UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Mitja Lapajne

TESTNO VODEN RAZVOJ PROGRAMSKE OPREME V JAVI

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Saša Divjak

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

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

Št. naloge: 01604/2009
Datum: 15.10.2009

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

Kandidat: **MITJA LAPAJNE**

Naslov: **TESTNO VODEN RAZVOJ PROGRAMSKE OPREME V JAVI**
TEST DRIVEN SOFTWARE DEVELOPMENT IN JAVA

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V diplomski nalogi se posvetite testno vodenemu razvoju programske opreme. Opišite tehniko in metode, ki se pri tem najpogosteje uporabljajo. Predstavite razmere, ki so pripeljale do pojavitve testno vodenega razvoja Opišite splošno testiranje programske opreme, ki je temelj za testno voden pristop. Podajte teoretično osnovo, prednosti in omejitve. Podajte metode in orodja, ki se pri tem uporabljajo. Preizkusite testno voden razvoj v praksi na primeru kratke aplikacije.

Mentor:

prof. dr. Saša Divjak

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Mitja Lapajne,

z vpisno številko 63030311,

sem avtor/-ica diplomskega dela z naslovom:

TESTNO VODEN RAZVOJ PROGRAMSKE OPREME V JAVI

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek) prof. dr. Saša Divjak
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 12.4.2010 Podpis avtorja/-ice: _____

ZAHVALA

Zahvaljujem se :

- *prof. dr. Saši Divjaku za mentorstvo in pomoč pri izdelavi diplomskega*
- *bratu Klavdiju Lapajnetu za motivacijo in lektoriranje*
- *staršem, ki so mi stali ob strani in me podpirali v času študija*
- *zaposlenim za podjetju Marand d.o.o.*

Posebna zahvala gre še Aljažu Likarju, Juretu Jerebu, Nejcu Mivšku, Juretu Jeramu, Gašperju Žgavcu in Petru Vidmarju.

KAZALO

POVZETEK

Testno voden razvoj je tehnika razvoja programske opreme, ki temelji na ponavljanju zelo kratkih razvojnih ciklov. Razvijalec najprej napiše kratek test, ki se izvede neuspešno in definira neko novo funkcionalnost ali izboljšavo sistema. Nato napiše programsko kodo, ki zadovolji test in jo preoblikuje v sprejemljive standarde.

Diplomska naloga opisuje to tehniko in metode, ki se pri njej najpogosteje uporabljajo. Uvodno poglavje predstavi razmere, ki so pripeljale do pojavitve testno vodenega razvoja. Poleg tega predstavi diplomsko delo in nekatere raziskave, ki so preučevale uporabnost tega pristopa k programiranju. V nadaljevanju je opisano splošno testiranje programske opreme, ki je temelj za testno voden pristop. Teoretična osnova, prednosti in omejitve so opisane v poglavju Testno voden razvoj. V četrtem poglavju si ogledamo še metode in orodja, ki se pri tem uporabljajo. Testno voden razvoj smo preizkusili tudi v praksi. Napisali smo kratko aplikacijo s upoštevanjem korakov in pravil, ki jih ta pristop zapoveduje. V diplomski nalogi je tako na nekaj primerih predstavljen postopek razvoja in tehnike, ki pomagajo pri testno vodenem razvoju.

Ključne besede: testno voden razvoj, ekstremno programiranje, testiranje, agilne metode, Java, razvoj programske opreme

ABSTRACT

Test driven development is a software development technique, that relies on the repetition of a very short development cycle. First the developer writes a failing test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

This thesis describes this technique and methods, that are usually used. The first chapter describes the conditions that led to the appearance of test driven development. It also presents this thesis and some other works, that researched the usefulness of this programming approach. Next, the software testing, that is a base for test driven development, is described. The theory, advantages and limitations are described in chapter Testno voden razvoj. Fourth chapter covers methods and tools that help with development. We also tried test driven development in practise. We have written a simple application using the rules and techniques that this approach describes. In the thesis are included some practical examples from this application, that represent the development process and methods used by test driven development.

Keywords: test driven development, extreme programming, software testing, agile methods, Java, software development

1 UVOD

Razvoj programske opreme se neprestano spreminja in prilagaja. Nove tehnologije omogočajo inovativne pristope in večjo avtomatizacijo. Vse večja je potreba po dobrem testiranju, ki zagotavlja kvaliteto programske opreme. Glaven izziv pa predstavljajo zahteve naročnikov, ki se skozi čas spreminjajo. Namesto, da bi bila specifikacija fiksna in dobo definirana, se skozi čas (predvsem pri večjih projektih) preoblikuje in nadgrajuje, kar lahko predstavlja resen problem za razvijalce. Kot posledica vseh teh dejstev so se razvile različne metodologije razvijanja programske opreme.

Za nas je zanimivo ekstremno programiranje, ki vsebuje tudi tehniko testno vodenega razvoja. Ekstremno programiranje spada v širšo skupino agilnega razvoja, katerega glavne lastnosti so prilagodljivost, razbijanje projekta v manjše iteracije, ki zgradijo celotno aplikacijo in poudarek na ekipnem delu.

Poleg testno vodenega razvoja ekstremno programiranje vsebuje še programiranje v parih, obsežne preglede kode, težnjo k preprostosti in jasnosti kode, večjo komunikacijo med naročnikom in razvijalci in pričakovanje sprememb skozi čas.

Cilj tega dela je preučiti tehniko testno vodenega razvoja in oceniti njeno uporabnost. Primerjali jo bomo s standardnim postopkom razvoja in videli njene prednosti in pomanjkljivosti. Natančneje si bomo ogledali postopek razvoja, metode in orodja, ki se s tem ukvarjajo. Na koncu bomo to tehniko preizkusili tudi pri izdelavi preproste aplikacije in tako na primerih prikazali metode testno vodenega razvoja.

Mnenja razvijalcev o testno vodenem razvoju se zelo razlikujejo. Narejenih je bilo več raziskav, da bi ugotovili njegovo uporabnost [5, 13, 14]. Čeprav se rezultati razlikujejo so načeloma pozitivni.

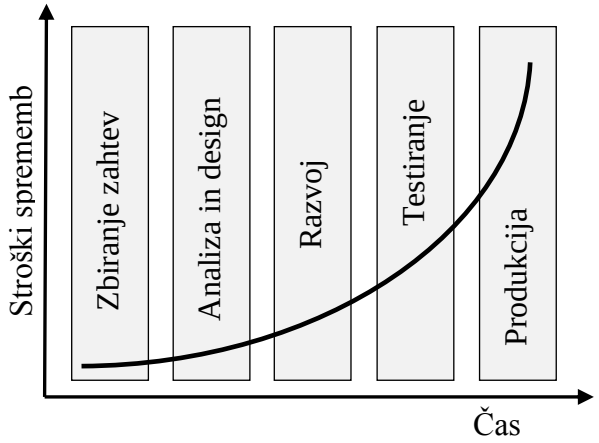
2 TESTIRANJE PROGRAMSKE OPREME

Testiranje programske opreme je proces izvajanja programa ali sistema z namenom iskanja napak. Lahko vsebuje tudi vsako aktivnost, usmerjeno v ocenjevanje zmogljivosti programske opreme ali njenih atributov. Gre za aktivnost, ki je tesno povezana s celotnim procesom razvoja programske opreme od samega začetka do zaključka projekta. Naloga testiranja je dokazati, da poslovna aplikacija ustreza zahtevam stranke in da hkrati odkrije čim več nepredvidljivih napak.

Učinkovito testiranje zahteva vpletenost stranke, končnih uporabnikov poslovne aplikacije, razvijalcev in za testiranje posebej usposobljenih testnih inženirjev.

Napake bodo vedno obstajale v vsakem večjem programskem modulu, saj je kompleksnost programske opreme prevelika za človeško razumevanje. Programa ne moremo nikoli popolnoma stestirati. Razlog za to je veliko število vhodnih in izhodnih podatkov in izvajalnih poti ter subjektivna specifikacija programske opreme. Testiranje programske opreme je stalno izbiranje kompromisov kaj testirati in kaj ne. **N**ikoli ne moremo preveriti vseh možnih scenarijev, lahko pa s pravilno zastavljenimi testi možnost napake v sistemu precej zmanjšamo. Če program deluje normalno pri robnih pogojih, lahko z veliko verjetnostjo trdimo, da bo deloval pravilno tudi pri normalnih pogojih.

Da odpravimo napake, moramo uvesti spremembe v programski opremi. Ker se stroški sprememb skozi čas povečujejo, moramo napake odkriti čim prej. Če jih odkrijemo v poznejših fazah razvojnega procesa, jih je težko odpraviti. Pri standardnem razvojnem procesu se testira relativno pozno, kar povzroča visoke stroške sprememb (slika 1).



Slika 1: Stroški sprememb skozi čas pri standardnem razvoju programske opreme [10]

2.1 Metode testiranja

Glede na način izvedbe ločimo metode testiranja na:

Statične metode

Programska oprema pri tem načinu testiranja ni uporabljena. Gre predvsem za preverjanje programske kode, dokumentacije in dokumentov. Večinoma se izvaja ročno, lahko pa tudi avtomatsko s preverjanjem programske sintakse ali skladnosti kode s postavljenimi standardi. Statične metode se uporabljajo za preprečevanje napak.

