UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matjaž Pančur

Vpliv testno vodenega razvoja na parametre agilnega razvojnega procesa

doktorska disertacija

mentor: doc. dr. Tone Vidmar

somentor: prof. dr. Blaž Zupan

Ljubljana, 2005

KAZALO

KAZALO **1**

POVZETEK **5**

ABSTRACT **7**

1 UVOD **11**

1.1 Pregled vsebine disertacije **13**

1.2 Prispevki k znanosti **15**

2 AGILNE METODOLOGIJE IN TESTNO VODEN RAZVOJ **17**

2.1 Metodologija XP **17**

2.2 Testno voden razvoj TDD **27**

2.3 Iterativni razvoj z naknadnim testiranjem (ITL) **29**

2.4 Primerjava med procesoma razvoja TDD in ITL **30**

2.5 Sorodne raziskave **32**

2.5.1 Sorodne kvalitativne raziskave 38

2.5.2 Kritika in primerjava dosedanjih kontroliranih eksperimentov 39

3 NAČRT IN IZVEDBA EKSPERIMENTA **43**

3.1 Cilji **43**

3.2 Eksperimentalne spremenljivke, teorije in hipoteze **44**

3.3 Tip eksperimenta **48**

3.4 Subjekti **48**

3.5 Eksperimentalne naloge **49**

3.6 Zbiranje podatkov **52**

3.6.1 Vodenje verzij kode 53

3.6.2 Zbiranje podatkov o procesu razvoja 53

3.7 Grožnje veljavnosti eksperimenta 55

3.7.1 Interna veljavnost 56

3.7.1.1 Hawthornov efekt..... 56
3.7.1.2 Pristranost zaradi smrtnosti..... 56
3.7.1.3 Pristranost kontrolne skupine..... 57
3.7.2 Eksterna veljavnost..... 57

3.7.3 Pristranost zaradi ene izvedbe 59

3.7.4 Pristranost zaradi nepopolnega upoštevanja zapovedanega procesa dela 60

3.8 Analiza podatkov 60

3.9 Ocena statistične moči 61

3.10 Potek eksperimenta 64

3.10.1 Trening posameznih aktivnosti v procesu razvoja 64

3.10.2 Izvedba eksperimenta 65

4 ANALIZA MERITEV IN REZULTATI 69

4.1 Produktivnost 70

4.1.1 Kontekst samostojnih razvijalcev 71

4.1.2 Kontekst programiranja v parih 73

4.2 Kvaliteta 76

4.2.1 Zunanja kvaliteta 76

4.2.1.1 Kontekst samostojnih razvijalcev 77
4.2.1.2 Kontekst programiranja v parih 78
4.2.2 Notranja kvaliteta 78

4.2.2.1 Kontekst samostojnih razvijalcev 79
4.2.2.2 Kontekst programiranja v parih 80
4.3 Testi 84

4.3.1 Število testov 85

4.3.1.1 Kontekst samostojnih razvijalcev 86
4.3.1.2 Kontekst programiranja v parih 87

4.3.2 Kvaliteta testov ... 89

4.3.2.1 Kontekst samostojnih razvijalcev ... 89
4.3.2.2 Kontekst programiranja v parih ... 91
4.4 O procesu razvoja ... 94

4.4.1 Frekvenca poganjanja testov ... 94

4.4.1.1 Kontekst samostojnih razvijalcev ... 94
4.4.1.2 Kontekst programiranja v parih ... 96
4.4.2 Odstotek uspešnih testov ... 98

4.4.2.1 Kontekst samostojnih razvijalcev ... 98
4.4.2.2 Kontekst programiranja v parih ... 99
4.4.3 Faze dela: kodiranje funkcionalnosti, testiranje, preoblikovanje ... 102

4.4.3.1 Kontekst samostojnih razvijalcev ... 103
4.4.3.2 Kontekst programiranja v parih ... 104
4.4.4 Razmerje med testi in kodo ... 107

4.4.4.1 Kontekst samostojnih razvijalcev ... 107
4.4.4.2 Kontekst programiranja v parih ... 109
4.4.5 Kvalitativni parametri procesa razvoja ... 112

4.5 Povzetek rezultatov analize ... 117

5 SKLEP ... 119

SLOVARČEK IZRAZOV ... 125

LITERATURA IN VIRI ... 127

IZJAVA ... 133

ZAHVALA ... 135

PRILOGA A – REZULTATI MERITEV EKSPERIMENTA S SAMOSTOJNIMI RAZVIJALCI 2003 (SR03) ... 137

PRILOGA B – EKSPERIMENTALNE NALOGE ... 145

PRILOGA C – PRAKTIČNI PRIKAZ TDD ... 193

PRILOGA D – PROCESSLOG ... 197

PRILOGA E – ANKETE IN REZULTATI ... 211

POVZETEK

Disertacija obravnava področje empiričnega določanja vrednosti kvantitativnih in kvalitativnih parametrov testno vodenega razvoja (TDD) v primerjavi z iterativnim razvojem z naknadnim testiranjem (ITL). TDD je osrednja praksa metodologije XP in ji pripisujejo največji delež pri njenem bliskovitem uspehu med privrženci agilnih metodologij. Dokazi o učinkih uporabe metodologije XP in TDD pa so bili do sedaj precej nepoznani, na tem področju je zelo malo znanstveno dokazanih učinkov uporabe.

Cilj naše raziskave je bil, da s kontroliranimi eksperimenti ugotovimo, če in v kakšni meri vnaprejšnje pisanje testov vpliva na parametre agilnega razvojnega procesa in v njem nastalega izdelka (programa). Predvsem so nas zanimali trije parametri: produktivnost, kvaliteta (zunanja in notranja) in testiranje (količina in kvaliteta testov). Ocenili smo tudi velikost nekaterih drugih parametrov, ki določajo posamezen proces dela: koliko časa razvijalci porabijo za posamezno fazo procesa razvoja, kako pogosto poganjajo teste, kolikšen je delež uspešnih testov in kakšno je količinsko razmerje med testi in kodo. Kvantitativne parametre procesa razvoja smo ovrednotili tudi skozi kvalitativna opažanja razvijalcev, ki ga uporabljajo. Med temi kvalitativnimi parametri nas je zanimal predvsem pogled razvijalcev na produktivnost, učinkovitost in kvaliteto ter težavnost osvojitve posameznega procesa razvoja.

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v produktivnosti in kvaliteti. V TDD v številu testov ni statistično značilnih razlik v primerjavi z ITL, vendar pa je njihova njihova kvaliteta za 7% večja. TDD torej usmerja razvijalce k pisanju bolj kvalitetnih testov od procesa ITL, ob približno enakem številu testov. Študenti so manj navdušeni kot profesionalci glede produktivnosti, učinkovitosti in kvalitete v TDD. V primerjavi s profesionalnimi programerji študentje dojemajo TDD kot precej težje osvojljiv proces razvoja. Proces razvoja ITL pa je po drugi strani za študente opazno bolj učinkovit v primerjavi s procesom razvoja TDD, lažje ga je osvojiti in ga imajo študentje v večini še namen uporabljati v prihodnosti. Na podlagi rezultatov eksperimentov smo podali priporočila za uvedbo TDD v profesionalna in univerzitetna okolja.

ABSTRACT

This dissertation is about the empirical assessment of quantitative and qualitative parameters of Test-Driven Development (TDD).

The introduction presents the problem of unsuccessfulness of software development projects and emergence of agile methodologies. The common characteristics of all agile methods including most popular amongst them – Extreme Programming (XP) – are presented. Detailed knowledge of methodologies' components enables focused tailoring of chosen methodology by introduction of proper techniques and development processes that minimize or eliminate its weaknesses.

The second chapter describes basic characteristics of the XP and the role of TDD within it. TDD represents central practice of XP and it is attributed with the biggest share of XP's successfulness among developers. In TDD, developer writes tests before corresponding implementation code. A process called ITL (Iterative Test-Last Development) was derived from TDD with all of its characteristics except one: developers in ITL write tests after the code. TDD and ITL are described and compared in detail. Currently there are no known proofs about the effects of TDD usage. Some case studies exist, but unfortunately only a few controlled experiments. At the end of the second chapter all related research is presented, reviewed and compared to our approach.

The third chapter focuses on detailed description of experiment design and its execution. The goal of our research is to identify if and how writing tests first influences **productivity, quality** (external and internal) and **tests** (quantity and quality).We also want to estimate various other parameters that describe development process: time spent in phases of coding, testing and refactoring; frequency of test evaluations, percentage of successful tests and test to code ratio.

All these quantitative parameters are evaluated through participating developers' qualitative observations, with focus on productivity, efficiency, quality and difficulties in adoption of TDD and ITL.

Based on our theories, hypotheses are formalized. All experimental subjects are described, along with experimental tasks, data collection and validation mechanisms. We explore and dismiss all threats to experimental validity, and then we present data analysis procedures with special emphasis on statistical power. Experiments are conducted in two sets: the first set from February 2003 to June 2003, the second set from February 2004 to June 2004. Our subjects were undergraduate Computer Science students in their eight semester of study. Two equivalent groups were formed and the only independent variable in the experiment was the development process (TDD or ITL). All subjects listened to the same classes, worked with same tools under same conditions, etc. Each set consisted of two separate experiments in two different contexts. The first context was programming in pairs (emulation of TDD in XP), while in the second context subjects worked alone (emulation of using TDD in some other agile or traditional methodology).

The fourth chapter presents results and analysis of experimental data. We found **no statistically significant differences between TDD and ITL regarding productivity and quality**. Because of sufficient statistical power we can conclude with high probability that t**here are no big differences in productivity and quality**. We also didn't find any statistically significant differences in the number of tests, but the **test quality was 7% higher in TDD group**. TDD steers its developers towards increased test quality whereas the number of tests stays roughly the same as in ITL.

ITL has 20% more successful test runs, which also confirms our theory and TDD group's process conformance. We found no differences in test-to-code ratio and test evaluation frequency. Because of contradictory results we can neither confirm nor reject hypotheses about differences in time spent in coding, testing and refactoring.

The effect size is essential in designing an experiment with sufficient power. Our effect size estimations will help other researchers in replicating our experiments or designing new experiments with focus on TDD.

On average, students were less enthusiastic about TDD's productivity, efficiency and quality compared to responses from professional developers. Students perceive TDD as much harder practice to adopt then professionals do. ITL, on the other hand, is much easier to adopt and is also perceived as much more efficient than TDD. Majority of students also tend to use ITL in the future, which is not the case with TDD.

In the conclusions, we evaluate our findings and contributions and offer some guidance about introduction of TDD and ITL in a professional development environment and in a University environment. Some suggestions for further research are presented at the end.

Our main contributions to science are:

- Design and implementation of the experimental environment.
- Estimation of effect sizes for quantitative parameters of TDD and ITL. The size of the effect can be small at most in productivity and external quality, medium in test quantity, frequency of test evaluations and test to code ratio. We proved a big effect in the difference of mean values between TDD and ITL in test quality and percentage of successful test runs.
- Method for empirical assessment of quantitative parameters (productivity, quality, tests, time spent in development phases, frequency of test evaluations, percentage of successful tests and test to code ratio) and qualitative parameters (productivity, efficiency and quality, difficulties in adoption) for TDD and ITL.
- Validity analysis of research hypotheses of the differences between TDD and ITL in productivity, quality, tests, time spent in development phases, frequency of test evaluations, percentage of successful tests and test to code ratio.
- Recommendations for introduction of TDD and ITL in a professional development environment and validation of suitability and its acceptance time in a University environment.

1 UVOD

Na področju programskega inženirstva je že dalj časa prisoten problem nizke uspešnosti velikega dela projektov razvoja programske opreme. Po podatkih skupine Standish Group je sicer delež uspešnih projektov[1] narasel s 16% v letu 1994 [Standish 1995] na 34% v letu 2002 [Standish 2003], vendar je delež neuspešnih (ukinjenih) in pogojno uspešnih[2] projektov v letu 2002 še vedno 66% [Standish 2003]. Podobno so v analizi 210 programskih razvojnih projektov [QSM 2000] raziskovalci ob poskusu merjenja napredka projekta ugotovili, da se je plan zaradi spreminjanja zahtev tolikokrat spremenil, da v skoraj polovici primerov razvijalci sploh niso več našli originalnih načrtov o časovnih in finančnih okvirih projekta. Takšna situacija je nastala zato, ker slepo podrejanje planu ni bilo več primarni cilj, ampak je prioriteta postala zadovoljitev novih zahtev stranke [Highsmith+ 2001]. Tradicionalni pristopi temeljijo na domnevi, da je z dovolj truda mogoče že zelo zgodaj v ciklu projekta predvideti celotno množico zahtev in tako zmanjšati stroške. Vendar ta togost dandanes ni več zaželena in pomeni manjšo sposobnost odzivanja na nove poslovne razmere, kar pa je za podjetja v t.i. novi, globalni internetni ekonomiji ključnega pomena.

V zadnjih nekaj letih so se tako kot odgovor na problem nenehnih sprememb ter na togost in birokratskost tradicionalnih "težkih" pristopov k razvoju programske opreme pojavile t.i. "lahke metodologije". Leta 2001 so se s skupno izjavo[3] najbolj znanih avtorjev s tega novega področja preimenovale v agilne metodologije [agilealliance, Abrahamsson+ 2002, Boehm 2002, Cockburn 2002, DeMarco+ 2002, Highsmith 2002,

---

[1] Uspešni projekti so projekti končani v dogovorjenih časovnih, finančnih in vsebinskih okvirih.

[2] Pogojno uspešni projekti (v originalu "*challenged*") so projekti s prekoračitvami proračuna, časovnih okvirov in z manj razvite funkcionalnosti, kot je bilo dogovorjeno na začetku.

[3] Ta je v krogih podpornikov agilnih metodologij znana kot agilni manifest (*agile manifesto*) [agilealliance].

Martin 2003, Melnik+ 2002, Reifer 2002]. Ideja za agilnimi pristopi razvoja je, da je lahko razvojna skupina bolj učinkovita v odzivanju na spremembe, če:

(1) zmanjšamo strošek toka informacij znotraj skupine in

(2) zmanjšamo čas med odločitvijo in vidnimi posledicami te odločitve (povečamo odzivnost[4]) [Cockburn+ 2001].

Za agilni razvoj je značilno, da se osredotoča na talente in veščine razvijalcev in oblikuje ter prilagaja proces specifični situaciji in razvojni skupini in ne obratno, kot je pogosta praksa v tradicionalnih metodologijah. Med vsemi agilnimi metodologijami je t.i. "metodologija ekstremnega programiranja" XP (*eXtreme Programming*) [Beck 2000] daleč najbolj razširjena in popularna, ostali manj razširjeni, a še vedno precej poznani, predstavniki agilnih metodologij pa so SCRUM, adaptivni razvoj programske opreme (Adaptive Software Development – ASD) [Highsmith 2000], razvoj po značilnostih (Feature Driven Development – FDD) in družina metodologij Crystal (http://www.crystalmethodologies.org).

Dokazov o učinkih uporabe večine agilnih metodologij in agilnih procesov ter pristopov k razvoju je relativno malo. Obstaja sicer veliko različnih študij primerov in opisov uspeha v praktični uporabi[5], zelo malo pa je trenutno na tem področju znanstveno dokazanih učinkov njihove uporabe. Pomanjkanje eksperimentiranja v procesu dokazovanja pravilnosti različnih teorij je v računalništvu prisotno že dalj časa, verjetno zaradi relativne mladosti same vede [Tichy 1998]. Da bi lažje ocenili uspešnost in primernost metodologije kot celote je potrebno prej dobro poznati dejstva o njenih sestavnih delih (praksah, procesih, vrednotah, itd.) ter njihovo uporabo in obnašanje v različnih kontekstih. Šele na osnovi podrobnega poznavanja teh "sestavnih delov" bo mogoče lažje načrtovati in spremljati tudi njihove sinergistične učinke, to je učinke metodologije kot celote.

---

[4] *Feedback* – odzivnost, povratne informacije.

[5] Veliko jih je objavljenih v zbornikih dveh mednarodnih konferenc, ki se ukvarjata s področjem agilnih metodologij ("International Conference on Extreme Programming and Agile Processes in Software Engineering" in "XP/Agile Universe").

Podrobno poznavanje "sestavnih delov" posamezne metodologije nam omogoča ciljno odpravljanje slabosti posamezne metodologije oz. procesa razvoja s tehnikami in procesi, ki te slabosti omilijo.

1.1 Pregled vsebine disertacije

V uvodu je predstavljen problem nizke uspešnosti velikega dela projektov razvoja programske opreme ter pojav agilnih metodologij. Podane so skupne značilnosti vseh agilnih metodologij in najbolj znane agilne metodologije, XP (*Extreme Programming*). Podrobno poznavanje "sestavnih delov" posamezne metodologije omogoča ciljno odpravljanje slabosti posamezne metodologije oz. procesa razvoja s tehnikami in postopki, ki te slabosti omilijo.

V drugem poglavju so opisane osnovne značilnosti metodologije XP ter kako je vanjo umeščen testno voden razvoj (angl. *Test-Driven Development* – TDD). TDD je osrednja praksa metodologije XP in ji pripisujejo največji delež pri bliskovitem uspehu metodologije XP. Pri TDD razvijalec vedno najprej napiše teste in šele nato funkcionalno kodo. Dokazov o učinkih uporabe testno vodenega razvoja do sedaj ni. Obstaja sicer kar nekaj študij primerov, zelo malo pa je bilo na tem področju kontroliranih eksperimentov. V nadaljevanju poglavja je opisan TDD in iz njega izveden kontrolni proces razvoja, ki smo ga poimenovali iterativni proces razvoja z naknadnim testiranjem (angl. Iterative Test-Last Development - ITL). Na koncu drugega poglavja so podrobno predstavljene še objavljene sorodne raziskave ter njihova kritika in primerjava z našim delom.

V tretjem poglavju je podrobno opisano načrtovanje in izvedba naših eksperimentov. Cilj naše raziskave je, da ugotovimo, če in v kakšni meri vnaprejšnje pisanje testov vpliva na **produktivnost, kvaliteto** ter **testiranje** (količina in kvaliteta testov). Oceniti želimo tudi vrednost nekaterih drugih parametrov, ki določajo posamezen proces dela: koliko časa razvijalci prebijejo v posamezni fazi procesa razvoja, kako pogosto poganjajo teste, kolikšen je delež uspešnih testov in kakšno je količinsko razmerje med testi in kodo.

Zgoraj zbrane kvantitativne parametre procesa razvoja smo želeli ovrednotiti tudi skozi **kvalitativna opažanja** razvijalcev, ki ga uporabljajo. Med temi kvalitativnimi parametri

je zanimiv predvsem pogled razvijalcev na produktivnost, učinkovitost in kvaliteto ter težavnost osvojitve posameznega procesa razvoja.

Za vse omenjene parametre so postavljene formalizirane hipoteze. Opisani so subjekti, eksperimentalne naloge in način zbiranja podatkov. Podrobno so obdelane vse grožnje veljavnosti eksperimenta, predstavljen je način analize podatkov. Sledi analiza statistične moči in podroben opis same izvedbe eksperimenta. V letih 2003 (februar – junij 2003) in 2004 (februar-junij 2004) sta bila izvedena dva niza eksperimentov, subjekti so bili študenti osmega semestra univerzitetnega študija računalništva in informatike na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Razdeljeni so bili v dve enakovredni skupini, edina neodvisna spremenljivka v vsakem eksperimentu je bila pisanje testov pred ali po kodi. Subjekti so tako poslušali enaka predavanja, delali z enakimi orodji, imeli enake pogoje dela, itd. Vsak niz eksperimentov je bil izveden v dveh kontekstih: kontekstu dela v parih, ki imitira tipično uporabo procesa razvoja TDD znotraj metodologije XP, in kontekstu dela samostojnih razvijalcev, ki imitira uporabo testno vodenega razvoja znotraj kakšne druge agilne ali tradicionalne metodologije.

Četrto poglavje zajema analizo v eksperimentih pridobljenih meritev in rezultate analize. Zaradi dovolj velike statistične moči testov lahko tudi s precejšnjo gotovostjo trdimo, da **velikih razlik med procesom TDD in ITL glede produktivnosti in kvalitete ni**. V procesu razvoja TDD **v številu testov ni statistično značilnih razlik** v primerjavi s procesom razvoja ITL, **precej večja pa je v TDD njihova kvaliteta** (7% izboljšanje). **Proces razvoja TDD torej** ob približno enakem številu testov **usmerja razvijalce k pisanju bolj kvalitetnih testov** od procesa razvoja ITL.

ITL ima v povprečju za 20% več uspešnih testov od procesa razvoja TDD, kar med drugim tudi potrjuje skladnost naših razvijalcev v TDD skupini s procesom razvoja TDD. V razmerju med količino testov in kode ter v frekvenci poganjanja testov razlik nismo zaznali. Zaradi nasprotujočih si rezultatov ne moremo ne zavrniti ne potrditi hipoteze o razlikah med TDD in ITL skupino v odstotku prebitega časa v fazi kodiranja, testiranja in preoblikovanja.

Velikost efekta je zelo pomembna pri načrtovanju novih eksperimentov na področju testno vodenega razvoja ali morebitnih replikacij naših eksperimentov, s katerimi bi dobili bolj zanesljive ocene o velikosti posameznih parametrov. V naših eksperimentih,

razen pri kvaliteti testov, nismo našli velikih razlik med primerjanjem kvalitete in produktivnosti procesov razvoja TDD in ITL. Zaradi dovolj velike statistične moči testa pa lahko s precejšnjo gotovostjo trdimo, da velikih razlik tam, kjer jih nismo zaznali, ni. Naše ocene o pričakovani velikosti efekta pri produktivnosti, kvaliteti in drugih parametrih bodo drugim raziskovalcem omogočile pravilno načrtovanje novih eksperimentov z dovolj velikim vzorcem, ki jim bo omogočal tudi dovolj veliko statistično moč testov za morebitno sprejetje ničelnih hipotez.

V povprečju so bili študenti manj navdušeni kot profesionalci glede produktivnosti, učinkovitosti in kvalitete v TDD. V primerjavi s profesionalnimi programerji študentje dojemajo TDD kot precej težje osvojljiv proces razvoja. Proces razvoja ITL je za študente opazno bolj učinkovit v primerjavi s procesom razvoja TDD. ITL je tudi lažje osvojiti in ga imajo študentje v večini še namen uporabljati v prihodnosti, prav tako kot tudi avtomatsko testiranje, ogrodje JUnit, razvojno okolje Eclipse in proces preoblikovanja.

V sklepu ovrednotimo glavne ugotovitve disertacije ter navajamo nekaj možnosti za nadaljnje raziskovanje na področju empiričnega programskega inženirstva. Podamo priporočila glede uvajanja procesa razvoja TDD in ITL v profesionalno razvojno skupino in v univerzitetno okolje.

1.2 Prispevki k znanosti

Disertacija vsebuje naslednje prispevke k znanosti:

- Zasnova in izdelava eksperimentalnega okolja.
- Ocenitev predvidenih velikosti razlik med parametri v TDD in ITL. Velikost efekta je lahko kvečjemu majhna pri produktivnosti in zunanji kvaliteti, kvečjemu srednje velika pri številu testov, frekvenci poganjanja testov ter razmerju med testi in kodo. Velik efekt smo dokazali pri kvaliteti testov in pri deležu uspešnih testov.
- Metoda za empirično ugotovitev vrednosti kvantitativnih parametrov (produktivnost, kvaliteta, testi, porabljeni čas za posamezno fazo razvoja, frekvenca poganjanja testov, delež uspešnih testov, razmerje med testi in kodo) in

kvalitativnih parametrov (produktivnost, učinkovitost in kvaliteta, težavnost osvojitve) za procesa razvoja TDD in ITL.

- Analiza veljavnosti raziskovalnih hipotez o razlikah med TDD in ITL glede produktivnosti, kvalitete, testiranja, porabljenega časa za posamezno fazo razvoja, frekvence poganjanja testov, deleža uspešnih testov in količinskega razmerja med testi in kodo.
- Priporočila za vpeljavo TDD in ITL v profesionalno okolje in empirična validacija smotrnosti uporabe ter hitrosti sprejetja testno vodenega razvoja v univerzitetnem okolju.

2 AGILNE METODOLOGIJE IN TESTNO VODEN RAZVOJ

Koncept izrazito iterativenega in inkrementalenega razvoja, ki je osnovna značilnost vseh agilnih metodologij, je že zelo star. Larman in Basili [Larman+ 2003] v svojem pregledu iterativnega in inkrementalnega razvoja navajata prve primere celo iz sredine petdesetih let prejšnjega stoletja! Sama ideja iterativnega razvoja tako nikakor ni zamisel gurujev agilnih metodologij. Treba pa je priznati, da so ravno agilne metodologije v zadnjih nekaj letih precej naredile za popularizacijo in ponovno prepoznavnost iterativnega razvoja, prav tako pa so tudi izpostavile slabosti nefleksibilnega klasičnega slapovnega razvoja.

Med vsemi agilnimi metodologijami je t.i. "metodologija ekstremnega programiranja" XP (angl. *eXtreme Programming*) [Beck 2000] daleč najbolj razširjena in popularna, saj je imela že v letu 2001 več kot 38% delež [Charette 2001], ki le še raste. Ravno tako je tudi število raziskav, objavljenih člankov in knjig o metodologiji XP že pred leti postalo veliko večje od vseh drugih agilnih pristopov skupaj [Succi+ 2001, Newkirk+ 2001, Jeffries+ 2001, Beck+ 2001, Wake 2002, Auer+ 2002, Maurer+ 2002]. Osnovne značilnosti metodologije XP smo povzeli po [Beck 2000, Pančur 2001]. Predstavili jo bomo na hitro in le v tolikšni meri, da se lažje vidi tipično umeščanje uporabe testno vodenega razvoja znotraj primera celovite agilne metodologije razvoja kot je XP.

2.1 Metodologija XP

Korenine metodologije XP ležijo v skupini uporabnikov Smalltalk-a (še posebej sta bila tu aktivna Kent Beck in Ward Cunningham) v poznih osemdesetih letih in začetku devetdesetih let prejšnjega stoletja. Najpomembnejši korak se je zgodil leta 1996, ko so se iz *ad-hoc* praks razvili začetki metodologije v za zagovornike metodologije XP znamenitem projektu C3 v podjetju Chrysler [Beck 2000].

XP v osnovi ne prinaša kakih novih revolucionarnih prvin. Vse ideje, ki jih vnaša v proces razvoja, so že znane. Tehnike programiranja, ki jih uporablja, so dobro znane že vrsto let. Novo pa je prepletanje teh tehnik v celoto, kjer si tehnike "pomagajo" do veliko večje skupne vrednosti, kot pa jo imajo posamezne tehnike.

XP upravlja potek projektov s pomočjo štirih spremenljivk:

- stroški,
- čas,
- kakovost, in
- obseg dela.

Stranka lahko izbere in določi katerekoli tri spremenljivke, razvojna ekipa pa vedno določi vrednost četrte spremenljivke.

Največkrat se s strani uprave/poslovnih strank določijo prve tri spremenljivke: npr. takšne in takšne vložke v tem časovnem obdobju s tako in tako kakovostjo. Razvojna skupina pa odloči, kolikšen obseg dela je možno v tem času opraviti. Nikakor pa ne sme ista stran izbrati in določiti vseh štirih spremenljivk. Smiselno je, da je kakovost sistema kar največja, saj ni ne v interesu stranke ne razvijalca imeti nekvalitetno programsko opremo.

Poslovne odločitve v metodologiji XP sprejema stranka/uprava/naročnik, tehnične pa razvojna skupina. Projekt mora biti voden s poslovnimi odločitvami, katerih osnova morajo biti tehnične odločitve glede ocene stroškov in tveganja.

Stranka in vodstvo tako običajno izberejo:

- obseg posameznih izdaj,
- datume izdaj,
- relativno prioriteto predlaganih funkcionalnosti (katere bodo narejene prej, pač glede na poslovno vrednost posamezne funkcionalnosti), in
- katere zgodbe (zahteva oz. enota funkcionalnosti v XP) naj bodo narejene v posamezni izdaji.

Razvojna skupina pa določi:

- oceno časa, potrebnega za implementacijo različnih funkcionalnosti (zgodb),
- oceno posledic uporabe različnih tehnoloških možnosti, ki jih stranki samo predlaga – še vedno pa stranka odloči, katero bodo uporabili,
- način organiziranosti razvojne skupine,
- tveganje pri vsaki značilnosti oz. zgodbi, in
- zaporedje razvijanja značilnosti oz. zgodb v okviru posamezne iteracije (prvo naredijo najbolj tvegano značilnost, saj s tem zmanjšajo tveganje).

Načrtuje se zelo pogosto (t.i. igra načrtovanja – angl. *planning game*). Ko se pojavi potreba, nam to omogoča, da lahko tako stranka kot tudi razvijalci spremenijo in prilagodijo načrt izdaj.

Glavne vrednote metodologije XP so:

- komunikacija

  Pri komunikaciji se osredotoči XP na gradnjo vzajemnega razumevanja problemov ob čim manjši možni formalni dokumentaciji in čim večji osebni komunikaciji. To velja tako za komunikacijo med razvijalci samimi, kot tudi za komunikacijo med razvijalci in stranko ter razvijalci in vodstvom.

- preprostost

  Preprostost je ena najtežje dosegljivih vrednot za razvijalca. Potrebno je odmisliti vse zahteve, ki trenutno niso res nujne. Vprašati se moramo: "Kaj je najpreprostejša stvar, ki še deluje?". Če naredimo stvari preproste, s tem ustvarimo okolje, v katerem bo strošek spremembe v prihodnosti nizek. V praksi se izkaže, da je ravno preprostost pri kompleksnih problemih težko zagotoviti in ohranjati skozi celoten cikel razvoja, saj to zahteva veliko praktičnih izkušenj.

- odzivnost

  Odzivnost je sposobnost strank seznaniti se z implementiranim delom sistema in reagirati nanj (povratne informacije). Težnja XP po kratkih razvojnih ciklih podpira predajo manjših enot sistema čim prej v uporabo, da bodo tudi morebitne napake čim prej ugotovljene.

- pogum

  V kontekstu prvih treh vrednot je potreben pogum. Pogum, da naredimo, kar je prav, tudi če na nas pritiskajo, naj naredimo nekaj drugega. Primer je pritisk na razvijalce, naj izdajo verzijo programa, ki ima še kar nekaj napak. Velikokrat se primeri, da ljudem ne manjka poguma, ampak je problem v prepričanju. Težko se je aktivno zavzemati za nekaj, o čemer nismo povsem prepričani in v kar ne verjamemo. Pogum je tudi rezultanta prejšnjih treh vrednot. Če naša praksa razvoja programske opreme temelji na komunikaciji, preprostosti in odzivnosti, dobimo tudi pogum in prepričanje, da je pot razvoja programske opreme z metodologijo XP prava.

Metodologija XP je sestavljena iz **dvanajstih praks**, ki jih skupaj povezujejo in podpirajo štirje zgoraj omenjeni principi oz. vrednote. Prakse metodologije XP so:

- igra načrtovanja (*the planning game*)

  Razvoj programske opreme je stalno razvijajoča se interakcija med poslovno in tehnično stranjo ekipe. Dialog teče glede na to, kaj je možno in kaj je zaželeno. Planiranje je torej v metodologiji XP podobno tistim v drugih iterativnih RAD pristopih do izvedbe projekta. Uporablja kratke, večinoma tritedenske cikle, pogoste popravke, razdelitev prioritet med poslovne in tehnične ter dodeljevanje "zgodb". Zgodba definira posamezno značilnost, ki jo mora program obsegati in se jo napiše na kartico.

- majhne in pogoste izdaje (*small releases*)

  Vsaka izdaja naj bo tako majhna, kot je le možno, da še zmeraj predstavlja zaključeno celoto. V poslovni svet naj prinese implementacijo najbolj koristnih zahtev (tistih z najvišjo prioriteto in največjo vrednostjo za stranko). Pogostost izdaj naj bo vsaj vsakih nekaj tednov.

- metafora (*metaphor*)

  Metafore služijo za skupno poimenovanje tehničnih elementov sistema in povezav med njimi, vse v smislu nedvoumne komunikacije med vsemi osebami, vpletenimi v razvoj sistema. Metafora in arhitektura sistema sta do neke mere sinonima: oba predstavljata širši, globalni pogled na cilje projekta. Vendar pa je

arhitektura velikokrat nerazumljiva in preveč zapletena (tudi notacija!), zato uporablja XP metaforo kot poskus definicije globalnega pogleda na cilje projekta, s katerim se lahko poistovetijo tako razvijalci kot tudi naročniki.

- preprosta zasnova oz. načrt (*simple design*)

  Sistem naj bo v vsakem trenutku načrtovan kar najbolj preprosto. Vsaka dodatna kompleksnost se odstrani takoj, ko se ugotovi (s procesom preoblikovanja).

- testiranje in **testno voden razvoj**

  Metodologija XP je vpeljala proces testno vodenega razvoja in sprotnega testiranja enot (*unit tests*). Med razvojem je ključnega pomena tudi oblikovanje in vzdrževanje nabora avtomatskih testov (t.i. regresijsko testiranje).

  Proces testno vodenega razvoja je ena od najbolj kontroverznih praks v metodologiji XP, ki ga zagovorniki opisujejo kot proces, ki **naj bi največ pripomogel k izboljšanju produktivnosti, kvalitete in števila testov**. Ocene o superiornosti izvirajo iz mnogih objavljenih študij primerov. Do sedaj **empiričnih dokazov** o tem **v literaturi ni zaslediti** – ravno to pa je tudi tema, ki jo pokrivamo v tej doktorski disertaciji.

- preoblikovanje (*refactoring*)

  Fowler [Fowler 2000] je definiral v svoji knjigi preoblikovanje kot "proces spreminjanja programskega sistema na tak način, da ne spremeni zunanjega obnašanja kode, ampak le izboljša njeno interno strukturo". Vedno, ko programer naleti na situacijo, ko mu preoblikovanje obstoječe kode lahko olajša nadaljnji razvoj novih zahtev, naj uporabi tehnike preoblikovanja.

  Preoblikovanje je tesno povezano z uporabo vzorcev in tako ne dodaja nobene nove funkcionalnosti v sistem, ampak le izboljša notranjo strukturo, berljivost in preprostost. Beck opisuje v [Beck 2000] svoj "dvojni" pristop (*two-hat approach*) k preoblikovanju, kjer **striktno loči** dodajanje nove funkcionalnosti in samo preoblikovanje kode. Ko je potrebno dodati novo funkcionalnost, se pogosto najprej preoblikuje koda - s tem pa dosežemo ravno lažje dodajanje nove funkcionalnosti. Preoblikovanje je zelo pomembno tudi zato, ker se brez stalnega preoblikovanja sistema le-ta čez čas tako poslabša, da ga ni mogoče več

vzdrževati brez zelo velikih naporov in stroškov, saj ima postopna izguba dobre notranje strukture kumulativen efekt.

Preoblikovanje je v procesu testno vodenega razvoja tako pomembno, da ga le-ta celo predpisuje kot enega od korakov pri izvedbi. Ker pri metodologiji XP in tudi pri uporabi TDD nimamo faze vnaprejšnjega podrobnega načrtovanja, je ravno proces preoblikovanje kode tisti dejavnik, ki nam sploh omogoča t.i. evolucijski in pojavljajoč design.

- programiranje v parih (*pair programming*)

  Je princip ustvarjanja nove kode s strani obeh programerjev, združenih v razvojni par. Pri tem eden uporablja računalnik, drugi pa mu asistira s stalno revizijo kode in pozornostjo na celotno sliko, ne le na trenutni detajl. Tudi testiranje poteka v dvoje. Prve raziskave na tem področju je objavila L. Williams v svoji doktorski disertaciji The Collaborative Software Process [Williams 2000]. V njej je dokazala s kontroliranim eksperimentom, da so prednosti programiranja v parih resnične in še kako otipljive – izboljša se kvaliteta, skrajša se čas izdaje sistema in zadovoljstvo ob delu v parih je za razvijalce večje [Williams+ 2000]. Poglejmo si na kratko številke: v povprečju uigran par razvijalcev konča delo do 40% hitreje kot posameznik, pri tem pa naredi bistveno manj napak (v povprečju več kot 14%). Ravno delo L. Williams je sprožilo številne ponovitve in nove raziskave o prednostih in pomanjkljivostih dela v parih.

  Lahko rečemo, da ima programiranje v parih še večjo vrednost zaradi treh dejstev:

  - Več časa kot so napake v sistemu, težje jih je najti in odpraviti. Tudi če se torej kvaliteta kode poveča le za malo, na dolgi rok prihranimo čas razvoja zaradi krajšega naknadnega iskanja napak.
  - Delo v parih omogoča stalno revizijo kode, kar izboljša fleksibilnost in preprostost arhitekture sistema. Nejasni in nefleksibilni načrti se kasneje težje spreminjajo, kar nam dolgoročno zmanjšuje kvaliteto zasnove.

  - "Stranski učinek" programiranja v parih je tudi visoka stopnja navzkrižnega učenja razvijalcev, kar nam na dolgi rok zmanjša stroške in poveča učinkovitost razvojne skupine.

- skupno lastništvo kode (*collective code ownership*)

  Vsakdo iz razvojne skupine lahko spreminja kodo, ki jo je napisal kdorkoli v skupini. Razvoj je skupinski, razvojna skupina komunicira neprestano in skrbi, da v skupnem repozitoriju ni slabe kode.

- stalna integracija (*continuous integration*)

  Razvojni cikli so kratki. Zaključene dele sistema je potrebno pogosto, včasih tudi večkrat dnevno testirati in povezati v delujočo celoto. To se naredi še preden se jo preda uporabniku kot naslednjo izdajo.

- 40 urni delovni teden (*40-hour week*)

  Projektno zasnovano delo seveda zahteva velike napore vseh sodelujočih. Zato XP že ob načrtovanju časa razvoja kot svoje omejitve računa z 8 urami dela dnevno na razvijalca. Na kratek rok se sicer da delati tudi več, vendar sodelujoči v projektu že po nekaj tednih postanejo demotivirani za delo. Zelo dobra praksa je tudi, da gredo ljudje ob koncu dneva domov brez nerešenih stvari, ki bi jim visele nad glavo. Te prakse ne smemo razumeti dobesedno, da je ravno 40 ur tedensko edina prava obremenitev. Potrebno jo je razumeti bolj v smislu, da mora biti delavnik omejen, ne pa da se sili razvijalce, da delajo dolgo časa po zelo veliko ur dnevno.

- stranka vedno prisotna (*on-site customer*)

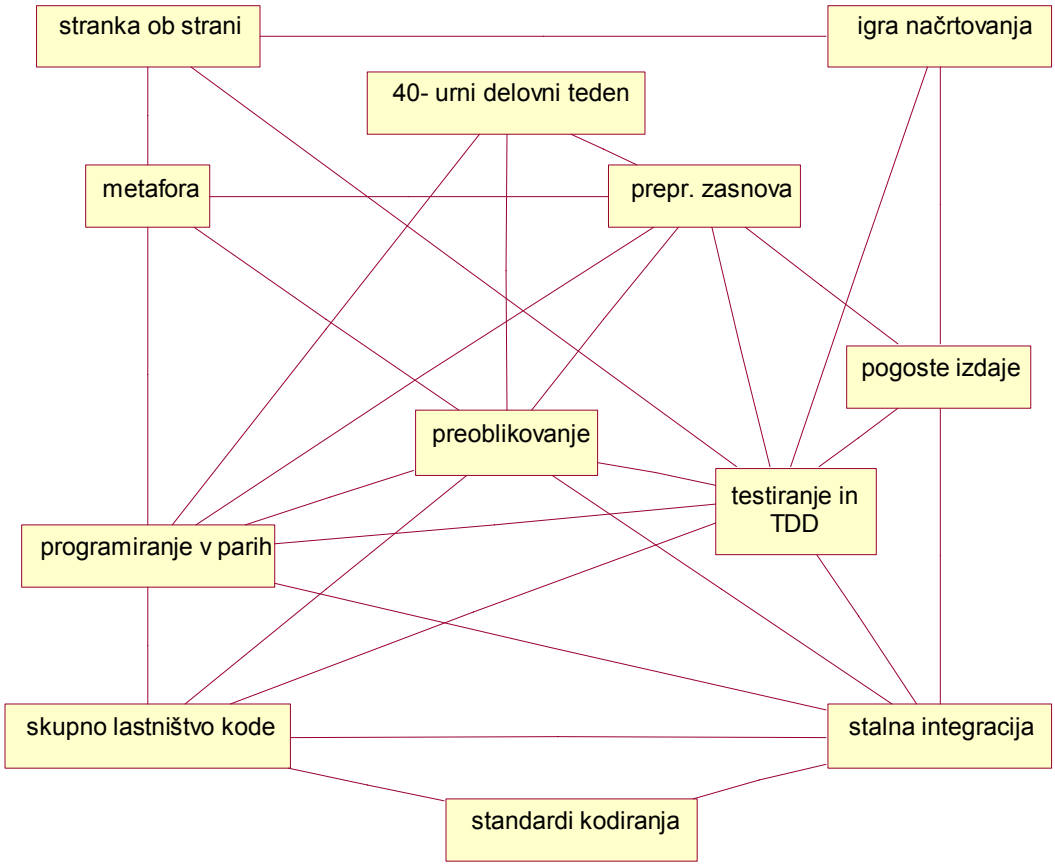
  Prisotnost akterjev iz poslovnega sveta, za katerega se sistem razvija, smo že omenili kot nenadomestljivo pridobitev v smislu izboljševanja zahtev, hitrega odziva, zmanjšanja tveganja napačno razvitega sistema, itd..

- standardi kodiranja (*coding standards*)

  Pravila kodiranja morajo biti jasna vsem programerjem, da lahko ti komunicirajo kar z izvorno kodo, ne pa s tekstovnimi opisi ali diagrami. Zelo pomembno je poznavanje vzorcev in njihova dosledna uporaba. Ravno zato, ker lahko kdorkoli

spreminja katerokoli kodo in zaradi stalnega dela v parih, so standardi kodiranja v metodologiji XP in v testno vodenem razvoju še bolj pomembni kot drugače.

Prakse se medsebojno dopolnjujejo. Slika 2-1 prikazuje njihovo medsebojno prepletanje.



Slika 2-1: Prepletanje praks v metodologiji XP

Projekt v metodologiji XP ima časovno gledano tri različne faze:

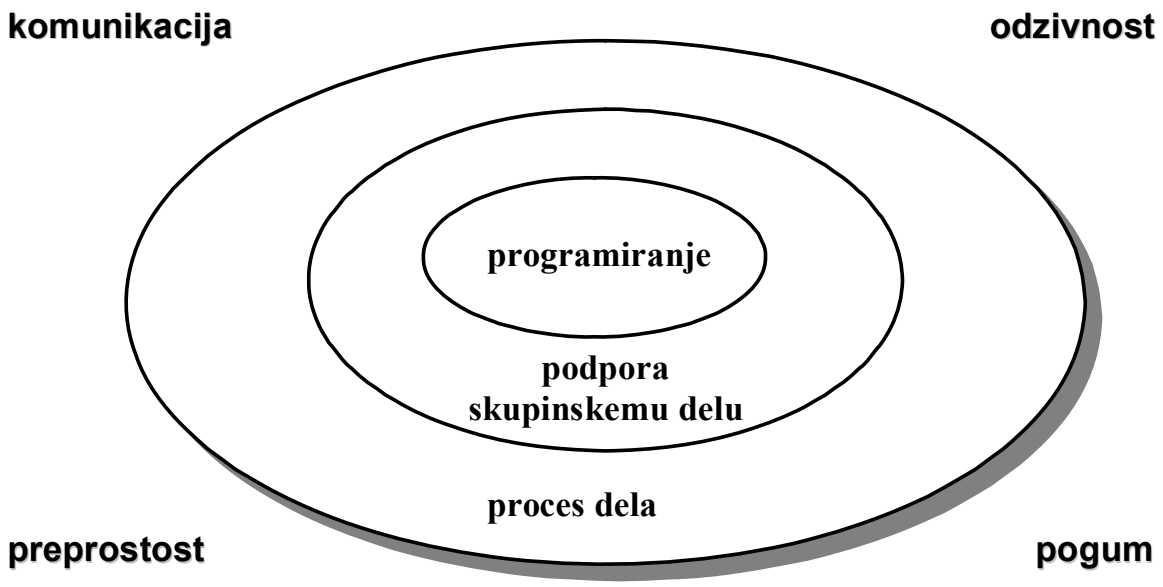
- faza planiranja izdaj (*release planning phase*): tu stranka definira funkcionalne zahteve (zgodbe), razvijalci jih ocenijo glede zahtevnosti implementacije, stranka nato določi prioritetni vrstni red njihove gradnje. Za končni izdelek je potrebno več izdaj.
- faza iteracije (*iteration phase*): stranka definira (običajno ob pomoči razvojne ekipe) sprejemne teste in odgovarja na vprašanja razvijalcev, programerji pa implementirajo zgodbe. Običajno je potrebno za eno izdajo več iteracij.

- faza izdaje oz. prevzema (*release phase*): razvijalci namestijo v tej izdaji razvit izdelek v produkcijsko okolje, stranka odobri rezultate in uporabnost izdelka

Na XP lahko gledamo tudi skozi tri plasti [Wake 2002]:

- **Notranja plast** ("programiranje"):
    - preprosta zasnova,
    - testno voden razvoj in testiranje (testi enot),
    - preoblikovanje, in
    - standardi kodiranja.
- **Srednjo plast** (podpora skupinskemu delu):
    - skupno lastništvo kode,
    - stalna integracija,
    - metafora,
    - standardi kodiranja,
    - 40 urni delovni teden,
    - programiranje v parih, in
    - majhne izdaje.
- **Zunanja plast** definira proces dela (kako razvojna skupina komunicira z stranko). V to plast štejemo:
    - stranka vedno prisotna pri razvoju (oz. tesno sodelovanje stranke v razvojnem procesu),
    - testiranje (predvsem sprejemni testi),
    - majhne izdaje, in
    - igra načrtovanja.

Nekatere prakse in procesi se nahajajo v več plasteh. Pod temi plastmi pa ležijo kot temelji štiri vrednote metodologije XP (preprostost, komunikacija, odzivnost (povratne informacije) in pogum). Slika 2-2 prikazuje razdelitev metodologije XP v tri plasti s štirimi vrednotami.



Slika 2-2: Plasti praks in vrednote kot temelji v metodologiji XP [Wake 2002]

V naši disertaciji se bomo osredotočili predvsem na samo notranje jedro metodologije XP, to je na prakse **notranje plasti -** testno voden razvoj in testiranje, preprosta zasnova, preoblikovanje in standardi kodiranja. Disertacija pa se bo osredotočila na **testno voden razvoj (TDD)**, ki ga bomo kar najbolj osamili od praks iz zgornjih dveh plasti. S tem se bomo izognili njihovemu vplivu ter ga preizkusili tudi v drugačnih kontekstih uporabe, kar je izrednega pomena **za vpeljavo procesa razvoja TDD v katero od drugih agilnih ali tradicionalnih metodologij**.

2.2 Testno voden razvoj TDD

Testno voden razvoj[6] (TDD) [Beck 2003, Beck 2001] je agilni iterativni proces razvoja, kjer razvijalec napiše avtomatizirane teste pred kodo, v kateri nato implementira določeno funkcionalnost.

Bistveno pri razumevanju testno vodenega razvoja je dejstvo, da tu ne gre (le) za testiranje, ampak tudi za načrtovanje [Beck 2001]. Vnaprejšnje definiranje testov razvijalcu pomaga pri razumevanju, definiranju in modeliranju problema.

Koraki pri izvajanju testno vodenega razvoja so:

1. groba analiza in načrtovanje zgodbe
2. napiši test, ki testira majhen košček funkcionalnosti zgodbe (t.i. opravilo - *task*)
3. zaženi test, samo ta test mora biti neuspešen[7]
4. napiši kodo za to opravilo
5. po potrebi preoblikuj kodo in/ali teste, da se znebiš potencialnih problemov (*code smells*, glej [Fowler 2000])
6. zaženi **vse** teste; če so vsi uspešni, pojdi na naslednji korak, drugače popravi kodo (korak 4)
7. če zgodba še ni končana, pojdi na 2. korak (naslednje opravilo)
8. če je zgodbe končana, pojdi na prvi korak (naslednja zgodba)

Celoten cikel treh osnovnih faz oz. aktivnosti testno vodenega razvoja (testiranje, kodiranje in preoblikovanje) se izvede in ponovi zelo hitro, v velikostnem redu od nekaj minut do 15 minut. Razmerje med testi in kodo je tipično najmanj 1:1 (torej vsaj toliko

---

[6] Pred izdajo prve samostojne knjige o TDD avtorja procesa K. Becka [Beck 2003] sta se v literaturi uporabljala tudi izraza "Test-First Development" in "Test-First Programming",

[7] Neuspešen je zato, ker še sploh ni napisana ustrezna koda. Velikokrat je tu potrebno zaradi prevajalnika prej napisati "prazno" implementacijo kode (t.i. skelet). To je npr. v Javi pri metodah, ki vračajo vrednost, kar skelet metode, ki enostavno vrača vrednost *0*, *false* ali *null*.

testne kode, kot je funkcionalne kode), po mnenju uporabnikov TDD je to razmerje v praksi velikokrat celo višje, to pomeni več testne kode kot je funkcionalne kode [eWorkshop3 2002]. Ker so testi tako prepleteni v sam proces razvoja, je nujno, da so avtomatizirani in da se izvajajo hitro. Značilno za testno voden razvoj je tudi to, da vedno poganjamo vse do sedaj napisane teste (t.i. regresijski testi), saj si s tem zagotovimo, da z dodajanjem nove funkcionalnosti ali preoblikovanjem kode nismo uvedli novih napak v obstoječo kodo. Pri testno vodenem razvoju ni samostojne faze obsežnega podrobnega vnaprejšnjega načrtovanja kode[8], ampak je načrtovanje **izrazito iterativno in agilno** (t.i. evolucijski oz. pojavljajoči se design - angl. *emergent design*). **Načrtovanje je tako implicitno vgrajeno v vse faze procesa razvoja in se izvaja neprestano**, skozi ves cikel testno vodenega razvoja.

Zagovorniki testno vodenega razvoja naštevajo več prednosti, ki naj bi jih imel proces pred ostalimi procesi. Naštejmo jih le nekaj:

- Boljše razumevanje programa.

  Razvijalci naj bi bolje razumeli programe, ker TDD spodbuja razvijalce da svojo kodo razložijo preko testov in same kode, ne pa preko opisov v dokumentaciji.

- Hitre povratne informacije.

  Ker so testi tako močno prepleteni v TDD razvoj, ponujajo razvijalcu takojšne povratne informacije glede pravilnosti kode in morebitnih napak, ki bi nastale zaradi na novo dodane funkcionalnosti kje drugje v kodi.

- Manjše število napak v kodi.

  Ker pri TDD vedno zaganjamo vse teste, se tako ne more zgoditi, da bi sprememba oziroma popravek napake v enem delu kode povzročila napako v kakšnem drugem delu. Eno od pravil v TDD pravi, da kadarkoli najdemo napako, najprej napišemo test, ki jo izpostavi, šele nato popravimo kodo. Tako ta test dodamo v zbirko testov, ki se vedno zaganjajo in takšne napake ne bomo naredili več oz. jo bomo takoj opazili.

- Nizkonivojsko načrtovanje.

[8] V agilnih krogih poznan tudi pod kratico BDUF (*big design up front*).

Testi usmerjajo celoten proces TDD razvoja. Določajo kontekst, v katerem formaliziramo naše nizkonivojske odločitve glede načrtovanja. Tako v bistvu razvijalci sami takoj preizkušajo sadove svojih odločitev nizkonivojskega načrtovanja: imena razredov in metod, katere razrede in metode je potrebno narediti, definicijo vmesnikov, kako jih uporabiti, itd. TDD razvijalci so tako prvi preizkuševalci svojih odločitev[9].

Testno voden razvoj je še najlažje pokazati na praktičnem primeru. Tudi prva knjiga o testno vodenem razvoju Test-Driven Development By Example [Beck 2003] prikaže principe TDD razvoja skozi en daljši primer. Vendar je tudi pri preprostem primeru potrebno pokazati kar nekaj kode, zato smo zaradi dolžine praktični prikaz TDD vključili v Prilogo C.

2.3 Iterativni razvoj z naknadnim testiranjem (ITL)

Če hočemo v našem kontroliranem eksperimentu osamiti komponento vnaprejšnjega pisanja testov kot edino neodvisno spremenljivko, je potrebno TDD primerjati s procesom dela, ki **ima vse njegove značilnosti**, vendar ne uporablja vnaprejšnjega testiranja. Še posebej je tu pomembna komponenta iterativen razvoj, saj lahko odsotnost le-tega v kontrolnem procesu popači rezultate eksperimenta, saj ne bo več jasno, zaradi katere komponente je morebitna razlika med procesoma nastala – zaradi vnaprejšnjega oz. naknadnega testiranja ali pa morda le zaradi odsotnosti iterativnega razvoja.

Takšen proces razvoja, ki smo ga **razvili sami na podlagi procesa razvoja TDD**, smo poimenovali **iterativni razvoj z naknadnim testiranjem** (*Iterative Test-Last Development – ITL*). Ker smo ga izpeljali iz procesa TDD, mu je tudi po korakih zelo podoben:

1. groba analiza in načrtovanje zgodbe
2. napiši majhen košček funkcionalnosti (opravilo)
3. napiši test, ki testira majhen košček funkcionalnosti zgodbe (t.i. opravilo - *task*)

[9] Američani uporabljajo za to izraz: "*Eat your own dogfood*" – to pomeni, da moraš najprej svoje odločitve sam preizkusiti in uporabiti, preden jih ponudiš ostalim.

4. zaženi vse teste, če so vsi uspešni pojdi na naslednji korak, drugače popravi kodo (korak 2)
5. po potrebi preoblikuj kodo in/ali teste, da se znebiš potencialnih problemov (*code smells*)
6. če zgodba še ni končana, pojdi na 2. korak (naslednje opravilo)
7. če je zgodbe končana, pojdi na prvi korak (naslednja zgodba)

Erdogmus in sodelavci [Erdogmus+2005] so definirali svoj kontrolni ITL proces malo drugače. Njihova osnovna enota razvoja v ITL je kar zgodba, ne opravilo. Razvijalci v njihovi kontrolni skupini so torej vedno napisali celotno zgodbo in jo potem testirali. Dokler so zgodbe zelo majhne, sta naš in njihov pristop enaka. Če pa so zgodbe malo obsežnejše, pride do razlik med njima ravno zaradi granulacije problema. Menimo, da je naš pristop bolj podoben TDD in zato bolj ustrezen za primerjavo v vseh okoliščinah, saj spodbuja razvijalce, tako kot TDD, k bolj tesnemu prepletanju kodiranja in testiranja (t.i. mikroiteracije).

S tako zasnovanim kontrolnim procesom smo dosegli to, da nismo ocenjevali vpliva prisotnosti testiranja na parametre agilnega procesa, ampak izključno **različno dinamiko testiranja** – torej ali se testira pred ali po pisanju kode – **vse ostale okoliščine so bile enake**.

