UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Peter Čebokli

AVTOMATIZACIJA TESTIRANJA KOT KLJUČ AGILNOSTI RAZVOJA PROGRAMSKE OPREME

MAGISTRSKO DELO

Mentor: prof. dr. Franc Solina

Ljubljana, december 2006

Zahvala

Prisrčno se zahvaljujem:

Mentorju prof. dr. Francu Solini, ki me je med izdelavo naloge vseskozi usmerjal in mi pomagal z nasveti ter pripombami.

Mateji Ivančič za pomoč pri izdelavi ilustracij in sploh za podporo.

G. Dariu Madžareviću in podjetju In2 d.o.o. za pomoč pri nabavi literature in orodij za testiranje.

Jaku Gantarju za razlago metodologije RUP in slovnične pripombe.

Mag. Boštjanu Žvanutu za številne nasvete in pripombe.

Ani Vinšek za pomoč pri prevajanju.

In seveda mali Anji, ki me je prisilila, da se celotna zadeva hitreje premakne.

Povzetek

Podjetja se morajo čedalje hitreje prilagajati spremembam in zahtevam, ki jih prednje postavljata trg in sodobno poslovanje. Temu ritmu mora slediti tudi informacijska podpora. Veliko razvojnih projektov programske opreme se kljub uporabi sodobnih razvojnih orodij in metodologij ne konča v predvidenih časovnih, stroškovnih in kakovostnih okvirih. Eden od razlogov za to so med drugim težave zaradi spreminjanja uporabniških zahtev med projektom, kar pri starejših razvojnih pristopih naglo povečuje stroške projekta. V zadnjem času so dokaj popularni agilni razvojni pristopi, ki omogočajo naglo in relativno poceni prilagajanje omenjenim spremembam.

V pričujoči nalogi je prikazan postopni razvoj procesa testiranja in vloga testiranja pri različnih razvojnih pristopih. Pregled se zaključi z agilnimi metodologijami oziroma ekstremnem programiranju kot najvidnejšem predstavniku, kjer je avtomatsko testiranje modulov nujno za izvajanje aktivnosti, kot sta stalno preoblikovanje in integriranje. Raziskani so načini avtomatizacije testiranja modulov in avtomatizacije funkcionalnega testiranja, težave in tveganja pri vpeljavi avtomatizacije ter načini ocene stroškov in koristi avtomatizacije.

Cilj naloge je prikazati avtomatizacijo testiranja kot enega ključnih dejavnikov obstoja agilnih metodologij in preučiti možnosti vpeljave avtomatizacije testiranja na obstoječih sistemih oziroma ob uporabi neagilnih razvojnih pristopov.

Abstract

Modern business demands constant adaptation to changes and the information technology has to cope with that. A lot of software projects fail or end challlenged, despite of using modern tools and methodologies. One of the reasons are also problems with constant changing of user requirements during project. Changes quickly rise the costs of project, especially when using older software developement models. At the moment there are quite popular Agile methodologies which enable fast and relatively cheap adaptation to requirements changes.

In this thesis is outlined the progress of the testing process and the role testing plays in different software developement models. More focus is given to agile methodologies or more preciselly to extreme programming (XP). Automated module (or unit) testing is crucial for performing XP activities like refactoring and continuous integration. Ways of automation of unit and functional testing are outlined, problems and risks involved and methods to evaluate costs and benefits of automation.

The main goal of thesis is to show automated testing as the key existence factor of agile methodologies. Also there are outlined different ways to introduce automation on existing maintained systems and non agile software developement models.

Vsebina

Kazalo slik

Kazalo preglednic

1. Uvod

Na vsak projekt razvoja programske opreme prežijo številna tveganja, zaradi katerih se ta lahko ne konča v predvidenih časovnih, stroškovnih in kakovostnih okvirih, kljub uporabi modernih razvojnih orodij in metodologij. Eden od razlogov za to so težave zaradi spreminjanja uporabniških zahtev med projektom, kar pri klasičnih razvojnih pristopih naglo povečuje stroške projekta. Ta težava je posebej očitna pri linearnih razvojnih pristopih. Pri teh je najbolj znan predstavnik zaporedni ali slapovni razvojni model, kjer si faze razvoja sledijo zaporedno ena za drugo, vsaka faza razvoja pa se lahko prične šele z zaključkom predhodne. Kaj kmalu so začeli iskati drugačne razvojne pristope, katerih skupna značilnost je ta, da se razvoja lotevajo v več korakih ali iteracijah. Vzporedno s tem se je razvijal in spreminjal tudi proces testiranja.

V zadnjem času so dokaj popularni agilni razvojni pristopi, ki omogočajo naglo in relativno poceni prilagajanje spremembam zahtev. Bistvo agilnosti je zmožnost hitrega prilagajanja spremembam, kar agilne metodologije dosežejo z enostavnim načrtovanjem z malo dokumentacije, kratkimi in pogostimi iteracijami ter tesnim sodelovanjem z naročnikom. Tako je po avtorjih ekstremnega programiranja težnja po stalnih spremembah vgrajena v razvojni proces (angl. *embracing change*).

Ekstremno programiranje kot najvidnejši predstavnik agilnih pristopov predpisuje dvanajst najboljših praks ali principov dela. Večina predpisanih praks je poznanih že vrsto let, vendar so jih tokrat prvič povezali v sinergijsko celoto. Prakse so med seboj soodvisne, izvajanje večine pa bi bilo nemogoče brez ustrezne kontrole kakovosti. To zagotavljajo v glavnem dinamične tehnike, predvsem avtomatsko testiranje modulov. Statične tehnike so za agilne metodologije manj primerne, ker bi njihovo izvajanje zahtevalo veliko človeških virov [1]. Avtomatizacija testiranja je pri agilnih metodologijah ključnega pomena, ker brez nje ni možno izvajati večine ostalih priporočenih praks, kot sta npr. stalno integriranje in preoblikovanje. Zaradi tega si avtomatske teste lahko predstavljamo kot lepilo, ki drži agilni proces razvoja skupaj.

Agilne metodologije so v tem času dokaj popularne, imajo pa svoje slabosti in omejitve, zato nekateri predvidevajo razvoj v smeri iskanja ravnovesja med agilnostjo in disciplino [2]. Vendar pa uporaba avtomatskega testiranja ni omejena samo na agilne metodologije. Tako je avtomatsko testiranje tudi del metodologije RUP, ki sicer spada med „težke“ metodologije. S primernimi orodji in pravim pristopom je lahko avtomatizacija testiranja koristna ne glede na uporabljeno metodologijo razvoja programske opreme, še posebej za regresijsko testiranje, s katerim ohranjamo kakovost in stabilnost programske opreme po spremembah.

Ločimo dve veji avtomatizacije, ki ju nameravam bolj podrobno preučiti: že omenjene avtomatske teste modulov in avtomatske sistemske teste. Testi modulov so v

zadnjem času skoraj vedno avtomatizirani in nekateri menijo, da je razvoj testiranja modulov v zadnjih letih najpomembnejši napredek v testiranju nasploh [3]. Za tovrstno testiranje obstajajo številna orodja in knjižnice za skoraj vse popularnejše programske jezike. Treba pa je poudariti, da tovrstni testi ne nadomeščajo ostalih ravni testiranja (npr. sistemskega testiranja).

Druga veja avtomatizacije testiranja je avtomatizacija sistemskih testov, ki temelji na oponašanju človekove interakcije z aplikacijo. Orodja za tovrstno avtomatizacijo so draga, zato je zelo pomembno izbrati pravo orodje, ki bo skladno z ostalimi razvojnimi orodji, in se avtomatizacije lotiti na pravi način. Zmotna je namreč predstava, da ta orodja že po nakupu znižajo stroške testiranja. V resnici jih vsaj na začetku vpeljave avtomatizacije povečajo, zato je treba skrbno pretehtati stroške in koristi. Predlaganih načinov za vpeljavo avtomatizacije testiranja je več, v pričujoči nalogi je predstavljen celovit pristop ATLM [4]. ATLM (angl. *Automated Test Lifecycle Methodology*) je strukturna metodologija, usmerjena k uspešni vpeljavi avtomatizacije testiranja skozi šest faz in množico aktivnosti, ki se izvajajo vzporedno s softverskim razvojnim ciklom.

V prvem delu naloge je prikazana vmeščenost testiranja pri različnih pristopih k razvoju programske opreme, posebej pa pri agilnih metodologijah, kjer predstavlja avtomatsko testiranje temelj vseh ostalih aktivnosti. V drugem delu so opisane različne tehnike avtomatizacije testiranja modulov in avtomatizacije sistemskega testiranja ter celovit pristop vpeljave avtomatizacije ATLM. Cilj naloge je prikazati avtomatizacijo testiranja kot ključni dejavnik obstoja agilnih metodologij in poiskati možnosti vpeljave avtomatizacije tudi pri neagilnih razvojnih pristopih.

2. Agilni razvoj programske opreme

Agilnost je pojem, ki se uporablja na različnih področjih in je z več vidikov predmet raziskav. Poznamo npr. agilno organizacijo, agilno proizvodnjo itd. Agilnost pri razvoju programske opreme razumemo kot zmožnost hitrega in pravilnega odzivanja ter prilaganja na spremembe v poslovnem okolju in na zahteve, ki jih to okolje postavlja. [5]. Agilni proces je torej tisti, ki je zmožen brez težav podpirati takšno stopnjo prilagodljivosti.

Pojem agilnosti je najbolje razložiti na podlagi manifesta, ki ga je leta 2001 predstavilo agilno zavezništvo (angl. *agile alliance*) [6]. Agilno zavezništvo je organizacija, ki združuje predstavnike različnih agilnih metodologij. Ti so med sicer konkurenčnimi agilnimi metodologijami našli skupne vrednote in principe ter jih predstavili v agilnem manifestu (angl. *agile manifest*) [7]. Vrednote agilnega manifesta so naslednje:

- **posamezniki in medsebojne interakcije** imajo prednost pred procesi in orodji,
- **delujoča programska oprema** pred obsežno dokumentacijo,
- **tesno sodelovanje s stranko** pred pogodbenim pogajanjem,
- **odzivanje na spremembe** pred sledenjem planu.

Vrednote na desni so označene kot manj pomembne od tistih na levi. To ne pomeni, da so nepomembne, le poudarek je drugje.

Prva vrednota sprejema procese in orodja kot pomemben člen v razvoju programske opreme. Brez dobrih orodij ni dobrega izdelka, zato je priporočljivo uporabljati vedno najboljša orodja, ki so na voljo. Toda poudarek naj bo bolj na posameznikih in na njihovi medsebojni interakciji oziroma komuniciranju. Posamezniki so enolične osebnosti, ki jih ne moremo popredalčkati v vloge (npr. analitik, programer itd.), ampak jih moramo obravnavati individualno [5]. V ustvarjalnem delovnem procesu, kakršno je razvijanje programske opreme, so posamezniki tudi sicer težje zamenljivi kot pa v kakem drugem, bolj enoličnem delovnem procesu. Zato morajo procesi in orodja podpirati posameznike ter razvojno ekipo in ne obratno.

Podobno velja za dokumentacijo, ki je koristna, vendar nima vrednosti, če hkrati ob njej ni delujočega izdelka. Ta je na koncu projekta edini, ki ima za stranko pravo vrednost. Toda tudi pri agilnih metodologijah se dokumentira in modelira[1], le da v manjšem obsegu [5].

Pogodbe in pogajanja s stranko oziroma naročnikom so potrebna za določanje mej

1 Znan je termin “agilno modeliranje“ (angl. *agile modeling*)

ter obsega projekta, niso pa zagotovilo, da bo projekt uspešen. Ker je na začetku projekta običajno težko dovolj dobro specificirati zahteve ali pa se te zaradi dogodkov v poslovnem okolju spreminjajo, morajo razvijalci tesno sodelovati z naročnikom skozi celoten projekt.

Tudi agilni razvoj potrebuje planiranje. Toda redkokateri plan ostane nespremenjen do konca projekta. Uporabniške zahteve se iz različnih vzrokov spreminjajo in temu se mora hočeš nočeš prilagajati tudi razvojni proces. Vsak plan je zato treba vsaj do neke mere spreminjati in prilagajati trenutnim okoliščinam. Agilne metodologije predvidevajo nujnost sprememb že od vsega začetka, zato so se nanje sposobne hitro odzivati.

Med levo in desno stranjo navedenih vrednot je treba najti ustrezen kompromis, za vsak projekt, razvojno skupino ali podjetje posebej. Dejansko se je ob pojavu agilnih metodologij delalo striktno ločnico na "lahke" in "težke" metodologije v smislu količine dokumentacije, ceremonije (angl. *ceremony*) in rigoroznosti ob razvoju. Toda bistvo agilnosti ni v manjši količini dokumentacije, temveč kot že rečeno v sposobnosti hitrega odziva na spremembe.

Eden od ključnih mehanizmov, ki nam zagotavljajo agilnost pri razvoju programske opreme, je avtomatsko testiranje, kar bom pokazal v naslednjih poglavjih.

3. Metode testiranja

Testiranje je proces, s katerim ugotavljamo pravilnost delovanja, popolnost, varnost in kakovost razvijane programske opreme. Testiranje večkrat enačijo s procesom zagotavljanja kakovosti programske opreme, kar je napačno. Proces zagotavljanja kakovosti je širši pojem, ki poleg testiranja vključuje še nekatere druge organizacijske procese.

Na testiranje lahko gledamo z različnih vidikov. Eden od teh je, ko želimo ugotoviti, ali gradimo programsko opremo na pravi način oziroma ali sploh gradimo pravo programsko opremo. Prvi vidik imenujemo tudi verifikacija, drugi vidik pa validacija [8].

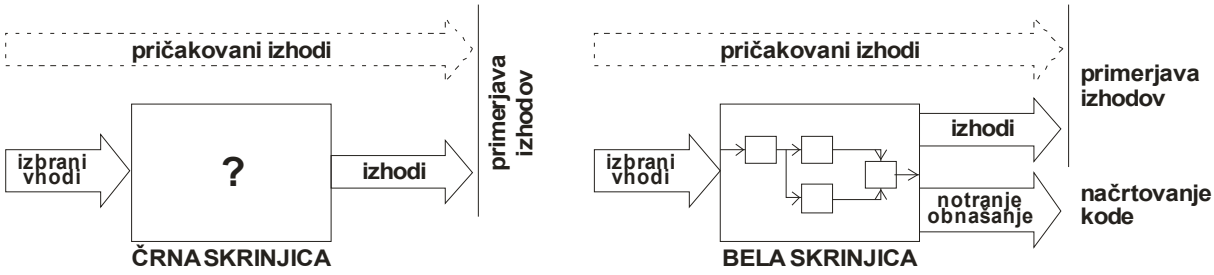
Po načinu testiranja lahko ločimo metode testiranja na:

- statične metode in
- dinamične metode.

Pri statičnih metodah ne izvajamo programov, temveč prebiramo dokumente in programsko kodo. Statične metode so večinoma ročne (npr. branje in sledenje kode), lahko pa tudi avtomatske (npr. preverjanje programske sintakse z orodji ali skladnosti kode s postavljenimi standardi). Pri dinamičnih metodah program izvajamo. Dinamične metode lahko naprej ločimo na tiste, ki zahtevajo prisotnost človeka ob testiranju (npr. preverjanje pravilnosti izpisa poročila na papir) in metode, ki jih lahko popolnoma avtomatiziramo.

Glede na vsebino testiranja ločimo vrste testiranja na

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



Slika 3.1: Testiranje po metodah črne in bele skrinjice

Pri metodah bele skrinjice se osredotočamo na strukturo kode, pri metodah črne skrinjice pa samo na vhode in izhode iz sistema, struktura programske kode je zastrta (slika 3.1). Pri strukturnem testiranju so testni podatki izvedejo iz strukture testirane aplikacije, pri funkcionalnem testiranju pa se testni podatki izvedejo iz formalno ali neformalno specificiranih zahtev. Pri obeh skupinah metod primerjamo dobljene izhode iz sistema s pričakovanimi.

Na sistem torej gledamo z dveh različnih perspektiv. Oba pristopa sta enakovredna in se dopolnjujeta. Uporabljamo ju hkrati na različnih ravneh testiranja. Testiranje bele skrinjice zahteva dostop do izvorne kode aplikacije, ki pa ni vedno na voljo (npr. če v aplikacijo vključimo kupljene komponente). Tako metode črne skrinjice kot tudi nekatere metode bele skrinjice lahko avtomatiziramo.

Ker se v praksi oba pristopa prepletata, nekateri predlagajo nov termin “siva skrinjica” [9], ki naj bi poudaril nujnost uporabe obeh pristopov za uspešno testiranje.

3.1. Metode bele skrinjice ali strukturno testiranje

Z metodami bele skrinjice največkrat testiramo posamezne module. Glavna značilnost teh metod je, da so osredotočene na notranjo strukturo kode v modulih. Cilj testiranja po metodah bele skrinjice je pokriti čim več možnih načinov izvajanja programa s čim manj testnimi primeri [10]. Drugače povedano, gre za to, v kolikšnem obsegu testiranje pokriva programsko kodo. Programska koda, ki se nikoli ne izvaja, je morda nepotrebna in jo lahko odstranimo.

Analiza pokrivanja kode je proces, s katerim [11]:

- odkrivamo področja programa, ki je še nepokrit s testnimi primeri,
- izdelamo dodatne testne primere, s katerimi povečamo pokritost kode,
- določimo kvantitativno količino pokritosti kode, s katero lahko posredno merimo kakovost.

Poleg tega lahko z analizo pokritosti kode odkrijemo redundantne teste, ki ne povečujejo pokritosti programske kode in jih zato lahko odstranimo. Za avtomatizacijo procesa analize pokrivanja kode obstajajo različna orodja, ki nam delo olajšajo in so dobra dopolnitev orodjem za avtomatsko testiranje modulov. Pri uporabi orodij za pokrivanje kode moramo navadno imeti dostop do izvorne kode programov, zato ta orodja uporabljamo že ob izdelavi testne množice primerov. Z analizo pokrivanja kode torej skušamo v večji meri zagotavljati kakovost testnih primerov in posredno s tem tudi kakovost testirane kode.

Poznamo naslednje metode pokrivanja kode oziroma metode bele skrinjice :

- pokrivanje stavkov (angl. *statement coverage*),
- pokrivanje odločitev (vej) in pogojev (angl. *decision, condition coverage*),
- pokrivanje glavnih poti (angl. *path coverage*),
- pokrivanje pretoka podatkov (angl. *data flow coverage*),
- testiranje zank (angl. *loop testing*).

3.1.1. Pokrivanje stavkov

S to metodo preverjamo, ali se ob testiranju vsi stavki izvorne kode programa izvedejo vsaj enkrat. Pogosto jo srečamo pri programih za optimizacijo hitrosti izvajanja. Nekateri tovrstni programi ne potrebujejo dostopa do izvorne kode, ampak lahko delujejo tudi na delno prevedeni kodi (angl. *object code*). Prednost metode je njena enostavnost, glavna slabost pa neobčutljivost na kontrolne strukture v kodi [11]:

Primer:

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

Če je pogoj izpolnjen, potem se stavek2 izvede in vsa koda po metodi pokrivanja stavkov velja za pokrito. V skrajnem primeru lahko zaradi enega samega pogoja dobimo od enega do sto odstotkov pokrito kodo, odvisno od tega, v katerih blokih kode se nadaljuje izvajanje po pogojih. V praksi želimo pri testiranju dobiti nekaj več informacij, kot samo to, ali se nek stavek izvorne kode izvede ali ne, zato se raje opremo na kontrolne strukture v kodi.

3.1.2. Pokrivanje odločitev (vej) in pogojev

Ta metoda je dokaj enostavna in hkrati odpravlja pomanjkljivosti prejšnje. Z njo preverjamo, ali testni primeri pokrivajo vse možne izhode iz pogojev. Pri pokrivanju odločitev se ne oziramo na to, ali je pogoj sestavljen, temveč pogoj jemljemo kot celoto. Pri pokrivanju pogojev preverjamo izhod vsakega logičnega izraza v pogoju posebej. Razlika med obema različicama metode je razvidna iz spodnjega primera:

```
če (pogoj1 IN (pogoj2 ALI funkcija1()))
    stavek1;
sicer
    stavek2;
```

Če obravnavamo zgornji pogoj kot celoto, bomo pokrili vso kodo brez klica funkcije1. Dovolj je, da sta pogoj1 in pogoj2 oba resnična. Če pogoj razbijemo na vsak izraz

posebej, potem moramo v testnih primerih obravnavati tudi funkcijo1. Pokrivanje pogojev je torej različica metode pokrivanja vej, ki je bolj občutljiva na kontrolne strukture.

3.1.3. Pokrivanje glavnih poti

Drugače pravimo tej metodi tudi predikatno pokrivanje [11]. Pri pokrivanju glavnih poti preverimo, ali se vse možne poti skozi programsko kodo izvedejo vsaj enkrat. To storimo tako, da poiščemo vse neodvisne poti skozi kodo in za vsako pot pripravimo testni primer [8]. Poti lahko prikažemo z grafom poteka. Prednost takšnega pokrivanja kode je, da nas pripelje do zelo izčrpnega testiranja. Slabost metode je, da število poti skozi kodo raste eksponencialno [11]. Funkcija z devetimi *if* stavki ima petsto dvanajst možnih poti, če dodamo še en *if* stavek se število poti poveča že na tisoč štiriindvajset. Druga slabost je ta, da se v praksi zaradi odvisnosti med podatki mnoge poti v programu nikoli ne bodo izvedle, kar pomeni veliko število nepotrebnih testnih primerov. Ta slabost je razvidna iz spodnjega primera psevdokode:

```
če (pogoj1)
    stavek1;
stavek2;
če (pogoj1)
    stavek3;
```

Zgornji primer ima teoretično štiri poti prehoda skozi kodo. V resnici sta možni samo dve: takrat ko je pogoj1 izpolnjen in takrat ko ni izpolnjen.

3.1.4. Pokrivanje pretoka podatkov

Metoda predstavlja posebni primer metode pokrivanja poti. Upošteva samo podpoti od deklaracije spremenljivk do njihove uporabe oziroma do sklicevanja nanje [10]. Prednost takega načina merjenja pokritosti je ta, da upošteva način, s katerim program obravnava podatke. Slabost te metode je njena kompleksnost in dejstvo, da metoda ne vsebuje pokrivanja pogojev.

3.1.5. Testiranje zank

Kot eno od metod bele skrinjice je vredno omeniti tudi testiranje zank. Skoraj vsak program vsebuje zanke. Te so večkrat vzrok za drobne in težko opazne napake, ki lahko potegnejo za sabo občutne posledice [8]. Zaradi tega je zankam ob pregledovanju kode dobro nameniti več pozornosti in to upoštevati tudi pri izdelavi testnih primerov.

3.2. Metode črne skrinjice ali funkcionalno testiranje

Pri teh metodah je struktura kode zastrta, opremo se na vhodne in izhodne podatke. Cilj testiranja po metodah črne skrinjice je preveriti, ali se sistem obnaša v skladu s specificiranimi zahtevami. Pri tem skušamo odkriti napake na različnih področjih:

- napačno ali pomanjkljivo delovanje sistema,
- težave z uporabniškim vmesnikom,
- problemi s performansami,
- problemi pri delu z več uporabniki,
- varnostni problemi in
- problemi shranjevanja in uporabe varnostnih kopij itd.

Testiranje izvajajo testerji in programerji, ki pri tem nimajo vpogleda v notranjost sistema. Na podlagi izbranih vhodov v sistem primerjajo izhode iz sistema s pričakovanimi izhodi. Testiranje bi bilo popolno, če bi lahko preverili obnašanje sistema z vsemi možnimi vhodnimi podatki. Ker to zaradi časovne zahtevnosti ni ne možno niti ni smiselno, je treba število testnih primerov omejiti, pri čemer pa želimo ohraniti čim večjo izčrpnost testiranja.

Za pripravo testnih primerov črne skrinjice se uporabljata naslednji metodi:

- metoda ekvivalentnih particij in
- analiza mejnih vrednosti.

3.2.1. Metoda ekvivalentnih particij

Vhodni podatki v sistem so običajno številčna vrednost, interval vrednosti, množica diskretnih vrednosti ali Boolova spremenljivka. Vhodne podatke razbijemo na razrede ali ekvivalentne particije, tako da vanje združujemo vrednosti, za katere ne pričakujemo znatnih razlik v obnašanju sistema. V idealnem primeru bi nato imeli za vsak razred en testni primer [8].

Kot primer vzemimo vhodno spremenljivko, ki lahko zavzame vrednosti v intervalu od sto do dvesto. Po metodi ekvivalentnih particij določimo tri razrede: prvega pod intervalom, drugega nad intervalom in tretjega znotraj intervala. Prva dva razreda sta nepravilna, tretji je pravilen.

Prednost metode ekvivalentnih particij je torej v tem, da z njo zmanjšamo obseg testiranja na omejeno, dobro definirano množico testnih primerov in se tako izognemo ad-hoc pristopu pri testiranju.

3.2.2. Analiza mejnih vrednosti

Izkušnje kažejo, da se več napak pojavlja na mejnih vrednostih intervalov vhodnih podatkov, kot pa znotraj intervala. Metoda analize mejnih vrednosti je komplementarna z metodo ekvivalentnih particij in to dejstvo izkorišča [10]. Pri pripravi testnih vhodnih podatkov se osredotočimo na mejne vrednosti ter vrednosti, ki so tik nad in pod to mejo [12].

V primeru, da je testna množica še vedno prevelika, naredimo dodatno izbiro testnih primerov z analizo občutljivosti. Ohranimo tiste testne primere, ki na izhodu iz sistema povzročijo največji odklon. Tiste, ki dajo podobne rezultate, odstranimo iz testne množice.

Zelo pomembno je pripraviti tudi testne primere za napačne vhodne podatke (t.i. negativni testni primeri). Tako preverimo ali se sistem tudi ob vnosu takšnih podatkov obnaša kontrolirano (npr. ali prikaže obvestilo o napaki, namesto da se nenadzorovano poruši).

3.3. Preverjanje in sledenje kode

Preverjanje programske kode je zelo učinkovit in hkrati relativno drag način za iskanje napak [13]. Formalno recenzijo kode (angl. *inspection*) predstavljajo sestanki, katerih namen je odkrivanje napak, ne pa njihovo popravljanje. Sestanki so formalni, udeleženci na njem imajo točno določene vloge. Zelo pomembno vlogo ima moderator ali vodja sestanka, ki mora poskrbeti za konstruktivno vzdušje, ki se ne sme sprevreči v linčanje programerjev. Preverjanje kode je lahko tudi neformalno. Izvaja se npr. med člani razvojne skupine, pri čemer so cilji lahko podobni kot pri formalni recenziji, lahko pa se dopolnjujejo (npr. izobraževanje novih članov skupine).

Sledenje (angl. *walktrough*) je sprehajanje po programski kodi z uporabo testnih podatkov. Cilj sledenja je iskanje problemov in izmenjava znanja. Za razliko od preverjanja je pri sledenju osrednja oseba avtor programske kode, ki jo predstavi ostalim udeležencem postopka. Tudi s pomočjo sledenja se lahko uspešno odkrijejo napake, še bolj pa je sledenje učinkovito za širjenje znanja.