Dinamične metode

Gre za izvajanje programa ali njegovih delov s pomočjo testnih primerov. Namen je preveriti pravilnost odziva sistema na spremenljivke, ki niso konstantne in se skozi čas spreminjajo. Testi se lahko izvajajo ročno, vse pogosteje pa razvijalci uporabljajo posebno programsko opremo za avtomatizirano testiranje. Dinamične metode se uporabljajo za odpravljanje napak.

Glede na pristop k testiranju jih delimo testiranje na:

Metode bele skrinjice ali strukturno testiranje

Večinoma testiramo posamezne module z vnaprej definiranimi testnimi primeri, ki temeljijo na poznavanju strukture programske kode. Cilj testiranja po metodah bele skrinjice je pokriti čim več kode z realnimi testnimi primeri. Programska koda, ki se nikoli ne izvaja, je morda nepotrebna in jo lahko odstranimo. Za to testiranje je potreben dostop do kode, zato ga navadno uporabljajo programerji že med, ali v našem primeru celo pred, razvojem posameznih modulov. Strukturno testiranje je stabilno in odporno, a tudi kompliciran. Glavna prednost je hitro lociranje problema. Za razliko od metode črne skrinjice nam pove točno v kateremu modulu je prišlo do napake.

Metode črne skrinjice ali funkcionalno testiranje

Programska koda in design programa sta pri teh metodah izvajalcu testov skrita. Cilj testiranja po metodah črne skrinjice je preveriti, ali se sistem obnaša v skladu s specificiranimi strankinimi zahtevami. Največkrat se uporablja za testiranje preko uporabniškega vmesnika. Zanimajo nas le izhodi pri določenih vhodih. Ne obremenjujemo se z notranjo zgradbo in operacijami, ki so se izvršile, da smo rezultat dobili. Z tem poskušamo odkriti napake in pomanjkljivosti v delovanju programa, težave s uporabniškim vmesnikom ter probleme z

zmogljivostjo ali varnostjo. Funkcionalno testiranje je lažje in bolj preprosto od strukturnega, je pa tudi manj poglobljeno in težje za vzdrževanje. Uporabniški vmesniki se namreč pogosto spreminjajo in niso vedno enaki na vseh sistemih in platformah [5].

2.2 Vrste testiranja

Testiranje enot

Princip testiranja enot (ang. *Unit testing*) obstaja že dolgo, vendar je prišel v ospredje šele s pojavom modernejših metodologij in orodij, ki omogočajo avtomatizacijo testiranja enot. Enota je najmanjši zaokrožen del programskega sistema, ki se lahko testira in navadno nastane kot delo enega razvijalca.

Ker testno voden razvoj že v osnovi temelji na testih enot, bo to področje podrobneje predstavljeno v nadaljevanju.

Integracijsko testiranje

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.

Testiranje integracije je morda eden najbolj kritičnih trenutkov v razvoju, ker se takrat pokažejo 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 je to vidno ob slabo izvedeni fazi načrtovanja pri slapovnem razvojnem modelu. 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 postopoma kliče teste modulov. Pred tem morajo biti odpravljene vse napake, odkrite med testiranjem modulov [5].

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. Odveč je pripomniti, da se ob tem bistveno povečajo stroški izdelave sistema in podaljša čas do predaje izdelka stranki. Obstajajo različne definicije o tem, kaj vse sodi v sistemsko testiranje. Spodaj so navedene najbolj znane metode.

- Testiranje varnosti sistema: Preverjamo zaščito sistema pred nepooblaščenim notranjim in zunanjim dostopom in namernim povzročanjem škode.

- Testiranje obremenitve: Imenovano je tudi testiranje zmogljivosti sistema. Sistem obremenjujemo z vse večjo količino transakcij, da vidimo katerih virov mu najprej zmanjka. Tako lahko določimo maksimalno obremenitev sistema. Če se izkaže, da je zmogljivost sistema nezadovoljiva, je treba zamenjati strojno opremo ali optimizirati programsko kodo.

- Testiranje uporabnosti: Gre za dokaj subjektivno testiranje. V poštev pridejo tehnike, kot so intervjuji z uporabniki, preučevanje njihovih navad, različne raziskave in preverjanje skladnosti sistema s smernicami, ki so že bile določene. Za to vrsto testiranja razvijalci in testni inženirji navadno niso primerni.

- Testiranje okrevanja: Lahko pride do izpada sistema zaradi okvare strojne opreme, napak v programski opremi, nedelovanje internetne povezave, izpad električne energije, … Testiramo v kakšnem času in kako uspešno se sistem povrne v delujoče stanje.

- Testiranje primerljivosti: Programsko opremo primerjamo z programsko opremo tekmecev ali katero drugo referenčno programsko opremo. Ocenimo prednosti, slabosti in možne izboljšave.

- Alfa testiranje: Testira se proti koncu razvojnega cikla, ko je večina funkcionalnosti že podprte. Tipično testirajo alfa izdelek testni inženirji programske hiše oziroma izbrani zunanji končni uporabniki.

- Beta testiranje: Je testiranje po koncu razvoja programske opreme, ko je bilo sistemsko 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.

Prevzemno testiranje

S prevzemnim testiranjem 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 oziroma enakem produkcijskemu okolju. Ker se funkcionalno in prevzemno testiranje prekrivata, večinoma uporabimo enake teste, predvsem pozitivne scenarije. 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.

Regresijsko testiranje

Regresijsko testiranje je 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 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 (ang. *Smoke testing*). To je zbirka regresijskih testov, ki jih poženemo vsakodnevno na trenutni različici poslovne aplikacije. Če se "dim" ne pokaže, je to zagotovilo, da trenutna različica programske opreme kljub spremembam v najslabšem primeru ohranja kvaliteto prejšnje. 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 tej vrsti testiranja, 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.

2.3 Testiranje enot

Testi enot so namenjeni testiranju majhnih delov produkta, idealno samo ene enote na test. Od posameznega produkta je odvisno, kaj ena enota pomeni: lahko je cela funkcionalnost, razred, metoda, ali le ena vrstica. Teste enot najpogosteje napišejo isti razvijalci, kot so napisali izvorno kodo, ki jo testi preverjajo.

Cilj testiranja enot je izolirati vsak del programa in pokazati, da so vsi posamezni deli pravilni. Test enote prinaša veliko prednosti, saj točno določa pogoj, ki ga mora del kode zadovoljiti. Razvijalec lahko kasneje preoblikuje kodo in se s testi prepriča, da se želene funkcionalnosti posameznih enot niso spremenile, kar tudi poveča njegovo zaupanje v projekt.

Idealno je vsak testni primer neodvisen od ostalih. Da dosežemo to neodvisnost, si lahko pomagamo z navideznimi metodami (ang. stubs) in navideznimi objekti (ang. mock objects). Testi enot morajo biti tudi neodvisni od dosegljivosti in podatkov pridobljenih z omrežij in podatkovnih baz.

Testi enot priskrbijo tudi dokumentacijo, ki se razvija vzporedno s sistemom. Razvijalec lahko pridobi osnovno razumevanje o določeni enoti in njeni uporabi, če pogleda njene teste enot. V njih so opisane poglavitne karakteristike te enote. To so poleg rezultatov, ki jih pod določenimi pogoji vrača, lahko tudi primerna/neprimerna uporaba in izjeme, ki jih vrača v primeru napak.

Testiranje enot je tesno povezano s strukturo programske kode, zato se mora programer potruditi, da piše takšno kodo, ki jo je lahko testirati. Tega problema pri testno vodenem razvoju ne opazimo, saj taka koda izvira že iz samega pristopa. Kot je opisano v nadaljevanju, so prav testi enot osnova za testno voden razvoj, saj nam natančno določajo, kaj mora posamezna enota delati, še preden obstaja.

3 TESTNO VODEN RAZVOJ

Testno voden razvoj (ang. Test driven development ali Test first development) ni le pristop k kodiranju programskih sistemov. Je tudi proces, ki zagotavlja kvaliteto in vodi razvoj programske opreme k večji preprostosti in lažjem testiranju.

Naslednje točke opišejo vidike testno vodenega razvoja :

- Preden napišemo vrstico produkcijske kode, moramo napisati test, ki se bo neuspešno prevedel/izvedel. Ta test bo narekoval kodo, ki jo moramo napisati.

- Napišemo le toliko kode, da se test čim hitreje uspešno prevede in izvede.

- Brž ko koda deluje, jo preoblikujemo (ang. refactor), da postane čim bolj »čista«. To vključuje predvsem odstranjevanje duplikatov.

- Proces razvoja poteka v majhnih korakih, kje se izmenjujeta testiranje in kodiranje. Take mikro-iteracije ne smejo trajati več kot 10 minut.

- Ko integriramo produkcijsko kodo v celotni sistem, se morajo vsi testi enot uspešno izvesti [9].