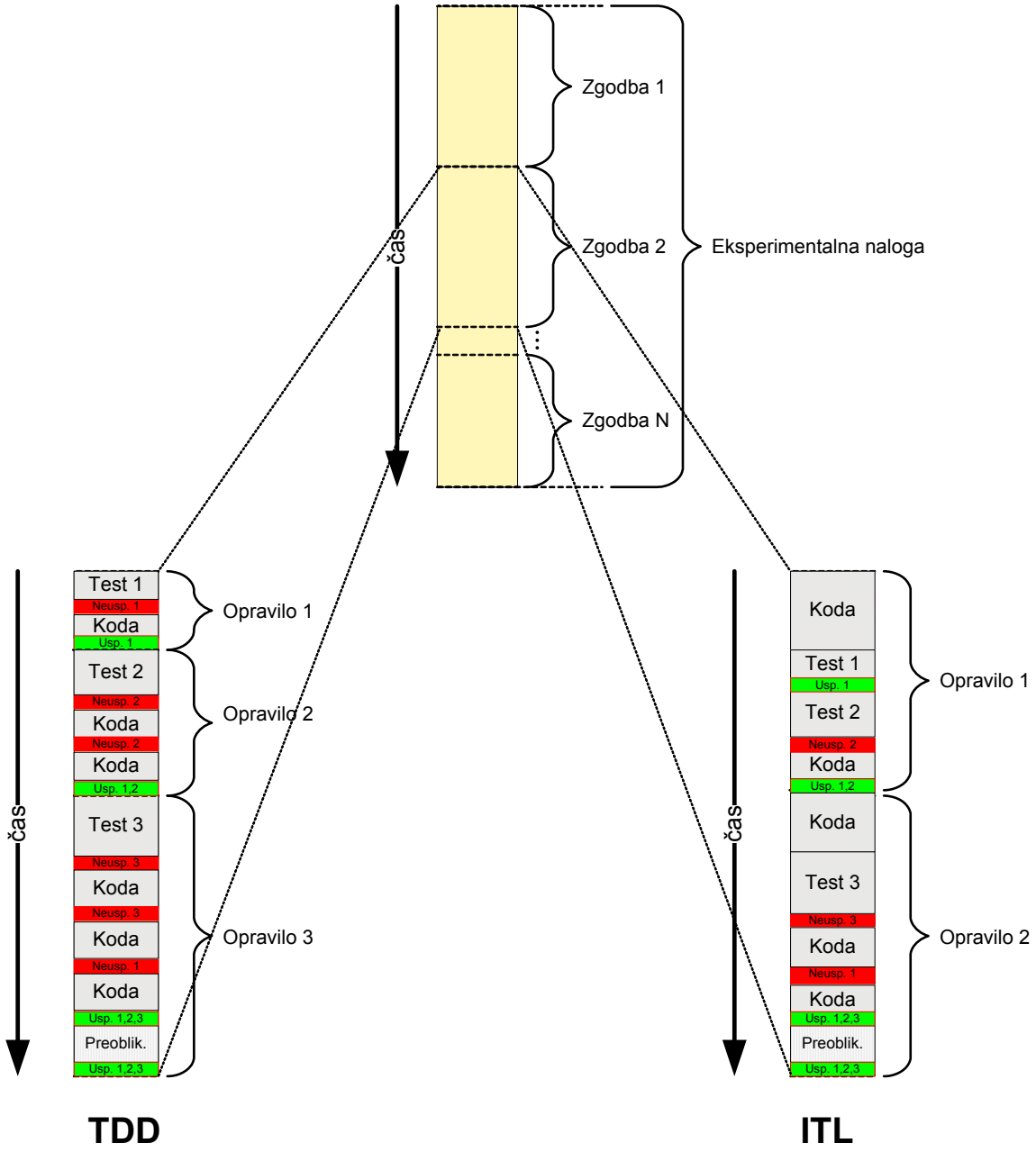
2.4 Primerjava med procesoma razvoja TDD in ITL

Procesa TDD in ITL je najbolj pregledno primerjati kar grafično (Slika 2-3).

Vidimo lahko, da je glavna razlika med procesoma TDD in ITL izključno v času pisanja testa za posamezno opravilo: pri TDD ga pišemo **pred** kodo, pri ITL **po** kodi. V vseh drugih stvareh sta procesa razvoja enakovredna, kar je bil pravzaprav tudi naš cilj pri snovanju kontrolnega procesa razvoja ITL.

Predvsem **v praksi** pa smo opazili **še eno razliko**: granulacija dela je pri procesu razvoja ITL vseeno malo večja kot pri procesu razvoja TDD. Razvijalci v ITL ponavadi kodirajo malo večje kose naloge naenkrat (opravilo je pri ITL obsežnejše kot pri TDD skupini). Predvidevamo, da pride do tega zato, ker so razvijalci navajeni kodirati daljše kose

funkcionalnosti in se morajo šele privaditi mikroiterativnemu delu, v katerega so bolj prisiljeni razvijalci v TDD. To smo tudi nakazali na sliki s tem, da smo razbili zgodbo pri procesu razvoja TDD v tri opravila, pri procesu razvoja ITL pa v dve opravili. ITL skupina na sliki je tako prvi dve opravili združila v eno. Potrebno je poudariti, da do te razlike ne pride vedno. Tudi pri ITL lahko razvijalec dela v enako velikih korakih (mikroiteracijah) kot razvijalec s TDD.



Slika 2-3: Primerjava med procesoma razvoja ITL in TDD

2.5 Sorodne raziskave

Med vsemi praksami v metodologiji XP je doslej objavljenih še največ empiričnih raziskav za prakso programiranja v parih [Williams 2000, Williams+ 2000, Succi+ 2001, Williams+ 2002, Nagappan+ 2003, Jensen 2003]. Raziskovalnih študij o procesu testno vodenega razvoja in ostalih praks metodologije XP pa je še zelo malo [Lindvall+ 2002]. Hiter pregled stanja empiričnih raziskav za področje metodologije XP so objavili Müller in sodelavci [Müller+ 2005], vendar vanj niso vključili vseh do sedaj objavljenih raziskav o TDD.

V času načrtovanja našega eksperimenta je bil objavljen le en kontroliran eksperiment o procesu razvoja TDD [Müller+ 2002], kasneje v letih 2003, 2004 in 2005 pa so bile objavljene še tri študije – ena pilotna študija [Geras+ 2004] in dva kontrolirana eksperimenta [George+ 2003, Erdogmus+ 2005]. Leta 2001 je bila objavljena ena prvih študij primera o lastnostih samega procesa razvoja TDD. Prva knjiga o testno vodenem razvoju [Beck 2003] je bila objavljena šele konec leta 2002, kar kaže ne to, da gre tu za mlado in zato neraziskano področje. Tabela 2-1 prikazuje vse objavljene kontrolirane eksperimente o testno vodenem razvoju, ki smo jih uspeli najti. Vanjo smo dodali tudi nekaj najbolj zanimivih študij primerov in ostalih raziskav, ki se tičejo obravnave naše problematike.

Müller in Hagner [Müller+ 2002] sta raziskovala testno voden razvoj v primerjavi z bolj tradicionalnim razvojnim procesom z naknadnim testiranjem. Merila sta učinkovitost programiranja, zanesljivost kode in razumevanje programa. Parametrov samega procesa pri tem nista spremljala. Kontrolno in testno skupino so sestavljali podiplomski študenti, ki so morali implementirati glavni razred grafične knjižnice. Delali so samostojno. Ugotovila sta, da testno voden razvoj v njihovem kontekstu uporabe ni prispeval ne k hitrejši implementaciji ne k večji zanesljivosti kode. So pa študentje v skupini, ki je uporabljala testno voden razvoj, v povprečju dosegali boljše rezultate pri razumevanju programa.

George in Williams [George+ 2003, George+ 2004] sta primerjala testno voden razvoj z verzijo slapovnega procesa razvoja[10], v katerem je razvoj potekal v diskretnih fazah (analiza, načrtovanje, nato implementacija (kodiranje), sledilo je testiranje). V primerjavi z agilnim pristopom je za slapovni razvoj tipična predvsem izrazita faza vnaprejšnje analize in načrtovanja rešitve. Testne osebe so bile profesionalni programerji z različnimi izkušnjami, eksperimenti so potekali v parih in v njihovem običajnem delovnem okolju. Problemska domena je bila aplikacija za spremljanje točkovanja pri kegljanju, ki je bila obdelana kot primer testno vodenega procesa razvoja v [Martin 2003]. Ugotovila sta, da so razvijalci, ki so uporabljali testno voden razvoj, napisali 18% bolj kvalitetno kodo, vendar so tudi porabili 16% več časa. Ugotovila sta tudi korelacijo med kvaliteto in časom dela, zato je možno, da je večja kvaliteta v TDD skupini nastala zaradi daljšega dela. Parametrov samega procesa nista spremljala. Težave s primerjanjem sta imela tudi zato, ker razvijalci v kontrolni skupini, ki so uporabljali slapovni proces razvoja, na koncu niso napisali testov oz. so bili testi precej pomanjkljivi, tako da je bila direktna primerjava med procesoma nemogoča.

Heričko in Rostaher [Heričko+ 2003, Rostaher+ 2002] sta se osredotočila na primerjavo dela v parih in samostojnega dela v testno vodenem razvoju. Raziskala sta predvsem čas dela in delež posameznih aktivnosti (dodajanje funkcionalnosti, testiranje, preoblikovanje) v procesu razvoja TDD. Testni subjekti so bili programerji v podjetju, ki že dalj časa uporablja metodologijo XP. Razdelili so jih v kategorije glede na izkušnje (od par mesecev do več let), ena skupina je delala samostojno, druga v parih. Ugotovili so, da v povprečju porabijo za testiranje 51%, za preoblikovanje 12% in za dodajanje funkcionalnosti (kodiranje) 37% časa razvoja. Vsaka aktivnost (cikel test-koda-preoblikovanje) je trajala v povprečju približno 5 minut, razvijalci v parih so zamenjevali vlogi voznika in navigatorja[11] povprečno 21 krat v celotnem eksperimentu, ki je trajal en delovni dan. Kode, nastale v eksperimentu, niso analizirali, osredotočili so se izključno na proces razvoja in porabljen čas. Razlike v času med kontrolno (samostojni razvijalci) in eksperimentalno skupino (pari) niso bile statistično značilne (t-test). Za beleženje faz

[10] Sama avtorja sta ga označila z "*waterfall-like process*".

[11] Voznik (*driver*) - razvijalec, ki upravlja z računalnikom; navigator – razvijalec, ki mu pri tem asistira [Beck 2000]

dela sta Heričko in Rostaher, podobno kot mi, napisala poseben samostojen program, po funkciji podoben našemu ProcessLogu (glej 3.6.2 Zbiranje podatkov o procesu razvoja na strani 53). Razlika med našim pristopom beleženja procesa dela in pristopom, ki sta ga ubrala Heričko in Rostaher, je:

- Naš program ProcessLog je popolno integriran v razvojno okolje (njihov je samostojen program, ki ga je potrebno posebej zaganjati ob razvojnem okolju).
- Mi beležimo tudi zaganjanja testov in uspešnost zagonov, ne le faze dela kot sta jih Heričko in Rostaher. Beleženje popolnoma avtomatsko izvaja naša komponenta ProcessLog, za to ni potrebna posebna intervencija razvijalca.

Maximillien in Williams [Maximillien+ 2003] sta izvedla študijo primera v podjetju IBM. V razvojni skupini programske opreme za prodajne točke v trgovini (POS) so z uvedbo TDD dosegli 50 % boljšo kvaliteto ob praktično istem nivoju produktivnosti. Na novo vpeljani TDD so primerjali s podatki o prejšnjih verzijah istega projekta, kjer so uporabljali klasičen razvoj z ad-hoc testiranjem.

Geras in sodelavci [Geras+2004] so izvedli eksperiment, s katerim so raziskovali produktivnost in kvaliteto med profesionalnimi razvijalci. Zaradi težav pri vzorčenju (prostovoljno vzorčenje) so študijo razglasili kot pilotno, torej ne gre za pravi kontrolirani eksperiment. Prostovoljce so razdelili v dve skupini, prva skupina je najprej naredila nalogo A s procesom TDD in za tem nalogo B s procesom ITL. Druga skupina pa je najprej naredila nalogo A s procesom ITL in nato nalogo B s procesom TDD. ITL proces so podobno kot mi zasnovali kot iterativen. Vsem subjektom so dali podrobne procesne navodila, narejene po vzoru procesa Personal Software Process (PSP). Razvijalci niso imeli na voljo sprejemnih testov[12], ampak so jih morali v celoti napisati sami iz tekstovnih funkcionalnih zahtev. Subjekti so delali vsak v svojem okolju in s svojimi razvojnimi orodji, uporabljati pa so morali modificirano verzijo ogrodja za testiranje JUnit, ki je štela zaganjanja testov. Rešeno nalogo so morali kot "prevedeno" kodo (*Java bytecode*) oddati preko anonimnega spletnega portala, pri čemer raziskovalci

[12] *Acceptance tests* – sprejemni testi. To so testi tipa črne škatle, ki testirajo funkcionalnost programa z uporabniškega stališča..

niso imeli dostopa do izvorne kode. Ugotovili niso nobenih večjih razlik med obema procesoma, statistične značilnosti rezultatov niso podali.

Erdogmus in sodelavci [Erdogmus+ 2005] so z dodiplomskimi študenti tretjega letnika izvedli kontroliran eksperiment, v katerem so primerjali TDD z ITL. Kontrolni ITL proces so definirali podobno kot mi (glej poglavje 2.3 na strani 29). Eksperimentalna naloga je bila enaka kot v eksperimentu Georga in Williamsa [George+ 2004], le da so jo predelali v večje število krajših zgodb. Subjekti so morali narediti kar največ zgodb (obseg dela je bil variabilen). Razvijalci niso imeli dostopa do sprejemnih testov, zgodba je bila upoštevana kot dovolj dobra za stranko takrat, ko je bilo najmanj 50% sprejemnih testov uspešnih, kar je enak kriterij, kot smo ga uporabili sami v eksperimentu s pari PP04. Ugotovili so, da je produktivnost linearno odvisna od števila testov, po drugi strani pa je TDD skupina napisala več testov kot ITL skupina. Tranzitivno so sklepali, da je zato TDD skupina bolj produktivna. Opazili so tudi, da je minimalna kvaliteta (oz. spodnji nivo kvalitete) odvisna od števila testov.

Tabela 2-1: Pregled sorodnih empiričnih raziskav na področju testno vodenega razvoja (kronološko zaporedje). Na koncu tabele so za primerjavo navedene tudi osnovne značilnosti naših eksperimentov

Študija	Tip	Subjekti	Velikost	Področje raziskave	Iter. razvoj	Kvalitativni parametri	Uporabljen prog. jezik	Obseg Dela	Statistična moč
[Müller+ 2001]	ŠP	Štud.	/	PP XP	/	Da	Java	/	/
[Müller+ 2002]	KE	Štud.	10 proti 9	TDD proti KR	Ne	Ne	Java	F	Da
[Rostaher+ 2002]	KE	Prof.	6 proti 4	PP TDD proti SR TDD***	/	Ne	Small Talk	F	Ne
[Maximillien+ 2003]	ŠP	Prof.	/	TDD proti K	/	Ne	Java	/	/
[George+2004]	KE	Prof.	6 proti 6	PP TDD proti PP K	Ne	Da	Java	F	Ne
[Geras+ 2004]	P	Prof.	7 proti 7	TDD proti ITL	Da	Ne	Java	F	Ne
[Erdogmus+ 2005]	KE	Štud.	13 proti 11	TDD proti ITL	Da	Ne	Java	V	Ne****
Naša raziskava (kontekst SR*)	KE	Štud.	14 proti 18	TDD proti ITL	Da	Da	Java	V	Da
Naša raziskava (kontekst PP*)	KE	Štud.	10 proti 9, 7 proti 9**	PP TDD proti PP ITL	Da	Da	Java	F	Da

* SR – samostojni razvijalci, PP – prog. v parih; ** prvi dve številki veljata za eksperiment leta 2003, drugi dve za leto 2004;

*** Raziskava primerja programiranje v parih in samostojno delo, fokus ni na TDD; **** Avtorji le pripomnijo, da je majhna.

Opis posameznih elementov in kratic, uporabljenih v tabeli Tabela 2-1:

- Tip – kakšen tip eksperimenta so raziskovalci izvedli:
    - KE - kontroliran eksperiment,
    - ŠP - študija primera,
    - P - pilotna študija (kvazi eksperiment) – zaradi težav pri vzorčenju ne gre za pravi kontroliran eksperiment.

  Najbolj objektivni glede rezultatov so kontrolirani eksperimenti.
- Subjekti – kdo je bil eksperimentalni subjekt v raziskavi:
    - Štud. – študentje kot subjekti raziskave,
    - Prof. – poklicni razvijalci kot subjekt raziskave.
- Velikost – velikost vzorca obeh primerjanih skupin, prva številka velja za velikost skupine, ki je uporabljala proces dela TDD.
- Področje raziskave – kaj je bil fokus raziskave:
    - XP – metodologija XP.
    - TDD – testno voden razvoj (*Test-Driven Development*).
    - PP – programiranje v parih (PP TDD pomeni programiranje v parih z uporabo TDD in analogno temu za ostale možnosti).
    - SR TDD – razvijalci programirajo samostojno (samostojni razvijalci), proces razvoja je TDD.
    - KR – klasični slapovni proces razvoja z regresijskim testiranjem na koncu. Med razvojem programa ni testiranja, je le čisto na koncu, kot zadnja faza razvoja. Testi so avtomatizirani in se zaganjajo vsi (regresijsko testiranje).
    - K – klasični slapovni proces razvoja, testiranja ni ali pa je neavtomatizirano (torej je potrebno vsak test pognati in interpretirati ročno).

- ITL – Iterativni razvoj z naknadnim testiranjem (*Iterative Test-Last Development*).

- Iter. razvoj – ali je kontrolna skupina, s katero so primerjali proces TDD, tudi uporabljala iterativni razvoj.
- Kvalitativni parametri – ali so avtorji v študiji raziskovali tudi kvalitativne prametre procesa razvoja.
- Obseg dela:
  - F – Fiksen, obseg dela je vnaprej določen z zahtevami, čas dela je neomejen.
  - V – Variabilen, obseg dela je variabilen, subjekti v eksperimentu poskušajo implementirati kar največ funkcionalnosti po specifikacijah zahtev (specifikacije se imenujejo zgodbe v terminologiji XP). Čas dela je lahko omejen ali neomejen.
- Statistična moč: Ali so avtorji raziskave v svoji statistični analizi upoštevali in objavili tudi podatke o statistični moči?

2.5.1 Sorodne kvalitativne raziskave

Poleg kvantitativnih parametrov sta George in Williams [George+ 2004] raziskovala tudi kvalitativne parametre, ki sta jih pridobila z anketo med razvijalci pred samo izvedbo eksperimenta. Zanimala ju je predvsem produktivnost, učinkovitost in težavnost osvojitve TDD. Ugotovila sta, da 80% razvijalcev meni, da je TDD učinkovit proces razvoja, 78% pa jih meni, da TDD izboljša njihovo produktivnost. Večina jih tudi meni, da je z uporabo TDD design programa preprostejši in da izostanek vnaprejšnje faze načrtovanja v procesu razvoja ni slabost. Vendar pa kar 40% razvijalcev meni, da je prehod na TDD težek in zapleten.

Müller in Tichy [Műller+ 2001] sta izvedla študijo primera razvoja z dvanajstimi študenti. Vsi so imeli predhodne izkušnje s programiranjem v Javi, delali so v dveh skupinah, vsaka je bila sestavljena iz 3 parov. Obe skupini sta uporabljali za razvoj metodologijo XP. Programirali so v Javi z uporabo ogrodja JUnit za avtomatizirane teste

enot. Najprej so napravili tri krajše in lažje naloge, s katerimi so se uvajali v samo metodologijo. Sledil je daljši in obsežnejši projekt simulacije prometa. Avtorja sta se v tej raziskavi osredotočila predvsem na kvalitativne parametre uporabljene metodologije, torej predvsem na analizo težav in prednosti, ki so si jih študenti zaznali med delom. Izvedla sta obsežno anketo s poudarkom na različnih delih metodologije XP: programiranjem v parih, iterativnim planiranjem, testno vodenim razvojem, preoblikovanjem in skalabilnostjo samega procesa razvoja.

Ugotovila sta, da so razvijalci hitro osvojili tehniko programiranja v parih, vendar je njena učinkovitost v vseh fazah razvoja vprašljiva. Tehniko pojavljajočega se (oz. evolucijskega) designa je težko osvojiti in uporabljati, prav tako kot vnaprejšnje pisanje testov pred samim kodiranjem. Ugotovila sta tudi, da metodologija XP ni dobro skalabilna, torej da jo je težko uporabljati v velikih in številčnih razvojnih skupinah. Zaradi vseh teh težav je pri uvedbi XP in TDD potrebno **posebno uvajanje razvijalcev** v same tehnike in proces dela, kar smo pri načrtovanju naših eksperimentov tudi upoštevali.

Kvalitativen pogled na problematiko odlično dopolnjuje v eksperimentu pridobljene kvantitativne ugotovitve. Če se na primer en proces izkaže kot bolj učinkovit, a se ga je po mnenju večine razvijalcev zelo težko naučiti in uporabljati, je praktična uporabnost takega procesa zmanjšana. Vsako vpeljavo takega procesa v razvojno skupino je zato potrebno vnaprej načrtovati in jo podpreti z ustreznimi aktivnostmi, kot so na primer daljše izobraževanje, vpeljava praks za nadzor nad razvojnim procesom, itd. Zato smo tudi sami ob zaključku eksperimenta zbrali in analizirali različne kvalitativne parametre.

2.5.2 Kritika in primerjava dosedanjih kontroliranih eksperimentov

Vse do sedaj objavljeni študije **kontroliranih eksperimentov** s fokusom na TDD imajo določene pomanjkljivosti, ki zmanjšujejo uporabnost in pravilnost njihovih rezultatov.

Müller in Hagner [Müller+ 2002] sta izvedla kontroliran eksperiment, v katerem sta primerjala TDD s klasičnim slapovnim razvojem z regresijskim testiranjem na koncu razvoja. Pri tem se zastavlja vprašanje, ali ni na tako dobljene rezultate vplival tudi mikroiterativen razvoj pri TDD skupini, medtem ko je kontrolna skupina delala teste

izključno na koncu, šele po tem, ko so implementirali **vso** funkcionalnost. Po našem mnenju je slabost njunega eksperimenta ravno ta, da tudi kontrolna skupina ni uporabljala iterativnega pristopa. S tem sta avtorja vpeljala v eksperiment še eno prikrito neodvisno spremenljivko – izrazito iterativen razvoj, ki je prisoten le v TDD skupini, ne pa tudi v kontrolni skupini. Podobno kritiko tega eksperimenta so podali tudi ostali avtorji [Geras+ 2004, Erdogmus+ 2005]. Müller in Hagner tudi poročata o relativno majhni statistični moči njunih testov.

George in Williams [George+ 2004] sta TDD primerjala s klasičnim slapovnim razvojem, kjer razvijalci v kontrolni skupini na koncu kodiranja niso pisali testov. S tem sta, podobno kot Müller in Hagner, v eksperiment vpeljala še dve dodatni prikriti neodvisni spremenljivki, iterativen razvoj in pomanjkanje faze testiranja. Njun vzorec je majhen (le 6 proti 6), prav tako nista kontrolirala spremenljivke okolja – razvijalci so delali z različnimi razvojnimi orodji. Med eksperimentom sta tudi spreminjala pogoje dela (po težavah v prvi skupini sta spremenila eksperimentalna navodila za ostali dve skupini).

Geras in sodelavci [Geras+2004] so svojo študijo zaradi izrazitih slabosti pri vzorčenju sami predstavili kot pilotno študijo oz. kvazi eksperiment. Vzorčenje, ki so ga uporabili, je bilo prostovoljno, pestila ga je tudi zelo visoka smrtnost. Druga skupina je potrebovala več vodenja kot prva, zato je zelo verjetno, da tudi pogoji dela niso bili enaki za obe skupini. Avtorji so se tudi odločili za fečfaktorski tip eksperimenta, ki je po našem mnenju manj primeren predvsem zaradi težavnosti osvojitve procesa TDD (glej razpravo v poglavju 3.3 Tip eksperimenta na strani 48).

V študiji Erdogmusa in sodelavcev [Erdogmus+ 2005] lahko prav tako najdemo več pomanjkljivosti. Avtorji niso upoštevali in objavili statistične moči testov (objavijo le pripombo, da je majhna), probleme so imeli tudi zaradi zelo visoke smrtnosti (angl. *mortality rate*). Vendar lahko kot **glavno pomanjkljivost njihove študije navedemo dejstvo, da so po našem mnenju premalo kritično presojali, katere meritve so sploh veljavne**. Tako v svojem članku napišejo, da: "*Dva subjekta v kontrolni skupini nista napisala nobenih testov.*". Zastavlja se vprašanje o veljavnosti takšnih dveh meritev: ITL

proces namreč jasno zapoveduje prepletanje testiranja in kodiranja[13]! Zato menimo, da bi Erdogmus in sodelavci morali te dve meritvi iz ITL skupine **izključiti**. S tem bi po njunih besedah razlika med obema skupinama v številu testov padla na iz 52% na 28%, še vedno v korist TDD skupine. Problem nastane, ker so večjo produktivnost v TDD skupini dokazali posredno, preko števila testov: večje kot je število testov, večja je produktivnost; TDD skupina ima večje število testov kot ITL skupina, torej je TDD skupina bolj produktivna. Ker so pri metriki število testov rezultati statistične značilnosti razlik povprečne vrednosti zelo blizu izbrane meje za značilnost (p=0,09), bi ta razlika v povprečni vrednosti po izključitvi teh dveh meritev najbrž ne bi bila več znotraj meje α=0,1. S tem pa se podre tudi njihova ugotovitev o boljši produktivnosti pri TDD skupini zaradi večjega števila testov!

Prav tako je njihova definicija ITL procesa zaradi večje privzete granulacije (uporabijo nivo zgodbe ne nivo dela zgodbe (opravilo), glej poglavje 2.3 na strani 29) manj primerna za vse situacije in še posebej za daljše, bolj kompleksne zgodbe.

Kritiko sorodnih raziskav lahko strnemo predvsem na:

- probleme neupoštevanja oz. neobjavljanja statistične moči v analizi,
- probleme pri vzorčenju,
- le eno ponovitev eksperimenta (ni replikacij),
- le en kontekst uporabe in ena eksperimentalna naloga,
- nekritičnost pri preverjanju, ali so vsi rezultati meritev sploh veljavni (ali je bila stopnja skladnosti z dodeljenim procesom razvoja dovolj velika),
- nepravilnost definiranja eksperimentalnega kontrolnega procesa (potrebna je izločitev iteracij kot prikrite spremenljivke).

Sami smo se v naši študiji izognili vsem naštetim pomanjkljivostim.

[13] Sami smo takšne meritve izločili.

3 NAČRT IN IZVEDBA EKSPERIMENTA

Pri objavi rezultatov empiričnih študij se je dobro držati vnaprej začrtanih smernic, ki bodo omogočale tudi drugim raziskovalcem, da lahko čim bolj koristno uporabijo rezultate raziskave pri svojem nadaljnjem delu. Še posebej so na področju empiričnih raziskav zelo zaželene in potrebne **replikacije eksperimenta**[14]. Miller [Miller 2005] zato predlaga uporabo enega od uveljavljenih modelov (smernic) za opis eksperimenta na področju programskega inženirstva. Predlaga npr. model avtorjev Lotta in Rombacha [Lott+ 1996] ali pa nekaj manj podrobno razčlenjen model, ki so ga razvili Basili in sodelavci [Basili+ 1986].

Sami smo se po tehtnem preudarku odločili, da bomo namesto razvoja svojega modela za izvajanje replikacij eksperimenta uporabili preizkušen model opisa eksperimentov avtorjev Lotta in Rombacha. Posamezne točke smo spremenili in premešali v modelu le v tolikšni meri, da je oblika teksta bolj primerna za doktorsko disertacijo, ohranili pa smo vsebino točk, ki jih navajata avtorja. S tem smo postavili temelje za celo družino sorodnih eksperimentov [Basili+ 1999].

3.1 Cilji

Cilj raziskave je ugotoviti, če (in v kakšni meri) vnaprejšnje pisanje testov vpliva na parametre agilnega razvojnega procesa in v njem nastalega izdelka (programa). Predvsem nas zanimajo trije parametri:

- **produktivnost,**

[14] Oziroma celo, kakor smo že omenili, kakršno koli empirično preverjanje teorij, kar v praksi velja še posebej na področju programskega inženirstva [Tichy 1998].

- **kvaliteta,**
- **testiranje** (količina in kvaliteta testov).

Ocenili bi radi tudi druge parametre, ki okarakterizirajo posamezen proces dela: koliko časa razvijalci prebijejo v posamezni fazi procesa razvoja, kako pogosto poganjajo teste, kolikšen je delež uspešnih testov in kakšno je količinsko razmerje med testi in kodo.

Zgoraj zbrane kvalitativne parametre procesa razvoja želimo ovrednotiti tudi skozi **kvalitativna opažanja** razvijalcev, ki ga uporabljajo. Med temi kvalitativnimi parametri nas zanima predvsem pogled razvijalcev na produktivnost, učinkovitost in kvaliteto ter težavnost osvojitve posameznega procesa razvoja.

3.2 Eksperimentalne spremenljivke, teorije in hipoteze

Edina neodvisna spremenljivka v eksperimentu je proces razvoja (torej pisanje testov pred ali po kodi). Posebej smo pazili, da ne bi v eksperiment vpeljali kakšnih skritih neodvisnih spremenljivk, ki bi vplivale na rezultate: subjekti so poslušali enaka predavanja, imeli enake naloge, enaka orodja, enake pogoje dela, uporaba iterativnega pristopa, itd. Vse te odvisne spremenljivke tvorijo **kontekst, v katerem izvajamo eksperiment**. Eksperiment smo izvedli v dveh različnih kontekstih, ki dobro pokrijeta morebitno uporabo TDD-ja v praksi:

- V kontekstu dela v parih, ki imitira pogoje dela, kot jih običajno ima proces razvoja TDD znotraj metodologije XP.
- V kontekstu samostojnih razvijalcev, ki imitira vpeljavo procesa razvoja TDD v kakšno od drugih agilnih metodologij (pa tudi tradicionalnih!), kjer razvijalci delajo samostojno, brez dosegljivih natančnih sprejemnih testov, ki bi fiksirali kvaliteto.

Podrobnejše oba konteksta definiramo v poglavju 3.5.

Že v opisu TDD procesa smo zapisali nekatere v literaturi omenjene trditve o domnevni superiornosti procesa razvoja TDD glede tako produktivnosti kot tudi kvalitete kode. Vendar do sedaj razen mnogih študij primerov še ni dokazov ne o obstoju takšnih pozitivnih učinkov ne o tem, da je za to zaslužen ravno proces razvoja TDD.

Glavne odvisne spremenljivke (parametri), ki jih bomo raziskovali, so produktivnost, kvaliteta in testi. V delu raziskave o procesu razvoja pa bomo poskušali še preveriti nekaj hipotez, ki se pojavljajo v literaturi in se nanašajo na lastnosti samega procesa razvoja TDD.

Znani so dolgoročni pozitivni učinki testiranja na produktivnost v tradicionalnih metodologijah, kjer za teste skrbi posebna, ločena razvojna skupina (*Quality Assurance Team*). Čeprav pomeni testiranje dodatno delo na krajši rok, se na daljši rok vložek časa več kot povrne v obliki **veliko manjše izgube časa, porabljenega za iskanje napak**. Zagovorniki procesa razvoja TDD pa gredo s svojimi trditvami celo korak dalje. Zaradi tesne prepletenosti vnaprejšnjega testiranja in kodiranja naj bi bili v TDD prisotni tudi ugodni kratkoročni učinki na produktivnost. Ravno ta tesna povezanost s testi naj bi omogočala razvijalcu v TDD boljše osredotočanje na problem in hitre povratne informacije, zaradi katerih naj bi bila produktivnost večja tudi na krajši rok. Zato pričakujemo, da bo **produktivnost razvijalcev v TDD večja kot produktivnost razvijalcev v ITL**, kar smo formalizirali v hipotezi PROD:

$$\text{PROD: } \text{PROD}_{\text{TDD}} > \text{PROD}_{\text{ITL}}$$

Učinkovito in podrobno testiranje je kritično za izboljšanje kvalitete. V TDD je proces testiranja tako prepleten, da lahko pričakujemo, da bodo programi, narejeni v **procesu razvoja TDD boljše kvalitete**. Potrebno je ločiti dve vrsti kvalitete: zunanjo in notranjo. Zunanja kvaliteta se odraža navzven, proti stranki, notranja pa navznoter, v strukturo programa. Stranka ob delu s programom neposredno občuti le učinke zunanje kvalitete (torej napake v kodi), skrita pa mu je morebitna slaba notranja kvaliteta. Hipotezi o superiornosti zunanje in notranje kvalitete kode, narejene s procesom razvoja TDD, smo formalizirali v hipotezah KVALZ za zunanjo kvaliteto in KVALN za notranjo kvaliteto:

$$\text{KVALZ: } \text{KVALZ}_{\text{TDD}} > \text{KVALZ}_{\text{ITL}}$$

$$\text{KVALN: } \text{KVALN}_{\text{TDD}} > \text{KVALN}_{\text{ITL}}$$

Ker je testiranje tako tesno prepleteno v proces razvoja, lahko pričakujemo, da bodo **razvijalci v procesu TDD napisali več testov** kot razvijalci, ki uporabljajo katero od drugih metodologij ali procesov. To našo predpostavko smo formalizirali v hipotezi TESTŠT:

$$\text{TESTŠT: } \text{TESTŠT}_{\text{TDD}} > \text{TESTŠT}_{\text{ITL}}$$

Vendar je tu potrebno biti previden. Večje število testov še ne pomeni tudi njihove večje kvalitete. Dober programer lahko z manjšim številom testov pokrije celo več kode, kot kateri od manj izkušenih kolegov, torej je kvaliteta njegovih testov boljša. Ker razvijalci v TDD teste pišejo **pred kodo**, lahko pričakujemo, da bodo zato **testi boljše kvalitete** kot tisti, ki jih razvijalci napišejo naknadno. Formalizirajmo to ugotovitev v hipotezo TESTKV:

$$\text{TESTKV: } \text{TESTKV}_{\text{TDD}} > \text{TESTKV}_{\text{ITL}}$$

S tem smo formalizirali naše hipoteze o treh najpomembnejših parametrih. V literaturi pa lahko zasledimo tudi nekatere predpostavke oz. trditve, ki se nanašajo na **lastnosti samega procesa razvoja TDD**.

Proces razvoja TDD naj bi spodbujal razvijalce k temu, da napišejo večje število testov (glej hipotezo TESTŠT), zato je smiselno tudi pričakovati, da bodo **razvijalci v TDD večkrat zagnali teste**, kot razvijalci, ki uporabljajo kakšen drug proces razvoja. To predpostavko o večji **frekvenci poganjanja testov** v TDD smo formalizirali v hipotezi TESTFR:

$$\text{TESTFR: } \text{TESTFR}_{\text{TDD}} > \text{TESTFR}_{\text{ITL}}$$

Ker razvijalci v TDD vsak na novo napisan test tudi najprej preizkusijo tako, da ga poženejo še preden je zanj napisana koda (test mora biti neuspešen!), lahko sklepamo, da bo **odstotek uspešnih testov pri TDD skupini manjši** kot odstotek uspešnih testov pri ITL skupini. Tej predpostavki v prid govori tudi dejstvo, da razvijalci v ITL skupini nevede načrtno pišejo teste, ki naj bi bili uspešni, saj so "okuženi" z natančnim poznavanjem delovanja kode. Zato tudi tradicionalne metodologije večinoma za testiranje uporabljajo samostojno ekipo, t.j. ekipo, ki kodo presoja bolj objektivno. Formalizacija ugotovitve sledi v hipotezi TESTOK:

$$\text{TESTOK: } \text{TESTOK}_{\text{TDD}} < \text{TESTOK}_{\text{ITL}}$$

Odstotek časa, ki ga razvijalci prebijejo v posamezni od treh faz proces razvoja (kodiranje, testiranje, preoblikovanje), je bolj trd oreh. O tem nismo našli veliko indicev, ki bi jih lahko uporabili kot predpostavke, na katerih bi zgradili hipotezo. Edino raziskavo o porabljenem času v posamezni fazi razvoja sta naredila Heričko in Rostaher [Heričko+ 2003, Rostaher+ 2002]. Ugotovila sta, da razvijalci v TDD povprečno prebijejo 51% časa v fazi testiranja, 37% v fazi kodiranja in 12% v fazi preoblikovanja.

Zato smo se osredotočili na **iskanje razlike med procesoma v odstotku prebitega časa v posamezni fazi**. Hipotezo (v bistvu tri, za vsako fazo posebej) smo formalizirali takole:

$$\text{ČASKTP: ČASKTP}_{TDD} \neq \text{ČASKTP}_{ITL}$$

Več virov trdi, da razvijalci s procesom TDD napišejo vsaj toliko testne kode, kot napišejo produkcijske. To razmerje med testi in kodo naj bi bilo pri TDD najmanj 1:1 [Beck 2003], velikokrat je v praksi celo višje [eWorkshop3 2002]. Konkretno razmerje med testi in kodo je verjetno še najbolj odvisno od posameznega razvijalca: bolj izkušeni razvijalci napišejo v povprečju več testov (in s tem posledično tudi več testne kode), manj izkušeni razvijalci pa napišejo manj testov (in s tem v povprečju tudi manj testne kode). Št. testov in količina testne kode sta v korelaciji, vendar lahko bolj izkušen razvijalec uporabi tudi bolj učinkovite testne primere in s tem napiše manj testne kode. Iščemo torej razlike v razmerju med testi in kodo pri obeh procesih razvoja, saj za ITL ni znanih ocen tega razmerja, zato tudi ne vemo, ali bi bilo to razmerje večje ali manjše od tistega pri TDD. Formalizirajmo predpostavke v hipotezi RAZMTK:

$$\text{RAZMTK: RAZMTK}_{TDD} \neq \text{RAZMTK}_{ITL}$$

Tabela 3-1 pregledno prikazuje vse naše formalizirane hipoteze, skupaj z ničelno hipotezo, ki jo potrebujemo pri klasičnem testiranju veljavnosti hipotez.

Tabela 3-1: Formalizirane hipoteze- povzetek

Opazovani parameter	**Ime hipoteze**	**Ničelna hipoteza H_0**	**Alternativna hipoteza H_a**
Produk-tivnost	PROD	$\text{PROD}_{TDD} = \text{PROD}_{ITL}$	$\text{PROD}_{TDD} > \text{PROD}_{ITL}$
Kvaliteta	KVALZ	$\text{KVALZ}_{TDD} = \text{KVALZ}_{ITL}$	$\text{KVALZ}_{TDD} > \text{KVALZ}_{ITL}$
	KVALN	$\text{KVALN}_{TDD} = \text{KVALN}_{ITL}$	$\text{KVALN}_{TDD} > \text{KVALN}_{ITL}$
Testi	TESTŠT	$\text{TESTŠT}_{TDD} = \text{TESTŠT}_{ITL}$	$\text{TESTŠT}_{TDD} > \text{TESTŠT}_{ITL}$
	TESTKV	$\text{TESTKV}_{TDD} = \text{TESTKV}_{ITL}$	$\text{TESTKV}_{TDD} > \text{TESTKV}_{ITL}$
Proces razvoja	TESTFR	$\text{TESTFR}_{TDD} = \text{TESTFR}_{ITL}$	$\text{TESTFR}_{TDD} > \text{TESTFR}_{ITL}$
	TESTOK	$\text{TESTOK}_{TDD} = \text{TESTOK}_{ITL}$	$\text{TESTOK}_{TDD} < \text{TESTOK}_{ITL}$
	ČASKTP	$\text{ČASKTP}_{TDD} = \text{ČASKTP}_{ITL}$	$\text{ČASKTP}_{TDD} \neq \text{ČASKTP}_{ITL}$
	RAZMTK	$\text{RAZMTK}_{TDD} = \text{RAZMTK}_{ITL}$	$\text{RAZMTK}_{TDD} \neq \text{RAZMTK}_{ITL}$

Konkretne metrike, ki smo jih uporabili za merjenje posameznega parametra, smo predstavili v poglavju *4 Analiza meritev in rezultati*.

3.3 Tip eksperimenta

Tip eksperimenta, za katerega smo se odločili, spada v kategorijo standardnih eksperimentov z enim faktorjem in dvema postopkoma[15] [Wohlin+ 2001]. Vsak postopek ustreza enemu od uporabljenih procesov dela: TDD (eksperimentalna skupine) in ITL (kontrolna skupina).

Namenoma se nismo odločili za večfaktorski tip ekspermienta, kot so ga npr. kasneje izvedli Geras in sodelavci [Geras+ 2004] in je opisan v poglavju Sorodne raziskave. Po našem mnenju bi subjekte v eksperimentu motilo, da bi morali enkrat testirati izključno pred in enkrat izključno po kodiranju. Čeprav sta procesa razvoja na videz dokaj podobna, pa zahteva osvojitev TDD nekakšen "miselni preskok" in kar nekaj vaje in treninga, o čemer poročajo tudi drugi avtorji npr. [Müller+ 2001, Mugridge 2003, Mugridge+ 2003]. Zato smo bili mnenja, da bi lahko bila "menjava konteksta" med dvema nalogama z različnima procesoma razvoja preveč moteča in bi bila s tem verjetnost popačenja rezultatov bistveno večja. Tu je potrebno še enkrat poudariti, da smo ravno zaradi tega, ker več avtorjev poroča o težavnosti pri osvojitvi procesa razvoja, veliko časa posvetili pravilnemu treningu in učenju na napakah.

3.4 Subjekti

Subjekti raziskave so bili študentje četrtega letnika univerzitetnega študija računalništva in informatike na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Ker gre za študente, ki so že blizu konca študija, je moč pričakovati, da je njihovo znanje primerljivo z znanjem manj izkušenih profesionalnih programerjev. Ker je na trgu prisotno pomanjkanje kadra v računalništvu, veliko podjetij zaposluje na razvijalska

[15] "*Single-factor, posttest-only, inter-subject design*" oz. "*standard design with one factor and two treatments*"

delovna mesta tudi kadre iz drugih tehničnih profilov (elektrotehniki, strojniki, …). Zato je mogoče pričakovati, da so študenti univerzitetnega študija računalništva in informatike tik pred koncem študija po teoretični podkovanosti celo bolj izobraženi od povprečnega manj izkušenega razvijalca v industriji[16].

Študenti so tako že v nižjih letnikih poslušali predavanja iz programiranja (tudi iz Jave), zato smo jim tudi lažje predstavili naprednejše tehnike programiranja, ki se uporabljajo v TDD in ITL procesu razvoja (preoblikovanje, testiranje enot, itd.).

Veliko študentov v višjih letnikih tudi že honorarno dela v podjetjih, zato je pričakovati, da se bodo ti študentje zaradi prakse bolje odrezali od vrstnikov, ki take prakse nimajo. Ob preverjanju uravnoteženosti obeh eksperimentalnih skupin smo te izkušnje tudi upoštevali, skupaj s povprečno oceno študija, znanjem programiranja, objektno orientiranega razvoja, Jave in mesecev izkušenj v programiranju v Javi[17].

Statistično značilnih razlik med skupinama nismo zaznali pri nobenem od parametrov usposobljenosti (Mann-Whitney U test, α=0,1). Edini sum o pristranosti zaradi smrtnosti se je pojavil le pri eni od skupin (ITL pri eksperimentu samostojnih razvijalcev iz leta 2003, podrobnejša razlaga je v Prilogi A).

3.5 Eksperimentalne naloge

Tema nalog je bila iz področja porazdeljenih sistemov in omrežnega programiranja. Za vsak eksperiment smo določili drugačno nalogo, ker se študentje navadno pozanimajo od prejšnje generacije, kaj so delali pri predmetu in bi zato imeli ob enakih nalogah prednost. Meritve tako ne bi bile veljavne.

V eksperimentu s pari leta 2003 (*PP03*) je bila naloga implementirati preprosto porazdeljeno podatkovno bazo z vgrajenim mehanizmom za avtomatsko replikacijo podatkov med njimi. Več (poljubno število) računalnikov (vozlišč) v omrežju ima vsak več svojih lokalnih podatkovnih baz, na katere se lahko "naročijo" tudi ostala vozlišča. Uporabnik poljubnega vozlišča lahko dodaja, briše in spreminja podatke v lokalni

[16] Glej tudi razpravo v poglavju 3.7.2 Eksterna veljavnost.

[17] Natančnejši pregled deljenja v skupine je v poglavju 3.10.2 Izvedba eksperimenta na strani 65.

podatkovni bazi, le-ti pa se morajo po vsakem takšnem ukazu ustrezno replicirati na vse ostale na to bazo naročene lokacije. Ko se vozlišče prvič priključi na eno od baz, dobi trenutno stanje te podatkovne baze in se takoj vključi v sistem porazdeljenega repliciranja. Ker je lahko naenkrat aktivnih več baz, je bilo potrebno strežnik napisati večnitno, da je lahko stregel hkrati več logičnim odjemalcem (podatkovnim bazam na drugih lokacijah v omrežju) naenkrat. Naloga je bila sestavljena iz desetih zgodb (glej prilogo B), ki so jih morali implementirati po vrsti. Napisali smo tudi podrobne sprejemne teste (več kot 700 preverb).

Eksperiment s pari leta 2004 (*PP04*) je definiral naslednjo nalogo: napišite večnitni strežnik, ki omogoča kramljanje uporabnikov med seboj (angl. *chat server),* podoben znanim IRC strežnikom. Strežniki se lahko tudi povezujejo med seboj v mrežo, vsak s svojimi uporabniki. Strežnik naj omogoči uporabnikom izmenjevanje datotek, ki jih vsak uporabnik prispeva v skupno rabo (preprost P2P sistem). Naloga je bila sestavljena iz trinajstih zgodb (glej prilogo B), ki so jih morali implementirati po vrsti. Vsaka od zgodb je imela predpisane podrobne sprejemne teste (več kot 1100 preverb).

Naloge v eksperimentu s samostojnimi razvijalci so morale biti zaradi na 4 ure omejenega časa izdelave precej enostavnejše od nalog v eksperimentu s pari. Odločili smo se za le dve zgodbi, pri čemer je druga nadgradnja prve, zato so morali subjekti najprej narediti prvo zgodbo in se šele nato lotiti druge (inkrementalni razvoj).

Pri eksperimentu s samostojnimi razvijalci leta 2003 (*SR03*) so ti morali implementirati števec nezakomentiranih vrstic v poljubni Javanski izvorni kodi, v drugi zgodbi pa prej razvito rešitev izpostaviti kot omrežno storitev (izgradnja večnitnega strežnika, ki po preprostem protokolu izračunava število nezakomentiranih vrstic za več odjemalcev hkrati). Razvijalcem smo že na začetku eksperimenta dali minimalne sprejemne teste, katere so morali uporabiti kot vodilo, kdaj lahko gredo v razvoj naslednje naloge. Minimalni sprejemni testi so vsebovali en povprečen scenarij uporabe kode. Uspešni minimalni sprejemni testi so zagotavljali **vsaj** približno **50% zunanjo kvaliteto**, ki smo jo naknadno izmerili s celotnim podrobnim naborom naših sprejemnih testov. Namen teh minimalnih sprejemnih testov ni bil v fiksiranje zunanje kvalitete (kot je to primer pri eksperimentu s pari!), saj so bili za kaj takega premalo podrobni. Razvijalci naj bi jih le uporabljali kot vodilo za določanje minimalne zunanje kvalitete, zato so bili na začetku

eksperimenta opozorjeni, da so to le minimalni testi in da bomo kasneje njihove rešitve ocenili s celotnim naborom sprejemnih testov.

Rešitev z vnaprej dostopnimi minimalnimi sprejemnimi testi se je izkazala za slabo. V analizi smo ugotovili, da so kar 4 razvijalci v ITL skupini uporabljali te minimalne teste kot nadomestilo za pisanje svojih. Tako sami niso napisali nobenih omembe vrednih testov, svoje rešitve so testirali le s temi minimalnimi sprejemnimi testi. Iterativni razvoj brez sprotnega testiranja pa ni v skladu s procesom dela ITL, zato smo takšne meritve izločili. Iz tega lahko sklepamo, da **razvijalci pod časovnim pritiskom najprej opustijo testiranje, kar še posebej velja za proces razvoja ITL.** V replikaciji eksperimenta naslednje leto tako študentom nismo dali nobenih sprejemnih testov in se s tem izognili problemu "ubiranja bližnjic", kar bi imelo za posledico večjo smrtnost in neveljavnost meritev zaradi neskladnosti s procesom dela.

Samostojni razvijalci v eksperimentu 04 (*SR04*) so morali v prvi zgodbi implementirati komponento za delo z ulomki (preračunavanje, krajšanje, aritmetične operacije, izgradnja ulomkov iz niza znakov, izpis, …), nato pa v drugi zgodbi to rešitev izpostaviti kot omrežno storitev po danem protokolu (večniten strežnik za več hkratnih odjemalcev). Glede na slabe izkušnje zaradi smrtnosti tokrat eksperimentalni subjekti minimalnih sprejemnih testov **niso dobili** ob izgradnji rešitve. Kot oceno, kdaj zgodba dosega minimalne zahteve, pa smo vzeli mejo nad 50% uspešnih sprejemnih testov za posamezno zgodbo, kar je logično gledano enak kriterij kot smo ga v eksperimentu l. 2003 (SR03) zagotavljali z minimalnimi sprejemnimi testi. Enako rešitev (minimum 50% sprejemnih testov) so uporabili tudi Erdogmus in sodelavci [Erdogmus+ 2005].

Povzemimo **glavne značilnosti eksperimentalnih nalog**:

- Kontekst samostojnih razvijalcev (SR03, SR04):
    - omejen čas dela (4 ure),
    - variabilen obseg dela (narediti kar največ zgodb v določenem časovnem obdobju),
    - inkrementalen razvoj (naslednja zgodba nadgrajuje prejšnje),
    - krajša eksperimentalna naloga (dve zgodbi),

- eksperiment leta 2003 (SR03) – minimalni sprejemni testi dostopni razvijalcem,
- eksperiment leta 2004 (SR04) – sprejemni testi niso dostopni razvijalcem.

- Kontekst dela v parih (PP03, PP04):
  - neomejen čas dela (v povprečju so študentje porabili cca. 26 ur),
  - fiksen obseg dela (narediti vse zgodbe ob vnaprej predpisani zunanji kvaliteti – podrobni sprejemni testi),
  - inkrementalen razvoj (naslednja zgodba nadgrajuje ali celo spreminja prejšnje),
  - daljša in bolj kompleksna eksperimentalna naloga,
  - podrobni sprejemni testi dostopni razvijalcem, zunanja kvaliteta je zato konstantna. Zunanji testi so bili napisani v ogrodju FIT (http://fit.c2.com), ki postaja de facto standard za pisanje sprejemnih testov, tako kot JUnit za teste enot.

3.6 Zbiranje podatkov

Podatke, ki smo jih v eksperimentu zbirali, lahko v grobem razdelimo v dve kategoriji:

- podatki o verzijah izvorne kode,
- podatki o procesu dela.

Prva kategorija so podatki o izvorni kodi in vseh njenih verzijah od začetka do konca. Verzije morajo biti časovno označene, tako da se da rekonstruirati potek dela za vsakega razvijalca od začetka do konca razvoja. Druga kategorija pa se tiče karakteristik samega procesa razvoja. Verzije izvorne kode nam ne dajejo popolnih podatkov o tem, v kateri fazi procesa je trenutno razvijalec, kdaj je zagnal teste in kakšen je bil njihov rezultat.

3.6.1 Vodenje verzij kode

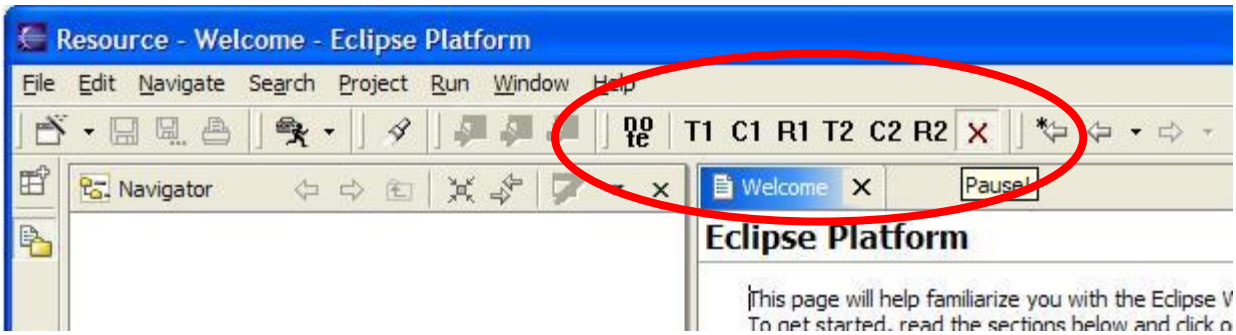
Ker smo hoteli imeti možnost rekonstruiranja poteka dela za vsakega razvijalca posebej od začetka do konca eksperimenta, se je pojavila potreba po shranjevanju vseh verzij izvorne kode in njihovem časovnem označevanju. Razmišljali smo o uvedbi katerega od standardnih sistemov za vodenje verzij (npr. CVS - http://www.cvs.org), vendar le-ti niso tako primerni za shranjevanje in delo **vseh** verzij kode. Njihov namen je bolj v tem, da npr. razvijalci po končani in uspešno testirani zgodbi pošljejo novo verzijo kode v skupen repozitorij. Prav tako bi nam uvedba CVS-ja dodatno otežila prenosljivost okolja in ga naredila precej bolj kompleksnega, saj bi morali študenti ali imeti stalen in hiter dostop do centralnega CVS strežnika ali pa si inštalirati svoj CVS strežnik. Ker so naši subjekti delali sami, tudi ni bilo potrebe za kakršnim koli deljenjem kode, kot je to pogoj pri delu v skupini.

Rešitev problema smo našli v okolju Eclipse, ki ima en takšen enouporabniški sistem za vodenje verzij že vgrajen v samo razvojno okolje. Ta tako imenovana "lokalna zgodovina" (angl. *local history*) je namenjena ravno temu, da ima razvijalec možnost priklicati verzijo katerekoli kode, ki jo je napisal v vnaprej določenem časovnem obdobju.

Namen zbiranja vseh verzij kode je bil tudi psihološki. Študentom smo povedali, da bodo morali oddati **izvorno kodo skupaj z vso zgodovino razvoja**, zato so le-ti bolj pazili na to, da so sledili izbranemu procesu razvoja, zmanjšali pa smo tudi verjetnost morebitnega kopiranje rešitev od sošolcev. Mnenja smo, da je to precej vplivalo na to, da so rešitve, ki so jih razvili, res naredili sami.