3.4. Ad-hoc testiranje in ostale metode

Eden od mitov testiranja je tudi ta, da je testiranje vedno rutinski proces, ki ne zahteva kreativnosti, kar je napačna predpostavka [12]. Predvsem pri programerjih velja testiranje za dolgočasno opravilo, ki se mu najraje izognejo, oziroma je to veljalo do uveljavitve avtomatskega testiranja modulov. V resnici testiranje ni zmeraj

rutinska naloga, še posebej ne, če ga želimo opraviti kakovostno. Zahteva znanje, izkušnje in tudi iznajdljivost. Tester z bogatimi izkušnjami testiranja in poznavanjem področja, ki ga podpira testirana aplikacija, bo lahko odkril veliko ključnih pomanjkljivosti in pripravil boljše testne primere kot tester brez izkušenj, ki se opira samo na teoretično znanje.

Zato je lahko zelo koristno tudi ad-hoc testiranje, katerega rezultati so odvisni ravno od znanja, izkušenj in ostalih sposobnosti testerja, ki se bo takšnega pristopa lotil. Pomembno pa je vse tako odkrite napake in testne primere ustrezno dokumentirati in jih vključiti v standardno testno množico.

Dobro je preučiti tudi bodoče uporabnike aplikacije. Kakšne so njihove navade, kako so delali do sedaj, kakšne so bile značilnosti sistema, ki so ga uporabljali. Tako lahko definiramo dodatne testne primere za področja aplikacije, kjer menimo, da bi lahko prišlo do težav.

4. Postopek testiranja

Glede na hierarhijo testiranja oziroma predmet verifikacije delimo vrste testiranj na:

- testiranje modulov,
- testiranje integracije in
- sistemsko testiranje.

Kot poseben pojem testiranja omenjam še regresijsko testiranje in prevzemno testiranje, ki je vrsta sistemskega testiranja.

4.1. Testiranje modulov

Princip testiranja modulov (angl. *unit testing* ali *module testing*) že dolgo obstaja, vendar je v zadnjem času prišlo v ospredje šele s pojavom modernejših metodologij in orodij, ki omogočajo avtomatizacijo testiranja modulov [13]. Modul je najmanjša zaokrožena enota programskega sistema, ki jo testiramo in navadno nastane kot delo enega programerja.

Cilj testiranja modula je verifikacija oziroma preverjanje pravilnosti delovanja modula, kot si ga je zamislil razvijalec. Zaradi tega je testiranje modula tesno povezano s programiranjem. Če obstajajo dobro definirane zahteve na ravni modula, lahko na ravni modula testerji opravijo tudi validacijo.

Testiranje modulov poteka v izolaciji. To pomeni, da preverjanje delovanja modula ne sme biti odvisno od delovanja drugih modulov, s katerimi je povezan. To dosežemo z navideznimi moduli, ki nadomeščajo manjkajoče module. Povezave med moduli in njihova medsebojna interakcija je domena testiranja integracije.

Testiranje modulov je v domeni programerjev in kot tako tesno povezano s strukturo programske kode. Ta lahko predstavlja problem, če se programer ne potrudi pisati takšne kode, ki jo je lahko testirati. Še posebej je to očitno pri avtomatizaciji testiranja modulov. Zaradi tega postaja čedalje bolj pomemben pojem testnosti in testno usmerjenega razvoja. Pri slednjem je bistveno pripraviti testni primer pred kodiranjem modula. Šele nato na podlagi testnega primera programer gradi programsko kodo. Na takšen način se sproti ohranja tudi testnost kode. Več o testnosti in avtomatizaciji testiranja modulov je opisanega v nadaljevanju.

4.2. Testiranje integracije

Če se pri testiranju modulov osredotočamo na vsak modul posebej, nas pri testiranju integracije zanimajo povezave med moduli in njihovo medsebojno prileganje. Testiramo možne interakcije med moduli, vključno s parametri in globalnimi spremenljivkami. Pri objektnem programiranju se interakcija nanaša na prenos sporočil med objekti. Za lažje razumevanje definirajmo pojem komponente.

Vsak modul je komponenta. Integracija dveh ali več komponent je nova komponenta. Dve komponenti sta uspešno integrirani, če sta bili uspešno prevedeni, povezani, naloženi in sta njuna vmesnika uspešno prestala vse integracijske teste. Največja komponenta je celoten sistem. Takrat govorimo o sistemskem testiranju.

Testiranje integracije je morda eden najbolj kritičnih trenutkov v razvoju, ker se takrat prikažejo morebitne težave v komuniciranju med razvijalci oziroma slabo načrtovanje programskega sistema. Razvoj zapletenega programskega sistema zahteva večje število razvijalcev, med katerimi je komunikacija vedno omejena, ne glede na kakovost organizacije. Pojavljajo se komunikacijski otoki, med katerimi je pretok informacij otežen.

Posebej pri slapovnem razvojnem modelu je to vidno ob slabo izvedeni fazi načrtovanja. Moduli bodo zasnovani s pomanjkljivimi vmesniki, kar se bo pokazalo šele ob integracijskem testu. Testi modulov takšnih napak ne bodo odkrili.

Avtomatizacija testa integracije se izvede kot skripta na višji ravni, ki kliče postopoma teste modulov. Teste modulov dodajamo zapovrstjo. Pred tem morajo biti vse napake, odkrite med testiranjem modulov, odpravljene [4].

Obstaja več pristopov k testiranju integracije komponent. Najbolj znani so naslednji:

- Big-bang,
- integracija od spodaj navzgor,
- integracija od zgoraj navzdol in
- kombinirani pristop.

4.2.1. Big-bang pristop

Gre za najbolj enostavno možno integracijo, pravzaprav pri tem pristopu postopek integracije spustimo. Module zložimo naenkrat v sistem in nato vse skupaj testiramo.

Big-bang pristop ni priporočljiv, se pa v praksi vseeno uporablja.

4.2.2. Integriranje od spodaj navzgor

Pri tej metodi začnemo s testiranjem komponent na najnižji ravni hierarhije. Zatem vzamemo komponento, ki je višje in vključuje že testirane komponente. Postopek ponavljamo, dokler niso vse komponente oziroma podsistemi vključeni v testiranje. Pri testiranju te vrste potrebujemo posebne module, ki jim pravimo testni gonilniki (angl. *drivers*). Ti nadomestijo manjkajoče module na višjih ravneh, ki jih še nismo testirali. Potrebujemo jih zato, ker z njimi kličemo testirane komponente in jim prenašamo vhodne podatke.

4.2.3. Integriranje od zgoraj navzdol

Pri integraciji te vrste začnemo na najvišji ravni s kontrolnim podsistemom, ki mu postopoma dodajamo testirane komponente. Manjkajoče podsisteme simuliramo z lupinami (angl. *stubs*).

4.2.4. Kombinirani ali sendvič pristop

Sendvič metoda je kombinacija integracije od spodaj navzgor in od zgoraj navzdol. Sistem si predstavljamo razdeljen na tri ravni: ciljna raven, spodnja in zgornja raven. Integracija poteka od spodaj navzgor in od zgoraj navzdol proti ciljni ravni. Ciljno raven izberemo tako, da potrebujemo čim manj testnih gonilnikov in lupin.

4.3. Sistemsko testiranje

Predmet sistemskega testiranja je popolnoma integriran programski sistem. Pred sistemskim testiranjem mora programski sistem uspešno prestati testiranje modulov in testiranje integracije. V primeru, da je sistemsko testiranje neuspešno, so bile uporabniške zahteve slabo definirane oziroma izvedba slabo načrtovana. Tedaj moramo rekonstruirati sistem, pri čemer vsaj pri neagilnih metodologijah ni bližnjic in hitrih popravkov. Odveč je pripomniti, da se ob tem bistveno povečajo tudi stroški izdelave sistema in podaljša čas do predaje izdelka kupcu.

Obstajajo različne definicije o tem, kaj vse sodi v sistemsko testiranje. Spodaj so navedene najbolj znane metode.

- **Testiranje varnosti sistema**. Z njim preverjamo zaščito sistema pred nepooblaščenim notranjim in zunanjim dostopom in namernim povzročanjem škode. Takšno testiranje je lahko zahtevno in so zanj potrebna specifična znanja in orodja.

- **Testiranje shranjevanja in uporabe varnostnih kopij podatkov**. Pri tej vrsti testiranja poskusimo narediti varnostno kopijo in z njo sistem povrniti v delujoče stanje. To je sicer razmeroma preprosto in jasno definirano opravilo, kjer pa lahko večkrat naletimo na nepredvidene težave. Če se te pojavijo takrat, ko resnično potrebujemo varnostno kopijo, lahko to pomeni katastrofo.
- **Testiranje okrevanja**. Preverjanje, kako hitro in uspešno se sistem povrne v delujoče stanje po izpadu elektrike, okvarah strojne opreme, nepredvidenih izpadih programske opreme itd.
- **Testiranje uporabnosti** ali prijaznosti programske opreme je dokaj subjektivno. V poštev pridejo tehnike, kot so intervjuji z uporabniki, preučevanje njihovih navad, različne raziskave, preverjanje skladnosti sistema s smernicami, ki so že bile določene itd. Za to vrsto testiranja razvijalci in testerji navadno niso primerni [4].
- **Testiranje obremenitve.** Sistem obremenimo z običajno količino transakcij, ki naj bi se pojavila v produkciji. Drugače temu pravimo tudi testiranje zmogljivosti sistema. Sistem postopoma obremenjujemo do stopnje, ko sistemu začne zmanjkovati virov in se čas transakcije poveča do neuporabnosti. V primeru ugotovljenih pomanjkljivosti je treba povečati zmogljivosti strojne opreme sistema ali odpraviti grla v programski opremi.
- **Stresno testiranje.** Stresno testiranje večkrat enačijo s testiranjem obremenitve, čeprav gre za metodi z dvema različnima ciljema. Pri stresnem testiranju sistem namenoma prekomerno obremenimo s transakcijami in mu postopoma odvzemamo vire do točke zloma. Pri tem si želimo, da sistem pade na neškodljiv način, brez izgube ali okvare podatkov. Če je sistem pravilno zasnovan, do takšnih obremenitev pri običajnem delu nikoli ne bi smelo priti. Zato odkrite pomanjkljivosti odpravimo ali pa tudi ne, odvisno od njihove teže in cene popravka.
- **Testiranje skladnosti.** Preverjanje skladnosti programske opreme z drugo programsko opremo, strojno opremo, mrežno opremo, okoljem itd.
- **Testiranje postopka namestitve in odstranitve aplikacije** je preverjanje delnega ali celotnega procesa namestitve aplikacije v ciljno okolje.
- **Testiranje primerljivosti** je primerjava prednosti in slabosti programske opreme, ki jo testiramo, v primerjavi s programsko opremo tekmecev ali katero drugo referenčno programsko opremo.
- **Alfa testiranje** je testiranje programske opreme proti koncu razvoja in je večina funkcionalnosti podprte. Tipično testirajo alfa izdelek testerji programske hiše oziroma izbrani zunanji uporabniki [12].
- **Beta testiranje** je testiranje po koncu razvoja programske opreme, ko je bilo predvideno testiranje že opravljeno in znane napake odpravljene. Cilj je odkriti in

odpraviti čim več preostalih skritih napak pred izdajo končne različice programa. Beta izdelek testirajo končni uporabniki.

4.4. Funkcionalno testiranje

Funkcionalno testiranje (angl. *functional testing*) v okviru sistemskega testiranja je namenjeno validaciji programskega sistema. S funkcionalnim testiranjem želimo preveriti ali sistem izpolnjuje specificirane uporabniške zahteve. Funkcionalni testi so po naravi testi črne skrinjice in jih izvajajo testerji preko uporabniškega vmesnika. Poudarek je na vhodih in izhodih iz sistema. Na delovanje sistema gledamo z uporabniškega vidika in se osredotočamo na zahtevano funkcionalnost sistema. Zanjo želimo, da je čim bližje uporabnikovim pričakovanjem ob prevzemnem testiranju.

4.5. Prevzemno testiranje

S prevzemnim testiranjem (angl. *acceptance testing*) preverjamo podobno kot pri funkcionalnem testiranju ali testirani sistem deluje v skladu z uporabniškimi zahtevami. Razlika je v tem, da ga izvajajo končni uporabniki v okolju podobnem produkcijskemu. Ker se funkcionalno in prevzemno testiranje prekrivata, lahko večinoma uporabimo enake teste. Zato bi morali biti testi, ki se izvajajo na prevzemnem testiranju, vedno vsebovani že v funkcionalnem sistemskem testiranju. S tem se izognemo morebitnim neljubim presenečenjem in poskrbimo, da je prevzemno testiranje zgolj formalnost.

Tukaj se pokaže prednost zgodnjega vključevanja uporabnikov v projekt. Tak uporabnik bo sistem že poznal in ga imel za svojega, kar prevzemno testiranje močno olajša [4].

4.6. Regresijsko testiranje

Tehnika, ki jo lahko izvajamo na katerikoli ravni testiranja. Z njo želimo obvladovati spremembe programske opreme in preprečiti ponovno pojavljanje napak, ki so enkrat že bile odpravljene. Z regresijskim testiranjem (angl. *regression testing*) poganjamo že obstoječe teste, ki jih izvajamo po vsakem popravku aplikacije, bodisi zaradi popravljanja ugotovljenih napak bodisi zaradi dodane funkcionalnosti.

S to tehniko in težnjo testirati čim prej je povezan tudi izraz dimno testiranje (angl. *smoke testing*). To je zbirka regresijskih testov, ki jih poženemo vsakodnevno na

trenutni različici ali gradnji izdelka. Če se “dim” ne pokaže, je to zagotovilo, da trenutna različica programske opreme kljub spremembam v najslabšem primeru ohranja kvaliteto prejšnje [14]. Dimni testi so podmnožica regresijskih, saj pokrivajo le osnovno funkcionalnost sistema. Regresijsko testiranje je obsežnejše kot dimno, z njim zagotavljamo, da so vse lastnosti testiranega sistema v skladu s specificiranimi zahtevami in da ni nepredvidenih sprememb v delovanju.

Prednosti avtomatizacije testiranja so najbolj vidne ravno pri te vrste testiranju, kjer gre za pogosto izvajanje večje množice testov, običajno kar preko noči. V primeru da avtomatizacije testiranja nimamo, je zbirka regresijskih testov seveda občutno manjša, ker za izvajanje potrebujemo več ljudi in časa. Avtomatsko regresijsko testiranje je koristno med razvojem programske opreme, še bolj pa v fazi vzdrževanja, ko je sistem v uporabi in ga je treba spreminjati, tako da so spremembe za uporabnike čim manj moteče.

5. Pristopi k razvoju programske opreme in testiranje

V računalništvu se je od začetnih časov, ko se je programiralo na t.i. način “kodiraj in popravi” (angl. *code and fix*), kaj kmalu pokazala potreba po bolj celovitih pristopih k razvoju programske opreme [15]. Postopoma so se pojavili različni razvojni modeli, katerih cilj je vedno enak – podpreti razvoj programske opreme tako, da bo končni izdelek ustrezen in dovolj kakovosten ter bo nastal v okviru predvidenega časa in stroškov. Iz različnih razvojnih pristopov so se sčasoma oblikovale celovite metodologije, ki so na osnovi izbranega pristopa natanko definirale metode, tehnike in proces razvoja programske opreme oziroma informacijskih sistemov.

Novi pristopi in metodologije so nastajali in nastajajo, zgoraj omenjeni cilj pa še vedno ostaja nedosežen. Raziskave kažejo, da je še vedno veliko projektov razvoja programske opreme neuspešnih ali delno uspešnih, z veliko zamudo in prekoračitvijo sredstev. Po raziskavah podjetja Standish Group, ki zbira podatke o IT projektih od leta 1994, se je število uspešnih projektov sicer povečalo (v letu 2004 na skoraj trideset odstotkov v primerjavi s katastrofalnimi šestnajstimi v letu 1994), več kot polovica projektov pa je še vedno le delno uspešnih, s prekoračitvijo rokov in predvidenih sredstev [16]. Čeprav se vsaka nova modna metodologija navadno pojavi kot čudežno zdravilo in končna rešitev, ostaja dejstvo, da takšna metodologija ali najboljši pristop zaenkrat ne obstaja.

Izbor prave metodologije razvoja programske opreme je odvisen od različnih dejavnikov, ki presegajo namen te naloge. V nadaljevanju bodo prikazane osnovne značilnosti izbranih razvojnih modelov oziroma metodologij in pomen testiranja oz. avtomatizacije testiranja pri vsakem od teh. Tu se kažejo velike razlike med klasičnimi planskimi metodologijami in novejšimi agilnimi metodologijami. Predvsem želim prikazati potek razvoja procesa testiranja in vedno bolj ključno vlogo, ki ga testiranje zavzema pri sodobnih razvojnih pristopih, še posebej agilnih.

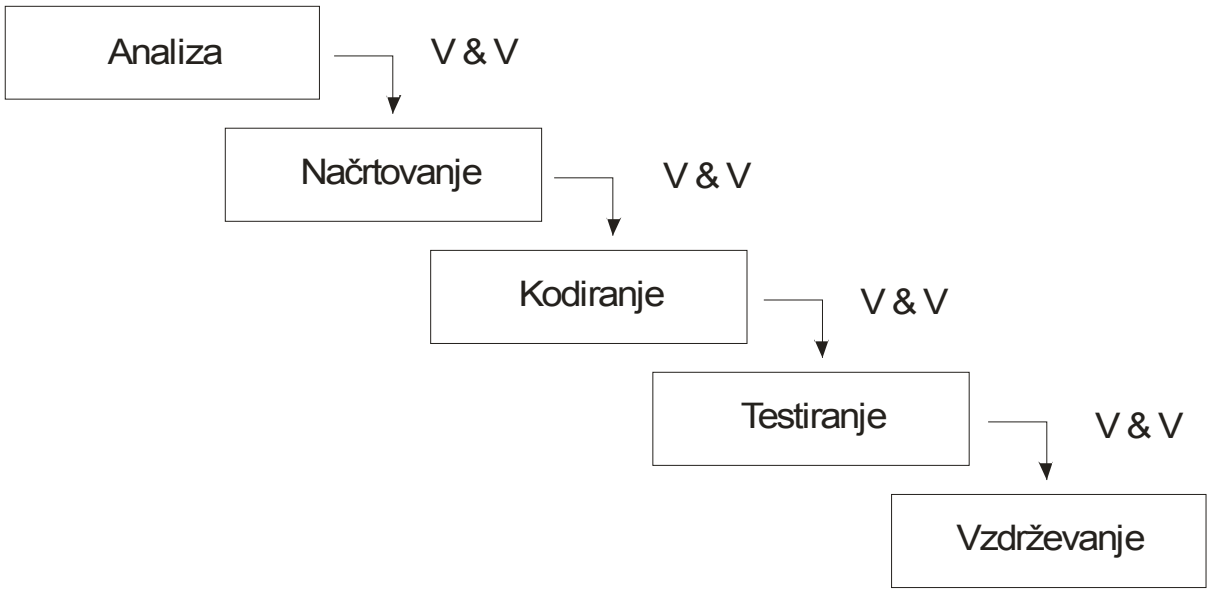
Obstaja več različnih delitev razvojnih modelov. Tako lahko razvrstimo pristope k razvoju programske opreme na [17]:

- linearne razvojne pristope,
- iterativne razvojne pristope in
- agilne razvojne pristope.

5.1. Linearni razvojni pristopi

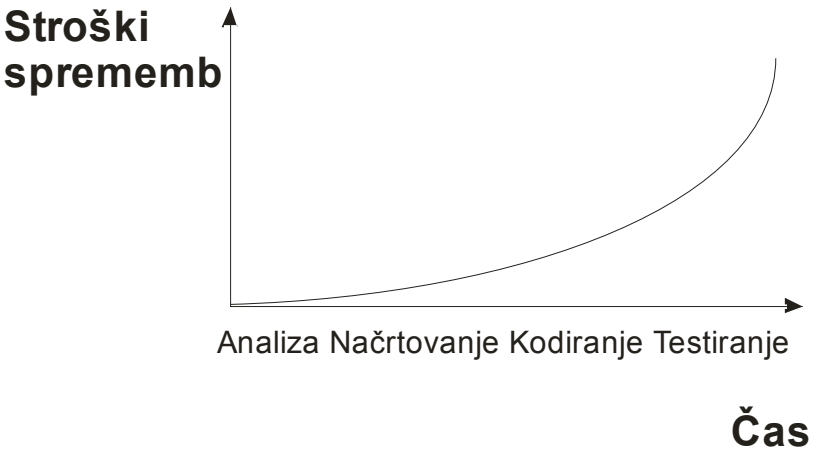
Najbolj znan model razvoja programske opreme je zaporedni ali slapovni model.

Nastal je v sedemdesetih letih prejšnjega stoletja. Čeprav je ta pristop zastarel in zaradi svojih slabosti deležen precej kritik, izkušnje kažejo, da veliko podjetij še vedno uporablja katero od njegovih različic [18].



Slika 5.1: Slapovni razvojni model

Pri linearnem razvojnem pristopu si faze razvoja sledijo ena drugi, vsaka faza se lahko prične šele po zaključku predhodne (slika 5.1). Vsaki fazi sledi postopek verifikacije in validacije (V & V). V praksi se je iz različnih razlogov skoraj vedno treba vrniti v katero od prejšnjih faz, npr. zaradi pomanjkljivo opravljene predhodne faze, težav pri komuniciranju, spreminjanja zahtev uporabnikov itd. Te težave, ali bolje rečeno narava razvoja programske opreme, so bolj ali manj vedno prisotne in se jim ne moremo popolnoma izogniti. S spreminjanjem uporabniških zahtev v kasnejših fazah in dodelavami, ki so zaradi tega potrebne, se stroški sprememb razvijanega sistema eksponencialno povečujejo [17]. (slika 5.2).

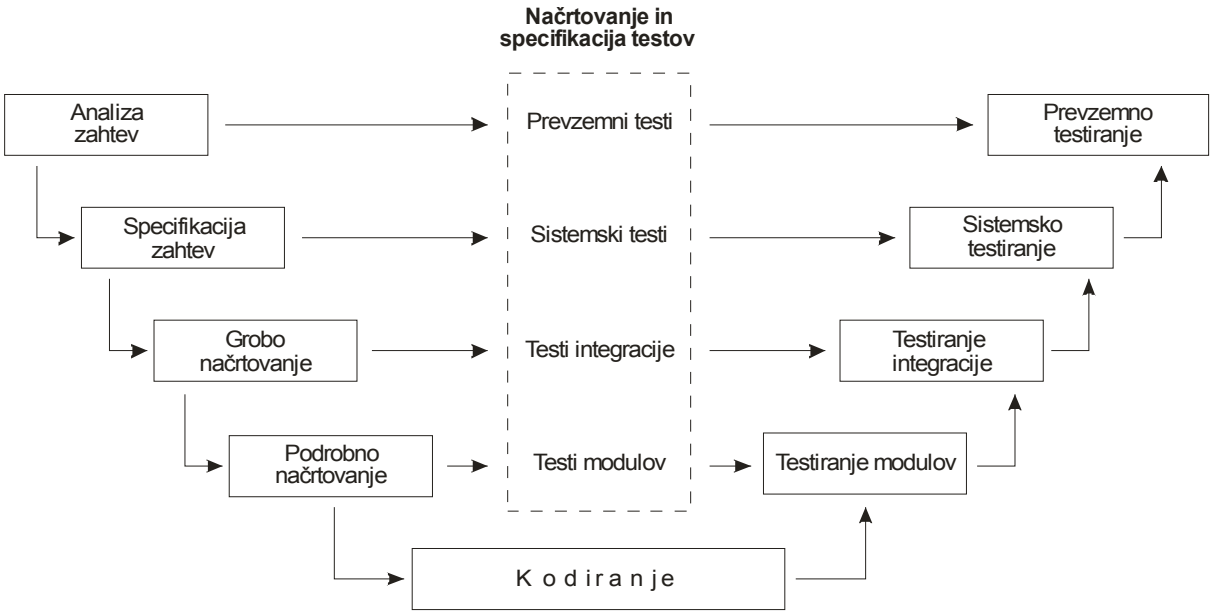


Slika 5.2: Stroški sprememb pri linearnem modelu

Linearni modeli in testiranje

Testiranje je pri slapovnem pristopu samostojna faza, ki sledi fazi kodiranja. Običajno testiranje izvaja posebna skupina testerjev, ločena od razvojne skupine. Sam postopek testiranja je navadno predpisan v množici dokumentov, ki definirajo kaj je treba narediti, ne pa kako. Uporabimo lahko vse že omenjene metode in tehnike testiranja. V praksi je za slapovni model značilno zanemarjanje testiranja, ko projekt zamuja. Posledica tega je zagotovo nižja kakovost izdelka, pri čemer je dejanski prihranek časa vprašljiv.

V-model je izboljšana različica čistega zaporednega modela. Bistvena razlika v primerjavi z osnovnim zaporednim pristopom je ta, da se aktivnosti v zvezi s testiranjem ne izvajajo šele na koncu, ampak so porazdeljene nad vse faze in se opravljajo takoj, ko je to mogoče [18]. Že v fazi analize pričnemo z izdelavo plana in strategije testiranja. Prav tako se v fazi analize izvaja specifikacija in načrtovanje prevzemnih ter sistemskih testov, najbolje vzporedno z analizo in najkasneje po zajemu zahtev. S tem skušamo odkriti pomajkljivosti v čim zgodnejši fazi razvoja in tako obvladovati tveganja ter znižati stroške zaradi spreminjanja in vračanja v prehodne faze. Sledi priprava testov integracije in modulov v fazi načrtovanja. Sicer vse faze potekajo podobno kot pri slapovnem modelu zaporedno. Vsaka naslednja faza se začne po zaključju predhodne. Izvajanje testiranja se tako prične za fazo kodiranja. Dodatna izboljšava V modela je W model (slika 5.3), ki skuša bolje povezati opravila testiranja, razhroščevanja in spreminjanja kode. Po izvajanju testiranja modulov je treba odpraviti napake. To je delo razvijalcev, ne testerjev, zato je nujno tesno sodelovanje med tema dvema skupinama, čemur W-model daje poseben poudarek [19]. Po vsaki spremembi kode je potrebno ponovno testiranje. Tako imamo manjši ponavljajoči se cikel, od testiranja, razhroščevanja, spreminjanja do ponovnega testiranja. Ta cikel se vrti na vseh ravneh testiranja od testiranja modulov do sistemskega testiranja.



Slika 5.3: Linearni razvojni model "W"

Linearni modeli imajo kljub izboljšavam očitne slabosti. To je predvsem togost ob skoraj vedno potrebnem vračanju v predhodne faze. Treba pa je upoštevati, da je bil tak pristop v preteklosti najbolj ekonomičen, ker so bile vse aktivnosti od načrtovanja do kodiranja in testiranja zelo zamudne [8]. Šele nova, boljša orodja za podporo razvoja programske opreme (avtomatsko testiranje!) so omogočila drugačne pristope in metodologije. Linearni razvojni cikel je lahko še vedno dobra izbira takrat, ko gre za rutinski projekt na področju, na katerem ima razvojna ekipa dovolj izkušenj.

5.2. Iterativni razvojni pristopi

Pri zaporednem razvojnem pristopu je težko že na začetku nedvoumno in popolno definirati zahteve. Zato so začeli razvijati drugačne pristope, ki bi bili bolj prilagodljivi in s katerimi bi se dalo lažje obvladovati tveganja. Nastali so različni razvojni modeli, katerim je skupno to, da ne skušajo z enim zamahom priti do dokončnega izdelka, ampak se končni različici izdelka približujejo postopoma, skozi več iteracij.