Je del ekstremnega programiranja (ang. Extreme programming), ki je metodologija razvoja programske opreme, namenjena manjšim projektnim skupinam, ki programsko podpirajo poslovne procese v okolju, kjer potrebe niso vnaprej natančno določene in/ali se nenehno spreminjajo.

3.1 Koraki testno vodenega razvoja

Kot je bilo že zapisano, je testno voden razvoj tehnika programiranja. Temelji na preprostem pravilu:

»Napiši le toliko kode, da popraviš neuspešen test [8].«

Najprej torej napišemo test in nato kodo, ki bo omogočila da se test uspešno izvede. To se najbrž zdi nenavadno vsem, ki so navajeni navadnega razvoja. Tam se najprej zastavi design,

ki se ga implementira s kodo, na koncu pa se s testi preveri programsko opremo, da odkrijemo napake, ki so nastale med razvojem (slika 2). Razvojni cikel se pri testni vodenem razvoju tako ravno obrne (slika 3) [8].

Slika 2: Navadni razvojni cikel

Slika 3: Razvojni cikel pri testno vodenem razvoju

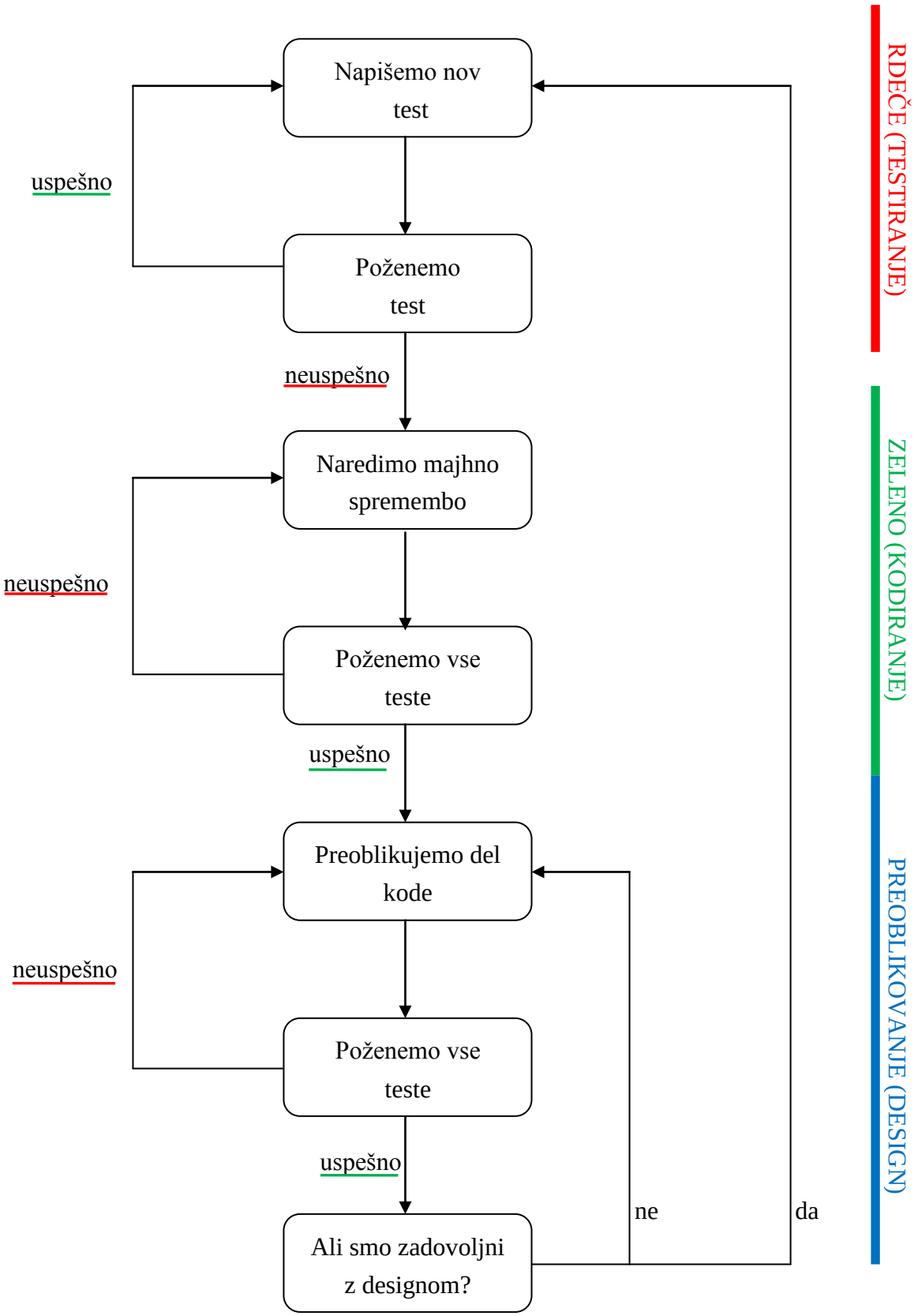
Namesto designa imamo v zadnjem koraku preoblikovanje. Z njim spreminjamo in izboljšujemo design, ki je nastal med kodiranjem. Naknadno spreminjanje kode za doseganje lepega designa se morda zdi kompliciran in zamudno v primerjavi s klasičnim pristopom. Vendar veliko novejših okolij za razvoj programske opreme vsebuje orodja za preoblikovanje, design pa je navadno preprostejši in lepši. Ker je preoblikovanje pomemben del testno vodenega razvoja, si ga bomo podrobneje ogledali v nadaljevanju.

Pogosto se pojavlja tudi alternativno poimenovanje razvojnega cikla pri testno vodenem razvoju: rdeče-zeleno-preoblikovanje (ang. red-green-refactor). Ime izvira iz barve, ki se prikaže po zagonu testov. Večina razvojnih okolij v primeru padlih testov prikaže rdeč znak, v primeru uspeha pa zelenega. Ko napišemo nov test dobimo rdečo barvo. To ne pomeni, da je v kodi napaka, ampak da želena funkcionalnost še ni podprta. Ko jo v drugem koraku implementiramo nam programsko okolje vrne zeleno (test se je pravilno izvedel). V zadnjem koraku izvedemo še preoblikovanje, da popravimo design.

Potek mikro-iteracije

Razvijamo torej v mikro-iteracijah, ki vključujejo testiranje, kodiranje in preoblikovanje. V vsaki iteraciji preverimo in po potrebi dodamo neko novo funkcionalnost ali pa le zadovoljimo nekemu pogoju, ki je določen v testu (na primer preverjanje izjem). Vsaka iteracija nas pripelje bližje končnemu produktu.

Potek mikro-iteracije je prikazan na sliki 4. Na začetku seveda napišemo test in ga zaženemo. Če se uspešno izvede, nam ni treba nič kodirati. Ker je želen rezultat že dosežen, lahko nadaljujemo z naslednjim testom. V nasprotnem primeru naredimo majhno spremembo v kodi. V tem koraku se na design ne oziramo; cilj je le, da se test uspešno izvede. Po vsaki spremembi poženemo vse teste in ne samo trenutnega. S popravki smo namreč lahko pokvarili katero od ostalih funkcionalnosti. Tako je v vsakem trenutku stestiran celoten sistem. Šele ko se uspešno izvedejo vsi testi, lahko nadaljujemo na preoblikovanje. Po vsaki spremembi spet zaženemo vse teste, saj sprememba designa lahko zelo vpliva na ostale module. Ko smo zadovoljni s kodo in se vsi testi izvedejo zeleno, nadaljujemo z novim testom.



Slika 4: Potek mikro-iteracije

3.2 Preoblikovanje

Pri testno vodenem razvoju napredujemo v zelo majhnih korakih (mikro-iteracijah), kar pomeni da konstantno razširjamo sistem z novimi funkcionalnostmi, ki jih trenutni design morda ne podpira. Spremembe torej lahko podrejo ali zelo spremenijo obstoječi design. Postane lahko nekonsistenten, kompliciran ter zahteven za razumevanje in spreminjanje. Rešitev za te težave je preoblikovanje (ang. refactoring).

Preoblikovanje je proces spreminjanja kode. Cilj je izboljšati notranjo strukturo kode, brez da bi spremenili njeno funkcionalnost. Preoblikovanje je v bistvu čiščenje slabe kode. Razvijalec, ki želi začeti s testno vodenim razvojem mora prej dobro poznati preoblikovanje, saj je eden izmed njegovih poglavitnih delov.

Pogosti razlogi za preoblikovanje:

- Neprimerno poimenovanje
- Nekonsistentna imena
- Podvojena koda
- Enaka funkcionalnost, drugačen algoritem
- Predolga metoda
- Prevelik razred
- Odvečne spremenljivke
- Predolg nabor parametrov

Nekatere izmed tehnik za preoblikovanje:

- Preimenovanje – med razvojem lahko razredi in metode dobijo drugačen pomen ali funkcionalnost
- Izloči metodo (ang. extract method) – del kode prenesemo v novo metodo
- Izloči razred (ang. extract class) – del kode prenesemo v nov razred

- Izloči vmesnik (ang. extract interface) – naredi nov vmesnik za obstoječi razred
- Inkapsulacija (ang. encapsulate) – polje postane privatno, kreirajo se metode za dostop
- Generalizacija tipa – namesto podrazredov uporabimo bolj splošne razrede in s tem izkoristimo prednosti objektno orientiranega programiranja
- Premikanje metode – metodo premaknemo v podrazred ali nadrazred

Glede na to, da ga moramo izvajati skoraj v vsakem koraku testno vodenega razvoja, bi nam ročno preoblikovanje pobralo preveč časa. Zato se uporabljajo orodja za preoblikovanje, ki jih ima večina naprednih razvojnih okolij. Že s par kliki lahko izvedemo večino sprememb, ki so opisane zgoraj.