3.6.2 Zbiranje podatkov o procesu razvoja

Za beleženje podatkov o procesu razvoja smo razvili posebno orodje, ki smo ga poimenovali ProcessLog (Priloga D). Gre za komponento napisano v Javi, ki se popolnoma integrira v okolje Eclipse. Slika 3-1 prikazuje uporabniški vmesnik naše komponente ProcessLog. S slike je razvidna popolna integracija v okolje Eclipse in orodno vrstico.



Slika 3-1: Uporabniški vmesnik naše komponente za beleženje podatkov o procesu razvoja ProcessLog z vidnim pritisnjenim gumbom za pavzo.

Pomen gumbkov:

- "Note": doda v dnevniško datoteko poljubno opombo razvijalca (npr. "pred 5. minutami sva pozabila pritisniti gumb C1", "testi večnitnega delovanja enkrat uspejo, drugič ne" itd.). Razvijalcu se prikaže okno, v katerega vpiše opombo. Opombe so bile primarno namenjene za vpis dogodkov, ki bi lahko kakorkoli vplivali na proces označevanja faz razvoja.
- "T1": testira prvi razvijalec (faza testiranja - *testing*).
- "C1": kodira prvi razvijalec (faza kodiranja - *coding*).
- "R1": preoblikuje prvi razvijalec (faza preoblikovanja - *refactoring*).
- "T2": testira drugi razvijalec (faza testiranja).
- "C2": kodira drugi razvijalec (faza kodiranja).
- "R2": preoblikuje drugi razvijalec (faza preoblikovanja).
- "X": Odmor (pavza, počitek, konec dela za eno sejo, čas za kosilo, itd.).

V dnevniško datoteko se zapisujejo zapisi v obliki "`datum ura akcija`", pri čemer je `akcija` lahko:

- "*`T1`*", "*`C1`*", "*`R1`*" – faze za prvega razvijalca v paru.
- "*`T2`*", "*`C2`*", "*`R2`*" – faze za drugega razvijalca v paru.
- "*`PA`*" – odmor.
- "*`LO opomba`*" – *`opomba`* je poljuben opis, ki se doda z gumbom "Note".

- "*TS n*" – zagon *n* testov (*n* testnih metod v ogrodju JUnit).
- "*TE OK čas*" – vsi testi, ki smo jih zagnali, so uspešni; zagon testov je trajal *čas* milisekund.
- "*TE FE čas F n E m*" – najmanj en test izmed vseh testov, ki smo jih zagnali, ni uspešen; *n* testov je neuspešnih, *m* testov je sprožilo izjemo v ogrodju JUnit; celoten zagon je trajal *čas* milisekund

Primeri izpisov v dnevniški datoteki:

```
…
2003.05.23 17.45.30 T1
2003.05.23 17.45.32 TS 4
2003.05.23 17.45.34 TE FE 1484 F 1 E 0
2003.05.23 17.47.08 C1
…
```

3.7 Grožnje veljavnosti eksperimenta

Študija je veljavna[18], če:

1. njene meritve tudi v resnici merijo tisto, kar naj bi merile,
2. če v sami študiji ni logičnih napak v analizi zaključkov iz dobljenih rezultatov.

V tem poglavju bomo natančno in sistematično obdelali možne grožnje veljavnosti naše raziskave in analizirali njihove morebitne posledice za naš eksperiment.

Preverili smo tudi, da dve različni metriki pri testih (število testov in kvaliteta testov) in kvaliteti (zunanjo in notranjo kvaliteto) res merita dva različna koncepta. V literaturi navajajo, da naj ne bi bil korelacijski koeficient med dvema metrikama večji kot 0.85, če

[18] *validity* – veljavnost.

naj bi metriki res merili dva različna koncepta. Ali povedano drugače, če je korelacija med obema metrikama iz iste kategorije prevelika, najbrž merita isto stvar.

Pri vseh naših eksperimentih smo to preverili, korelacijski koeficient je pri vseh eksperimentih v pravilnih mejah (od 0,5 do 0,6). Zato lahko trdimo, da dve metriki za kvaliteto in dve metriki za teste res merita dve različne stvari[19].

3.7.1 Interna veljavnost

Študija ima dobro interno veljavnost, če vanjo nismo vpeljali prikritih spremenljivk[20], ki bi lahko vplivale na proces, ki ga preučujemo.

3.7.1.1 Hawthornov efekt

Hawthornov efekt je pojav, kjer raziskovalec s svojimi pričakovanji vpliva na subjekte v eksperimentu. Ime je dobil po znanem (ponesrečenem) eksperimentu v tovarniškem okolju, kjer napačno povezali izboljšanje produktivnosti pri delavcih s količino svetlobe na delovnem mestu, čeprav je v resnici bilo povečanje produktivnosti v eksperimentalni skupini moč pripisati le večjemu zanimanju raziskovalcev za opazovane delavce. S tem so raziskovalci v eksperiment nehote vpeljali prikrito spremenljivko (večja pozornost raziskovalcev kontrolni skupini), saj so delavci delali bolje zato, ker so takšno vedenje od njih pričakovali raziskovalci.

Študente smo seznanili z namenom študije (primerjava dveh procesov razvoja), **vendar jim hipotez nismo razkrili**. S tem smo se vplivu Hawthornovega efekta v našem eksperimentu izognili.

3.7.1.2 Pristranost zaradi smrtnosti

Pristranost zaradi smrtnosti[21] nastane takrat, ko je vzorec kasneje v eksperimentu zaradi smrtnosti subjektov (izločevanja subjektov iz eksperimenta) ni več reprezentativen, t.j. ne predstavlja več ciljne populacije.

---

[19] Seveda pa je tu potrebno povedati, da obe metriki sta v korelaciji, kar potrjuje tudi logično razmišljanje, da npr. večje število testov prinaša tudi večjo pokritost kode.

[20] *Covert variables*

Problem pristranosti zaradi smrtnosti je v večji ali manjši meri prisoten pri skoraj vseh eksperimentih na področju empiričnega programskega inženirstva. V poglavju 2.5 smo opisali sorodne raziskeve, od katerih nekateri posebej izpostavljajo problem zaradi smrtnosti [Erdogmus+2005, Geras+ 2004].

Na ta problem smo naleteli tudi mi, najbolj izražen pa je v eksperimentu s samostojnimi razvijalci SR03. Po izločanju neustreznih subjektov zaradi očitne neskladnosti s procesom dela je v ITL skupini ostalo le 11 subjektov (od 18), kar nam da kar 39% smrtnost! Subjekti, ki smo jih izločili, so bili vsi iz skupine slabših študentov. Še več, izločili smo celo vse študente iz najslabše skupine po povprečni oceni (iz spodnjih 25% vseh študentov). V TDD skupini je bila smrtnost v istem eksperimentu (SR03) le 15%. Natančnejši opis je v Prilogi A.

3.7.1.3 Pristranost kontrolne skupine

Pristranost zaradi kontrolne skupine je sorodna Hawthornovemu efektu. Če subjekti v kontrolni skupini vedo, da so dodeljeni kontrolni skupini, bi se lahko zato drugače obnašali kot če bi vedeli, da so v "pravi", eksperimentalni skupini. Drugače povedano, kontrolna skupina ne sme vedeti, da je "kontrolna", da to ne vpliva na njeno obnašanje v eksperimentu.

Tej nevarnosti smo se izognili tako, da študentom nismo razkrili, katera skupina je v našem eksperimentu kontrolna.

3.7.2 Eksterna veljavnost

Študija ima dobro eksterno veljavnost, če je zaključke dobljene na vzorcu mogoče posplošiti na celotno populacijo oziroma na druge (sorodne) populacije in kontekste uporabe.

Idealen eksperiment za naša raziskovalna vprašanja bi bil nekaj let dolg eksperiment v realnem okolju profesionalnih razvijalcev, kjer bi imeli dve popolnoma enakovredni skupini razvijalcev, ki bi istočasno reševali enak problem, ena bi uporabljala proces

[21] *Mortality bias*

razvoja TDD, druga ITL. Takšnega idealnega eksperimenta v praksi ni mogoče izvesti. Še posebej so problematični vsi eksperimenti v profesionalnem okolju, saj si nobeno podjetje ne more privoščiti velikega izpada dohodkov. Težavnost eksperimentiranja v realnem okolju profesionalnih razvijalcev se vidi iz študije Georga in Williamsa [George+ 2004], še bolj pa iz študije Gerasa in sodelavcev [Geras+ 2004], kjer težave pri vzorčenju in velika smrtnost študijo degradirajo na raven pilotne študije oz. kvazi-eksperimenta. Sami za izvedbo eksperimenta v realnem okolju žal nismo imeli možnosti, zato smo se osredotočili na izvedbo v univerzitetnem okolju.

Eksterna veljavnost naših raziskovalnih rezultatov bi bila lahko zato nižja ravno zaradi tega, ker smo namesto profesionalnih razvijalcev uporabili študente. Vendar študije različnih raziskovalcev to tezo ovržejo. Tako sta raziskavi Porterja in Votte [Porter+ 1995] ter Hösta in sodelavcev [Höst+ 2000] pokazali, da se **rezultati eksperimentov s študenti lahko ekstrapolirajo na profesionalne programerje**, še posebej to velja za študente višjih letnikov in podiplomske študente.

Tudi Tichy [Tichy 2000] v svojih **smernicah za ocenjevanje empiričnih raziskav** na področju **programskega inženirstva zagovarja uporaba študentov**. Uporaba študentov je po Tichy-ju sprejemljiva v naslednjih situacijah:

- Študenti so bili dovolj pripravljeni na aktivnosti v eksperimentu (nujen je trening aktivnosti!).
- Študenti se uporabljajo za dokazovanje trenda, kot je na primer primerjava med dvema metodama z namenom ugotovitve, katera je boljša. Če se ugotovi trend pri študentih, je tudi zelo velika verjetnost, da je trend v isti smeri prisoten tudi med profesionalci (mogoče le z različno velikostjo).
- Študenti se uporabljajo za eliminacijo alternativnih hipotez.
- Študije s študenti so ponavadi predpogoj za to, da k nadaljevanju raziskav pritegnemo tudi profesionalce. Tu omenimo še primer iz preteklosti: znan in v praksi potrjen model razvoja CMM (http://www.sei.cmu.edu/cmm) je bil najprej preizkušen na študentski populaciji, šele ko so bili rezultati ugodni v tem kontekstu, je dobil več kredibilnosti in se je lažje uveljavil in preizkusil tudi v praksi.

Naša raziskava se v celoti drži Tichy-jevih priporočil.

Ima pa naša študija, tako kot vse sorodne raziskave, ki smo jih opisali v poglavju 2.5, glede eksterne veljavnosti tudi slabost: eksperimentalne naloge so vseeno relativno majhne in preproste v primerjavi s tipičnimi realnimi projekti v industriji. To slabost nekoliko omili dejstvo, da v obeh primerjanih procesih razvoja uporabljamo izrazito iterativno delo, zato ocenjujemo, da se zato da ekstrapolirati rezultate tudi na daljše projekte. Žal eksperimenta z bolj kompleksnimi in daljšimi nalogami zaradi objektivnih okoliščin ni bilo mogoče izvesti.

3.7.3 Pristranost zaradi ene izvedbe

Če eksperiment izvedemo le enkrat, lahko zaradi tega pride pri rezultatih do zaključkov, ki niso pravilni. Ta efekt je poznan kot pristranosti zaradi (samo) ene izvedbe eksperimenta (*mono-operation bias*). Vse kontrolirane eksperimente, ki smo jih našteli v sorodnih raziskavah, pesti ta problem.

Nevarnosti pristranosti zaradi ene izvedbe smo se sami izognili tako, da smo **eksperiment ponovili** (**replicirali**) z **različnimi** eksperimentalnimi nalogami in na **drugem** vzorcu subjektov.

Nam v prid glede odpravljanja te nevarnosti je tudi dejstvo, da smo **vsak eksperiment poleg replikacije** izvedli še **v dveh različnih kontekstih:**

- kontekst programiranja v parih (emulacija TDD v XP),
- kontekst samostojnih razvijalcev (emulacija TDD znotraj kakšne druge metodologije).

S tem smo nevarnost pristranosti zaradi le ene izvedbe eksperimenta **še dodatno zmanjšali**.

3.7.4 Pristranost zaradi nepopolnega upoštevanja zapovedanega procesa dela

V vseh empiričnih študijah se pojavlja problem, kako preverjati, ali so se subjekti res v celoti držali navodil. Slabo upoštevanje predpisanega procesa razvoja bi v našem primeru lahko popačilo rezultate eksperimenta. Sami smo se problema lotili na več načinov:

- Subjekte smo tekom treninga in eksperimenta nenehno opozarjali o pomembnosti upoštevanja procesa razvoja za našo raziskavo.
- Študente smo za delo prej dobro pripravili, zato so proces razvoja dobro poznali (glej poglavje 3.10.1 Trening posameznih aktivnosti v procesu razvoja).
- Meritve tistih subjektov, pri katerih smo z analizo poteka dela (verzije kode in dnevnik razvoja) ugotovili, da se niso v popolnosti držali zapovedanih pravil, smo izločili.

S temi ukrepi smo po našem mnenju v danih okoliščinah maksimizirali stopnjo skladnosti s procesom dela.

3.8 Analiza podatkov

Za analizo podatkov smo predvideli klasično statistično testiranje hipotez o razlikah med povprečnima vrednostma parametrov med dvema neodvisnima skupinama meritev.

Testiranje statistične značilnosti tipa Neyman-Pearson je oblika zavrnitve ali sprejetja ničelne hipoteze (označimo jo s H_0), kjer je ničelna hipoteza postavljena samo z namenom, da z njeno zavrnitvijo omogočimo raziskovalcu, da sprejme alternativno hipotezo H_a in s tem zaključi, da je v eksperimentu prisoten opazovani efekt.

Odločili smo se za uporabo neparametričnega Mann-Whitney U testa [Glantz 2002]. Mann-Whitney U test kot vsi neparamatrični testi ne predvideva normalne porazdelitve meritev znotraj vsake od testnih skupin, njegova slabost pa je malo manjša moč testa v primerjavi s testi iz skupine parametričnih testov (npr. t-test). Pri obdelavi rezultatov smo uporabili statistični program SPSS, ki ima posebne popravke za natančen izračun Mann-Whitney U testa na manjših vzorcih (*Mann-Whitney U Exact Test*). Pri analizi podatkov smo si veliko pomagali tudi z Microsoftovim Excel-om.

Pomembno je poudariti tudi to: pri **hipotezah, ki določajo smer efekta** (npr. da bo povprečna vrednost metrike ena od skupin večja kot povprečna vrednost v drugi skupini), smo uporabili **enostranski** test (*one-tailed test*). Pri **hipotezah, kjer smer ni določena** (torej ko iščemo le razliko med skupinama, brez vnaprej predpisane smeri efekta), pa smo uporabili **dvostranski** test (*two-tailed test*). Pri enosmernih hipotezah je potrebno vsak efekt v nasprotni smeri označiti kot naključje [Miller+ 1997], torej da ni statistično značilen. Ne smemo pa v takšnem primeru spremeniti hipoteze in uporabiti dvosmernega testa, kot je to pogost primer v praksi (primer je npr. sorodna raziskava Erdogmusa in sodelavcev [Erdogmus+ 2005]).

3.9 Ocena statistične moči

Analiza statistične moči je neločljivo povezana s testiranjem značilnosti. **Statistična moč je definirana kot verjetnost, da bo statistični test pravilno zavrnil napačno ničelno hipotezo** (oz. je verjetnost, da efekt najdemo, če le-ta obstaja). Na primer: če je statistična moč testa 0,4 to pomeni, da če efekt obstaja, ga bomo v desetih ponovitvah eksperimenta našli le štiri krat.

V literaturi je kot zadovoljivo visoka **statistična moč testa** običajno navedena **vrednost 0,8** [Miller+ 1997]. Raziskovalec, ki ne analizira statistične moči testa pri svojih eksperimentih tako nima nobene informacije o tem, kakšno vlogo ima naključje v njegovih eksperimentalnih rezultatih. Vendar je še posebej v empiričnih raziskavah na področju programskega inženirstva navajanje in upoštevanje statistične moči testa prej izjema kot pravilo, saj jo navajajo le zelo redki raziskovalci!

Statistična moč testa je tesno povezana s tremi komponentami:

- Kriterij statistične značilnosti (α).

  Kriterij statistične značilnosti je verjetnost, da napačno zavrnemo ničelno hipotezo (tveganje za napako I tipa (*Type I error*) – torej da najdemo efekt, ki ga v resnici ni). Usmerjenost testa je tudi pomembna, saj imajo enosmerni testi pri istem kriteriju značilnosti večjo moč. Sami smo v raziskavi uporabili tako enosmerne hipoteze kot tudi hipoteze, kjer smer ni določena.

Zaradi relativno majhnih vzorcev in izjemnega pomanjkanja kakršnih koli informacij glede vrednosti opazovanih parametrov v procesu razvoja TDD smo se odločili, da bomo sami **kriterij statistične značilnosti** testa postavili **na vrednost α=0,1**.

- Velikost vzorca (*N)*.

  Večji kot je vzorec, manjša je napaka in s tem je večja tudi statistična moč testa, seveda pri predpostavki, da je vzorec reprezentativen. V določanju velikosti vzorca smo bili pri našem eksperimentu precej omejeni (študenti v letniku), prav tako nismo mogli veliko vplivati na stopnjo smrtnosti (*mortality rate*) v eksperimentu.

  Idealno je, če sta obe skupini vzorcev (eksperimentalna in kontrolna) enako veliki. Eksperimentalni skupini pa sta velikokrat v praksi predvsem zaradi smrtnosti neenaki. V tem primeru izračunamo t.i. harmonično povprečje vzorca:

$$N = \frac{2 * N_1 * N_2}{N_1 + N_2} \tag{3.1}$$

- Velikost efekta (Cohen *d*).

  Velikost efekta je stopnja prisotnosti predmeta zaznavanja (angl. *phemomenon*) v populaciji. Večja kot je velikost efekta, večja je verjetnost za zaznavo efekta in s tem za zavrnitev ničelne hipoteze.

  Sami smo imeli precej velike težave z začetno ocenitvijo velikosti efekta. Na podlagi literature in svojih izkušenj smo se odločili, da se bomo v eksperimentu osredotočili le na **velike efekte** (po Cohenu je velik efekt pri *d*=0,8), zato vse naše ugotovitve veljajo pri predpostavki velikega efekta – treba je priznati, da so ravno **veliki efekti tudi najbolj zanimivi za preučevanje, saj imajo tudi precejšnjo praktično vrednost**. Primer: lahko bi namreč ugotovili, da efekt v produktivnosti je, vendar je v praksi tako majhen, da je praktično nepomemben (npr. če bi ugotovili večjo produktivnost za 0,05%, je to v praksi povsem nepomemben rezultat, tudi če je statistično značilen).

  Velikost efekta *d* se izračuna po formuli (test pri dveh neodvisnih vzorcih):

$$d = \frac{|\mu_1 - \mu_2|}{\sigma}, \quad \sigma = \sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}} \tag{3.2}$$

pri čemer je μ povprečna vrednost, σ pa je standardna deviacija.

Ravno ocena velikosti efekta omogoča načrtovanje učinkovite replikacije eksperimenta in tudi meta-študijo različnih raziskav na isto temo.

Če poznamo zgornje tri komponente, je mogoč tudi izračun statistične moči testa. Mi smo za izračun statistične moči uporabili prosto dostopen program G*Power avtorjev Erdfelderja, Faula in Buchnerja [Erdfelder+ 1996] (http://www.psycho.uni-duesseldorf.de/aap/projects/gpower). Program nam omogoča tako *a priori* izračunavanje statistične moči testa (uporabili smo ga pri načrtovanju naših eksperimentov) kot tudi *post-hoc* analizo statistične moči, ki smo jo zaradi smrtnosti morali izračunati naknadno po eksperimentu v začetni fazi analize eksperimentalnih podatkov.

Glede na izkušnje iz prejšnjih let smo pričakovali pri eksperimentu približno 40 študentov, kar bi nam v idealnem primeru dalo dve skupini po 20 študentov. Izračun statistične moči enosmernega testa za primerjavo povprečnih vrednosti med eksperimentalno in kontrolno skupino nam tako da vrednost 0,89 (N_1=20, N_2=20, α=0,1, d=0,8), kar je precej nad mejo 0,8. Za eksperiment s pari smo predvidevali torej dve skupini po 10 udeležencev (število subjektov se prepolovi!), kar nam žal precej zmanjša tudi statistično moč enostranskega testa (statistična moč je v takem primeru 0,69 (N_1=10, N_2=10, α=0,1, d=0,8). Malo manjšo statistično moč testa smo zato poskušali kompenzirati s tem, da smo eksperiment replicirali, statistično moč pa objavili, tako da si lahko bralec ustvari tudi svoje mnenje. Veliko vrednost eksperimenta v kontekstu parov smo videli ravno v imitaciji tipičnega okolja, v katerem se testno voden razvoj uporablja, zato smo se tu pač sprijaznili z nekoliko manjšo statistično močjo testa.

Kasneje se je izkazalo, da so te ocene statistične moči prevelike zaradi manjšega števila sodelujočih subjektov in zaradi večje smrtnosti, kot smo jo pričakovali. V letu 2003 je tako sodelovalo 38 študentov, leta 2004 pa 34 študentov. Nekaj meritev (še posebej velja to za leto 2003!) smo morali iz različnih razlogov izločiti, tako da so končne ocene statistične moči naši testov manjše: za kontekst samostojnih razvijalcev SR04 je ocena statistične moči enosmernega testa 0,82 (N_1=14, N_2=18, α=0,1, d=0,8), v kontekstu parov leta 2004 PP04 je statistična moč enostranskega testa 0,6 (N_1=7, N_2=9, α=0,1, d=0,8), v

kontekstu parov leta 2003 PP03 je statistična moč enostranskega testa 0,66 (N_1=10, N_2=9, α=0,1, d=0,8), v kontekstu samostojnih razvijalcev SR04 (ki smo ga izključili zaradi suma pristranosti zaradi smrtnosti, glej Prilogo A) pa je bila pričakovana statistična moč enosmernega testa 0,77 (N_1=11, N_2=17, α=0,1, d=0,8).

Sklenimo: dovolj visoka statistična moč testa nam omogoča, da sprejmemo ničelno hipotezo, če efekta v vzorcu nismo zaznali.

3.10 Potek eksperimenta

Eksperiment je potekal v dveh fazah. Prva je bila temeljita priprava na eksperiment, druga pa sama izvedba eksperimenta. Eksperimenta v letih 2003 in 2004 sta potekala enako.

3.10.1 Trening posameznih aktivnosti v procesu razvoja

Iz izkušenj drugih raziskovalcev o uporabi agilnih metodologij v univerzitetnem okolju [Mugridge 2003, Melnik+ 2002, Müller+ 2001] je moč sklepati na to, da je na začetku razvijalcem takšen način razvoja tuj in da ga je težko osvojiti[22]. Zato je potrebno razvijalce pred eksperimentom temeljito pripraviti, tako s potrebno **teoretično** podlago (predvsem o testiranju) kot tudi še bolj pomembno, **s praktičnim delom**. Praktično delo pri uvajanju razvijalcev v metodologijo XP je tako pomembno, da metodologija predvideva celo posebno vlogo, to je vlogo **trenerja**[23] [Beck 2000].

Pred izvedbo eksperimenta so morali naši študentje (poleg številnih krajših primerov razvoja, ki smo jih skupaj naredili na vajah) samostojno napraviti tudi 2-3 krajše naloge, katere smo potem pregledali in vsakega študenta posebej opozorili na napake in pomanjkljivosti. Tako smo zagotovili, da je bil **proces dela v eksperimentu že utečen** in

[22] Kar potrjujejo tudi naši rezultati, glej poglavje 4.4.5
Kvalitativni parametri procesa razvoja.

[23] Coach – trener.

se študentje niso več "lovili" s samim procesom razvoja in so se zato lahko osredotočili na reševanje problema.

3.10.2 Izvedba eksperimenta

Načrtovali in izvedli smo dva niza po dva kontrolirana eksperimenta, znotraj vsakega niza po en eksperiment v vsakem od obeh kontekstov. Zaradi objektivnih omejitev (vezanih na predavani predmet in razpoložljivost ustreznih študentov) je bilo moč eksperiment izvesti **le v letnem semestru posameznega študijskega leta**. Obeh nizov eksperimentov tako nismo mogli končati prej kot v dveh letih.

Leta 2002 smo izvedli **študijo tehničnih zahtev, pogojev in kritičnih dejavnikov uspeha** za kar najbolj učinkovit in olajšan **razvoj v TDD** procesu dela [Pančur+ 2003a, Pančur+ 2003b]. Ugotovili smo, da je najprimernejše razvojno okolje za naš eksperiment (torej za oba procesa razvoja – TDD in ITL):

- Uporaba programskega jezika Java – študentje so že vsi znali Javo, saj so se je učili pri drugih predmetih v nižjih letnikih. To je bil tudi eden izmed izločilnih dejavnikov za okolje .NET, večina študentov ga namreč ni poznala.
- Razvojno okolje Eclipse (http://www.eclipse.org):
    - Je projekt odprte kode, zato ni problema z licencami za uporabo pri študentih
    - Ima velik nabor avtomatiziranih preoblikovanj – preoblikovanje kode je lahko, če se dela "ročno", zelo zapleten proces. Glede na to nismo mogli od slabših študentov pričakovati, da bi ga uporabljali dovolj pogosto, saj bi se ga najbrž ravno zaradi kompleksnosti izogibali. Avtomatizirana preoblikovanja pa je po drugi strani veliko lažje uporabljati. Tako je npr. eno najpogostejših preoblikovanj "Izloči metodo" (*extract method*) moč izvesti v okolju Eclipse tako, da najprej označimo kos kode, ki ga želimo izločiti v svojo metodo. Iz menija izberemo ustrezno preoblikovanje, nato pa razvojno okolje avtomatsko preveri vse pogoje (lokalnost spremenljivk v izbranem fragmentu kode, itd. – glej katalog preoblikovanj v [Fowler

2000] – *Extract method*) in ustrezno preoblikuje kodo. Tako izločimo iz procesa preoblikovanja nehotne napake.

  - Ima odlično integrirano podporo za JUnit – *de-facto* standardno ogrodje za pisanje testov enot v Javi (http://www.junit.org), podpirajo ga tudi vsa komercialna razvojna okolja. Eclipse na tem mestu še posebej blesti, podpora in integracija ogrodja JUnit je res tesna in omogoča kar najlažji možen razvoj testov.
  - Se ga da nadgrajevati s svojimi moduli (*plug-in*) – zaradi specifičnih zahtev pri eksperimentu smo morali sami razviti orodje za vodenje dnevnika procesa dela, ki naj bi bilo po možnosti čim bolj integrirano v samo razvojno okolje. Boljša kot bo integracija in lažja kot bo uporaba, več uspeha in natančnejše meritve smo se lahko nadejali pri rezultatih.
  - Zaradi optimiziranih gradnikov je uporabniški vmesnik okolja Eclipse zelo odziven in hiter, prav tako tudi sam razvoj – v času izvajanja eksperimentov je bil Eclipse zaradi SWT grafičnih gradnikov bolj odziven pri uporabi kot ostala okolja, zgrajena s standardnimi Javinimi grafičnimi gradniki Swing. V računalniških učilnicah na Fakulteti za računalništvo in informatiko v letih 2003 in 2004 računalniki niso bili ravno zelo zmogljivi, zato je boljša odzivnost okolja prinesla bolj udobno delo.

- Za sprejemne teste v eksperimentu s pari smo uporabili ogrodje FIT (http://fit.c2.com), katerega smo nadgradili s posebnim modulom za testiranje programov preko vtičev SAT (*Socket Acceptance Tester*, http://fit.c2.com/files/SocketAcceptanceTester/sat/TestSat.html). Ogrodje FIT pridobiva v zadnjem času veljavo **de facto standarda za sprejemne teste**, podobno kot je to že postalo izjemno razširjeno ogrodje JUnit de facto standard za teste enot.

Prvi niz eksperimentov (SR03, PP03) je trajal od februarja 2003 do junija 2003. Drugi niz eksperimentov (SR04, PP04) pa od februarja 2004 do junija 2004.

Sodelujočim študentom smo na začetku semestra razdelili **prvo anketo** (priloga E), na podlagi katere smo lahko ocenili parametre, ki po našem mnenju najbolj vplivajo na uspešnost študenta v eksperimentu.

Posebno smo se osredotočili na to, da ne bi bila ena od skupin boljša od druge. Pri ocenjevanju enakovrednosti študentov smo upoštevali naslednje parametre:

- povprečna ocena opravljenih izpitov,
- znanje programskega jezika Java,
- znanje programiranja,
- znanje objektno orientiranih principov,
- sodelovanje študenta v projektih zunaj fakultete (konkretne izkušnje pri razvoju v podjetjih).

Študente smo nato razdelili v tri razrede glede povprečne ocene (najboljši, srednji, najslabši študenti). Nato smo jih iz vsake skupine naključno razdelili v dve skupini, nakar smo preverili še, ali sta skupini enakovredni tudi po parametrih, pridobljenih iz anket.

Razlike med obema skupinama pri upoštevanju vseh teh parametrov so statistično neznačilne (Mann-Whitney U test, α=0,1). S tem smo si zagotovili, da ni ena skupina boljša od druge, kar bi vplivalo na rezultat eksperimenta. Test smo zaradi smrtnosti morali še enkrat ponoviti na koncu eksperimenta, pred fazo analize rezultatov. Tudi tu razlike niso bile statistično značilne.

Vsi študentje so poslušali enaka predavanja, na katerih smo enakovredno predstavili oba procesa razvoja. Poseben pomen smo dali predstavitvi samih tehnik, ki se uporabljajo v procesu razvoja ITL in TDD: testiranje enot, JUnit, preoblikovanje, agilen design, najpogostejše napake, itd.

Študentom smo tudi predpisali standard kodiranja, ki so se ga morali držati in uporabljati že od vsega začetka. V standardu, ki smo ga bazirali na Sun-ovih smernicah za programiranje v Javi (http://java.sun.com/docs/codeconv/), smo natančno definirali stil kodiranja, ki se ga morajo študenti držati (zamiki, max. dolžina vrstic, itd.). S tem smo kar se da minimizirali razlike v produktivnosti, ki bi lahko nastale zaradi drugačnega stila kodiranja (npr. nekateri študenti bi pisali JavaDoc dokumentacijo in komentarje, drugi pa ne). Skladnost s standardom kodiranja smo preverjali tudi formalno – s posebno dodatno komponento v okolju Eclipse (CheckStyle, http://checkstyle.sourceforge.net), ki je študente avtomatsko opozarjala na vse kršitve standarda.

V fazi treninga so študenti delali enake naloge, uporabljali enaka orodja, itd. vse z namenom, da jih kar najbolje pripravimo za uporabo dodeljenega procesa razvoja. Tako smo zagotovili, da je edina neodvisna spremenljivka v eksperimentu proces razvoja; torej kdaj se pišejo testi: pred ali po kodi, ki jo testirajo.

V prvem nizu eksperimentov je bilo udeleženih 38 študentov, od tega jih je 20 uporabljalo TDD, 18 pa ITL proces razvoja[24]. V drugem nizu leta 2004 je sodelovalo 34 študentov (16 TDD, 18 ITL). Pri tem je seveda potrebno upoštevati, da se v kontekstu dela s pari število sodelujočih (in s tem meritev) ravno zaradi dela v paru prepolovi. Zaradi smrtnosti iz različnih vzrokov (neupoštevanje pravil procesa razvoja, neudeležba, itd.) je končno **število ustreznih meritev** manjše:

- Leto 2003:
    - Kontekst parov PP03 (TDD 10, ITL 9),
    - Kontekst samostojnih razvijalcev SR03 (17 TDD, 11 ITL, glej tudi Prilogo A).
- Leto 2004:
    - Kontekst parov PP04 (TDD 7, ITL 9),
    - Kontekst samostojnih razvijalcev SR04 (TDD 14, ITL 18).

Sama izvedba eksperimenta v posameznem letu je potekala v dveh delih. V prvem delu so vsi študentje delali v parih (kontekst PP03, PP04), v drugem delu so delali samostojno (kontekst SR03, SR04). Eksperiment dela v parih je potekal 3-4 tedne, eksperiment dela samostojnih razvijalcev pa smo izvedli v enem dnevu sredi junija 2003 in 2004.

Na koncu smo med udeleženci eksperimenta izvedli **drugo anketo** (priloga E), s katero smo pridobili kvalitativne podatke o procesu razvoja, njihova opažanja, težave in prednosti ob uporabi posameznega razvojnega procesa. **Anketa je bila anonimna, saj smo tako želeli zagotoviti čimbolj objektivne odgovore.** Del rezultatov prvega eksperimenta smo objavili v [Pančur+ 2003c].

[24] Ker so v prvem delu eksperimenta delali v parih, skupini po številčnosti ne moreta biti popolnoma enaki.

4 ANALIZA MERITEV IN REZULTATI

V tem poglavju bomo predstavili rezultate analize v eksperimentih pridobljenih podatkov. Najprej bomo obdelali tri najpomembnejše parametre – **produktivnost, kvaliteto** in **teste**. Sledi podpoglavje z analizo metrik, ki nam okarakterizirajo sam proces razvoja. V naslednjem podpoglavju so analizirani kvalitativni delavniki, pridobljeni z anketami. Tu obravnavamo predvsem odnos razvijalcev do posameznega procesa razvoja in njegovih komponent.

V analizi bomo rezultate meritev predstavili z grafom, ki prikazuje kvantile (angl. *boxplot graph*). To je graf, ki nam nazorno prikaže variabilnost meritev za posamezno skupino in je kot tak po našem mnenju najprimernejši za primerjalni pogled na podatke za obe skupini.

V grafu prikaza kvantilov škatla grafa objema 50% meritev (zgornji rob škatle je 75 percentil, spodnji rob 25 percentil). Polna horizontalna linija označuje mediano, črtkana modra horizontalna linija označuje povprečno vrednost meritev. Dolžina vertikalne črte nad in pod škatlo označuje meje zunanjih vrednosti (*outlier limits*) oziroma zadnjo meritev, ki je še znotraj meja.

Zunanje vrednosti (imenovane tudi ekstremne vrednosti) so meritve, ki so za več kot 1,5-kratnik interkvartilnega razpona oddaljene od zgornjega ali spodnjega roba škatle grafa. (*IQR- Inter Quartile Range*, vrednost 75 precentila - vrednost 25 percentila = razpon dolžine škatle). V grafu jih označujemo z rdečim križcem, pomembne pa so zato, ker lahko močno vplivajo na povprečje v skupini.

Rezultate analize posameznih parametrov smo podali v tabeli, ki vsebuje:

- eksperimentalno skupino,
- število veljavnih meritev za vsako od skupin,
- povprečno vrednost metrike, standardno deviacijo in mediano (srednja vrednost),

- verjetnost *p*, da razlika v povprečni vrednosti med skupinama ni statistično značilna (Mann-Whitney U Exact test, α=0,1). Če je vrednost p<0,1 lahko zavrnemo ničelno hipotezo in sprejmemo alternativno hipotezo,
- velikost efekta razlike med povprečnima vrednostma (Cohen *d*) - glej tudi poglavje 3.9 Ocena statistične moči. Cohen predlaga naslednje vrednosti za opis velikosti efekta:
    - 0,2 majhen efekt,
    - 0,5 srednje velik efekt,
    - 0,8 velik efekt.

4.1 Produktivnost

V metodologiji XP imenujemo majhen zaključen del funkcionalnosti zgodba. Zgodba je končana, ko so vsi sprejemni testi za to zgodbo uspešni. Ravno ti sprejemni testi nam zagotavljajo minimum funkcionalne kakovosti, kakor ga je definirala stranka.

Najbolj očitna metrika za produktivnost je čas dela. Sami smo mnenja, da je ta metrika lahko nepravilna in zavajajoča. Razvijalec, ki naredi program hitro, vendar je koda slabe kvalitete, je napačno predstavljen kot zelo produktiven. Podobne težave imamo lahko z drugo običajno metriko za produktivnost - številom vrstic izvorne kode.

Zato je potrebno metriko za produktivnost:

a. ali povezati z uspešno končanimi zgodbami (torej s številom zgodb, ki zadovoljijo kriterij stranke). Samo takšna funkcionalnost predstavlja stranki neko vrednost (torej minimum zunanje kvalitete).
b. ali pa predpisati konstantno kvaliteto in obseg dela.

V vsakem drugem primeru je metrika za produktivnost lahko zavajajoča. Sami smo uporabili za merjenje produktivnosti oba dva pristopa, vsakega v enem kontekstu:

- Kontekst samostojnih razvijalcev (SR04).

Zaradi variabilnega obsega dela (dve zgodbi, cca 4 ure) smo tu metriko za produktivnost definirali kot **število uspešno končanih zgodb normalizirano s porabljenim časom**.

- Kontekst dela v parih (PP03, PP04).

  Zaradi fiksnega obsega dela in **vnaprej predpisane kvalitete** (podrobni sprejemni testi) v tem primeru **lahko uporabimo kot metriko za produktivnost** kar **čas dela**.

Na produktivnost lahko vpliva še en faktor- stil kodiranja. Razvijalec, ki npr. ne bi pisal JavaDoc komentarjev v kodi, bi zato lahko več časa porabil za pisanje kode in s tem imel časovno prednost pred študentom, ki bi po pravilih pisal tudi komentarje in bi se na splošno držal zapovedanega standarda kodiranja. Skladnost kode s predpisanim standardom kodiranja smo formalno preverjali s posebno dodatno komponento v okolju Eclipse (CheckStyle), ki je razvijalce sproti opozarjala na kršitve standarda kodiranja. S tem smo se v večji meri izognili morebitnim razlikam v produktivnosti, ki bi lahko nastale zaradi neupoštevanja standarda kodiranja.

4.1.1 Kontekst samostojnih razvijalcev

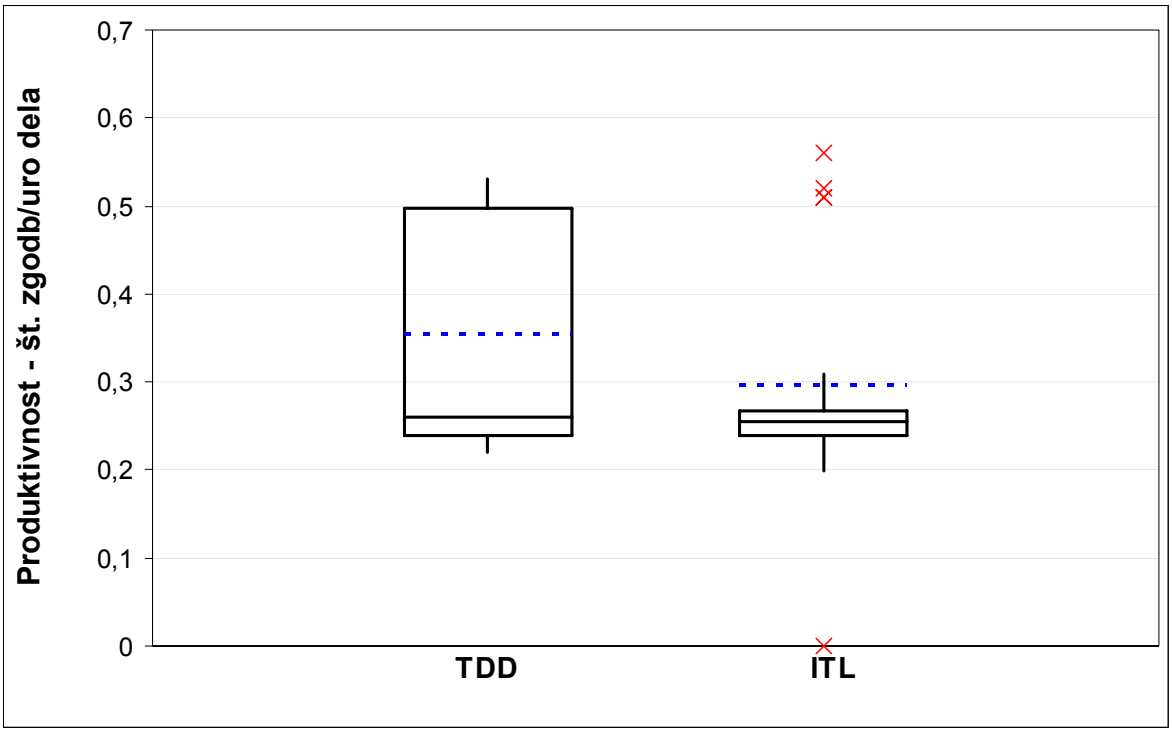
Produktivnost smo v kontekstu samostojnih razvijalcev merili z metriko **števila uspešno dokončanih zgodb normalizirano s porabljenim časom**. Normalizacija s porabljenim časom je tu nujna zaradi tega, ker je bil čas dela omejen na cca. štiri ure, lahko pa so razvijalci kodo oddali že prej, če so menili, da so z delom že končali. Takšni razvijalci so bolj produktivni, zato tudi imajo višjo metriko produktivnosti. Na primer razvijalec, ki konča uspešno obe zgodbi v dveh urah tako dobi vrednost metrike 1, drugi razvijalec, ki pa uspešno konča le eno zgodbo v 4 urah pa ima vrednost metrike za produktivnost 0,25 dokončanih zgodb/uro dela.

Naš formalni kriterij, ki smo ga uporabili, je: zgodba je **uspešno končana** in kot takšna stranki nudi zadovoljivo stopnjo funkcionalnosti, **če je uspešnih najmanj 50% sprejemnih testov**.

Večja kot je ta metrika, večja je produktivnost razvijalca. Tabela 4-1 prikazuje rezultate meritev produktivnosti v kontekstu SR04, Slika 4-1 pa prikazuje graf meritev.

Tabela 4-1: Rezultati meritev produktivnosti (št. dokončanih zgodb/čas dela (h)) – ***samostojni razvijalci SR04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	14	0,35	0,14	0,26	0,37	0,41
ITL	18	0,30	0,14	0,255		



Slika 4-1: Produktivnost (št. dokončanih zgodb/čas dela (h) – ***samostojni razvijalci 04***

Vidimo, da je TDD skupina v povprečju bolj produktivna od ITL skupine (za 17%), vendar je mediana praktično enaka v obeh skupinah. Iz grafa se tudi vidi, da je večina razvijalcev v ITL skupini osredotočena zelo ozko okoli mediane s tremi izrazito produktivnimi razvijalci (tri vrednosti nad 0,5), ki dvigujejo povprečje skupine. TDD skupina ima notranjih 50% meritev (meritve med 25-tim in 75-tim percentilom, prikazane kot velikost škatle v grafu) v povprečju višje kot je to pri ITL skupini, meja 75-tega percentila je celo veliko višja v primerjavi s ITL skupino.

Efekt je v pričakovani smeri (TDD skupina je bolj produktivna od ITL skupine (za 17%), vendar **razlika v produktivnosti ni statistično značilna** (p=0,37). Zato **v kontekstu samostojnih razvijalcev ne moremo zavrniti ničelne hipoteze PROD(H_0)**: $PROD_{TDD} = PROD_{ITL}$**.** Velikost efekta po Cohenu je majhna do srednje velika.

Sklenemo lahko: v kontekstu samostojnih razvijalcev razlik v produktivnosti med ITL in TDD skupino nismo zaznali**.**

4.1.2 Kontekst programiranja v parih

V kontekstu parov imamo lahko za metriko produktivnosti kar **čas dela**, saj je obseg dela (št. zgodb) in njihova kvaliteta za vse razvijalce enaka (konstantna).

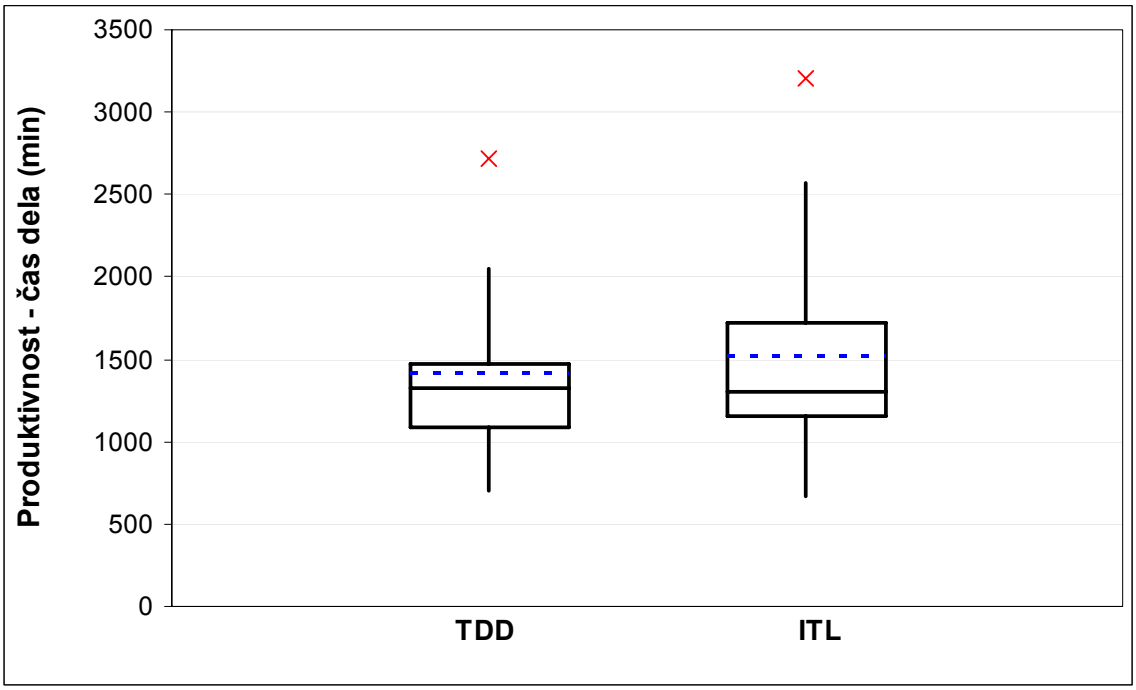
Tako je tu potrebno posebej poudariti razliko s prejšnjo metriko produktivnosti: **manjši kot je čas dela, večja je produktivnost**.

Tabela 4-2 prikazuje rezultate analize meritev za leto 2004, Slika 4-2 pa graf meritev. Tabela 4-3 pa prikazuje rezultate analize za leto 2003 (Slika 4-3 prikazuje graf meritev).

Tabela 4-2: Rezultati meritev produktivnosti (čas dela (min)) – ***pari 04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	1412	644	1320	0,5	0,16
ITL	9	1522	756	1306		

Za **leto 2004** lahko sklenemo: efekt je v predvideni smeri (TDD skupina je bolj produktivna od ITL skupine), vendar je **velikost efekta majhna in statistično neznačilna** (p=0,5). Meritve za **leto 2003** (Tabela 4-3, Slika 4-3) so na videz neugodne za našo hipotezo zaradi majhne prednosti v produktivnosti ITL skupine. Ta je nastala izključno zaradi ene meritve v ITL skupini, ki je izrazito hitro opravila nalogo (3 krat hitreje kot ostali) in s tem znižala povprečje ITL skupine pod povprečje TDD skupine. Če te ekstremne meritve ne bi upoštevali, bi bila razlika spet v korist TDD skupine. Ekstremno meritev v ITL skupini smo proglasili za veljavno, saj ni analiza kode in dnevnika dela pokazala nobenih nepravilnosti v razvoju, zaradi česar bi jo morali izključiti.

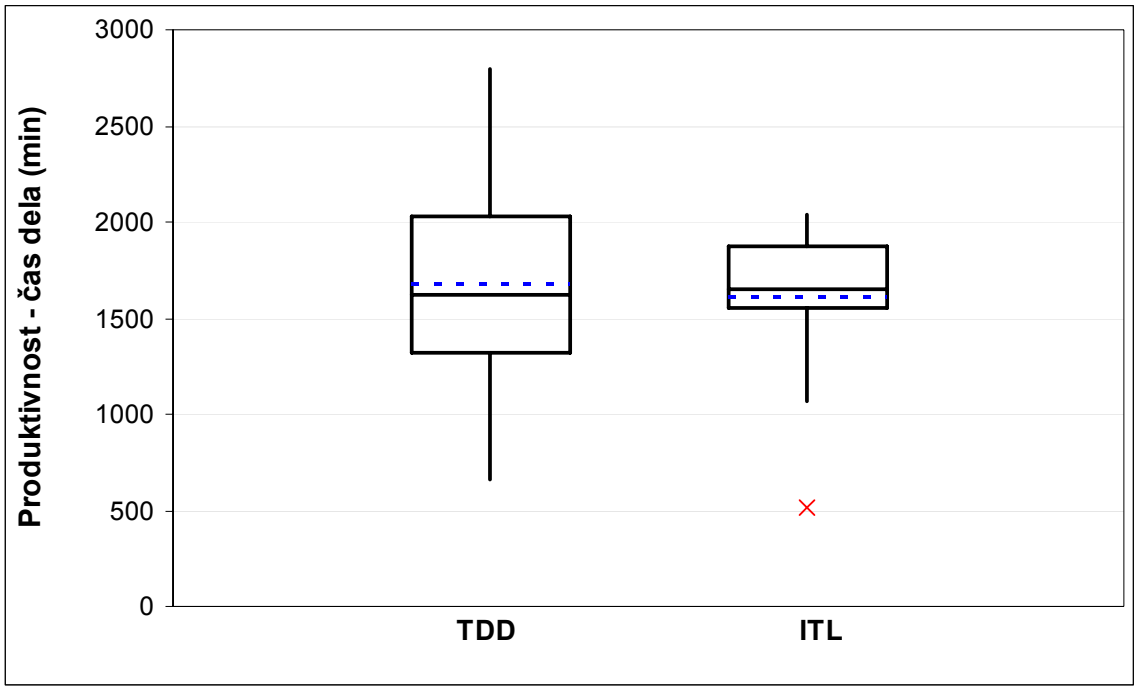


Slika 4-2: Produktivnost – čas dela (min) – ***pari 04***

Razlike tudi tu **niso statistično značilne** (p=0,48). Zato **v kontekstu parov ne moremo zavrniti ničelne hipoteze PROD(H_0)**: $PROD_{TDD}$ = $PROD_{ITL}$**. Velikost efekta** po Cohenu **je majhna.**

Tabela 4-3: Rezultati meritev produktivnosti (čas dela (min)) – ***pari 03 (PP03)***

Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p***	**Cohen *d***
TDD	10	1680	596	1621	0,48	0,12
ITL	9	1615	453	1647		



Slika 4-3: Produktivnost – čas dela (min) – ***pari 03 (PP03)***

Skupna ugotovitev glede produktivnosti ob upoštevanju obeh kontekstov:

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v produktivnosti. Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da velikih razlik med procesom TDD in ITL glede produktivnosti ni.

Z ozirom na izvedene meritve ocenjujemo, da gre v primeru produktivnosti kvečjemu za majhen efekt.

4.2 Kvaliteta

Kvaliteto ocenjujemo z dveh vidikov:

- Prvi je **zunanja kvaliteta** (oz. kvaliteta "navzven"[25], proti stranki), ki jo običajno merimo s sprejemnimi testi. Sprejemni testi spadajo v kategorijo testov tipa črne škatle in merijo predvsem število napak[26], za katere je merilo število neuspelih sprejemnih testov.
- **Notranja kvaliteta** kode[27] pa se ne kaže "navzven" z napačnim delovanjem, ampak je to merilo kakovosti napisane kode, ki se izraža kot jasnost, fleksibilnost arhitekture kode, enostavnost razumevanja, itd. Te lastnosti še posebej pridejo do izraza pri skupinskem delu na isti kodi (t.i. skupno lastništvo kode), ki jo zagovarja metodologija XP.

V praksi sta seveda zelo pomembna oba parametra, saj je na primer program, ki sicer dela vse brez znanih napak, a je napisan komplicirano in slabo, zelo težko spreminjati in vzdrževati. Kvaliteta takega programa je torej vsaj vprašljiva oz. jo lahko označimo kot slabo. Ker je danes zaradi poslovnih potreb in globalne konkurence potrebna veliko večja fleksibilnost v poslovanju, je realno pričakovati, da se bo tudi programska oprema, s katero podpiramo poslovanje, morala pogosto spreminjati in dopolnjevati. Zato je potrebno splošno kvaliteto ocenjevati skozi oba vidika, tako **zunanjo kot tudi notranjo kvaliteto**, saj je zelo "lepo" napisan program, ki ne dela tistega, kar stranka od njega pričakuje, neuporaben oz. je na splošno slabše kvalitete.

4.2.1 Zunanja kvaliteta

Metrike za oceno zunanje in notranje kvalitete so lahko različne. Samo smo izbrali kot merilo zunanje kvalitete kode **odstotek uspelih sprejemnih testov, normaliziran s**

---

[25] V tuji literaturi se uporablja izraz "*external quality*".

[26] *defect*

[27] *internal quality*

porabljenim časom. Normalizacija glede na porabljeni čas je potrebna zato, ker so razvijalci porabili za posamezno kodo različno količino časa, kar posledično seveda vpliva na število napak v programu. Odstotek uspelih sprejemnih testov smo tako delili s porabljenim časom (v urah). Vsak neuspeli sprejemni test torej štejemo kot napako v programu, ki kot taka direktno vpliva na zmanjšano funkcionalnost izdelka za stranko oz. naročnika.

Vseh sprejemnih testov je bilo 122 za prvo zgodbo, 32 za drugo zgodbo, vseh skupaj je bilo torej 154.

4.2.1.1 Kontekst samostojnih razvijalcev

Tabela 4-4 prikazuje povzetek rezultatov meritev zunanje kvalitete v kontekstu samostojnih razvijalcev (SR04), graf meritev je na sliki Slika 4-4.