Eden od načinov prehoda na iterativni razvojni model je uporaba načela deli in vladaj. Celotni sistem razbijemo na več delov, imenujemo jih tudi inkrementi, in vsakega razvijamo posebej z uporabo slapovnega razvojnega cikla. Tako z vsako iteracijo razvijemo del funkcionalnosti celotnega sistema, pri čemer se v začetnih iteracijah lotimo najbolj tveganih delov sistema. S tem kritične pomanjkljivosti odkrijemo in odpravimo dovolj zgodaj ter zmanjšamo tveganje prekoračitve

predvidenih stroškov projekta. Razvijalci so osredotočeni na konkretne, vidne izdelke oziroma programske izdaje (angl. *software release*), ki so pogoj za zaključitev vsake iteracije. Za uspeh na projektu je med drugim zelo pomembna povratna informacija s strani naročnika ali uporabnikov po vsaki zaključeni iteraciji. Končno je uspešnost takšnega pristopa odvisna od težav pri integraciji oziroma združevanju posameznih sklopov v enoten sistem. Za fazo testiranja velja pri opisanem inkrementalnem razvojnem modelu enako kot pri slapovnem razvojnem modelu. Testiranje je torej ločena faza za fazo kodiranja, zaradi kritičnosti združevanja posameznih sklopov v enoten sistem pa je še posebej pomembno testiranje integracije in sistemsko testiranje.

Ne glede na izbrani razvojni pristop je eden trših orehov pri razvoju programske opreme zajemanje uporabniških zahtev. Večina uporabnikov si težko predstavlja končno aplikacijo na podlagi tekstovnih opisov ali diagramov v začetnih fazah projekta. Ena od možnosti iterativnega razvoja in sredstvo za lažjo komunikacijo med uporabniki in razvijalci izdelka je uporaba prototipov. Z njimi uporabniki lažje dobijo predstavo o bodočem izdelku, razvijalci pa povratno informacijo, če so na pravi poti. Prototipi so lahko samo sredstvo za zajemanje zahtev in jih kasneje zavržejo, lahko pa se skozi več iteracij oblikujejo v končni izdelek [8].

Med bolj poznane iterativne modele se uvršča Spiralni model, ki v vsaki iteraciji namenja poseben poudarek analizi in obvladovanju tveganj [5]. Ena od bolj popularnih novejših iterativnih metodologij pa je objektna metodologija podjetja IBM (prej podjetja Rational) Rational Unified Process (RUP).

5.2.1. Rational Unified Process

Rational Unified Process (RUP) je novejša objektna metodologija razvoja programske opreme, ki jo je razvilo podjetje Rational. Temelji na modelirnem jeziku UML in ima kot vse metodologije natančno definirane metode, tehnike in proces razvoja. Poleg tega se naslanja na nekatere najboljše izkušnje (angl. *best practices*) podjetij pri razvoju programske opreme. Te so [20]:

- iterativni razvoj,
- vodenje zahtev,
- uporaba komponent,
- preverjanje kakovosti programske opreme,
- vizualno modeliranje sistema,
- vodenje sprememb.

Iterativni razvoj

RUP spada med iterativne razvojne modele, kar pomeni, da podpira razvoj programske opreme skozi več iteracij. Iterativni razvoj ima več že omenjenih prednosti pred linearnimi pristopi.

Vodenje zahtev

Zahteve je treba sistematično zbirati, urejati in dokumentirati. Vzdrževati je treba povezavo med njimi in ostalimi projektnimi izdelki. Zahteve morajo odločilno vplivati na vse kasnejše faze, na načrt, izvedbo in testiranje programske opreme, ker s tem povečamo verjetnost, da bo končni sistem ustrezal uporabniku.

Uporaba komponent

Da bi zmanjšali zapletenost sistema, je priporočljiva uporaba komponent. To so netrivialni moduli oziroma podsistemi, ki opravljajo določeno funkcijo.

Vizualno modeliranje sistema

RUP poudarja grafični prikaz modela sistema z jezikom UML. UML je grafični jezik za vizualizacijo, specificiranje, izdelavo in dokumentiranje artefaktov programskega sistema [20]. Na ta način naj bi bil sistem bolj razumljiv in komunikacija lažja, tako med razvijalci, kot tudi med razvijalci in uporabniki.

Preverjanje kakovosti programske opreme

Kakovost izdelka je treba preverjati z različnih vidikov: z vidika funkcionalnosti, zanesljivosti in vidika zmogljivosti sistema. Ocenjevanje kakovosti je vgrajeno v proces, v vse faze in vključuje vse sodelujoče na projektu.

Vodenje sprememb

Poskrbeti je treba za učinkovito vodenje sprememb (angl. *configuration management*) in sicer v zahtevah, tehnologiji, virih, izdelkih, platformah ipd. Vse spremembe je treba tudi učinkovito uskladiti med vsemi razvojnimi skupinami, izdajami, izdelki, platformami ipd.

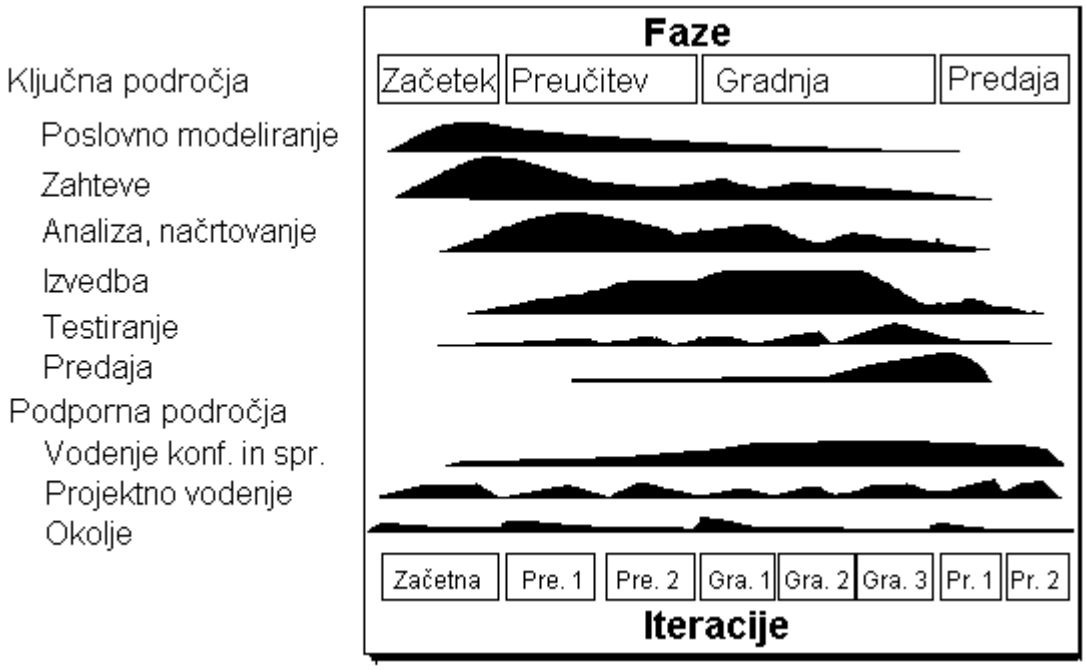
Faze razvoja RUP

RUP lahko opišemo dvodimenzionalno: časovno in vsebinsko (slika 5.4). Vodoravna os predstavlja čas in vidike procesa, ki so povezani z njegovim življenjskim ciklom. To dimenzijo lahko opišemo kot faze in iteracije. Ločimo naslednje faze [20]:

- začetek – razumeti moramo, kaj nam je zgraditi,

- preučitev – razumeti moramo, kako graditi,
- gradnja – izdelava različice beta izdelka,
- prehod – izdelava končne različice izdelka.

Navpična os predstavlja vsebino in prikazuje področja (discipline), ki tvorijo logično vsebino procesa. Poudarek se po področjih časovno močno spreminja. V začetnih iteracijah je več pozornosti posvečene zahtevam, kasneje pa seveda sami izvedbi. Vodenja sprememb, okolja in projekta pa se izvajajo neprestano. Treba je poudariti, da mora biti vsako področje upoštevano v vsaki iteraciji.



Slika 5.4: Faze razvoja RUP

Testiranje pri RUP

Kot je razvidno na sliki 5.4 je testiranje pri RUP prisotno skozi celoten časovni okvir projekta, v vsaki iteraciji razvoja. Na ta način lahko odkrijemo napake dovolj zgodaj, kar poceni njihovo odpravljanje. Testiranje je podobno kot ostala področja razdeljeno na več faz in aktivnosti, kar imenujemo tudi testni cikel [21]. Vsaka iteracija lahko vsebuje več testnih ciklov, večinoma pa samo enega. Pogoj za začetek izvajanja testnega cikla so uspešno prevedeni vsi programski moduli ali z drugimi besedami uspešno zgrajen programski paket. RUP priporoča naj gre vsaka nova gradnja (angl. *build*) ali novo prevedeni programski paket v testni cikel. Včasih to ni mogoče, ker testni cikel zahteva več časa, kot ga je na voljo med posameznimi gradnjami (npr. pri dnevnem prevajanju vseh modulov). V tem primeru gre v testni cikel vsaka N-ta

gradnja.

Pred vsakim testnim ciklom mora biti definiran in vsem vpletenim razumljiv namen ter cilji testiranja. Cilji testiranja naj sledijo ciljem vsake iteracije oziroma nove različice. Še posebej je to pomembno v situacijah, ko so cilji medsebojno izključujoči. Cilji naj bodo enostavno razumljivi in jasni, kot npr.: “Testirati ustreznost specificiranim zahtevam“ ali “Iskati napake, ki sesujejo sistem“ itd. Če ekipa za testiranje nima jasno definiranih ciljev, potem navadno opiše svoje delo kot “Mi pač testiramo“ ali pa “Testiramo vse mogoče“ [21].

Poleg že omenjenega lahko testiranje pri RUP strnemo na še nekaj naslednjih principov [22]:

- iterativni razvoj testnih primerov,
- sledenje testne dokumentacije razvoju,
- celovit pristop k testiranju in
- avtomatizacija testiranja.

Iterativni razvoj testnih primerov

Aktivnosti v zvezi s testiranjem naj se začnejo zgodaj in to ne samo z načrtovanjem testiranja ampak tudi s pripravo testov in samim izvajanjem testiranja. Testni primeri pri RUP navadno izhajajo iz primerov uporabe (angl. *use case*). Primeri uporabe so razmeroma učinkovit in preprost način zajemanja uporabniških zahtev. Predstavljajo interakcijo med akterji in sistemom za dosego določenega poslovnega ali drugega cilja. Akter predstavlja vlogo uporabnika v sistemu, lahko pa je to tudi zunanji sistem ali drugi zunanji dejavnik, s katerim sistem komunicira.

Med načrtovanjem testiranja pretvorimo primere uporabe v testne primere [21]. Vsak primer uporabe je osnova za eno ali več poti oziroma scenarijev, ki jih bo uporabnik “preigraval“ pri vsakdanji uporabi sistema. Poleg normalnih scenarijev moramo predvideti in obravnavati tudi robne primere. Zatem za vsak scenarij zgradimo najmanj en testni primer in določimo pogoje, ki ga bodo “izvedli“. Za vsak testni primer določimo vrednosti testnih podatkov, s katerimi bomo testirali.

Oglejmo si zgornji postopek na naslednjem preprostem primeru spletne aplikacije (preglednica 1):

Primer uporabe	Prijava novega uporabnika v spletno trgovino
Opis	Za prijavo v spletno trgovino mora uporabnik predhodno izpolniti obrazec z osebnimi in drugimi podatki. Vnos zaključi s pritiskom na gumb.
Predpogoji	Brez prepogojev
Scenarij 1	Podatki so pravilno in popolno vnešeni. Postopek je zaključen po običajni poti.
Scenarij 2	Podatki so nepopolno vnešeni. Izpiši napako in zahtevaj ponoven vnos.
Scenarij 3	Uporabnik s temi podatki že obstaja v sistemu. Izpiši napako in končaj.

Preglednica 1: Prikaz enostavnega primera uporabe

Iz scenarijev zgornjega primera uporabe sledi izdelava testnih primerov in priprava testnih podatkov (preglednica 2):

Testni primer	**Scenarij/ Pogoj**	**Ime**	**Priimek**	**E-naslov**	**...**	**Pričakovani rezultat**
Test 1	Scenarij 1	Janez	Novak	janez@naslov.si		Izpis sporočila o uspešnem vnosu in vhod v trgovino.
Test 2	Scenarij 2	Janez	Novak			Izpis sporočila o manjkajočih podatkih.
Test 3	Scenarij 3	Jože	Vnešen	joze@naslov.si		Izpis sporočila o že obstoječih podatkih. Prekini vnos.

Preglednica 2: Matrika testnih primerov s testnimi podatki

Primeri uporabe so sredstvo za zajemanje uporabniških zahtev in jih definiramo že v fazi preučitve, zato se lahko tudi priprava testnih primerov začne zgodaj. Na ta način lahko ekipa za testiranje sledi implementaciji posameznih primerov uporabe in sproti pripravlja ustrezne testne primere. Seveda je poleg omenjenih testov po vsaki iteraciji treba narediti tudi ostala predvidena testiranja (kot npr. stresno testiranje), ki jih načrtujemo posebej. Po vsaki razvojni iteraciji se na novo razviti testni primeri dodajo v množico regresijskih testov. S tem dobi razvojna ekipa pomembno povratno informacijo, pa tudi sami testi se lahko sproti razvijajo s tem, ko se problemski prostor jasni in se sistem vedno bolj približuje končni podobi.

Sledenje testne dokumentacije razvoju

Natančno načrtovanje testiranja naj se izvaja za vsako iteracijo posebej, vendar v smiselnem obsegu, sledi naj potrebam razvojne ekipe in ciljem vsake iteracije. Npr. v fazi preučitve, kjer se osredotočamo na arhitekturo sistema, se tudi pri načrtovanju in dokumentiranju testov osredotočimo na arhitekturo. Pripravimo lahko načrte za testiranje zmogljivosti, ne glede na to, da je npr. uporabniški vmesnik šele v zgodnji

fazi razvoja. Pri tem se izogibamo birokratizaciji in dokumentiramo toliko, da je razmerje med koristmi in stroški čim večje. To mejo določimo v praksi na podlagi izkušenj.

Celovit pristop k testiranju

Testi naj ne izhajajo samo iz uporabniških zahtev, ampak tudi iz drugih virov. Razlog je v tem, da je v uporabniških zahtevah večinoma navedeno, kaj naj sistem počne, ne pa česa naj ne počne. Treba je zamenjati zorni kot in se osredotočiti tudi na nepredvidene situacije in možne napake. Ta princip izhaja iz splošne prakse dobrega testiranja, po kateri je treba testirati tudi negativne testne primere.

Avtomatizacija testiranja

RUP daje pri celotnem razvoju močan poudarek orodjem in vizualizaciji. Uporaba primernih orodij je zelo priporočljiva tudi ob testiranju. Dejstvo je, da je veliko tehnik testiranja nepraktičnih brez ustreznih orodij in se zato tudi ne izvajajo (ali pa se izvajajo površno). RUP zato priporoča uporabo množice orodij, ki nam delo olajšajo. To so orodja za generiranje testnih primerov, avtomatsko testiranje modulov, avtomatsko funkcionalno testiranje, orodja za merjenje pokritosti kode s testi itd. Tu se RUP delno prekriva z agilnimi razvojnimi pristopi, ki se ravno tako močno naslanjajo na uporabo orodij za podporo testiranju. Razlika je v tem, da so orodja, ki podpirajo RUP, relativno draga in kompleksna, podpirajo močno dokumentiranje in vizualizacijo procesa, agilne metodologije pa se, v duhu “lahkosti“, naslanjajo na prostokodno in cenejše programje.

5.3. Agilni razvojni pristopi

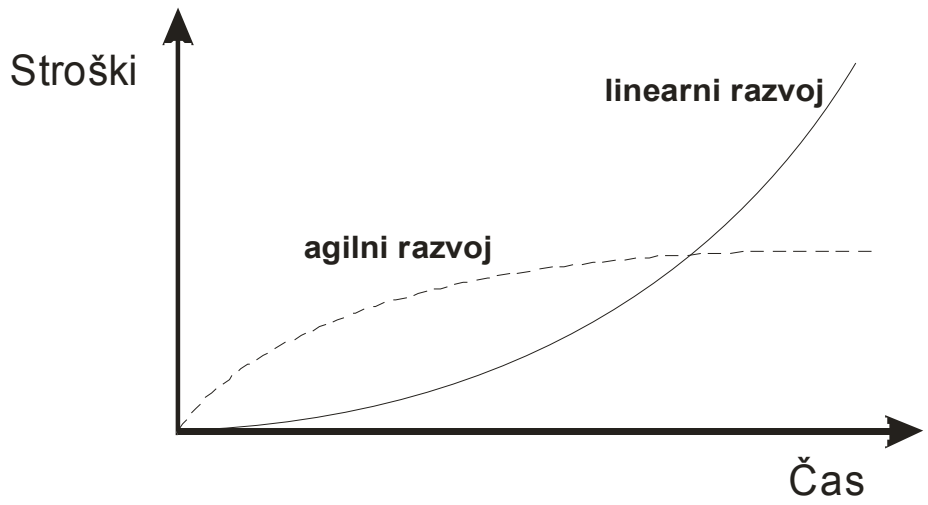
Agilne metodologije razvoja programske opreme so nastale v devetdesetih letih prejšnjega stoletja kot odgovor na “težke metodologije“, takšne kot je npr. slapovni model ali RUP. Kritiki so težke metodologije videli kot neživljenjske, počasne, birokratske in v nasprotju z dejanskim načinom dela programerjev. Predlagali so drugačen način dela, se organizirali v Agilno zavezništvo [6] in sklepe objavili v že omenjenem manifestu [7], ki ostaja skupni imenovalec vseh agilnih metodologij.

Poglavitne agilne metodologije metodologije so:

- ekstremno programiranje,
- Scrum,
- Sprint,
- Crystal Clear in

- DSDM.

Agilnost bi lahko prevedli tudi kot gibkost. Bistvo agilnosti je zmožnost hitrega prilagajanja spremembam, kar agilne metodologije dosežejo z bistveno poenostavljenim procesom razvoja in z manj dokumentacije. Zaradi tega jih imenujejo tudi lahke metodologije. Agilne metodologije so iterativne, vendar so izdaje različic veliko bolj pogoste kot pri običajnih iterativnih pristopih (tedenske ali pa še bolj pogoste). Poleg tega so iteracije opredeljene bolj časovno kot vsebinsko. Omejene so s časom, morebitne aktivnosti, ki niso bile zaključene iz različnih vzrokov v predhodni iteraciji, se premaknejo v naslednjo iteracijo. Naslednja pomembna lastnost agilnih metodologij je, da se ne skušajo izogniti pogostim spremembam zahtev, ampak jih že v osnovi podpirajo. Stroški sprememb so tako pri agilnih pristopih bistveno lažje obvladljivi kot pri linearnih pristopih (slika 5.5). Načrtovanje je kratkoročno in minimalno. Bodočega sistema ne skušajo do potankosti definirati in modelirati, ampak je že na začetku razvoja v ospredju kodiranje.



Slika 5.5: Stroški sprememb pri agilnem razvoju

Pogoste spremembe zahtevajo posebno organizacijo dela in uporabo posebnih orodij, to pa je tudi eden od razlogov, da so se takšni razvojni pristopi pojavili razmeroma pozno. Ostali pomembni dejavniki, ki so omogočili nastanek agilnih metodologij so povečanje moči procesiranja in drugih strojnih zmogljivosti, kar je omogočilo časovno znosno pogosto integriranje in poganjanje množice regresijskih testov. Novejša programerska orodja s tesno integriranimi orodji za testiranje so omogočila tesnejšo in enostavnejšo povezavo med kodiranjem in testiranjem (predvsem testiranjem modulov). Nenazadnje tudi pravilna uporaba modernih objektnih programskih jezikov omogoča večjo granulacijo in lažje spreminjanje kode ter s tem tudi lažje pisanje avtomatskih testov modulov.

Testiranje in avtomatizacija testiranja imata pri agilnih pristopih ključno vlogo. Če se o nekaterih drugih agilnih principih še da razpravljati o njihovi koristnosti in potrebnosti, tega glede avtomatskega testiranja pri agilnih metodologijah ne moremo storiti. Brez avtomatizacije testiranja agilni pristopi padejo na raven že omenjenega pristopa (ali bolje rečeno nepristopa) “kodiraj in popravi“.

5.3.1. Ekstremno programiranje

Ekstremno programiranje je verjetno najbolj znana agilna metodologija in je vsekakor veliko prispevala k popularnosti samega agilnega pristopa. Izraz ekstremno ali skrajno programiranje sicer zveni nekoliko “skrajno”, ideja same metodologije in posledično tudi izvor njenega imena pa je v bistvu preprosta: zbrati skupaj najboljše programerske izkušnje in jih brezpogojno uporabljati. Tudi sicer so poimenovanja v tej metodologiji za nepoznavalce čudna in včasih zvenijo neresno (npr. načrtovalna igra, pripovedovanje zgodb...). Najbrž je bil eden od ciljev avtorjev tudi preprečiti mistifikacijo izrazov in metod. V računalništvu smo namreč že navajeni na vedno nove modne besede in kratice.

Razvoj metodologije ekstremnega programiranja (XP) se je začel sredi devetdesetih let, ko so Kent Beck, Ron Jeffries in Ward Cunnigham delali za podjetje DaimlerChrysler [5]. Ponovno so oživili propadli programski projekt in poskusili delati na drugačen način. Bili so uspešni in iz njihovih izkušenj je nastala metodologija ekstremnega programiranja. Kot osnovne predpogoje za uspešen razvoj programske opreme so izpostavili naslednje vrednote:

- izboljšati komuniciranje,
- iskati preprostost,
- dobiti povratno informacijo in
- nadaljevati s pogumom.

Najboljše izkušnje ekstremnega programiranja

Ekstremno programiranje temelji na kopici najboljših principov oziroma izkušenj in drugih vodil. Temeljnih principov je dvanajst, tem postopoma (metodologija se namreč še vedno razvija) dodajajo nove. Principi so stari in v računalništvu poznani že mnogo let, vendar so tokrat prvič povezani v celoto in tako dopolnjujejo drug drugega. Metodologija je deležna očitkov, da naj bi osnovnih dvanajst principov poudarjalo v glavnem programersko plat razvoja programske opreme (pri ekstremnem programiranju je najpomembnejša koda!), manj pa druge plati (npr. organizacijsko). Kljub temu ostaja naslednjih dvanajst principov temelj metodologije ekstremnega programiranja [5, 23]:

- načrtovalna igra (angl. *planning game*),
- pogoste in majhne izdaje (angl. *small releases*),
- metafora sistema ali skupno besedišče (angl. *metaphore*),
- enostavno načrtovanje (angl. *simple design*),
- preoblikovanje (angl. *refactoring*),
- programiranje v parih (angl. *pair programming)*,
- skupno lastništvo programske kode (angl. *collective ownership*),
- standardi kodiranja (angl. *coding standards*),
- stalno integriranje (angl. *continuous integration*),
- ustaljeni delovni ritem (angl. *sustainable pace*),
- stalna prisotnost stranke (angl. *customer on-site*) in
- testno usmerjeni razvoj (angl. *test-driven developement*).

Načrtovalna igra

Načrtovanje potrebnih aktivnosti in postavljanje rokov je zaradi narave informacijskih projektov težka naloga. Raje kot skušati na začetku izdelati popolno sliko poslovnega sistema (ki je ne glede na porabljen čas vedno le grob približek stalno spreminjajočega se poslovnega ali drugega sistema) ekstremno programiranje skuša zgraditi to sliko postopoma med potekom projekta. Cilj je v najkrajšem možnem času podpreti tiste naročnikove zahteve, ki imajo zanj največjo uporabno vrednost. Programska oprema se tako izboljšuje in nadgrajuje postopoma, skozi iteracije. Poleg tega je stranski produkt načrtovalne igre pretvorba tihega znanja (angl. *tacit knowledge*) v eksplicitno znanje s pomočjo tehnik, kot so uporabniške zgodbe oziroma kartice z zgodbami (angl. *story cards*).

Načrtovalno igro sestavljata dva koraka:

a) *načrtovanje izdaje***,** v kateri stranka določi, katere uporabniške zgodbe bodo realizirane v naslednji izdaji, in

b) *načrtovanje iteracije***,** v kateri programerji razbijejo uporabniške zgodbe na aktivnosti oziroma določijo bolj podrobno način implementacije zgodb.

Pogoste in majhne izdaje

Razvojna skupina naj pogosto izdaja različice programske opreme in postopoma dodaja funkcionalnost programski opremi. Dve različici naj se med seboj razlikujeta le v implementaciji manjšega števila dodatnih zahtev. S tem bo razvojna skupina pridobivala dragoceno povratno informacijo in bo lahko sprotno prilagajala razvoj. Ekstremno programiranje predlaga razbitje razvojnega procesa na iteracije, dolge od

enega do treh tednov. Rezultat vsake iteracije je delujoča programska oprema. Ker je cikel iteracije zelo kratek, je jasno, da bo lahko imela vsaka naslednja različica v primerjavi s predhodno le malo dodane funkcionalnosti.

Metafora sistema ali skupno besedišče

Metafora predstavlja skupni pogled razvijalcev in stranke o delovanju bodočega programskega sistema. Opis delovanja mora biti takšen, da ga lahko razume tudi stranka. Metafora sistema zajema opis celotnega sistema, uporabniške zgodbe pa opisujejo posamezne funkcionalne dele in značilnosti sistema. Včasih je takšno metaforo težko najti, zato nekateri predlagajo še uporabo skupnega besedišča, s katerim si pomagamo, da ne pride do nesporazumov. Oblikovanje skupnega besedišča je že sicer priporočljivo ne glede na razvojni pristop, ker je to osnova, da lahko izvajalec in naročnik govorita skupni jezik.

Enostavno načrtovanje

Če je preprosto dovolj dobro, potem je najbolje narediti sistem, ki ne bo bolj zapleten in bolj splošen, kot je nujno potrebno za podporo funkcionalnih zahtev. Drugače rečeno, gre za pristop KISS (angl. *Keep it simple and stupid*) oziroma YAGNI (angl. *You aren't going to need it*). Večina programerjev tako ali drugače preveč posplošuje pri programiranju, da bi bila aplikacija bolj prilagodljiva. To je nekakšna naložba v prihodnost, ki pa se večinoma izkaže za drago (ker porabimo dragoceni čas) in nekoristno (ker večino razširitev ne bomo nikoli uporabili [23]). S tem je za funkcionalnost v aplikaciji, ki bo morda nekoč prišla prav, porabljenega znatno več časa pri že tako časovno omejenih projektih. Najbrž bi se lahko programerji do določene mere zgledovali pri konstruktorjih oprijemljivih izdelkov, ki vedno težijo k uporabi najcenejših, vendar še vedno primerno kakovostnih materialov (npr. težnja k uporabi najpočasnejših še dovolj hitrih čipov).

