Dobra WEB stran je več kot samo nekaj za pogledat. Je funkcionalna in interaktivna. Funkcionalnost omrežnih aplikacij mora biti stestirana na enak način kot funkcionalnost običajnih aplikacij, saj trend prodaje mrežnih aplikacij malce izpodriva običajne aplikacije.

### *Šablone*

Verjetno bo omrežna stran vsebovala tekstna polja za vnos podatkov, gumbe za potrditev, polja in liste za izbiro podatkov. Vse to mora bit preverjeno kot v običajnih aplikacijah. Verjetno bomo naleteli na veliko več napak, saj skriptni jezik ni tako zanesljiv kot C++. Posebno pozornost velja posvetiti preverjanju tipov podatkov in lovljenju napak. Na primer:

- Lahko tekst vstavimo v polje za številke?
- So podpičja, narekovaji, šumniki dovoljeni?
- Je prazen znak dovoljen?
- Kako je z dolgimi nizi?
- So začetne vrednosti pravilne?

Poznavanje tehnologij ActiveX, Jave, ASP in drugih lahko zelo pomaga pri testiranju, saj vsaka vsebuje svoje prednosti in pomanjkljivosti. Zavedati se moramo različnih verzij JVM (Java Virtuale Mashine), ki lahko povzročijo probleme in nenazadnje sta Java in JavaScript popolnoma različna jezika.

### *Baze*

Testiranje baze mora bit narejeno skozi brskalnik. Preverjeno mora biti iskanje po bazi, dodajanje dupliciranih informacij, dodajanje, popravljanje, brisanje. Ne smemo pozabiti na dolge nize in šumnike.

### *Varnost*

Tudi varnost mora biti predmet preverjanja. Če aplikacija zahteva prijavo z geslom, je potreben podroben test te avtentikacije. Preveriti je potrebno, če uporabnik ne more priti direktno na URL, ki sledi po avtentikaciji. Potrebno je dobro preveriti spremembo gesla Za dodatno varnost in enkripcijo pa preverimo, če aplikacija uporablja Secure Sockets Layer (SSL). URL take aplikacije se mora začet s https.

### *Internet stran in okna*

Okna na strani so lahko dobra ali slaba, na eni strani lahko pripomorejo k dobri organizaciji strani, na drugi k zbeganosti. Popolnoma lahko zmedejo navigacijo strani. Preveriti moramo:

- Se pojavijo gumbi za premik strani (ang. scroll bari), ko seznam postane dolg?
- Lahko okno povečamo oz. zmanjšamo brez večjih težav?
- Če brskalniku spremenimo velikost, se mora prilagoditi tudi vsebina.
- Se stran pravilno prikaže, če uporabimo gumb refresh?
- Ali okna avtomatsko in pravilno spreminjajo velikost? Lahko uporabnik sam prilagaja velikost oken?
- Ali povezave, ki so vključene v stran, resnično obstajajo?

### *Animacije ter Avdio in Video komponente*

Animacije in avdio ter video komponente so popularne in lahko v stran vlijejo malo življenja. Na kaj moramo biti pozorni:

- Ali animacija gladko teče?
- So kakšni bliski med animacijo?
- Koliko prostora zasede animacija?
- Je kvaliteta slike in zvoka zadovoljiva?
- Se avdio in video gumb ustavi na pravem mestu?

### *Tiskanje*

Pomembno bi bilo preveriti tiskanje strani na različnih tiskalnikih. Lahko se zgodi kaj nepredvidljivega, posebno z okni. Ne smemo pozabiti na vsebino stiskane strani in čas tiskanja.

### *Navigacija*

Navigacija WEB strani bi morala biti enostavna, logična in intuitivna. Uporabnik mora krmariti po straneh v vseh smereh in se ne sme izgubiti. Vedno mora obstajati možnost vrniti se na prejšnjo stran in

začetek. Vse povezave morajo ustrezno delovati in privesti uporabnika na želeno stran.