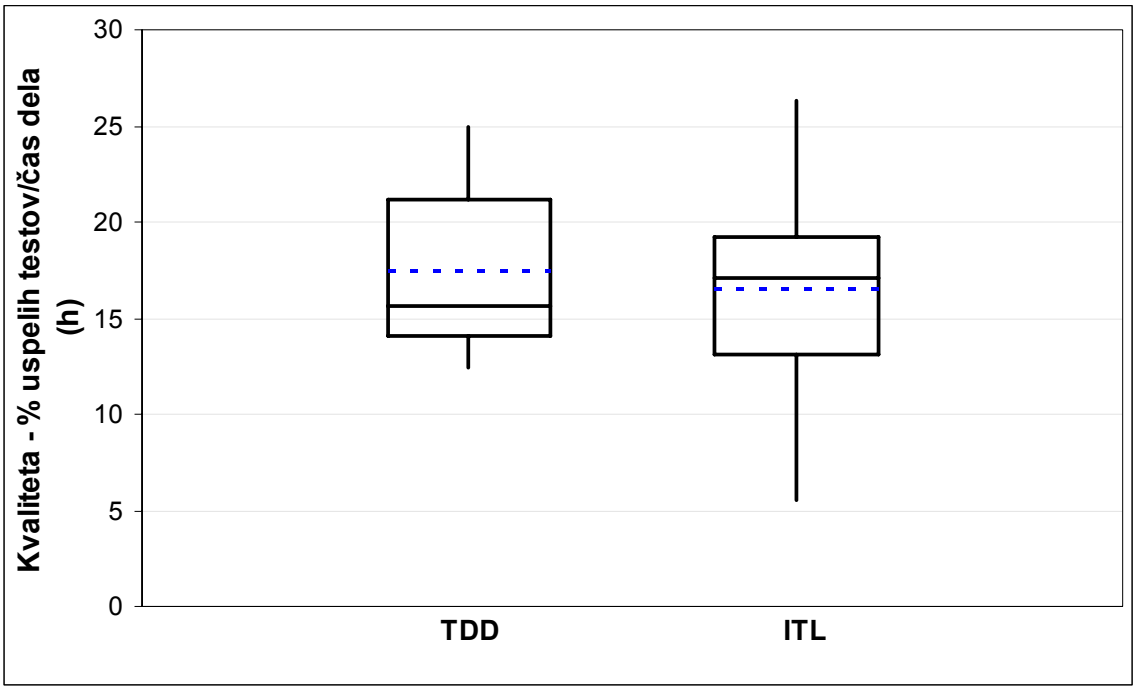
*Tabela 4-4: Rezultati meritev zunanje kvalitete kode (% uspelih sprejemnih testov/čas dela v urah (h)) – **samostojni razvijalci 04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	14	17,5	4,15	15,63	0,26	0,22
ITL	18	16,5	4,64	17,04		

Vidimo lahko, da je zunanja kvaliteta pri TDD skupini malo večja kot pri ITL skupini (6%). Efekt je torej v pravi smeri, vendar **razlika statistično ni značilna** (p=0,26). Zato **ne moremo zavrniti ničelne hipoteze KVALZ(H_0)**: $KVALZ_{TDD} = KVALZ_{ITL}$. Velikost efekta je majhna.

Zanimivo je tudi dejstvo, da so meritve ITL skupine precej bolj razpršene (imajo večji obseg vrednosti). To lahko nakazuje dejstvo, da je zunanja kvaliteta v tem procesu bolj odvisna od sposobnosti posameznega razvijalca kot pa od pri procesu TDD. Zaradi manjše razpršenosti je **lahko proces TDD bolj predvidljiv** glede zunanje kvalitete.

Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v zunanji kvaliteti med ITL in TDD skupino nismo zaznali.**



Slika 4-4: Zunanja kvaliteta kode (% uspelih sprejemnih testov/čas dela (h)) – ***samostojni razvijalci 04(SR04)***

4.2.1.2 Kontekst programiranja v parih

Zunanjo kvaliteto izračunavamo samo v eksperimentu s samostojnimi razvijalci, saj smo jo za eksperiment programiranja v parih predpisali zelo podrobno in so razvijalci morali popravljati in izboljševati svoje izvedbe toliko časa, da so **vsi sprejemni testi uspeli** (vseh preverjanj pri sprejemnih testih je bilo v letu 2003 več kot 700, v letu 2004 pa celo več kot 1100). Pri **eksperimentu s pari** je torej **zunanja kvaliteta konstantna** in jo kot takšno **ne merimo.** Seveda pa merimo notranjo kvaliteto, saj je dobra zunanja kvaliteta pri slabi notranji kvaliteti za stranko in razvijalca prav tako slaba rešitev – glej poglavje 4.2.2.

4.2.2 Notranja kvaliteta

Množica možnih metrik za notranjo kvaliteto kode je bistveno večja kot množica tistih, s katerimi lahko merimo zunanjo kvaliteto. Sami smo se odločili za preprosto metriko **"povprečno število poti skozi kodo metod"** (t.i. **McCabe-ovo ciklomatično**

kompleksnost) oziroma povprečno število neodvisnih poti skozi graf vseh možnih poti skozi kodo, ki je groba ocena kompleksnosti napisane kode. Več kot je možnih neodvisnih poti skozi kodo, bolj je koda kompleksna, torej je več možnosti za napake. Tako kodo je težje razumeti, testirati in spreminjati – torej je notranja kvaliteta take kode slabša. McCabe-ovo kompleksnost izračunamo za vsako metodo posebej, nato pa izračunamo še povprečje glede na vse metode v kodi. Tako npr. razvijalec, ki v skladu z dobro objektno orientirano prakso kodiranja kodo razbija (proces preoblikovanja!) v večje število enostavnejših metod in razredov, posledično doseže manjšo McCabe-ovo ciklomatično kompleksnost in s tem večjo notranjo kvaliteto.

Povzemimo: **manjše kot je povprečno število poti skozi kodo, večja je notranja kvaliteta kode**.

Ciklomatična kompleksnost kode naj bi pri iterativnem razvoju ne bila odvisna od časa razvoja, saj se zaradi iterativnega dela in procesa preoblikovanja sproti odpravlja prekompleksna koda in se s tem tudi sproti zmanjšuje njena povprečna vrednost. Zato te metrike ni potrebno normalizirati s časom razvoja.

Kodo lahko štejemo kot zelo kompleksno takrat, ko je McCabe-ova kompleksnost večja od 10. Takšno kodo je tudi zapleteno testirati.

4.2.2.1 Kontekst samostojnih razvijalcev

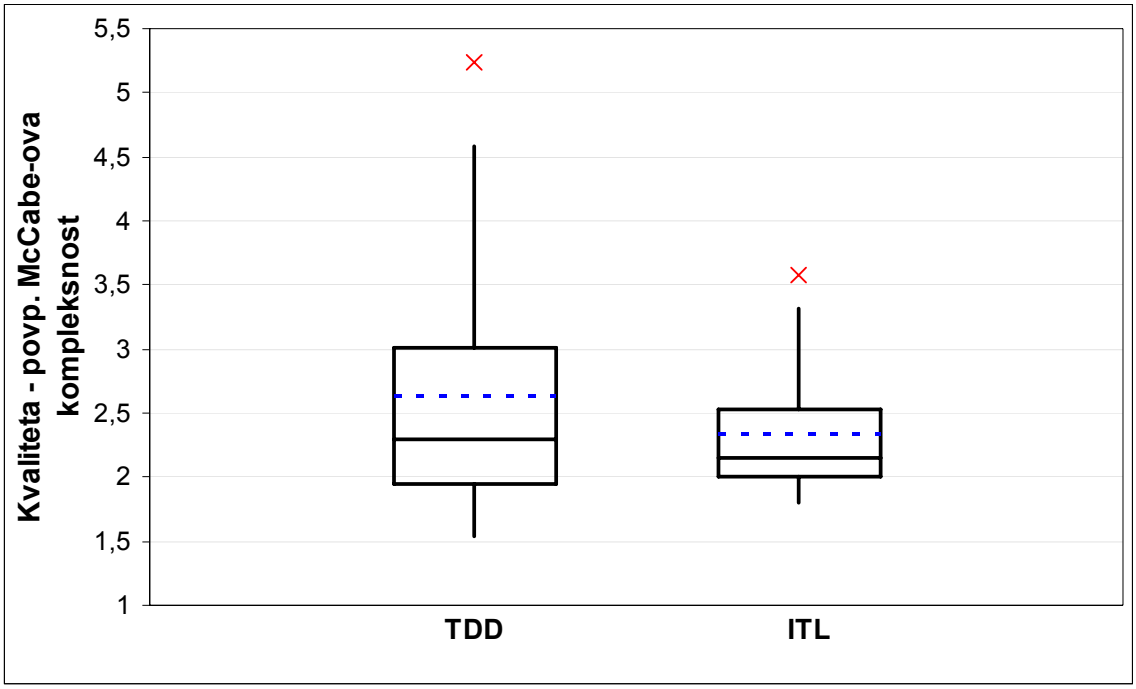
Tabela 4-5 prikazuje povzetek rezultatov meritev notranje kvalitete kode v kontekstu samostojnih razvijalcev (SR04), Slika 4-5 pa graf meritev.

Tabela 4-5: Rezultati meritev notranje kvalitete (povp. št. poti skozi metodo kode) – ***samostojni razvijalci 04 (SR04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	14	2,63	0,97	2,29	0,36	0,37
ITL	18	2,34	0,51	2,15		

Efekt **ni v predvideni smeri** (ITL ima boljšo notranjo kvaliteto kot TDD), vendar razlika ni statistično značilna (p=0,36). Zato v kontekstu samostojnih razvijalcev **ne moremo zavrniti ničelne hipoteze KVALN(H_0)**: $KVALN_{TDD} = KVALN_{ITL}$.

Velikost efekta po Cohenu je majhna do srednje velika, vendar je tu potrebno stvari pogledati še z zornega kota absolutnih vrednosti metrike. Absolutna razlika v povprečnih vrednostih notranje kvalitete je tu le 0,3 poti več v TDD skupini. Lahko trdimo, da je ta razlika v praksi tako majhna, da je zanemarljiva.



Slika 4-5: Notranja kvaliteta (povp. št. poti skozi metodo kode) – ***samostojni razvijalci 04***

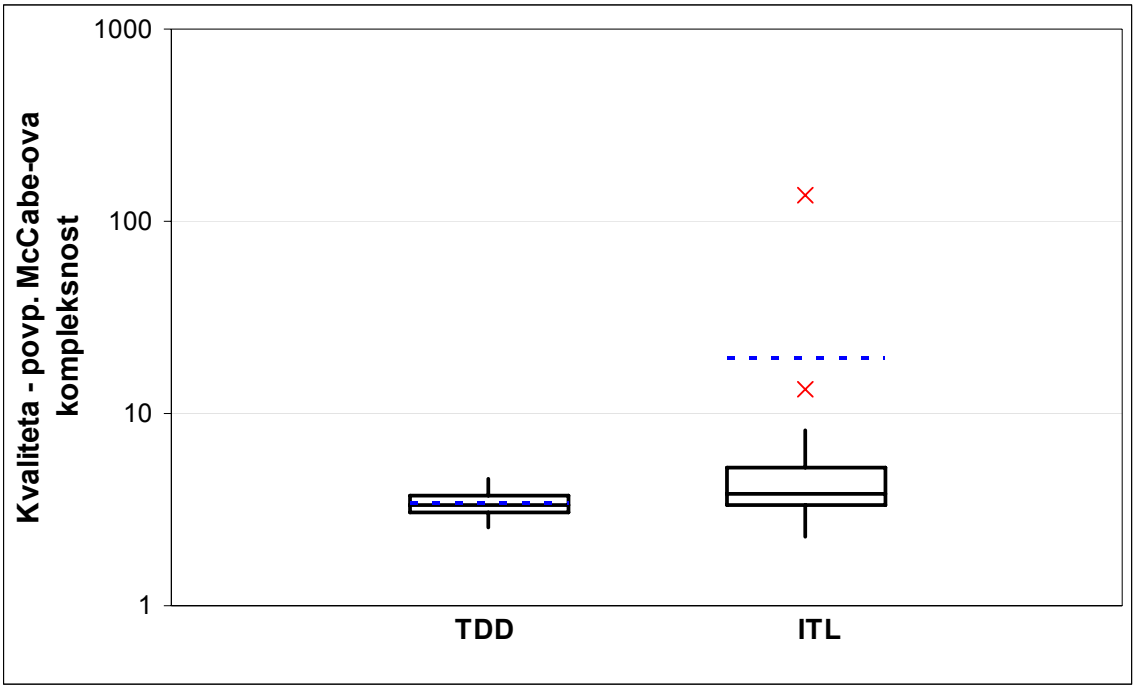
Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v notranji kvaliteti med ITL in TDD skupino nismo zaznali.**

4.2.2.2 Kontekst programiranja v parih

Tabela 4-6 prikazuje povzetek rezultatov meritev notranje kvalitete kode v kontekstu parov leta 2004 (PP04), Tabela 4-7 pa meritve za leto 2003 (PP03). Slika 4-6 in Slika 4-7 prikazujeta grafe za leti 2004 in 2003.

*Tabela 4-6: Rezultati meritev notranje kvalitete (povp. št. poti skozi kodo metode) – **pari 04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	3,44	0,66	3,34	0,11	0,51
ITL	9	19,73	44,47	3,857		



*Slika 4-6: Notranja kvaliteta (povp. št. poti skozi kodo metode) – **pari 04** (logaritemska skala na osi Y)*

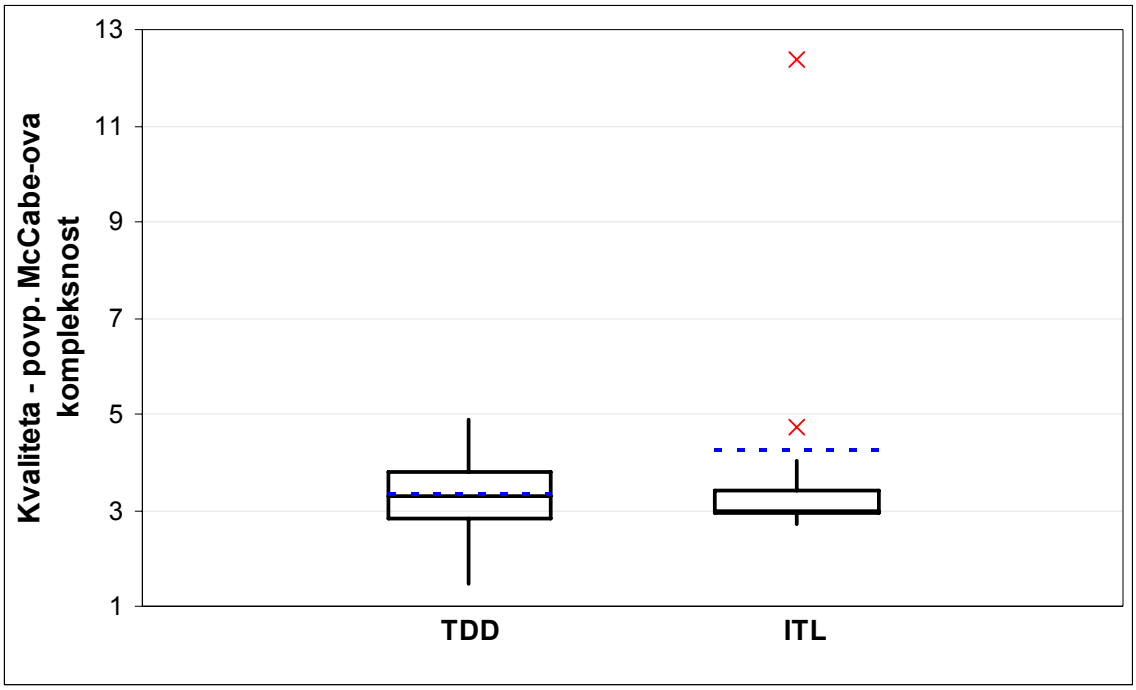
Razlika v notranji kvaliteti je za leto 2004 zelo velika, kar 5,7 kratna! Vendar tako veliko razliko ustvarita le dve meritvi (glej graf za skupino ITL, skala na osi Y je logaritemska). Ob analizi in preverjanju te metrike se je izkazalo, da sta dva para napisala vsak le po eno izrazito kompleksno in dolgo metodo z ogromnim številom pogojnih *if-then-else* konstruktov. Ker sta se oba držala predpisanega ITL procesa dela (kdaj se katera od metod začne preoblikovati v procesu ITL je namreč prepuščeno razsojanju razvijalca), smo obe meritvi proglasili za veljavni.

Podoben primer, le ne tako ekstremen, se je zgodil v letu 2003. Razlika v notranji kvaliteti je v tem letu 30%. Razlika je tudi tu nastala predvsem zaradi ene slabe in kompleksne rešitve.

Efekt je v predvideni smeri, vendar **razlika statistično ni značilna** ne za leto 2004 (p=0,11) ne za leto 2003 (p=0,46). Zato v kontekstu parov **ne moremo zavrniti ničelne hipoteze KVALN(H_0)**: $KVALN_{TDD} = KVALN_{ITL}$. Velikost efekta *d* po Cohenu je **srednje velika**.

*Tabela 4-7: Rezultati meritev notranje kvalitete (povp. št. poti skozi kodo metode) – **pari 03 (PP03)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	10	3,31	0,93	3,28	0,46	0,41
ITL	9	4,27	3,09	3		



*Slika 4-7: Notranja kvaliteta (povp. št. poti skozi kodo metode) – **pari 03 (PP03)***

Zanimivo pa je, da je do zelo slabe notranje kvalitete (McCabe-ova kompleksnost več kot 10) prišlo izključno v primeru procesa razvoja ITL (in to kar trikrat v dveh različnih

eksperimentih - PP03 in PP04!). Iz tega bi lahko sklepali, da proces razvoja **TDD razvijalce bolj usmerja k odpravljanju slabe kode** kot ITL oziroma da je možnost, da bodo razvijalci kar pustili slabo in kompleksno kodo nepreoblikovano, pri procesu ITL večja kot pri TDD.

Skupna ugotovitev glede kvalitete, tako zunanje kot notranje v obeh kontekstih uporabe:

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v kvaliteti. Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da velikih razlik med procesom TDD in ITL glede kvalitete ni.

Z ozirom na izvedene meritve ocenjujemo, da gre lahko v primeru zunanje kvalitete kvečjemu za majhen efekt, v primeru notranje kvalitete pa kvečjemu srednje velik efekt.

4.3 Testi

Ker je faza testiranja pri ITL in TDD procesu razvoja izrednega pomena, smo temu parametru razvojnega procesa posvetili posebno pozornost. Zanimala sta nas predvsem število testov in njihova kvaliteta.

Za metriko, ki nam poda **število testov**, smo izbrali **št. preverjanj**[28] **v testni kodi, normalizirano s porabljenim časom**. Za štetje preverjanj in ne testnih primerov[29] smo se odločili zato, ker je možno testne primere (metode) v ogrodju JUnit pisati z različno granulacijo. Nekateri študenti so nagnjeni k temu, da imajo manjše število bolj kompleksnih testov, drugi pa k večjemu številu manj kompleksnih testov. Da bi uravnotežili in odpravili to neskladje, smo torej tu prešteli vsa logična preverjanja v testni kodi. S tem smo povsem odpravili napako, ki bi nastala, če bi šteli le testne primere, ki so zaradi različne granulacije od primera do primera precej različni. Normalizacija s porabljenim časom je tu potrebna zaradi različno dolgega dela na eksperimentalnih nalogah, saj je število testov v korelaciji s porabljenim časom za testiranje.

Merilo za **kvaliteto testov** je **pokritost kode.** Pokritost kode nam pove, v kolikšnem delu razvijalčeva testna koda "pokrije" funkcionalno kodo. Sestavljena je iz treh komponent:

- pokritosti metod POK_m (*method coverage*)

  Pokritost metod je definiran kot odstotek vseh metod v kodi, ki jih testi aktivirajo (izvedejo) vsaj enkrat (N_m = število vseh metod v kodi). Torej je:

$$POK_m = \frac{N_m(izvedenih)}{N_m} \tag{4.1}$$

- pokritosti stavkov POK_s (*statement coverage*)

  Pokritost stavkov je definirana kot odstotek vseh stavkov, ki jih testi aktivirajo (izvedejo) vsaj enkrat (N_s = število vseh stavkov v kodi). Torej je:

---

[28] Stavke "`assert`", ki smo jim prišteli tudi št. stavkov "`fail`" – glej dokumentacijo ogrodja JUnit. (http://www.junit.org)

[29] *Test case* - v ogrodju JUnit ena metoda v razredu `TestCase`, ki predstavlja en posamezen testni primer.

$$POK_s = \frac{N_s(izvedenih)}{N_s} \tag{4.2}$$

- pokritosti pogojev POK_p (*branch coverage*)

  Pogoje (oz. pogojni stavki v kodi) je potrebno ovrednotiti po obeh vejah: ko je pogoj resničen in ko je pogoj neresničen, saj vsaka od obeh vej določa drugo pot skozi program (N_p = število pogojnih stavkov v kodi, na primer stavkov *if*, *while*, *case*,…). Pokritost pogojev je torej:

$$POK_p = \frac{N_p(izvedenih \mid pogoj\ resnicen) + N_p(izvedenih \mid pogoj\ neresnicen)}{2 * N_p} \tag{4.3}$$

Skupno pokritost kode (*POK_{skupna}*) izračunamo po sledeči formuli (4.4):

$$POK_{skupna} = \frac{N_m(izvedenih) + N_s(izvedenih) + N_p(izvedenih \mid pogoj\ resnicen) + N_p(izvedenih \mid pogoj\ neresnicen)}{N_m + N_s + 2 * N_p}$$

Popolna (100%) pokritost kode nam tako zagotavlja, da je testna koda izvedla vse možne poti skozi kodo **vsaj enkrat** (ter da so pokriti vsi stavki in da je vsaka metoda izvedena najmanj enkrat). Tako, kot je pokritost definirana, sama po sebi seveda ne more zagotavljati tudi logične pravilnosti testov (torej ali testi sploh testirajo tisto, kar naj bi testirali), je le ocena za popolnost oziroma kvaliteto testne kode. Logična pravilnost testiranja je tako kot zunanja kvaliteta odvisna od posameznega razvijalca in njegovega razumevanja problema.

Pri izračunavanju kvalitete testov smo si pomagali z orodjem Clover (http://www.cenqua.com/clover), katerega brezplačno uporabo so nam dovolili avtorji posebej za našo raziskavo.

4.3.1 Število testov

Število testov smo merili z metriko števila preverjanj v testih na porabljeno uro dela. Ne smemo pa te metrike enačiti s številom napisanih testov na uro dela! En test (v ogrodju JUnit je to ena testna metoda) ponavadi vsebuje več logičnih preverjanj stanja, še posebej to velja za bolj kompleksne teste, ki jih je nemogoče napisati samo z enim logičnim preverjanjem.

Zato je tu potrebno biti previden pri absolutnem primerjanju naših rezultatov meritev z meritvami drugih avtorjev, saj so le-ti večinoma objavljali kar število testnih metod na časovno enoto. Štetje le števila testov oziroma testnih metod pa je lahko, kot smo že omenili, zaradi različne granulacije testov precej zavajajoče. Tudi sami smo pri pregledu testne kode naleteli na zelo raznoliko granulacijo testov – nekateri subjekti so za logično testiranje ene funkcionalnosti napisali le par testnih metod, drugi pa za isto stvar večje število bolj preprostih testnih metod.

4.3.1.1 Kontekst samostojnih razvijalcev

Pri analizi dela števila testov pri samostojnih razvijalcih smo zaradi različne granulacije dela v ITL in TDD skupini to metriko izračunali le za prvo zgodbo, ki smo jo normalizirali s porabljenim časom za njeno dokončanje. Druge zgodbe nismo upoštevali zato, ker je bil čas za delo pri eksperimentu s samostojnimi razvijalci omejen, granulacija dela pri ITL skupini pa je zaradi procesa razvoja večja kot pri TDD skupini. Ker so morali vsi, tako ITL kot TDD skupina končati delo po izteku časa ne glede na to, do kam so naredili drugo zgodbo, bi bili rezultati štetja testov pri ITL skupini lahko zavajajoči (manjši, saj bi bilo veliko razvijalcev ITL skupine sredi kodiranja funkcionalnosti, testov pa še niso imeli časa napisati; obratno velja tudi za TDD skupino).

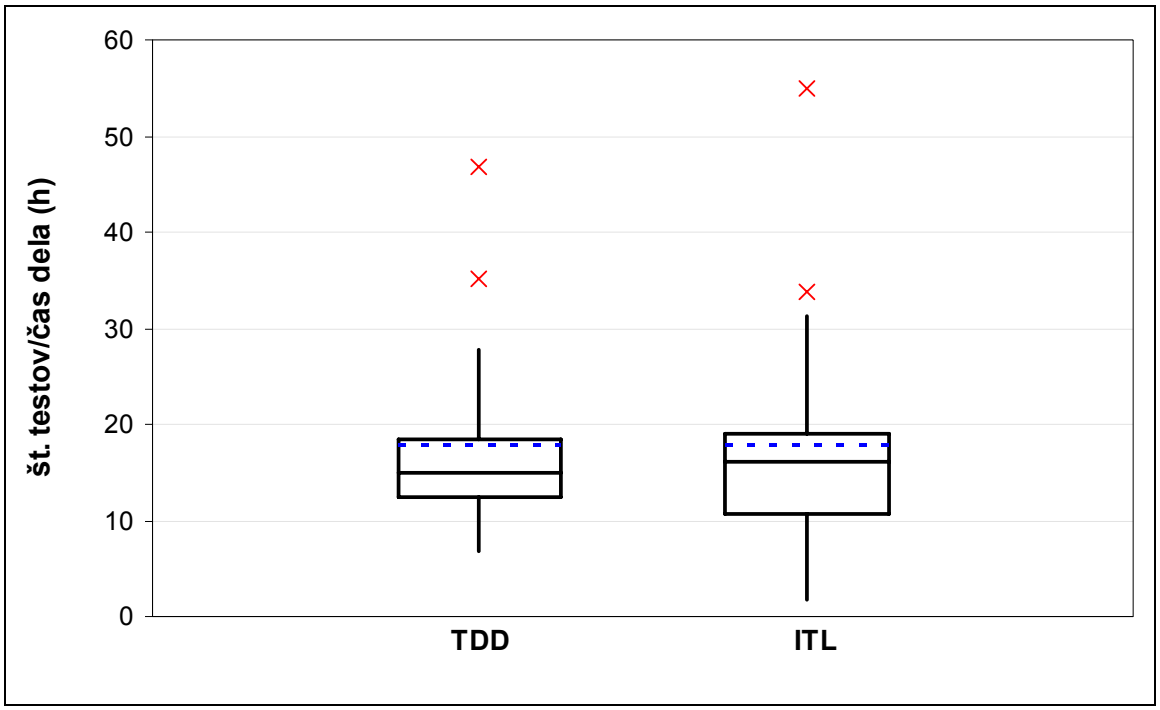
Tabela 4-8 prikazuje povzetek rezultatov meritev števila testov v kontekstu samostojnih razvijalcev (SR04), Slika 4-8 pa ustrezen graf meritev.

Tabela 4-8: Rezultati meritev št. testov (št. preverjanj/čas dela (h)) – ***samostojni razvijalci 04 (SR04)***

Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p***	**Cohen *d***
TDD	14	17,84	10,73	14,88	0,48	<0,1
ITL	18	17,77	11,99	16,10		

Efekt je v predvideni smeri, vendar je absolutna razlika zanemarljivo majhna in statistično ni značilna (p=0,48). Zato v kontekstu samostojnih razvijalcev **ne moremo zavrniti ničelne hipoteze TESTŠT(H_0)**: $TESTŠT_{TDD} = TESTŠT_{ITL}$. Tudi velikost efekta po Cohen-u je zanemarljivo majhna.

Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v številu testov med ITL in TDD skupino nismo zaznali.**



*Slika 4-8: Št. testov (št. preverjanj/čas dela (h)) – **samostojni razvijalci 04 (SR04)***

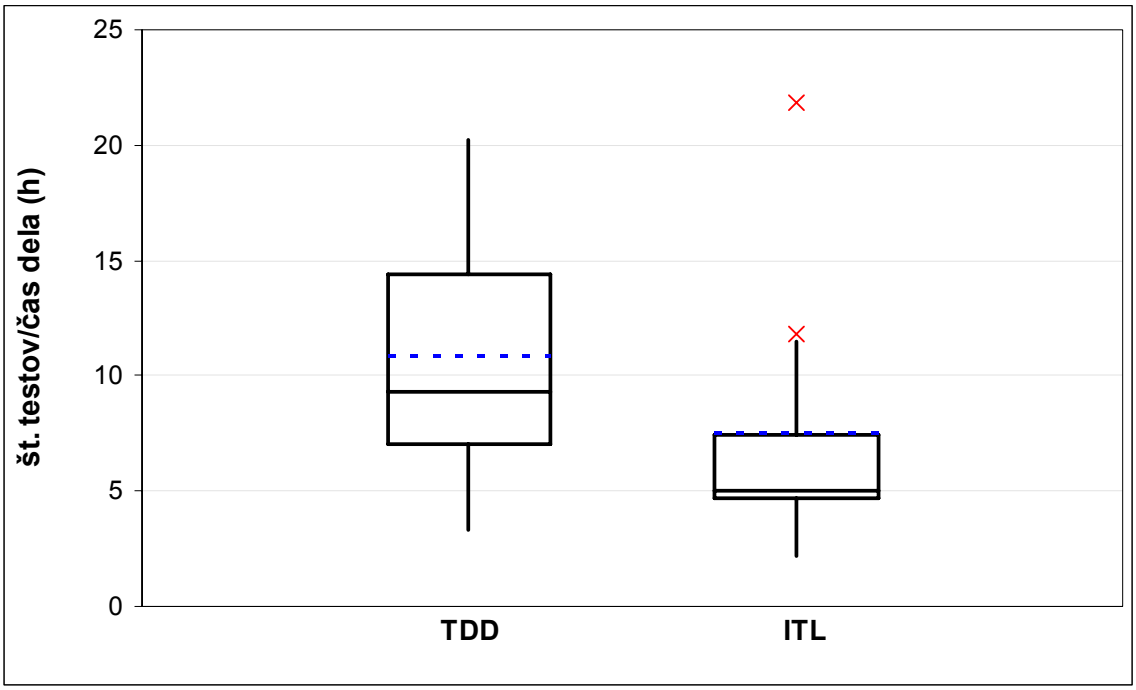
4.3.1.2 Kontekst programiranja v parih

Tudi v kontekstu parov je metrika za število testov število preverjanj na uro dela, tokrat seveda zaradi drugačnih pogojev dela izračunana za celotno eksperimentalno nalogo.

Tabela 4-9 prikazuje povzetek rezultatov meritev števila testov v kontekstu parov leta 2004 (PP04), Tabela 4-10 pa meritve za leto 2003 (PP03). Slika 4-9 in Slika 4-10 prikazujeta grafe meritev za leti 2004 in 2003.

*Tabela 4-9: Rezultati meritev št. testov (št. preverjanj/čas dela (h)) – **pari 04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	10,82	6,18	9,32	0,13	0,54
ITL	9	7,51	6,04	5,04		



Slika 4-9: Št. testov (št. preverjanj/čas dela (h)) – ***pari 04 (PP04)***

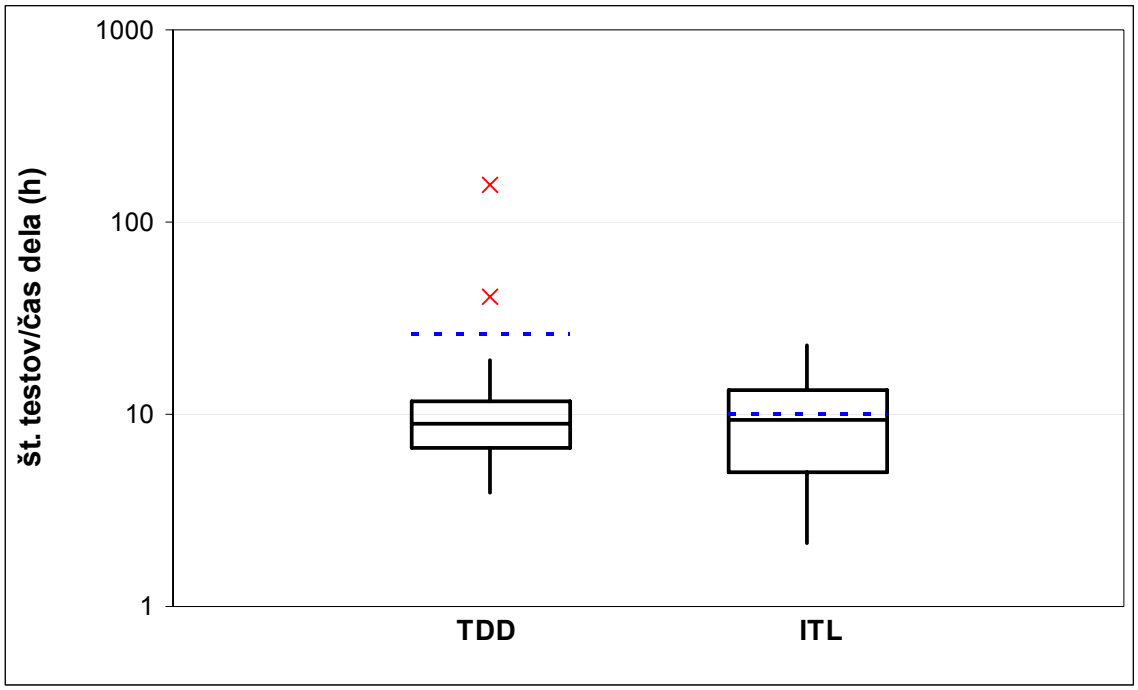
Absolutne razlike med povprečnim številom testov so tu precej v prid procesu razvoja TDD. V letu 2004 je tako razlika več kot 40%, v letu 2003 pa zaradi dveh ekstremnih meritev v TDD skupini kar 2,6-kratna! Ob analizi dnevnika razvoja in verzij kode smo ugotovili, da sta si para z ekstremnimi vrednostmi zelo dobro pripravila ogrodje za testiranje (s preoblikovanjem skupnih testnih nastavitev v pomožne metode), s katerim sta potem lahko zelo učinkovito in zelo hitro dodajala testne primere. Od tod tudi sledi izjemno veliko število preverjanj na uro dela. Meritvi smo zaradi tega, ker sta zadovoljevali vse naše kriterije, upoštevali v analizi.

Efekt je v predvideni smeri, vendar **razlika statistično ni značilna** ne za leto 2004 (p=0,13) ne za leto 2003 (p=0,45). Zato v kontekstu parov **ne moremo zavrniti ničelne hipoteze TESTŠT(H_0)**: $\text{TESTŠT}_{\text{TDD}} = \text{TESTŠT}_{\text{ITL}}$. Velikost efekta d po Cohenu je za oba eksperimenta **srednje velika**.

Sklenimo: med TDD in ITL skupino nismo zaznali statistično značilnih razlik v številu testov. Glede na meritve ocenjujemo, da gre lahko v primeru števila testov kvečjemu za srednje velik efekt.

*Tabela 4-10: Rezultati meritev št. testov (št. preverjanj/čas dela (h)) – **pari 03 (PP03)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	10	25,9	46,78	8,92	0,45	0,47
ITL	9	10,10	6,7	9,38		



*Slika 4-10: Rezultati meritev št. testov (št. preverjanj/čas dela (h)) – **pari 03 (PP03)** (logaritemska skala na osi Y)*

4.3.2 Kvaliteta testov

Kvaliteto testov smo ocenili s odstotkom pokritosti kode. Za lažjo primerjavo absolutnih vrednosti naj omenimo, da so ocene glede stopnje pokritosti kode v industriji nekje v razredu 80%.

4.3.2.1 Kontekst samostojnih razvijalcev

Pri kvaliteti testov smo pri eksperimentu s samostojnimi razvijalci pokritost računali le za prvo zgodbo (od dveh definiranih). Razlog je podoben, kot smo ga omenili pri parametru število testov (različna granulacija dela pri TDD in ITL, kar lahko pokvari meritev pri

zadnji nedokončani zgodbi, ki smo jo prekinili ob izteku časa). Proces dela TDD sam po sebi usmerja razvijalca k čim boljši pokritosti (zagovorniki trdijo, da celo k 100% pokritosti). To v malo manjši meri velja tudi za proces dela ITL, vendar je tu kvaliteta testov bolj odvisna od izkušenosti razvijalca kot pri TDD, kjer je razvijalec skoraj prisiljen k pisanju kvalitetnih testov.

TDD skupina razvijalcev ima za doseganje večje kvalitete testov še eno orožje: ker naj bi se vsak test prvič zagnal še preden sploh napišemo kodo, ki jo test testira, lahko razvijalci v TDD dosegajo boljšo pokritost kode ravno zaradi tega, ker lahko s tem takoj izločijo teste, ki so jih napisali narobe (t.i. prazni testi oz. testi, ki vedno uspejo, ne glede na kodo).

Slika 4-11 in Tabela 4-11 prikazujeta meritve kvalitete testov v kontekstu samostojnih razvijalcev (SR04).

Tabela 4-11: Rezultati meritev kvalitete testov (odstotek pokritosti) – ***samostojni razvijalci (SR04)***

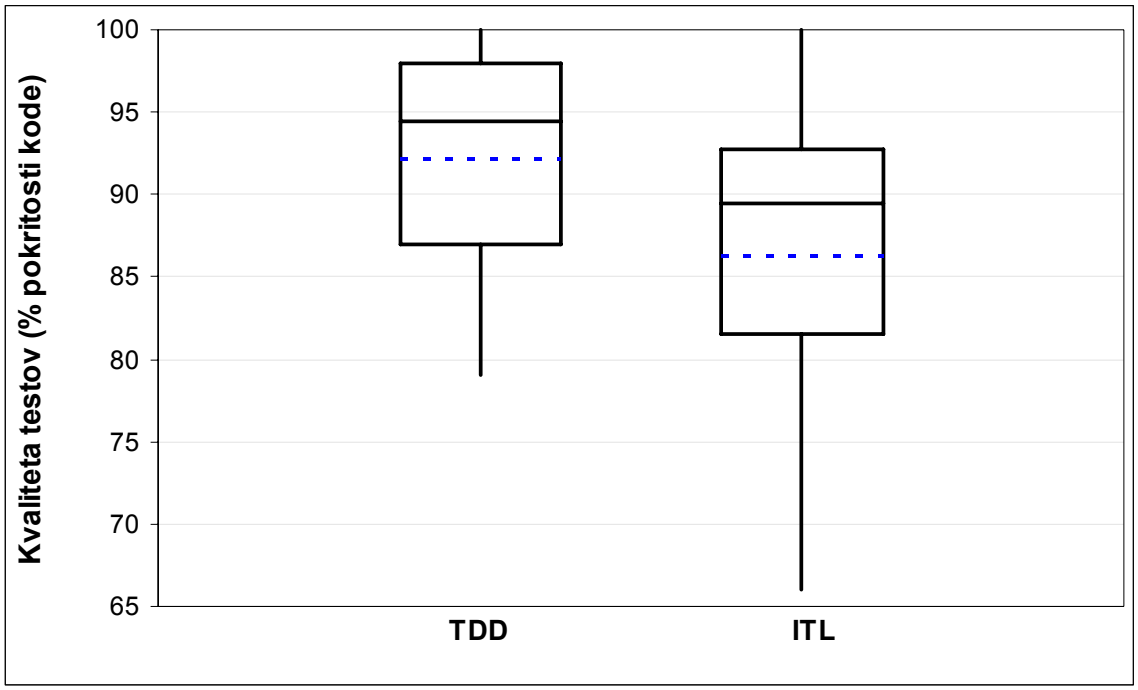
Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p***	**Cohen *d***
TDD	14	92	7,27	94,5	0,04	0,69
ITL	18	86	9,6	92,12		

Efekt je v pravi smeri in **je statistično značilen (*p*=0,04).** Zato lahko v kontekstu samostojnih razvijalcev **zavrnemo ničelno hipotezo TESTKVL(H_0)**: $TESTKV_{TDD}$ = $TESTKV_{ITL}$ in **sprejmemo alternativno hipotezo TESTKVL**

$$\mathbf{TESTKVL(H_a): TESTKV_{TDD} > TESTKV_{ITL}.}$$

Ocena velikosti efekta *d* po Cohenu je velika, relativna razlika med kvaliteto testov v procesu razvoja TDD in ITL pa je za 7% v prid TDD.

Sklenimo: v kontekstu samostojnih razvijalcev smo med TDD in ITL skupino zaznali statistično značilne razlike (*p*=0,04) v kvaliteti testov (razlika za 7%). Po Cohenu gre za tu za velik efekt.



Slika 4-11: Kvaliteta testov (% pokritosti kode) – ***samostojni razvijalci 04 (SR04)***

4.3.2.2 Kontekst programiranja v parih

Tabela 4-12 prikazuje povzetek rezultatov meritev števila testov v kontekstu parov leta 2004 (PP04), Tabela 4-13 pa meritve za leto 2003 (PP03). Slika 4-12 in Slika 4-13 prikazujeta grafe meritev za leti 2004 in 2003.

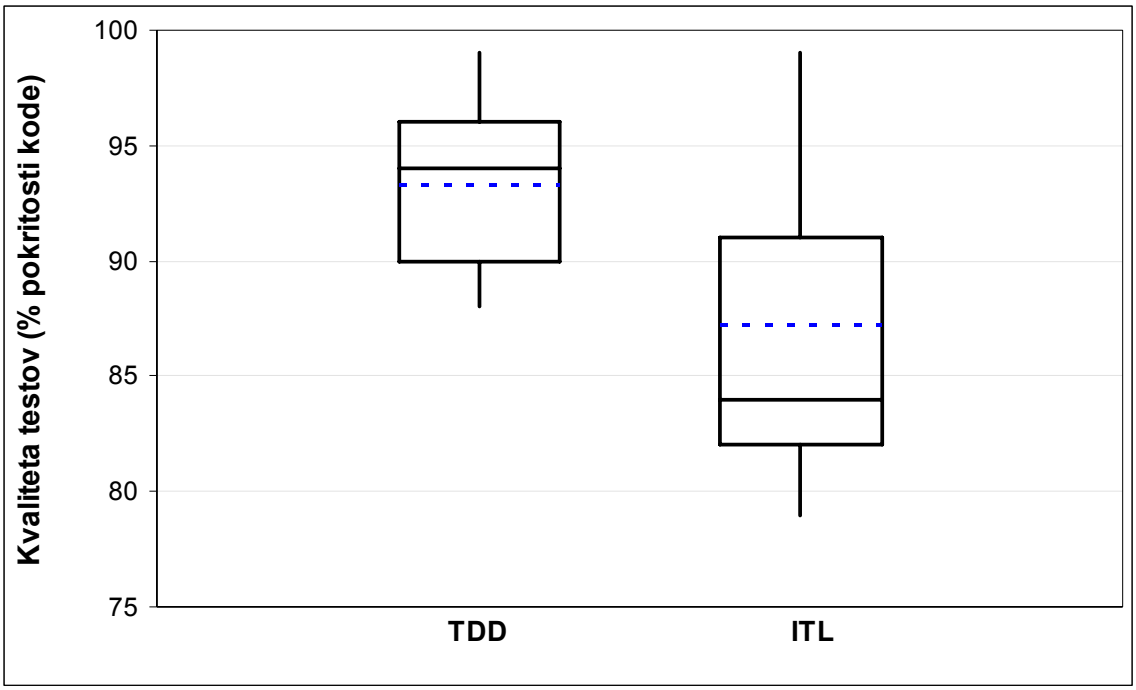
Tabela 4-12: Rezultati meritev kvalitete testov (odstotek pokritosti) – ***pari 04 (PP04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	93	4,19	94	0,05	1,02
ITL	9	87	7,27	84		

Za **leto 2004** lahko sklenemo, da je efekt v pričakovani smeri in **je statistično značilen (*p*=0,05).** Ocena velikosti efekta *d* po Cohenu je velika, relativna razlika med kvaliteto testov v procesu razvoja TDD in ITL pa je za 7% v prid TDD.

Za **leto 2003** pri parih (PP03) iz podatkov vidimo, da efekt ni v pravi smeri (povprečni vrednosti sta namreč enaki), vendar je zato mediana pri TDD skupini za 2% višje kot pri ITL skupini, zato menimo, da je vseeno prednost pri TDD skupini. Razlika med

povprečnima vrednostma kvalitete testov **ni statistično značilna (*p*=0,52).** Ocena velikosti efekta *d* po Cohenu je majhna.



Slika 4-12: Kvaliteta testov (% pokritosti kode) – ***pari 04 (PP04)***

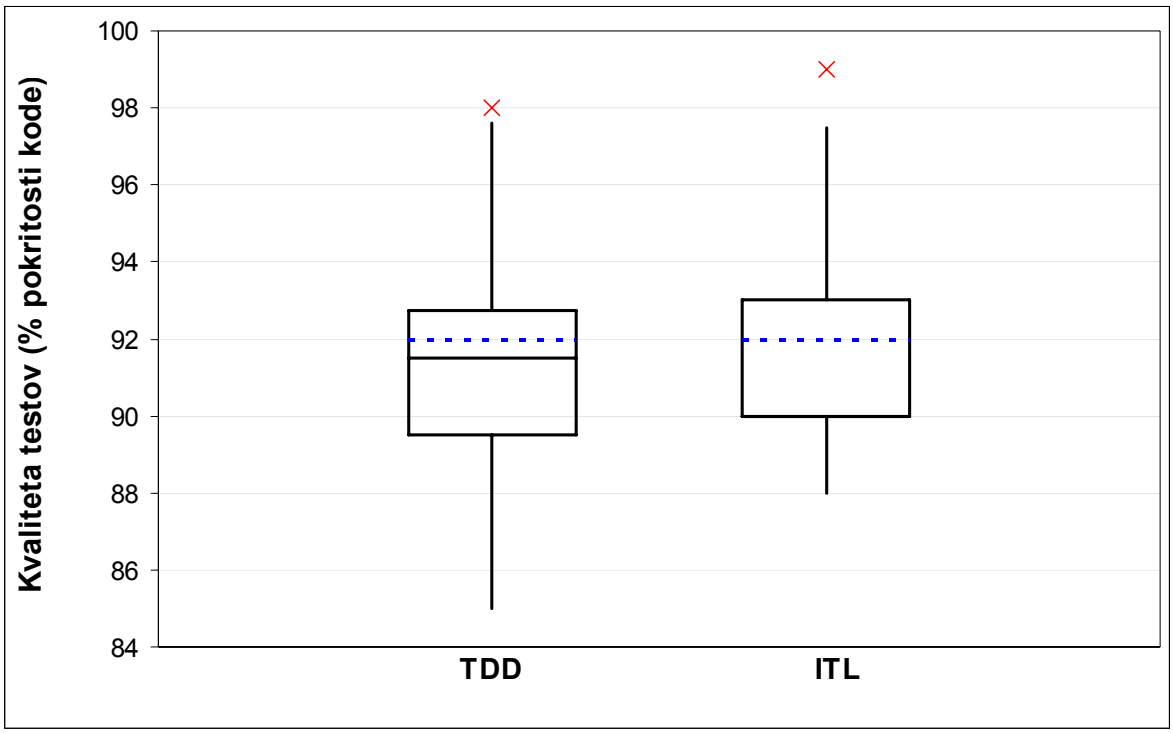
Zaradi dejstva, da imajo testi pri delu s pari malo manjšo statistično moč zaradi manjšega vzorca (delo v parih prepolovi število posameznih meritev), niti nismo pri eksperimentu s pari pričakovali, da bo efekt vedno statistično značilen, kot se je to zgodilo v eksperimentu PP03. Spomnimo se: manjšo statistično moč smo nekoliko kompenzirali z replikacijo eksperimenta.

Tabela 4-13: Rezultati meritev kvalitete testov (odstotek pokritosti) – **pari 03 (PP03)**

Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p***	**Cohen *d***
TDD	10	92	4	91,5	0,52	0,1
ITL	9	92	3,85	90		

Zato menimo, da lahko **tudi v kontekstu parov zavrnemo ničelno hipotezo TESTKVL(H_0)**: $TESTKV_{TDD}$ = $TESTKV_{ITL}$ in **sprejmemo alternativno hipotezo TESTKVL**

TESTKVL(H_a): $TESTKV_{TDD} > TESTKV_{ITL}$.



Slika 4-13: Kvaliteta testov (% pokritosti kode) – ***pari 03 (PP03)***

Povzemimo skupno ugotovitev glede števila in kvalitete testov v obeh kontekstih uporabe:

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v številu testov, zaznali pa smo statistično značilne razlike v kvaliteti testov.

Sklenemo lahko, da je v procesu razvoja TDD število testov enako kot v procesu razvoja ITL, precej večja pa je njihova kvaliteta (za 7%). Proces razvoja TDD torej usmerja razvijalce k pisanju bolj kvalitetnih testov kot proces razvoja ITL.

Z ozirom na izvedene meritve ocenjujemo, da gre lahko pri številu testov kvečjemu za srednje velik efekt.

V primeru razlike med TDD in ITL glede kvalitete testov gre za velik efekt v korist TDD.

4.4 O procesu razvoja

V tem poglavju smo zbrali podatke o parametrih, ki se tičejo samih značilnosti procesa razvoja. Zanimalo nas je, ali so kakšne razlike med primerjanima procesoma glede frekvence poganjanja testov, razmerja med testi in kodo (merjeno v vrsticah izvorne kode), odstotka uspešnih testov in koliko časa razvijalci prebijejo v posamezni fazi dela. Na podlagi teh značilnosti lahko vidimo slabosti in prednosti posameznega razvojnega procesa, zato bi lahko pri načrtovanju svoje metodologije dela vanjo vključili tudi kakšno komponento, ki kakšno od slabosti omili ali odpravi. Tako lahko na podlagi globljega poznavanja značilnosti posameznega razvojnega procesa lažje sestavljamo metodologijo, ki je kar najbolj prilagojena razvojni skupini.

4.4.1 Frekvenca poganjanja testov

Metrika, ki smo jo tu uporabili, je število zagonov testov, normalizirano s porabljenim časom (v urah).

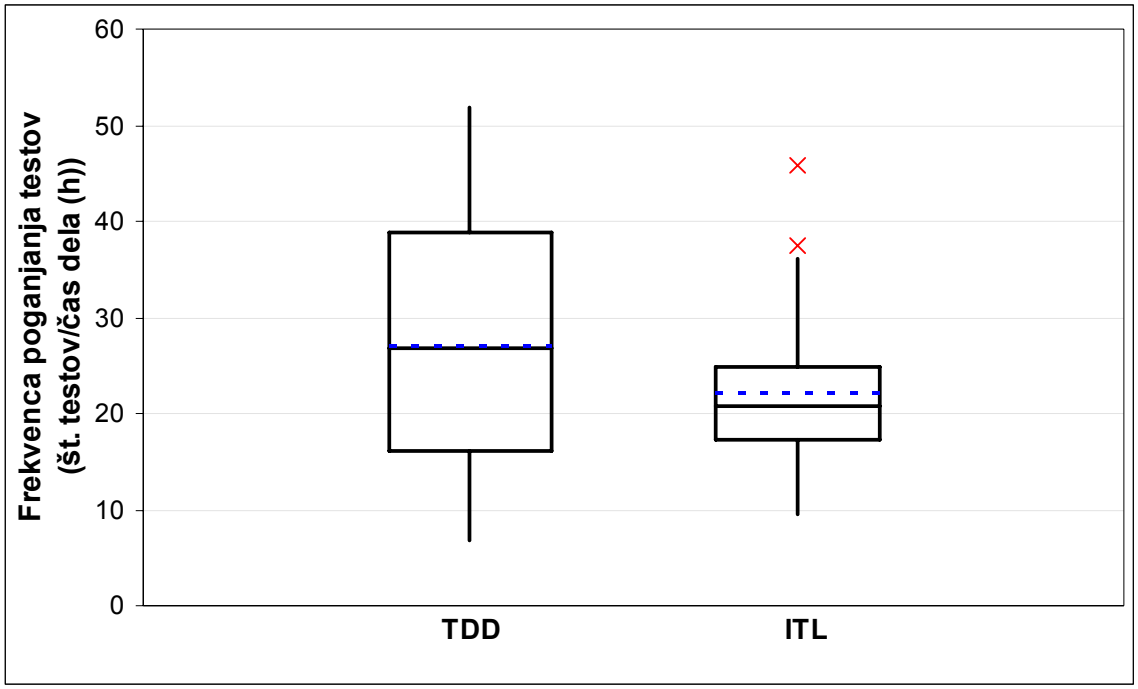
4.4.1.1 Kontekst samostojnih razvijalcev

Tu dveh meritev pri eksperimentu s samostojnimi razvijalci 04 v ITL skupini nismo mogli upoštevati zaradi napačnih nastavitev v razvojnem okolju (študenta sta najbrž napačno nastavila različico Jave; potrebovali smo najmanj JDK 1.4, onadva pa sta uporabila JDK 1.3, ki je bil tudi instaliran na računalniku). Zato se pri njima število zaganjanj testov ni beležilo v dnevniški datoteki našega programa ProcessLog. Podobna težava se je pojavila tudi pri enem paru v eksperimentu s pari 04. Tudi tu te meritve nismo mogli upoštevati. Z zmanjšano velikostjo vzorca v skupini pa se je rahlo tudi zmanjšala statistična moč testa, kar moramo upoštevati pri interpretaciji rezultatov. Enak razlog velja tudi pri naslednji metriki "odstotek uspešnih testov" (glej poglavje 4.4.2 Odstotek uspešnih testov).

Tabela 4-14 prikazuje povzetek rezultatov meritev frekvence poganjanja testov v kontekstu samostojnih razvijalcev (SR04), Slika 4-14 pa ustrezen graf meritev.

*Tabela 4-14: Rezultati meritev frekvence poganjanja testov (št. zagonov testov/čas dela(h)) – **samostojni razvijalci 04 (SR04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	14	27,07	13,74	26,71	0,19	0,41
ITL	16	22,17	9,64	20,76		



*Slika 4-14: Frekvenca poganjanja testov (št. zagonov/čas dela(h)) – **samostojni razvijalci 04***

Efekt je v predvideni smeri, vendar **razlika statistično ni značilna** (p=0,19). Zato v kontekstu samostojnih razvijalcev **ne moremo zavrniti ničelne hipoteze TESTFR(H_0)**: $TESTFR_{TDD} = TESTFR_{ITL}$. Velikost efekta po Cohen-u je srednje velika.

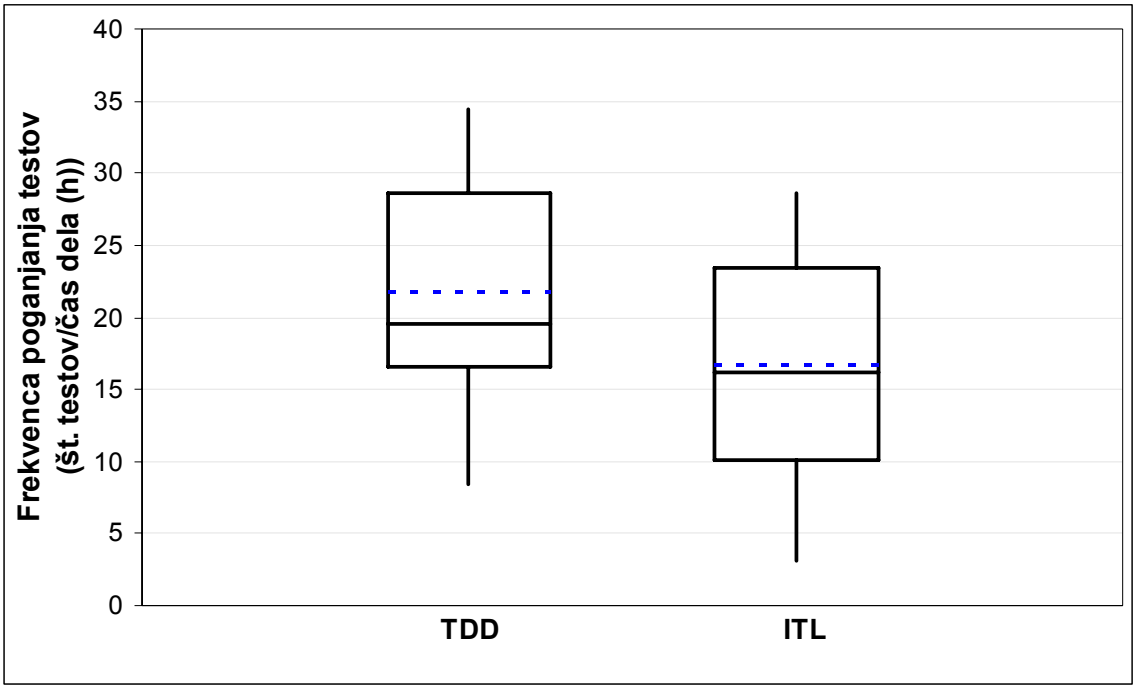
Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v frekvenci poganjanja testov med ITL in TDD skupino nismo zaznali.**

4.4.1.2 *Kontekst programiranja v parih*

Tabela 4-15 prikazuje povzetek rezultatov meritev frekvence poganjanja testov v kontekstu parov leta 2004 (PP04), Tabela 4-16 pa meritve za leto 2003 (PP03). Slika 4-15 in Slika 4-16 prikazujeta grafe meritev za leti 2004 in 2003.