Brez skrbi lahko izvajamo vse vrste preoblikovanj, saj po vsaki spremembi poženemo teste enot in takoj vidimo, če se je pokvarila katera izmed funkcionalnosti. Cikel popravkov in izvajanja testov ponavljamo, dokler nismo zadovoljni z programsko kodo.

3.3 Prednosti

- Vsak posamezen del kode je stestiran. Če s spremembami pokvarimo obstoječe funkcionalnosti, to opazimo takoj preko testov. To je še posebej pomembno v poznejših fazah razvojnega cikla, saj lahko služi za neprestano regresijsko testiranje (ang. continuous regression testing).
- Testi dokumentirajo kodo, saj kažejo normalno uporabo in pričakovan odziv v primeru izjeme.
- Mikro-iteracije so zelo kratke, kar pomeni hiter odziv. V 10 minutah ne moremo sprogramirati veliko kode, zato ne moremo narediti veliko napak in jih lahko hitro odpravimo.
- Testi določajo design programa. To v večini primerov vodi do bolj preprostejše kode, saj je za kompliciranо strukturo težko pisati teste. Zato se testno voden razvoj šteje bolj za tehniko programiranja, kot pa za tehniko testiranja.
- Večja enkapsulacija in modalnost

- Ni odvečne kode. Vsa koda, ki jo napišemo, je posledica testov, kateri predstavljajo funkcionalnosti.

- Že od začetka imamo delujoč sistem, ki mu postopoma dodajamo funkcionalnosti. V vsaki fazi bo sistem deloval (v omejenem obsegu) in bo stestiran.

- Bolj osredotočeno delo razvijalcev. Pozornost zmeraj usmerijo le v trenutni test in kodo, ki k njemu spada [9].

3.4 Omejitve

- Testno voden razvoj je namenjen izkušenim razvijalcem, saj zahteva visoko stopnjo znanja in razumevanja programskega jezika in okolja. Ravno izkušeni razvijalci pa so navadno že preveč navajeni na navadno tehniko razvoja. Zato je za njih glavna težava mentalni preskok v pisanje testov pred kodo. Že med pisanjem testov si morajo namreč zamisliti, kakšna bo programska koda. Pri prehodu na testno voden razvoj pogosto za isto nalogo porabijo več časa kot navadno.

- V sistemih, kjer so za ugotavljanje pravilnega delovanja potrebni obsežnejši testi funkcionalnosti, je uporaba testno vodenega razvoja lahko kompliciran. Tukaj gre predvsem za uporabniške vmesnike in za aplikacije, ki delujejo v povezavi s podatkovnimi bazami ali z omrežjem. Testno voden razvoj spodbuja razvijalce, da v te module dajo čim manj kode in logiko raje implementirajo drugje. Pri tem zunanji svet predstavijo z navideznimi objekti, metodami in vmesniki.

- Celotna organizacija mora verjeti, da bo testno voden razvoj izboljšal produkt. Vodstvu se bo pisanje testov drugače lahko zdela izguba časa.

- Testi postanejo del vzdrževanja projekta. Slabo napisani testi, ki na primer vsebujejo trdo kodirane nize izjem, so težki in dragi za vzdrževanje. Obstaja tveganje, da se teste, ki stalno padajo zaradi slabega vzdrževanja, začne ignorirati. Ko pride do dejanske napake, je tako ne odkrijemo.

- Z ponavljajočimi cikli testno vodenega razvoja postopoma dosežemo veliko natančnost testiranja in stopnjo pokritosti kode. To pa je težko doseči kasneje za nazaj. Slaba arhitektura, pomanjkljivo testiranje ali sprememba specifikacije lahko v poznejših fazah razvoja zahteva večje popravke kode, ki povzročijo, da se veliko

testov izvede neuspešno. V tem primeru moramo popraviti vse posamezne teste, da se izvedejo pravilno. Če pri tem nismo natančni ali jih preprosto le zbrišemo, lahko nastanejo luknje v pokritosti s testi in nezaznane napake. Popravljanje testov za nazaj je navadno težavno in dolgotrajno.

- Vsi razvijalci morajo dosledno slediti pravilom testno vodenega razvoja. Če eden ali več izmed njih piše pomanjkljive teste se lahko pojavijo luknje v pokritosti s testi in zaupanje v delovanje sistema pade. Vsi morajo tudi sproti preoblikovati grdo kodo. Če s tem odlašajo lahko povzročijo veliko težav kasneje.

- Teste pišejo isti razvijalci, kot pišejo programsko kodo. Vsak modul torej vidijo le z enega zornega kota in lahko spregledajo kakšno napako, ki bi jo tester odkril. Lahko celo napačno razumejo specifikacijo za posamezen modul, kar se bo odražalo v testu in v kodi. Čeprav se bodo vsi testi izvedli pravilno, bo sistem deloval napačno, kar bo težko odkriti.

- Veliko število uspešno izvedenih testov in velika pokritost kode lahko prinese lažen občutek varnosti in preveliko zaupanje v projekt. Obvezno je še nadaljnje testiranje, ki zagotovi skladnost s specifikacijo in delovanje obsežnejših operacij, na primer integracijsko in sistemsko testiranje [12].

4 TEHNIKE IN ORODJA ZA TESTNO VODEN RAZVOJ V JAVI

Teoretično bi testno voden razvoj lahko uporabili skoraj z vsakim programskim jezikom. Najbolj pa se je uveljavil v objektno usmerjenih jezikih, še posebej v Javi. Pri razvoju si pomagamo z različnimi orodji in vzorci.

4.1 Vzorci za testno vodenem razvoju

Vzorci za testno voden razvoj (ang. Test driven development patterns) nam služijo kot pomoč pri pisanju testov. Pomagajo nam določiti obliko testa in olajšajo pisanje kode.

Podtest (ang. Child test)

Včasih napišemo test, za katerega ugotovimo, da bi potrebovali veliko časa in produkcijske kode, da bi ga rešili. Ker to ni v skladu z načeli testno vodenega razvoja ga moramo popraviti. Očitna rešitev je, da ga zbrišemo in napišemo več testov, ki predstavljajo isto funkcionalnost. Namesto tega lahko napišemo manjši podtest, ki predstavlja en nedelujoč del celotnega testa. Poskrbimo, da se podtest uspešno izvede in se vrnemo k osnovnemu testu, ki ga zdaj lažje rešimo.

Navidezni objekti (ang. Mock Object)

Kako naj testiramo objekt, ki je odvisen od kompliciranega ali negotovega vira? Naredimo navidezno verzijo tega vira, ki nam vrača le konstante, ki jih potrebujemo za testiranje.

Tipičen primer so podatkovne baze. So počasne, podatki v njih pa se lahko brez naše vednosti spreminjajo. Če se nahajajo na oddaljenem strežniku, pa vežejo teste na fizično lokacijo v omrežju. So pogost vir napak v razvoju.

Ta problem lahko rešimo tako, da v testiranju sploh ne uporabljamo podatkovnih baz. Nadomestimo jih z navideznimi objekti, ki se obnašajo kot prave baze. Za točno določen vhod (v tem primeru vprašanje SQL) nam vrne fiksno določen izhod.

Navidezni objekti so dobri zaradi zanesljivosti in hitrosti, moramo pa se prepričati da se dejansko obnašajo kot pravi objekti. Lahko napišemo teste, ki se izvajajo na navideznih in pravih objektih, ko so ti na voljo.

Self Shunt

Izraz izhaja iz elektrotehnike in predstavlja situacijo, ko je eden izmed izhodov povezan nazaj vhod.

Ta vzorec se uporablja za testiranje komunikacije med dvema objektoma. Enega izmed objektov implementiramo kar z našim testom. Tako bo komunikacijo potekala med objektom in testom, kar nam omogoča dostop do informacij, ki jih je objekt poslal.

Crash Test Dummy

Testno voden razvoj nam pravi, da koda, ki ni stestirana, ne dela. Če želimo biti prepričani, da se bo naš sistem obnašal pravilno tudi v kompliciranih, malo verjetnih situacija, moramo napisati teste tudi za te primere.

Recimo, da bi radi videli kaj se zgodi, ko v datotečni sistem, ki je poln poskušamo zapisati novo datoteko. Nesmiselno je pisati test, ki nam bo zapolnil ves prazen prostor, samo da bo testiral to malo verjetno situacijo. Lahko pa jo simuliramo. Java nam omogoča, da kar v testu redefiniramo metodo kreiranja nove datoteke. Tako lahko v njej prožimo izjemo in testiramo odziv sistema.