Pravilnost takšnega pristopa pri samem programiranju se zdi neizpodbitna, manj pa je očitna pri samem načrtovanju. Tu gre za popolnoma nasproten pristop kot pri težkih metodologijah, kjer skušamo že pri načrtovanju čimbolje zajeti sliko celotnega sistema. Pri ekstremnem programiranju slika sistema nastaja in se postopoma jasni sproti, med samim programiranjem. Kritiki pravijo, da je tak pristop podoben zlaganju kock, za katere ne vemo, ali bodo na koncu sploh stale skupaj [24]. Uporaba takšnega pristopa je možna samo s stalnim preoblikovanjem in integriranjem ter uporabo orodij, ki spremembe čim bolj lajšajo.

Preoblikovanje

Preoblikovanje je proces spreminjanja programskega sistema na takšen način, da se spremembe programske kode ne odražajo navzven, ampak z njimi izboljšamo notranjo zgradbo kode. Cilj je odpraviti podvojeno programsko kodo, izboljšati

kakovost kode in tako doseči lažje vzdrževanje. Preoblikovanje mora biti podprto z avtomatskim testiranjem modulov, s čimer lahko vsaj delno zagotovimo, da spremembe v strukturi kode ne vplivajo na samo funkcionalnost kode.

Kritiki ekstremnega programiranja opozarjajo [24], da lahko prepogosto preoblikovanje kode vodi k nepotrebnemu “poliranju in šminkanju” kode ter zato predlagajo, da se preoblikovanje izvaja občasno, tako da se ne izgublja preveč časa z že delujočo programsko kodo.

Programiranje v parih

Praksa programiranja v parih je precej poudarjena v ekstremnem programiranju in je najbrž tudi najbolj revolucionarna za programerje, ki delajo na klasičen način. Gre za to, da vso programsko kodo delajo programerji v parih. Programerja v paru uporabljata en računalnik in vedno skupaj delata bodisi na nekem diagramu, algoritmu, programski kodi ali testu. Medtem ko en programer piše kodo, drugi podaja komentarje in skuša sprotno odkrivati napake. Vsi programerji naj bi imeli še svoj osebni računalnik za ostale dejavnosti (npr. branje e-pošte).

Prednosti takšnega pristopa so naslednje:

- celotno programsko kodo v sistemu pregledata vsaj dva programerja,
- vsaj dva človeka sta seznanjena z vsakim delom programskega sistema,
- večja je disciplina oziroma manjša verjetnost, da bosta oba programerja zanemarjala dogovorjena opravila (npr. pisanje testov) in
- menjava parov samodejno širi znanje v skupini.

Programiranje v parih deluje kot stalno pregledovanje kode (angl. *continuous code inspection*). Pokazala sta se tudi pozitivna učinka: višja kakovost načrtovanja kode in manjša stopnja napak [1]. V nekaterih programerskih hišah uporabljajo idejo o delu v parih samo pri določenih opravilih. Tako pri Microsoftu testirajo v parih, pri čemer par sestavljata razvijalec in tester [14].

Skupno lastništvo programske kode

Vsa programska koda se nahaja v repozitoriju. V vsakem trenutku lahko en par integrira, testira in vnaša spremembe v repozitorij. Temu je lahko namenjena dodatna delovna postaja, če se delovna skupina nahaja na isti lokaciji. Na svoji delovni postaji lahko dvojica vnaša spremembe kadarkoli; ko pa želi te spremembe integrirati v repozitorij, mora počakati, da pride na vrsto.

Standardi kodiranja

Zaradi omenjenega skupnega lastništva kode in sicer je pomembno, da se skupina

pri programiranju drži predpisanih standardov in vzorcev. Pri tem ekstremno programiranje ne predpisuje, kakšni naj bi ti standardi bili, pomembno je, da obstajajo. Dobro je imeti predpisane načine poimenovanja objektov v programski kodi, predpisane načine zamikanja programske kode in predpisano uporabo vzorcev ter skupnih knjižnic, kjer je to mogoče. Cilj je narediti sistem s tako poenoteno programsko kodo, da se ne da ugotoviti, kateri par je posamezen sklop sprogramiral.

Stalno integriranje

Deli programskega sistema naj bodo stalno in popolnoma integrirani. Stalna integracija zmanjšuje možnost konfliktov zaradi pogostih sprememb programske kode (posledica enostavnega načrtovanja in pogostega preoblikovanja). Avtorji XP celo trdijo, da dnevne oziroma tedenske izdaje programske kode niso dovolj pogoste in predlagajo prevajanje kode večkrat dnevno [5]. Za primerjavo: Microsoft uporablja dnevne in tedenske prevode programske kode (enkrat dnevno posamezne dele paketa Office in enkrat tedensko celotni paket) [14]. Stalno integriranje je dinamična metoda, ki deluje kot stalno verificiranje in validiranje ter tako omogoča dovolj zgodnjo zaznavo problemov in njihovo hitrejše reševanje [1].

Integriranje kode večkrat dnevno je ostra zahteva. Pri tem se pojavita dve vprašanji, in sicer ali je to sploh tehnično izvedljivo pri določenih razvojnih orodjih in kako to vpliva na storilnost programerjev. Pri večjih, zapletenih projektih, lahko čas prevajanja programske kode znaša tudi več dni. Ta čas je lahko bistveno krajši, če delamo s prevajalniki, ki prevajajo samo spremenjene dele kode. Pomagamo si lahko tudi z gručo računalnikov, postavljeno v ta namen. Pri nekaterih predvsem starejših razvojnih orodjih pa je tako pogosta integracija nemogoča. Drug problem, ki se pojavlja pri tako pogosti integraciji, je, da je za programiranje določenih logičnih enot programske kode enostavno potrebno več časa (tudi več dni), programerji pa se bodo osredotočili na to, da se jim programska koda prevede brez napak in da uspejo vsi testi [5].

Ustaljeni delovni ritem

Zaposlenci naj ne delajo nadur oziroma naj bodo te čim manj pogoste (štirideseturni delavnik). Programerji naredijo pod pritiskom več napak, pade jim motivacija in navadno se poslabša tudi delovna klima. Posledica vsega tega je programska oprema slabše kakovosti. Zato ustaljeni delovni ritem na dolgi rok prinaša boljše rezultate.

Stalna prisotnost stranke

Pri ekstremnem programiranju je izredno poudarjeno ekipno delo. Pomemben in stalni član ekipe je tudi predstavnik naročnika (ali stranka). Ta pomaga pri definiranju zahtev, postavlja prioritete med razvojem in usmerja projekt. Zaželjeno je, da stranka

dobro pozna problemsko domeno projekta, je sposobna videti slabosti že obstoječega sistema in ima hkrati vizijo o bodočem. V praksi je takšnih ljudi malo. Stranka bodisi ne pozna omejitev informacijske tehnologije bodisi ne pozna možnosti, ki jih slednja nudi. V večjih združbah so zaposlenci pogosto ozko usmerjeni na svoje področje delovanja in ne vidijo sistema kot celote. Radi tudi zagovarjajo obstoječe rešitve, čeprav so zastarele ali neustrezne, ker so jih vajeni. Poleg vsega naj bi stranka sama pisala prevzemne teste, to pa je precej vprašljivo, saj je za to kljub vsem sodobnim pripomočkom še vedno potrebno poznavanje programiranja.

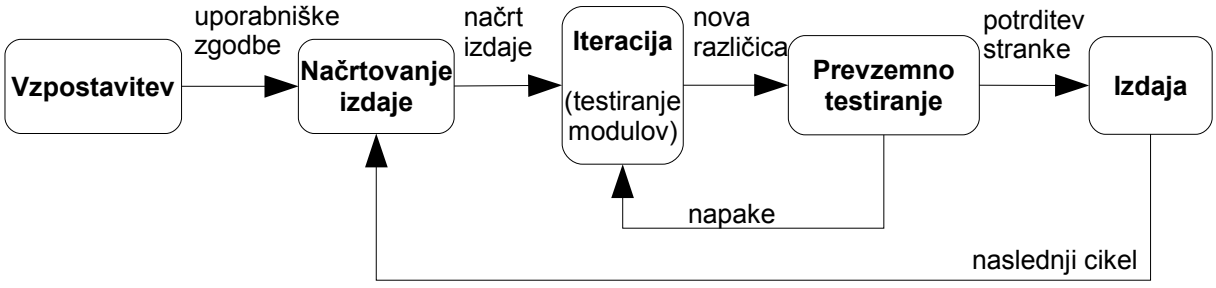
Ne glede na to je ideja o stalni prisotnosti stranke na projektu dobra, vsaj zaradi utrjevanja zaupanja med njo in izvajalci, lažje bo tudi sprejela aplikacijo in jo zagovarjala, če bo to potrebno. Ne gre pa zanemariti, da metodologija XP od ljudi na naročnikovi strani zahteva široko znanje in prilagodljivost, ki ju ni vedno moč dobiti.

Testno usmerjeni razvoj

Ključni del metodologije ekstremnega programiranja (XP) je testiranje. Lahko bi rekli, da celotna metodologija XP stoji ali pade s testiranjem, saj brez te prakse ni možno učinkovito izvajati nekaterih ostalih praks, kot so preoblikovanje ali pa stalno integriranje (slika 5.7). Testno usmerjeni razvoj je dokaj revolucionaren koncept, ki predvideva, da se testi napišejo pred kodo. To pomeni, da je programska koda pisana na kožo testom in ne obratno, kar je sicer običajna praksa. Ker je cilj kodiranja samo pokriti test, naj bi bila tudi koda zapletena samo toliko, kolikor je nujno potrebno.

Proces ekstremnega programiranja

Idealen projekt ekstremnega programiranja naj bi imel kratko vzpostavitveno fazo, z izvedbo za stranko najbolj vredne funkcionalnosti naj bi hitro prinesel uporabne rezultate in jih nato skozi iteracije in majhne ter pogoste izdaje postopno izboljševal [25] (slika 5.6).



Slika 5.6: Proces ekstremnega programiranja

Vzpostavitvena faza ali faza iniciacije naj bi bila relativno kratka in je namenjena začetnemu zajemu uporabniških zahtev. Izhod faze je dokumentacija v obliki uporabniških zgodb in prevzemnih testov.

Uporabniške zgodbe predstavljajo opis delčka obnašanja sistema, kot ga vidijo uporabniki. Pišejo jih uporabniki s svojim izrazoslovjem. Za vsako uporabniško zgodbo programerji ocenijo težavnost implementacije in tveganje. Te ocene so v kasnejši fazi načrtovanja izdaje podlaga za oceno celotnega potrebnega časa za izdajo naslednje različice. Uporabniške zgodbe so tudi podlaga za izdelavo prevzemnih testov. Vsaka uporabniška zgodba mora biti podprta z najmanj enim takšnim testom [25].

Primer enostavne uporabniške zgodbe z oceno tveganja in ceno izvedbe je prikazan v spodnji preglednici:

Uporabniku naj bo omogočeno iskanje osebe po njenem celotnem ali delnem nazivu. Rezultati naj se prikažejo v seznamu.		
Tveganje: Majhno		Cena: 2 točki

Preglednica 3: Primer uporabniške zgodbe

V fazi načrtovanja izdaje stranka določi poslovno vrednost želene funkcionalnosti, zajete z uporabniškimi zgodbami. Poleg tega stranka razvrsti uporabniške zgodbe po pomembnosti in določi, katere zgodbe naj bodo vključene v posamezno izdajo. Prednost pri implementaciji imajo zgodbe z najvišjo prioriteto in največjim tveganjem.

V iteraciji programerji implementirajo izbrane uporabniške zgodbe. Med načrtovanjem iteracije razbijejo uporabniške zgodbe na aktivnosti. Aktivnosti si poljubno razdelijo, vendar je glede na ocenjeno zahtevnost posameznih aktivnosti skupno dodeljenih le toliko aktivnosti, kolikor so jih programerji uspeli dokončati v prejšnji iteraciji.

Rezultat vsake iteracije je delujoča programska oprema, ki v primerjavi s prejšnjo iteracijo, na podlagi realizacije uporabniških zgodb, prinaša novo funkcionalnost. Realizacijo uporabniških zgodb preverja stranka s predhodno napisanimi prevzemnimi testi.

Posamezne izdaje oziroma iteracije imajo nespremenljiv končni datum, toda spremenljiv obseg. Uporabniške zgodbe, ki niso bile implementirane ali ne ustrezajo testom, gredo v naslednjo iteracijo.

V naslednji iteraciji programerji najprej skušajo odpraviti odkrite napake in pomanjkljivosti, stranka pa lahko napiše dodatne uporabniške zgodbe. Na takšen

način se lahko projekt nadaljuje več let ali pa tudi propade, kar se seveda lahko zgodi tudi z uporabo ekstremnega programiranja. Vodstvo projekta mora stalno spremljati potek projekta in ga pogumno prekiniti, če ne prinaša pričakovanih rezultatov.

Ekstremno programiranje in testiranje

Pri linearnih in iterativnih razvojnih modelih se uporabljajo tako statične kot dinamične metode testiranja. Pri agilnih metodologijah se zaradi pogostosti sprememb in števila iteracij v glavnem uporabljajo dinamične metode [1].

Ekstremno programiranje se osredotoča na dve vrsti testiranja:

- testiranje modulov in
- prevzemno testiranje.

Teste modulov ali programerske teste pišejo programerji hkrati s programsko kodo ali pa preden začnejo programirati, če izvajamo testno usmerjeni razvoj. Namen testov je hitro preverjanje delovanja celotnega programskega sistema, po vsakem posegu v sistem. Tega seveda ni možno učinkovito narediti z ročnim, ad-hoc testiranjem, temveč je za to treba narediti avtomatske programe. Po vsaki spremembi programske kode (npr. preoblikovanju neke procedure) mora avtor spremembe pognati vse avtomatske teste in se tako prepričati, da programska koda deluje za primere, ki so bili predvideni.

Kot je že bilo omenjeno pri ekstremnem programiranju stranka izdela prevzemne teste, in sicer po definiranju uporabniških zgodb. Namen teh testov je preveriti, ali sistem deluje tako, kot to zahteva stranka (validacija). Zaželjeno je, da so tudi ti testi avtomatizirani. Prevzemno testiranje se izvaja bolj pogosto in veliko bolj zgodaj kot pri linearnih razvojnih modelih [1].

Tester je pri XP oseba, ki stranki pomaga pisati teste, vzdržuje orodja za testiranje in obvešča udeležence o rezultatih testiranja.

Za razliko od linearnih razvojnih pristopov, ki testiranje obravnavajo ločeno in ga umeščajo za fazo kodiranja, je pri XP testiranje poudarjeno že od samega začetka projekta. Še bolj je v ospredju pri testno usmerjenem razvoju, kjer programerji napišejo testno kodo pred kodiranjem samega modula. Teste je treba napisati za vsako metodo, ki ni trivialna. Prednosti takšnega pristopa k testiranju sta naslednji:

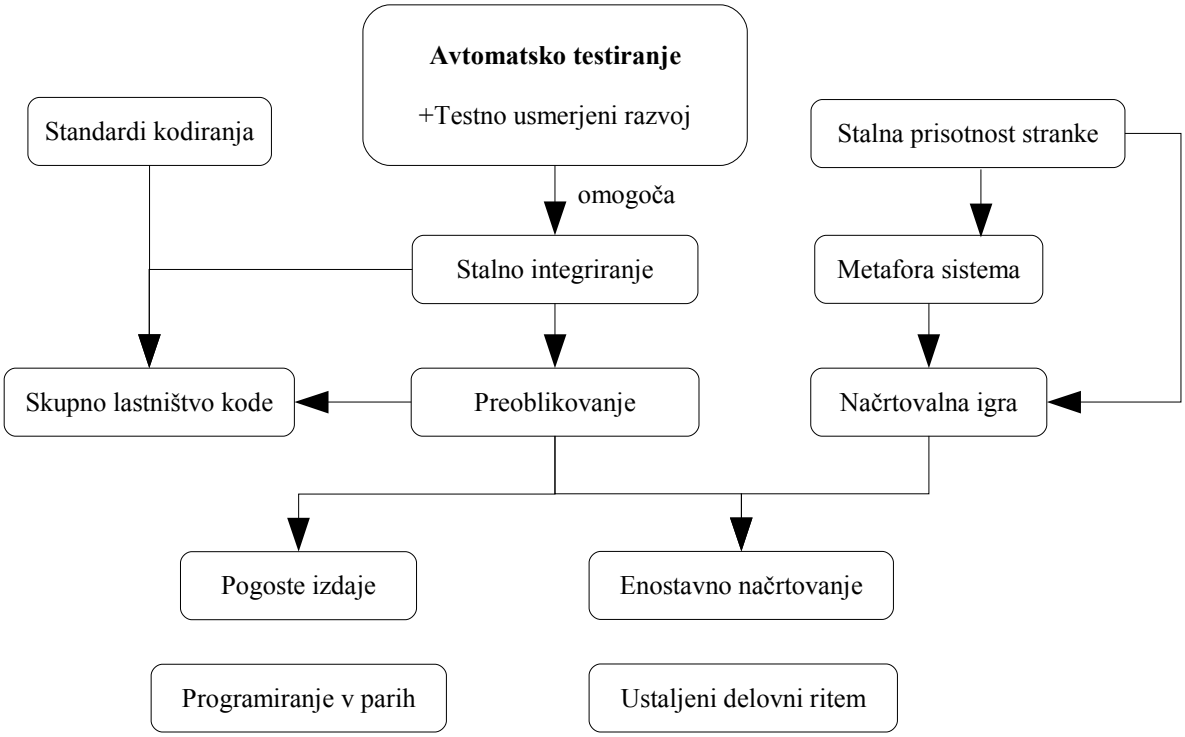
- relativno velika množica testov, kar povečuje zaupanje v programsko kodo in
- najpreprostejša še delujoča programska koda, ki jo je kasneje lažje preoblikovati (v skladu s principom enostavnega načrtovanja) in razumeti.

Avtomatski testi modulov imajo še eno pomembno lastnost – nadomeščajo

dokumentacijo. Pri ekstremnem programiranju in ostalih “lahkih“ metodologijah je dokumentacija potisnjena v ozadje oziroma je minimalna. Testi modulov jo delno lahko nadomeščajo, ker iz njih programer relativno enostavno razbere, kako naj bi modul deloval. Takšna dokumentacija ima še eno lepo lastnost: je vedno ažurna, ker je treba teste sproti vzdrževati.

Sinergija principov in pomen avtomatskega testiranja

Principi ekstremnega programiranja delujejo skupaj in podpirajo drug drugega (slika 5.7). Uporabimo lahko tudi samo nekatere principe, vendar avtorji menijo, da je korist njihove skupne uporabe večja od seštevka koristi vsakega posameznega. Vsekakor je avtomatsko testiranje modulov praksa, brez katere ni ekstremnega programiranja. Ob enostavnem načrtovanju nimamo jasne slike bodočega sistema, ampak se ta gradi sproti, s pogostimi izdajami in neprestanim preoblikovanjem ter integriranjem programske kode. Takšen sistem drži skupaj (kot nekakšno lepilo) množica avtomatskih testov modulov, ki preprečuje, da bi sistem razpadel. Vsak programer mora po spreminjanju programske kode zagotoviti, da se bodo vsi že napisani avtomatski testi uspešno izvedli. Testi so pri programiranju tudi dobro jamstvo za enostavno načrtovanje, ker programerja vodijo k pisanju takšne programske kode, ki naj samo uspešno „preživi“ test in ničesar več. To velja še posebej ob uporabi testno usmerjenega razvoja. Tudi skupno lastništvo kode je brez avtomatskih testov in stalnega integriranja težje izvedljivo. Programer se bo popravljanja tuje kode bolj pogumno lotil, če bo vedel, da ima vedno na voljo avtomatske teste, s katerimi bo lahko preveril, ali ni porušil predvidenega delovanja. Zaradi tega so avtomatski testi najpomembnejša praksa ekstremnega programiranja. Skupaj z ostalimi navedenimi principi omogočajo naglo prilagajanje spremembam (angl. *embracing change*), zato lahko trdimo, da so ključ do agilnosti.



Slika 5.7: Medsebojna odvisnost principov XP

5.4. Agilnost ali disciplina?

Tako kot vse druge dosedanje razvojne metodologije, niti agilne ne predstavljajo dokončne rešitve razvoja programske opreme. Ne obstaja (in najbrž tudi nikoli ne bo obstajala) metodologija ali pristop, ki bi bila primerna za vse vrste projektov. Tudi agilne metodologije imajo svoje omejitve in slabosti. Primerne so za dinamične projekte, kjer se zahteve pogosto spreminjajo oziroma so cilji naročnika ali proračun projekta nejasni oziroma spremenljivi. Metodologijo ekstremnega programiranja njeni avtorji priporočajo za manjše skupine razvijalcev (do deset ljudi), ki morajo biti tudi fizično na isti lokaciji. Priporočljivo je, da je tudi stranka na isti lokaciji. Metodologija je zelo zahtevna za naročnika (stranke), saj od njega med drugim pričakuje, da bo sam izdeloval prevzemne teste. Poleg tega ekstremno programiranje zahteva primerna razvojna orodja. Ta morajo dobro podpirati avtomatsko testiranje, pogosto integracijo, preoblikovanje programske kode, ne pozabimo pa tudi na dobro podporo preoblikovanju strukture podatkovne baze, če je del aplikacije. Programiranje v parih pa tudi ostali principi so socialno zelo intenzivni, zato se od programerjev pričakuje, da so vsi komunikativni in sodelujoči. Vemo pa, da so ljudje različni, nekateri introvertirani, drugi ekstravertirani. Zatorej način dela, kot ga predpisuje ekstremno programiranje, ni primeren za vsakogar.

V preglednici 4 podajam primerjavo med agilnimi in planskimi metodologijami glede na različne kriterije. V zvezi z aktivnostmi za zagotavljanje kakovosti lahko povzamemo naslednje bistvene razlike med agilnimi in planskimi metodologijami (predvsem slapovnim modelom) [1]:

- večina aktivnosti se pri agilnih metodologijah prične izvajati veliko bolj zgodaj,
- pogostost izvajanja teh aktivnosti je pri agilnih metodologijah veliko večja in
- agilne metodologije uporabljajo manj statičnih metod zagotavljanja kakovosti.

Oba svetova metodologij imata svoj "domači teren" na katerem se obneseta najbolje. Zato se v zadnjem času veliko dela na iskanju ravnotežja med disciplino planskih metodologij in lahkostjo agilnih metodologij [26]. Zbliževanje poteka z obeh strani. Tako se o agilnosti govori v zvezi z metodologijo RUP, ki je sicer v osnovi "težka" metodologija. Nastajajo namreč prilagojene lažje ali agilnejše različice RUP. Na drugi strani se z nekaterimi dodatki v smeri večje discipline razvijajo tudi nekatere agilne metodologije (npr. Code Science, AgilePlus itd. [26]). Vsekakor je agilnost, če jo razumemo kot sposobnost hitrega prilagajanja spremembam, v današnjem poslovnem svetu zaželena. Ostaja samo vprašanje, kako jo doseči. Eden od odgovorov na to je avtomatizacija testiranja.

	Agilne metodologije	**Planske metodologije**
Pristop k projektu	Prilagojevalni	Predvidevalni
Merjenje uspešnosti	Poslovna vrednost	Izpolnitev plana
Načrtovanje	Minimalno	Obsežno
Dokumentiranje	Minimalno	Obsežno
Obseg projekta	Majhen	Velik
Razvojna skupina	Majhna, lokalna	Večja, lahko razpršena
Domena	Nepredvidljiva/raziskovalna	Predvidljiva
Število iteracij	Neomejeno	Omejeno
Znanje	Implicitno tiho	Eksplicitno zapisano
Testiranje	Avtomatski testi brez dokumentacije	Dokumentiran testni plan in testne procedure

Preglednica 4: Razlike med agilnimi in neagilnimi pristopi

6. Avtomatizacija testiranja

Ločimo dve veji avtomatizacije testiranja, ki ju nameravam podrobno opisati v nadaljevanju: avtomatske teste modulov in avtomatske funkcionalne sistemske teste. Testi modulov ali programerski testi so v zadnjem času skoraj vedno avtomatizirani in po mnenju nekaterih je avtomatizacija testiranja modulov (v nadaljevanju ATM) sploh najpomembnejši preboj, ki se je zgodil na področju testiranja v zadnjem desetletju [3].

Druga veja avtomatizacije testiranja je avtomatizacija funkcionalnih testov, ki navadno temelji na oponašanju človekove interakcije s testirano aplikacijo. Orodja za tovrstno avtomatizacijo so draga, zato je treba skrbno pretehtati stroške in koristi (poglavje 6.4), izbrati pravo orodje, ki bo skladno z ostalimi razvojnimi orodji in se avtomatizacije lotiti na pravi način. Eden možnih načinov za vpeljavo avtomatizacije testiranja je strukturna metodologija ATLM (Automated Test Lifecycle Methodology), ki jo opisujem v poglavju 6.3.

6.1. Avtomatsko testiranje modulov

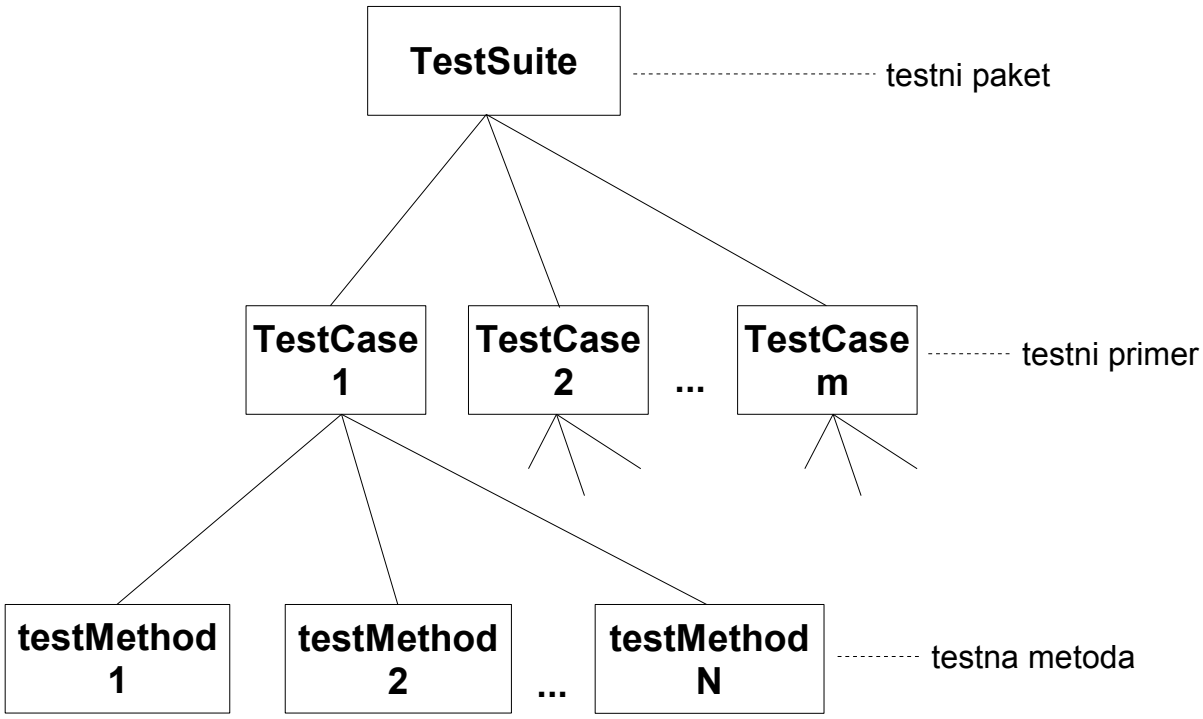
Modul je enota kode, ki zaokroža določeno funkcionalnost. V praksi je to lahko funkcija, procedura oziroma v objektnem programskem jeziku metoda nekega razreda. Princip testiranja modulov je poznan že vrsto let, vendar je posebej pridobil na pomenu z razvojem tehnik in orodij za avtomatizacijo in s prodorom ekstremnega programiranja. To ima poleg številnih zagovornikov tudi ostre kritike (npr. [27]). Toda tudi ti vidijo v avtomatizaciji testiranja modulov pomemben doprinos k razvoju testiranja. Predvsem zato, ker se s tem testiranje modulov sploh izvaja. O njem se namreč govori že vrsto let, vendar se je v praksi pred pohodom avtomatizacije redko izvajalo [27]. Izjema so morda bili kritični sistemi (npr. v medicini), kjer je zanesljivost delovanja na prvem mestu. Vsak proces, ki se dejansko izvaja, pa je zagotovo koristnejši kot popoln proces, ki ostaja na papirju [27].