*Tabela 4-15: Rezultati meritev frekvence poganjanja testov (št. zagonov testov/čas dela(h)) – **pari 04 (PP04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	21,81	9,09	19,53	0,2	0,56
ITL	8	16,63	9,36	16,16		



*Slika 4-15: Frekvenca poganjanja testov (št. zagonov/čas dela(h)) – **pari 04***

Za **leto 2004** lahko sklenemo, da je efekt v pravi smeri, vendar **razlika statistično ni značilna** (p=0,2). Velikost efekta po Cohen-u je srednje velika.

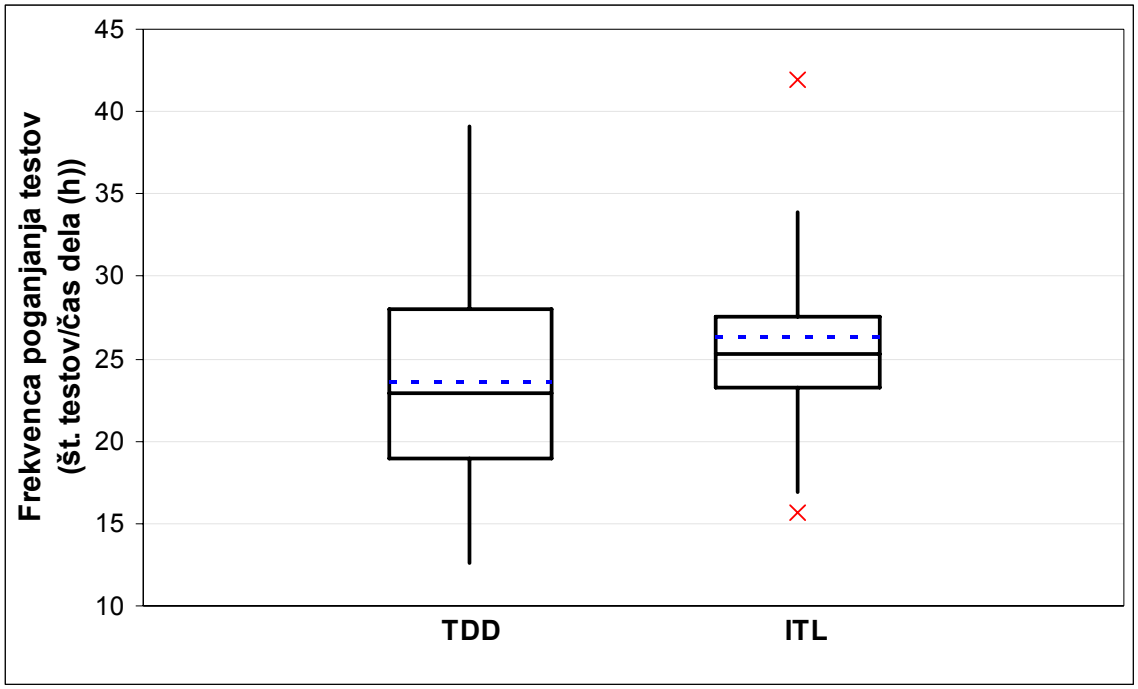
Za **leto 2003** pri parih (PP03) iz podatkov vidimo, da efekt ni v predvideni smeri (ITL skupina je tu v povprečju večkrat zagnala teste kot TDD skupina), zato tudi ne moremo zavrniti ničelne hipoteze TESTFR(H_0). Ob analizi meritev je razvidno, da k temu efektu

največ prispeva ena sama meritev, ki je izrazito večja od povprečja (kar 42 zagonov testov na uro, kar je absolutno največkrat med obema skupinama). Tu sta razvijalca izrazito velikokrat zagnala teste zaradi tega, ker sta imela težave pri pisanju večnitnega strežnika (problemi sinhronizacije niti), ki sta jih kar nekaj časa reševala s pogostim testiranjem. Razlika med povprečnima vrednostma frekvence poganjanja testov pri PP03 **ni statistično značilna** (p=0,18). Ocena velikosti efekta d po Cohenu je majhna do srednje velika. V kontekstu parov **ne moremo zavrniti ničelne hipoteze TESTFR(H_0)**: $TESTFR_{TDD} = TESTFR_{ITL}$.

Sklenemo lahko: v kontekstu parov **razlik v frekvenci poganjanja testov med ITL in TDD skupino nismo zaznali.**

*Tabela 4-16: Rezultati meritev frekvence poganjanja testov (št. zagonov testov/čas dela(h)) – **pari 03 (PP03)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	10	23,62	7,91	22,93	0,18	0,35
ITL	9	26,36	7,72	25,34		



*Slika 4-16: Frekvenca poganjanja testov (št. zagonov/čas dela(h)) – **pari 03***

Skupna ugotovitev glede frekvence poganjanja testov v obeh kontekstih uporabe:

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v frekvenci poganjanja testov. Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da velikih razlik med procesom TDD in ITL glede frekvence poganjanja testov ni.

Z ozirom na izvedene meritve ocenjujemo, da gre lahko v primeru frekvence poganjanja testov kvečjemu za srednje velik efekt.

4.4.2 Odstotek uspešnih testov

Pri delu smo merili tudi to, ali se je pravkar zagnani nabor testov končal z uspehom (**vsi** testi uspešni), ali pa je bil **najmanj en** test neuspešen. Razvijalci v TDD procesu bodo imeli po pričakovanjih tu zaradi specifičnosti TDD procesa razvoja večje število neuspešnih testov kot razvijalci z ITL procesom razvoja. Spomnimo se: pri TDD mora razvijalec najprej napisati test, nato ga pognati (test bo neuspešen, saj še ni kode, ki jo testira), šele nato doda ustrezno funkcionalnost.

Tudi tu nam iz enakih razlogov, kot smo jih opisali že v poglavju 4.4.1 Frekvenca poganjanja testov, manjkata 2 meritvi pri eksperimentu samostojni razvijalci 04 in ena meritev pri eksperimentu pari 04. Manjši vzorec tudi vpliva na manjšo statistično moč testa, kar je potrebno upoštevati pri končnem sklepu.

To metriko lahko tudi jemljemo kot preprost preizkus, ali so se v povprečju razvijalci TDD skupine držali do potankosti prvega dela predpisanega procesa razvoja (torej da teste pred prvo uporabo preizkusijo – torej morajo biti najmanj enkrat neuspešni). Naša hipoteza je torej, da bodo imeli razvijalci TDD skupine manjši odstotek uspešnih testov.

4.4.2.1 Kontekst samostojnih razvijalcev

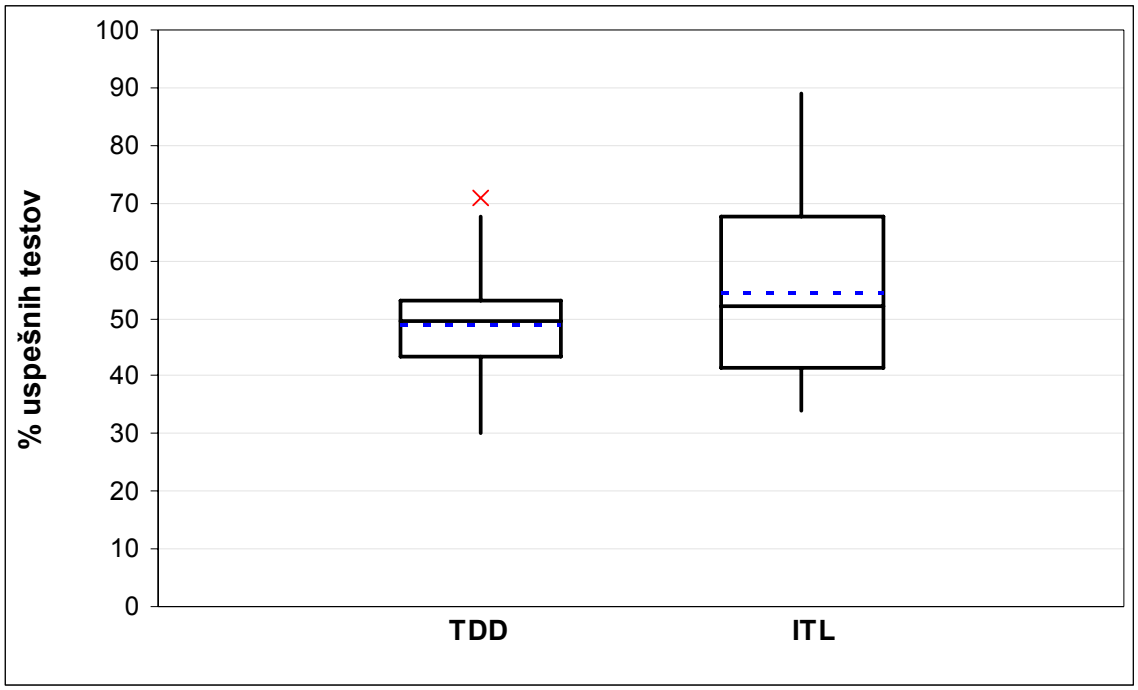
Tabela 4-17 prikazuje povzetek rezultatov meritev odstotka uspešnih testov v kontekstu samostojnih razvijalcev (SR04), Slika 4-17 pa ustrezen graf meritev.

Efekt je v predvideni smeri, vendar **razlika statistično ni značilna** (p=0,28). Zato v kontekstu samostojnih razvijalcev **ne moremo zavrniti ničelne hipoteze TESTOK(H_0)**: $TESTOK_{TDD} = TESTOK_{ITL}$. Velikost efekta po Cohen-u je srednje velika.

Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v frekvenci poganjanja testov med ITL in TDD skupino nismo zaznali.**

*Tabela 4-17: Rezultati meritev za % uspešnih testov – **samostojni razvijalci 04 (SR04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	14	48,71	10,05	49,5	0,28	0,42
ITL	16	54,44	16,28	52,00		



*Slika 4-17: % uspešnih testov – **samostojni razvijalci 04 (SR04)***

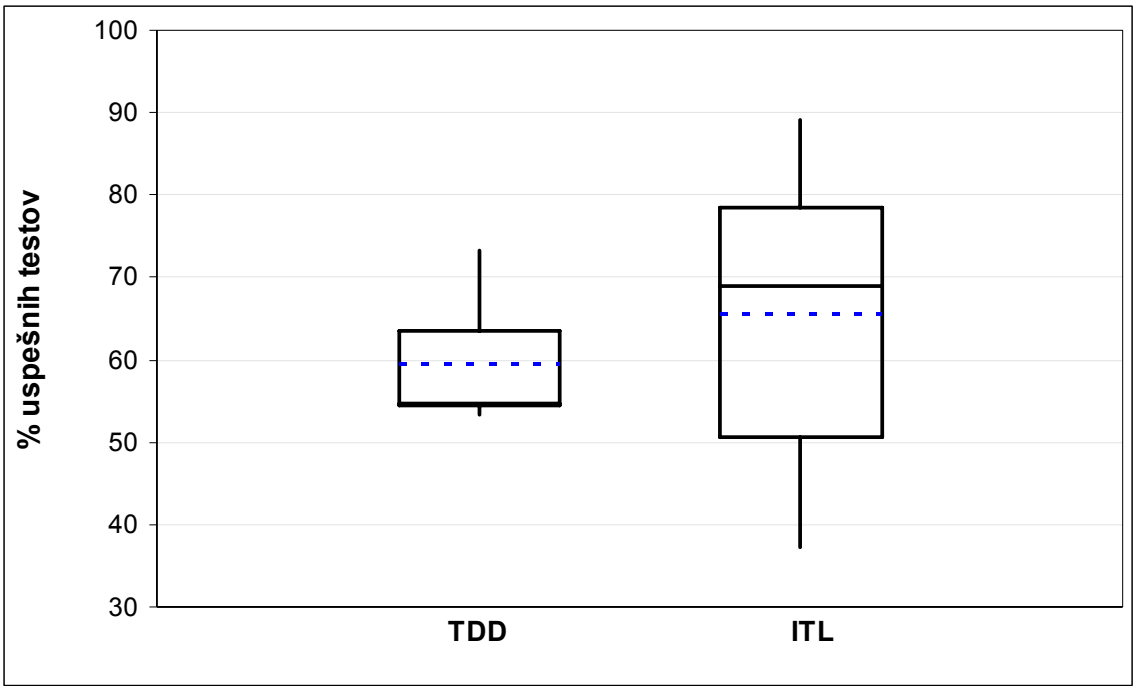
4.4.2.2 Kontekst programiranja v parih

Tabela 4-18 prikazuje povzetek rezultatov meritev uspešnih testov v kontekstu parov leta 2004 (PP04), Tabela 4-19 pa meritve za leto 2003 (PP03). Slika 4-18 in Slika 4-19 prikazujeta grafe meritev za leti 2004 in 2003.

Za **leto 2004** lahko sklenemo, da je efekt v predvideni smeri, vendar **razlika statistično ni značilna** (p=0,31). Velikost efekta po Cohen-u je srednje velika.

*Tabela 4-18: Rezultati meritev za % uspešnih testov – **pari 04***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	7	59,54	7,74	54,66	0,31	0,44
ITL	8	65,63	17,9	69		



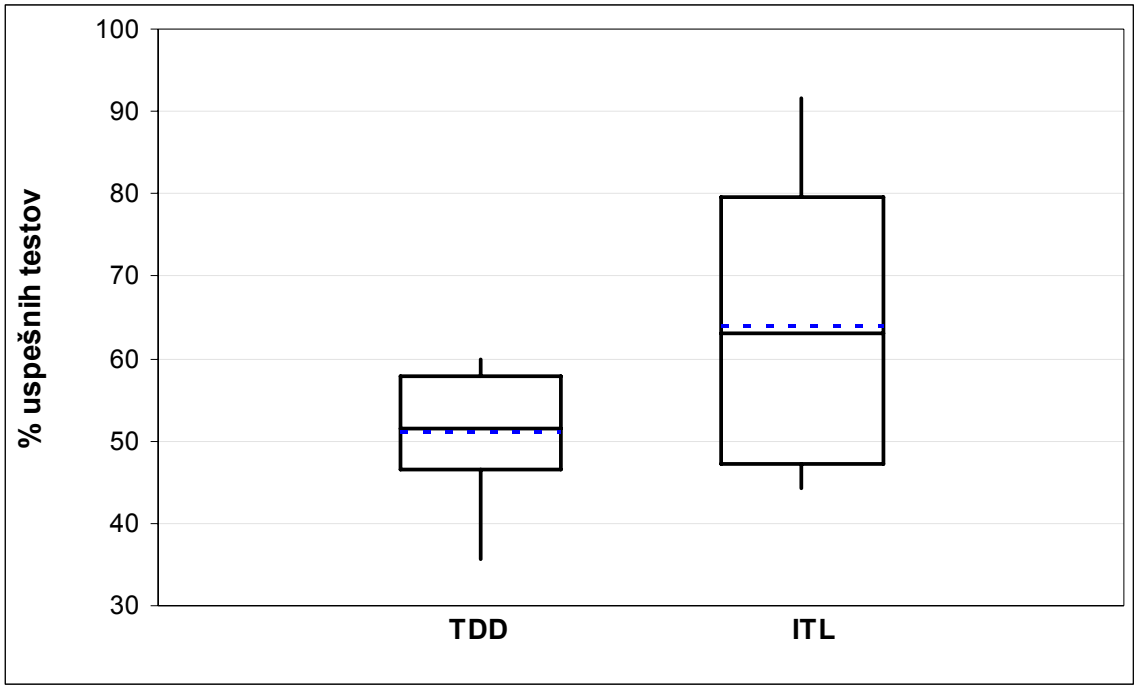
*Slika 4-18: % uspešnih testov – **pari 04 (PP04)***

Za **leto 2003** je efekt v pravi smeri, **razlika je statistično značilna** (p=0,07). Zato lahko v kontekstu parov **zavrnemo ničelno hipotezo TESTOK(H_0)**: $TESTOK_{TDD} = TESTOK_{ITL}$ in sprejmemo alternativno hipotezo **TESTOK(H_a)**: $TESTOK_{TDD} < TESTOK_{ITL}$. Velikost efekta po Cohen-u je velika.

Opazili smo tudi, da so meritve odstotka uspešnih testov za ITL proces razvoja precej bolj razpršene. To po našem mnenju nakazuje, da je odstotek uspešnih testov pri procesu razvoja ITL precej bolj odvisen od posameznega razvijalca in njegove izkušenosti pri testiranju. Proces razvoja TDD je tu bolj predvidljiv.

*Tabela 4-19: Rezultati meritev za % uspešnih testov – **pari 03 (PP03)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p*	Cohen *d*
TDD	10	51,1	8,02	51,41	0,07	0,94
ITL	9	63,95	17,49	63,01		



*Slika 4-19: % uspešnih testov – **pari 03 (PP03)***

Čeprav v kontekstu samostojnih razvijalcev statistično značilnih razlik nismo zaznali, smo vseeno mnenja, da so razlike v odstotku uspešnih testov med procesoma razvoja TDD in ITL v obeh kontekstih. Ker je v kontekstu SR04 in PP04 zaradi smrtnost manj meritev v ITL skupini, se zato malo zmanjša tudi statistična moč testa, kar posledično pomeni, da je večja tudi verjetnost napake II reda (torej da efekt je, a ga nismo zaznali – *Type II Error*). Domnevamo, da je velikost efekta pri odstotku uspešnih testov precej odvisna tudi od konkretne eksperimentalne naloge in zato ocenjujemo, da je velikost efekta v velikostnem razredu srednje velik do velik.

Skupna ugotovitev glede frekvence poganjanja testov v obeh kontekstih uporabe:

Med TDD in ITL skupino smo zaznali statistično značilne razlike (p=0,07) v odstotku uspešnih testov. TDD skupina je imela v povprečju za 20% manj uspešnih testov kot ITL skupina.

Ocenjujemo, da je velikost efekta srednje velika do velika.

4.4.3 Faze dela: kodiranje funkcionalnosti, testiranje, preoblikovanje

Zanimalo nas je, koliko časa razvijalci prebijejo v posamezni fazi dela (programiranje funkcionalnosti – to fazo imenujemo tudi kar kodiranje, testiranje in preoblikovanje). V ITL skupini treh meritev nismo upoštevali pri analizi, saj smo ugotovili da so ti trije razvijalci zelo površno označevali faze dela – eden jih celo ni[30].

Ker gre tu za povprečne odstotke prebitega časa v fazi dela za posamezno skupino, ne moremo pričakovati, da bo seštevek vseh treh faz dal sto odstotkov. Številke je potrebno gledati le primerjalno med obema skupinama za vsako kategorijo posebej.

Edino raziskavo na tem področju sta do sedaj opravila Rostaher in Heričko [Rostaher+ 2002, Heričko+ 2003]. Ugotovila sta, da v povprečju profesionalni razvijalec z uporabo procesa razvoja TDD prebije 51% časa v fazi testiranja, 12% v fazi preoblikovanja in 37% v fazi kodiranja funkcionalnosti. Ker podobnih ocen za proces razvoja ITL nismo mogli pridobiti, smo se tu orientirali na iskanje razlik med obema procesoma razvoja (hipoteza ČASKTP(H_a): $ČASKTP_{TDD} \neq ČASKTP_{ITL}$). Zaradi **dvostranskega testa** statistične značilnosti imamo v tem primeru tudi **manjšo statistično moč testa**.

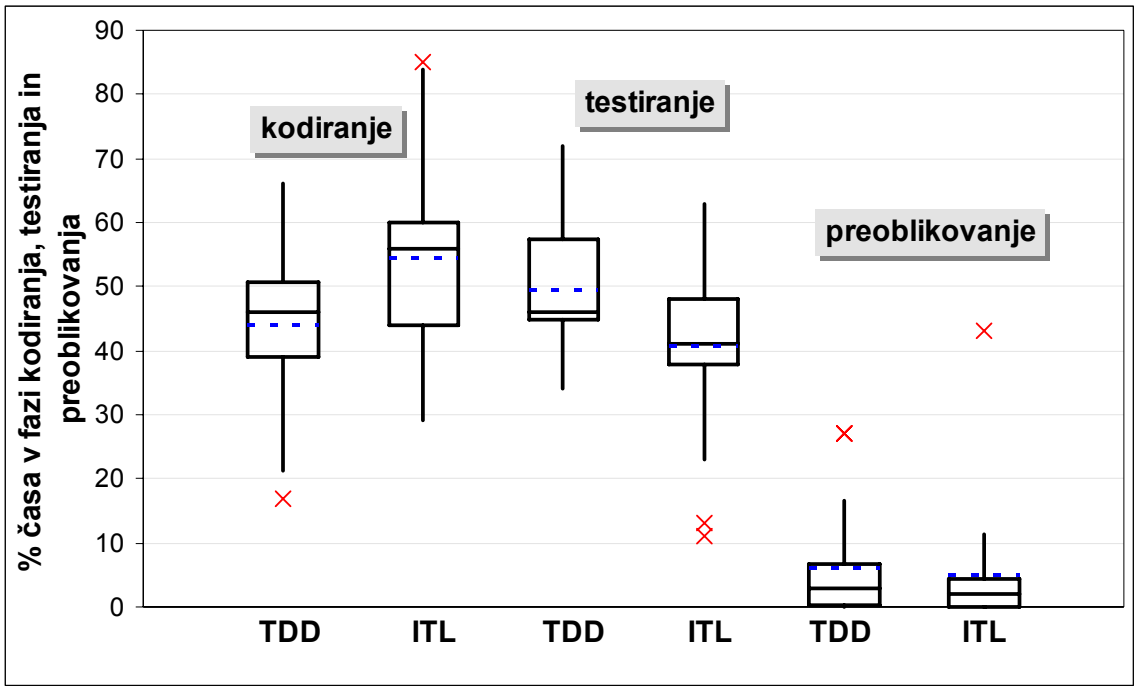
[30] Z razlogom: "…zaradi živčnosti in zatopljenosti v programiranje sem popolnoma pozabil na označevanje faz dela". Označil je vhod v fazo kodiranja, čeprav se iz rekonstrukcije dela vidi, da je delal pravilno po procesu razvoja ITL, le faz ni označeval.

4.4.3.1 Kontekst samostojnih razvijalcev

Tabela 4-20 prikazuje povzetek rezultatov meritev odstotka porabljenega časa v posamezni fazi dela (kodiranje, testiranje, preoblikovanje) v kontekstu samostojnih razvijalcev (SR04), Slika 4-20 pa ustrezen graf meritev.

*Tabela 4-20: Rezultati meritev za % časa v posamezni fazi (kodiranje funkcionalnosti, testiranje, preoblikovanje) – **samostojni razvijalci 04 (SR04)***

Metrika	Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p* *(dvosmerni)*	Cohen *d*
% časa v kodiranju	TDD	14	44	12,14	46	0,06	0,72
	ITL	15	54	16,22	56		
% časa v testiranju	TDD	14	50	10,25	46	0,102	0,69
	ITL	15	41	15,03	41		
% časa v preoblikovanju	TDD	14	6	9,16	3	0,38	0,13
	ITL	15	5	10,76	2		



*Slika 4-20: % časa v posamezni fazi – **samostojni razvijalci 04 (SR04)***

Iz tabele vidimo, da je **razlika med TDD in ITL v odstotku časa v kodiranju statistično značilna** (p=0,06), velikost efekta je velika. Zato lahko tu zavrnemo ničelno hipotezo in sprejmemo alternativno hipotezo ČASKTP(H_a): $ČASKTP_{TDD} \neq ČASKTP_{ITL}$. Razvijalci v **ITL skupini so prebili 22% več časa v fazi kodiranja kot razvijalci v TDD skupini**.

Razlika v odstotku časa v testiranju ni statistično značilna (p=0,102), vendar le za las, velikost efekta je srednje velika do velika. Prav tako razlika v odstotku časa v preoblikovanju ni statistično značilna (p=0,38).

Sklenemo lahko: v kontekstu samostojnih razvijalcev SR04 so razvijalci ITL skupine 22% več časa prebili v fazi kodiranja od razvijalcev TDD skupine.

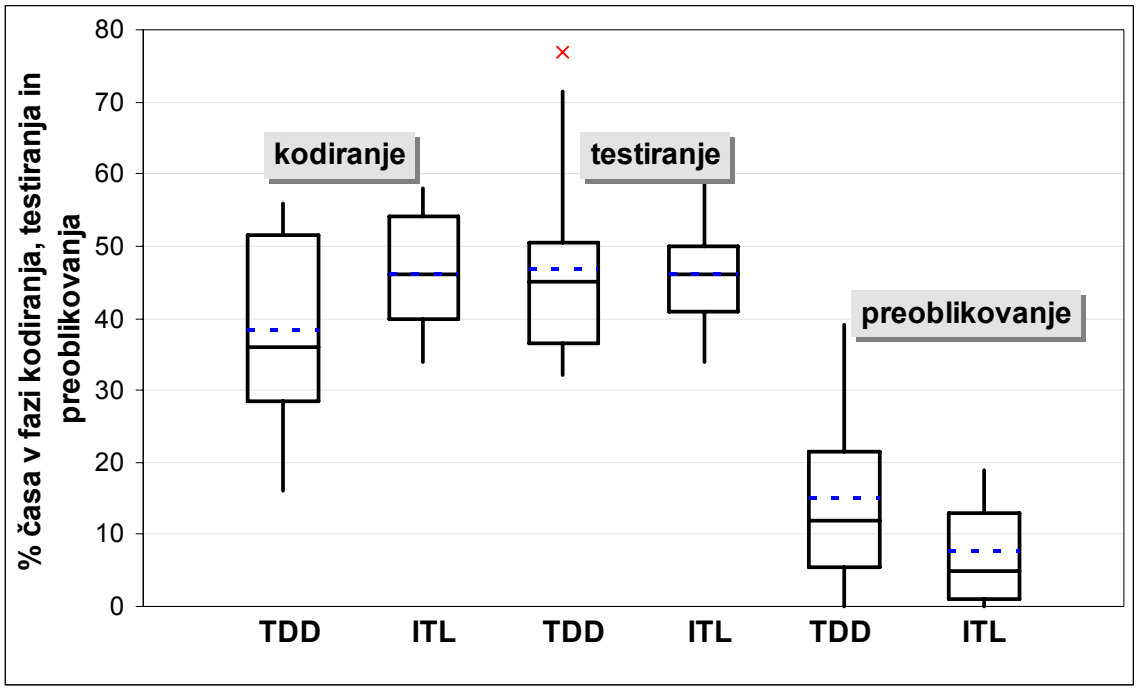
4.4.3.2 Kontekst programiranja v parih

Tabela 4-21 prikazuje povzetek rezultatov meritev uspešnih testov v kontekstu parov leta 2004 (PP04), Tabela 4-22 pa meritve za leto 2003 (PP03). Slika 4-21 in Slika 4-22 prikazujeta grafe meritev za leti 2004 in 2003.

Tabela 4-21: Rezultati meritev za % časa v posamezni fazi (kodiranje funkcionalnosti, testiranje, preoblikovanje) – ***pari 04 (PP04)***

Metrika	Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p* *(dvosmerni)*	Cohen *d*
% časa v kodiranju	TDD	7	38	15,17	36	0,3	0,63
	ITL	9	46	8,74	46		
% časa v testiranju	TDD	7	47	15,29	45	0,76	<0,1
	ITL	9	46	8,22	46		
% časa v preoblikovanju	TDD	7	15	13,44	12	0,26	0,66
	ITL	9	8	7,51	5		

Za **leto 2004** vidimo, da razlike med TDD in ITL v odstotku časa v kodiranju, testiranju in preoblikovanju statistično niso značilne (p=0,3 za kodiranje, p=0,76 za testiranje in p=0,26 za preoblikovanje). Velikost efekta je pri odstotku časa v kodiranju in preoblikovanju srednje velika, pri odstotku časa v fazi testiranja pa zanemarljivo majhna.



*Slika 4-21: % časa v posamezni fazi –**pari 04 (PP04)***

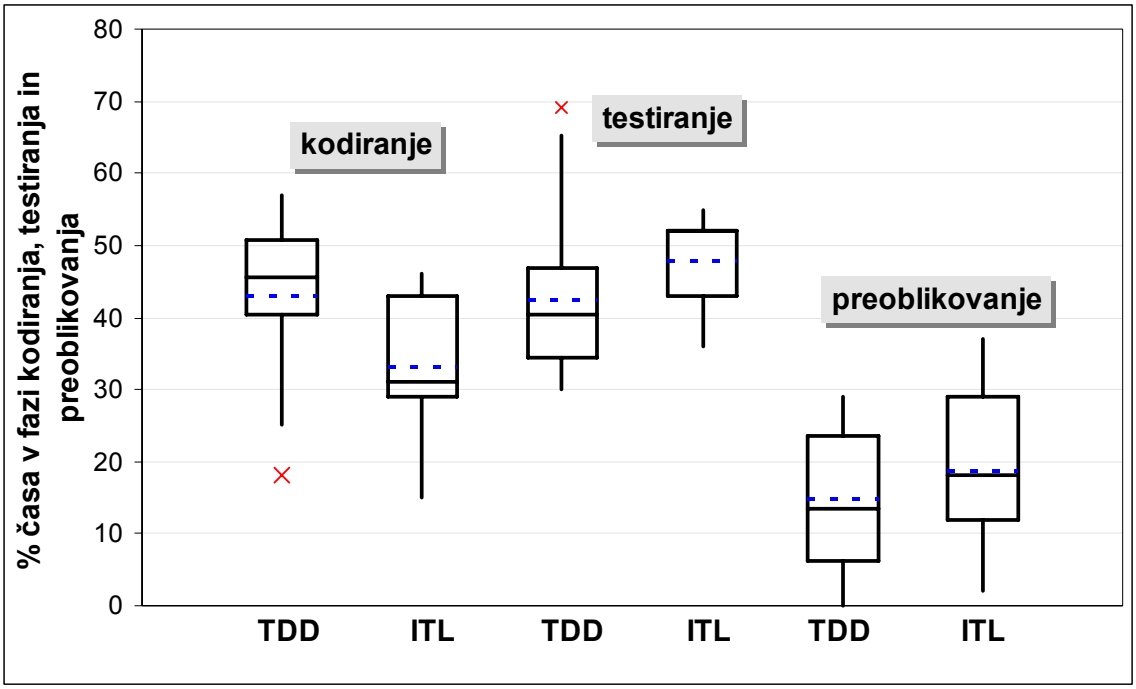
Za **leto 2003** so razlike med TDD in ITL v odstotku časa v fazi kodiranja **statistično značilne** (p=0,095), velikost efekta je velika. Zato lahko tu zavrnemo ničelno hipotezo in sprejmemo alternativno hipotezo za odstotek časa v fazi kodiranja ČASKTP(H_a): $ČASKTP_{TDD} \neq ČASKTP_{ITL}$. Razvijalci v **TDD skupini so prebili 30% več časa v fazi kodiranja kot razvijalci v ITL** skupini.

*Tabela 4-22: Rezultati meritev za % časa v posamezni fazi (kodiranje funkcionalnosti, testiranje, preoblikovanje) – **pari 03 (PP03)***

Metrika	Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p* *(dvosmerni)*	Cohen *d*
% časa v kodiranju	TDD	10	43	12,48	45,5	0,095	0,86
	ITL	9	33	10,21	31		
% časa v testiranju	TDD	10	43	11,83	40,5	0,18	0,58
	ITL	9	48	6,40	52		
% časa v preoblikovanju	TDD	10	15	10,37	13,5	0,45	0,34
	ITL	9	19	12,00	18		

Razliki v odstotku časa v fazi testiranja in preoblikovanja statistično nista značilni (*p*=0,18 za testiranje in p=0,45 za preoblikovanje). Velikost efekta je pri odstotku časa v fazi testiranja srednje velika, pri odstotku časa v fazi preoblikovanja pa majhna do srednje velika.

Sklenemo lahko: v kontekstu parov PP04 in PP03 so razvijalci TDD skupine 30% več časa prebili v fazi kodiranja od razvijalcev ITL skupine.



Slika 4-22: % časa v posamezni fazi – ***pari 03 (PP03)***

Rezultati za fazo kodiranja v obeh kontekstih so si nasprotujoči.

Zato menimo, da obstaja možnost, da je temu tako zaradi tega, ker razvijalci na splošno niso dovolj natančno in vestno definirali faze dela in da je to vplivalo v veliki meri na izid analize. Sami smo storili vse, da bi bila ta natančnost kar največja. Zato menimo, da bi morebiti lahko izboljšali natančnost označevanja faz dela izključno z uporabo avtomatskega razbiranja faz s kompleksno analizo dnevnikov procesa razvoja in shranjenih različic kode.

Dokončnih zaključkov tu tako ne moremo postaviti. **Predvidevamo, da na odstotek časa razvoja v posamezni fazi bolj vpliva kakšna druga spremenljivka kot le**

prisotnost oziroma odsotnost vnaprejšnjega testiranja (torej uporaba procesa razvoja TDD in ITL).

Skupna ugotovitev glede odstotka prebitega časa v posamezni fazi kodiranja, testiranja in preoblikovanja v obeh kontekstih uporabe:

Zaradi nasprotujočih si rezultatov eksperimentov ne moremo ne zavrniti ne potrditi hipoteze o razlikah med TDD in ITL skupino v odstotku prebitega časa v fazi kodiranja, testiranja in preoblikovanja. Zato sklepamo, da na odstotek prebitega časa v posamezni fazi dela v večji meri kot izbrani proces (ITL ali TDD) vpliva kakšna druga spremenljivka.

4.4.4 Razmerje med testi in kodo

Več avtorjev navaja oceno, da je pri metodologiji XP in pri uporabi procesa TDD razmerje med testi in kodo (šteto preko vrstic izvorne kode - LOC) vsaj 1:1 oz. celo več v korist testne kode [Beck 2003, eWorkshop3 2002]. Ta opažanja so nastala na podlagi več realnih projektov narejenih z metodologijo XP, ostaja pa vprašanje, ali je razmerje testi-koda tako visoko tudi pri procesu razvoja ITL. Ali drugače povedano: zanima nas, ali je komponenta vnaprejšnjega testiranja pri TDD tista, ki usmerja razvijalca k večji količini testne kode ali pa so za to bolj odločilne iteracije in tesno prepletanje testiranja s kodiranjem.

Tu bi lahko za metriko izbrali tudi nezakomentirane vrstice izvorne kode (NCLOC – *non commented lines of code*). Odločili smo se za metriko LOC, saj je bolj običajna, navajajo jo tudi drugi avtorji, poleg tega pa smo s striktnim avtomatskim preverjanjem uporabe standarda kodiranja poskrbeli, da velikih razlik med LOC in NCLOC ni (uporaba komponente CheckStyle, glej poglavje 3.10.2 Izvedba eksperimenta).

4.4.4.1 Kontekst samostojnih razvijalcev

Pri eksperimentu s samostojnimi razvijalcih 2004 (SR04) smo upoštevali le prvo zgodbo (glej razlago v poglavju 4.3.1 Število testov).

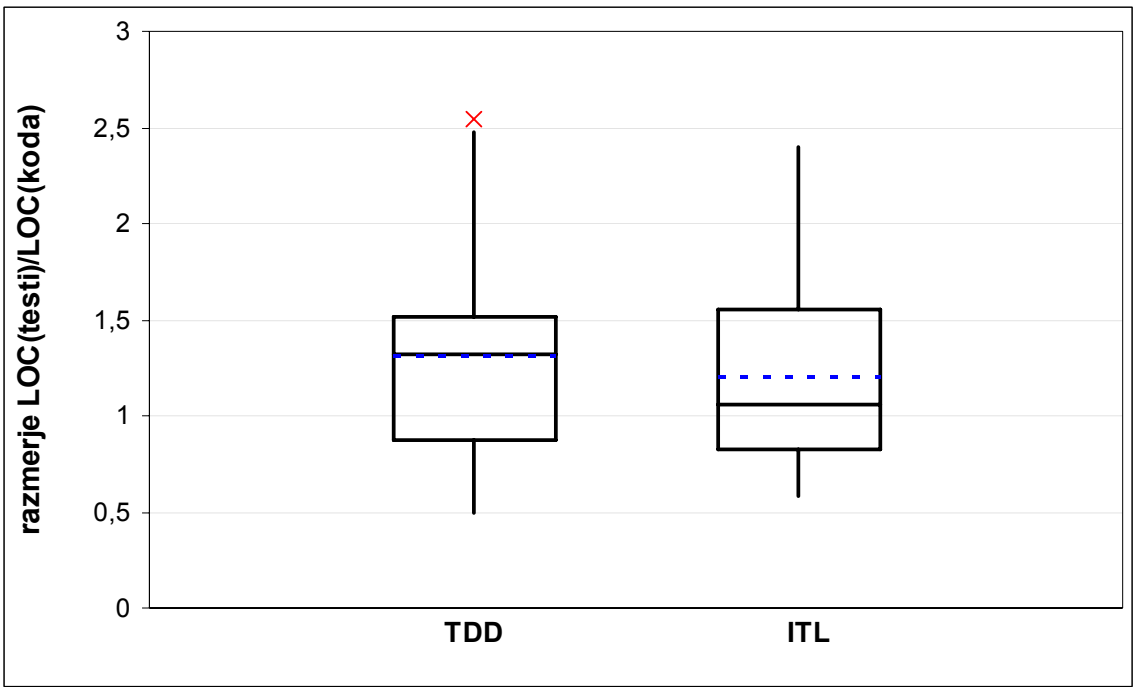
Tabela 4-23 prikazuje povzetek rezultatov meritev razmerja med testno kodo in funkcionalno kodo (merjeno v vrsticah izvorne kode - LOC) v kontekstu samostojnih razvijalcev (SR04), Slika 4-23 pa ustrezen graf meritev.

Razlike med TDD in ITL v razmerju med testi in kodo statistično niso značilne (p=0,69). Velikost efekta je majhna. Zato ne moremo zavrniti ničelne hipoteze RAZMTK(H_0): $RAZMTK_{TDD} = RAZMTK_{ITL}$. Zaradi manjše statistične moči dvosmernega testa, ki smo ga uporabili tu, tudi ne moremo z veliko gotovostjo sprejeti hipoteze RAZMTK(H_0) in trditi, da velikih razlik ni.

Sklenemo lahko: v kontekstu samostojnih razvijalcev **razlik v razmerju med testi in kodo med ITL in TDD skupino nismo zaznali.**

Tabela 4-23: Rezultati meritev razmerja LOC(testi) / LOC(koda) – ***samostojni razvijalci 04 (SR04)***

Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p* *(dvosmerni)***	**Cohen *d***
TDD	14	1,31	0,56	1,32	0,69	0,19
ITL	18	1,21	0,52	1,06		



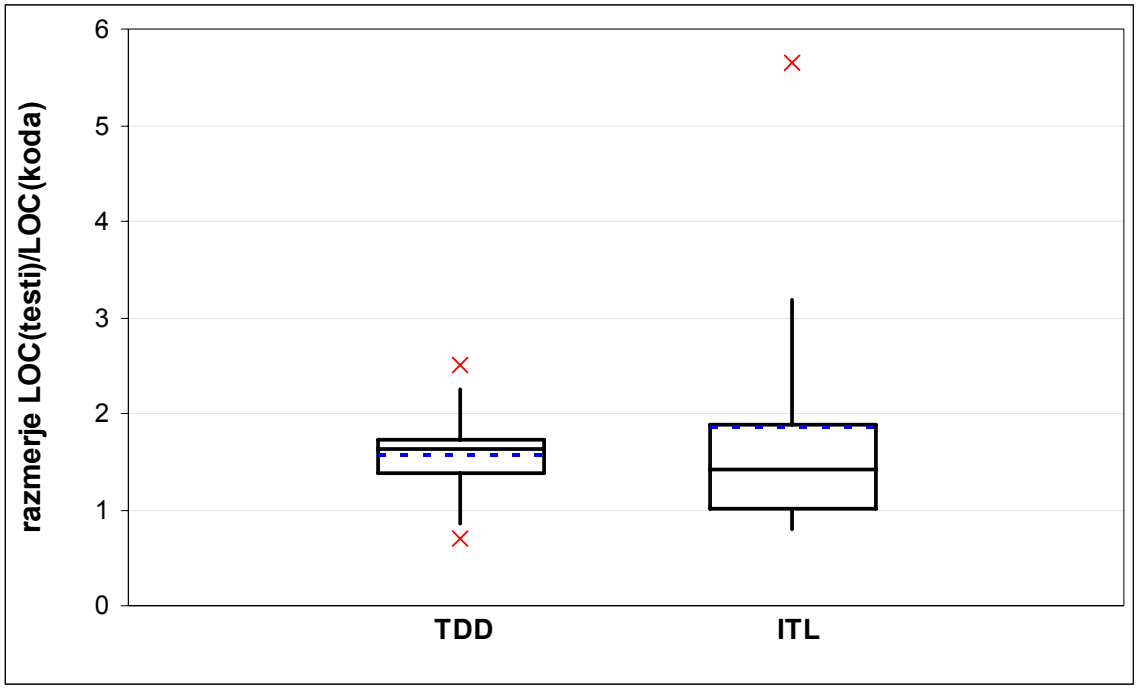
*Slika 4-23: LOC(testi) / LOC(koda) – **samostojni razvijalci 04 (SR04)***

4.4.4.2 Kontekst programiranja v parih

Tabela 4-24 prikazuje povzetek rezultatov meritev uspešnih testov v kontekstu parov leta 2004 (PP04), Tabela 4-25 pa meritve za leto 2003 (PP03). Slika 4-24 in Slika 4-25 prikazujeta grafe meritev za leti 2004 in 2003.

*Tabela 4-24: Rezultati meritev razmerja LOC(testi) / LOC(koda) – **pari 04 (PP04)***

Skupina	Št. meritev	Povp. vr.	St. dev.	Mediana	Mann-Whitney *p* *(dvosmerni)*	Cohen *d*
TDD	7	1,58	0,55	1,64	0,84	0,25
ITL	9	1,87	1,49	1,42		



*Slika 4-24: Razmerje LOC(testi) / LOC(koda) – **pari 04 (PP04)***

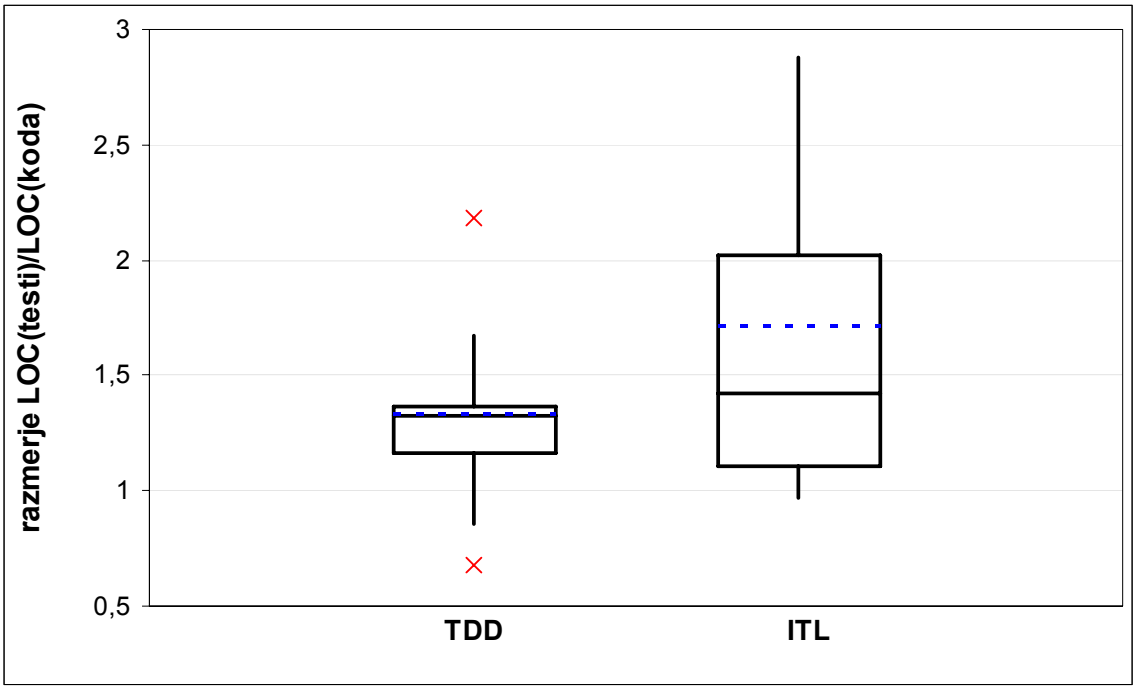
Za **leto 2004** vidimo, da razlike med TDD in ITL v razmerju med testi in kodo **statistično niso značilne** (p=0,84). Velikost efekta je majhna.

Tudi za **leto 2003** razlike med TDD in ITL v razmerju med testi in kodo **statistično niso značilne** (p=0,45). Velikost efekta je srednje velika do velika. Povprečno razmerje je pri

ITL skupini tako visoko zaradi ene same meritve. Tu sta razvijalca pisala izredno veliko število testov (tud v tesni fazi sta prebila več časa kot je povprečje pri ITL). Ob analizi dnevnika razvoja in različic kode smo opazili, da gre za v testiranju manj izkušena razvijalca, ki sta s kvantiteto poskušala nadomestiti manjšo kvaliteto njunih testov, zato sta testirala obsežne scenarije. Meritev je regularna, zato smo jo upoštevali v analizi. Če te meritve ne bi upoštevali, bi bilo povprečje razmerja tudi pri ITL skupini bolj podobno tistemu, ki je v TDD skupini.

*Tabela 4-25: Rezultati meritev razmerja LOC(testi) / LOC(koda) – **pari 03 (PP03)***

Skupina	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p* *(dvosmerni)***	**Cohen *d***
TDD	10	1,33	0,39	1,33	0,45	0,66
ITL	9	1,71	0,71	1,42		



*Slika 4-25: Razmerje LOC(testi) / LOC(koda) – **pari 03 (PP03)***

V kontekstu parov **ne moremo zavrniti ničelne hipoteze** RAZMTK(H_0): $RAZMTK_{TDD}$ = $RAZMTK_{ITL}$. Zaradi manjše statistične moči dvosmernega testa, ki smo ga uporabili

tu, tudi ne moremo z veliko gotovostjo sprejeti hipotezo RAZMTK(H_0) in trditi, da velikih razlik ni.

Sklenemo lahko: v kontekstu parov **razlik v razmerju med testi in kodo med ITL in TDD skupino nismo zaznali.**

Opazimo lahko tudi, da so meritve v ITL skupini bolj razpršene kot meritve TDD skupine. Domnevamo, da do tega efekta pride zaradi tega, ker je zaradi naknadnega testiranja kode v procesu razvoja ITL količina testiranja bolj odvisna od izkušenosti posameznega razvijalca (boljši razvijalci testirajo več) kot je to pri procesu TDD. Drugače povedano, proces razvoja TDD zaradi vnaprejšnjega testiranja bolj usmerja vse razvijalce, tako boljše kot slabše, k določeni količini testne kode kot proces razvoja ITL, kjer je to bolj prepuščeno posamezniku in njegovi usposobljenosti za učinkovito testiranje.

Skupna ugotovitev glede razmerja med testi in kodo v obeh kontekstih uporabe:

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v razmerju med vrsticami izvorne kode za teste in kodo. Opazili smo, da so naše meritve skladne z oceno praktikov (razmerje testi-koda najmanj 1:1, velikokrat tudi več v korist testne kode). Sami smo zaznali v kontekstu samostojnih razvijalcev (omejen čas razvoja) razmerje med testi in kodo okrog 1,25 v korist testov, v kontekstu dela parov (neomejen čas razvoja) pa je bilo razmerje višje, nekje okrog 1,5 v korist testov. To kaže, da oba procesa razvoja usmerjata razvijalca k podrobnemu pisanju testne kode.

Z ozirom na izvedene meritve ocenjujemo, da gre lahko v primeru razmerja med testi in kodo kvečjemu za majhen do srednje velik efekt.

4.4.5 Kvalitativni parametri procesa razvoja

Z našim eksperimentom smo želeli pridobiti ne le kvantitativne parametre glede produktivnosti, kvalitete in testiranja, ampak smo tudi zbrali in analizirali kvalitativna opažanja razvijalcev o procesu razvoja. Kvalitativna komponenta procesa razvoja je zelo pomembna zato, ker če so npr. razvijalci proti kakšnemu procesu dela izrazito nenaklonjeni in ga odklanjajo, bo na dolgi rok uvedba takšnega procesa razvoja v skupino neuspešna. Zanimale so nas predvsem mnenja razvijalcev o koristnosti posameznih komponent razvojnega procesa, zbrali pa smo tudi opažanja glede tehničnih pogojev dela v eksperimentu (razvojno okolje, uporaba testnega ogrodja JUnit, …), katere so namenjene predvsem olajšanju izvedbe morebitne replikacije našega eksperimenta za druge raziskovalce.

Mnenja razvijalcev smo pridobili z **anonimno anketo** (priloga E), saj smo se le tako lahko nadejali odkritih odgovorov in s tem pridobili kar najbolj kritično in realno oceno. Število izpolnjenih anket za vsako posamezno leto prikazuje Tabela 4-26.

Tabela 4-26: Število izpolnjenih anket o procesu razvoja za posamezno serijo eksperimentov (2003 in 2004)

Leto izvedbe eksperimenta	**2003**	**2004**
Št. vseh subjektov, ki so sodelovali pri prvem eksperimentu v nizu (programiranje v parih)	**38**	**34**
TDD skupina	20	16
ITL skupina	18	18
Oddali anketo (sodelovali tudi v eksperimentu s samostojnimi razvijalci)	**34**	**34**
Uporabne ankete	**34**	**31**
TDD skupina	19	14
ITL skupina	15	17

V letu 2004 so sicer vsi študenti, ki so sodelovali pri končnem eksperimentu oddali anketo, vendar trije niso označili, kateri proces dela so uporabljali (dva sta bila iz ITL skupine, eden iz TDD skupine). Ker so bile ankete anonimne, nismo mogli naknadno

ugotoviti, katera od anket spada v katero od skupin in smo zato te tri ankete izločili iz obdelave in analize.

George in Williams [George+ 2004] sta merila mnenje razvijalcev o produktivnosti, učinkovitosti procesa dela in težavnosti osvajanja TDD **pri profesionalnih programerjih**. Trditve sta agregirala v tri področja: **produktivnost, učinkovitost (in kvaliteta) ter težavnost osvojitve procesa razvoja**.

Tudi sami smo uporabili njihovo ogrodje vprašanj na študentih, ki so sodelovali v našem eksperimentu. Vprašanja so reševali vsi študentje, vendar so se nanašala le na **tisti proces razvoja, ki so ga aktivno uporabljali** v eksperimentu – torej TDD skupina je ocenjevala proces razvoja TDD, ITL skupina pa proces razvoja ITL. Tabela 4-27 prikazuje zbrane rezultate analize. V posameznih celicah tabele so zbrani odstotki razvijalcev, ki **se strinjajo s posamezno trditvijo** za vsako leto eksperimenta posebej in agregirano (stolpec "Povp.").

*Tabela 4-27: **% razvijalcev, ki se strinjajo** s trditvijo o produktivnosti, učinkovitosti in težavnosti osvojitve posameznega procesa dela. Desno so rezultati profesionalnih programerjev.*

		ITL			**TDD**			**TDD [George+ 2004]**
		2003	**2004**	**Povp.**	**2003**	**2004**	**Povp.**	**Skupaj**
Produktivnost		49	76	**63**	47	65	**56**	78
1.1	Uporaba TDD/ITL omogoča programerju, da bolje razume zahteve	60	82	**71**	63	71	**67**	88
1.2	Uporaba TDD/ITL zmanjša čas razhroščevanja	53	88	**71**	58	79	**69**	96
1.3	Uporaba TDD/ITL zmanjša čas razvoja programa	33	59	**46**	21	46	**34**	50

Učinkovitost (in kvaliteta)		51	65	**58**	50	53	**52**	80
2.1	S TDD/ITL je koda boljše kvalitete	73	76	**75**	78	93	**86**	92
2.2	Z uporabo TDD/ITL je design programa preprostejši	40	65	**53**	47	36	**42**	79
2.3	S TDD/ITL je delo opazno bolj učinkovito	40	53	**47**	26	29	**28**	71
Težavnost osvojitve		40	50	**45**	69	57	**63**	40
3.1	Osvojiti način TDD/ITL dela je težko	20	23	**22**	63	43	**53**	56
3.2	Slabost TDD/ITL je pomanjkanje posebne faze načrtovanja na začetku	60	76	**68**	74	71	**73**	23

Iz tabele lahko vidimo, da so profesionalni razvijalci bolj prepričani v višjo produktivnost procesa razvoja TDD kot študenti, kar lahko trdimo za vsa tri vprašanja iz te skupine. Podobno velja tudi za proces razvoja ITL, za katerega so številke zelo podobne številkam študentov, ki so uporabljali TDD.

Študenti torej manj zaupajo v večjo produktivnost TDD in ITL procesa razvoja kot profesionalni razvijalci. Predvidevamo, da je temu tako zaradi tega, ker pred tem eksperimentom študentje niso pisali testov za svoje programe in zato še vedno pojmujejo testiranje kot zmanjšanje produktivnosti na račun kodiranja.

Podobno kot pri produktivnosti je tudi pri učinkovitosti in kvaliteti. Študentje v manjši meri kot profesionalni razvijalci verjamejo, da so z uporabo TDD in ITL bolj učinkoviti in da pišejo bolj kvalitetne programe. Razlike pripisujemo predvsem relativni težavnosti osvojitve procesa razvoja v primerjavi z le na videz bolj učinkovitim "kodiraj in popravi" ("*code and fix*") načinom razvoja, ki so ga v večini uporabljali pred eksperimentom.

Pri pogovorih s študenti po eksperimentu smo opazili, da študentje, ki so imeli že kakšne realne izkušnje iz industrije, bolj naklonjeno gledajo na TDD in ITL (in so s tem bližje mnenju profesionalnih razvijalcev).

Študenti so tudi precej bolj prepričani, da je slabost TDD in ITL procesa razvoja pomanjkanje posebne faze načrtovanja na začetku dela (cca. 70% študentov v TDD in ITL skupini v primerjavi z le 23% profesionalnih razvijalcev). Efekt pripisujemo predvsem temu, da je za t.i. pojavljajoč design pri obeh procesih razvoja potrebno bolj intenzivno uporabljati proces preoblikovanja, le-tega pa je moč učinkovito uporabiti le z dovolj velikimi izkušnjami. K efektu tudi pripomore navajenost študentov na klasični slapovni proces razvoja, ki so ga uporabljali do sedaj.