Fake It ('Til You Make It)

Gre za vzorec, kjer test spravimo v delujoče stanje z vračanjem konstant. Tako hitro pridemo do zelenih rezultatov. Nato postopno konstante zamenjamo z spremenljivkami, ki nam vračajo pravilne rezultate.

Te navidezne implementacije nam služijo kot pomoč pri predstavljanju strukture. Iz njih lahko vidimo, če je naš test pravilno napisan. Lažje tudi napišemo pravo implementacijo, če imamo prej določene konstante.

Triangulacija

Rešitev je pri vzorcu triangulacije precej bolj očitna kot pri vzorcu Fake it. Tam moramo sami implementirati rešitev, ki bo rešila naš test, zraven pa vračala pravilne rezultate tudi v vseh ostalih primerih. Pri večini metod to ni problem, pri bolj zapletenih pa je pravo rešitev lažje dobiti s triangulacijo.

Prvi test zadovoljimo na čim bolj preprost način (naprimer z vračanjem konstant). Nato za isto metodo napišemo še en test z drugimi parametri. To nas prisili, da naredimo rešitev bolj splošno in tako vodi abstrakcijo. Lahko nam pomaga tudi pri gradnji algoritmov.

4.2 Pokritost kode s testi

Pokritost kode (ang. Code coverage) je merilo, ki se uporablja v testiranju programske opreme. Opiše kakšne del produkcijske kode je bilo testirane. Ker se izvaja neposredno na kodi, gre za metodo bele škatlice. Pri avtomatiziranem testiranju predstavlja procent kode, ki jo vsi testi, ki se izvajajo, pokrijejo.

Teoretično bi s testno vodenim razvojem lahko dosegli 100% pokritost kode, vendar to v kompleksnejših projektih ni mogoče. Tipična pokritost v večjih projektih je okoli 85%. Razlog za odstopanje leži v API-jih, kjer je določene situacije nesmiselno testirati, saj se ne morejo zgoditi.

Pomembno pa je, da testi pokrijejo vse pogoje, ki se lahko pojavijo v produkciji. To nam bi načeloma morala zagotavljati že specifikacija, iz katere testi izvirajo. Večkrat pa se zgodi, da so specificirani le najbolj pogosti ali le pravilni primeri. Paziti moramo, da napišemo teste tudi za te primere.

Koda, ki bo nastala pri razvoju projekta, bo imela več različnih poti izvajanja. Pokritost kode lahko izboljšamo tako, da napišemo toliko testov, da se vsaka pot izvede vsaj enkrat. Takšen primer je recimo stavek if-else. Napisati moramo torej dva testa, da pokrijemo vso kodo.

Testno voden razvoj nam teoretično že zagotavlja pokritost vseh poti, saj vsako pot dodamo le kot posledico testa in je tako že stestirana. Vsi razvijalci v ekipi pa se ne držijo dosledno pravil testno vodenega razvoja. Pokritost kode je tako dober pokazatelj, če je kakšna pot brez testa. Če odkrijemo kakšne nepokrit del je dobro raziskati zakaj nimamo testa in zakaj ta del sploh obstaja.

Orodja za merjenje pokritosti kode spremljajo izvajanje testov in preverjajo, katera vrstica kode je bila izvedena vsaj enkrat. V kombinaciji z avtomatskim testiranjem dobimo informacijo o pokritosti v celem projektu. Napredna razvojna okolja imajo že vgrajena orodja za ta namen.

4.3 Orodja za testno voden razvoj

Brez naslednjih orodij bi bil testno voden razvoj v Javi skoraj nemogoč.

Integrirano razvojno okolje (IDE), ki podpira preoblikovanje

Programsko kodo v Javi lahko pišemo z vsakim urejevalnikom teksta ali namenskim urejevalnikom programske kode. Za vse večje projekte, ki vsebujejo veliko kode pa potrebujemo integrirano razvojno okolje. Poleg ostalih prednosti, ki jih prinese so za testno voden razvoj poglavitna orodja za preoblikovanje. Brez njih bi bil proces preveč dolgotrajen. Namesto da bi popravke izvajali sami s pisanjem kode, nam omogočajo, da s par kliki naredimo vse tipična preoblikovanja.

Najbolj popularna integrirana razvojna okolja za Javo so:

- JDeveloper: http://technet.oracle.com/software/products/jdev/content.html
- IDEA: http://www.intellij.com/idea
- Eclipse: http://www.eclipse.org

JUnit

JUnit je odprtokodno ogrodje za pisanje in poganjanje testov enot v Javi. Je del širšega ogrodja xUnit. Omogoča nam avtomatizacijo testov enot, ki jih lahko poganjamo med razvojem. Brez JUnit testno voden razvoj v Javi ne bi obstajal.

Dobre lastnosti JUnit so:

- Primerjave (assertions) za testiranje pričakovanjih rezultatov
- Testi si lahko delijo skupne podatke (fixtures, naprimer metoda setUp())
- Organiziranje in zaganjanje testov je preprosto (TestSuits)
- Grafični prikaz rezultatov

Zaradi široke razširjenosti obstaja tudi veliko razširitev, ki zelo povečajo uporabno vrednost JUnit.

Orodje za navidezne objekte

Potrebujemo knjižnice, ki nam omogočajo delo z navideznimi objekti. Najpogosteje uporabljene so:

- EasyMock: http://easymock.org
- Mockio: http://mockito.org/
- MockLib: http://mocklib.sourceforge.net/
- jMock: http://www.jmock.org/

Za testno voden razvoj je najbolj primeren EasyMock, saj je odporen na preoblikovanje.

Orodja za testiranje uporabniških vmesnikov

Testiranje uporabniških vmesnikov je lahko precej kompleksno. Lahko je testirati podatke, ki jih posamezne metode vračajo, težje pa je preverjati, če se na zaslonu prikazujejo prave stvari. Zato potrebujemo posebna orodja (navadno so to razširitve za JUnit), ki znajo ravnati z prikazanimi objekti. Omogočati morajo, da s pomočjo testov vpisujemo polja, pritiskamo gumbe, preverjamo vsebino polj, … Nekatera orodja posnamejo akcije v uporabniškem vmesniku, v drugih pa jih moramo definirati sami s kodo.

Nekaj orodij za testiranje uporabniškega vmesnika:

- SWTBot: http://swtbot.sourceforge.net
- Jacareto: http://jacareto.sourceforge.net/
- Jemmy: Module: https://jemmy.dev.java.net/

Orodja za pokritost kode s testi

Ta orodja nam prikažejo pokritost kode za posamezne metode, razrede, pakete ali za celoten projekt. Pomagajo nam odkriti dele, ki niso stestirani. Večina jih poleg tega izvaja tudi analizo kompleksnosti kode. Tako lažje vidimo katero kodo moramo preoblikovati.

Nekaj orodij za pokritost kode s testi:

- Emma: http://emma.sourceforge.net/
- CodeCoverage: http://www.codecover.org/
- Cobertura: http://cobertura.sourceforge.net/
- Jester: http://jester.sourceforge.net/

5 PRIMER UPORABE TESTNO VODENEGA RAZVOJA

Testno voden razvoj se najlažje razloži na primeru, zato sem to tehniko programiranja tudi sam uporabil pri razvoju preproste aplikacije. S pomočjo programske kode, ki je pri tem nastala, bom v tem poglavju predstavil določene lastnosti in metode, ki sem jih prej opisal le teoretično.

Aplikacija je namenjena študentom. Z njeno pomočjo bi lahko pregledovali opravljene in preostale izpite. Lahko bi si planirali prihodnje izpitne roke in računali statistiko. Bolj kot na samo aplikacijo sem se osredotočil na uporabo testno vodenega razvoja. Opisal bom le tiste teste in metode, ki so zanimivi in predstavljajo različne metode in pristope, ki sem jih opisal prej.

5.1 Kako začeti

Razvoj se vedno začne s čim bolj natančno specifikacijo funkcionalnosti, ki jih mora končni produkt podpirati. Po pravilih ekstremnega programiranja se za specifikacijo uporabljajo uporabniške zgodbe (ang. user stories), ki s stališča uporabnika opišejo procese in funkcionalnosti. Ponavadi se te zgodbe pretvorijo v naloge (ang. tasks), ki določajo kaj je treba narediti. Tako se specifikacija iz nivoja uporabnika prenese na nivo razvijalca. V našem primeru pa bomo naloge nadomestili s testi. Vsako nalogo namreč lahko prevedemo v enega ali več testov, ki nam ne določajo le kaj moramo narediti, ampak tudi točno, kdaj je stvar narejena. Ti testi morajo biti kratki in izolirani.

Primer uporabniške zgodbe za filter po letniku:

- Uporabnik želi pogledati samo izpite drugega letnika

Primer naloge za filter po letniku:

- Naredi metodo, ki vrača le izpite določenega letnika

Primer testa za filter po letniku:

- Preveri, če metoda za iskanje po letniku vrača pravilne rezultate

Testi so lahko manj opisni, nam pa točno določajo rezultate, ki jih mora metoda vrniti. Ko se testi pravilno izvedejo smo lahko prepričani, da funkcionalnost deluje. Paziti pa moramo, da med pisanjem dejanskih testov pokrijemo čim več možnosti, ki bi lahko povzročile napake.