Povezave so lahko tekstne ali grafične in nekateri zemljevidi lahko imajo celo več povezav. Preveri, da so vse povezave pravilne, spremenijo barvo, ko jih izberemo in odprejo želeno stran. Povezave je potrebno preveriti večkrat, ker lahko postanejo zastarele.

Poleg povezav imajo brskalniki gumbe za nazaj in naprej, listo zgodovine. Vse te metode navigacije moramo preveriti, da zadostimo funkcionalnosti in uporabnosti.

### *Izgled strani*

Izgled strani je odvisen od izkušenj uporabnika. Poleg dobrega izgleda prve strani je pomembno, da je izgled enak skozi vse strani. Še enkrat pa naj omenim, da je potrebno testirati z več operacijskimi sistemi in kombinacijami, ker lahko pride do velikih razlik.

Naslednji problem je resolucija monitorja in intenzivnost barv . Pri resoluciji 600 x 800 je lahko stran popolna, medtem ko ji pri resoluciji 640 x 480 nekaj manjka.
Pozorni moramo biti na naslednje:

- barve linkov,
- prelomljene slike,
- premajhen barvni kontrast,
- presledke v tabelah,
- prekrivanje tekstov,
- robove,
- poravnavo in velikost teksta in
- poravnavo kontrol.

### *Uporabnost*

Če spletne strani niso uporabniku prijazne oz. intuitivne, si lahko ta kaj hitro premisli in odide do spletne strani konkurenčnega podjetja. V izogib temu, so potrebni temeljiti testi uporabnosti. Primerjaj svoje strani z podobnimi na spletu. Preverjati pa je dobro začeti že zgodaj v razvoju projekta, saj bo namreč v zgodnjih fazah razvoja lažje narediti

spremembe dizajna. Na tole mora bit testni inženir še posebej pozoren (Shea, 2000), (Ritter, 1999):

- Ali je strani lahko usmerjati?
- Ali je izgled strani dosleden?
- Ali lahko uporabnik enostavno in intuitivno najde iskano informacijo ?
- Ali je na strani kazalo?
- Ali so meniji standardni in na pravih lokacijah?
- So povezave na vse strani?
- Imajo strani povezave na začetno stran?
- So povezave, katere mora uporabnik s klikom izbrati, dovolj velike, da lahko to brez problemov tudi stori?
- Uporabnik vpiše želene podatke, nato se sprehodi po straneh naprej in nazaj. Ostanejo podatki zapisani ali jih mora uporabnik še enkrat vpisati?
- Na strani datoteke, ki jo uporabnik lahko prekopira, je potrebno preveriti, ali je poleg imen datotek zapisana tudi njihova dolžina.
- Za posebne vrste datotek, ki potrebujejo posebne programe za njihovo predavanje mora biti označeno, katere programe je potrebno namestiti in tudi njihovo dolžino.
- Ali je na strani informacija o organizaciji vključno z e-naslovom?

***Performanca in grafika***

Obiskovalci strani pričakujejo hitre rezultate in nočejo dolgo čakati na prikaz strani. Med testiranjem se moramo tega zavedati. Seveda so lahko upočasnitve zaradi dostopa do interneta, ampak teh se morajo programerji zavedati in najti rešitve, da jih zmanjšajo.

Grafika mora biti dovolj majhna, da nebo povzročila problemov z nalaganjem. Potrebno pa je tudi preveriti, če se slike res naložijo.

Poskusimo meriti čas nalaganja strani. Ta čas je čas od klika na povezavo pa do prikaza celotne strani v brskalniku. Stran je v celoti prikazana, ko v desnem kotu brskalnika zagledamo besedico '' done ''. Čas nalaganja moramo preveriti večkrat ob različnih trenutkih in izračunati povprečen čas nalaganja. V ta namen lahko uporabimo tudi avtomatska testna orodja, kot npr. Rational Software's Visual Test.

Testiranje bi moralo vključevati različne metode konekcije na stran. Z različnimi modemi in ISDN (ang. Integrated Services Digital Network paketi). S tem simuliramo uporabnika doma in na delu v pisarni.