Prvo znano ogrodje (angl. *framework*), ki je razširilo uporabo testiranja modulov, je bilo prostokodni JUnit, napisan v Javi, ki sta ga izdelala Kent Beck in Erich Gamma [13]. JUnit je kmalu postal priljubljen med razvijalci in kmalu zatem so podobna ogrodja nastala tudi za večino drugih programskih jezikov. Ogrodja so sicer prilagojena značilnostim posameznega jezika, osnovni principi pa ostajajo povsod podobni. Razvoj ATM je kmalu podprla tudi industrija, ki skuša podporo testiranju čim bolj tesno vključiti v programska orodja, kar dodatno olajšuje in niža stroške izvajanja te aktivnosti.

Avtomatski test modula je enota kode, s katero testiramo točno določeni programski

modul. Testiranje poteka po principu črne in bele skrinjice. Razvijalec prve teste naredi samo na osnovi vmesnika modula (črna skrinjica), kasneje doda po potrebi še dodatne teste na osnovi strukture modula (bela skrinjica). Pri zadnjih je cilj povečati pokritost testiranja.

V JUnit so testni primeri vključeni v hierarhijo razredov, ki je popolnoma ločena od grozda produkcijske kode, s čimer ne prihaja do konfliktov med testnim in produkcijskim okoljem.



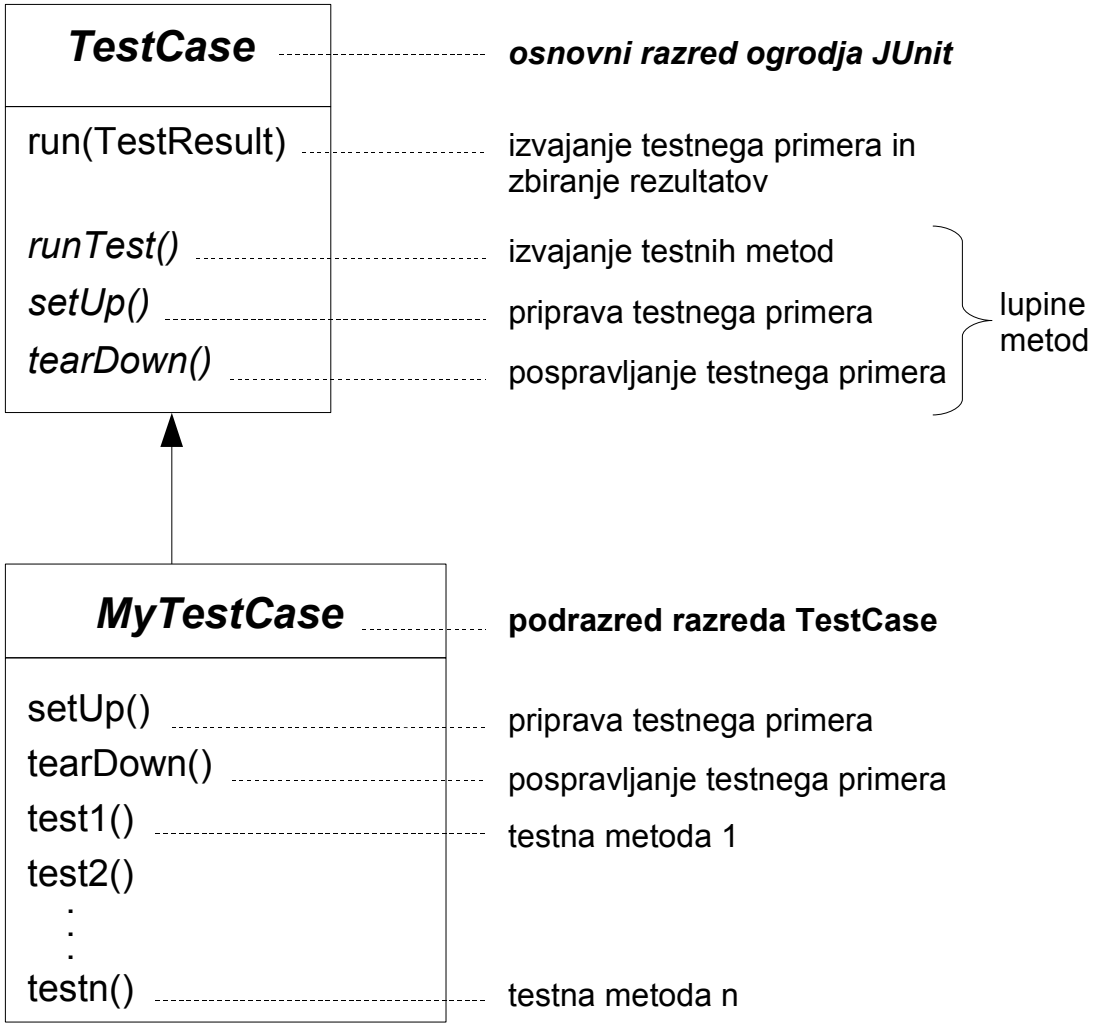
Slika 6.1: Združevanje testnih metod in testnih primerov v pakete

Testni primer v ogrodju JUnit označuje razred TestCase, ki vsebuje eno ali več testnih metod. Testne primere lahko naprej združujemo v zbirke z razredom TestSuite (slika 6.1).

Programer izpelje testni primer vedno kot podrazred razreda TestCase (slika 6.2). Za izvajanje testnih primerov sta možna dva načina: statični in dinamični. Pri statičnem načinu z metodo runTest() navedemo, kateri testi se izvedejo in v kakšnem vrstnem redu. Pri dinamičnem načinu JUnit samodejno izvede vse teste v testnem primeru z uporabo mehanizma zrcaljenja (angl. *reflection mechanism*). Mehanizem požene vse metode, katerih ime se začne z besedo „test“ [13].

Testiranje modulov pri JUnit poteka v naslednjih treh fazah:

- pripravljalna faza (angl. *setup*),
- izvajalna faza s preverjanjem rezultatov in
- pospravljalna faza (angl. *teardown*).



Slika 6.2: Testni primer v JUnit

S pripravljalno fazo pripravimo okolje in podatke, ki jih potrebuje testni primer, s pospravljalno fazo pa počistimo za njim. Pripravljalna in pospravljalna faza se lahko izvajata za vsak test posebej, za vsak testni primer posebej (torej za vse teste v testnem primeru) ali pa samo enkrat za celoten testni paket. Vsak test posebej ima torej lahko ponovno pripravljalno fazo, fazo izvajanja kode, ki jo želimo testirati, in

fazo preverjanja rezultata. Pripravljalni in pospravljalni fazi nam omogočata testiranje v izolaciji, kar pomeni, da rezultat nekega testa ne vpliva na rezultate ostalih testov. Fazi sta neobvezni, programer ju pripravi s prepisom lupin metod setup() in tearDown(). Včasih se omenjeni fazi izvajata samo enkrat za celotni paket testov, navadno zaradi večje hitrosti testiranja [13]. V tem primeru izgubimo neodvisnost testov, ker si morajo testi slediti v določenem zaporedju, to pa posledično prispeva k večji zapletenosti testiranja. V določenih okoliščinah je to neizogibno (npr. ob časovno potratni pripravi podatkov v podatkovni bazi).

Preverjanje rezultatov izvajamo s pomočjo trditev (angl. *asserts*). Trditve so predefinirani podprogrami, s katerimi preverjamo in beležimo resničnost postavljenih pogojev. Lahko so enostavne, kot je npr. primerjava dveh števil, lahko gre za preverjanje, ali ima spremenljivka definirano vrednost, lahko so pa tudi bolj zapletene. Nekatera orodja ATM omogočajo tudi primerjavo enakosti dveh vektorjev, tabel, datotek itd.

Primeri trditev v JUnit:

- assertEquals – trditev je uspešna, ko je dobljeni rezultat enak predhodno določeni vrednosti,
- assertNull – trditev je uspešna, če ima argument nedefinirano vrednost,
- assertTRUE – trditev je uspešna, če ima argument Booleanovo vrednost TRUE.
- fail – takoj označi test za neuspešen. S tem navadno označimo programsko kodo, ki je test ne bi smel doseči.

Rezultat testiranja lahko zavzame vrednosti uspešen, neuspešen ali pa napaka. Test je uspešen, ko so vse njegove trditve uspešne. Test se konča neuspešno, ko naleti na prvo neuspešno trditev. Test se konča z napako, ko pride do napake, ki je programer ni predvidel.

Vsak test izvede neko produkcijsko funkcijo, katere izhod primerjamo s pričakovanimi rezultati. Izhod lahko vrne klicana funkcija, lahko pa samo spremeni stanje nekega objekta. V tem primeru mora obstajati funkcija, ki vrača stanje tega objekta. Na ta način pri testno usmerjenem razvoju gradimo vmesnik razreda. Dokaj verjetno je, da bo ista funkcionalnost potrebna tudi v produkcijski kodi. Za preverjanje izhodov lahko uporabimo tudi inverzno operacijo funkcije, ki jo testiramo. Če testirano funkcijo že imamo v delujoči obliki in jo samo preoblikujemo (npr. zaradi nepregledne kode), lahko primerjamo izhode nove in stare izvedbe.

Testi modulov naj bi po izkušnjah vsebovali vsaj toliko kode, kot je vsebujejo testirani produkcijski moduli [13]. Logika v testih mora biti karseda preprosta, zato da ne prihaja do dodatnih nepotrebnih napak. Če je logika testa prezapletena, je treba testirano kodo preoblikovati. Vsak test mora biti berljiv, njegov namen mora biti takoj

razviden tudi tistemu, ki ga ni pisal. Test naj ima en objekt testiranja in ta naj bo prepoznaven že iz imena testa.

Seveda se lahko tudi v testno programsko kodo prikradejo napake. Kljub temu testov ne testiramo posebej, ker bi to pomenilo dodaten strošek, ampak se skušamo čim bolj držati prej navedenih principov. Poleg tega je priporočljivo redno izvajati pregledovanje kode testov (angl. *code inspection*), predvsem, ali ti zadovoljujejo postavljene standarde kodiranja.

Za primer avtomatskega testiranja modulov je v nadaljevanju prikazan praktičen primer testiranja v Javi, in sicer na primeru izvedbe sklada celih števil. Sklad je seznam, za katerega je značilen LIFO dostop do elementov (zadnji noter prvi ven). V spodnjem primeru ima lahko omejeno ali neomejeno količino elementov. Elemente dodajamo na sklad z metodo Dodaj in odvzemamo z metodo Vzemi. Poleg teh imamo še metodi za preverjanje, ali je sklad prazen oziroma napolnjen do roba.

```
public class Sklad  {

      int[] elementi;

      int zgornjiElement = -1;

      public Sklad()    { this(10); }
      public Sklad(int velikost) { elementi = new int[velikost]; }

      public boolean jePrazen() {
         return zgornjiElement == -1;
      }

      public boolean jePoln() {
         return zgornjiElement == elementi.length-1;
      }

      public void dodaj( int element ) {
         zgornjiElement++;
         elementi[zgornjiElement] = element;
      }

      public int vzemi() {
         return elementi[zgornjiElement--];
      }
}
```

Za zgornjo kodo sedaj pripravimo testni primer. Pripravljalne in pospravljalne faze v tem primeru ne potrebujemo, zato ju lahko izpustimo.

```
public class SkladTest extends TestCase {
```

Testni primer je seveda kot vsak drugi testni primer v JUnit podrazred razreda

TestCase. Za začetek lahko naredimo nekaj testov izdelave sklada, enega z omejeno in drugega z neomejeno kapaciteto elementov.

```
public void testNoviSklad() {
   Sklad sklad = new Sklad();
   assertTrue( sklad.jePrazen() );
}

public void testNoviOmejenSklad() {
   Sklad sklad = new Sklad(5);
   assertTrue( sklad.jePrazen() );
}
```

V testu postavimo trditev, da bo sklad po izdelavi prazen. Za preverjanje stanja uporabimo kar produkcijsko metodo jePrazen(), ki nam vrača Booleanovo vrednost. S tem na videz prekršimo načelo neodvisnosti testov, saj testiramo izdelavo sklada, hkrati pa drugo produkcijsko metodo uporabljamo za preverjanje stanja. Če slednja ne bi bila pravilna, bi lahko bil tudi rezultat testa napačen. V resnici je to še najboljša možnost. Metoda, ki smo jo uporabili, je trivialna, saj samo vrača vrednost spremenljivke in ji zato lahko zaupamo. V nasprotnem primeru bi morali izpostaviti notranjo spremenljivko razreda, da bi jo lahko prebrali, česar pa si ne želimo.

Sledi nekaj pozitivnih testov, s katerimi preverjamo delovanje kode v običajnih okoliščinah. V tem primeru testiramo običajne lastnosti sklada, kot so dodajanje in odvzemanje elementa, ter glavno značilnost sklada – LIFO zaporedje.

```
public void testDodajOdvzemi(){
   Sklad sklad = new Sklad();
   sklad.dodaj(15);
   assertFalse(sklad.jePrazen());
   assertEquals(15, sklad.vzemi());
   assertTrue(sklad.jePrazen());
}

public void testSkladZaporedje(){
   Sklad sklad = new Sklad();
   sklad.dodaj(15);
   sklad.dodaj(25);
   sklad.dodaj(35);
   assertEquals(35, sklad.vzemi());
   assertEquals(25, sklad.vzemi());
   assertEquals(15, sklad.vzemi());
}
```

Nikoli ne smemo pozabiti še na negativne, robne primere. Kaj se zgodi, če skušamo vzeti element pri praznem skladu ali pa dodati element na prepoln sklad? Kaj se zgodi, če dodamo črko namesto številke itd. Pravilno obnašanje v teh primerih je težko določiti in je odvisno od potreb produkcijskega sistema. Če ocenimo, da

napaka ni pomembna za nadaljnje delovanje, jo lahko samo prestrežemo in utišamo. Sicer jo prestrežemo in izpišemo uporabniku ustrezno sporočilo. Ni torej splošnega recepta, kako naj se programska koda v takšnih primerih obnaša. Če ni na voljo zelo natančnih specifikacij, je odločitev prepuščena programerju. Njegove domneve, četudi napačne, pa so vedno jasno razvidne iz avtomatskih testov. Ti so veliko bolj enostavni in pregledni, kot je testirana programska koda, zato predstavljajo tudi dobro in vedno ažurno (programersko) dokumentacijo.

Iz naslednjih primerov je razvidno, da prepustimo programu dvig sistemske napake, če je sklad prepoln ali je že prazen. To so robni primeri, za katere predvidevamo, da je majhna verjetnost, da bodo nastali v produkcijskem okolju. Če pa se vseeno pojavijo, bomo nase morali prevzeti breme slabega načrtovanja.

```
public void testPrazenSklad() {
   Sklad sklad = new Sklad(512345);
   sklad.dodaj(1);
   sklad.vzemi();
   try {
      sklad.vzemi();
      fail( "Sklad je prazen");
   }
   catch (Exception overflow) {
     }
   }

public void testPolnSklad() {
   Sklad sklad = new Sklad(2);
   sklad.dodaj(10);
   sklad.dodaj(20);
   assertTrue( sklad.jePoln() );
   try { sklad.dodaj(30);
         fail( "Sklad je poln");
   }
   catch (Exception overflow) {
      }
   }
```

6.1.1. Navidezni moduli

Testiranje modulov v idealnem primeru poteka v izolaciji, kar pomeni, da izid testiranja enega modula ne sme vplivati na izid testiranja drugega modula. Testi v izolaciji so medsebojno neodvisni, zato jih lahko izvajamo v poljubnem vrstnem redu. Včasih je to v praksi težko doseči. Med testno kodo lahko obstaja soodvisnost (angl. *coupling*). O soodvisnosti govorimo, ko prvi test predpostavi, da je drugi test postavil nek objekt v določeno stanje.

Za lažje pisanje testov lahko uporabimo navidezne module (angl. *mock objects*). Z

njimi nadomeščamo programsko kodo s ciljem olajšati pisanje testov in pohitriti testiranje. Nadomeščamo kodo, ki je zapletena, nedokončana ali pa zahteva dostop do virov, ki nam v času testiranja niso na voljo. Z njimi zato lažje zagotovimo testiranje v izolaciji.

Navidezne module torej uporabimo v naslednjih primerih:

- kadar se pravi moduli obnašajo nedeterministično,
- kadar module težko pripravimo za testiranje,
- ko so moduli počasni,
- kadar moduli zahtevajo uporabo uporabniškega vmesnika,
- takrat ko moduli še ne obstajajo,
- moduli zahtevajo dostop do zunanjih virov in aplikacij.

Navidezne module lahko izdelamo s pomočjo vmesnikov razreda, če seveda uporabljamo objektno usmerjeni programski jezik. Za razred, katerega objekte želimo nadomestiti, definiramo vmesnik. Nato na podlagi vmesnika naredimo dva razreda. En je produkcijski razred, drugi razred je navidezni in namenjen samo testiranju.

Spodaj je primer uporabe navideznega modula. Z razredom dokument številčimo dokumente po naslednjem ključu: “Leto/Zaporedna številka“. Zaporedna številka je štirimestna tekstovna z vodilnimi ničlami (npr. 2005/0123). Ob menjavi leta začnemo šteti znova (npr. “2006/0001“). Dokumenti so shranjeni v podatkovni bazi.

Potrebujemo metodo, ki nam bo vrnila novo številko dokumenta. To bo zadnja zaporedna številka iz podatkovne baze za trenutno leto. Če se je leto menjalo ali pa gre za prvo številko sploh, bomo pričeli štetje z ena, sicer bomo ustrezno povečali zaporedno številko.

```
public String novaStevilka() {
      String novaStevilka;
      int zadnjeLeto;

      String zadnjaStevilka = dokDB.vrniZadnjoStevilko();
      int letos = okolje.vrniTekoceLeto();

      if (zadnjaStevilka!="") {
            zadnjeLeto = vrniZadnjeLeto(zadnjaStevilka);
            if (zadnjeLeto==letos) {
                  //povečaj številko
                  novaStevilka=letos + "/"
                  + naslednjaZaporednaSt(zadnjaStevilka);
            }
            else { //novo leto
                  novaStevilka = letos+"/"+"0001";
            }
```

```
            }
            else { //prva številka
                  novaStevilka = letos+"/"+"0001";
            }

      return novaStevilka;
}
```

Metoda potrebuje dvoje ključnih podatkov: zadnjo številko dokumenta in tekoče leto. Zadnjo številko dobimo s poizvedbo v podatkovni bazi (metoda vrniZadnjoStevilko()), tekoče leto pa iz sistemskega (trenutnega) datuma (metoda vrniTekoceLeto()). Oboje je nerodno za pripravo avtomatskih testov. Če želimo testirati menjavo leta, moramo imeti možnost spreminjanja sistemskega datuma ali pa podati metodi vhodni datumski parameter, za kar pa v produkcijskem okolju ni nobene potrebe.

Problem je tudi metoda, ki nam vrne zadnjo številko dokumenta. Za dostop do podatkovne baze mora biti izpolnjenih kar nekaj pogojev, baza mora biti na voljo, v delujočem stanju, imeti moramo pravico do dostopa, pa še čas izvedbe testa bi se bistveno povečal. Zaradi tega izdelamo nadomestne razrede, s katerimi simuliramo delovanje razredov, ki vsebujeta omenjeni dve metodi. To storimo s pomočjo vmesnikov.

```
public interface IOkolje {

      public int vrniTekoceLeto();
      // ostale metode...

}

public interface IDokumentDB {

      public String vrniZadnjoStevilko();
      // ostale metode...
}
```

Naredili smo torej dva vmesnika za razreda, ki nam povzročata težave oziroma ju želimo nadomestiti. Zatem izpeljemo dve izvedbi vmesnikov, eno produkcijsko in drugo za testiranje:

```
public class DokumentDB implements IDokumentDB {

public String vrniZadnjoStevilko(){
            /*
             * Koda ki poišče v podatkovni bazi zadnjo številko
             * dokumenta
             */
            return zadnjaStevilka;
      }
```

```
}
```

Denimo, da produkcijskega razreda še nimamo dokončanega. Sledi navidezni razred, ki je bistveno enostavnejši za izvedbo.

```
public class MockDokumentDB implements IDokumentDB{

      public String vrniZadnjoStevilko(){
            return zadnjaSt;
      }

      public void nastaviZadnjoStevilko(String pZadnjaSt) {
            zadnjaSt = pZadnjaSt;
      }

    private String zadnjaSt;

}
```

Podobno storimo še za tekoče leto:

```
public class SistemskoOkolje implements IOkolje {

      public int vrniTekoceLeto() {
          Calendar cal = Calendar.getInstance();
            return cal.get(Calendar.YEAR);
      }
    //ostale metode...
}


public class MockOkolje implements IOkolje {

      public int vrniTekoceLeto(){
            return tekoceLeto;
      }

      public void nastaviTekoceLeto(int pLeto){
            tekoceLeto = pLeto;
      }

    private int tekoceLeto;

}
```

Princip je torej vedno podoben. Ne glede na zapletenost izvedbe v produkcijskem modulu lahko naredimo nadomestno izvedbo namenjeno testiranju, ki je za razliko od produkcijske kode namerno čim bolj preprosta. V glavnem vsebuje metode za nastavljanje in vračanje vrednosti. Tako je navidezni razred hitro narejen, možnost dodatnih napak, s katerimi bi dodatno zapletli testiranje, pa bistveno manjša.

Kot primer sledi še test številčenja za primer novega leta.

```
public void testNovoLeto() {
      mockOkolje = new MockOkolje();
      mockOkolje.nastaviTekoceLeto(2005);

      mockDokDB = new MockDokumentDB();
      mockDokDB.nastaviZadnjoStevilko("2004/1234");

      Dokument dok = new Dokument(mockOkolje, mockDokDB);
      String novaSt = dok.novaStevilka();

      assertEquals("2005/0001", novaSt);
}
```

V testu pripravimo nadomestna modula z želenimi vrednostmi in ju podtaknemo razredu Dokument kot kukavičje jajce. V produkcijski kodi bi seveda uporabili prava modula in z njima poklicali razred Dokument. Na tak način izpeljemo še vse ostale potrebne teste.

6.1.2. Testiranje podatkovno intenzivnih aplikacij

Poseben izziv pri vpeljavi ATM predstavljajo podatkovno intenzivne aplikacije. To so aplikacije, katerih delovanje močno sloni na interakciji s podatkovno bazo. Takšne aplikacije predstavljajo zaradi več razlogov poseben problem pri pisanju avtomatskih testov. Dejstvo je, da je večina današnjih najbolj razširjenih baz podatkov relacijskih in da se te ne skladajo najbolje z objektno usmerjenim načinom programiranja. Obstajajo sicer čisto objektno usmerjene podatkovne baze, vendar se na trgu še niso prijele. Zaradi tega ostaja pri takšnih aplikacijah arhitekturni prepad med dvema različnima svetovoma [28]. Na eni strani imamo poslovno logiko narejeno s čisto objektno tehnologijo, na drugi strani pa podatke shranjene v podatkovni bazi z relacijskim modelom iz sedemdesetih let prejšnjega stoletja. Ta prepad skušamo premostiti z vmesniki, ki preslikajo podatke iz objektnega sveta v relacijski. Obstaja kar nekaj takšnih vmesnikov [1], vendar pa se programer lahko še vedno odloči in gre mimo njih iz performančnih ali drugih razlogov. Posledica tega je bistveno manjša testnost programske kode, ne glede na to, da programiramo z najnovejšim razvojnim orodjem.

Naslednje, kar je treba upoštevati pri podatkovnih bazah, je trajnost (angl. *persistence*) podatkovnih objektov in akcij. Če izvedemo test, ki bo nekaj izbrisal iz baze in test ponovimo, potem se brisanja ne bo dalo več ponoviti. Zato moramo zagotoviti vedno enake začetne pogoje. Pri tem si lahko pomagamo s skriptami, ki v

1 Glej http://www.hibernate.org

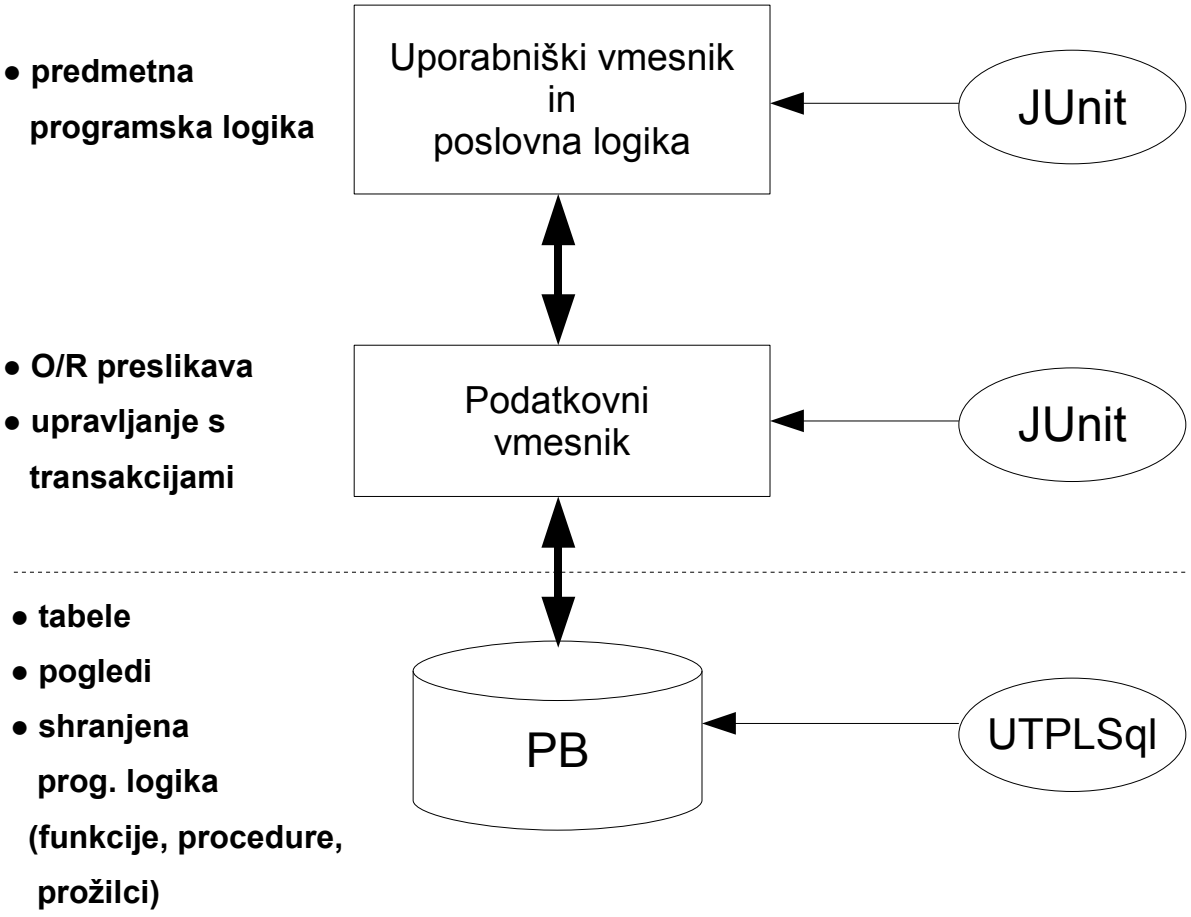
bazi pripravijo vse potrebno v pripravljalni in pospravljalni fazi (npr. izdelamo kopije tabel, ki jih potrebujemo). Problem takšnega pristopa je prevelika časovna zahtevnost. Pri ATM želimo, da je interval med testiranji čim krajši in da testiranje poteka tako, da ne moti programerja. V nasprotnem primeru obstaja velika verjetnost, da se ne bo izvajalo tako pogosto, kot bi bilo potrebno [13]. Naslednja možnost priprave podatkov je enkratna izvedba pripravljalne in pospravljalne faze (npr. za celotni testni paket), pri tem pa se bomo morali odpovedati principu izolacije testov in jih izvajati v točno določenem vrstnem redu. Najbolj elegantna rešitev se zdi možnost povrnitve stanja v bazi v določeno točko v preteklosti, podobno kot da bi prevrteli film na začetek traku. Nekatere podatkovne baze to že omogočajo [1].