Dejanski testi za filter po letniku :

- Preveri če metoda s parametrom drugi letnik iz seznama izpitov ARS1 (2. letnik), PSS (1. letnik) in APS1 (2. letnik), vrača izpita ARS1 in APS1.

- Preveri če metoda s parametrom tretji letnik iz seznama izpitov ARS1 (2. letnik) in PSS (1. letnik) vrača prazen seznam.

- Preveri če metoda s parametrom prvi letnik iz praznega seznama izpitov vrača prazen seznam.

Napišemo si okviren seznam stvari, ki jih moramo s našimi testi preveriti. Dejanskih testov bo več, saj lahko za testiranje določene stvari potrebujemo več testov. Ta seznam nam služi le kod pomoč, da kakšna možnost ne ostane nestestirana. Ker ga napišemo pred začetkom ne more biti preveč natančen, saj še ne poznamo strukture. V procesu razvoja lahko zraven dodajamo nove teste ali brišemo stare, ki jih ne bomo potrebovali.

Seznam testov za našo aplikacijo

Preveri, če:

- se pravilno izpišejo vsi izpiti v seznamu

- dodajanje izpita doda pravi izpit v seznam

- se pravilno izračuna povprečje ocen izpitov

- lahko dodamo izpit brez ocene

- napačen vnos podatkov pri dodajanju izpita vrne opozorilo in ne doda, izpita

- deluje naknadno vpisovanje ocene za izpit

- deluje brisanje izpita

- uporabniški vmesnik omogoča dodajanje izpita
- uporabniški vmesnik omogoča brisanje izpita
- uporabniški vmesnik omogoča urejanje ocene
- uporabniški vmesnik pravilno izpiše povprečje ocen
- …

5.2 Kako izbrati prvi test

Kot sem že napisal programiranje začnemo s testom. Kako pa prvi (ali kasneje v razvoju naslednji) test izberemo? Pregledamo seznam in izberemo tisti tistega, ki ga je najlažje implementirati in nam bo prinesel nekaj novega. Izogibamo se testom, za katere bi porabili preveč časa in z njimi lažje opravimo kasneje.

A začetek lahko izberemo nek realen test iz seznama (na primer dodajanje izpita), ki nam bo odgovoril na več vprašanj:

- Kam ta operacija spada?
- Kakšni so pravilni vhodi?
- Kakšni so pričakovani izhodni rezultati za te vhode?

Pri izbiri realnega primera pogosto ostanemo dolgo brez odziva. Mikro-iteracija rdeče, zeleno, preoblikuj mora biti čimkrajša in nam mora zagotavljati hiter odziv. Za rešitev realnega testa pa lahko potrebujemo veliko kode in celo več razredov.

Zato po navadi začnemo s preprostim testom, ki nam ne zagotavlja delovanja nobene funkcionalnosti. Pomaga nam le postaviti osnovo za delovanje in preverja, če ta zasnova deluje pravilno.

5.3 Prvi test in triangulacija

Naš prvi test:

```
public class TestExam extends TestCase {
    public void testCreateNewExam(){
        Exam exam = new Exam("ARS1", 2, 6 );
        assertEquals("Name should be ARS1", "ARS1", exam.getName());
        assertEquals("Year should be 2", 2, exam.getYear());
        assertEquals("Grade should be 6", 6, exam.getGrade());
    }
}
```

Test preverja kreiranje novega objekta Exam (izpit). Pove nam, da se uporablja konstruktor s tremi parametri. Narediti moramo tudi tri metode, ki vračajo podatke iz konstruktorja. Testi enot morajo biti v razredu, ki deduje (ang. extends) razred TestCase, da jih lahko zaganjamo.

Napišemo kodo, da se naš test prevede. Čeprav je tukaj rešitev očitna, bom razred izpeljal po pravilih testno vodenega razvoja.

```
public class Exam {
    public Exam(String n, int y, int g) {
    }

    public String getName() {
        return null;
    }

    public int getYear() {
        return 0;
    }

    public int getGrade() {
        return 0;
    }
}
```

Zaženemo test, ki se izvede neuspešno (rdeče). Nadaljujemo torej na drugi korak mikro-iteracije. Poskrbeti moramo, da test čim hitreje preide na zeleno. To najhitreje naredimo kar z vračanjem konstantnih vrednosti.

```
public class Exam {
    public Exam(String exam, int year, int grade) {
    }

    public String getName() {
```

```
        return »ARS1«;
    }

    public int getYear() {
        return 2;
    }

    public int getGrade() {
        return 6;
    }
}
```

Test se sedaj izvede uspešno(zeleno). Tak postopek je seveda v tako preprostih primerih nesmiseln in se ga v realnem razvoju uporablja pri bolj zapletenih metodah. Nam pa pokaže, da zelen test še ne pomeni pravilnega delovanja. Ker tukaj preoblikovanje ni potrebno nadaljujemo z naslednjim testom. Napišemo še en test za kreiranje z drugačnimi parametri. Ta postopek se imenuje triangulacija, saj z več različnimi parametri pridemo do prave rešitve.

```
public class TestExam extends TestCase {
    public void testCreateAnotherExam(){
       Exam exam = new Exam("PSS", 1, 9 );
       assertEquals("Name should be PSS ", "PSS", exam.getName());
       assertEquals("Year should be 1", 1, exam.getYear());
       assertEquals("Grade should be 9", 9, exam.getGrade());
    }
}
```

Novi test zdaj pade, zato popravimo kodo.

```
public class Exam {
    private String name;
    private int year;
    private int grade;

    public Exam(String n, int y, int g) {
        name = n;
        year = y;
        grade = g;
    }

    public String getName() {
        return name;
    }
```

```
    public int getYear() {
        return year;
    }

    public int getGrade() {
        return grade;
    }
}
```

Preoblikovanje še vedno ni potrebno saj ni duplikatov in smo s strukturo kode zadovoljni.

5.4 Testi narekujejo arhitekturo

Večkrat se moramo že med pisanjem testov odločiti katere razrede bomo naredili in kako bodo med seboj povezani. Tako nam že testi narekujejo design, ki ga kasneje pri preoblikovanju lahko še uredimo in popravimo. V takih primerih se lahko zgodi, da bomo več časa porabili za pisanje testa, kot pa dejanske produkcijske kode, bomo pa pri tem že sprejeli vse pomembne odločitve.

Kako testi določajo design se vidi pri prvi funkcionalnosti, ki jo bomo implementirali. Pri pisanju testa sem se odločil, da bo dodajanje novega predmeta potekalo preko razreda ExamManager, v katerem bodo vse metode za upravljanje s seznamom izpitov.

Metoda setUp() pri JUnit se vedno izvede na začetku in nam nastavi predpogoje za izvajanje testov. V razvojnem okolju moramo nastaviti, da se zaganjajo testi vseh razredov, saj ima načeloma vsak razred tudi svoj testni razred.

```
public class TestExamManager extends TestCase {
    private ExamManager examManager;
    private Exam exam;

    public void setUp() {
        examManager = new ExamManager();
        exam = new Exam("PSS", 1, 9);
    }

    public void testAddExam() {
        examManager.addExam(exam);
        assertTrue("ExamList should contain added exam",
                        examManager.getExams().contains(exam));
    }
```

```
}
```

Ko pišemo test zraven ustvarimo še vse potrebne razrede in metode, ki jih potrebujemo, da se test prevede.

```
public class ExamManager {

    public void addExam(Exam exam) {
    }

    public List getExams(){
        return null;
    }
}
```

Ker smo naredili le prazno osnovo, nam test pade. Poskrbeti moramo dan se test izvede:

```
public class ExamManager {
    private List<Exam> examList;

    ExamManager()
    {
       examList = new ArrayList<Exam> ();
    }

    public void addExam(Exam exam){
        examList.add(exam);
    }

    public List<Exam> getExams(){
        return examList;
    }
}
```

5.5 Testiranje izjem

Radi bi zagotovili, da bodo imena izpitov unikatna. Ob poskusu dodajanja izpita v seznam, kjer že obstaja izpit z enakim imenom se mora prožiti izjema.

Kot za vso ostalo kodo moramo tudi za izjeme najprej napisati test. V njem bomo ustvarili pogoje, v katerih bi se morala prožiti izjema. To izjemo bomo lovili in preverjali, če se res proži.

```
    public void testDuplicateExamException() throws Exception
    {
        examManager.addExam(exam);
        try{
            examManager.addExam(exam2);
            fail("Should throw DuplicateExamException");
        } catch (DuplicateExamException expected) {}
    }
```

Test nam postavlja pogoje in tip izjeme, ki se mora prožiti.

```
public void addExam(Exam exam) throws Exception{
    Iterator<Exam> iter = examList.iterator();
    while (iter.hasNext()){
      if(iter.next().getName().equals(exam.getName())){
              throw new DuplicateExamException(exam.getName());
      }
    }
    examList.add(exam);
}
```

5.6 Preoblikovanje

Pri preoblikovanju najpogosteje odstranjujemo duplikate in preimenujemo ali krajšamo predolge razrede ter metode. Lahko delamo katerekoli spremembe, ki ohranjajo funkcionalnost modula.