Testiranje spletnih strani ni le postranska zadeva, ampak postaja vse bolj kritična za oddelke, ki se ukvarjajo s kvaliteto programske opreme. Kvaliteta poslov se vedno bolj ocenjuje po kvaliteti spletnih strani. Zaradi hitrega razvoja internetne tehnologije se morajo testni inženirji neprestano izobraževati, učiti različnih tehnologij in spoznati kateri brskalniki jih podpirajo.

Počasi se že uveljavlja elektronsko poslovanje, z razširitvijo tega bo testiranje spletnih strani postalo še bolj kompleksno in pomembno.

### *Frekvenca testnih ciklov v projektih z WEB stranmi in aplikacijami*

Razvoj tehnologije pomeni tudi nenehno spreminjanje strani in ponovno testiranje. To pomeni, da imajo projekti več razvojnih in testnih ciklov. Vse to zahteva dobro konfiguracijo strani in začetek testiranja že v zgodnji fazi razvoja projekta. Idealno je, da tester pozna specifikacijo projekta še pred začetkom kodiranja.

Da pa ponavljanje ciklov ni odveč, velja omeniti elektronsko poslovanje. Kompleksnost spletnih strani je vse večja, potreba po varnih straneh pa več kot le nujna za vzdrževanje zaupanja pri stranki. Izguba podatkov, nepredviden vdor v uporabniške podatke in nenazadnje izguba denarja na tekočem računu zaradi napake v aplikaciji je nedopustna. Zato je po vsaki spremembi in dodatni zahtevi stranke ponovno testiranje neizogibno.

# 7. Zaključek

Premalo testiranja je zločin, preveč pa greh. Če se določena funkcionalnost izkaže za tvegano, takrat ji verjetno ne bo posvečeno preveč testiranja in investicija v temeljito testiranje te funkcionalnosti ne bo odveč, saj se pomanjkljivo testiranje direktno izraža v produkcijskem okolju. V tem primeru je odločitev programskega vodje za povečanje števila ur, namenjene testiranju rizične funkcionalnosti, upravičena.

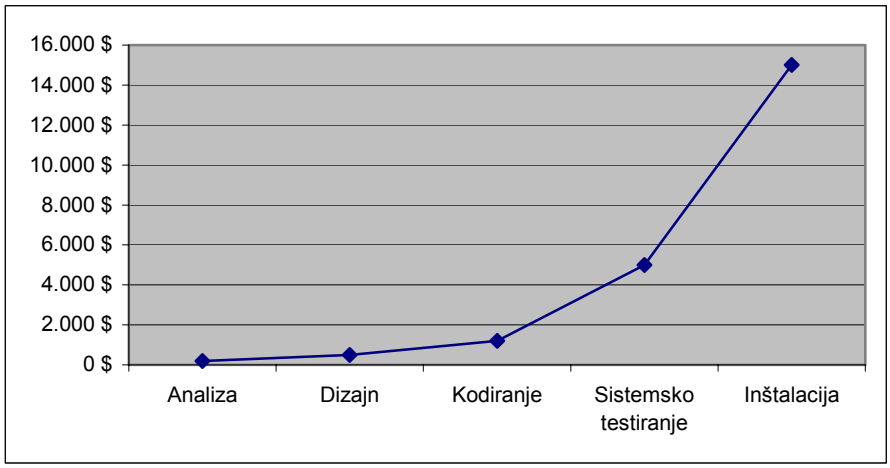
Preveč temeljito testiranje lahko povzroča dodatne stroške, in nepotrebno angažiranje določenih resursov. Tega menedžerjem ponavadi ni potrebno poudarjati in se dobro zavedajo posledic nepotrebnega testiranja. Za obsežnost testiranja se morajo pogajati tudi s stranko, ki nenazadnje plača tudi ta del razvoja, ker želi kvaliteten produkt. Pogosto je dilema o tem ali je testiranje sploh potrebno in želijo število ur testiranja kar se da zmanjšati, medtem ko se o želji pretiranega testiranja le malokdaj sliši.