Poleg omenjenih dejavnikov je ob testiranju aplikacij s podatkovno bazo treba upoštevati še nekatere njene lastnosti. Težave lahko povzroči uporaba zunanjih ključev, zapleteni prožilci, ki lahko posredno ažurirajo še dodatne tabele poleg ciljne, in tri-vrednostna logika v bazi (“Resnično“, “Neresnično“ in “Neznano“).

Omenjenih težav seveda ne rešujemo tako, da se izogibamo uporabi koristnih lastnosti, ki jih imajo današnje podatkovne baze, ampak s primerno zasnovo arhitekture sistema. Ta predpostavlja plastno zasnovo sistema, s čim bolj medsebojno neodvisnimi plastmi. Na sliki 6.3 je primer takšne aplikacije. Programska koda za uporabniški vmesnik in poslovno logiko je ločena od kode, ki neposredno upravlja s podatki v bazi. Programer se pri delu osredotoča na poslovne dogodke, podatkovni vmesnik pa poskrbi za pretvorbo podatkov iz baze v objekte in upravljanje s transakcijami. Ta lahko nudi osnovne podatkovne operacije (t.i. CRUD [2]), kot tudi bolj zapletene. Velik del programske kode je lahko shranjen v sami podatkovni bazi v obliki programskih procedur ali v prožilcih tabel.

ATM pri takšni arhitekturi lahko poteka na več ravneh. Za testiranje zgornjih plasti (poslovne logike) npr. uporabimo JUnit ali drugo primerno ogrodje in razrede vmesnikov zamenjamo z navideznimi moduli. S tem se izognemo podatkovni bazi, poenostavimo in pohitrimo testiranje.

1 Npr. tehnologija Oracle Flashback
2 Angl. akronim za Create, Read, Update, Delete



Slika 6.3: Primer uporabe ATM za podatkovno intenzivne aplikacije

Podatkovni vmesnik se navadno spreminja manj pogosto, vendar zanj potrebujemo dostop do podatkovne baze, zato bodo ti testi počasnejši. Za programsko kodo v bazi uporabimo druga primerna testna orodja, če seveda obstajajo. Za jezik PL/SQL je takšno orodje utPL/SQL. Nastal je po zgledu JUnit, seveda pa je prilagojen značilnostim jezika PL/SQL in delu s podatkovno bazo. Podobno kot jezik PL/SQL združuje procedure in funkcije v pakete so tudi testne procedure utPL/SQL združene v testne pakete. Testni paket lahko vsebuje pripravljalno kodo v prodeduri ut_setup in pospravljalno v proceduri ut_teardown. Produkcijska procedura, ki jo testiramo, vsebuje vsaj eno testno proceduro ut_*imeprocedure*. Testne pakete lahko izdelamo s pripomočkom utGen, tako da mu kot parameter podamo matriko vhodnih parametrov v proceduro in pričakovani rezultat. Rezultate testov podobno kot pri JUnit preverjamo s trditvami. Možno je primerjati enostavne vrednosti kot npr. dve števili, pa tudi vrednosti dveh tabel, vektorjev, datotek, poizvedb itd. Prav tako je možno preverjati, ali ima spremenljivka vrednost nedefinirano. Sledi primer testa v utPL/SQL:

```
PROCEDURE ut_sprememba_priimka
IS
BEGIN
   /* ažuriraj kopijo tabele */
   EXECUTE IMMEDIATE '
   UPDATE ut_oseba_sprememba SET
      PRIIMEK = ''TEST'',
    WHERE
      OSEBA_ID = 7600
   ';
   /* ažuriraj tabelo s produkcijsko proceduro */
   te_oseba.upd (
      7600,
      priimek_in => 'TEST',
   );
   /* Testni rezultati (primerjaj s poizvedbo v obeh tabelah) */
   utassert.eqquery (
      'Update three columns',
      'select priimek from oseba',
      'select priimek from ut_oseba_sprememba'
   );
   ROLLBACK;
END;
```

V zgornjem primeru testiramo proceduro za ažuriranje priimka v tabeli oseba. Uporabimo testno tabelo, ki jo predhodno pripravimo v proceduri ut_setup (ni prikazana). Testno tabelo ažuriramo ročno, produkcijsko pa s testirano proceduro. Na koncu mora biti vsebina obeh tabel identična, kar potrdimo s trditvijo utassert.eqquery. Ta primerja rezultat dveh poizvedb.

Prikazan je bil način avtomatskega testiranja modulov za aplikacije, ki komunicirajo s podatkovno bazo. Ta predstavlja posebno težavo pri uporabi ATM, ki pa jo lahko omilimo z dosledno uporabo podatkovnega vmesnika in primernih orodij.

6.1.3. Testnost kode in testno usmerjeni razvoj

Uspešnost in cena izvajanja ATM je neposredno odvisna od testnosti kode (angl. *testability*). S testnostjo je mišljena prilagojenost programske kode za pisanje avtomatskih testov. Boljša testnost pomeni lažje in cenejše izvajanje ATM, slaba testnost pa lahko pomeni, da ATM sploh ne moremo izvajati.

Za doseganje dobre testnosti morajo moduli biti majhni in funkcionalno zaokroženi (kohezivni), s čim bolj preprosto notranjo logiko. Soodvisnost med moduli naj bo čim manjša, tako bo tudi vpliv napačno delujočih modulov manjši. Dobra strategija je tudi že omenjeno razbitje sistema na več plasti [4]. Na tak način logično ločimo programsko kodo v smiselno zaokrožene celote, ki so med seboj šibko povezane prek vmesnikov. Tako ločimo npr. kodo uporabniškega vmesnika od kode za izvajanje poslovnih pravil in kode za dostop do podatkov itd.

Določen vpliv na testnost ima vsekakor tudi objektna usmerjenost programskih jezikov, ki programerja sili k večji granulaciji pri programiranju. Vendar uporaba objektnih programskih jezikov sama po sebi še ni rešitev. Tudi strukturni programski jezik omogoča izdelavo kode, ki jo je lahko testirati, po drugi strani pa lahko z najnovejšim orodjem in popolnoma objektno usmerjenim jezikom naredimo slabo kodo.

Na testnost zelo pozitivno vpliva testno usmerjeni razvoj. Pri njem mora programer že ob načrtovanju razreda imeti v mislih, kako bo z njim čim lažje izpeljal predvidene teste. Na tak način tudi lažje gradi vmesnik razreda, saj je velika verjetnost, da bodo metode, ki jih je treba narediti zaradi testiranja, uporabne tudi v produkciji. Po drugi strani po principu enostavnega načrtovanja ne dodaja nobenih drugih metod "na zalogo". S tem se poveča tudi možnost ponovne uporabnosti razreda (angl. *reusability*) [29]. Pri ponovni uporabnosti modula gre za večno dilemo, do kakšne mere posplošiti delovanje modula. Če je presplošen, ga lahko po nepotrebnem zapletamo, če je njegova funkcionalnost preveč omejena, pa je premalo uporaben. Kot praktičen primer vpliva ATM na načrtovanje modulov si oglejmo naslednji primer:

Za potrebe izdelave testov moramo izpostaviti nekatere vmesne vrednosti oziroma spremenljivke v razredu. Na prvi pogled je seveda najlažje spremenljivko izpostaviti tako, da je drugim razredom vidna brez omejitev. To je slaba rešitev, ker s tem razbijamo ograjevanje (angl. *encapsulation*) razreda. Boljša rešitev je izdelava funkcije, ki nam vrača vrednost spremenljivke.

Raziskave kažejo, da pri testno usmerjenem razvoju ni porabljenega bistveno več časa kot pri običajnem razvoju [30], ugotovljen pa je bil pozitiven vpliv na ponovno uporabnost in razumljivost kode [29]. Testno usmerjeni razvoj ali pa vsaj sprotna izdelava testov, hkrati z ostalo programsko kodo, ima posreden vpliv na načrtovanje modulov in s tem posledično visoko testnost programske kode. Zato mora biti testom prijazno načrtovanje kode pomemben cilj. Ostane vprašanje, kako je z naknadno izdelavo avtomatskih testov po izdelavi ostale programske kode, o čemer več v naslednjem podpoglavju.

6.1.4. Avtomatizacija testiranja modulov za obstoječo programsko kodo

Izdelava avtomatskih testov ni posebej zahtevna s testno usmerjenim pristopom in uporabo primernih razvojnih orodji. Razvijalci porabijo sicer nekaj več časa, kot če testov ne bi pisali [30, 31], vendar so koristi bistveno večje. Čas, porabljen za pisanje testov, se zmanjša tudi z večanjem izkušenj razvijalcev ob pisanju testov. Pojavi pa se vprašanje, ali lahko avtomatiziramo testiranje modulov za nazaj, torej za obstoječo programsko kodo, pri kateri ni bil uporabljen testno usmerjeni pristop ob razvoju. Ta problem je dokaj aktualen, saj je avtomatizacija testiranja modulov v razmahu šele zadnjih nekaj let, v uporabi pa je še vedno veliko starih sistemov, kar posledično zahteva vzrževanje stare programske kode. To je bilo moč videti ob

prehodu v leto 2000, ko je bilo treba prenavljati programske sisteme, za katere najbrž niti avtorji sami niso domnevali, da bodo toliko časa v uporabi. Ob prenovi programskih sistemov je treba večkrat vključiti stare podsisteme, razen če se seveda ne odločimo za big-bang pristop in vse zamenjamo v enem koraku.

Poleg tega je dejstvo, da večino časa življenjskega cikla programske opreme zavzema faza vzdrževanja. Vzdrževanje programske opreme je bilo do nedavnega najbolj zanemarjena aktivnost v življenjskem ciklu programske opreme [8]. Teorija razvoja programske opreme je navadno predvidevala, da začenjamo graditi sistem od začetka, kar pa v praksi redkokdaj drži [17]. Veliko programske opreme je tehnično zastarele, vendar jo je treba kljub temu prilagajati novim zahtevam, zakodonodaji itd. V praksi imamo torej velikokrat opravka s staro programsko kodo, narejeno s starimi programskimi orodji in brez namena programerjev, da bi bila kadarkoli avtomatsko testirana. Posledično je testnost takšne programske kode praktično nična. Ali se v tem primeru sploh splača razmišljati o avtomatizaciji testiranja modulov?

V fazi vzdrževanja, ko je sistem v produkciji, je treba obstoječo programsko kodo spreminjati bodisi zaradi odkritih napak bodisi zaradi dodajanja nove funkcionalnosti. Pri tem seveda ne sme priti do prevelikih motenj delovnega procesa. Za to bi prav prišla množica avtomatskih testov, ki bi omogočala lažje in varnejše spreminjanje kode. Raziskave kažejo [30, 31], da je avtomatske teste zelo težko pisati za nazaj, če se programerji temu niso posvečali ob pisanju kode. Da bi teste izdelali, moramo bistveno povečati testnost kode. Za povečanje testnosti kode se moramo lotiti predelave, brez testov pa nam je običajno v oporo le zastarela dokumentacija in v najboljšem primeru obilo izkušenj in znanja programerjev. Problem spominja na zgodbo o tem, kaj je bilo prej – kokoš ali jajce? Zato je testnost tesno povezana z enostavnostjo vzdrževanja (angl. *maintainability*). Visoka testnost sistema pomeni tudi lažje vzdrževanje sistema.

Popolna rekonstrukcija programske kode vzdrževanega sistema je v večini primerov predraga in pretvegana. Če je sistem dolgo v produkciji, vsebuje navadno stabilna jedra, v katera je bilo vloženega veliko časa in izkušenj. Za takšna jedra ni smiselno pisati avtomatskih testov, ker bi bile koristi bistveno manjše od stroškov njihove izdelave in tveganja, da bi delujočo kodo med rekonstrukcijo pokvarili. Za dele sistema, ki večkrat povzročajo težave in jih moramo stalno popravljati, pa je rekonstrukcija kode smiselna in takrat tudi izdelamo avtomatske teste za predelano kodo, če je to mogoče. Lahko se namreč izkaže, da je izdelava avtomatskih testov pretežavna ali pa nimamo na voljo ustreznih orodij za podporo ATM.

Naknadna uvedba ATM je smiselna npr. za programski paket, ki ga podjetje želi še naprej razvijati in ponujati trgu. V tem primeru je izdelava avtomatskih testov in potrebna rekonstrukcija kode smiselna, saj bodo sčasoma koristi zagotovo večje od stroškov. Pri vsem skupaj moramo upoštevati tudi čas (oziroma oportunitetne

stroške), ki ga razvijalci porabijo za obvladovanje entropije sistema oziroma težnje pred razpadom, namesto da bi se posvetili razvijanju nove funkcionalnosti.

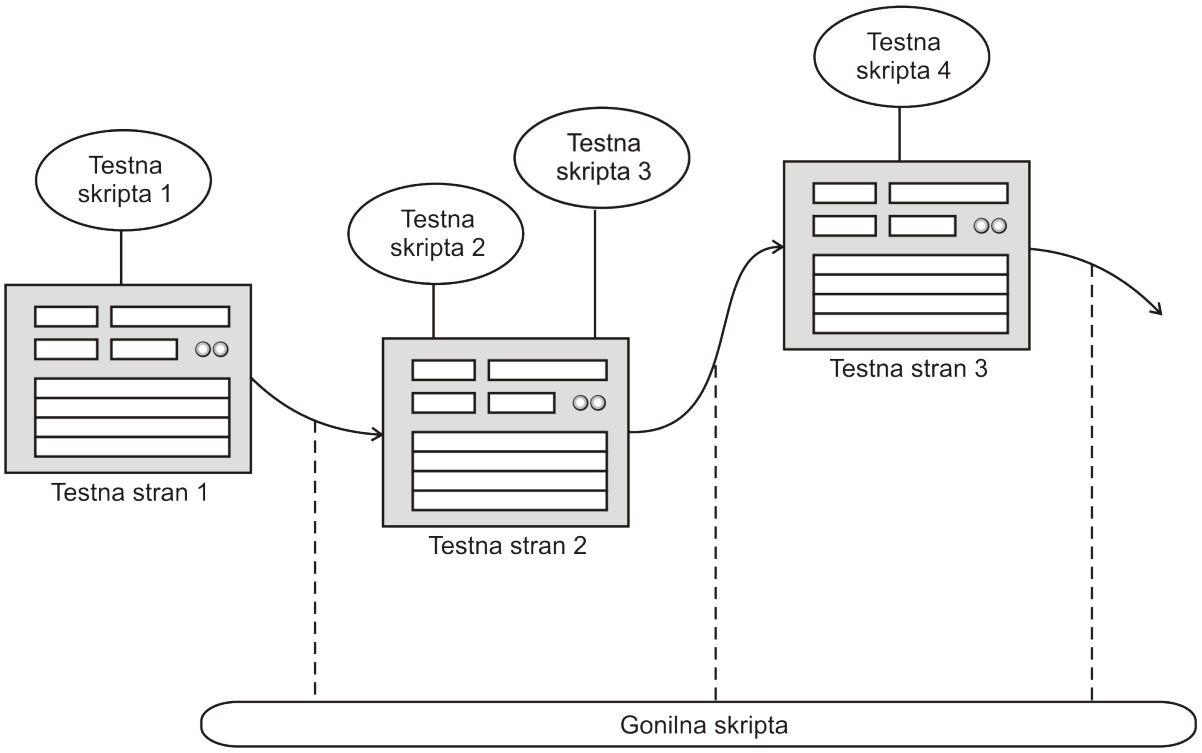
6.1.5. Težave in izzivi avtomatskega testiranja modulov

Večjo težavo pri ATM in doseganju izolacije testov predstavlja soodvisnost med testiranimi moduli (angl. *coupling*). Soodvisnost je stopnja povezanosti med moduli sistema. Več kot določeni modul ve o drugem modulu, večja je soodvisnost, kar zapleta izdelavo testov. Soodvisnosti se ne moremo izogniti, vendar težimo k temu, da je je čim manj. Že omenjena testnost kode je ključnega pomena za uspešno izvajanje ATM. Težavo lahko predstavlja neusposobljenost in nemotoviranost razvijalcev za pisanje avtomatskih testov, kar je treba reševati z ustreznimi motivacijskimi prijemi. Poleg omenjenih težav je treba opozoriti še na naslednje:

- Avtomatsko testiranje modulov ni nadomestek ostalih vrst testiranja. To je testiranje, ki ga izvaja razvijalec. Vse ostale vrste testiranj, kot so testiranje integracije in ostala sistemska testiranja, so še vedno nujno potrebna. Izvajajo jih testerji. Tudi nekateri ročni postopki, kot so formalno pregledovanje kode (angl. *code inspection)* so koristni. Samo izvajanje ATM brez ostalih vrst testiranj bo poslabšalo pregled nad sistemom oziroma, kot pravi stari rek, zaradi dreves ne bomo videli gozda.
- Testiranje uporabniškega vmesnika je težavno, zato se mu skušamo ob izvajanju ATM izogniti (npr. z navideznimi moduli). To ne pomeni, da uporabniškega vmesnika sploh ne testiramo. Testiramo ga v sklopu sistemskega oz. funkcionalnega testiranja bodisi ročno bodisi z uporabo orodij za funkcionalno testiranje. Da ATM ni dovolj, nam nazorno prikaže naslednji primer. Ob pritisku na gumb se izvede koda, ki smo jo testirali v sklopu ATM in je popolnoma pravilna. Toda neki drugi modul je gumb skril, zato ga uporabnik nikoli ne more videti. Odkrivanje takšnih vrst napak ni v domeni ATM.
- Lažen občutek varnosti. Avtomatski testi ne zagotavljajo, da je programska oprema brez napak. Razvijalec se mora zavedati, da je to le eden od pripomočkov pri razvoju. Zelena luč, ki se prikaže, ko gredo vsi testi uspešno skozi je lahko zavajajoča. Če dodamo še kak pripomoček za merjenje pokritosti kode, lahko pride do tega, da se razvijalec osredotoči na doseganje ugodne statistike namesto na uresničevanje uporabniških zahtev.

6.2. Avtomatsko funkcionalno testiranje

Avtomatsko funkcionalno testiranje je v literaturi navadno enačeno z avtomatskim testiranjem prek uporabniškega vmesnika (angl. *automated GUI testing*), čeprav se včasih ob testiranju vmesniku lahko izognemo. Tudi v pričujoči nalogi bom pojma enačil. Tovrstno testiranje po principu črne skrinjice posnema interakcijo uporabnika z aplikacijo, zato je izdelava testov manj odvisna od strukture oziroma testnosti programske kode. Zaradi tega je to včasih edini racionalen način avtomatizacije testiranja brez večjih predelav programske kode. Tveganost vpeljave avtomatskega funkcionalnega testiranja (v nadaljevanju AFT) in potrebni vložek sta veliko večja v primerjavi z avtomatizacijo testiranja modulov, zato se je treba tovrstne avtomatizacije lotiti na takšen način, da se tveganja minimizirajo.



Slika 6.4: Avtomatsko funkcionalno testiranje

Podobno kot teste modulov tudi funkcionalne avtomatske teste (v nadaljevanju testne skripte) smiselno združujemo v testne pakete. Testni paket ima več testnih skript, ki se lahko, gledano s funkcionalnega vidika, izvajajo na različnih mestih testirane aplikacije. Testne skripte navadno poganja gonilna skripta (angl. *driver*), katere naloga je pripeljati testirano aplikacijo prek uporabniškega vmesnika na pravo

mesto in po potrebi pripraviti potrebne podatke (slika 6.4). Zatem gonilna skripta izvede eno ali več testnih skript in nazadnje spet prevzame nadzor nad izvajanjem. Dobro pripravljena testna skripta mora po izvajanju pospraviti za seboj in vrniti aplikacijo na izhodiščno mesto. S tem ohranimo neodvisnost testov in lažje vzdržujemo gonilno skripto.

Orodja za podporo funkcionalnega testiranja so relativno draga, njihovo delovanje pa je treba preveriti in prilagoditi vsakemu testiranemu sistemu oz. projektu posebej. Navadno imajo ta orodja všečno funkcijo posnami-predvajaj (angl. *capture-replay*), ki sledi uporabnikovemu sprehodu skozi testirano aplikacijo in kasneje ta sprehod natančno ponovi (v nadaljevanju funkcija PP). Čeprav industrija to lastnost orodij ob prodaji venomer potiska v ospredje, izkušnje kažejo, da je vpeljava avtomatizacije testiranja s takšnim pristopom skoraj vedno obsojena na neuspeh [4, 32]. Glavni razlog je v težavnosti vzdrževanja tako izdelanih skript in dejstvu, da je takšen pristop popolnoma skregan z osnovnimi principi programiranja. Skripte, ki jih posnamemo z načinom PP, imajo nestrukturirano kodo s fiksnimi vnosnimi in drugimi podatki in so močno občutljive na najmanjše spremembe obnašanja testirane aplikacije. Če bi se podobno lotili programiranja ostalih aplikacij, bi vsaka delovala samo za točno določeno situacijo (npr. izpisali bi lahko položnico samo za točno določeno osebo) in bi zato bila neuporabna. Edina prednost tega pristopa je hitrost izdelave testa, kar pride prav pri aplikacijah, kjer se uporabniški vmesnik zelo malo spreminja, ali pa pri preizkusu orodja. Zato je treba uporabiti boljše pristope, kot sta npr. podatkovno ali ogrodna zasnovani arhitekturi, ki sta opisani v nadaljevanju.

Pomembne značilnosti orodij za avtomatizacijo testiranja prek uporabniškega vmesnika so naslednje:

- razpoznavanje objektov,
- sinhronizacija testiranja,
- preverjanje rezultatov,
- obravnavanje napak in
- programski jezik orodja.

Boljša orodja imajo še dodatne koristne funkcije, kot so vizualizacija kode, podpora za izdelavo preglednic, možnost različnih hitrosti predvajanja skript itd.

6.2.1. Razpoznavanje objektov in sinhronizacija testiranja

Novejša orodja za funkcionalno testiranje so objektno usmerjena. To pomeni, da med izvajanjem testa lahko prepoznajo elemente uporabniškega vmesnika, kot so vnosna polja, gumbi, okna itd. Preden takšen test izvede pritisk na gumb, ga mora najprej zaznati. Prva generacija orodij iz zgodnjih devetdesetih let dvajsetega stoletja tega

mehanizma ni imela, zato je pri izvajanju skript prihajalo do težav s sinhronizacijo. Skripta je lahko prehitevala in je izvedla pritisk na gumb še preden se je ta pojavil na vnosni maski.

Naslednji problem povezan z razpoznavanjem objektov je bilo premikanje po zaslonu. Skripta je hranila bodisi koordinate kazalca miške bodisi zaporedje tipk, s katerimi se je uporabnik premaknil na želeno mesto na zaslonu. Pri najmanjši spremembi uporabniškega vmesnika ali zaporedja vnosnih polj, testi niso več delovali.

Na primer namesto:

vnesi('okno.blok.polje','vrednost'))

imamo brez razpoznavanja objektov ukaz kot npr:

[TAB][TAB][TAB][TAB][TAB][Enter][TAB][TAB]{vrednost}[Enter]

Takšne skripte so neberljive in zato težavne za vzdrževanje, poleg tega pa še zelo občutljive na vsako spremembo položaja ter razporeda vnosnih polj. Torej predstavlja razpoznavanje objektov uporabniškega vmesnika pomemben dvig na višjo raven programiranja skript, kjer so nebistvene podrobnosti programerju skrite.

6.2.2. Preverjanje rezultatov in obravnavanje napak

Zelo pomembno je vedeti, kdaj je bil test uspešen ali neuspešen. Pri AFT to pomeni, da mora skripta znati prebrati vrednost nekega polja na zaslonu ali pa preveriti stanje v podatkovni bazi. Če orodje razpozna objekt, ki ga želimo preveriti, je naloga trivialna, saj samo preberemo vrednost objekta in jo primerjamo s pričakovanim rezultatom. V nasprotnem primeru imamo na voljo slikovno primerjavo, ki shranjeno sliko uporabniškega zaslona primerja z dobljeno. To seveda ni dobra rešitev, ker so s tem testi zelo občutljivi na najmanjše spremembe testirane aplikacije. Poleg tega takšni testi odpovejo pri spremembi ločljivosti zaslona. Slaba lastnost slikovne primerjave je tudi veliko pomnilnika, ki ga zahteva shranjevanje in obdelava slik. Kljub temu je včasih to edina rešitev, kadar na primer orodje ne prepozna zunanjih komponent (angl. *third-party controls*) ali pa želimo preveriti izpis poročila na zaslonu.

Razpoznava posameznega objekta je pomembna tudi pri lovljenju nepričakovanih napak. Ko se pojavi okno z napako, ga s takim orodjem lahko zaznamo in napako obravnavamo ali pa test takoj prekinemo kot neuspešen. Poleg tega tudi vemo, v kateri fazi testa je prišlo do napake. Boljša orodja znajo ob tem shraniti tudi sliko uporabniškega zaslona. Pomembno pri tem je, da programska logika obravnave napak pravilno počisti vse potrebno in vrne nadzor gonilni skripti, saj lahko samo tako ohranimo neodvisnost testov.

6.2.3. Programski jezik in ponovna uporabnost

Večina orodij za AFT omogoča programiranje skript z lastnimi programskimi jeziki, ki pa so relativno omejeni in ne omogočajo načina programiranja, kot ga omogočajo sodobni programski jeziki. Razlog za to je najbrž v tem, da so orodja za avtomatsko testiranje tržno usmerjena in pisana na kožo testerjem, poleg tega so navadno zasnovana za podporo testiranja na različnih ciljnih platformah. Ne glede na to mora programski jezik orodja omogočati vsaj osnovno modularnost, da lahko združimo dele programske kode v smiselne celote. To je osnovni pogoj za doseganje ponovne uporabnosti programske kode. Primer takšne programske kode je npr. prijava uporabnika v sistem, zapiranje in ponovni zagon brskalnika itd.