V razredu ExamManager imamo seznam izpitov. Vidimo lahko, da je koda za dodajanje izpitov s preverjanjem duplikatov grda. Lahko bi izločili metodo, ki preverja, če izpit z enakim imenom že obstaja. Iz strukture pa lahko ugotovimo, da je za naše namene namesto liste precej bolj primerna mapa. Nima duplikatov in omogoča hitrejše iskanje elementov po ključu. V preoblikovanju bomo torej listo nadomestili za mapo. Z uporabo orodji za preoblikovanje iz razvojnega okolja so spremembe hitre. Izločimo še metodo za preverjanje duplikatov in dobimo lepšo kodo, ki deluje hitreje in ohranja isto funkcionalnost. Seveda moramo ponovno pognati teste in se prepričati, da nismo ničesar pokvarili.

```
public class ExamManager {
    private Map<String, Exam> examMap;

    ExamManager()
    {
```

```
        examMap = new HashMap<String, Exam>();
    }

    public void addExam(Exam exam) throws Exception{
        String examName = exam.getName();
        checkForDuplication(examName)
        examMap.put(examName, exam);
    }

    private void checkForDuplication(String examName) throws
                                            DuplicateExamException {
        if(examMap.get(examName)!= null)
        {
            throw new DuplicateExamException(examName);
        }
    }

    public Map<String, Exam> getExams(){
        return examMap;
    }
}
```

5.7 Uporaba navideznih (mock) objektov

V naslednjem primeru bomo videli, kako se uporablja navidezne objekte. Kot orodje za simuliranje navideznih objektov sem uporabil EasyMock.

Podatke o datumih izpitov dobimo iz nekega zunanjega sistema. To je lahko podatkovna baza, internet ali datoteka. Radi bi napisali test za metodo, ki bo datume pravilno razporedila po izpitih. Ta metoda bo dobila podatke iz metode nekega drugega modula, ki komunicira z zunanjim sistemom. Pravilo testno vodenega razvoja pravi, da morajo biti testi izolirani in med seboj neodvisni. Zunanji sistem pa je lahko tudi nedosegljiv, kar ne sme ustaviti naših testov. Zato bomo s pomočjo navideznih objektov simulirali odziv metode, ki nam posreduje podatke zunanjega sistema.

Vsa koda tega modula je za naš namen preobsežna. Zato bom za lažje razumevanje le na kratko opisal razrede in spremenljivke, ki v testu nastopajo:

- *ExamImporterInterface* – vmesnik razreda, ki komunicira z zunanjim sistemom

- *List<ExamDate> importExamDates()* – metoda, ki nam posreduje podatke iz zunanjega sistema

- *ExamDate* – Razred, ki vsebuje podatke o imenu in datumu izpita

- *ExamDateOrganizer* – razred, v katerem bomo izvajali operacije z datumi

- *ExamDateOrganizer* – razred, v katerem bomo izvajali operacije z datumi

- *void setDatesToExams(Map<String, Exam> examMap)* – metoda, ki jo testiramo; razporedi datume med izpite

- *private List<Date> futureDates* – atribut razreda *Exam*; vsebuje datume izpitov

V testu z metodo *EasyMock.createMock(ExamImporterInterface.class)* naredimo navidezni objekt, ki predstavlja *ExamImporterInterface*. Nato določimo izhod, ki ga bo ta metod vrnila: *EasyMock.expect(examImporterMock.importExamDates().andReturn(examDateList);*

V testu zaenkrat preverjamo le, če se je k izpitu vpisal en nov datum. Za realno testiranje bi morali napisati več testov s katerimi bi preverili, če se je vpisal pravi datum na pravo mesto, kaj se zgodi, če predmet ne obstaja…

```
public class TestExamDateOrganizer extends TestCase {

      protected static final String pssName = "PSS";
      protected static final Date pssDate = new Date(110, 6, 20);

      private ExamDateOrganizer examDateOrganizer;
      private ExamImporterInterface examImporterMock;
      private List<ExamDate> examDateList;
      private Map<String, Exam> examMap;

public void setUp() {

    examDateOrganizer = new ExamDateOrganizer();
    examImporterMock =
                  EasyMock.createMock(ExamImporterInterface.class);

    examDateOrganizer.setExamImporterInterface(examImporterMock);
    examDateList = new ArrayList<ExamDate>();
    examDateList.add(new ExamDate(pssName, pssDate));
```

```
    examMap = new HashMap<String, Exam>();
    examMap.put(pssName, new Exam(pssName, 1, 9));
}

public void testExamDateImports(){

    EasyMock.expect(examImporterMock.importExamDates())
                                            .andReturn(examDateList);
    EasyMock.replay(examImporterMock);

    examDateOrganizer.setDatesToExams(examMap);
    assertEquals(1, examMap.get(pssName).getFutureDates().size());
    }
}
```

Po vseh korakih testno vodenega razvoja razred zgleda tako:

```
public class ExamDateOrganizer {
    private ExamImporterInterface examImporterInterface;

    public void setExamImporterInterface(ExamImporterInterface ei) {
        examImporterInterface = ei;
    }

    public void setDatesToExams(Map<String, Exam> examMap){
        List examDateList = examImporterInterface.importExamDates();

        Iterator<ExamDate> iter = examDateList.iterator();
        String examName;
        ExamDate examDate;
        Exam currentExam;

        while (iter.hasNext()){
            examDate = iter.next();
            examName = examDate.getName();
            currentExam = examMap.get(examName);
            currentExam.getFutureDates().add(examDate.getDate());
        }
     }
}
```

5.8 Testiranje uporabniškega vmesnika

Pisanje testov za uporabniški vmesnik je lahko precej zapleteno, saj si moramo v glavi ustvariti grafično predstavo komponente. Najlažje je, če si na list skiciramo ali le opišemo katere komponente potrebujemo in kakšna bo njihova funkcija. Postavitev lahko naredimo le okvirno in jo popravimo kasneje pri preoblikovanju.

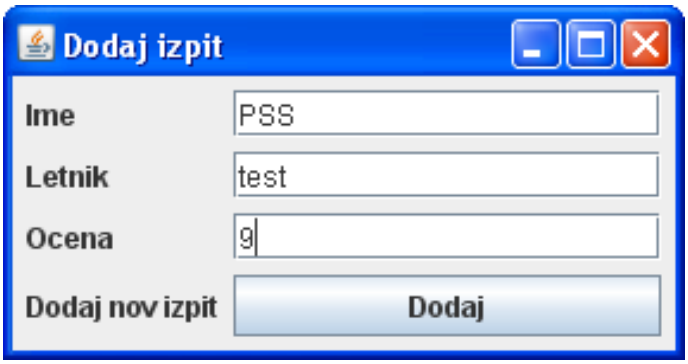
Uporabiti moramo orodje, ki nam prepoznava okno in komponente. Vanje vpisuje besedilo, klika gumbe in bere odzive. V tem primeru se uporablja Jemmy, ki uporablja razrede *Operators* za zaznavanje komponent. Med seboj jih ločuje po imenih ali vrstem redu.

Pri dodajanju novega izpita naj se preveri, če je letnik številka. V nasprotnem primeru nam mora aplikacija vrniti opozorilo. Napisati moramo torej test, ki bo odprl okno in vanj vpisal podatke, ki bodo povzročili izjemo. Nato mora test preveriti, če se je odprlo novo okno z točno določenim tekstom.

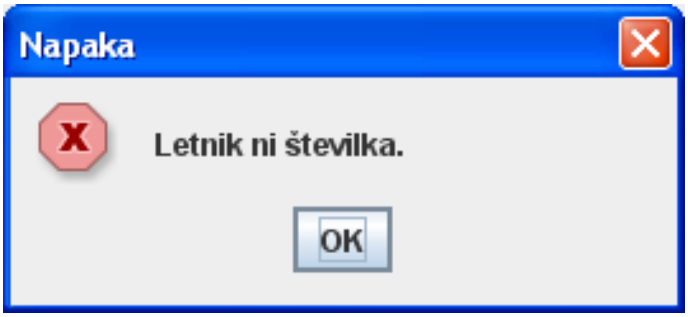
```
public class TestAddExamGUI extends TestCase {
    private JFrameOperator addWindow;
    private JButtonOperator addButton;

protected void setUp(){
     AddExamGUI.start();
     addWindow = new JFrameOperator("Dodaj izpit");
     addButton = new JButtonOperator(addWindow, "Dodaj");
 }

public void testYearIsNotANumberWarning()  {

    JTextField name = JTextFieldOperator.findJTextField(
                      addWindow.getWindow(), null, false, false, 0);

    JTextFieldOperator nameField = new JTextFieldOperator(name);
    nameField.enterText("PSS");

    JTextField year = JTextFieldOperator.findJTextField(
                      addWindow.getWindow(), null, false, false, 1);

    JTextFieldOperator yearField = new JTextFieldOperator(year);
    yearField.enterText("test");

    JTextField grade = JTextFieldOperator.findJTextField(
                      addWindow.getWindow(), null, false, false, 2);
```

```
    JTextFieldOperator gradeField = new JTextFieldOperator(grade);
    gradeField.enterText("9");

    addButton.pushNoBlock();

    JDialogOperator opozorilo = new JDialogOperator("Napaka");
    assertNotNull(opozorilo);
    JLabelOperator message = new JLabelOperator(opozorilo);
    assertEquals("Napačno opozorilo.",
                      "Letnik ni številka.", message.getText());

    JButtonOperator okGumb = new JButtonOperator(opozorilo, "OK");
    okGumb.doClick();
    }
}
```

Jemmy poskrbi, da test pade tudi, če se okno ne odpre. Grafični prikaz komponent, ki so nastale kot posledica testa je viden na sliki 5 in sliki 6.