Ocene študentov glede **razlik med procesoma razvoja TDD in ITL** so večinoma bolj ali manj skladne (razlika do cca. 10%), **z dvema izjemama, kjer je razlika veliko bolj izrazita**:

- Težavnosti osvojitve (vprašanje 3.1)
- Učinkovitost dela (vprašanje 2.3)

Študenti dojemajo ITL kot opazno bolj učinkovit proces od TDD, proces razvoja ITL je tudi lažje osvojiti.

V okviru zadnje ankete nas je tudi zanimalo, ali imajo študenti **še namen uporabljati procesa razvoja TDD in ITL, njihove sestavne elemente ter v eksperimentu uporabljana razvojna okolja**. Tabela 4-28 prikazuje zbrane odgovore skupno za leti 2003 in 2004.

*Tabela 4-28: Odstotek razvijalcev, ki imajo **še namen uporabljati** procesa razvoja TDD in ITL, njihove sestavne elemente ter v eksperimentu uporabljana razvojna okolja.*

Tudi v prihodnosti imam še namen uporabljati:	ITL	TDD
Avtomatizirano testiranje	**91**	**76**
Izrazito iterativno reševanje problemov	**75**	**55**
JUnit	**81**	**70**
Eclipse	**84**	**76**
Preoblikovanje	**91**	**82**
TDD	**32**	**42**
ITL	**81**	**59**
Stalno programiranje v paru	**39**	**19**

Najprej lahko opazimo dejstvo, da je ITL skupina v povprečju bolj navdušena nad uporabljenimi elementi procesa dela kot TDD skupina. Študenti so v večini navdušeni nad avtomatskim testiranjem, uporabo ogrodja JUnit, uporabo razvojnega okolja Eclipse in procesom preoblikovanja. Presenetljivi pa so odgovori glede TDD, ITL in stalnega programiranja v paru.

Večina študentov se strinja, da bodo še naprej uporabljali ITL proces razvoja, v nasprotju s TDD procesom razvoja, ki ga bo še naprej uporabljala in preizkušala manj kot polovica študentov! To kaže na to, da je **proces razvoja ITL študentom veliko bolj pri srcu** in se z njim hitreje poistovetijo.

Prav tako študenti v večini ne vidijo neke posebne vrednosti v stalnem programiranju v parih. Odgovori in pogovori s študenti nakazujejo, da je tako predvsem zaradi:

- Težav pri uskladitvi tako velikega števila ur za skupno delo (argument, ki v profesionalnem okolju odpade).
- Ker menijo, da je določene stvari bolj učinkovito delati samostojno, nato pa skupaj naknadno le pregledati kodo (*code review*). Kot izrazit primer so navedli na primer pisanje JavaDoc komentarjev.

Velika večina razvijalcev (95%) tudi rada med razvojem zelo nazorno vidi, da so se testi izvedli uspešno (t.i. *green bar* v ogrodju JUnit oz. v okolju Eclipse).

Presenetljiva je še ena ugotovitev. Čeprav imajo študenti v povprečju slabše mnenje o procesu razvoja ITL in TDD od profesionalnih programerjev, **nimajo odklonilnega odnosa do uporabe TDD in ITL v njihovem bodočem profesionalnem okolju, kjer bi ga morali uporabljati vsak dan**. Na vprašanje: "*Ali bi sprejeli delo v podjetju, kjer uporabljajo izključno TDD oz. ITL proces dela?*" so si bili študenti precej enotni: **večina**, to je 88 % študentov iz ITL skupine in 85% študentov iz TDD skupine, **bi se zaposlila** pri takšnem delodajalcu.

Na koncu omenimo tudi na splošno pozitiven vtis študentov navkljub zelo veliki količini vloženega dela. V letu 2004 in 2005 nam je v pogovorih nekaj študentov iz prvega eksperimenta leta 2003 potrdilo, da so tehnike programiranja, ki so se jih naučili v sodelovanju pri tem eksperimentu, že koristno uporabili v realnih projektih.

4.5 Povzetek rezultatov analize

Ker je celotno poglavje analize rezultatov zaradi množice meritev zelo obsežno, na koncu zelo na kratko povzemimo glavne ugotovitve poglavja.

O kvaliteti in produktivnosti

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v produktivnosti in kvaliteti (tako zunanji kot notranji). Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da **velikih razlik med procesom TDD in ITL glede produktivnosti in kvalitete ni**. Ocenjujemo, da gre v primeru produktivnosti in zunanje kvalitete kvečjemu za majhen efekt, v primeru notranje kvalitete pa kvečjemu za srednje velik efekt. Pri procesu razvoja ITL smo opazili večjo možnost za nastanek izrazito slabe notranje kvalitete kot pri procesu razvoja TDD.

O številu in kvaliteti testov

V procesu razvoja TDD **v številu testov ni statistično značilnih razlik** v primerjavi s procesom razvoja ITL, **precej večja pa je v TDD njihova kvaliteta** (za 7%, velikost efekta je velika). Proces razvoja **TDD torej usmerja razvijalce k pisanju bolj kvalitetnih testov** od procesa ITL ob enakem številu testov. Ocenjujemo, da gre lahko v primeru števila testov kvečjemu za srednje velik efekt.

O procesu dela

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v frekvenci poganjanja testov. Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da velikih razlik med procesom TDD in ITL glede frekvence poganjanja testov ni. Ocenjujemo, da gre lahko v primeru frekvence poganjanja testov kvečjemu za srednje velik efekt.

Zaznali smo statistično značilne razlike v odstotku uspešnih testov. ITL skupina ima za 20% več uspešnih testov. Velikost efekta je srednje velika do velika.

Zaradi nasprotujočih si rezultatov eksperimentov ne moremo ne zavrniti ne potrditi hipoteze o razlikah med TDD in ITL skupino v odstotku prebitega časa v fazi kodiranja, testiranja in preoblikovanja. Zato sklepamo, da na odstotek prebitega časa v posamezni

fazi dela v večji meri kot izbrani proces (ITL ali TDD) vpliva kakšna druga spremenljivka.

Statistično značilnih razlik med procesom TDD in ITL v razmerju med vrsticami izvorne kode (LOC) za teste in kodo nismo zaznali. Opazili pa smo, da so vse naše meritve skladne z oceno praktikov (razmerje testi-koda najmanj 1:1, velikokrat tudi več [Beck 2003, eWorkshop3 2002]. Sami smo zaznali v kontekstu samostojnih razvijalcev (omejen čas razvoja) razmerje med testi in kodo okrog 1,25 v korist testov, v kontekstu dela parov (neomejen čas razvoja) pa je bilo razmerje višje, nekje okrog 1,5 v korist testov. To kaže, da oba procesa razvoja usmerjata razvijalca v podrobno pisanje testne kode. Glede na meritve ocenjujemo, da gre lahko v primeru razlike med TDD in ITL v razmerju med testi in kodo kvečjemu za majhen do srednje velik efekt.

Kvalitativna opažanja

Razlike med dojemanjem TDD pri študentih in profesionalnih programerjih so opazne – v povprečju so študenti manj navdušeni nad TDD kot profesionalci glede produktivnosti, učinkovitosti in kvalitete. V primerjavi s profesionalnimi programerji študentje dojemajo TDD kot precej težje osvojljiv proces razvoja.

Proces razvoja ITL je za študente opazno bolj učinkovit v primerjavi s procesom razvoja TDD. ITL je tudi lažje osvojiti in ga imajo študentje v večini še namen uporabljati v prihodnosti. Zelo so navdušeni tudi nad avtomatskim testiranjem, uporabo ogrodja JUnit, uporabo razvojnega okolja Eclipse in procesom preoblikovanja.

Odklonilnega odnosa do TDD in ITL nimajo, saj bi se velika večina študentov zaposlila v podjetju, kjer uporabljajo izključno procesa razvoja TDD ali ITL. Stalno programiranje v parih jim ne ustreza.

5 SKLEP

Disertacija obravnava področje empiričnega preverjanja vrednosti kvantitativnih in kvalitativnih parametrov testno vodenega razvoja (TDD).

Testno voden razvoj je osrednja praksa metodologije XP in ji pripisujejo največji delež pri bliskovitem uspehu metodologije XP med privrženci agilnih metodologij. Dokazi o učinkih uporabe metodologije XP in testno vodenega razvoja pa so bili do sedaj precej nepoznani. Obstaja sicer kar nekaj študij primerov, zelo malo pa je bilo na tem področju znanstveno dokazanih učinkov uporabe principov in praks metodologije XP.

Cilj naše raziskave je bil, da ugotovimo, če in v kakšni meri vnaprejšnje pisanje testov vpliva na parametre agilnega razvojnega procesa in v njem nastalega izdelka (programa). Predvsem so nas zanimali trije parametri:

- **produktivnost,**
- **kvaliteta** (zunanja in notranja),
- **testiranje** (količina in kvaliteta testov).

Ocenili smo tudi velikost nekaterih drugih parametrov, ki okarakterizirajo posamezen proces dela: koliko časa razvijalci prebijejo v posamezni fazi procesa razvoja, kako pogosto poganjajo teste, kolikšen je delež uspešnih testov in kakšno je količinsko razmerje med testi in kodo.

Zgoraj zbrane kvalitativne parametre procesa razvoja smo ovrednotili tudi skozi **kvalitativna opažanja** razvijalcev, ki ga uporabljajo. Med temi kvalitativnimi parametri nas je zanimal predvsem pogled razvijalcev na produktivnost, učinkovitost in kvaliteto ter težavnost osvojitve posameznega procesa razvoja.

Izvedli smo dva niza eksperimentov v letih 2003 (februar – junij 2003) in 2004 (februar – junij 2004), subjekti so bili študenti osmega semestra univerzitetnega študija računalništva in informatike na Fakulteti za računalništvo in informatiko Univerze v

Ljubljani. Razdelili smo jih v dve enakovredni skupini, edina neodvisna spremenljivka v vsakem eksperimentu je bila pisanje testov pred ali po kodi. Študenti so tako poslušali enaka predavanja, delali z enakimi orodji, imeli enake pogoje dela, … Vsak niz eksperimentov smo izvedli v dveh kontekstih: kontekstu dela v parih, ki imitira tipično uporabo procesa razvoja TDD znotraj metodologije XP in kontekstu dela samostojnih razvijalcev, ki imitira uporabo testno vodenega razvoja znotraj kakšne druge agilne ali tradicionalne metodologije.

Med TDD in ITL skupino nismo zaznali statistično značilnih razlik v produktivnosti in kvaliteti (tako zunanji kot notranji). Zaradi dovolj velike statistične moči naših testov lahko tudi s precejšnjo gotovostjo trdimo, da **velikih razlik med procesom TDD in ITL glede produktivnosti in kvalitete ni**. Pri procesu razvoja ITL smo opazili večjo možnost za nastanek izrazito slabe notranje kvalitete kot pri procesu razvoja TDD.

V procesu razvoja TDD **v številu testov ni statistično značilnih razlik** v primerjavi s procesom razvoja ITL, **precej večja pa je v TDD njihova kvaliteta** (za 7%, velikost efekta je velika). **Proces razvoja TDD torej usmerja razvijalce k pisanju bolj kvalitetnih testov** od procesa ITL, ob približno enakem številu testov.

ITL ima v povprečju za 20% več uspešnih testov od procesa razvoja TDD, kar med drugim tudi potrjuje skladnost naših razvijalcev v TDD skupini s procesom razvoja TDD. V razmerju med testi in kodo ter frekvenci poganjanja testov razlik nismo zaznali. Opazili pa smo, da so naše meritve razmerja med testi in kodo skladne z oceno praktikov (razmerje testi-koda od 1,25 do 1,5 v korist testov). Zaradi nasprotujočih si rezultatov ne moremo ne zavrniti ne potrditi hipoteze o razlikah med TDD in ITL skupino v odstotku prebitega časa v fazi kodiranja, testiranja in preoblikovanja. Zato sklepamo, da na odstotek prebitega časa v posamezni fazi dela v večji meri kot izbrani proces (ITL ali TDD) vpliva kakšna druga spremenljivka.

Velikost efekta je zelo velikega pomena pri načrtovanju novih eksperimentov na področju testno vodenega razvoja ali morebitne replikacije naših eksperimentov, s katerimi bi dobili bolj zanesljive ocene o velikosti posameznih parametrov. V naših eksperimentih razen pri kvaliteti testov nismo našli velikih efektov med primerjanjem kvalitete in produktivnosti procesov razvoja TDD in ITL, zaradi dovolj velike statistične moči testa pa lahko s precejšnjo gotovostjo trdimo, da velikih razlik tam, kjer jih nismo

zaznali, ni. Naše ocene o velikosti efekta pri produktivnosti in kvaliteti bodo raziskovalcem omogočile pravilno načrtovanje novih eksperimentov z dovolj velikim vzorcem, ki jim bo omogočal tudi dovolj veliko statistično moč testov za morebitno sprejetje ničelnih hipotez (torej da efekta med raziskovalnima skupinama ni).

Razlike med dojemanjem TDD pri študentih in profesionalnih programerjih so opazne – v povprečju so bili študenti manj navdušeni nad TDD kot profesionalci glede produktivnosti, učinkovitosti in kvalitete. V primerjavi s profesionalnimi programerji študentje dojemajo TDD kot precej težje osvojljiv proces razvoja.

Proces razvoja ITL je za študente opazno bolj učinkovit v primerjavi s procesom razvoja TDD. ITL se je tudi lažje osvojiti, z njim se študentje hitreje poistovetijo in ga imajo v večini še namen uporabljati v prihodnosti. Zelo so navdušeni tudi nad avtomatskim testiranjem, uporabo ogrodja JUnit, uporabo razvojnega okolja Eclipse in procesom preoblikovanja.

Odklonilnega odnosa do TDD in ITL nimajo, saj bi se velika večina študentov zaposlila v podjetju, kjer uporabljajo izključno procesa razvoja TDD ali ITL. Stalno programiranje v parih jim ne ustreza.

Priporočila glede **uvajanja procesa razvoja TDD in ITL v profesionalno razvojno skupino** sledijo iz rezultatov analize kvalitativnih in kvantitativnih dejavnikov:

- ITL proces razvoja je za povprečnega razvijalca bolj sprejemljiv kot TDD, zato priporočamo v razvojno skupino najprej uvedbo procesa razvoja ITL. S tem se bo skupina uvedla v vse komponente, ki jih uporablja tudi TDD (testiranje enot, preoblikovanje, standardi kodiranja, …), zato bo morebitni kasnejši prehod na TDD veliko lažji.
- Proces razvoja TDD ima prednosti pred ITL predvsem pri boljši kvaliteti testov, zato je potrebno pri uporabi ITL več časa posvetiti učenju učinkovitega testiranja.
- Razvijalci v procesu razvoja ITL pod časovnim pritiskom in ob vnaprej znanih sprejemnih testih prej opustijo pisanje svojih testov enot kot razvijalci v procesu razvoja TDD (ugotovitev sledi iz izkušenj izvedbe eksperimenta s samostojnimi razvijalci 2003 (SR03)). Zato je smiselno pri procesu razvoja ITL vpeljati tudi formalne mehanizme za preverjanje, ali razvijalci pišejo svoje teste enot ali ne.

- TDD bolje sprejemajo izkušenejši razvijalci, zato se nam uvajanje TDD zdi smiselno predvsem pri:
    - izkušeni razvojni skupini,
    - ali pa razvojni skupini, ki je procesu razvoja TDD in ostalim praksam agilnih metodologij že v začetku bolj naklonjena.

Pri **uvajanju TDD v univerzitetno okolje** imamo manj pomislekov kot pri uvajanju TDD v okolje profesionalnih razvijalcev. Po naših izkušnjah in pogovorih s študenti so TDD bolje sprejeli boljši študentje, še posebej takšni, ki so že imeli tudi izkušnje s projekti izven študijskega okolja in so bili zato bolj naklonjeni temeljitemu testiranju, ki ga TDD še posebej izpostavlja. Mislimo, da je proces razvoja TDD navkljub težavnosti osvojitve (še posebej pri slabših študentih) vredno vpeljati v redni študijski proces. Ne nazadnje študentom s tem le širimo arzenal različnih tehnik, ki jim lahko pridejo prav na njihovi nadaljnji strokovni poti. Vsekakor pa je potrebno proces dela TDD šteti med naprednejše tehnike, zato je težavnost osvojitve večja zaradi tega, ker študenti na ta način prej niso razmišljali in jim je zato precej tuj. Ravno zato prof. Mugridge [Mugridge 2003] navaja hipotezo, da je potrebno TDD vpeljati v prvi letnik študija, v osnovni predmet programiranja. Tako naj bi študentje s TDD imeli manj problemov in bi ga lažje osvojili. Glede na rezultate naših eksperimentov pa se lahko vprašamo, ali ne bi bila za podoben efekt dovolj že vpeljava procesa razvoja ITL, ki povzroča študentom veliko manj težav pri osvojitvi v primerjavi s procesom razvoja TDD.

V disertaciji predstavljena problematika je zelo aktualna in se hitro razvija, pojavljajo se tudi nove empirične študije o procesu razvoja TDD, metodologiji XP in drugih agilnih metodologijah. Med možnostmi za nadaljnje raziskave naj omenimo le nekatere, ki so ozko povezane z našo temo:

- Ponovitev našega eksperimenta na večjem vzorcu (večja moč eksperimenta bi pomenila tudi možnost, da sprejmemo ali zavrnemo naše raziskovalne hipoteze pri srednje velikih ali manjših velikostih efekta (Cohenov *d*).
- Ponovitev našega eksperimenta v drugih kontekstih, predvsem na primer v kontekstu profesionalnih razvijalcev, v realnem okolju ali z bolj kompleksnimi eksperimentalnimi nalogami.

- Razvoj metode za avtomatsko določanje faze dela razvijalca iz različic izvorne kode in dnevnika procesa dela (zaganjanje testov in njihov uspeh) – s tem bi odpadli problemi ročnega vodenja in s njim povezane napake meritev.
- Meta študija do sedaj objavljenih empiričnih eksperimentov.
- V naši raziskavi smo se osredotočili na direktne efekte vnaprejšnjega testiranja na posamezne parametre razvojnega procesa. Postavlja pa se vprašanje, ali ni možno nekaterih efektov razložiti le indirektno, preko drugih spremenljivk. Smiselno bi bilo torej razširiti raziskavo tudi na indirektne efekte, podobno kot so to v svojem delu že nakazali Erdogmus in sodelavci [Erdogmus+ 2005].
- Hibridni razvoj: glede na to, da imata tako TDD kot ITL svoje prednosti in pomanjkljivosti, bi bil lahko hibriden pristop, v katerem bi pisali teste tako pred kot po kodi, tudi privlačna možnost.

SLOVARČEK IZRAZOV

Spodaj smo podali kratek slovarček s prevodi v slovenščio. Posebej smo izpostavili nekatere za našo temo specifične izraze, ki zaradi mladosti obravnavane teme še nimajo uveljavljenih prevodov v slovenščino.

Acceptance test	Sprejemni test
Black box test	Test tipa črne škatle
Boxplot graph	Grafični prikaz kvantilov
ITL – Iterative Test-Last Development	Iterativni razvoj z naknadnim testiranjem
LOC, lines of code	Število vrstic kode
NCLOC, non-commented lines of code	Št. nezakomentiranih vrstic izvorne kode
Refactoring	Preoblikovanje kode
Story	Zgodba; neformalni opis funkcionalnih zahtev, ki naj bi jih program izpolnjeval.
TDD – Test-Driven Development	Testno voden razvoj
Unit test	Test (programske) enote
Waterfall development	Slapovni proces razvoja
White box test	Test tipa bele škatle
XP – Extreme programming	Metodologija ekstremnega programiranja

LITERATURA IN VIRI

Abrahamsson+ 2002	P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, Agile Software Development Methods: Review and Analysis, VTT Publications 478, Espoo, 2002.
AgileAlliance	Agile Alliance, http://www.agilealliance.org, 2005.
Auer+ 2002	K. Auer, R. Miller, Extreme Programming Applied: Playing to Win, Addison-Wesley, 2002.
Basili+ 1986	V Basili, R.W. Selby, D.H. Hutchens, Experimentation in software engineering, IEEE Transactions on Software Engineering 12 (7), 1986, str. 733-743.
Basili+ 1999	V. Basili, F. Shull, F. Lanubile, Building Knowledge through Families of Experiments, IEEE Transactions on Software Engineering, Vol. 25(4), julij/avgust 1999, str. 456-473.
Beck 2000	K. Beck: Extreme programming explained: Embrace change, Addison-Wesley, Reading, Mass., 2000.
Beck 2001	K. Beck, Aim, Fire, IEEE Software, Vol. 18(5), september/oktober 2001, str. 87-89.
Beck 2003	K. Beck, Test-Driven Development: By Example, Addison-Wesley, 2003.
Beck+ 2001	K. Beck, M. Fowler, Planning Extreme Programming, Addison-Wesley, 2001.
Boehm 2002	B. Boehm, Get Ready for Agile Methods, with Care, IEEE Computer, Vol. 35(1), januar 2002, str. 64-69.
Charette 2001	R. Charette, The Decision is in: Agile Versus Heavy Methodologies, Executive Update, Vol. 2(19), Cutter Consortium, 2001.
Cockburn 2002	A. Cockburn, Agile Software Development, Addison-Wesley, 2002.
Cockburn+ 2001	A. Cockburn, J. Highsmith, Agile Software Development: The People Factor, IEEE Computer, november 2001, str. 131-133.
DeMarco+ 2002	T. DeMarco, B. Boehm, The Agile Methods Fray, IEEE Computer, Vol. 35(6), junij 2002, str. 90-92.

Erdfelder+ 1996 E. Erdfelder, F. Faul, A. Buchner: GPOWER: A general power analysis program, Behavior Research Methods, Instruments & Computers, 28, 1996, str. 1-11.

Erdogmus+ 2005 H. Erdogmus, M. Morisio, M. Torchiano: On the effectiveness of the Test-First Approach to programming, IEEE Transactions on Software Engineering, vol. 31, št. 1, 2005.

eWorkshop3 2002 več avtorjev, Summary of the Third eWorkshop on Agile Methods, University of Maryland, Fraunhofer Center for Experimental Software Engineering, Maryland, 29. oktober, 2002, http://fc-md.umd.edu/projects/Agile/3rd-eWorkshop/3rd-eWorkshop-Summary.pdf.

Fowler 2000 M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000.

George+ 2003 B. George, L. Williams: An Initial Investigation of Test Driven Development in Industry, ACM Symposium on Applied Computing SAC 2003, USA, 2003.

George+ 2004 B. George, L. Williams: A structured experiment od test-driven development, Information and Software Technology, vol. 46, Elsevier, 2004, str. 337-342.

Geras+ 2004 A. Geras, M. Smith, J. Miller: A Prototype Evaluation of Test Driven Development, Proceedings of the METRICS 2004 Conference, 2004.

Glantz 2002 S. Glantz: Primer of Biostatistics, Fifth Edition, McGraw Hill, 2002.

Heričko+ 2003 M. Heričko, M. B. Jurič, M. Rostaher, P. Repinc, Applying Practices of Extreme Programming, : HAMZA, M. H. (ur.). 21st IASTED International Multi-Conference Applied Informatics, February 10-13, 2003 Innsbruck, Austria. ACTA Press, 2003, str. 980-985.

Highsmith 2000 J. Highsmith, Adaptive Software Development: a collaborative approach to managing complex systems, Dorset House Publishing, New York, 2000.

Highsmith 2002 J. Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002.

Highsmith+ 2001 J. Highsmith, A. Cockburn, Agile Software Development: The Business of Innovation, IEEE Computer, vol. 34(9), september 2001, str. 120-122.

Höst+ 2000 M. Höst, B. Regnell, C. Wohlin, Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, Empirical Software Engineering, Vol. 5(3), 2000, str. 201-214.

Jeffries+ 2001 R. Jeffries, A. Andreson, C. Hendrickson, Extreme Programming Installed, Addison-Wesley, 2001.

Jensen 2003 R. W. Jensen, A Pair Programming Experience, Crosstalk, The Journal of Defense Software Engineering, vol. 16(3), marec 2003, str. 22-24.

Larman+ 2003 C. Larman, V. Basili: Iterative and Incremental Development: A Breif History, IEEE Computer, junij 2003.

Lindvall+ 2002 M. Lindvall, V. Basili, B. Boehm, P. Cpsta, K. Dangle, F.Shull, R. Tesoriero, L. Williams, M. Zelkowitz: Empirical Findings in Agile Methods, Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science, Vol. 2418, Springer, 2002, str. 197-207.

Lott+ 1996 C.M. Lott, H.D. Rombach, Repeatable software engineering experiments for comparing defect-detection techniques, Journal of Empirical Software Engineering 1,1996, str. 241-277.

Martin 2003 R. C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, 2003.

Maurer+ 2002 F. Maurer, S. Martel, Extreme Programming: Rapid Development for Web-Based Applications, IEEE Internet Computing, Vol. 6(1), januar/februar 2002, str. 86-90.

Maximillien+ 2003 E. M. Maximillien, L. Williams, Assesing Test-Driven Development at IBM, Proceedings of the International Conference on Software Engineering (ICSE 2003), 2003.

Melnik+ 2002 G. Melnik, F. Maurer, Perceptions of Agile Practices: A Student Survey, Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science, Vol. 2418, Springer, 2002.

Miller+ 1997 J. Miller, J. Daly, M. Wood, M. Roper, A. Brooks, Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research, Information and Software Technology, Vol. 39, 1997, str. 285-295.

Miller 2005 J. Miller, Replicating software engineering experiments: a poisoned chalice or the Holy Grail, Information and Software Technology, Vol. 47(4), 2005, str.233-244.

Mugridge 2003 R. Mugridge, Challenges in Teaching Test Driven Development, Proceedings of the 4th International Conference on Extreme Programming

and Agile Processes in Software Engineering, Genova, Italija, maj 2003.

Mugridge+ 2003 R. Mugridge in sodelavci, Five Challenges in Teaching XP, Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, Genova, Italija, maj 2003.

Müller+ 2001 M. M. Müller, W. F. Tichy, Case Study: Extreme Programming in a University Environment, Proceedings of the International Conference on Software Engineering (ICSE 2001), Toronto, Kanada, maj 2001.

Müller+ 2002 M. M. Müller, O. Hagner: Experiment about Test-first programming, Proceedings of the Conference on Empirical Assessment in Software Engineering EASE 2002, Keele, april 2002.

Müller+ 2005 M. M. Müller, F. Padberg, W. Tichy: Ist XP etwas für mich? Empirische Studien zur Einschätung von XP, Software Engineering 2005, Lecture Notes in Informatics LNI, Gesellschaft für Informatik, Essen, 2005.

Nagappan+ 2003 N. Nagappan in sodelavci, Improving the CS1 Experience with Pair Programming, Proceedings of SIGCSE 2003, 19-22 februar 2003, str. 359-362.

Newkirk+ 2001 J. Newkirk, R. C. Martin, Extreme Programming in Practice, Adison-Wesley, 2001.

Pančur 2001 M. Pančur, Analiza in določanje primernosti uporabe lahkih metodologij razvoja informacijskih sistemov, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, magistrska naloga, 2001.

Pančur+ 2003a M. Pančur, M.Ciglarič, M. Trampuš, T. Vidmar, Comparison of frameworks and tools for test-driven development. V: HAMZA, M. H. (ur.). 21st IASTED International Multi-Conference Applied Informatics, February 10-13, 2003 Innsbruck, Austria, ACTA Press, str. 980-985.

Pančur+ 2003b M. Pančur, M.Ciglarič, M. Trampuš, T. Vidmar, Testno voden razvoj v okoljih Java in .NET, Dnevi slovenske informatike = Slovenian Informatics Conference, Portorož, Slovenija, 16. – 18. april 2003. DSI 2003: zbornik posvetovanja, str. 437-442.

Pančur+ 2003c M. Pančur, M.Ciglarič, M. Trampuš, T. Vidmar: Towards Empirical Evaluation of Test-Driven Development in a University Environment, Proceedings of the IEEE EUROCON 2003, Ljubljana, Slovenia, 22 – 24 september, 2003.

Porter+ 1995 A. Porter, L. Votta: Comparing Detection Methods for Software Requirements Inspection: A Replicatiojn Using Professional Subjects,

Empirical Software Engineering, vol. (3), 1995, str. 355-380.

QSM 2000 QSM Associates: Companies endure Delays in Order to Obtain Beter Results, News & Newsworthy, QSM Associates, maj 2000, http://www.qsma.com/news_companies_endure.html, 5.4.2003.

Reifer 2002 D. J. Reifer, How Good Are Agile Methods?, IEEE Software, vol. 19(4), julij/avgust 2002, str. 16-18.

Rostaher+ 2002 M. Rostaher, M. Heričko, Tracking Test First Pair Programming - An Experiment, XP/Agile Universe 2002, Extreme Programming and Agile Methods, Lecture Notes in Computer Science 2418, 2002, str. 174-185.

Standish 1995 The Standish Group, The CHAOS Report (1994), http://www.standishgroup.com/sample_research/chaos_1994_1.php, 11.4.2003.

Standish 2003 The Standish Group, Latest Standish Group CHAOS Report Shows Project Success Rates Have improved by 50%, Press Release - March 25, 2003, http://www.standishgroup.com/press/article.php?id=2, 4.4.2003.

Succi+ 2001 G. Succi, M. Marchesi, (eds.), Extreme Programming Examined, Addison-Wesley, 2001.

Tichy 1998 W. F. Tichy, Should Computer Scientist Experiment More?, IEEE Computer, vol. 31(5), maj 1998, str. 32-40.

Tichy 2000 W. Tichy: Hints for Reviewing Empirical Work in Software Engineering, Empirical Software Engineering, vol. 5, 2000, str. 309-312.

Wake 2002 W.C. Wake, Extreme Programming Explained, Addison-Wesley, 2002.

Williams 2000 L. Williams, The Colaborative Software Process, doktorska disertacija, University of Utah, avgust 2000.

Williams+ 2000 L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries, Strenghtening the Case for Pair-Programming, IEEE Software, vol. 17(4), julij/avgust 2000, str. 19-25.

Williams+ 2002 L. Williams in sodelavci, In Support of Pair Programming in the Introductory Computer Science Course, Computer Science Education, vol.12(3), 2002, str. 197-212.

Wohlin+ 2001 C. Wohlin, M. Höst, Special section: Controlled Experiments in Software Engineering, Information and Software Technology, Vol. 43, 2001, str. 921-924.

IZJAVA

Izjavljam, da sem doktorsko disertacijo izdelal samostojno pod vodstvom mentorja doc. dr. Toneta Vidmarja in somentorja prof. dr. Blaža Zupana. Pomoč ostalih sem v celoti navedel v zahvali.

Ljubljana, september 2005

mag. Matjaž Pančur

ZAHVALA

Najprej se zahvaljujem mentorju doc. dr. Tonetu Vidmarju in somentorju prof. dr. Blažu Zupanu za strokovno vodstvo pri delu, vse koristne nasvete in veliko mero potrpežljivosti.

Zahvala velja tudi doc. dr. Mojci Ciglarič za natančen pregled, pripombe, pomoč pri izvedbi eksperimenta in številnim koristnim diskusijam ob kavi. Sotrpinu dr. Mateju Trampušu hvala za vse konstruktivne debate (tokrat ob čaju) in pomoč pri izvedbi eksperimenta s samostojnimi razvijalci v letu 2003, doc. dr. Patriciju Buliču pa za pomoč pri izvedbi eksperimenta s samostojnimi razvijalci v letu 2004.

Zahvala za sodelovanje v eksperimentu gre tudi študentom četrtega letnika FRI v šolskem letu 2002/2003 in 2003/2004.

Najlepša hvala tudi mojim in ženinim staršem za spodbudo in razumevanje.

Na koncu pa še iskrena zahvala moji boljši polovici Nataši: brez tvoje potrpežljivosti, spodbujanja in podpore ne bi šlo!

PRILOGA A – REZULTATI MERITEV EKSPERIMENTA S SAMOSTOJNIMI RAZVIJALCI 2003 (SR03)

Rezultati meritev za eksperiment s samostojnimi razvijalci 2003 (SR03), ki smo ga zaradi suma pristranosti zaradi smrtnosti izločili. Ker neizpodbitnih statističnih dokazov o tem ni, meritve objavljamo v prilogi, da bi jih drugi raziskovalci lahko po potrebi uporabili v svojih študijah. Tu je potrebno poudariti, da **tudi če bi rezultate proglasili za veljavne in bi jih upoštevali v analizi**, le-ti **ne spremenijo naših ugotovitev**.

V skupini TDD je bilo na začetku predvidenih 20 razvijalcev, od tega se eden ni udeležil tega dela eksperimenta, dva razvijalca pa smo zaradi nespoštovanja procesa dela izločili sami (neustrezno testiranje – nista v popolnosti sledila procesu razvoja TDD). V tej skupini je tako 17 ustreznih meritev, kar nam da 15% mero smrtnosti.

V skupini ITL je bilo na začetku 18 razvijalcev, od tega se trije niso udeležili tega dela eksperimenta, 4 razvijalce pa smo izločili zaradi nespoštovanja procesa dela (neustrezno testiranje oz. pomanjkanje le tega). V tej skupini je tako le 11 ustreznih meritev, kar nam da kar 39% mero smrtnosti! V primeru treh metrik (glej tabelo spodaj, % časa v posamezni fazi) pa je to razmerje še večje (17-9). Dva razvijalca v tej skupini sta namreč zelo zelo slabo označevala posamezne faze dela, zato njunih meritev tu nismo upoštevali.

Skupini sta tako najverjetneje postali neuravnovešeni, čeprav statistično značilnih razlik z neparametričnim MannWhitney U testom (α=0,1) nismo zaznali. Še posebej je k sumu na pristranost zaradi smrtnosti pripomoglo to, da so bili vsi kandidati, ki smo jih izločili zaradi neustreznega testiranja v ITL skupini, slabši po ocenah in je zato večja tudi možnost, da bodo rezultati ITL skupine popačeni. V ITL skupini tako na primer ni bilo pri tem eksperimentu **niti enega študenta več iz najslabše kategorije** (od treh – slabi, srednji in dobri študentje) gledano po povprečni oceni (torej spodnjih 25% študentov

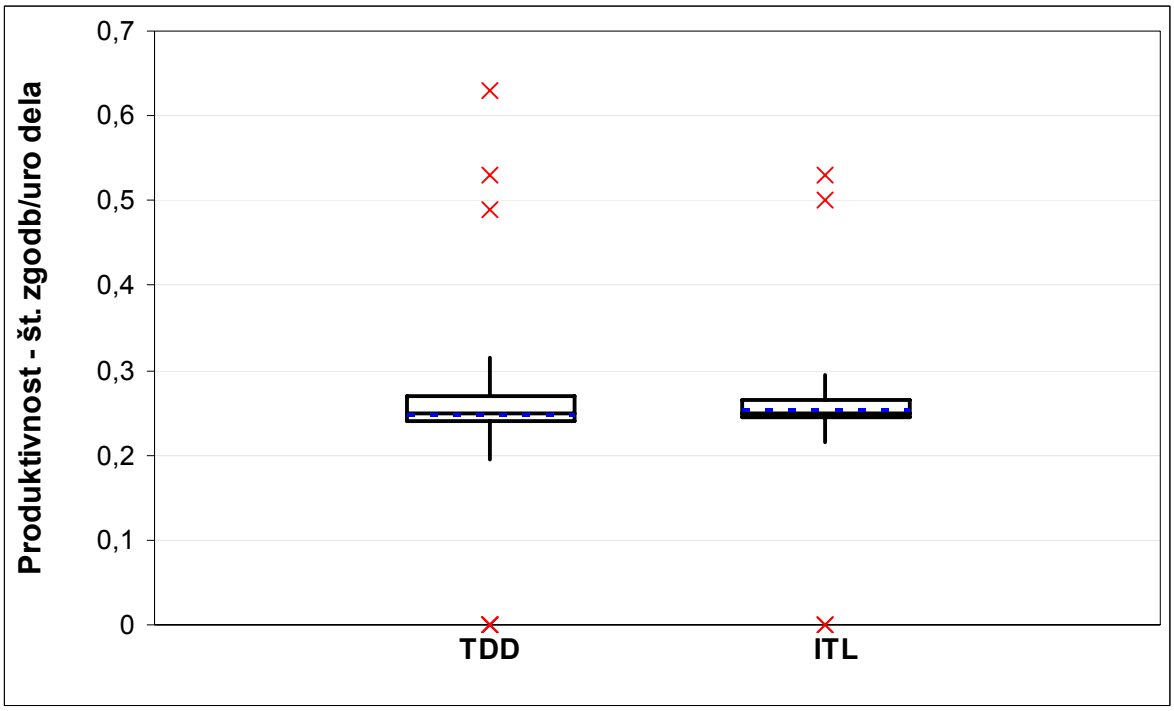
gledano po povprečni oceni). Prav tako se v tem testu zaradi manjšega števila subjektov zmanjša tudi statistična moč testa, kar še dodatno oteži postavljanje zaključkov.

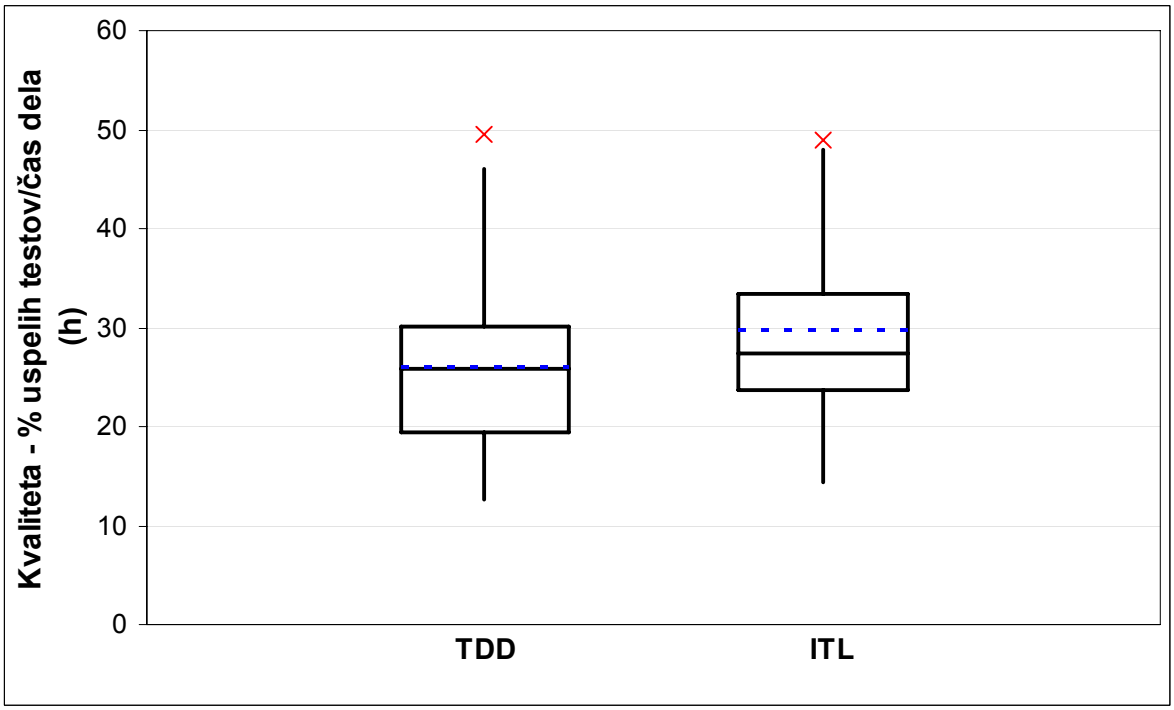
Rezultati neparametričnega Mann-Whitney testa po večini niso statistično značilni. Izjema sta le metriki "odstotek uspešnih testov" (p=0,01) (kar le potrjuje naše zaključke) in "odstotek časa v fazi kodiranja" (p=0,06), ki pa vseeno ne prinaša ničesar novega v naše zaključke. V Tabeli Priloga A-1 so zbrani povzetki meritev, sledijo grafi prikaza kvantilov (*boxplot graph*) za vsako metriko posebej.

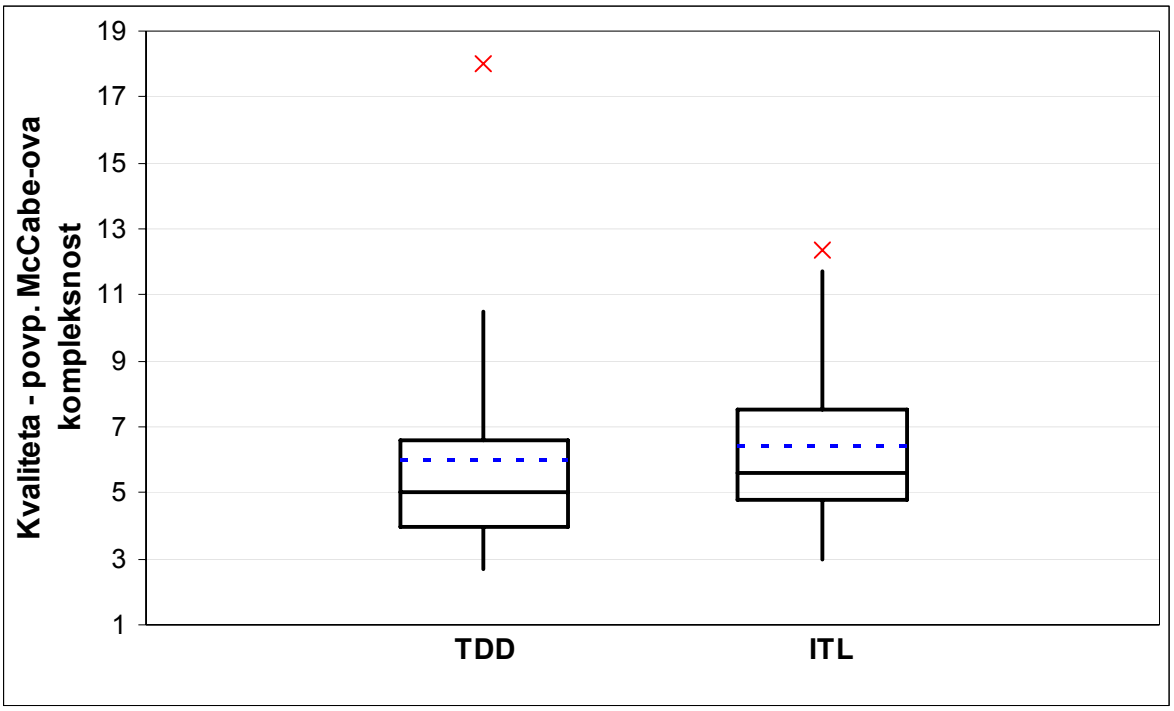
Tabela Priloga A-1: Povzetek rezultatov meritev v eksperimentu s samostojnimi razvijalci 03 (SR03)

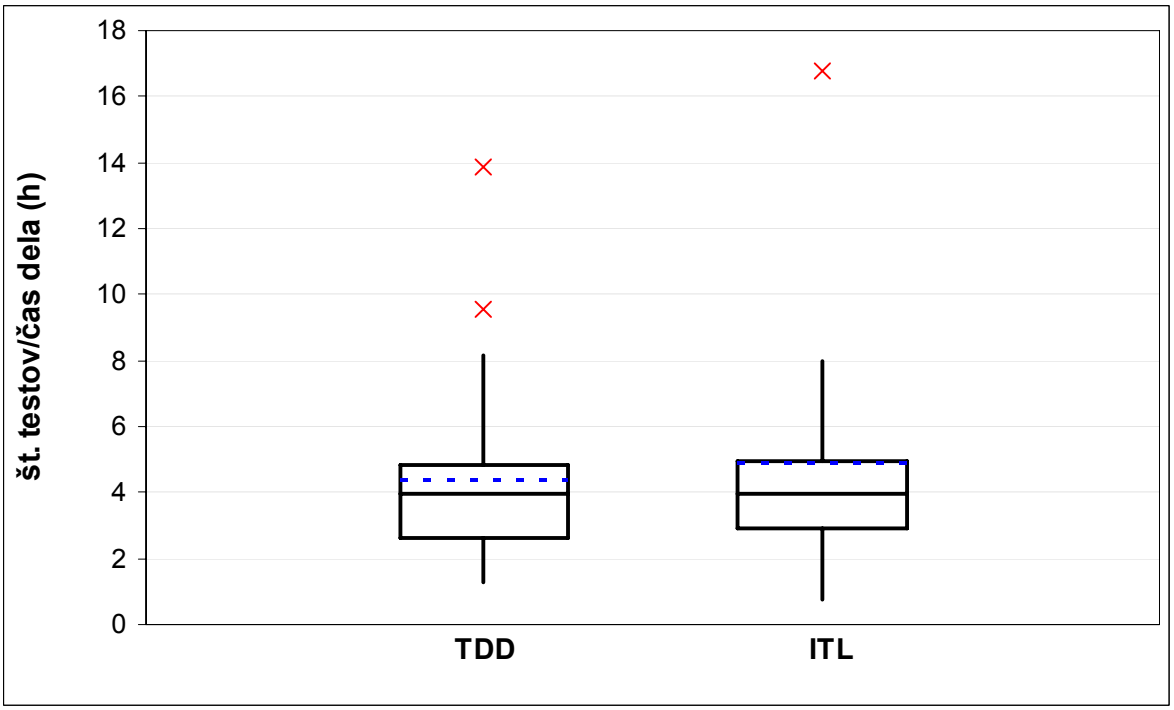
	Metrika	**Skupina**	**Št. meritev**	**Povp. vr.**	**St. dev.**	**Mediana**	**Mann-Whitney *p***	**Cohen *d***
Produktivnost	št. dokončanih zgodb/čas dela (h)	TDD	17	0,25	0,18	0,25	0,49 enosmerni	<0,1
		ITL	11	0,25	0,16	0,25		
Kvaliteta	Zunanja za prvo zgodbo (% uspelih spr. testov/čas dela(h))	TDD	17	26,08	9,04	25,74	0,22 enosmerni	0,36
		ITL	11	29,62	10,45	27,43		
	Notranja (povpr. McCabeova kompl.)	TDD	17	6,01	3,69	5,00	0,18 enosmerni	0,13
		ITL	11	6,43	2,94	5,60		
Testi	št. preverjanj v uri dela	TDD	17	4,39	3,11	3,99	0,42 enosmerni	0,13
		ITL	11	4,87	4,35	3,99		
	Kvaliteta testov (% pokritosti)	TDD	17	95,29	4,38	97,00	0,4 enosmerni	<0,1
		ITL	11	95,00	5,37	96,00		

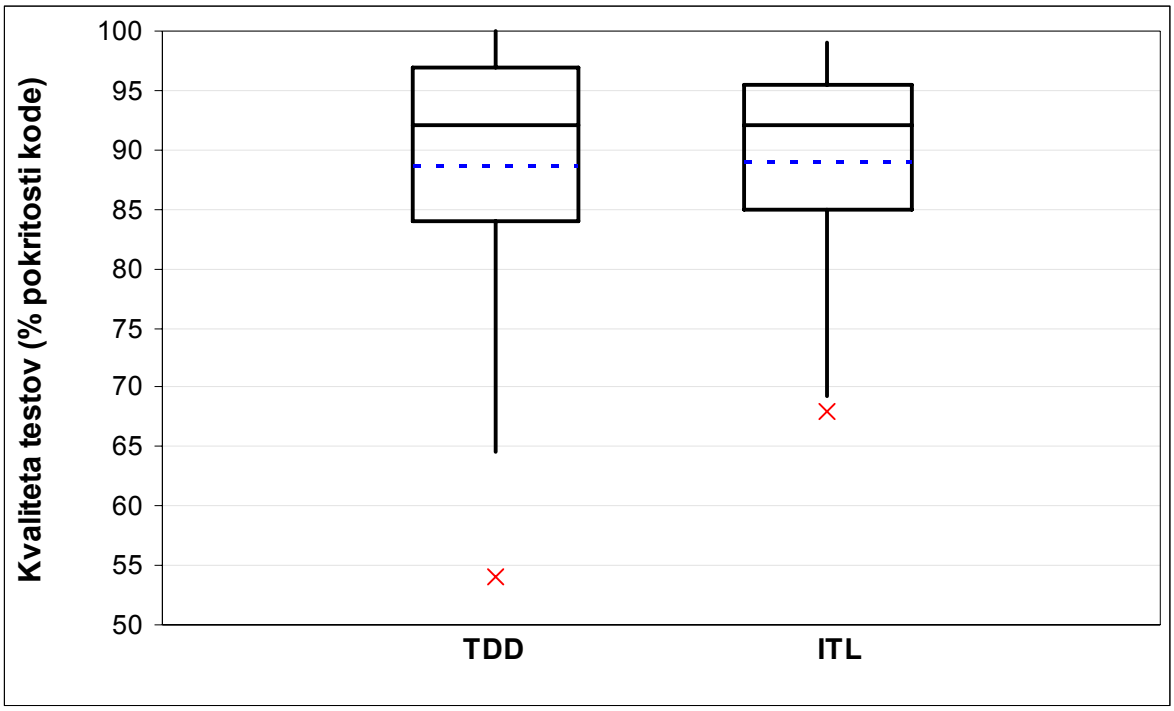
Proces dela	Frekv. poganjanja testov (testi/uro dela)	TDD	17	28,02	9,32	28,81	0,17 enosmerni	0,54
		ITL	11	33,52	11,03	30,07		
	% uspešnih testov	TDD	17	33,59	14,17	30,00	0,01 enosmerni	0,97
		ITL	11	46,18	11,57	44,00		
	% časa v kodiranju	TDD	17	60,71	16,66	57,00	0,06 dvosmerni	0.97
		ITL	9	46,89	11,38	49,00		
	% časa v testiranju	TDD	17	36,35	15,37	39,00	0,26 dvosmerni	0,42
		ITL	9	42,56	14,28	42,00		
	% časa v preoblikovanju	TDD	17	3,12	3,33	2,00	0,26 dvosmerni	0,68
		ITL	9	10,67	15,32	5,00		
	Razmerje testi/koda	TDD	17	1,20	0,92	0,91	0,71 dvosmerni	0,12
		ITL	11	1,11	0,45	1,05		

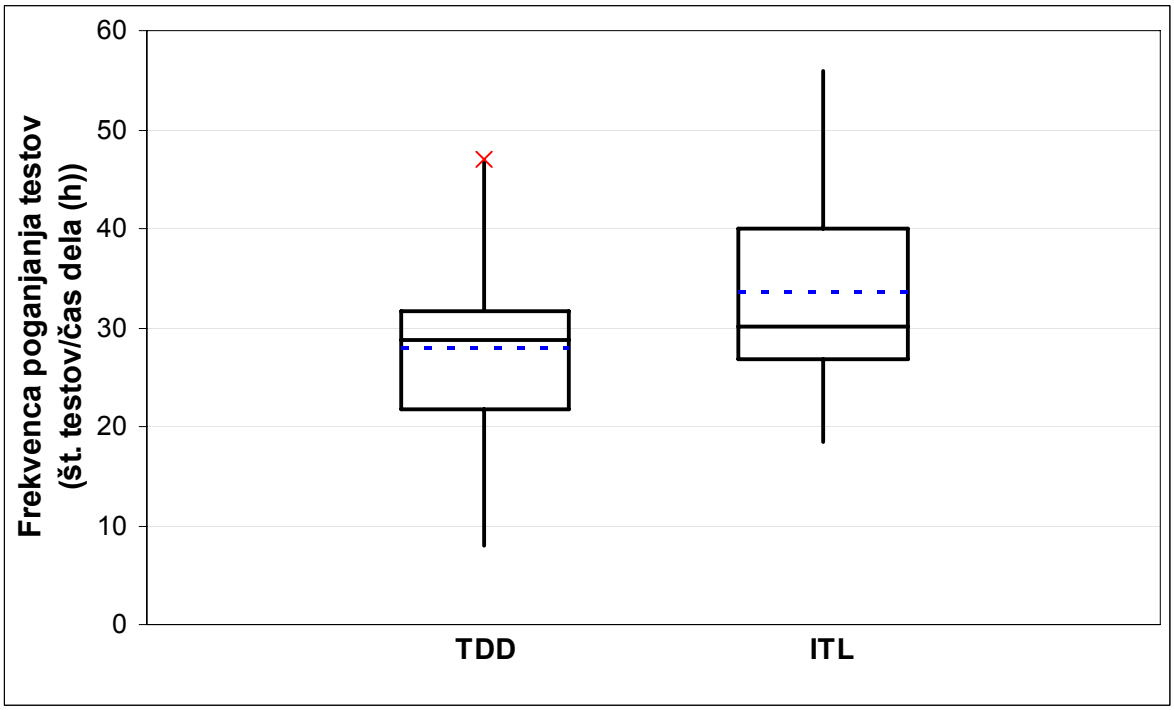

Produktivnost - št. zgodb/uro dela
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
TDD
ITL

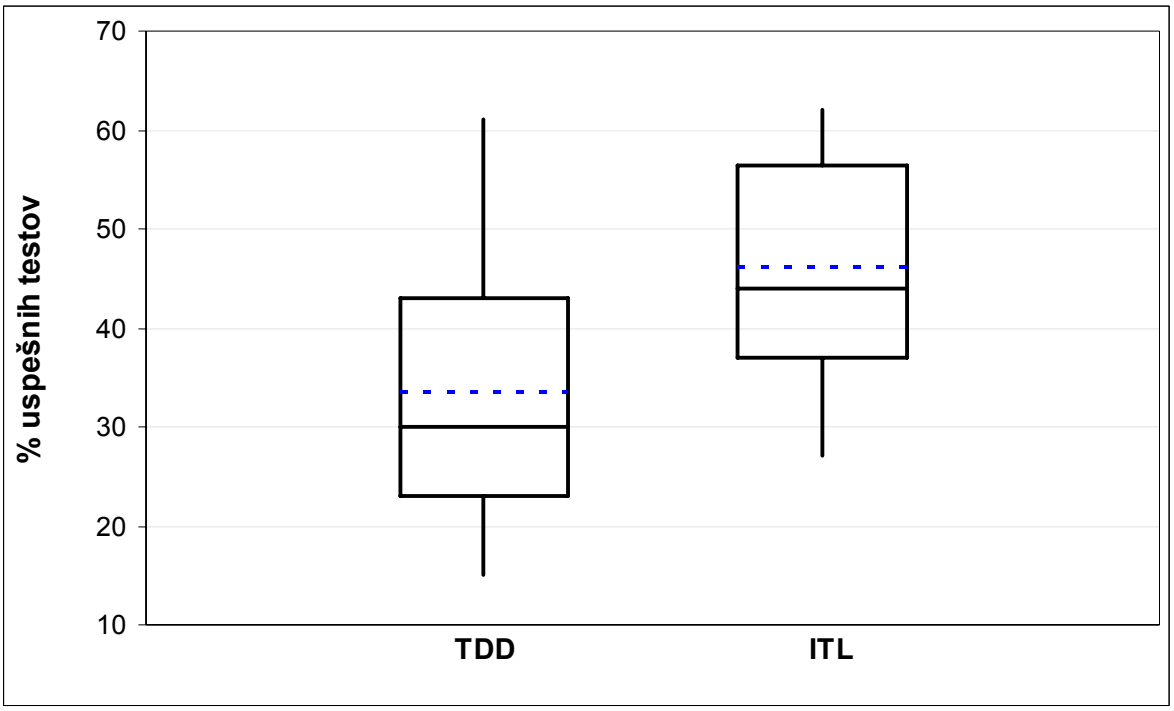

Kvaliteta - % uspelih testov/čas dela (h)
60
50
40
30
20
10
0
TDD
ITL

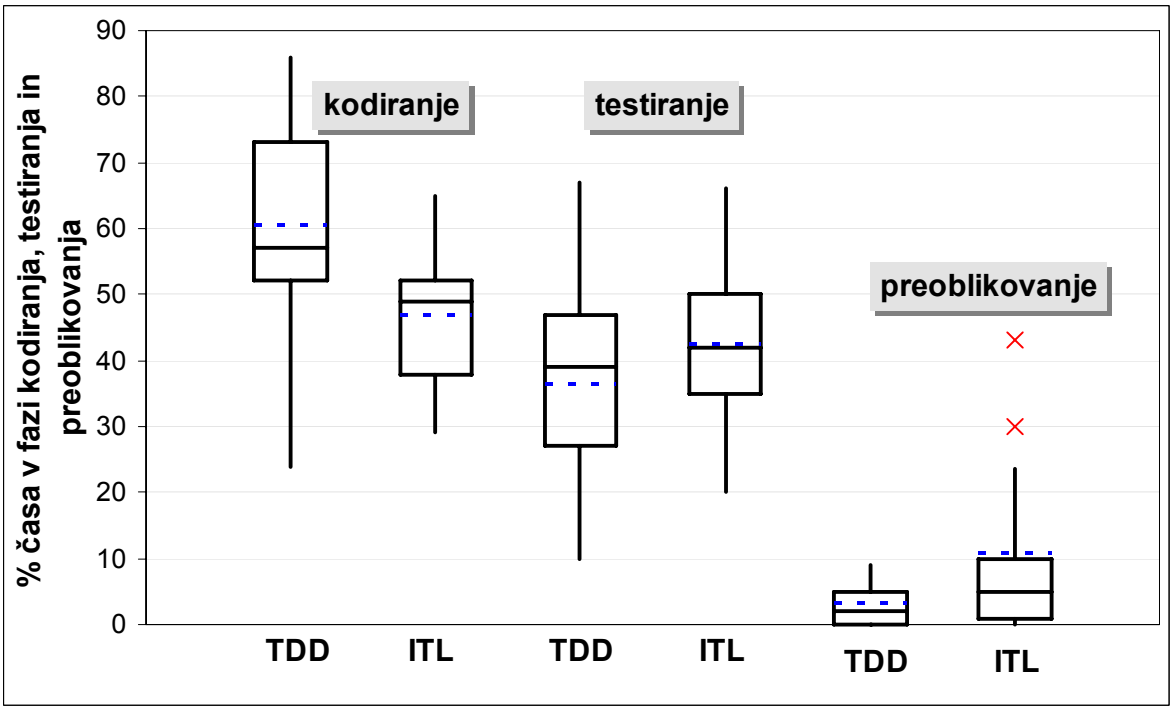

Kvaliteta - povp. McCabe-ova kompleksnost
19
17
15
13
11
9
7
5
3
1
TDD
ITL

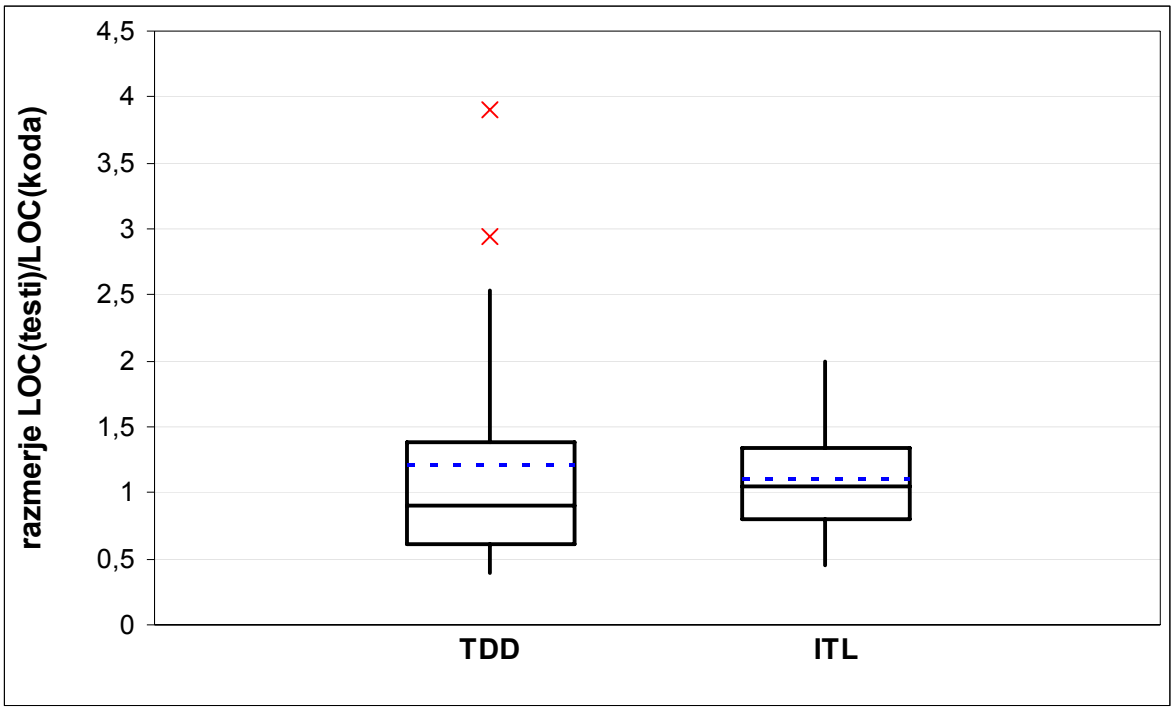

št. testov/čas dela (h)
18
16
14
12
10
8
6
4
2
0
TDD
ITL


Kvaliteta testov (% pokritosti kode)
100
95
90
85
80
75
70
65
60
55
50
TDD
ITL


Frekvenca poganjanja testov
(št. testov/čas dela (h))
60
50
40
30
20
10
0
TDD
ITL


% uspešnih testov
70
60
50
40
30
20
10
TDD
ITL


% časa v fazi kodiranja, testiranja in preoblikovanja
90
80
70
60
50
40
30
20
10
0
kodiranje
testiranje
preoblikovanje
TDD
ITL
TDD
ITL
TDD
ITL


razmerje LOC(testi)/LOC(koda)
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
TDD
ITL

PRILOGA B – EKSPERIMENTALNE NALOGE

V prilogi podajamo **le najpomembnejši del** pisnih navodil, ki so jih dobili subjekti v eksperimentu (manjkajo navodila za vspostavitev okolja, proces dela, itd.). Ob vsaki nalogi so bila podana tudi ustna navodila s podrobnejšo razlago problema.