Cenovno efektivnost testnega procesa je možno določiti le, če znamo meriti njegov učinek. Raziskave organizacije IEEE (vir. Perry, 2000 str.63) so pokazale, da cene odprave napak dramatično naraščajo s časom, v katerem so bile odkrite. Cena odkritja napak je v fazi zbiranja zahtev veliko manjša kot na koncu razvoja projekta ob sistematičnem testiranju.

Zaključki raziskav v podjetjih IBM, GTE in TRV so pokazali, da se za odpravo napake v fazi analize porabi 200 dolarjev, v fazi dizajna 500, fazi kodiranja 1200, fazi sistemskega testiranja 5000 dolarjev in v fazi inštalacije 15000 dolarjev (vir. Perry, 2000 str. 63). Če narišemo graf (slika 7.1), vidimo, da stroški naraščajo eksponentno.

V zadnjih letih je bilo na razvojnem procesu narejenega zelo veliko, tako da je v večini organizacij postalo testiranje razvoja programske opreme neizogibno. Čeprav je včasih testiranja še vedno premalo se počasi tudi programerji zavedajo njegove pomembnosti. Razvite so bile nove in nove metode za povečanje produktivnosti testov.

Slika 2.1: Stroški odkrivanja napak skozi faze v razvoju programa eksponentno naraščajo.




Nekateri naročniki programske opreme že zahtevajo tudi dokumentacijo o izvedenih testih, zato je tudi tej potrebno posvetiti veliko pozornosti.

Izbira pravega orodja za testiranje je pomemben korak v testnem procesu. Orodje za testiranje bi moralo biti izbrano glede na njegove zmožnosti, kako izvrši in zadovolji testnim tehnikam in metodam. Metodologija testiranja se od organizacije do organizacije razlikuje, vendar ima skupne značilnosti. Osnova je na možnih tveganjih in mora biti skrbno premišljena, saj ima le takšna zadovoljiv učinek.

Kvaliteta poslov se vedno bolj ocenjuje tudi po kvaliteti spletnih strani. Zaradi hitrega razvoja internetne tehnologije, se morajo testni inženirji neprestano izobraževati, učiti različnih tehnologij in poznat kateri brskalniki jih podpirajo. Enako velja za mobilno tehnologijo. Tudi pri nas je tretja generacija mobilnih telefonov vse bližje.

Zanimivo bi bilo temeljito preučiti možna tveganja in metodologijo testiranja na primeru kakšnega večjega podjetja za razvoj programske opreme to analizirati, kako je bilo testiranje izvedeno na konkretni aplikaciji.

# 8. Literatura in viri

**Literatura**


1.Ada-Europe International Conference on Reliable Software Technologies; 1999: Reliable software technologies . Ada-Europe'99: proceedings, Berlin, 1999, 449 str.

2.Asam Robert: Qualitätsprüfung von Softwareprodukten: Defieneren und Prüfen von Benutzfreundlichkeit, Wartungsfreundlichkeit, Zuverlässigkeit. Berlin, München : Siemens Aktiengesellschaft, 1987, 336 str.

3.Cabiroy Frank: CNE testing guide for IntranetWare. San Francisco: Sybex cop., 1997, 1035 str.

4.Courtney P.: Testing e-commerce. Applications Development Trends , B.k., Julij 1999, (str. 24-34).