Slika 5: Vmesnik za dodajanje novega izpita



Slika 6: Dialog, ki se prikaže ob napačnem vnosu

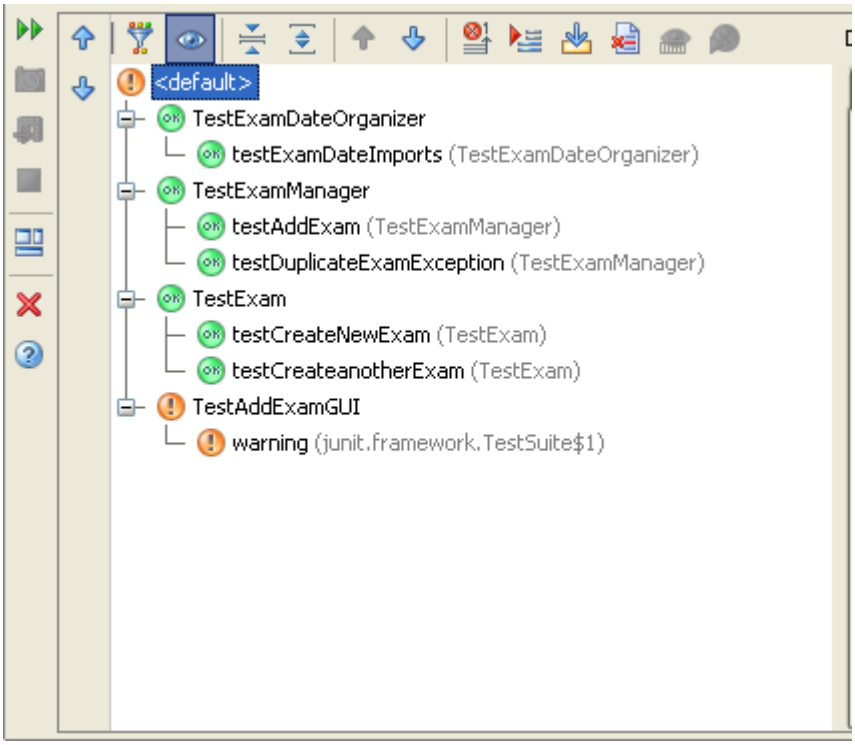
5.9 Analiza nastale aplikacije

Pri razvoju aplikacije je bil uporabljen pristop testno vodenega programiranja. Posledično več kot polovica vse kode pripada testom.

Uporabljala so se naslednja orodja:

- IntelliJ Idea – integrirano razvojno okolje, posebej namenjeno Javi
- EasyMock – knjižnica za navidezne objekte
- Jemmy – knjižnica za testiranje grafičnega vmesnika
- Emma – orodje za analizo pokritosti testov s kodo

Na sliki 7 je prikaz zaganjanja testov JUnit s programom IntelliJ Idea med razvojem.



Slika 7: Zaganjanje testov JUnit

Aplikacija je preprosta zato je pokritost s kodo 100%, kar pomeni da smo se dosledno držali načel testno vodenega razvoja. Ker smo se šele spoznavali s takim načinom razvoja, se je porabilo več časa, kot bi se pri standardnem razvoju. Smo pa dobili nabor testov, ki pokriva celotno aplikacijo. Tako smo lahko prepričani, da vse enote delujejo pravilno.

6 SKLEPNE UGOTOVITVE

V diplomski nalogi smo raziskali tehniko testno vodenega razvoja. Preučili smo teoretično delovanje in se spoznali z različnimi metodami, ki se uporabljajo pri takem razvoju. Primerjali smo ga s standardnim razvojem in videli smo, da ima veliko prednosti a tudi nekaj omejitev. Koristen je predvsem v projektih, kjer se specifikacija skozi čas spreminja. Osredotočili smo se na testno voden razvoj v Javi in preučili orodja, ki se pri tem uporabljajo.

Ta pristop k razvoju smo tudi uspešno uporabili pri izdelavi preproste aplikacije, ki uporabniku (v tem primeru študentu) omogoča pregled nad izpiti. Lahko jih dodaja, briše, računa povprečja ocen ali planira izpitne roke. Pri razvoju smo se dosledno držali pravil testno vodenega razvoja, za vsako vrstico kode prej napisali test in tako dosegli 100% pokritost kode. Na podlagi kode, ki je pri tem nastala smo predstavili potek razvoja in tehnike, ki se najpogosteje uporabljajo (preoblikovanje, triangulacija, navidezni objekti, testiranje izjem, testiranje uporabniškega vmesnika). Spoznali smo, da se testno voden razvoj precej razlikuje od standardnega. Težave smo imeli predvsem na začetku, saj se je težko navaditi na pisanje testov pred kodo. Zato smo za izdelavo porabili več časa, kot bi ga drugače. Videli pa smo, da ima ta pristop velik potencial za bolj zahtevne projekte.

Testno voden razvoj zelo zmanjša verjetnost napak med razvojem, saj je vsak del kode stestiran. Zmanjša se količina produkcijske kode, saj pišemo kodo, ki jo res potrebujemo. Izboljša se design, saj ga narekujejo testi. Poleg teh ima še veliko drugih prednosti. Ima pa tudi svoje omejitve, ki se kažejo predvsem na začetku uporabe.

Uveljavil se je predvsem z zadnjih dveh letih. Pogosto se uporablja v kombinaciji z drugimi tehnikami ekstremnega programiranja.

Začetek uporabe testno vodenega razvoja je lahko precej težaven. Razvijalci morajo upoštevati časovne roke in se raje držijo preverjenega standardnega načina razvoja. Veliko časa se porabi, preden se navadijo na nov način dela, ki se ga mora držati cela ekipa. To so glavni razlogi, zakaj se večino podjetij še ne odloča za testno voden razvoj, kljub njegovim očitnim prednostim.

Večina razvijalcev torej še vedno ne sprejema novega pristopa. Uporaba pa zaradi dobrih rezultatov raste. V prihodnosti lahko pričakujemo, da se bo ta tehnika uveljavila predvsem v podjetjih, ki izdelujejo poslovne aplikacije. Rast popularnosti pa se kaže tudi v pojavu tečajev, ki razvijalce učijo testno vodenega razvoja.

LITERATURA

[1.] **S. W. Ambler,** *Introduction to Test Driven Design (TDD),* http://www.agiledata.org/essays/tdd.html, 20.3.2010

[2.] **D. Astels,** *Test-Driven Development: A Practical Guide,* Prentice Hall PTR, 2003

[3.] **K. Beck,** *Test-Driven Development By Example,* Addison-Wesley Professional, Boston, 2002

[4.] **K. Beck, C. Andres,** *Extreme Programming Explained: Embrace Change, Second Edition,* Addison Wesley Professional, 2004

[5.] **P. Čibokli**, *Avtomatizacija testiranja kot ključ agilnosti razvoja programske opreme*, Magistrsko delo, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, 2006

[6.] **R. Gold, T. Hammell, T. Snyder,** *Test Driven Development: A J2EE Example,* Apress, 2004

[7.] **U. Kočevar**, *Testiranje programske opreme in izdelava načrta za testiranje time & space sistema*, Diplomsko delo, Ekonomska fakulteta, Univerza v Ljubljani, Ljubljana, 2006

[8.] **L. Koskela,** *Test driven, TDD and Accaptance TDD for Java Developers,* Manning Publications Co, *2007*

[9.] **J. Link, P. Frohlich,** *Unit Testing in Java: How Tests Drive the Code,* Morgan Kaufmann, Heidelberg, 2003

[10.] **E. M. Maximilien, L. Williams,** *Assessing Test-Driven Development at IBM,* Proc. 25th Int'l Conf. Software Eng (ICSE 03), IEEE CS Press, 2003

[11.] **C. Murphy**, *Improving Application Quality Using Test-Driven Development*, Methods & Tools, Spring 2005

[12.] **Wikipedia**, *Test driven development*, http://en.wikipedia.org/wiki/Test-driven_development, 15.3.2010

[13.] **M. Pancur**, *Vpliv testno vodenega razvoja na parametre agilnega razvojnega procesa*, Doktorska disertacija, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, 2005

[14.] **N. Nagappan, E. M. Maximilien, T. Bhat, L. Williams**, *Realizing quality improvement through test driven development: results and experiences of four industrial teams*, Springer Science Business Media, 2008