6.2.4. Vizualizacija testne kode in druge dodatne funkcije

Uporabna lastnost nekaterih orodij je prikaz vsakega koraka testne skripte s sliko uporabniškega zaslona. Tako je takoj razvidno, kaj skripta izvede ob določenem koraku, kar olajša popravljanje skript. Drugi stranski učinek te funkcije je izobraževalni. S pomočjo slik lahko tester vidi, kaj naj bi test delal, po drugi strani pa kako naj bi predvidoma delovala testirana aplikacija. V zvezi s tem je koristna tudi možnost nastavljanja hitrosti izvajanja testov. S tem lahko izbrane teste uporabimo tudi v izobraževalne ali predstavitvene namene.

6.2.5. Metode izdelave testnih skript (podatkovno in ogrodno zasnovana arhitektura)

Veliko projektov avtomatizacije funkcionalnega testiranja je obsojenih na neuspeh. Za to lahko obstaja množica razlogov, eden od teh je naivno zanašanje na uporabo že omenjene funkcije PP in odsotnost projektnega pristopa. Če želimo izdelati testne skripte, ki bodo primernejše za vzdrževanje, jih je treba programirati in prilagoditi za daljšo uporabo. Pri uspešnih projektih avtomatizacije se uporabljata dva delujoča pristopa izdelave testnih skript [32]:

- podatkovno zasnovana arhitektura (angl. *data-driven architecture*) in
- ogrodno zasnovana arhitektura (angl. *framework driven architecture*).

Možno in priporočljivo je uporabiti kombinacijo obeh. Podatkovno zasnovana arhitektura uporablja principe programskih jezikov četrte generacije, ogrodno zasnovana pa principe jezikov tretje generacije. Pri obeh pristopih gre za to, da dvignemo skripte na višji nivo, saj so potem lažje za vzdrževanje in testerjem bolj razumljive.

Podatkovno zasnovana arhitektura

Pri tem pristopu naredimo ločnico med podatki v testni skripti in testno logiko. Podatke hranimo v testni matriki. Vrstice matrike so testni primeri, stolpci so parametri. Testni primer je tako kombinacija vrednosti več parametrov. Test prebere vrstico in izvede mini skripto za vsak parameter. V matriki so lahko vhodni testni podatki v skripto [4], lahko pa kot podatke obravnavamo tudi programske ukaze, elemente uporabniškega vmesnika in pričakovane vrednosti [4, 32]. Na takšen način je v matriki istočasno dokumentirano predvideno delovanje testirane aplikacije, kot tudi samo izvajanje testa, ki je opisano korak za korakom. Z uporabo testne matrike torej ločimo podatkovno stran od ostale programske kode. Poleg veliko lažjega vzdrževanja takšnih skript, so preglednice bolj razumljive tudi testerjem brez znanja programiranja. Ti lahko sami do določene mere vzdržujejo testne podatke in dodajajo testne primere.

Možnih izvedb podatkovno zasnovane arhitekture je več. Testne matrike lahko vsebujejo samo vhodne podatke, lahko pa tudi programske ukaze, kar je odvisno od lastnosti posameznega orodja in izvedbe pristopa. Ob tem ne smejo postati nepregledne in prezapletene, sicer se vse omenjene prednosti pristopa izničijo.

Princip delovanja testne matrike je razviden iz naslednje psevdokode:

```
naloži testno matriko M
   za vsako vrstico matrike M(i)
      za vsako polje matrike M(i,j) izvedi skripto:
      - postavi se v ustrezno polje uporabniškega vmesnika
      - postavi vrednost iz matrike M(i,j) polju
      - preveri vnos
   preveri rezultat
```

Testna matrika je lahko navadna preglednica. Njena struktura in vsebina je prilagojena strukturi uporabniškega vmesnika oziroma zaokroža neko funkcionalno celoto testirane aplikacije (preglednica 5).

Testni primer	Opis	Davčna št. izvajalca	Številka računa	Datum računa	Znesek računa	Pričakovani izhod
RZP001	Vnos navadnega računa izvajalca	21706468	<<RANDOM>>	<<SYSDATE>>	1234,56	<<OK>>
RZP002	Vnos računa z datumom večjim od sistemskega	21706468	<<RANDOM>>	<<SYSDATE+1>>	1234,56	Napaka(001)
RZP003	Vnos računa z negativnim zneskom	21706468	<<RANDOM>>	<<SYSDATE>>	-1234,56	Napaka(002)
...						

Preglednica 5: Primer testne matrike

V preglednici je naveden primer testiranja vnosa računa z enim pozitivnim in dvema negativnima testnima primeroma. V preglednici so samo vnosni podatki, programski ukazi niso parametrizirani. Vsak stolpec preglednice je v paru z določenim poljem uporabniškega vmesnika (izjema so opisni stolpci in stolpec pričakovanega izhoda). Pred vnosom v polje vsak podatek obdela še posebna skripta oziroma modul, ki nekatere dogovorjene vrednosti logično pretvori (npr. polje <<SYSDATE>> pretvori v trenutni datum). Na koncu dobljeni izhod testirane aplikacije primerjamo s pričakovanim izhodom.

Ogrodno zasnovana arhitektura

Pri tem pristopu testirano aplikacijo ločimo od testnih primerov z izdelavo množice programskih modulov, ki delujejo kot vmesna plast. Module hranimo v skupni knjižnici in jih uporabljamo kot ostale ukaze programskega jezika testirnega orodja. Gre za sicer drugačen pristop od podatkovnega, vendar se oba lepo dopolnjujeta, najboljše rezultate dosežemo z uporabo obeh.

Z izdelavo knjižnic želimo: [32]:

- doseči modularnost kode testnih skript,
- izboljšati ponovno uporabnost kode,
- zmanjšati vpliv sprememb uporabniškega vmesnika testirane aplikacije,
- lažje obvladovati zunanje komponente in
- zbrati na enem mestu splošno koristne funkcije (npr. kodo za obravnavo napak).

Ogrodje je lahko sestavljeno iz različnih modulov, od preprostih, ki samo kličejo drugi modul, do bolj zapletenih sestavljenih skript, ki izvedejo zaokroženo nalogo. V

ogrodje spada testna programska koda, ki pokriva del funkcionalnosti testirane aplikacije, ukazi orodja za testiranje, večji zapleteni kosi testnih primerov, ki so uporabni na več mestih, in razni drugi splošno uporabni moduli.

Primer kode iz aplikacije je npr. skripta za odpiranje datoteke odpriDatoteko(x). Ta lahko vsebuje množico ukazov, kot npr. vhod v meni, odpiranje pogovornega okna za odpiranje datoteke, vnosa imena datoteke v pogovorno okno, pritisk na gumb itd. Poleg tega lahko vsebuje obravnavanje napak in drugo primerno dodatno kodo. Programerja ob pisanju testnih skript seveda ne bo zanimala izvedba modula, ampak se bo lahko osredotočil na bolj pomembne probleme testiranja. V primeru, da se spremeni pot do menija za odpiranje datotek ali se celo želimo izogniti meniju in odpreti datoteko mimo uporabniškega vmesnika, je treba spremeniti kodo samo na enem mestu. V posebne module moramo spraviti kodo zunanjih komponent, če jih orodje slabo podpira. Za prepoznavanje takšnih komponent je ponavadi treba poiskati zasilne rešitve in trike, ki običajno niso najbolj robustni, zato so ti moduli bolj občutljivi na spremembe uporabniškega vmesnika.

Včasih moramo v posebne module vgraditi tudi ukaze orodja za testiranje. Na ta način se npr. izognemo hroščem v delovanju orodja, dodamo svojo obravnavo napak ali razširimo uporabnost ukaza. Poleg tega je dobro na enem mestu zbrati razne uporabne splošne funkcije kot je npr. prijava v sistem, priprava in pospravljanje testnega okolja itd.

V knjižnice lahko spravimo tudi skripte, ki smiselno zaokrožajo večji kos funkcionalnosti ali poslovne logike testirane aplikacije. Pred tem jih seveda ustrezno posplošimo. Pri tem je treba upoštevati zdravo mero, zaradi splošnega problema izdelave ponovno uporabne kode, že omenjenega v poglavju 6.1.3. Če je namreč modul premalo splošen, je tudi njegova uporabnost majhna, če je prezapleten za uporabo, pa prav tako.

Spodaj je kot primer prikazan del skripte, kjer so uporabljeni principi podatkovne in ogrodno zasnovane arhitekture:

```
Function f_nova_stevilka_racuna:var
    -- generator naključnih števil
    RandomSeed( secs()*mins() )

    -- izdelaj naključno številko po pravilu
    l_x = Random( 1234567 )
    l_st = strcat("", "TEST/", str(l_x) )

    -- obravnavanje napak ?
    return l_st
End function
```

```
Function f_vpisi_stevilko_racuna:var

    -- preberi številko iz datoteke
    l_st_racuna = f_beri_iz_datoteke( st_vrstice, 'rsv.st_racuna' );

    -- če ni določena naredi naključno
    if l_st_racuna="<<RANDOM>>"
       l_st_racuna=f_nova_stevilka_racuna()
    endif;

    -- izpolni polje v upo. vmesniku
    EditText "$RSV_VNOS_STEVILKA_RACUNA_0", l_st_racuna
    TypeToControl "Edit", "$RSV_VNOS_STEVILKA_RACUNA_0", "{Tab}"

    -- obravnavanje napak ?
    return l_st_racuna
End function
```

V navedenem primeru je prikazan del kode, kjer v uporabniški vmesnik vnašamo račun. Podatki so od programske kode ločeni v posebni preglednici. Z njo si tester predhodno pripravi testne podatke. Eden od podatkov je tudi številka računa. Če je to dogovorjena vrednost (npr. “<<RANDOM>>“), sistem izdela naključno številko, sicer velja vrednost, prenesena iz datoteke. Zatem se šele prenese v uporabniški vmesnik v ustrezno polje.

Pri izdelavi ogrodja naj bi se izognili skušnjavi, da bi izdelali popolno, vse obsegajočo knjižnico, ki bi pokrivala vse potrebe testiranja [32]. Kar nekaj projektov avtomatizacije testiranja je namreč propadlo zaradi tega, ker je pred dokončanjem ogrodja ravnateljstvo projekta izgubilo potrpljenje in projekt ustavilo. Ogrodje je treba graditi in izboljševati sproti ter seveda ne pozabiti, da je končni cilj testiranja odkrivanje napak v testirani aplikaciji, ogrodje in ostali pripomočki pa so samo sredstvo na poti do tega cilja.

Podobnost razvoja avtomatskih testov z razvojem programske opreme

Navedeni principi izdelave testnih skript za AFT kažejo, da je med izdelavo avtomatskih funkcionalnih testov in izdelavo druge programske opreme veliko stičnih točk in podobnosti. Npr. podatkovno zasnovana arhitektura je v grobem podobna ločevanju podatkovne plasti pri avtomatskem testiranju modulov ali bolj splošni MVC arhitekturi (angl. *Model View Controller*) pri razvoju programske opreme.

Razvoj avtomatskih testov mora zato biti podoben razvoju ostale programske opreme [4, 12]. Predvsem ravnateljstvo (angl. *management*) navadno dobi napačen vtis, da bodo testne skripte lahko razvijali testerji brez znanja programiranja. To je možno samo z uporabo pristopa PP, katerega slabosti sem že navedel. V praksi je tudi za sam proces programiranja testov značilno, da se ga ne jemlje dovolj resno in je nekako potisnjen na stranski tir. Sicer že uveljavljeni standardi za razvoj programske opreme v podjetju se pri programiranju testov ne upoštevajo, načrtuje se

sproti ali pa sploh ne [33]. Brez projektnega pristopa in načrtovanja ni mogoče razviti kakovostne programske kode z obvladljivimi stroški vzdrževanja. Vse to velja tudi za programsko kodo avtomatskih testov.

6.3. Strukturna metodologija ATLM

ATLM (angl. *Automated Test Lifecycle Methodology*) je strukturna metodologija, usmerjena k uspešni vpeljavi avtomatizacije testiranja skozi šest faz in množico aktivnosti, ki se izvajajo vzporedno z razvojnim ciklom programske opreme (slika 6.5). Obravnava celovit pristop k avtomatizaciji na vseh ravneh, od testiranja modulov do sistemskega testiranja. Pri vpeljavi avtomatizacije je poudarjen projektni pristop, upravljanje s tveganji in koristmi avtomatizacije ter pomen pridobitve podpore ravnateljstva za projekt. Posebej obravnava planiranje, analizo, načrtovanje, izvajanje in upravljanje procesa testiranja. Doseg testiranja je zajet v testnem planu, ki predstavlja opis izbranega pristopa in izvedbe testiranja na višji ravni. V nadaljevanju je doseg testiranja podrobneje razdelan skozi strategijo, namen, cilje testiranja in testne zahteve. Podobno kot pri razvoju programske opreme so testne zahteve definirane pred načrtovanjem testiranja.

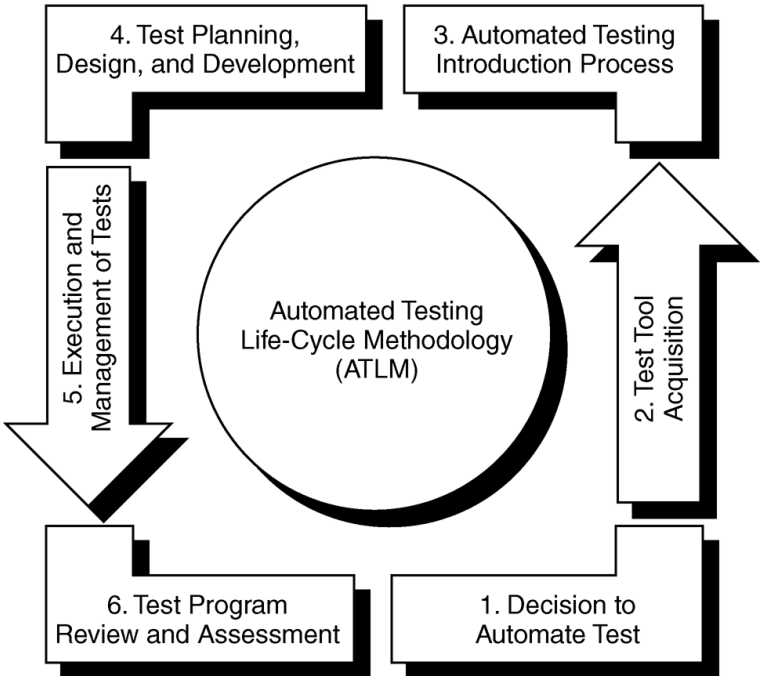
Metodologija ATLM je razdeljena na naslednjih šest faz [4]:

1. Odločitev o avtomatizaciji testiranja
2. Izbor orodij
3. Uvod v proces avtomatizacije
4. Analiza, načrtovanje in razvoj testov
5. Izvajanje testov in upravljanje s konfiguracijo
6. Spremljanje in analiza rezultatov testiranja

6.3.1. Odločitev o avtomatizaciji testiranja

To je prva faza metodologije, v kateri je poudarjen pomen pridobitve popolne podpore ravnateljstva za vpeljavo avtomatizacije testiranja. Potrebno je razjasniti morebitna napačna pričakovanja o avtomatizaciji, prikazati resnične koristi in pasti uvedbe avtomatizacije, predstaviti orodja, ki so na trgu, ter narediti analizo stroškov in koristi avtomatizacije (poglavje 6.4).

Pridobitev ključnih odgovornih oseb (ali sponzorjev) je odločilnega pomena za uspeh vsakega projekta, tudi projekta vpeljave avtomatizacije testiranja. Treba je zagotoviti, da bo na voljo dovolj sredstev ne samo za nakup orodij, temveč med drugim tudi za plačilo dodatnih ljudi na projektu. Poleg testerjev so za uspeh avtomatizacije testiranja ključni izkušeni programerji. Upoštevati je treba, da bodo na začetku stroški avtomatizacije večji od koristi, možni so tudi negativni vplivi na doseganje rokov zaradi kratkoročno zmanjšane učinkovitosti testerjev. Testna skupina, oziroma njeni odgovorni, morajo dovolj zgodaj ugotoviti, ali se ravnateljstvo projekta zaveda vseh



Slika 6.5: Metodologija življenjskega cikla avtomatskega testiranja ATLM

predvidenih stroškov avtomatizacije in ali jih je pripravljeno pokriti. Če temu ni tako, lahko skušajo vodstvu bolj nazorno predstaviti potencialne koristi s podrobno analizo stroškov in koristi. Če tudi to ne zaleže, potem je treba prilagoditi projekt avtomatizacije sredstvom, ki so na voljo, ali pa ga opustiti.

V praksi se večkrat zgodi, da ima ravnateljstvo projekta nemogoča pričakovanja o avtomatizaciji testiranja [4, 32, 34]. Najslabša možnost je tista, ko vodstvo odobri nakup orodja, ni pa pripravljeno dovolj vlagati v osebje in izobraževanje, ali pa se vpeljave avtomatizacije ne loti na ustrezen projektni način. Po dragem začetnem vložku v orodje in veliki količini porabljenega časa in ostalih virov, orodje navadno konča na polici (postane t.i. *shelfware*).

Omenjena nemogoča pričakovanja ravnateljstva ali najpogostejši miti o avtomatizaciji so [4]:

- **Popolna avtomatizacija procesa testiranja.** Ni orodja, ki bi podprlo celoten proces testiranja od analize prek načrtovanja testov, izdelave testov, izvajanja testov in analize rezultatov. S primernimi orodji lahko nadgradimo obstoječi proces ročnega testiranja in na dolgi rok dosežemo prihranke. Če pa ni niti ročni proces testiranja dobro definiran in ostaja na ravni ad-hoc testiranja, potem je uvajanje avtomatizacije nekoristno, v najboljšem primeru lahko deluje kot sprožilec k izboljšavi organizacije.

- **Orodje pokriva vse platforme.** Večina orodij za testiranje podpira širok spekter operacijskih sistemov in platform (npr. testiranje spletnih Java aplikacij, aplikacij v DOS okolju itd.), kar pa je hkrati tudi njihova slabost, ker posledično orodja niso popolnoma prilagojena določeni platformi. Nobeno orodje pa ne podpira popolnoma vseh okolij. Zaradi tega je treba vedno računati na težave, še posebej pri zunanjih, manj standardnih komponentah testiranega sistema (angl. *third party components*).
- **Takojšnji rezultati z uporabo funkcije PP (posnami-predvajaj).** Orodja za avtomatsko testiranje naj bi bila s funkcijo posnami-predvajaj in ničelnim vložkom takoj uporabna za testerje brez znanja programiranja. V resnici je proces uvedbe avtomatizacije testiranja soroden procesu razvoja programske opreme in se ga je tako tudi treba lotiti. Potrebno je znanje programiranja in velik začetni vložek v izdelavo vmesnika, ki bo primeren za testerje. Zaradi tega izdelava avtomatskih testov vsaj na začetku zahteva veliko časa, navadno nekajkrat več kot priprava ročnega testa.
- **Orodje je dobro, ker omogoča funkcijo PP**. To je sicer koristna lastnost orodja, vendar nam sploh ne zagotavlja uspešne avtomatizacije. V praksi je uspešnost s takšnim pristopom redka, ker takšne skripte težko vzdržujemo [3, 9]. Pristop k avtomatizaciji s to funkcijo je uspešen le v primeru, da je testirani izdelek tako stabilen, da skoraj ne zahteva popravljanja testov.
- **Vsi testi bodo avtomatizirani**. To v resnici ni praktično izvedljivo in niti stroškovno upravičeno.
- **Takojšen prihranek v času in zaposlencih**. Včasih se koristi avtomatizacije testiranja pokažejo takoj, v večini primerov pa se stroški povrnejo šele kasneje. Izdelava avtomatskih testov zahteva veliko vlaganja in dela, prihranki se navadno pokažejo, ko so testi robustno izdelani in jih lahko večkrat uporabimo z relativno malo vzdrževanja. Velja splošno pravilo, da se koristi uvajanja avtomatizacije na izdelku različice N pokažejo šele v različici N+1 ali kasneje.
- **Proces testiranja ni definiran, kar lahko rešimo z avtomatizacijo**. Proces testiranja mora biti dobro definiran, sicer ni kaj avtomatizirati. Včasih je veliko koristi pri uvajanju avtomatizacije testiranja posredne. Organizacija je prisiljena povečati disciplino pri testiranju in izboljšati načrtovanje testiranja. Te nujne spremembe so sicer le stranski učinek avtomatizacije in bi bile koristne tudi pri ročnem testiranju. Vpeljava avtomatizacije vodi k večji profesionalizaciji, izboljšanemu procesu samega testiranja, boljše je sodelovanje z razvijalci itd.

S pravim pristopom vpeljave avtomatizacije so se v praksi pokazale koristi, ki naložbo upravičijo. Avtorji ATLM in ostali zagovorniki avtomatizacije testiranja navajajo naslednje:

- **Dimno in regresijsko testiranje.** Avtomatizacija regresijskega testiranja je osrednja tema te naloge, ker omogoča lažje in varnejše preoblikovanje programske kode. Izkušnje kažejo, da je količina napak ob spreminjanju programske kode večja ko pri pisanju nove. Obsežno regresijsko testiranje je praktično nemogoče v celotnem obsegu izvajati ročno in dovolj pogosto. Poleg tega je ponavljajoče in dolgočasno. S tem negativno motivira testerje.
- **Obremenitveno in stresno testiranje.** Nekatera orodja omogočajo obremenitveno in stresno testiranje, tako da hkrati teče več procesov, s čimer simuliramo delo več uporabnikov. Če želimo izvajati ročno testiranje na podoben način, potrebujemo veliko število testerjev, ki bodo istočasno izvajali teste. To je nepraktično in tudi drago, ker je treba najprej imeti na voljo ustrezno število ljudi in jih tudi plačati. Hkrati lahko z nekaterimi orodji simuliramo delovanje oz. izpad nekaterih delov sistema (npr. simuliramo obnašanje aplikacije pri različnih hitrostih omrežja). Možno je tudi iskanje grl v testiranem sistemu, kjer orodja ob tem ponujajo različne metrike in poročila. Kot nasprotje lahko vzamemo primer večjega podjetja v ZDA, kjer so še leta 1998 merili odzivnost sistema s štoparicami [4].
- **Prihranek v času in zaposlencih.** Kot je že bilo omenjeno, se koristi avtomatizacije funkcionalnega testiranja pokažejo šele kasneje, zato so pričakovanja o takojšnjih prihrankih in drugih pozitivnih učinkih napačna. Na dolgi rok lahko dejansko pričakujemo znaten prihranek v času in tudi v zaposlencih. Drastičen primer je npr. zgoraj omenjeno ročno obremenitveno testiranje, kjer potrebujemo npr. deset ljudi, če želimo testirati obnašanje sistema pri desetih uporabnikih. Z uspešno vpeljavo ustreznega orodja lahko ti zaposlenci opravljajo koristnejše in bolj zanimivo delo.
- **Izobraževanje in lažje uvajanje.** Avtomatski testi so med drugim zelo kakovostna dokumentacija. Avtomatski testi modulov delujejo kot primeri, ki pomagajo pri razumevanju kode modula, kako naj deluje in kako ne sme delovati. Programer lahko iz testov razbere, kaj je imel drugi programer v mislih oziroma kako si je predstavljal pravilno delovanje modula. Podobno so lahko tudi sistemski avtomatski testi uporabna dokumentacija o delovanju sistema. Ker se hitreje učimo na podlagi primerov, lahko na ta način izobražujemo nove ljudi na projektu, tako programerje kot testerje. Nekatera orodja za testiranje prek uporabniškega vmesnika nudijo možnost nastavljanja hitrosti izvajanja skript in shranjevanja zaslona po vsakem koraku, ki ga izvede skripta. Na ta način je lažje razumeti delovanje skript, kot tudi delovanje testirane aplikacije. Zelo pomembno pri tem je, da je tovrstna dokumentacija za razliko od uradne vedno ažurna in se nanjo lahko zanesemo, ker moramo teste vedno uspešno izvesti.
- **Testiranje izven delovnega časa.** Obsežno regresijsko ali kako drugo testiranje lahko izvajamo izven delovnega časa, ko so viri nezasedeni, brez

človeške prisotnosti in tako dodatno izkoristimo zmogljivosti, ki so na voljo.

6.3.2. Izbor orodij

Ta faza vodi testnega inženirja skozi celoten odločitveni postopek izbire orodij. Začne se takrat, ko je popolnoma zagotovljena podpora ravnateljstva. Poleg orodij za testiranje prek uporabniškega vmesnika obstaja še širok nabor ostalih orodij za podporo procesa testiranja v vseh fazah razvoja. Idealno bi bilo izbrati primerna testna orodja hkrati z ostalimi razvojnimi orodji, kar pa je v praksi redko. Navadno je razvojno okolje že postavljeno in utečeno, zato se je s testnimi orodji treba temu prilagajati. Pri izboru moramo nujno upoštevati razvojno okolje v organizaciji (angl. *systems engineering environment*). Na podlagi tega določimo potencialna orodja za podporo testiranju, ki so na voljo na trgu. Idealno bi bilo podpreti vsako fazo življenjskega cikla programske opreme med drugim tudi z ustreznimi orodji za testiranje. V pregled možnih orodij so zato poleg strokovnjakov za testiranje vključeni tudi zaposlenci zadolženi za optimizacijo procesa razvoja programske opreme. Določiti je treba merila izbora oz. množico kriterijev, ki bo osnova za pripravo odločitvenega modela. Kriteriji so npr. cena, združljivost, uporabnost na različnih projektih, težavnost integracije z ostalimi orodji itd.

Treba je pripraviti pilotni projekt in infrastrukturo za preizkus orodij. Zatem lahko testna skupina na podlagi predhodno določenih meril izbere orodja in jih, če je le mogoče, pridobi na preizkus. Pri preizkušanju orodij za avtomatsko testiranje prek uporabniškega vmesnika pride prav funkcija PP. Preizkusna doba je navadno kratka, zato je to najhitrejši način, da izdelamo nekaj delujočih testov. Ob tem je dobro preveriti, ali orodje prepoznava elemente uporabniškega vmesnika, kako uspešno je lovljenje napak itd. Vsi ti elementi morajo biti jasno določeni že v merilih izbora.

6.3.3. Uvodni proces vpeljave avtomatizacije v projekt

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

Analiza procesa testiranja

Tretja faza se prične z analizo obstoječega procesa testiranja v organizaciji. Navadno se testiranje tako ali drugače že izvaja. Če proces testiranja ni dobro definiran (ad-hoc testiranje), je prvi korak, ki ga je treba storiti, definiranje procesa. Testni proces mora biti dokumentiran v takšni obliki, da ga lahko predstavimo vsem vpletenim. Če proces ni dokumentiran, ni ponovljiv, niti razumljiv vsem udeležencem. To posledično pomeni, da se proces ne bo izvajal. Poleg tega nedokumentiranega procesa ne

moremo ne meriti niti izboljševati in še manj avtomatizirati. Definirati je treba strategijo, namen in cilje testiranja. Izbrati in dokumentirati je treba primerne tehnike testiranja in najboljšo prakso (bodisi že obstoječo v organizaciji bodisi iz zunanjih virov).