Eksperiment PP03

Simulacija podatkovne baze z replikacijo

Imamo strežnik (DBService.java), ki nam nudi dostop do več logičnih podatkovnih baz (npr. baza1, baza2, krenabaza, ...). Strežnik je povezan z drugimi istovrstnimi strežniki na drugih lokacijah, skupaj s katerimi formira porazdeljeno podatkovno bazo. Potrebno je zagotoviti replikacijo - vse spremembe podatkov iz posamezne logične baze se pošljejo tudi tistim bazam na drugih lokacijah, ki jih so nanje "naročene" (torej tistim, ki imajo tudi same kopijo posamezne logične podatkovne baze).

Podatki, ki so v posamezni logični podatkovni "bazi" (beri: tabeli z dvema stolpcema), so sestavljeni iz para atributov: ID tipa int in vrednost tipa String. Da bo lažje, lahko predpostavite, da je torej ID pozitivno celo število, vrednost (vsebina zapisa) pa je sestavljena iz številk in znakov (presledki, ločila in drugi znaki v vsebini niso dovoljeni).

Vaša naloga je, da implementirate strežnik DBService. DBService je večniten strežnik, torej lahko hkrati streže več odjemalcem. Odjemalci so v bistvu baze na drugih lokacijah, ki se priklapljajo nanj in mu pošiljajo svoje zahteve oz. ukaze (select, insert, delete, update, connect, stop, ...). Format ukazov (sporočil) je predpisan (glej spodaj), vedno se zaključi se z "newline" znakom (torej je ukaz vedno v eni vrstici). Odjemalci so na strežnik stalno priklopljeni, razen če se sami ne odklopijo

Posmezne "zgodbe" (primeri uporab oz. zahteve), ki jih morate implementirati, so:

1. Ugasni DBService
2. Predstavitev odjemalca strežniku DBService
3. Kreiraj bazo
4. Priklopi se na bazo
5. Vpis v bazo
6. Izpiši vse zapise v posamezni bazi
7. Sprememba zapisa v bazi
8. Brisanje zapisa v bazi
9. Iskanje po vsebini

1. Ugasni DBService

Strežniku pošljete sporočilo Stop, ki povzroči, da se strežnik ugasne (neha streči). Preden se strežnik ugasne, vsem svojim priklopljenim odjemalcem pošlje sporočilo Stopping. Odjemalcu ni potrebno biti ne predstavljen (zgodba 2) ne priklopljen (zgodba 4).

2. Predstavitev odjemalca strežniku DBService

Odjemalec (baza na drugi lokaciji) se priklopi na strežnik z ukazom Name ime_lokacije, s čimer se pravzaprav predstavi našemu strežniku. Ime lokacije je sestavljeno le iz črk in številk in brez šumnikov (brez ločil, presledkov,...).

- Če je ime_lokacije že zasedeno, strežnik odgovori z NameTaken.
- Če je ime napačno (ni sestavljeno le iz črk in številk ali pa je prazno), strežnik odgovori z NameInvalid.
- Če je vse OK, vrne server Named ime_lokacije.
- Če je klient že uspešno predstavljen, pa se hoče še enkrat predstaviti (z istim imenom ali z drugim imenom!), vrne strežnik NameAlreadyDefined.

3. Kreiraj bazo

Odjemalec kreira bazo z ukazom Create ime_baze. Prej mora biti odjemalec strežniku že predstavljen (zgodba 2).

- Če odjemalec ni predstavljen, mu odgovori z NameNotDefined. Tako dela tudi v vseh ostalih primerih (preostalih zgodbah spodaj)!
- Če baza že obstaja, strežnik odgovori s CreateFailed.
- Če je baza uspešno kreirana, strežnik odgovori z Created ime_baze. Ime_baze je podobno kot ime_lokacije sestavljeno le iz črk in številk, brez presledkov in ločil in šumnikov.
- Kreiranje baze še ne pomeni, da je odjemalec nanjo "naročen" (glej "Priklopi se na bazo" spodaj).
- Če je ime napačno (nepravilno), vrne Error (glej pravilo, opisano na koncu zgodb)

4. Priklopi se na bazo

Odjemalec se "naroči" (priklopi) na določeno bazo tako, da pošlje ukaz Connect ime_baze. Strežnik ga vpiše v svojo listo odjemalcev, ki jih bo ob spremembah podatkov v bazi ime_baze obveščal. Odjemalec lahko sedaj pričakuje, da bo od strežnika dobival sporočila o spremembah podatkov v bazi ime_baze. Prej mora biti odjemalec strežniku že predstavljen (zgodba 2).

- Če je priklop uspešen, strežnik odgovori z ConnectedOn ime_baze.
- Če baza ne obstaja, strežnik odgovori z "NoSuchDatabase.
- Odjemalec se lahko priklopi na isto bazo tudi večkrat, vsakič pač dobi ConnectedOn ime_baze.

5. Vpis v bazo

Z ukazom Insert ime_baze ID/vrednost lahko odjemalec vpiše v bazo ime_baze par ID/vrednost (npr. Insert baza1 1/krneki1). Prej mora biti odjemalec predstavljen in priklopljen na ime_baze.

- Če odjemalec ni priklopljen (zgodba 4), odgovori strežnik z NotConnected ime_baze. Tako dela tudi v zgodbah 7, 8, 9, 10!
- če je v bazi ime_baze že zapis s tem IDjem, strežnik vrne DuplicateID ime_baze ID/vrednost (npr. DuplicateID baza1 1/krneki1).
- če je vse ok, v bazi ime_baze zapisa "ID/vrednost" še ni, ga strežnik doda v bazo ime_baze. Takoj nato obvesti vse odjemalce, "priklopljene" na bazo ime_baze (tudi tega, ki je Insert sprožil), o spremembi s sporočilom Inserted ime_baze ID/vrednost (npr. Inserted baza1 1/krneki1). ("Insert ..." in "Inserted ..." se piše z veliko začetnico!).

6. Izpiši vse baze

Odjemalec lahko z ukazom Select * dobi izpis vseh baz v strežniku. Prej mora biti odjemalec strežniku predstavljen, priklopljen pa mu ni treba biti!

- Če nobena baza še ne obstaja, odgovori z NoDatabases
- Če je ena baza kreirana, strežnik odgovori z Databases lista_baz_ločena_z_vejicami (npr. "Databases ime_baze1, ime_baze2, ime_baze3, ime_baze4"), pri čemer so imena baz urejena po abecedi in ločena z vejicami in presledki za vejicami. Za zadnjim imenom ni vejice in presledka!

7. Izpiši vse zapise v posamezni bazi

Odjemalec lahko z ukazom Select * from ime_baze dobi izpis vseh zapisov (ID/vsebina) iz baze ime_baze. Prej mora biti odjemalec predstavljen in priklopljen v bazo ime_baze.

- Strežnik odgovori z Database ime_baze ID1/vsebina1, ID2/vsebina2, ID3/vsebina3 ..., IDN/vsebinaN pri čemer so IDi/vsebine urejeni naraščajoče po vrednosti IDja in ločena z vejicami in presledki za vejicami. Za zadnjim imenom ni vejice in presledka!
- Če je baza prazna, vrne NoSuchRecord ime_baze.

8. Sprememba zapisa v bazi

Z ukazom Update ime_baze ID/vrednost (npr. Update baza1 1/krneki2) lahko odjemalec spremeni zapis pri določenem identifikatorju ID. Prej mora biti odjemalec predstavljen in priklopljen v bazo ime_baze.

- Če je v bazi ime_baze zapis s takšnim ID ne obstaja, strežnik vrne NoSuchID ime_baze ID/vrednost (npr. NoSuchID baza1 1/krneki2).
- Če je vse OK (takšen ID v bazi ime_baze obstaja), strežnik spremeni zapis s tem IDjem na novo vrednost. Takoj nato obvesti vse odjemalce, "priklopljene" na bazo ime_baze (tudi tega, ki je Update sprožil), o spremembi s sporočilom Updated ime_baze ID/vrednost (npr. updated baza1 1/krneki2).

9. Brisanje zapisa v bazi

Z ukazom Delete ime_baze ID (npr. Delete baza1 1) lahko odjemalec zbriše zapis z določenem identifikatorjem ID.

- če je v bazi ime_baze zapis s takšnim ID ne obstaja, strežnik vrne NoSuchID ime_baze ID (npr. NoSuchID baza1 1).
- če je vse OK (takšen ID v bazi ime_baze obstaja), strežnik zbriše zapis "ID/vrednost". Takoj nato obvesti vse odjemalce, "priklopljene" na bazo ime_baze (tudi tega, ki je Delete sprožil), o spremembi s sporočilom Deleted ime_baze ID/vrednost (npr. Deleted baza1 1/krneki2). PAZI! Deleted vrne TUDI VSEBINO ZAPISA, torej celoten "ID/vsebina"!

10. Iskanje po vsebini zapisov

Z ukazom Select iskani_niz from ime_baze (npr Select krneki from baza1) dobi odjemalec izpis vseh tistih zapisov, ki vsebujejo kjerkoli v atributu vrednost podniz iskani_niz. Iskanje razlikuje med velikimi in malimi črkami. "Jokerji" v nizu iskani_niz niso dovoljeni (zaenkrat).

- Strežnik odgovori z Database ime_baze ID1/vsebina1, ID2/vsebina2, ID3/vsebina3 ..., IDN/vsebinaN pri čemer so IDi/vsebine urejeni naraščajoče po vrednosti ID-ja in ločena z vejicami in presledki za vejicami. Za zadnjim imenom ni vejice in presledka! Izpiše pa seveda le tiste zapise, ki imajo v atributu vrednost podniz iskani_niz.
    - primer: v bazi baza1 so zapisi 1/krneki123, 2/2krneki2, 3/blabla, 4/tralala. Ukaz Select krneki from baza1 vrne

      Database baza1 1/krneki123, 2/2krneki2 (bodite pozorni, da na koncu ni vejice in presledka!)
- če v bazi ni takšnih zapisov, strežnik odgovori NoSuchRecord ime_baze (npr. NoSuchRecord baza1)

Ukazi, ki spreminjajo podatke, se lahko v strežniku izvajajo serijsko (eden za drugim - uporabite sinhronizacijo za dostop do lokalnih baz!). Vsi ukazi se začnejo z veliko črko. Vsa imena, ukazi, itd. so občutljivi na male in velike črke (case-sensitive).

Pomembno:

Zgodbe implementirajte po vrsti, od št. 1 naprej. Šele ko ste končali implementacijo posamezne zgodbe, nadaljujte z naslednjo zgodbo! Pred začetkom implementacije vsake zgodbe z Note napišite komentar "začetek zgodba 1", ko je implementacija končana, napišite "konec zgodba 1". Vedno poganjajte vse teste (AllTests.java, ko napišete nov TestCase, ga ne pozabite ažurirati (še enkrat poženete Wizard :), da bo vključil tudi nove teste). Struktura DBService in ostalih vaših razrtedov je poljubna, tako da si npr. lahko dodajate metode, ki jih potrebujete pri testiranju. Preden začnete, si tudi malo poglejte Java Collections framework (razredi v java.util, ki jih boste uporabljali za implementacijo "baze" (torej za shranjevanje zapisa ključ/vsebina, za shranjevanje referenc na delovne niti, ...)). Še namig: Collections framework ima tudi Sorted, prav tako ima že tudi ovojnice (wrapper) za sinhronizacijo dostopa do kolekcije (primer tega ste že imeli pri Craftsmanu - spomnite se na Collection.synchronizedList(new LinkedList())).

Kadar je karkoli narobe s katerimkoli ukazom (npr. nepravilen format ukaza, nepravilen format imen baz, lokacij, nepravilen ID (ni pozitivno število), presledki tam, kjer jih ne sme biti, itd.), naj strežnik odgovori z "Error", razen če ni zgoraj že predpisan drugačen odgovor.

Če česa ne veste ali ni jasno definirano, vprašajte! (v splošnem v takem primeru vprašate stranko - tukaj sem seveda to jaz :))

Strežnik (oz. razred DBService) naj (poleg morebitnih vaših) implementira tudi interface sat.Server (uporabljal ga bom za testiranje vaših implementacij). Importajte v projektni Java BuildPath (podobno kot dodate junit.jar, z desnim gumbom kliknete na projekt, nato properties/Java BuildPath/Libraries/Add External JARs) sat.jar (v LRI4 in LRK je v c:\eclipse\lib poddirektoriju). sat.jar in fit.jar si lahko skopirate doma kamorkoli, le v vaš workspace direktorij ne (da ne bodo problemi z local history).

Interface sat.Server je definiran v paketu sat.jar, vsebina pa vam je že znana:

```
public interface Server {
    public int getPortNo();
    public boolean isRunning();
}
```

- getPortNo() vrne št. porta, na katerem posluša vaš stežnik (npr. 9999)
- isRunning() pa vrne true, če strežnik in strežniške niti še tečejo, in false, ko se strežniške niti (in primarna strežniška nit) končajo.

V primarni strežniški niti (nit, ki kreira ostale strežniške niti, ki poskrbijo za posameznega klienta) tako npr. na začetku dajte privatno spremenljivko running na true, ko se delo strežniških niti konča, jo dajte na false. isRunning() vrača stanje te spremenljivke (in s tem stanje strežnika - ali še kaj dela ali je končal). Na ta način bom lahko npr. ugotovil, ali se strežnik konča, ko dobi ukaz stop, itd...

SAT (SocketAcceptanceTester) je program, ki omogoča izvajanje sprejemnih (acceptance) testov programov, ki uporabljajo vtiče. Je nadgradnja programa FIT (http://fit.c2.com). Lokalna kopija sat.jar in fit.jar je v \\lrk-vaje.fri.uni-lj.si\install\pvis\eclipse - additional plugins\lib direktoriju, drugače pa ga lahko dol potegnete tudi z Interneta (download 111K). Vaše teste pišite tako kot do sedaj, z uporabo JUnit-a, in ne morebiti s SAT-om, ki je namenjen za t.i. sprejemne teste (acceptance tests), ki jih bom definiral jaz! Sprejemne teste za seminar boste dobili naslednji teden (skupaj z navodili za uporabo), seveda pa bodo ti testi pregledovali le minimum, še zdaleč ne bodo preigrali vseh možnih scenarijev, zato se samo na njih ne zanašajte, ampak pišite svoje JUnit teste! Na koncu bom jaz testiral vašo kodo s precej večjim naborom testov.

Pisanja strežnika se lotite inkrementalno (podobno kot v seriji craftsman). Koda naj bo čim bolj robustna (torej neobčutljiva za napake "uporabnika").

Eksperiment PP04

Strežnik za kramljanje s P2P prenosom datotek

Imamo strežnik (ChatService.java), ki nam nudi možnost kramljanja z ostalimi uporabniki. Naš strežnik je povezan z drugimi istovrstnimi strežniki na drugih lokacijah, skupaj s katerimi formira enoten sistem. Potrebno je zagotoviti njihovo medsebojno komuniciranje (chat sistem) in možnost skupne uporabo "datotek", ki jih vsak odjemalec prispeva v sistem.

Datoteke se ne bodo prenašale v resnici, ampak se bodo prenašali samo njihova imena (tipa String), sestavljena iz številk/znakov (brez šumnikov) ter pike, minusa in podčrtaja ("_"). Ostalo ni dovoljeno.

Vaša naloga je, da implementirate strežnik ChatService. ChatService je večniten strežnik, torej lahko hkrati streže več odjemalcem. Format ukazov (sporočil) je predpisan (glej spodaj), vedno se zaključi se z "newline" znakom (torej je ukaz vedno v eni vrstici). Odjemalci so na strežnik stalno priklopljeni, razen če se sami ne odklopijo.

Posmezne "zgodbe" (primeri uporab oz. zahteve), ki jih morate implementirati, so:

1. Ugasni ChatService

Strežniku pošljete sporočilo Stop, ki povzroči, da se strežnik ugasne (neha streči). Preden se strežnik ugasne, vsem svojim priklopljenim odjemalcem pošlje sporočilo Stopping. Odjemalcu ni potrebno biti ne predstavljen (zgodba 2) ne priklopljen v kakšnega od kanalov (zgodba 4).

2. Predstavitev odjemalca strežniku ChatService

Odjemalec se predstavi strežniku z ukazom Name ime_odjemalca. Ime ime_odjemalca je sestavljeno le iz črk/številk in brez šumnikov (brez ločil, presledkov,...).

- če je ime_odjemalca že zasedeno, strežnik odgovori z NameTaken.
- če je ime napačno (ni sestavljeno le iz črk in številk ali pa je prazno), strežnik odgovori z Error.
- če je vse OK, vrne server Named ime_odjemalca. S tem je odjemalec predstavljen (poimenovan), imena pa ne more več spreminjati.
- Če je odjemalec že uspešno predstavljen, pa se hoče še enkrat predstaviti (ali z istim imenom ali z drugim veljavnim imenom, ne glede na to ali je to zasedeno ali še prosto !), vrne strežnik NameAlreadyDefined. Če se hoče preimenovati v neveljavno ime, strežnik odgovori z Error.

3. Kreiraj kanal (chat channel)

Odjemalec kreira kanal, na keterem se pogovarja z ostalimi priklopljenimi uporabniki z ukazom Create ime_kanala. Prej mora biti odjemalec strežniku že predstavljen (zgodba 2).

- Če odjemalec ni predstavljen, mu odgovori z NameNotDefined. Tako dela tudi v vseh ostalih primerih (preostalih zgodbah spodaj, razen če ni drugače definirano)!
- Če kanal že obstaja, strežnik odgovori s ChannelAlreadyCreated.
- Če je baza uspešno kreirana, strežnik odgovori z Created ime_kanala. Ime_kanala je podobno kot ime_odjemalca sestavljeno le iz črk in številk, brez presledkov, ločil in šumnikov.
- Kreiranje kanala še ne pomeni, da se je odjemalec vanj priklopil (glej "Priklopi se na kanal" spodaj).
- Če je ime napačno (nepravilno), vrne strežnik Error.

4. Priklopi se na kanal

Odjemalec se priklopi na določen kanal tako, da pošlje ukaz Connect ime_kanala. Strežnik ga vpiše v svojo listo odjemalcev, ki jih bo ob novih sporočilih v kanal ime_kanala obveščeni. Odjemalec lahko sedaj pričakuje, da bo od strežnika dobival sporočila ostalih uporabnikov ime_kanala. Prej mora biti odjemalec strežniku že predstavljen (zgodba 2 - če ni, strežnik vrne NameNotDefined). Odjemalec je lahko priklopljen v več kanalov hkrati.

- Če je priklop uspešen, strežnik pošlje vsem priklopljenim (tudi temu, ki se je pravkar priklopil) [ime_kanala] ime_odjemalca connected (npr. če odjemalec Jagoda napiše Connect PVIS, strežnik vsem priklopljenim pošlje [PVIS] Jagoda connected). Sedaj dobiva tudi ta odjemalec sporočila s tega kanala.
- Če kanal ne obstaja, strežnik odgovori z NoSuchChannel.
- Odjemalec se ne more priklopiti večkrat v isti kanal (dobi od strežnika AlreadyConnected).
- Če je ime kanala nepravilno, vrne Error.

5. Odklopi se s kanala

Odjemalec se odklopi s kanala tako, da pošlje ukaz Disconnect ime_kanala. Strežnik ga zbriše z svoje liste odjemalcev, ki jih bo ob novih sporočilih v kanal ime_kanala obveščal. Prej mora biti odjemalec strežniku že predstavljen (zgodba 2 - če ni, strežnik vrne NameNotDefined).

- strežnik pošlje vsem priklopljenim (tudi temu, ki se je pravkar odklopil) [ime_kanala] ime_odjemalca disconnected (npr. če odjemalec Jagoda napiše Disconnect PVIS, strežnik vsem priklopljenim na kanal PVIS pošlje [PVIS] Jagoda disconnected). Odjemalec od sedaj naprej ne dobiva več sporočil s tega kanala.
- Če kanal ne obstaja, strežnik odgovori z NoSuchChannel.
- Če odjemalec ni priklopljen v ta kanal, strežnik odgovori NotConnected

6. Vpis sporočila

Ko je odjemalec priklopljen na kanal ime_kanala, lahko vanj piše sporočila in dobiva sporočila tistih uporabnikov, ki so že priklopljeni na isti kanal. To stori z ukazom Say ime_kanala sporočilo. Sporočilo je vse do konca vrstice, lahko vsebuje črke (brez šumnikov), številke, presledke in ločila. Odjemalec mora biti prej predstavljen.

- Če odjemalec ni priklopljen v kanal, mu vrne NotConnected. Če kanal ne obstaja, vrne NoSuchChannel.
- Če je vse OK, strežnik pošlje vsem priklopljenim odjemalcem (tudi temu, ki je sporočilo napisal!):

  [ime_kanala/ime_odjemalca] sporočilo (med "[ime_kanala/ime_odjemalca]" in "sporočilo" je presledek). Npr.:

  - Jagoda napiše Say PVIS Kok tele naloge tezijo..., strežnik pošlje vsem sporočilo [PVIS/Jagoda] Kok tele naloge tezijo...
  - Jagoda napiše *Say RKO Tud tele niso veliko boljse...*, strežnik pošlje vsem sporočilo *[RKO/Jagoda] Tud tele niso veliko boljse...*

7. Izpiši vse kanale

Odjemalec lahko z ukazom Channels dobi izpis vseh kanalov v strežniku. Prej mora biti odjemalec strežniku predstavljen, priklopljen pa mu ni treba biti!

- Če noben kanal še ne obstaja, odgovori z NoChannels.
- Če je vsaj en kanal kreiran, strežnik odgovori z Channels lista_kanalov_ločena_z_vejicami (npr. "Channels kanal1, kanal2, kanal3, kanal4"), pri čemer so imena kanalov urejena po abecedi in ločena z vejicami in presledki za vejicami. Za zadnjim imenom ni vejice in presledka!

8. Izpiši vse uporabnike v kanalu

Odjemalec lahko z ukazom GetUsers ime_kanala dobi izpis vseh uporabnikov, priklopljenih v kanal. Prej mora biti odjemalec predstavljen, priklopljen pa mu ni treba biti!.

- Strežnik odgovori z Users ime_kanala uporabnik1, uporabnik2, uporabnik3, ..., uporabnikN pri čemer so imena urejena naraščajoče po abecedi in ločena z vejicami in presledki za vejicami. Za zadnjim imenom ni vejice in presledka!
- Če je kanal prazen (nihče ni priklopljen), vrne NoUsers.

- Če kanal ne obstaja, vrne NoSuchChannel.

9. Vpis datotek za izmenjavo v skupno bazo

Ko je odjemalec predstavljen (če ne, glej 2. zgodbo), lahko pove strežniku, katere datoteke ima na voljo za izmenjavo. Z ukazom Insert ime_datoteke lahko odjemalec vpiše v skupno bazo datoteko ime_datoteke. Strežnik si zapomni, kdo ima na voljo posamezno datoteko (njegovo ime). Datoteke se lahko podvajajo, vendar ne pri istem odjemalcu (torej lahko imamo več enakopoimenovanih datotek pri različnih imenih odjemalcev). V okviru datotek enega odjemalca so torej imena unikatna.

- Če je v bazi že datoteka s tem imenom od tega odjemalca, strežnik vrne DuplicateName.
- Če je vse OK, v bazo doda datoteko ime_datoteke in vrne odjemalcu Inserted ime_datoteke (npr. Inserted mojadatoteka1).

10. Brisanje datoteke v skupni bazi

Z ukazom Delete ime_datoteke (npr. Delete mojadatoteka) lahko odjemalec zbriše ime datoteke, ki jo ponuja ostalim iz skupne baze.

- Če je v bazi datoteka ne obstaja, strežnik vrne NoSuchFile ime_datoteke.
- Če je odjemalec ni lastnik datoteke, mu vrne NotYourFile
- Če je vse OK (takšena datoteka v bazi obstaja in jo je dodal prav ta odjemalec), strežnik zbriše zapis ime_datoteke iz baze in odjemalca obvesti z Deleted ime_datoteke (npr. Deleted mojadatoteka).

11. Izpiši vse datoteke v skupni bazi

Odjemalec lahko z ukazom GetFiles (pazi: za ukazom ni presledka) dobi izpis vseh datotek, sortiranih naraščajoče po imenu datotek. Če je več datotek z istim imenom, jih podsortira še naraščajoče po imenu odjemalca, ki je datoteko vpisal. Prej mora biti odjemalec, ki ukaz vpiše, predstavljen.

- Strežnik odgovori z Files datoteka1/odjemalec1, datoteka2/odjemalec2,..., datotekaN/odjemalecN pri čemer so pari imen: datoteka/odjemalec urejeni naraščajoče po abecedi in ločeni z vejicami in presledki za vejicami. Za zadnjim ni vejice in presledka!
- Če je baza prazna, vrne NoFiles.

12. Iskanje po vsebini baze po imenu datoteke

Z ukazom GetFiles ime_ali_del_imena (npr GetFiles Jag) dobi odjemalec izpis vseh tistih datotek in njihovih lastnikov (odjemalcev, ki so jih vpisali), ki vsebujejo kjerkoli v imenu datoteke podniz ime_ali_del_imena. Iskanje razlikuje med velikimi in malimi črkami. "Jokerji" v nizu iskani_niz niso dovoljeni. Izpis je urejen po abecedi, naraščajoče.

- Strežnik odgovori z Files datoteka1/odjemalec1, datoteka2/odjemalec2,..., datotekaN/odjemalecN pri čemer so pari imen: datoteka/odjemalec urejeni naraščajoče po abecedi in ločeni z vejicami in presledki za vejicami. Za zadnjim ni vejice in presledka!
- Če temu kriteriju ne zadošča noben zapis v bazi, vrne NoFiles.

13. Iskanje po vsebini baze po lastniku

Z ukazom GetFilesFrom ime_ali_del_imena_latnika (npr GetFilesFrom Jag) dobi odjemalec izpis vseh tistih datotek in njihovih lastnikov (odjemalcev, ki so jih vpisali v bazo), ki vsebujejo kjerkoli v imenu lastnika datoteke podniz ime_ali_del_imena_lastnika. Iskanje razlikuje med velikimi in malimi črkami. "Jokerji" v nizu ime_ali_del_imena_lastnika niso dovoljeni. Izpis je urejen po abecedi, naraščajoče.

- Strežnik odgovori z Files datoteka1/odjemalec1, datoteka2/odjemalec2,..., datotekaN/odjemalecN pri čemer so pari imen: datoteka/odjemalec urejeni naraščajoče po abecedi in ločeni z vejicami in presledki za vejicami. Za zadnjim ni vejice in presledka!
- Če temu kriteriju ne zadošča noben zapis v bazi, vrne NoFiles.

Pomembno:

Zgodbe implementirajte po vrsti, od št. 1 naprej. Šele ko ste končali implementacijo posamezne zgodbe, nadaljujte z naslednjo zgodbo! Pred začetkom implementacije vsake zgodbe z Note napišite komentar "začetek zgodba 1", ko je implementacija končana, napišite "konec zgodba 1". Vedno poganjajte vse teste oz. AllTests.java, ko napišete nov TestCase, pa ne pozabite ažurirati AllTests (še enkrat poženete Wizard :), da bo vključil tudi nove teste - z desno tipko na miški kliknite na AllTests, nato s kontekstnega menija izberete "Recreate test suite"). Struktura ChatService in ostalih vaših razredov je poljubna, tako da si npr. lahko dodajate metode, ki jih potrebujete pri testiranju. Preden začnete, si tudi malo poglejte Java Collections framework (razredi v java.util, ki jih boste uporabljali za implementacijo "baze" (torej za shranjevanje imen datotek, za shranjevanje referenc na delovne niti, ...)). Še namig: Collections framework ima tudi "Sorted" razrede (ki avtomatsko soritrajo člane strukture ob dodajanju), prav tako ima že tudi ovojnice (wrapper) za sinhronizacijo dostopa do kolekcije (primer tega ste že imeli pri Craftsmanu - spomnite se na Collection.synchronizedList(new LinkedList())).

Kadar je karkoli narobe s katerimkoli ukazom (npr. nepravilen format ukaza, nepravilen format imen, presledki tam, kjer jih ne sme biti, itd.), naj strežnik odgovori z "Error", razen če ni zgoraj že predpisan drugačen odgovor. Ukazi so občutljivi na velike in male črke (case-sensitive), prav tako imena in vse ostalo!

Če česa ne veste ali ni jasno definirano, vprašajte! (v splošnem v takem primeru vprašate stranko - tukaj sem seveda to jaz :)) Če bo vprašanje zadevalo vse, bom poslal odgovor kar na našo email listo.

Strežnik (oz. razred ChatService) naj ima obvezno tudi metodo isRunning():

- isRunning() vrne true, če strežnik in strežniške niti še tečejo, in false, ko se strežniške niti (in primarna strežniška nit) končajo.

V primarni strežniški niti (nit, ki kreira ostale strežniške niti, ki poskrbijo za posameznega klienta) tako npr. na začetku dajte privatno spremenljivko running na true, ko se delo strežniških niti konča, jo dajte na false. isRunning() vrača stanje te spremenljivke (in s tem stanje strežnika - torej ali ta še kaj dela ali je končal). Na ta način bom lahko npr. ugotovil, ali se strežnik konča, ko dobi ukaz stop, itd...

Odobritveni testi (acceptance tests)

Odobritveni testi bodo napisani s pomočno paketa SAT. SAT (SocketAcceptanceTester) je program, ki omogoča izvajanje sprejemnih (acceptance) testov programov, ki uporabljajo vtiče. Je nadgradnja programa FIT (http://fit.c2.com). Lokalna kopija sat.jar in fit.jar je v LRI4 v c:\eclipse\lib direktoriju, lahko pa ga dobite tudi na Internetu (download 111K). Vaše teste pišite tako kot do sedaj, z uporabo JUnit-a, in ne morebiti s SAT-om, ki je namenjen za t.i. sprejemne teste (acceptance tests), ki jih bom definiral jaz! Sprejemne teste za seminar boste dobili naslednji teden (skupaj z navodili za uporabo), seveda pa bodo ti testi pregledovali le minimum, še zdaleč ne bodo preigrali vseh možnih scenarijev, zato se samo na njih ne zanašajte, ampak pišite svoje JUnit teste! Na koncu bom jaz testiral vašo kodo s precej večjim naborom testov.

Pisanja strežnika se lotite inkrementalno (podobno kot v seriji Craftsman). Koda naj bo čim bolj robustna.

Sprejemni testi pri eksperimentu s pari

Sledijo sprejemni testi, s katerimi smo zagotavljali **konstantno zunanjo kvaliteto** dela pri eksperimentu s pari. Uporabili smo odprtokodno ogrodje FIT (http://fit.c2.com), katerega smo nadgradili s posebnim modulom za testiranje programov preko vtičev SAT (*Socket Acceptance Tester*, http://fit.c2.com/files/SocketAcceptanceTester/sat/TestSat.html). Ogrodje FIT pridobiva v zadnjem času veljavo de facto standarda za sprejemne teste, podobno kot je to že postalo ogrodje JUnit (http://www.junit.org) za teste enot. Če je test, zapisan v celici tabele uspešen, se celica obarva zeleno. Rdeče so celice tabele tam, kjer se je pojavila napaka in v rumeno obarvanih celicah so označeni testi, kjer se je sprožila izjema, katere opis se vpiše v posamezno celico. Na začetku vsake zgodbe se najprej zažene strežnik (`startServer`), nato pa se s serijo pošiljanj (`send`) in sprejemanj (`rx`) funkcionalno testira koda, lahko tudi za več odjemalcev hkrati (preizkus pravilnosti delovanja večnitnosti in sinhronizacije)[31]. S tem se s podrobnimi testi tipa črne škatle preveri ustrezno funkcionalnost zgodbe na način, kot bi ga delal uporabnik programa.

V ogrodju FIT je običajno, da spremni tekst, ki razlaga teste, napišemo kar v isto datoteko kot same teste (v našem primeru HTML dokument). Primera pognanih sprejemnih testov za oba eksperimenta s pari (PP03 in PP04) sta podana v nadaljevanju**.** Dokumenta sta realen izpis popolnoma uspelih sprejemnih testov.

[31] Obstaja še več ukazov – za celoten spisek glej spletno stran ogrodja SAT oz. ogrodju priloženo dokumentacijo.

Sprejemni testi za eksperiment PP03

Za testiranje potrebujemo razred, ki v svojem konstruktorju **brez parametrov** (oz. privzeti - default konstruktor) **tudi avtomatsko zažene** naš strežnik. Lahko popravite DBService (ne pozabite pri svojih testih potem morebiti spremeniti tudi setUp()), večini bo pa verjetno najlažje narediti nov razred npr. RunDBService, katerega privzeti konstruktor opravi avtomatski zagon strežnika na določenem portu (npr. 1234). Spodaj je (na hitro narejeni) vzorčni razred, prilagodite ga seveda svoji implementaciji!

```
/**
 * Ovoj okoli DBService, implemenira vmesnik sat.Server. Uporablja se za
 * poganjanje sprejemnih testov (acceptance tests).
 *
 * @author matjaz
 */
public class RunDBService implements sat.Server {
        /** port, na katerem posluša */
        private static final int PORT = 1234;
        /** instanca našega strežnika (Service) */
        private DBService dbs;

        /**
         * Privzeti konstruktor, ki avtomatsko zažene DBService na določenem portu.
         * Pri tem uporabi implementacijo strežnika ClientServer (zamenjajte z vašim imenom!)
         */
        public RunDBService() throws Exception {
                dbs = new DBService();
                dbs.serve(PORT, new ClientServer());  //zamenjajte z vašim imenom!
        }

        /** vrne port, na katerem posluša*/
        public int getPortNo() {
                return PORT;
        }

        /** vrne true, če je server aktiven */
        public boolean isRunning() {
                return dbs.isRunning();
        }
}
```

Razred RunDBService.java lahko damo kar v isti paket kot DBService. Ne pozabite k projektu pripeti sat.jar (glej navodila za seminar, na koncu)!

Kako pripraviti podatke:

V **vsaki tabeli morate obvezno popraviti oz. preimenovati** (najbolje da uporabite kar Search-Replace *RunDbService* v vaše ime paketa) v drugi vrstici tabele (ukaz "*startServer*") v na vaše ime paketa (npr *fri.pvis.seminarska.RunDBService*)!

Testov ne spreminjajte, lahko si popravite (zmanjšate ali povečate) pavze ("*delay*"), timeoute ("*timeOut*") in pavze po send ukazih ("*postSendDelay*"), ki so odvisne tudi od hitrosti računalnika, na katerem stvari izvajate (počasnejši kot je, večja mora biti pavza).

"timeOut" nastavite na neko veliko vrednost (npr. par sekund), ne pa na 0 (čaka v neskončnost), da stvari ne zablokirajo, če jih niste prav implementirali, ampak vam bodo testi vseeno padli (po timeOut-u).

Vsi testi **lahko trajajo kar dolgo** (par minut - odvisno predvsem od tega, kako dolga je pavza med posameznimi ukazi), zato potrpljenje! Lahko si tudi razbijete tole datoteko na manjše dele ali pa določene dele zakomentirate (s HTML komentarjem).

Kako pognati vso zadevo:

- potrebujete (popravljeno na zgoraj opisan način) TestDBService.htm (ravnokar ga berete :)), sat.jar, fit.jar (imate ju tudi lokalno na mrežnem disku \\lrk-vaje.fri.uni-lj.si\install ali c:\eclipse\lib v LRI4 in LRK) in vašo implementacijo seminarja (*.class datoteke).
- naredite direktorij (npr. c:\sat), v katerega skopirate vse razen vaše implementacije (ki naj ostane kar v workspace-u).
- premaknite se v ta direktorij
- naredite datoteko RunAcceptanceTests.bat, v kateri naj bo (**v eni vrstici!**):

```
c:\pot_do_jave\java -classpath
.;fit.jar;sat.jar;C:\pot_do_vasega_workspacea\workspace\ime_projekta
fit.FileRunner TestDBService.htm AcceptanceTestResults.htm
```

pri čemer so datoteke vaše implementacije v direktoriju `C:\pot_do_vasega_workspacea\workspace\ime_projekta` (tu so torej direktoriji, ki vsebujejo vaše .class datoteke).

- poženete iz ukazne vrstice *RunAcceptanceTests.bat*, ki vzame TestDBService.htm, ga predela in izpiše rezultate v novo datoteko **AcceptanceTestResults.htm** (v istem direktoriju kot *.bat* datoteka).
- Odprite AcceptanceTestResults v brskalniku in poglejte tabele. Zeleno označi celice, kjer je vse ok, rdeče tam kjer je napaka in rumeno tam, kjer se je sprožila izjema, ki jo vpiše v posamezno celico.

Kako popravljati padle teste:

- Ko vam test pade, vedno najprej popravite kodo za najbolj zgodnji padli test (najvišje v tabeli), ker so dogodki v tabeli med seboj velikokrat "vezani" - to je odvisni eden od drugega.

Če mislite, da je s katerimkoli testom kaj narobe (da po vaše napačno testira), mi to prosim takoj sporočite!

1. Ugasni DBService

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Stop			
rx	Stopping			
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
comment	strežnik ugasnjen; naslednji odjemalci se ne morejo več priklopiti			
rxRefused		Stopping	Stopping	Stopping
sendRefused		Stop	Stop	Stop
rxRefused		Stopping	Stopping	Stopping
stopServer				

2. Predstavitev odjemalca strežniku DBService

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name		Name .
rx	Named L1	NameInvalid		NameInvalid
send	Name L1	Name L2	Name L1	Name L1!
rx	NameAlreadyDefined	Named L2	NameTaken	NameInvalid
send	Name L10	Name L3	Name	Name L2
rx	NameAlreadyDefined	NameAlreadyDefined	NameInvalid	NameTaken
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

3. Kreiraj bazo

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Create baza1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Create baza1
rx	Created baza1
comment	ze obstojeca baza, nepravilna imena baze
send	Create baza1
rx	CreateFailed
send	Create
rx	Error
send	Create baza.1!
rx	Error
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	Created baza1	Created baza2	Created baza3	Created baza4
comment	ze obstojeca baza, nepravilna imena baze			
send	Create baza1			
rx	CreateFailed			
send		Create		Create
rx		Error		Error
send			Create baza.1!	
rx			Error	
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

4. Priklopi se na bazo

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Connect baza1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Connect baza1
rx	NoSuchDatabase
send	Create baza1
rx	Created baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Connect baza2
rx	NoSuchDatabase
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Connect baza1	Connect baza2	Connect baza3	Connect baza4
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	Created baza1	Created baza2	Created baza3	Created baza4
send	Connect baza1	Connect baza2	Connect baza3	Connect baza4
rx	ConnectedOn baza1	ConnectedOn baza2	ConnectedOn baza3	ConnectedOn baza4
send	Connect baza1	Connect baza1	Connect baza1	Connect baza1
rx	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1
send	Connect baza1234	Connect baza1234	Connect baza4321	
rx	NoSuchDatabase	NoSuchDatabase	NoSuchDatabase	
send		Connect baza1234		Connect baza4
rx		NoSuchDatabase		ConnectedOn baza4
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

5. Vpis v bazo

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Insert baza1 1/krneki1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Insert baza1 1/krneki1
rx	NotConnected baza1
send	Create baza1
rx	Created baza1
send	Insert baza1 1/krneki1
rx	NotConnected baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Insert baza1 1/krneki1
rx	Inserted baza1 1/krneki1
send	Insert baza1 1/krneki1
rx	DuplicateID baza1 1/krneki1
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	Created baza1	Created baza2	Created baza3	Created baza4
send	Connect baza1	Connect baza2	Connect baza3	Connect baza4
rx	ConnectedOn baza1	ConnectedOn baza2	ConnectedOn baza3	ConnectedOn baza4
send	Insert baza1 1/krneki1	Insert baza2 1/krneki1	Insert baza3 1/krneki1	Insert baza4 1/krneki1
rx	Inserted baza1 1/krneki1	Inserted baza2 1/krneki1	Inserted baza3 1/krneki1	Inserted baza4 1/krneki1
send	Insert baza1 1/krneki1	Insert baza2 1/krneki1	Insert baza3 1/krneki1	Insert baza4 1/krneki1
rx	DuplicateID baza1 1/krneki1	DuplicateID baza2 1/krneki1	DuplicateID baza3 1/krneki1	DuplicateID baza4 1/krneki1
send		Connect baza1	Connect baza1	
rx		ConnectedOn baza1	ConnectedOn baza1	
send	Insert baza1 2/krneki2			
rx	Inserted baza1 2/krneki2	Inserted baza1 2/krneki2	Inserted baza1 2/krneki2	
send		Insert baza1 3/krneki3		
rx	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping

comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

6. Izpisi vse baze

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Select *
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Select *
rx	NoDatabases
send	Create bbaza
rx	Created bbaza
send	Select *
rx	Databases bbaza
send	Create abaza
rx	Created abaza
send	Select *
rx	Databases abaza, bbaza
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Select *	Select *	Select *	Select *
rx	NoDatabases	NoDatabases	NoDatabases	NoDatabases
send	Create dbaza	Create cbaza	Create abaza	Create bbaza
rx	Created dbaza	Created cbaza	Created abaza	Created bbaza
send	Select *			
rx	Databases abaza, bbaza, cbaza, dbaza			
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

7. Izpiši vse zapise v posamezni bazi

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Select * from baza1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Create baza1
rx	Created baza1
send	Select * from baza1
rx	NotConnected baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Select * from baza1
rx	NoSuchRecord baza1
send	Insert baza1 2/akrneki
rx	Inserted baza1 2/akrneki
send	Insert baza1 1/bkrneki
rx	Inserted baza1 1/bkrneki
send	Select * from baza1
rx	Database baza1 1/bkrneki, 2/akrneki
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1			
rx	Created baza1			
send	Connect baza1	Connect baza1	Connect baza1	Connect baza1
rx	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1
send	Select * from baza1	Select * from baza1	Select * from baza1	Select * from baza1
rx	NoSuchRecord baza1	NoSuchRecord baza1	NoSuchRecord baza1	NoSuchRecord baza1
send	Insert baza1 3/krneki3			
rx	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3
send	Insert baza1 2/krneki2			
rx	Inserted baza1	Inserted baza1 2/krneki2	Inserted baza1	Inserted baza1

	2/krneki2		2/krneki2	2/krneki2
send	Insert baza1 1/krneki1			
rx	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1
send		Select * from baza1		
rx		Database baza1 1/krneki1, 2/krneki2, 3/krneki3		
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

8. Sprememba zapisa v bazi

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Update baza1 1/krneki1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Update baza1 1/krneki1
rx	NotConnected baza1
send	Create baza1
rx	Created baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Update baza1 1/krneki1
rx	NoSuchID baza1 1/krneki1
send	Insert baza1 1/krneki1
rx	Inserted baza1 1/krneki1
send	Update baza1 1/krneki2
rx	Updated baza1 1/krneki2
send	Select * from baza1
rx	Database baza1 1/krneki2
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100

send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	Created baza1	Created baza2	Created baza3	Created baza4
send	Connect baza1	Connect baza2	Connect baza3	Connect baza4
rx	ConnectedOn baza1	ConnectedOn baza2	ConnectedOn baza3	ConnectedOn baza4
send	Update baza1 1/krneki1	Update baza2 1/krneki1	Update baza3 1/krneki1	Update baza4 1/krneki1
rx	NoSuchID baza1 1/krneki1	NoSuchID baza2 1/krneki1	NoSuchID baza3 1/krneki1	NoSuchID baza4 1/krneki1
send	Insert baza1 1/krneki1	Insert baza2 1/krneki1	Insert baza3 1/krneki1	Insert baza4 1/krneki1
rx	Inserted baza1 1/krneki1	Inserted baza2 1/krneki1	Inserted baza3 1/krneki1	Inserted baza4 1/krneki1
send	Update baza1 1/krneki2	Update baza2 1/krneki2	Update baza3 1/krneki2	Update baza4 1/krneki2
rx	Updated baza1 1/krneki2	Updated baza2 1/krneki2	Updated baza3 1/krneki2	Updated baza4 1/krneki2
send		Connect baza1	Connect baza1	
rx		ConnectedOn baza1	ConnectedOn baza1	
send	Update baza1 1/krneki3			
rx	Updated baza1 1/krneki3	Updated baza1 1/krneki3	Updated baza1 1/krneki3	
send		Update baza1 1/krneki4		
rx	Updated baza1 1/krneki4	Updated baza1 1/krneki4	Updated baza1 1/krneki4	
send			Select * from baza1	
rx			Database baza1 1/krneki4	
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

9. Brisanje zapisa v bazi

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Delete baza1 1
rx	NameNotDefined
send	Name lokacija1
rx	Named lokacija1
send	Delete baza1 1
rx	NotConnected baza1
send	Create baza1
rx	Created baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Delete baza1 1
rx	NoSuchID baza1 1
send	Insert baza1 1/krneki1
rx	Inserted baza1 1/krneki1
send	Insert baza1 2/krneki2
rx	Inserted baza1 2/krneki2
send	Delete baza1 1
rx	Deleted baza1 1/krneki1
send	Select * from baza1

rx	Database baza1 2/krneki2
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1	Create baza2	Create baza3	Create baza4
rx	Created baza1	Created baza2	Created baza3	Created baza4
send	Connect baza1	Connect baza2	Connect baza3	Connect baza4
rx	ConnectedOn baza1	ConnectedOn baza2	ConnectedOn baza3	ConnectedOn baza4
send	Delete baza1 1	Delete baza2 1	Delete baza3 1	Delete baza4 1
rx	NoSuchID baza1 1	NoSuchID baza2 1	NoSuchID baza3 1	NoSuchID baza4 1
send	Insert baza1 1/krneki1	Insert baza2 1/krneki1	Insert baza3 1/krneki1	Insert baza4 1/krneki1
rx	Inserted baza1 1/krneki1	Inserted baza2 1/krneki1	Inserted baza3 1/krneki1	Inserted baza4 1/krneki1
send	Insert baza1 2/krneki2	Insert baza2 2/krneki2	Insert baza3 2/krneki2	Insert baza4 2/krneki2
rx	Inserted baza1 2/krneki2	Inserted baza2 2/krneki2	Inserted baza3 2/krneki2	Inserted baza4 2/krneki2
send	Delete baza1 1	Delete baza2 1	Delete baza3 1	Delete baza4 1
rx	Deleted baza1 1/krneki1	Deleted baza2 1/krneki1	Deleted baza3 1/krneki1	Deleted baza4 1/krneki1
send	Insert baza1 3/krneki3			
rx	Inserted baza1 3/krneki3			
send			Connect baza1	Connect baza1
rx			ConnectedOn baza1	ConnectedOn baza1
send				Delete baza1 2
rx	Deleted baza1 2/krneki2		Deleted baza1 2/krneki2	Deleted baza1 2/krneki2
send			Select * from baza1	
rx			Database baza1 3/krneki3	
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

10. Iskanje po vsebini zapisov

en odjemalec

sat.fit.SatFixture	
startServer	RunDBService
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Select kr from baza1
rx	NameNotDefined

send	Name lokacija1
rx	Named lokacija1
send	Select kr from baza1
rx	NotConnected baza1
send	Create baza1
rx	Created baza1
send	Connect baza1
rx	ConnectedOn baza1
send	Select krneki from baza1
rx	NoSuchRecord baza1
send	Insert baza1 2/abcdef
rx	Inserted baza1 2/abcdef
send	Insert baza1 1/defghi
rx	Inserted baza1 1/defghi
send	Insert baza1 3/jklabc
rx	Inserted baza1 3/jklabc
send	Select ab from baza1
rx	Database baza1 2/abcdef, 3/jklabc
send	Select efg from baza1
rx	Database baza1 1/defghi
send	Select def from baza1
rx	Database baza1 1/defghi, 2/abcdef
send	Select krneki from baza1
rx	NoSuchRecord baza1
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	RunDBService			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create baza1			
rx	Created baza1			
send	Connect baza1	Connect baza1	Connect baza1	Connect baza1
rx	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1	ConnectedOn baza1
send	Insert baza1 3/krneki3			
rx	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3	Inserted baza1 3/krneki3
send	Insert baza1 2/krneki2			
rx	Inserted baza1 2/krneki2	Inserted baza1 2/krneki2	Inserted baza1 2/krneki2	Inserted baza1 2/krneki2
send	Insert baza1 1/krneki1			
rx	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1	Inserted baza1 1/krneki1
send	Select k from baza1	Select krneki from baza1	Select krneki from baza1	Select k from baza1
rx	Database baza1 1/krneki1, 2/krneki2, 3/krneki3	Database baza1 1/krneki1, 2/krneki2, 3/krneki3	Database baza1 1/krneki1, 2/krneki2, 3/krneki3	Database baza1 1/krneki1, 2/krneki2, 3/krneki3
send	Select 3 from baza1	Select krneki from baza1	Select 3 from baza1	Select 3 from baza1
rx	Database baza1 3/krneki3	Database baza1 1/krneki1,	Database baza1 3/krneki3	Database baza1 3/krneki3

		2/krneki2, 3/krneki3		
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

Povzetek

fit.Summary	
counts	702 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\pvis\sat\TestDBService.htm
input update	Mon Jun 02 17:05:58 CEST 2003
output file	C:\pvis\sat\AcceptanceTestResults.htm
run date	Wed Jun 04 16:37:20 CEST 2003

Sprejemni testi za eksperiment PP04

Za testiranje potrebujemo razred, ki v svojem konstruktorju **brez parametrov** (oz. privzeti - default konstruktor) **tudi avtomatsko zažene** naš strežnik ter implementira vmesnik `sat.Server`. Razred ChatServiceSatWrapper je ovoj okoli vašega strežnika, katerega privzeti konstruktor opravi avtomatski zagon strežnika na določenem portu (npr. 1234). Spodaj je (na hitro narejeni) vzorčni razred, prilagodite ga seveda svoji implementaciji!

```
/**
 * Ovoj okoli ChatService, implemenira vmesnik sat.Server. Uporablja se za
 * poganjanje sprejemnih testov (acceptance tests).
 *
 * @author matjaz
 */
public class ChatServiceSatWrapper implements sat.Server {
        /** port, na katerem posluša */
        private static final int PORT = 1234;
        /** instanca našege storitve (Service) */
        private ChatService cs;

        /**
         * Privzeti konstruktor, ki avtomatsko zažene ChatService na določenem portu.
         * Pri tem uporabi implementacijo strežnika ClientServer (zamenjajte z vašim imenom!)
         */
        public ChatServiceSatWrapper() throws Exception {
                cs = new ChatService();
                cs.serve(PORT, new ClientServer());  //zamenjajte z vašim imenom!
        }

        /** vrne port, na katerem posluša*/
        public int getPortNo() {
                return PORT;
        }

        /** vrne true, če je server aktiven */
        public boolean isRunning() {
                return cs.isRunning();
        }
}
```

Razred ChatServiceSatWrapper.java lahko damo kar v isti paket kot ChatService. Ne pozabite k projektu v classpath pripeti sat.jar (podobno kot pripnete junit.jar)!