5.Crispin L.: Stranger in a Strange Land: Bringing QA to a Web Startup, B.k., 15. junij, 2001, [URL:http://www.stickyminds.com/docs_index/XUS247559file1.doc ]

6.Driscoll, S. (1997). Systematic Testing of WWW Applications., B.k., 2001, [URL: http://www.oclc.org/webart/paper2/ ]

7.Gerrard, P. Risk-Based E-Business Testing: Part 1 – Risks and Test Strategy.,B.k., (2000a), [URL:http://www.evolutif.co.uk/articles/EBTestingPart1.pdf ]

8.Gerrard, P. Risk-Based E-Business Testing: Part 2 – Test Techniques and Tools.Retrieved B.k., (2000b), [URL : http://www.evolutif.co.uk/articles/EBTestingPart2.pdf

9.Greenberg, Jeff R.: A methodology for developing and deploying Internet and Intranet solutions: Upper Saddle River: Prentice Hall, cop.1998

10.Hagen, M. (2000). Performance Testing E-Commerce Web Systems Presentation Paper, 2001,

URL:http://www.stickyminds.com/docs_index/XDD2070filelistfilename1.doc

11.Hayes, L.: Testing distributed applications: Unraveling the Web. Datamation, 1996, (str.108-114).

12.Ireson, William Grant: HANDBOOK of reliability engineering and management. New York , McGraw-Hil Book Company, 1988

13.Kaner, Cem: Testing computer software: New York : J. Wiley & Sons, cop.1999, 480 str.

14. Kiger, Jack E.: Auditing: Boston, New York: Houghton Mifflin cop., 1997, 918 str.

15. Kung, David C.: Testing object-oriented software, Los Alamitos, IEEE Computer Society , 1998, 269 str.

16.Nguyen Hung Quoc: Testing applications on the Web: New York, J. Wiley & Sons, cop. 2001, 402 str.

17.Nguyen Hung Quoc: Tracking Down a Defect management Tool: SQEThe Software testing & quality engineer Volume3, issue 4 july/august 2001

18.Nielsen, Jakob: Usability inspection methods: New York [etc.] : John Wiley & Sons, 1994, 413 str.

19.Nierstrasz, Oscar: Software engineering - ESEC/FSE '99: Berlin Springer, 1999,

20.MacIntosh, A. & Strigel, W.: "The Living Creature" – TestingWeb Applications, B.k., 2000,
[URL:http://www.stickyminds.com/docs_index/XUS11472file1.PDF ]

21.Ritter, D.: Web Testing Is Not an Oxymoron. Intelligent Enterprise, februar 1999, (str. 66-69).

22.Sabourin Robert: A look at Testing Web Applications with eValid: SQE The Software testing & quality engineer Volume3, issue 4 july/august 2001

23. Samaroo, A., Allott, S., & Hambling, B. (1999). E-ffective Testing for E-Commerce.
[URL:http://www.stickyminds.com/docs_index/XML0471.doc]

24. Shea, B. :Avoiding Scalability Shock. Software Testing & Quality Engineering Magazine, (2000, May/June), (str.42-46).

25. Smith, R. : Behind Closed Doors: What every tester should know about web privacy, Software Testing & Quality Engineering Magazine, (2001, Jan/Feb). (str. 43-48).

26. Weyuker E.& Vokolos, F. : Experience with Performance Testing of Software Systems:Issues, an Approach, and Case Study, IEEE Transactions on Software Engineering, (2000), (str.1147-1156).

27. Wilson J.: Testing in Internet Time, 2001, [URL:http://www.stickyminds.com/docs_index/XDD2540filelistfilenme1.doc ]

28. Perry William: Effective Methods for Software Testing, 2nd ed.New York: J. Wiley & Sons cop., 2000 , 812 str.

29. Perry, William E.: How to test software packages: a step-by-step guide to assuring they do what you want. New York: Wiley, 1986, 321 str.

30.PERRY E. William: Planning EDP Audits. Altamonte Springs: EDP Auditors Foundation, 1981. 127 str.

31. PERRY E. William, KUONG F. Javier: EDP Risk Analysis and Controls Justification. Wellesley Hills: Management Advisory Publications, 1981. 151 str.

**Viri**

1.HERMES SoftLab Business Manual, Release 2,HERMES SoftLab d.d, 2000,
URL: http://grunf.hermes.si/QualityManual/release2/default1.html

2.Rational Software DDTS, URL:http://dirlija.hermes.si:8005/ddts/ ddts_main

3.Test Assistant, URL: http://jadran.hermes.si/testassistant