Preučitev testnih orodij

V prejšnji fazi je testna ekipa definirala nabor perspektivnih testnih orodij, ki naj bi na splošno podpirala proces testiranja v organizaciji. V tej fazi je treba preveriti, ali so izbrana orodja dejansko ustrezna in ali jih lahko uporabimo v ciljnem projektu. Pri tem moramo upoštevati dejanske zahteve testiranja v ciljnem projektu, ljudi in opremo, ki so na voljo, ciljno okolje in značilnosti testirane aplikacije. Treba je uskladiti urnik projekta in preveriti, ali je na voljo dovolj časa za vpeljavo orodij. Idealno bi bilo, če bi se vse te priprave izvajale že od samega začetka ciljnega projekta, kar pa se ne zgodi vedno. Razvojni skupini na projektu, v katero spadajo razvijalci, testni inženirji, zadolženi za upravljanje s konfiguracijo in drugi, je treba predstaviti orodja in pri tem prikazati dejanske koristi, pa tudi morebitne težave pri delu z orodji. Če testirana aplikacija že obstaja, je treba na njej preizkusiti orodja in preveriti združljivost z manj standardnimi komponentami (angl. *third-party controls*). V primeru odkritih problemov z združljivostjo je treba najti nadomestne rešitve.

Jasno je treba določiti vloge in odgovornosti na projektu glede na usposobljenost in znanje kadra v skupini za testiranje. Če je usposobljenost kadra pomanjkljiva, je rešitev v dodatnem usposabljanju, v pomoč so lahko tudi zunanji strokovnjaki, specializirani za konkretno orodje.

6.3.4. Planiranje, analiza, načrtovanje in izdelava testov

Testno planiranje zajema pregled vseh aktivnosti, potrebnih za izvajanje testiranja in verifikacijo, da bodo procesi, metodologije, tehnike, osebje, orodja in oprema organizirani in uporabljeni na učinkovit način. Učinkovita uporaba avtomatskih orodij za testiranje zahteva znatni vložek v testno planiranje oziroma pripravo testiranja, ker lahko avtomatiziramo le dobro definiran in dokumentiran proces. Rezultat testnega planiranja je testni plan, v katerem je zbrana vsa potrebna dokumentacija o testiranju. Testni plan se prične oblikovati že v predhodnih fazah in se sproti dopolnjuje skozi vse nadaljnje faze projekta.

V testnem planu so med drugim zajeti:

- vloge in odgovornosti udeležencev projekta,
- doseg testnega procesa, glede na omejitve v času, zaposlencih in ostalih potrebnih virih,
- strojna, programska oprema, omrežne zahteve za testno okolje, način upravljanja s konfiguracijo,

- uporabljene tehnike testiranja po principu črne in bele skrinjice, pristop k načrtovanju testov, standardi kodiranja in poimenovanja testnih procedur,
- časovni plan za razvoj in izvajanje testnih primerov,
- zahteve po testnih podatkih in način njihove pridobitve oziroma generiranja,
- razne povezovalne matrike, kot npr. matrika povezav med testnimi primeri in primeri uporabe, matrika odvisnosti testnih primerov itd.

Na podlagi testnega plana in časovnih okvirov v njem mora testna skupina pripraviti testno okolje.

Analiza in načrtovanje testiranja

Avtomatizacija testiranja zahteva celovit pristop podobno kot razvoj programske opreme in sicer z definicijo zahtev, analizo, načrtovanjem in kodiranjem. Pred fazo načrtovanja je treba izvesti analizo zahtev testiranja in jih dokumentirati. Zahteve testiranja izhajajo iz zahtev testiranega sistema. Pridobimo jih lahko z različnimi tehnikami bodisi iz primerov uporabe bodisi iz drugih modelov iz analize testiranega sistema. Iz analize zahtev nastane povezovalna matrika med zahtevami testiranja in primeri uporabe ter povezovalna matrika med tehnikami testiranja in zahtevami testiranja.

Po definiranju zahtev testiranja se prične načrtovanje testnih primerov. Za vsak testni primer je treba med drugim določiti, ali gre za ročni ali avtomatski test. Treba je dokončno definirati in dokumentirati standarde kodiranja ter pristop k izdelavi programske kode testov, ki bo v največji meri omogočal ponovno uporabnost, robustnost in čim bolj enostavno vzdrževanje kode. Pri tem uporabimo podobne principe kot pri razvoju programske opreme. Določiti je treba dinamične in statične testne tehnike, za katere je bilo predhodno ugotovljeno, da je njihova uporaba upravičena glede na ugotovljene zahteve testiranja. Sledi definicija testnih primerov. Definicija testnega primera med drugim zajema osnovne podatke, kot so ime, avtor in ostali podatki, kot so vhodi, izhodi, predpogoji, potrebne akcije, odvisnosti, doseg testiranja, in metodo verifikacije v povezavi s testno zahtevo. Podrobnost definicije je odvisna od zapletenosti testnega primera. Bolj zapletene primere je treba označiti in jih podrobneje preučiti. Definirane testne primere je treba še povezati z viri testnih podatkov in jih smiselno združiti v skupine.

Razvoj testnih primerov

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

Treba je podrobno razdelati časovni okvir razvoja testov in ga seveda med razvojem tudi ustrezno prilagojevati. Pristop k razvoju naj bo tak, da bo omogočal čim večjo modularnost kode in ponovno uporabnost (npr. uporaba principov podatkovno in ogrodno usmerjene arhitekture). Za vsako ugotovljeno nezdružljivost z orodjem je treba najti nadomestno rešitev, oceniti potreben čas za izvedbo in rešitev ustrezno dokumentirati. Podobno kot pri razvoju programske opreme je priporočljivo uporabljati tehnike, kot je npr. pregledovanje kode, da se zagotovi njena ustrezna kakovost in preverja upoštevanje postavljenih standardov. Potrebno je zagotoviti tudi upravljanje s konfiguracijo za testne primere, podatke, knjižnice itd.

6.3.5. Izvajanje testiranja in analiza rezultatov

V fazi izvajanja testiranja se v skladu s testnim planom in urnikom v okviru plana izvedejo predhodno pripravljeni testni primeri. Hkrati se dobljeni rezultati testiranja zbirajo in analizirajo. Tu je mišljeno izvajanje testiranja na vseh ravneh, od testiranja modulov do sistemskega in prevzemnega testiranja. Po izvedenem testiranju se beležijo odkrite napake in naknadno izvedeni popravki ter ustrezno ažurira testna dokumentacija. Raven dokumentiranja in formaliziranja celotnega procesa testiranja je odvisna od potreb posamezne organizacije oziroma konkretnega projekta.

Rezultati posameznih testov so lahko naslednji:

- Pozitivni rezultat – test se je izvedel uspešno, ker testirana komponenta oziroma sistem deluje pravilno.
- Negativni rezultat – test se je izvedel neuspešno, ker testirana komponenta oziroma sistem deluje nepravilno.
- Lažni pozitivni rezultat – test se je izvedel uspešno, čeprav v resnici testirana komponenta deluje napačno. Razlogi so lahko v napačnem branju rezultatov testiranja, šibkih kontrolah, napačnem razpoznavanju objektov uporabniškega vmesnika itd. Takšni testi imajo majhno vrednost, ker dajejo lažen občutek varnosti, zato jih moramo popraviti, tako kot teste, ki dajejo lažne negativne rezultate.
- Lažni negativni rezultat – test se je izvedel neuspešno, čeprav testirana komponenta deluje pravilno. Po izvedbi testiranja dobljeni rezultat primerjamo s pričakovanim. Če test ne uspe, to ne pomeni nujno, da je nekaj narobe s testirano aplikacijo. Razlog je lahko v spremembi aplikacije, kateri test ni bil prilagojen, v napakah pri pripravi testnih podatkov, v logičnih napakah v testnih skriptah, v napakah testnega okolja itd.

Za potrebe analize rezultatov lahko uporabimo različne metrike. Z njimi beležimo kazalce o napredovanju testiranja in stopnji pokrivanja testov. Možnih metrik je veliko, ker pa vsaka analiza zahteva čas, moramo smiselno izbrati tiste, ki prinesejo

največ informacij. Ena od nalog testne skupine je odkrivanje komponent sistema, kjer je napak nadpovprečno veliko. Kot koristna in relativno preprosta je omenjena metrika *stopnja sprememb*, ki vsebuje komponente sistema, kjer je največ spreminjanja. Za takšne komponente je priporočljivo pripraviti dodatne teste. To so hkrati primerni kandidati za rekonstrukcijo oziroma izboljšavo programske kode. Primeri ostalih možnih metrik so še:

- pokrivanje testiranja – število testnih procedur v primerjavi s številom testnih zahtev,
- analiza trenda napak – število najdenih napak v primerjavi s številom izvedenih testnih primerov,
- trenutna stopnja kakovosti – število uspešnih testnih primerov v primerjavi s številom vseh izvedenih,
- gostota napak – število vseh najdenih napak v primerjavi s številom izvedenih testnih primerov po določenem področju (npr. za primer uporabe ali za testno zahtevo),
- vrednost avtomatizacije – število ur za razvoj skript v primerjavi s številom odkritih napak, ki bi jih sicer težko odkrili z ročnim testiranjem.

6.3.6. Pregled in ocena procesa testiranja

Ta faza se izvaja skozi celoten življenjski cikel avtomatizacije testiranja s ciljem stalnih izboljšav procesa. Med procesom in predvsem po izvajanju testnih primerov je treba izvajati izbrane metrike ter analizirati rezultate. Pri tem ATLM predlaga naslednje korake:

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

ATLM dokaj poudarja ravnateljsko plat (angl. *managers view*) testiranja in manj izvedbeno. Glede na pristop je to sicer “težka“ metodologija, ki daje precej poudarka dokumentiranju oziroma formalizaciji procesa. Kljub temu je to edina celovita metodologija za vpeljavo avtomatizacije testiranja. Obravnava pristop k avtomatizaciji na vseh ravneh od testiranja modulov do sistemskega testiranja, čeprav največ poudarka daje avtomatizaciji funkcionalnega oz. sistemskega testiranja. V tem pogledu se ustrezno dopolnjuje z agilnimi metodologijami, kjer je v ospredju avtomatizacija testiranja modulov. Ker gre pri avtomatizaciji funkcionalnega testiranja za bistveno večje vložke in tveganja kot pri avtomatizaciji testiranja modulov, je tudi formalizacija in večplastnost te metodologije do neke mere upravičena. Kljub temu da ATLM ne moremo šteti med agilne metodologije, pa nas na koncu lahko kljub temu pripelje do želenega cilja, to je do lažjega obvladovanja sprememb ob razvoju programske opreme in s tem do večje agilnosti.

6.4. Analiza stroškov in koristi avtomatizacije

S povračilom naložbe (angl. *Return of investment* – v nadaljevanju ROI) merimo učinkovitost investicije. Dobimo jo tako, da vrednost koristi investicije delimo z vrednostjo, ki smo jo porabili za investicijo [34].

Finančne posledice avtomatizacije testiranja najlažje izračunamo s primerjavo stroškov in koristi iz obdobja pred uvedbo avtomatizacije. Predpostavimo torej, da je ročno testiranje že vpeljano. Stroške avtomatizacije lahko delimo na fiksne in spremenljive. Fiksni stroški so neodvisni od števila testov oziroma števila izvajanj testov. To so stroški opreme, orodij, usposabljanj itd. Spremenljivi stroški so odvisni od števila testov in pogostosti njihovih izvajanj. To so poleg drugega izdatki zaradi načrtovanja, izdelave, poganjanja in analize testov. Dejavnikov, ki vplivajo na izračun, je veliko, od teh je mnogo skupnih tako avtomatskemu kot ročnemu testiranju. Nekateri dejavniki so sicer nujni za uspešno vpeljavo avtomatizacije (kot npr. izdelava načrtov testov), vendar so enako koristni pri ročnem testiranju, zato jih lahko izpustimo.

Za izračun moramo določiti časovni okvir (**t**). Smiselno je vzeti naravno obdobje projekta, npr. med dvema kontrolnima točkama ali med dvema različicama programske opreme. Ker se koristi navadno pokažejo šele kasneje, je dobro izračune nekajkrat ponoviti.

Pri dejavnikih, kjer so stroški fiksni, je treba upoštevati njihovo amortizacijo, hkrati z življenjsko dobo in časovnim okvirom izračuna. Če npr. stane orodje 10.000 denarnih enot in pričakujemo, da bo uporabno pet let, potem nas to orodje v prvem letu stane 10.000/5, kar znese 2000 denarnih enot. Če je orodje uporabno samo prvo leto in ga moramo kasneje zamenjati, potem gredo vsi stroški v prvo leto. Podobno velja, če usposabljamo ljudi in ti zapustijo podjetje. V tem primeru gredo stroški njihovega usposabljanja v izgubo in jih ne moremo amortizirati.

Največja prednost avtomatskih testov je njihovo cenejše ponovno izvajanje. Pričakovano število poganjanj avtomatskih testov (**n1**) v primerjavi s pričakovanim število poganjanj ročnih testov (**n2**) sta zato najpomembnejša dejavnika izračuna. Pri tem moramo biti realistični. Kljub temu, da lahko avtomatske teste večkrat poženemo, je korist ponovnega zagona največja po spremembi testirane programske opreme (npr. ob dnevni izdaji - angl. *build*). Seveda lahko avtomatske teste brez dodatnih stroškov poženemo kolikokrat želimo tudi na isti različici, vendar bo korist nična.

Naslednji pomemben dejavnik je povezan z vzdrževanjem avtomatskih testov. Koliko krat lahko test poženemo preden ga moramo popraviti (**N**)? Izkaže se, da je pri uporabi pristopa PP (posnami-predvajaj) ta faktor blizu ena, kar zopet kaže na to, da so takšne skripte težavne za vzdrževanje.

Naslednji obrazec podaja izračun povračila naložbe (angl. *Return Of Investment*) na podlagi primerjav med ročnim in avtomatskim testiranjem [34]:

$$\boldsymbol{ROI}_{\boldsymbol{avtomatizacije}}(v\,\check{c}asu\,\boldsymbol{t}) = \frac{\Delta(Koristi\ avtomatizacije\ napram\ ročnemu\ testiranju)}{\Delta(Stroški\ avtomatizacije\ napram\ ročnemu\ testiranju)} = \frac{\Delta\,\boldsymbol{B}_a}{\Delta\,\boldsymbol{C}_a}$$

Pri čemer so vrednosti v obrazcu naslednje:

$$\begin{aligned}\Delta\,\boldsymbol{B}_a(v\,\check{c}asu\,\boldsymbol{t}) = &\sum(zmanjšani\ \ fiksni\ stroški\ zaradi\ avtomatizacije\,(\boldsymbol{t}/\boldsymbol{Uporabna\ življenjska\ doba}))\\ &+\sum(spremenljivi\ stroški\ \ poganjanja\ ročnih\ testov\ \boldsymbol{n}_2\ krat\ med\ časom\ \boldsymbol{t})\\ &-\sum(spremenljivi\ stroški\ \ poganjanja\ avtomatskih\ testov\ \boldsymbol{n}_1\ krat\ med\ časom\ \boldsymbol{t})\end{aligned}$$

$$\begin{aligned}\Delta\,\boldsymbol{C}_a(v\,\check{c}asu\,\boldsymbol{t}) = &\sum(\ povečani\ \ fiksni\ stroški\ zaradi\ avtomatizacije\,(\boldsymbol{t}/\boldsymbol{Uporabna\ življenjska\ doba}))\\ &+\sum(spremenljivi\ stroški\ izdelave\ avtomatskih\ testov)\\ &-\sum(spremenljivi\ stroški\ izdelave\ ročnih\ testov)\\ &+\sum(spremenljivi\ stroški\ vzdrževanja\ avtomatskih\ testov)(\boldsymbol{n}_1/\boldsymbol{N})\end{aligned}$$

V večini primerov lahko gledamo na povračilo naložbe avtomatizacije kot na dodano vrednost naložbe. Stroški ročnega testiranja v organizaciji so dani, avtomatizacija zato predstavlja samo dodaten strošek, od katerega pričakujemo dodatno povračilo. Torej koristi avtomatizacije ne računamo v absolutnem znesku, ampak v primerjavi z alternativo, t.j. ročnim testiranjem.

Kot primer si oglejmo avtomatizacijo testiranja prek uporabniškega vmesnika ob naslednjih predpostavkah:

- testiramo novo aplikacijo brez obstoječih testov,
- 5 človek-let za razvoj ročnih testov,
- 15 človek-let za razvoj avtomatskih testov,
- 1 človek za vzdrževanje po prvem letu za avtomatske teste,
- 10 ljudi za izvajanje ročnih testov, 1 človek za poganjanje avtomatskih,
- fiksni stroški orodij za avtomatske teste 90.000 DE (denarnih enot) z uporabno življenjsko dobo treh let,
- izbrano časovno obdobje za izračun (***t***): 12 mesecev (250 dni) in 24 mesecev (500 dni)
- cena zaposlencev 100.000 DE na leto = 400 DE na dan = 50 DE na uro

Na podlagi navedenih predpostavk izračunamo stroške, koristi in povračilo naložbe:

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

= 900.000 DE

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

= 1.800.000 DE

ΔC_a(v 12 mesecih) = (90.000 DE * (1/3)) + (15 ljudi * 100.000 DE) – (5 ljudi * 100.000 DE) +0 DE

= 30.000 DE+ 1.500.000 DE - 500.000 DE = 1.030.000 DE

ΔC_a(v 24 mesecih) = (90.000 DE * (2/3)) + (15 ljudi * 100.000 DE) – (5 ljudi * 100.000 DE) + 100.000 DE

= 60.000 DE + 1.500.000 DE - 500.000 DE + 100.000 DE = 1.160.000 DE

$ROI_{avtomatizacija}$(v 12 mesecih) = 900.000 DE / 1.030.000 DE = 0.874 (majhna izguba)

$ROI_{avtomatizacija}$(v 24 mesecih) = 1.800.000 DE/ 1.160.000 DE = 1.552 (55 % povračilo)

Čeprav je navedeni primer izmišljen, se rezultati ujemajo z izkušnjami v praksi. Avtomatizacija funkcionalnega testiranja prek uporabniškega vmesnika je težavna in bo težko povrnila stroške, če se je ne lotimo na pravi način. Vložek se navadno povrne v daljšem obdobju, navadno šele po projektu, ko je bila avtomatizacija uvedena. V obrazcu je zajeta tudi največja slabost pristopa posnami-predvajaj. To so visoki stroški vzdrževanja skript. Faktor **N,** ki nam pove, koliko krat lahko poženemo test preden ga moramo popraviti, je pri takšnem pristopu blizu številki ena, kar je seveda slabo.

7. Zaključek

Pojav agilnih metodologij in ekstremnega programiranja kot najvidnejšega predstavnika teh metodologij je nedvomno prinesel nekaj svežine k razvoju programske opreme. Agilnost je predvsem v smislu hitrega prilagajanja spremembam postala vrednota in doseganje agilnosti zaželen cilj tudi izven uporabe agilnih metodologij. Kljub temu, da ekstremno programiranje vsebuje že dolgo znane principe (npr. programiranje v parih), so jih avtorji metodologije prvič povezali v sinergijsko celoto. Naslednji zelo pomemben doprinos ekstremnega programiranja je avtomatizacija testiranja. Metode testiranja modulov so že dolgo znane, vendar je pri ročnem izvajanju tehnik vedno vprašljivo, ali se te dejansko izvajajo. Z izdelavo ogrodij za avtomatizacijo testiranja modulov in razširitvijo uporabe teh ogrodij se je verjetno zgodil najpomembnejši kakovostni preskok v testiranju v zadnjem desetletju [3]. Ali, če si sposodim besede enega kritikov ekstremnega programiranja: „Vsak proces, ki se dejansko izvaja, je zagotovo koristnejši kot popoln proces, ki ostaja na papirju“ [27].

Proces ekstremnega programiranja kot lepilo držijo skupaj avtomatski regresijski testi modulov. Brez njih bi bilo nemogoče izvajati aktivnosti kot so stalno preoblikovanje in integriranje. Zaradi tega je avtomatsko testiranje modulov temelj ali ključ do agilnosti in ne npr. pičla dokumentacija. Avtomatsko testiranje modulov seveda lahko uporabljamo tudi pri drugih razvojnih pristopih ne samo agilnih. Pri tem pa se je treba spopasti z nekaterimi omejitvami in težavami. Težko je pisati avtomatske teste modulov za obstoječo programsko kodo, če je testnost kode majhna. Če sistem ni zasnovan v plasteh, lahko težave povzroča povezava s podatkovno bazo. V teh primerih je včasih edina rešitev za avtomatsko regresno testiranje avtomatizacija funkcionalnega testiranja.

Avtomatizacija testiranja prek uporabniškega vmesnika je neodvisna od strukture kode, zato je primerna rešitev za uvedbo avtomatizacije testiranja na obstoječih vzdrževanih sistemih. Slabost te vrste avtomatizacije je visok začetni vložek in vrsta tveganj ob njenem uvajanju. Potrebna je naložba v draga orodja, usposabljanje in delo programerjev. Naložba se ob uspešni uvedbi avtomatizacije navadno povrne šele na drugem ali tretjem projektu. Zaradi tega je treba pretehtati stroške in koristi ter se uvedbe tovrstne avtomatizacije lotiti na projektni način. Eden možnih načinov uvedbe avtomatizacije je strukturna metodologija ATLM, ki sicer pokriva uvedbo avtomatizacije testiranja na vseh ravneh, ne samo funkcionalnega testiranja.

V tem trenutku sta torej omenjena načina avtomatizacije testiranja pot do povečanja agilnosti in lažjega obvladovanja entropije sistema (oziroma težnje pred razpadom). V bodoče lahko pričakujemo nadaljnji razvoj avtomatizacije testiranja in hkrati boljša ter cenejša orodja. Agilne metodologije bodo najbrž iskale svoje načine za poceni

avtomatizacijo sistemskega testiranja, ki ga sedaj nekoliko slabše pokrivajo. Na drugi strani se jim bodo morda proizvajalci dragih orodij za avtomatizacijo sistemskega testiranja približali z bolj programerjem prijaznimi orodji. Ta orodja so sicer primarno namenjena testerjem, katerih vloga pa pri agilnih metodologijah ni najbolj jasna. Razvoj metodologij za izdelavo programske opreme se bo seveda nadaljeval. Pri tem je ena od možnih poti, ki je videti dokaj logična, iskanje ravnovesja med disciplino planskih metodologij in lahkostjo agilnih [2].

8. Literatura

[1] Ming Huo in drugi (2004): “How does agility ensure quality?“, *2nd Workshop on Software Quality*, ICSE 2004, Edinburgh, Scotland, 2004, str. 36-40.

[2] Bary Boehm, Richard Turner (2003): “Observations on Balancing Discipline and Agility“, *Proceedings of the Agile Development Conference ADC '03*, , 2003, str. 32-39.

[3] Cem Kaner (2004): “The Ongoing Revolution in Software Testing“, *Software Test & Performance Conference*, Baltimore, 2004.

[4] Elfriede Dustin in drugi (1999), Automated Software Testing, Introduction, Management, and Performance, Addison-Wesley, 1999.

[5] Jonna Kalermo and Jenni Risanen (2002), Agile software developement in theory and practice, Software Business Program Master's thesis, University of Jiväskilä, Jiväskilä, 2002.

[6] Agile Alliance, http://www.agilealliance.org, <15.10.2006>

[7] Agile Manifesto, http://www.agilemanifesto.org, <15.10.2006>

[8] Franc Solina (1997), Projektno vodenje razvoja programske opreme, Fakulteta za računalništvo in informatiko, Ljubljana, 1997.

[9] Elfriede Dustin (2002), Effective Software Testing, Addison-Wesley, 2002.

[10] Henry Muccini (2002), Software Architecture for Testing, Coordination and Views Model Checking, Dottorato di Ricerca in Informatica, Universita degli Studi di Roma La Sapienza, 2002.

[11] Steve Cornett (2004), Code Coverage Analysis, Bullseye Testing Technology, 2004.

[12] Cem Kaner in drugi (2002), Lessons Learned in Software Testing, A Context-Driven Approach, John Wiley and Sons Inc., New York, 2002.

[13] Tuomas Pelkonen (2004), Using Automated Unit Testing to Improve Software Quality, Master's Thesis, Helsinki University of Technology, 2004.

[14] Ed Sullivan (2001), Under Pressure and On Time, Microsoft Press, Redmond, 2001.

[15] Winston Royce (1970), “Managing the Development of Large Software Systems“, Proc. Westcon, IEEE CS Press, 1970.

[16] Standish Group International, Inc.: Chaos Chronicals (1994 in 2004). Delno dostopno na http://www.standishgroup.com/sample_research/chaos_1994_1.php in http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf. <15.10.2006>

[17] Urs Kuhlmann (2004), Maintenance Activities in Software Process Models: Theory and Case Study Practice, Master's Thesis, University of Koblenz Landau, 2004.

[18] Anne Mette Jonassen Hass (2003), Configuration Management Principles and Practice, Addison-Wesley, 2003, pogl. 18.

[19] A. Spillner (2002): “The W-Model - Strengthening the Bond Between Development and Test“, *Proceedings of Software Testing Analysis & Review Conference*, STAReast, 2002, .

[20] Philippe Kruchten (2000), The Rational Unified Process - An Introduction, Addison-Wesley, 2000.

[21] Paul Szymkowiak (2003), Testing: The RUP Philosophy, Rational Software, 2003.

[22] Jim Heumann (2001): Generating Test Cases From Use Cases, Rational Software, 2001.

[23] Ali Khan (2004): “A tale of two methodologies for web development: Heavyweight versus agile“, *Tenth Australian World Wide Web Conference (AusWeb)*, 2004.

[24] Matt Stephens, Doug Rosenberg (2003), Extreme Programming Refactored: The case against XP, Apress, 2003.

[25] Ekstremno programiranje, http://www.extremeprogramming.org, <15.10.2006>.

[26] Barry Boehm, Richard Turner (2003): “Observations on Balancing Discipline and Agility“, *Proceedings of the Agile Development Conference ADC '03*, 2003, str. 32-39.

[27] Gerold Keefer (2003), Extreme Programming Considered Harmful for Reliable Software Development 2.0, AVOCA GmbH, 2003.

[28] Claus A. Christensen in drugi (2004), Unit Testing Database Applications, Department of Computer Science, Aalborg University, 2004.

[29] Matthias M. Muller, Oliver Hagner (2002): “Experiment about Test-first programming“, *IEEE Proceedings – Software*, October 2002, Volume 149, Issue 5, 2002, str. 131-136.

[30] Boby George, Laurie Williams (2003): “An Initial Investigation of Test Driven Developement in Industry“, *Proceedings of the 2003 ACM symposium on Applied computing*, ACM Press, New York, 2003, str. 1135 - 1139.

[31] Boby George, Laurie Williams (2004): “A structured experiment of test-driven developement“, *Information and Software Technology*, Volume 46, Issue 5, 2004, str. 337-342.

[32] Cem Kaner (1997): Improving the Maintainability of Automated Test Suites, Software QA, 1997.

[33] Stefan Berner in drugi (2005): “Observations and lessons learned from automated testing“, *Proceedings of the 27th international conference on Software engineering*, St. Louis, USA, 2005, str. 571 - 579.

[34] Douglas Hoffman (1999): “Cost Benefits Analysis of Test Automation“, *Software Testing*, Analysis, and Review (STAR) Conference, 1999.

Izjava

Izjavljam, da sem magistrsko delo izdelal samostojno pod vodstvom mentorja prof. dr. Franca Soline. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Peter Čebokli

V Izoli, december 2006