Kako pripraviti podatke:

V **vsaki tabeli morate obvezno popraviti oz. preimenovati** (najbolje da uporabite kar Search-Replace) *koda.ChatServiceSatWrapper* v vaše ime paketa v drugi vrstici tabele (ukaz "*startServer*") npr. *fri.pvis.seminarska.ChatServiceSatWrapper*)!

Testov ne spreminjajte, lahko si popravite (zmanjšate ali povečate) pavze ("*delay*"), timeoute ("*timeOut*") in pavze po send ukazih ("*postSendDelay*"), ki so odvisne tudi od hitrosti računalnika, na katerem stvari izvajate (počasnejši kot je, večja mora biti pavza).

"timeOut" nastavite na neko veliko vrednost (npr. par sekund), ne pa na 0 (čaka v neskončnost), da stvari ne zablokirajo, če jih niste prav implementirali, ampak vam bodo testi vseeno padli (po timeOut-u).

Vsi testi **lahko trajajo kar dolgo** (par minut - odvisno predvsem od tega, kako dolga je pavza med posameznimi ukazi), zato potrpljenje! Lahko si tudi razbijete tole datoteko na manjše dele ali pa določene dele zakomentirate (s HTML komentarjem).

Kako pognati vso zadevo:

- potrebujete (popravljeno na zgoraj opisan način) ChatServiceAcceptanceTest.htm (ravnokar ga berete :)), sat.jar, fit.jar (imate ju tudi lokalno na c:\eclipse\lib v LRI4) in vašo implementacijo seminarja (*.class datoteke).
- naredite direktorij (npr. c:\sat), v katerega skopirate vse razen vaše implementacije (ki naj ostane kar v workspace-u).
- premaknite se v ta direktorij
- naredite datoteko RunAcceptanceTests.bat, v kateri naj bo (**v eni vrstici!**):

```
c:\pot_do_jave\java -classpath
.;fit.jar;sat.jar;C:\pot_do_vasega_workspacea\workspace\ime_projekta
fit.FileRunner ChatServiceAcceptanceTest.htm AcceptanceTestResults.htm
```

pri čemer so datoteke vaše implementacije v direktoriju `C:\pot_do_vasega_workspacea\workspace\ime_projekta` (tu so torej direktoriji, ki vsebujejo vaše .class datoteke).

- poženete iz ukazne vrstice *RunAcceptanceTests.bat*, ki vzame ChatServiceAcceptanceTest.htm, ga predela in izpiše rezultate v novo datoteko **AcceptanceTestResults.htm** (v istem direktoriju kot *.bat* datoteka)**.**
- Odprite AcceptanceTestResults v brskalniku in poglejte tabele. Zeleno označi celice, kjer je vse OK, rdeče tam kjer je napaka in rumeno tam, kjer se je sprožila izjema, ki jo vpiše v posamezno celico.

Kako popravljati padle teste:

- Ko vam test pade, vedno najprej popravite kodo za najbolj zgodnji padli test (najvišje v tabeli), ker so dogodki v tabeli med seboj velikokrat "vezani" - to je odvisni eden od drugega.

Če mislite, da je s katerimkoli testom kaj narobe (da po vaše napačno testira), mi to prosim takoj sporočite!

1. Ugasni ChatService

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Stop			
rx	Stopping			
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
comment	strežnik ugasnjen; naslednji odjemalci se ne morejo več priklopiti			
rxRefused		Stopping	Stopping	Stopping
sendRefused		Stop	Stop	Stop
rxRefused		Stopping	Stopping	Stopping
stopServer				

2. Predstavitev odjemalca strežniku ChatService

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name		Name .
rx	Named L1	Error		Error
send	Name L1	Name L2	Name L1	Name L1_
rx	NameAlreadyDefined	Named L2	NameTaken	Error
send	Name L10	Name L3	Name	Name L2
rx	NameAlreadyDefined	NameAlreadyDefined	Error	NameTaken
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

3. Kreiraj kanal (chat channel)

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Create kanall1
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	Create kanal1
rx	Created kanal1
comment	ze obstojec kanal, nepravilna imena kanalov
send	Create kanal1
rx	ChannelAlreadyCreated
send	Create
rx	Error
send	Create kanall.1!
rx	Error
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	Created kanal1	Created kanal2	Created kanal3	Created kanal4
comment	ze obstojeca baza, nepravilna imena baze			
send	Create kanal1	Create kanal3	Create kanal2	Create kanal1
rx	ChannelAlreadyCreated	ChannelAlreadyCreated	ChannelAlreadyCreated	ChannelAlreadyCreated
send		Create		Create
rx		Error		Error
send			Create kanal.1!	
rx			Error	
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

4. Priklopi se na kanal

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Connect kanal_1
rx	Error
send	Connect kanal1
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	Connect kanal1
rx	NoSuchChannel
send	Create kanal1
rx	Created kanal1
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	Connect kanal2
rx	NoSuchChannel
send	Connect kanal1
rx	AlreadyConnected
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Connect kanal1	Connect kanal2	Connect kanal3	Connect kanal4
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	Created kanal1	Created kanal2	Created kanal3	Created kanal4
send	Connect kanal1	Connect kanal2	Connect kanal3	Connect kanal4
rx	[kanal1] L1 connected	[kanal2] L2 connected	[kanal3] L3 connected	[kanal4] L4 connected
send		Connect kanal1	Connect kanal3	Connect kanal4
rx	[kanal1] L2 connected	[kanal1] L2 connected	AlreadyConnected	AlreadyConnected
send			Connect kanal1	
rx	[kanal1] L3 connected	[kanal1] L3 connected	[kanal1] L3 connected	
send		Connect kanal_1		Connect kanal5
rx		Error		NoSuchChannel
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

5. Odklopi se s kanala

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Disconnect kanal_1
rx	Error
send	Disconnect kanal1
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	Disconnect kanal1
rx	NoSuchChannel
send	Create kanal1
rx	Created kanal1
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	Disconnect kanal1
rx	[kanal1] Jagoda disconnected
send	Disconnect kanal1
rx	NotConnected
send	Disconnect kanal3
rx	NoSuchChannel
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Disconnect kanal1	Disconnect kanal2	Disconnect kanal3	Disconnect kanal4
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	Created kanal1	Created kanal2	Created kanal3	Created kanal4
send	Connect kanal1	Connect kanal2	Connect kanal3	Connect kanal4
rx	[kanal1] L1 connected	[kanal2] L2 connected	[kanal3] L3 connected	[kanal4] L4 connected
send		Connect kanal1	Disconnect kanal3	Disconnect kanal4
rx	[kanal1] L2 connected	[kanal1] L2 connected	[kanal3] L3 disconnected	[kanal4] L4 disconnected
send			Connect kanal1	

rx	[kanal1] L3 connected	[kanal1] L3 connected	[kanal1] L3 connected	
send	Disconnect kanal1			Disconnect kanal5
rx	[kanal1] L1 disconnected	[kanal1] L1 disconnected	[kanal1] L1 disconnected	NoSuchChannel
send	Disconnect kanal1	Disconnect kanal1		Disconnect kanal4
rx	NotConnected	[kanal1] L2 disconnected	[kanal1] L2 disconnected	NotConnected
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

6. Vpis sporočila

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Say kanal_1 krneki!
rx	Error
send	Say kanal1 krneki!
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	Say kanal1 krneki!
rx	NoSuchChannel
send	Create kanal1
rx	Created kanal1
send	Say kanal1 krneki!
rx	NotConnected
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	Say kanal1 krneki!
rx	[kanal1/Jagoda] krneki!
send	Say kanal1 krneki in krneki2.
rx	[kanal1/Jagoda] krneki in krneki2.
send	Say kanal1 stop
rx	[kanal1/Jagoda] stop
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100

send	Say kanal1 krneki!	Say kanal2 krneki!	Say kanal3 krneki!	Say kanal4 krneki!
rx	NameNotDefined	NameNotDefined	NameNotDefined	NameNotDefined
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	Created kanal1	Created kanal2	Created kanal3	Created kanal4
send	Connect kanal1	Connect kanal2	Connect kanal3	Connect kanal4
rx	[kanal1] L1 connected	[kanal2] L2 connected	[kanal3] L3 connected	[kanal4] L4 connected
send		Connect kanal1	Say kanal3 A, b in c.	Say kanal3 Krneki!
rx	[kanal1] L2 connected	[kanal1] L2 connected	[kanal3/L3] A, b in c.	NotConnected
send			Connect kanal1	Say kanal4 A, b in c.
rx	[kanal1] L3 connected	[kanal1] L3 connected	[kanal1] L3 connected	[kanal4/L4] A, b in c.
send	Say kanal1 Ojla!			Say kanal1 Jaz tudi!
rx	[kanal1/L1] Ojla!	[kanal1/L1] Ojla!	[kanal1/L1] Ojla!	NotConnected
send		Say kanal1 Ojla!		Say kanal1 Halooo?
rx	[kanal1/L2] Ojla!	[kanal1/L2] Ojla!	[kanal1/L2] Ojla!	NotConnected
send		Disconnect kanal1		Say kanal1 Kje ste banda?
rx	[kanal1] L2 disconnected	[kanal1] L2 disconnected	[kanal1] L2 disconnected	NotConnected
send		Say kanal1 Ojla!	Say kanal1 L2 je sel :(	Say kanal5 Hojla!
rx	[kanal1/L3] L2 je sel :(	NotConnected	[kanal1/L3] L2 je sel :(	NoSuchChannel
send			Disconnect kanal1	Say kanal1 Kje ste banda?
rx	[kanal1] L3 disconnected		[kanal1] L3 disconnected	NotConnected
send	Say kanal1 L3?	Say kanal2 Ojla!	Say kanal1 L1?	Say kanal4 Se bom pa s sabo pogovarjal...
rx	[kanal1/L1] L3?	[kanal2/L2] Ojla!	NotConnected	[kanal4/L4] Se bom pa s sabo pogovarjal...
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

7. Izpiši vse kanale

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Channels
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	Channels
rx	NoChannels
send	Create kanal1
rx	Created kanal1
send	Channels
rx	Channels kanal1
send	Channels krneki
rx	Error
send	Create kanal2

rx	Created kanal2
send	Channels
rx	Channels kanal1, kanal2
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	Channels
rx	Channels kanal1, kanal2
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Channels		Channels	
rx	NoChannels		NoChannels	
send	Create kanal1			
rx	Created kanal1			
send	Channels	Channels	Channels	Channels
rx	Channels kanal1	Channels kanal1	Channels kanal1	Channels kanal1
send	Create akanal	Create bkanal	Create ckanal	Create dkanal
rx	Created akanal	Created bkanal	Created ckanal	Created dkanal
send	Channels	Channels	Channels	Channels
rx	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1
send	Connect kanal1	Connect akanal	Connect bkanal	Connect ckanal
rx	[kanal1] L1 connected	[akanal] L2 connected	[bkanal] L3 connected	[ckanal] L4 connected
send	Channels	Channels	Channels	Channels
rx	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1	Channels akanal, bkanal, ckanal, dkanal, kanal1
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

8. Izpiši vse uporabnike v kanalu

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000

comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	GetUsers kanal_1
rx	Error
send	GetUsers kanal1
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	GetUsers kanal1
rx	NoSuchChannel
send	Create kanal1
rx	Created kanal1
send	GetUsers kanal1
rx	NoUsers
send	Connect kanal1
rx	[kanal1] Jagoda connected
send	GetUsers kanal1
rx	Users kanal1 Jagoda
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	GetUsers kanal1		GetUsers kanal3	
rx	NoSuchChannel		NoSuchChannel	
send	Create kanal1	Create kanal2	Create kanal3	Create kanal4
rx	Created kanal1	Created kanal2	Created kanal3	Created kanal4
send	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1
rx	NoUsers	NoUsers	NoUsers	NoUsers
send	Connect kanal1	Connect kanal2	Connect kanal3	Connect kanal4
rx	[kanal1] L1 connected	[kanal2] L2 connected	[kanal3] L3 connected	[kanal4] L4 connected
send	GetUsers kanal1	GetUsers kanal2	GetUsers kanal1	GetUsers kanal2
rx	Users kanal1 L1	Users kanal2 L2	Users kanal1 L1	Users kanal2 L2
send		Connect kanal1	Disconnect kanal3	Disconnect kanal4
rx	[kanal1] L2 connected	[kanal1] L2 connected	[kanal3] L3 disconnected	[kanal4] L4 disconnected
send			Connect kanal1	GetUsers kanal3
rx	[kanal1] L3 connected	[kanal1] L3 connected	[kanal1] L3 connected	NoUsers
send	GetUsers kanal1	GetUsers kanal4	GetUsers kanal1	GetUsers kanal2
rx	Users kanal1 L1, L2, L3	NoUsers	Users kanal1 L1, L2, L3	Users kanal2 L2
send	Disconnect kanal1			
rx	[kanal1] L1 disconnected	[kanal1] L1 disconnected	[kanal1] L1 disconnected	

send	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1
rx	Users kanal1 L2, L3	Users kanal1 L2, L3	Users kanal1 L2, L3	Users kanal1 L2, L3
send		Disconnect kanal1		
rx		[kanal1] L2 disconnected	[kanal1] L2 disconnected	
send	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1	GetUsers kanal1
rx	Users kanal1 L3	Users kanal1 L3	Users kanal1 L3	Users kanal1 L3
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

9. Vpis datotek za izmenjavo v skupno bazo

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Insert moj.filet
rx	NameNotDefined
send	Insert moj.filet!
rx	Error
send	Name Jagoda
rx	Named Jagoda
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	Insert moj-filet_1.0.filet
rx	DuplicateName
send	Insert a-riba.exe
rx	Inserted a-riba.exe
send	Insert moj-filet_1.0.filet
rx	DuplicateName
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Insert file1	Insert file1	Insert file1	Insert file1
rx	Inserted file1	Inserted file1	Inserted file1	Inserted file1

send	Insert file1	Insert file1	Insert file1	Insert file1
rx	DuplicateName	DuplicateName	DuplicateName	DuplicateName
send	Insert virus.exe	Insert virus.exe	Insert virus.exe	Insert virus.exe
rx	Inserted virus.exe	Inserted virus.exe	Inserted virus.exe	Inserted virus.exe
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

10. Brisanje datotek v skupni bazi

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	Delete moj.filet
rx	NameNotDefined
send	Delete moj.filet!
rx	Error
send	Name Jagoda
rx	Named Jagoda
send	Delete moj-filet_1.0.filet
rx	NoSuchFile moj-filet_1.0.filet
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	Delete moj-filet_1.0.filet
rx	Deleted moj-filet_1.0.filet
send	Delete moj-filet_1.0.filet
rx	NoSuchFile moj-filet_1.0.filet
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	Delete file1	Delete file1	Delete file1	Delete file1
rx	NoSuchFile file1	NoSuchFile file1	NoSuchFile file1	NoSuchFile file1
send	Insert file1	Insert file1	Insert file1	Insert file1

rx	Inserted file1	Inserted file1	Inserted file1	Inserted file1
send	Delete file1	Delete file1	Delete file1	
rx	Deleted file1	Deleted file1	Deleted file1	
send	Delete file1	Delete file1	Delete file1	
rx	NotYourFile	NotYourFile	NotYourFile	
send				Delete file1
rx				Deleted file1
send	Delete file1	Delete file1	Delete file1	Delete file1
rx	NoSuchFile file1	NoSuchFile file1	NoSuchFile file1	NoSuchFile file1
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

11. Izpiši vse datoteke v skupni bazi

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	GetFiles
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	GetFiles
rx	NoFiles
send	Insert moj-filet_1.1.filet
rx	Inserted moj-filet_1.1.filet
send	GetFiles
rx	Files moj-filet_1.1.filet/Jagoda
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	GetFiles
rx	Files moj-filet_1.0.filet/Jagoda, moj-filet_1.1.filet/Jagoda
send	Delete moj-filet_1.0.filet
rx	Deleted moj-filet_1.0.filet
send	GetFiles
rx	Files moj-filet_1.1.filet/Jagoda
send	Delete moj-filet_1.1.filet
rx	Deleted moj-filet_1.1.filet
send	GetFiles
rx	NoFiles
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	GetFiles	GetFiles	GetFiles	GetFiles
rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert file1	Insert file1	Insert file1	Insert file1
rx	Inserted file1	Inserted file1	Inserted file1	Inserted file1
send	GetFiles	GetFiles	GetFiles	GetFiles
rx	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4
send	Delete file1	Delete file1	Delete file1	
rx	Deleted file1	Deleted file1	Deleted file1	
send	GetFiles	GetFiles	GetFiles	GetFiles
rx	Files file1/L4	Files file1/L4	Files file1/L4	Files file1/L4
send				Delete file1
rx				Deleted file1
send	GetFiles	GetFiles	GetFiles	GetFiles
rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert bfile1	Insert cfile1	Insert dfile1	Insert afile1
rx	Inserted bfile1	Inserted cfile1	Inserted dfile1	Inserted afile1
send	GetFiles	GetFiles	GetFiles	GetFiles
rx	Files afile1/L4, bfile1/L1, cfile1/L2, dfile1/L3	Files afile1/L4, bfile1/L1, cfile1/L2, dfile1/L3	Files afile1/L4, bfile1/L1, cfile1/L2, dfile1/L3	Files afile1/L4, bfile1/L1, cfile1/L2, dfile1/L3
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

12. Iskanje po vsebini baze po imenu datoteke

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	GetFiles asd!
rx	Error
send	GetFiles fil
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	GetFiles fil
rx	NoFiles
send	Insert moj-filet_1.1.filet
rx	Inserted moj-filet_1.1.filet
send	GetFiles fil

rx	Files moj-filet_1.1.filet/Jagoda
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	GetFiles fil
rx	Files moj-filet_1.0.filet/Jagoda, moj-filet_1.1.filet/Jagoda
send	Delete moj-filet_1.0.filet
rx	Deleted moj-filet_1.0.filet
send	GetFiles fil
rx	Files moj-filet_1.1.filet/Jagoda
send	GetFiles gil
rx	NoFiles
send	Delete moj-filet_1.1.filet
rx	Deleted moj-filet_1.1.filet
send	GetFiles fil
rx	NoFiles
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name L1	Name L2	Name L3	Name L4
rx	Named L1	Named L2	Named L3	Named L4
send	GetFiles e1	GetFiles e1	GetFiles e1	GetFiles e1
rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert file1	Insert file1	Insert file1	Insert file1
rx	Inserted file1	Inserted file1	Inserted file1	Inserted file1
send	GetFiles e1	GetFiles e	GetFiles 1	GetFiles f
rx	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4	Files file1/L1, file1/L2, file1/L3, file1/L4
send	Delete file1	Delete file1	Delete file1	
rx	Deleted file1	Deleted file1	Deleted file1	
send	GetFiles 1	GetFiles f	GetFiles i	GetFiles e
rx	Files file1/L4	Files file1/L4	Files file1/L4	Files file1/L4
send				Delete file1
rx				Deleted file1
send	GetFiles f	GetFiles f	GetFiles f	GetFiles f
rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert bfile1	Insert cfile1	Insert dfile1	Insert afile1
rx	Inserted bfile1	Inserted cfile1	Inserted dfile1	Inserted afile1
send	GetFiles a	GetFiles b	GetFiles c	GetFiles 1
rx	Files afile1/L4	Files bfile1/L1	Files cfile1/L2	Files afile1/L4, bfile1/L1, cfile1/L2, dfile1/L3
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			

stopServer

13. Iskanje po vsebini baze po imenu lastnika

en odjemalec

sat.fit.SatFixture	
startServer	koda.ChatServiceSatWrapper
comment	nastavimo odjemalski timeout v ms
timeOut	2000
comment	nastavimo zakasnitev v ms po vsakem "send"
postSendDelay	100
send	GetFilesFrom asd!
rx	Error
send	GetFilesFrom fil
rx	NameNotDefined
send	Name Jagoda
rx	Named Jagoda
send	GetFilesFrom Jag
rx	NoFiles
send	Insert moj-filet_1.1.filet
rx	Inserted moj-filet_1.1.filet
send	GetFilesFrom Jag
rx	Files moj-filet_1.1.filet/Jagoda
send	Insert moj-filet_1.0.filet
rx	Inserted moj-filet_1.0.filet
send	GetFilesFrom god
rx	Files moj-filet_1.0.filet/Jagoda, moj-filet_1.1.filet/Jagoda
send	Delete moj-filet_1.0.filet
rx	Deleted moj-filet_1.0.filet
send	GetFilesFrom da
rx	Files moj-filet_1.1.filet/Jagoda
send	GetFilesFrom dada
rx	NoFiles
send	Delete moj-filet_1.1.filet
rx	Deleted moj-filet_1.1.filet
send	GetFilesFrom Ja
rx	NoFiles
send	Stop
rx	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo
delay	200
stopServer	

več odjemalcev

sat.fit.SatFixture				
startServer	koda.ChatServiceSatWrapper			
comment	nastavimo odjemalski timeout v ms			
timeOut	2000			
comment	nastavimo zakasnitev v ms po vsakem "send"			
postSendDelay	100			
send	Name Lili1	Name Lili2	Name Lali3	Name Lali4
rx	Named Lili1	Named Lili2	Named Lali3	Named Lali4
send	GetFilesFrom 1	GetFilesFrom i1	GetFilesFrom i3	GetFilesFrom al

rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert file1	Insert file1	Insert file1	Insert file1
rx	Inserted file1	Inserted file1	Inserted file1	Inserted file1
send	GetFilesFrom L	GetFilesFrom li	GetFilesFrom i	GetFilesFrom l
rx	Files file1/Lali3, file1/Lali4, file1/Lili1, file1/Lili2	Files file1/Lali3, file1/Lali4, file1/Lili1, file1/Lili2	Files file1/Lali3, file1/Lali4, file1/Lili1, file1/Lili2	Files file1/Lali3, file1/Lali4, file1/Lili1, file1/Lili2
send	Delete file1	Delete file1	Delete file1	
rx	Deleted file1	Deleted file1	Deleted file1	
send	GetFilesFrom 4	GetFilesFrom i	GetFilesFrom a	GetFilesFrom L
rx	Files file1/Lali4	Files file1/Lali4	Files file1/Lali4	Files file1/Lali4
send				Delete file1
rx				Deleted file1
send	GetFilesFrom Lili	GetFilesFrom 3	GetFilesFrom Li	GetFilesFrom ili
rx	NoFiles	NoFiles	NoFiles	NoFiles
send	Insert bfile1	Insert cfile1	Insert dfile1	Insert afile1
rx	Inserted bfile1	Inserted cfile1	Inserted dfile1	Inserted afile1
send	GetFilesFrom 4	GetFilesFrom 1	GetFilesFrom 2	GetFilesFrom li
rx	Files afile1/Lali4	Files bfile1/Lili1	Files cfile1/Lili2	Files afile1/Lali4, bfile1/Lili1, cfile1/Lili2, dfile1/Lali3
send	Stop			
rx	Stopping	Stopping	Stopping	Stopping
comment	pocakajmo npr. 200 ms, da se stvari na strezniku sprocesirajo			
delay	200			
stopServer				

Povzetek

fit.Summary	
counts	1103 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\Program Files\eclipse\chat\Acceptance Tests ChatService.htm
input update	Fri Jun 04 17:06:42 CEST 2004
output file	C:\Program Files\eclipse\chat\AcceptanceTestResults.htm
run date	Mon Jun 07 11:12:13 CEST 2004

Verzija dokumenta: 1.2

Eksperiment SR03

Zgodba 1 – Izračun NCLOC

Za dano izvorno kodo v Javi želimo prešteti njeno število vrstic (*t.i. NCLOC – Non Commented Lines Of Code*).

Pravila, ki pri tem veljajo, so:

- Prazne vrstice ne štejejo (sem spadajo tudi vrstice, ki vsebujejo le t.i. "bele presledke" (*white spaces* – to so presledki, tabulatorji, znaki *newline*, itd).
- Vrstice, ki vsebujejo le komentarje, ne štejejo.
- Vrstice, ki imajo komentar na koncu vrstice, pred tem pa je v vrstici tudi koda, seveda štejejo (glej npr. vrstico št. 11 spodaj).
- znak *newline* ("\n") označuje konec vrstice, tako da je npr. v formatu Javinega Stringa "1\n" ena vrstica (NCLOC=1).
- ne ukvarjajte se s pravilnostjo sintakse Javine kode, t.j. NCLOC od "1\n2\n" = 2, čeprav to ni veljavna Javina koda.

Kot veste, pozna Java dve vrsti komentarjev: **enovrstične**, kjer se komentar začne z // in nadaljuje do konca vrstice in **večvrstične**, kjer se komentar začne z znakoma /* in konča z znakoma */. Ne pozabite upoštevati, da lahko znaki //, /* in */ nastopajo tudi pri inicializaciji stringov (*string literals*) in znotraj drugih komentarjev. Primer:

```
1   /**
2    * Acceptance test case for NCLOC (NCLOC=8).
3    * @author R. Mugridge, 2002, rick@cs.auckland.ac.nz
4    * @modified M. Pancur, 2003
5    */
6   public class Krneki { /* a multiline comment
7   that stretches over more than one line
8     and is terminated
9   by */ int oneInt; //note this line is a loc
10
11        int secondInt; // above is a blank line
12        String string1 = "/*";
13  //note /* in above string & here - ignore
14        // a comment
15      public static void main(String[] args) {
16              String string2 = "*/"; // */
17      }
18  } // NCLOC = 8
```

Za zgornji primer je tako NCLOC = 8 (vrstice 6, 9, 11, 12, 15, 16, 17, 18).

Napišite razred `LinesOfCode.java`, ki vsebuje (seveda poleg vseh vaših metod) tudi naslednje:

- konstruktor `public LinesOfCode (Reader reader)`, ki inicializira razred LinesOfCode za uporabo z določenim Readerjem (ki je abstraktni razred vseh Reader-jev, npr. StringReader, BufferedReader). Namig: znotraj LinesOfCode ovijte npr. `BufferedReader` okoli reader-ja (npr. `BufferedReader in = new BufferedReader(reader);`), da lahko uporabite znano `in.readLine()` metodo, ki bere iz vira po vrsticah. `readLine()` vrne `null`, če je prišel do konca ("EOF").

- metodo `public int lineCount()`, ki vrne število vrstic kode NCLOC, po zgoraj opisanih pravilih.

Zgodba 2 – LocService.java

Zgornji razred LinesOfCode.java, ki zna izračunati št. vrstic izvorne kode NCLOC, bi radi implementirali kot mrežni strežnik. Strežnik naj zna naenkrat streči večim odjemalcem. Posluša naj na vratih 1234 (port).

Napišite `LocService.java`, ki implementira naš strežnik.

Protokol (ki je sicer precej pomanjkljiv, a zato dokaj preprost :)), po katerem se odjemalci pogovarjajo s strežnikom, je naslednji:

- odjemalec najprej pošlje strežniku v prvi vrstici ukaz "*lineCount ime_datoteke N"*. lineCount je ukaz za izračun NCLOC, nato sledi presledek in ime datoteke, nato še presledek in **skupno št. vseh vrstic kode** (*N*), ki jih bo poslal (zato da strežnik ve, koliko vrstic mora pričakovati)
  Npr: "*lineCount Krneki.java 18*"
- strežnik odgovori z "*lineCount is expecting N lines for ime_datoteke"* (pri čemer je *N* številka vrstic izvorne kode, *ime_datoteke* pa ime datoteke)
  Npr: "*lineCount is expecting 18 lines for Krneki.java*"
- nato odjemalec pošlje strežniku posamezne vrstice izvorne kode.
- Ko strežnik dobi vse vrstice, odgovori z "*lineCount=NCLOC*", npr. "*lineCount=8*".

Strežnik ima nastavljen timeout na Socket-u, s katerim se pogovarja s klientom, na 3 sekunde (3000 ms). Uporabite `setSoTimeout(int miliseconds)`, ki povzroči, da se na strežniku, ki je blokiran na `readLine()` ukazu (čaka odjemalca, da mu kaj pošlje), vrže ob izteku timeouta izjema java.net.SocketTimeoutException. Strežnik v takem primeru predvideva, da je odjemalec "padel dol z mreže, zato zapre Socket do tega neaktivnega odjemalca in pospravi za sabo (nit, ki streže odjemalcu, konča z delom, ...). Ostale niti strežejo klientom normalno naprej!

Pred ukazom in za zadnjim argumentom ni presledkov, če so, to ni veljaven ukaz (vrne "*Error"*). Če gre karkoli narobe, strežnik odgovori z "*Error*" (če je prejel preveč/premalo vrstic, ukazi niso pravilni, itd.) Ukazi so občutljivi na velike in male črke (*case sensitive*). Predvidevamo tudi, da ni vrstic kode nikoli več kot MAX_INT (2^{31}-1).

Primer komunikacije ("←" je sporočilo (poslana vrstica) strežniku, "→" pomeni odgovor strežnika):

```
←lineCount Krneki.java 18
→lineCount is expecting 18 lines for Krneki.java
←/**
```

```
← * Acceptance test case for NCLOC (NCLOC=8).

... ostalih 14 vrstic ...

←     }
←} // NCLOC = 8
→lineCount=8
```

Strežnik teče (posluša za novimi odjemalci na portu 1234) "v neskončnost" (oz. dokler ga "ročno" ne zaustavite), odjemalci ne pošiljajo nobenega "stop" ukaza strežniku, zato ne komplicirajte po nepotrebnem implementacije strežnika!

Acceptance teste za to zgodbo ne pozabite **prej pripraviti** (dopišite setUp() in tearDown(), da vam zažene in nato ugasne vaš strežnik!). Lahko pa dopišete ali popravite tudi vmesne pavze (Thread.sleep()). Kot sem že omenil v navodilih, so acceptance testi (še posebej v tej zgodbi) le **minimum**.

Sprejemni testi, kot so jih dobili subjekti pri eksperimentu SR03:

- LinesOfCodeAcceptanceTest.java (za 1. zgodbo):

```
import java.io.StringReader;
import junit.framework.TestCase;

/**
 * Acceptance test za prvo zgodbo (NCLOC)
 *
 * @author Matjaž, 2003
 */
public class LinesOfCodeAcceptanceTest extends TestCase {

        public void testLinesOfCodeAcceptanceTest() throws Exception {
                String lines =
                                "/**\n"
                              + " * Acceptance test case for NCLOC (NCLOC=8). \n"
                              + " * @author R. Mugridge, 2002, rick@cs.auckland.ac.nz\n"
                              + " * @modified M. Pancur, 2003\n"
                              + " */\n"
                              + "public class Krneki { /* a multiline comment\n"
                              + "that stretches over more than one line \n"
                              + "  and is terminated\n"
                              + "by */ int oneInt; //note this line is a loc\n"
                              + " \n"
                              + "    int secondInt; // above is a blank line\n"
                              + "    String string1 = \"/*\";\n"
                              + "//note /* in above string & here - ignore\n"
                              + "    // a comment\n"
                              + "    public static void main(String[] args) {\n"
                              + "        String string2 = \"*/\"; // */\n"
                              + "    }\n"
                              + "} // NCLOC = 8\n";

                LinesOfCode loc = new LinesOfCode(new StringReader(lines));
                assertEquals(8, loc.lineCount());
        }
}
```

- LocServiceAcceptanceTest.java (za 2. zgodbo):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;

import junit.framework.TestCase;

/**
 * Acceptance test za drugo zgodbo (LocService.java).
 *
 * @author Matjaz, 2003
 */
public class LocServiceAcceptanceTest extends TestCase {

	private LocService locService;

	protected void setUp() throws Exception {
		// vaša koda, ki zažene strežnik
		// npr. nekaj v stilu: locService = new LocService();
		//                     locService.serve(1234, new LocServer());
	}

	protected void tearDown() throws Exception {
		// vaša koda, ki ustavi strežnik
		// npr.nekaj v stilu: locService.stop();
	}

	public void testLocService() throws Exception {

		// Spodaj je nepreoblikovana koda, zato je NE KOPIRAJTE direktno
		// kot vzorec za vaše teste!
		// Namenjena je le kot zelo preprosti acceptance test,
		// ki še zdaleč ne pokriva vseh možnosti!

		Socket client1 = new Socket("localhost", 1234);
		BufferedReader br1 =
			new BufferedReader(new
InputStreamReader(client1.getInputStream()));
		PrintStream ps1 = new PrintStream(client1.getOutputStream(), true);

		Socket client2 = new Socket("localhost", 1234);
		BufferedReader br2 =
			new BufferedReader(new
InputStreamReader(client2.getInputStream()));
		PrintStream ps2 = new PrintStream(client2.getOutputStream(), true);

		Thread.sleep(200);

		ps1.println("lineCount Krneki.java 18");
		String answer = br1.readLine();
		assertEquals("Napačen odgovor strežnika",
			"lineCount is expecting 18 lines for Krneki.java", answer);

		String lines =
				"/**\n"
				 + " * Acceptance test case for NCLOC (NCLOC=8). \n"
				 + " * @author R. Mugridge, 2002, rick@cs.auckland.ac.nz\n"
				 + " * @modified M. Pancur, 2003\n"
				 + " */\n"
				 + "public class Krneki { /* a multiline comment\n"
				 + "that stretches over more than one line \n"
				 + "  and is terminated\n"
				 + "by */ int oneInt; //note this line is a loc\n"
				 + " \n"
				 + "    int secondInt; // above is a blank line\n"
				 + "    String string1 = \"/*\";\n"
				 + "//note /* in above string & here - ignore\n"
				 + "    // a comment\n"
				 + "    public static void main(String[] args) {\n"
				 + "        String string2 = \"*/\"; // */\n"
				 + "    }\n"
```

```
                                + "} // NCLOC = 8\n";
                ps1.print(lines);

                Thread.sleep(200);

                answer = br1.readLine();
                assertEquals("Napačen odgovor strežnika",
                                        "lineCount=8", answer);

                ps2.println("Krneki");
                answer = br2.readLine();
                assertEquals("Napačen odgovor strežnika", "Error", answer);

                client1.close();
                client2.close();
        }
}
```

Eksperiment SR04

Zgodba 1 – Ulomki

Napišite razred Ulomek.java, ki predstavlja ulomke tipa "a/b", pri čemer sta števec a in imenovalec b celi števili (int), imenovalec b je pri tem različen od 0. Instance tega razreda lahko kreiramo z uporabo katerekoli vrednosti, vendar se vedno shranijo in prikažejo v "pokrajšani" obliki (kanonični obliki). Ulomek a/b ima tako kanonično obliko c/d, ki je ekvivalentna a/b, le da je 1 edino število, ki deli c in d, pri čemer je d>0.

pokrajšani ulomki so torej oblike: 1/2, 3/4, -3/4, -12/19, itd.
nepokrajšani pa so oblike 1/-1, 2/-1, 2/4, 6/-2, -6/2, itd.

Razred mora vsebovati naslednje (poleg morebitnih elementov vaše implementacije):

konstruktor `Ulomek()` - kreira ulomek "0/1" (število 0)
`int getStevec()` - vrne stevec ulomka pokrajšane oblike.
`int getImenovalec()` - vrne imenovalec ulomka pokrajšane oblike.
konstruktor `Ulomek(int stevec)` - kreira ulomek, kjer je imenovalec 1 (stevec/1)
konstruktor `Ulomek(int stevec, int imenovalec)` - kreira pokrajšan ulomek tipa "stevec/imenovalec". Če je imenovalec 0, vrže izjemo.
`String toString()` - kreira String verzijo pokrajšanega ulomka "stevec/imenovalec" (npr. "2/3", "-4/5", "0", "-7", itd.). Kadar je imenovalec 1, ga NE izpiše (izpiše le števec), kadar je števec 0, NE izpiše imenovalca (izpiše le "0"). Plusov ("+") ne izpisuje. Presledkov med znaki ni.
`boolean equals(Object obj)` - Vrne ali je ta ulomek numerično ekvivalenten drugemu ulomku (ne pozabite na pretvarjanje tipa ("cast") iz Object v ustrezen tip. Namig: poglejte si tudi Javadoc dokumentacijo `java.lang.Object.equals(Object)`.
`int hashCode()` – vrne hash vrednost objekta (uporablja se npr. v java.util.Hashtable). Vedno, ko »prepišemo« (override) equals metodo, je potrebno spremeniti tudi hashCode() metodo, saj morata ekvivalentna objekta (pri uporabi equals()) vrniti isto hash vrednost. Glej tudi Javadoc za `java.lang.Object.hashCode()`.
`int compareTo(Object obj)` - vrne -1, 0, 1 glede na to, ali je ulomek manjši, enak ali večji od podanega v parametru (spet ne pozabite na "cast" iz Object) .
`Ulomek invert()` - vrne inverzno vrednost ulomka (obrne števec in imenovalec) v NOVEM objektu tipa Ulomek.
`Ulomek multiply(Ulomek ulomek)` - zmnoži med seboj dva ulomka, vrne NOV objekt.
`Ulomek add(Ulomek ulomek)` - sešteje med seboj dva ulomka, vrne NOV objekt Ulomek
konstruktor `Ulomek(String strUlomek)` - kreira pokrajšan ulomek tipa "a/b" iz Stringa. Če je imenovalec 0, vrže izjemo. Presledki ga ne motijo, "+" in "-" pred števili ga ne motijo, izjemo vrže le takrat, kadar je String tako "napačen" (nerazumljiv), da se ne da razbrati, kateri ulomek smo hoteli kreirati (npr. "-+1/--2", "a/b", itd. – v takem primeru vrzite izjemo). Števec in imenovalec imata lahko največ po en predznak (če jih imata več, vrzite izjemo). Med števcem, imenovalcem, znakom "/" in predznaki (+,-) je lahko poljubno število presledkov in ostalih nevidnih znakov (*whitespace*). Oklepaji niso dovoljeni, vrzite izjemo.

Pozor! Zgornji opis pomeni, da mora delati tudi s stringi, ki niso v kanonski obliki (t.j. "-2/-4", "2/-4", " - 2 / + 4", itd. so regularni stringi, ki morajo kreirati ustrezno pokrajšan ulomek).

Metode naj bodo TOČNO takšne kot so opisane zgoraj, ne spreminjati njihovih podpisov (imen, tipov parametrov in vrnjenih vrednosti)! Ne pozabite na AllTests, ki zažene vse teste. Razred Ulomek naj tudi implementira vmesnik `Comparable` (v tem je vsebovana metoda `int compareTo(Object)`).

Za izračunavanje največjega skupnega delitelja dveh celih števil (greatest common divisor - gcd) lahko uporabite kar Evklidov algoritem (napisan v pascalski psevdokodi):

```
gcd(a,b)
  if(b=0) then
    return a
  else
    return gcd(b, a mod b)
```

Zgodba 2 – UlomekService.java

Zgornji razred Ulomek.java, ki predstavlja delo z ulomki, bi radi implementirali kot mrežno storitev. Strežnik naj zna naenkrat streči večim odjemalcem. Posluša naj na vratih 1234 (port).

Napišite `UlomekService.java`, ki implementira naš strežnik. Protokol, po katerem se pogovarjajo strežnik in odjemalci, je vnaprej predpisan in obvezen. Če karkoli ni po protokolu, strežnik odgovori s sporočilom o napaki (Error). Strežnik pozna 5 ukazov:

- `equals prvi_ulomek, drugi_ulomek` → vrača "true" ali "false"
- `compareTo prvi_ulomek, drugi_ulomek` → vrača -1, 0 ali 1, glede na to ali je prvi ulomek manjši, enak ali večji od drugega
- `multiply prvi_ulomek, drugi_ulomek` → vrača zmnožek
- `add prvi_ulomek, drugi_ulomek` → vrača seštevek
- `invert prvi_ulomek` → vrača inverzni ulomek

Pri čemer veljajo za parametre ukazov podobna pravila kot pri Ulomek(String) konstruktorju.

Protokol, po katerem se odjemalci pogovarjajo s strežnikom, je naslednji:

- odjemalec pošlje strežniku ukaz v eni vrstici znakov (vrstica je zaključena *newline*). Vrstica je sestavljena iz ukaza in dveh parametrov, vmes je vejica. Razen pri ukazu invert, ki sprejme le en parameter.
- Strežnik odgovori z rezultatom, če je vse OK. Če ni, vrne Error.

Strežnik ima nastavljen timeout na Socket-u, s katerim se pogovarja s klientom, na 3 sekunde (3000 ms). Uporabite `setSoTimeout(int miliseconds)`, ki povzroči, da se na strežniku, ki je blokiran na `readLine()` ukazu (čaka odjemalca, da mu kaj pošlje), vrže ob izteku timeouta izjema `java.net.SocketTimeoutException`. Strežnik v takem primeru predvideva, da je odjemalec "padel" dol z mreže, zato zapre Socket do tega neaktivnega odjemalca in pospravi za sabo (nit, ki streže odjemalcu, konča z delom, ...). Ostale niti strežejo klientom normalno naprej!

Pred in za ukazi, parametri in vejico, ki lahko loči parametra, je lahko poljubno št. presledkov in ostalih "belih znakov" (t.i. *white space*). Če gre karkoli narobe, strežnik odgovori z "*Error*" (če je prejel nepravilen ukaz, preveč/premalo parametrov, parametri niso pravilni, če izvajanje ukaza sproži izjemo, itd.) Ukazi so občutljivi na velike in male črke (*case sensitive*). Vejica je pri uporabi dveh parametrov obvezna.

Primeri komunikacije ("←" je sporočilo (poslana vrstica) strežniku, "→" pomeni odgovor strežnika):

Primeri pravilnih ukazov	Primeri nepravilnih ukazov
←equals 1 / 2, 2/4 →true ←equals 1 / 2, 2/4	←Add 1/2,1/2 →Error ← compareTe 1/2,1/2

→true ← compareTo 1/2 , 1/4 →1 ← compareTo 1/2 , 1 →-1 ← multiply 1/2,1/4 →1/8 ←add 1/2,1/2 →1 ← invert 1/2 →2	→Error ←add 1/2 1/2 →Error ← invert 0/2 →Error

Strežnik teče (posluša za novimi odjemalci na portu 1234) "v neskončnost" (oz. dokler ga "ročno" ne zaustavite), odjemalci **ne pošiljajo nobenega "stop" ukaza strežniku**, zato **ne komplicirajte po nepotrebnem implementacije strežnika**!

PRILOGA C – PRAKTIČNI PRIKAZ TDD

V tej prilogi bomo prikazali praktični prikaz kratkega razvoja zelo preprostega programa v procesu razvoja TDD. Gre za **pretirano preprost program**, ki pa ravno zato dobro prikaže osnovni ritem razvoja TDD. Gre za enega od prikazov razvoja, ki smo ga tudi sami najprej pokazali študentom. Povzeli smo ga po [Beck 2003]. Uporablja Javo in ogrodje JUnit (http://www.junit.org).

Problem, ki ga rešujemo, je razvoj metode, ki naj nam izračuna Fibonaccijeva števila.

Najprej moramo pokazati, da je fib(0)=0. Napišimo ustrezen test (`assertEquals` je metoda ogrodja JUnit, ki primerja enakost dveh vrednosti, ki sta ji podana kot parametra):

```
public void testFibonacci() {
      assertEquals(0, fib(0));
}
```

Koda se ne prevede, saj še ni metode fib. Da poenostavimo, jo dodajmo kar v testni razred, vrne naj kar 0.

```
int fib(int n) {
      return 0;
}
```

Zaženemo test, ki je v tem primeru slučajno uspešen (ponavadi prazna implementacija ni dovolj, da test uspe).

Naslednji primer testa je za fib(1)=1. Dodajmo testni primer:

```
public void testFibonacci() {
      assertEquals(0, fib(0));
      assertEquals(1, fib(1));
}
```

Zaženemo vse teste, zadnje preverjanje ne uspe. Napišemo kodo tako, da je za nas kar najlažje:

```
int fib(int n) {
      if (n == 0) return 0;
      return 1;
}
```

Zaženemo teste, testi uspejo.

Pred dodajanjem novega testnega primera opazimo, da so se začele stvari v testu podvajati. Preoblikujmo testno kodo tako, da bo lažje dodajati primere.

```
public void testFibonacci() {
      int cases[][]= {{0,0},{1,1}};
      for (int i= 0; i < cases.length; i++)
            assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Zaženemo teste, še vedno je vse OK.

Dodamo naslednji testni primer fib(2)=1

```
public void testFibonacci() {
      int cases[][]= {{0,0},{1,1},{2,1}};
      for (int i= 0; i < cases.length; i++)
            assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Zaženemo teste, ki so, na naše začudenje, vsi uspešni. Slučajno je konstanta 1, ki jo vračamo, dobra tudi za zadnji test. Ker smo vse preverili, dodamo nov test za fib(3)=2

```
public void testFibonacci() {
      int cases[][]= {{0,0},{1,1},{2,1},{3,2}};
      for (int i= 0; i < cases.length; i++)
            assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Zaženemo teste, zadnje preverjanje je neuspešeno. Dodajmo kar najmanj v kodo, da test uspe.

```
int fib(int n) {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      return 2;
}
```

Zaženemo teste, vsi so uspešni. Vendar, ko pogledamo kodo, vidimo v njej precej podvajanja, ki je glavni indikator slabe kode (*code smell*). Vemo, da ko bomo dodali nov testni primer, bi se koda le še poslabšala. Torej je sedaj pravi korak za **generalizacijo** v kodi.

Ko v kodi vrnemo konstanto 2, pravzaprav mislimo na 1+1:

```
int fib(int n) {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      return 1+1;
}
```

Prva enka je pravzaprav primer fib(n-1):

```
int fib(int n) {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      return fib(n-1) + 1;
}
```

Druga enka pa je primer fib(n-2):

```
int fib(int n) {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      return fib(n-1) + fib(n-2);
}
```

Sedaj pogledamo celotno kodo in vidimo, da v bistvu enak princip velja tudi za fib(2), zato lahko drugi pogoj spremenimo:

```
int fib(int n) {
      if (n == 0) return 0;
      if (n == 1) return 1;
      return fib(n-1) + fib(n-2);
}
```

Zaženemo teste, vsi so uspešni, program je končan.

PRILOGA D – PROCESSLOG

Za beleženje podatkov o procesu razvoja smo razvili posebno orodje, ki smo ga poimenovali ProcessLog (glej poglavje 3.6.2 Zbiranje podatkov o procesu razvoja). Gre za komponento (angl. *plug-in*) napisano v Javi, ki se popolnoma integrira v okolje Eclipse (http://www.eclipse.org). Minimalne zahteve komponente so Java 1.4 ali novejša, okolje Eclipse 2.x. Podajamo samo kodo, ki pa mora biti ustrezno strukturirana (pravila pisanja komponent v okolju Eclipse), manjkajo pa tudi slikice (ikone) gumbov.

Datoteka **plugin.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="si.fri.processlog"
   name="Processlog Plug-in"
   version="1.0.3"
   provider-name="Matjaz Pancur"
   class="si.fri.processlog.ProcessLogPlugin">

   <runtime>
      <library name="processlog.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.jdt.junit"/>
   </requires>


   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Process Log"
            description="Process Log"
            visible="true"
            id="si.fri.processlog.actionSet">
         <action
               label="Add Note to Log"
               style="push"
               icon="icons/note_black.gif"
               tooltip="Add Note to Process Log"
               class="si.fri.processlog.actions.AddNoteToLogAction"
               toolbarPath="processlog/note"
               id="si.fri.processlog.addnotetolog">
         </action>
         <action
               label="Pause (break, distraction, ...)"
               hoverIcon="icons/x_red.gif"
               style="radio"
               icon="icons/x_black.gif"
               tooltip="Pause!"
               class="si.fri.processlog.actions.PauseAction"
               toolbarPath="processlog"
               id="si.fri.processlog.pause">
         </action>
         <action
               label="Refactor - 2nd developer"
               hoverIcon="icons/r2_red.gif"
               style="radio"
               icon="icons/r2_black.gif"
               tooltip="Refactor - 2nd developer"
               class="si.fri.processlog.actions.Refactor2Action"
               toolbarPath="processlog"
               id="si.fri.processlog.refactor2">
         </action>
         <action
               label="Code -  2nd developer"
               hoverIcon="icons/c2_red.gif"
               style="radio"
               icon="icons/c2_black.gif"
               tooltip="Code - 2nd developer"
               class="si.fri.processlog.actions.Code2Action"
               toolbarPath="processlog"
               id="si.fri.processlog.code2">
         </action>
         <action
               label="Test - 2nd developer"
               hoverIcon="icons/t2_red.gif"
               style="radio"               icon="icons/t2_black.gif"
               tooltip="Test - 2nd developer"
               class="si.fri.processlog.actions.Test2Action"
               toolbarPath="processlog"
```

```
            id="si.fri.processlog.test2">
        </action>
        <action
            label="Refactor - 1st developer"
            hoverIcon="icons/r1_red.gif"
            style="radio"
            icon="icons/r1_black.gif"
            tooltip="Refactor - 1st developer"
            class="si.fri.processlog.actions.Refactor1Action"
            toolbarPath="processlog"
            id="si.fri.processlog.refactor1">
        </action>
        <action
            label="Code -  1st developer"
            hoverIcon="icons/c1_red.gif"
            style="radio"
            icon="icons/c1_black.gif"
            tooltip="Code - 1st developer"
            class="si.fri.processlog.actions.Code1Action"
            toolbarPath="processlog"
            id="si.fri.processlog.code1">
        </action>
        <action
            label="Test - 1st developer"
            hoverIcon="icons/t1_red.gif"
            style="radio"
            icon="icons/t1_black.gif"
            tooltip="Test - 1st developer"
            class="si.fri.processlog.actions.Test1Action"
            toolbarPath="processlog"
            id="si.fri.processlog.test1">
        </action>
      </actionSet>
   </extension>
   <extension
        point="org.eclipse.jdt.junit.testRunListeners">
      <testRunListener
            class="si.fri.processlog.LogTestRunListener">
      </testRunListener>
   </extension>
   <extension
        point="org.eclipse.ui.startup">
      <startup>
      </startup>
   </extension>

</plugin>
```

Datoteka **si.fri.processlog. LogTestRunListener.java**

```
package si.fri.processlog;

import org.eclipse.jdt.junit.ITestRunListener;

public class LogTestRunListener implements ITestRunListener {

        private long countFail = 0;
        private long countError = 0;
        /* (non-Javadoc)
         * @see org.eclipse.jdt.junit.ITestRunListener#testRunStarted(int)
         */
        public synchronized void testRunStarted(int testCount) {
                ProcessLogPlugin.log("TS " + testCount);
                this.countFail = 0;
                this.countError = 0;
        }

        /* (non-Javadoc)
         * @see org.eclipse.jdt.junit.ITestRunListener#testRunEnded(long)
         */
        public synchronized void testRunEnded(long elapsedTime) {
```

```
		if (this.countFail > 0 || this.countError > 0) {
			ProcessLogPlugin.log(
				"TE FE " + elapsedTime
				+ " F " + this.countFail
				+ " E " + this.countError);
		} else {
			ProcessLogPlugin.log("TE OK " + elapsedTime);
		}
	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testRunStopped(long)
	 */
	public void testRunStopped(long elapsedTime) {
		ProcessLogPlugin.log("TE STOP " + elapsedTime);
	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testStarted(java.lang.String,
java.lang.String)
	 */
	public void testStarted(String testId, String testName) {

	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testEnded(java.lang.String,
java.lang.String)
	 */
	public void testEnded(String testId, String testName) {

	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testFailed(int, java.lang.String,
java.lang.String, java.lang.String)
	 */
	public synchronized void testFailed(
		int status,
		String testId,
		String testName,
		String trace) {
		//ProcessLogPlugin.log("testFailed "+status+" "+testName);
		if (status == STATUS_ERROR) {
			this.countError++;
		} else if (status == STATUS_FAILURE) {
			this.countFail++;
		}

	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testRunTerminated()
	 */
	public void testRunTerminated() {
		testRunEnded(0);
	}

	/* (non-Javadoc)
	 * @see org.eclipse.jdt.junit.ITestRunListener#testReran(java.lang.String,
	 * java.lang.String, java.lang.String, int, java.lang.String)
	 */
	public void testReran(String testId, String testClass, String testName, int
status,
			String trace) {

		ProcessLogPlugin.log("TS 1");
		if (status == STATUS_OK) {
			ProcessLogPlugin.log("TE OK 0");
		} else if (status == STATUS_FAILURE) {
			ProcessLogPlugin.log("TE FE 0 F 1 E 0");
		} else if (status == STATUS_ERROR) {
			ProcessLogPlugin.log("TE FE 0 F 0 E 1");
		}
```

```
        }

}
```

Datoteka **si.fri.processlog.ProcessLogFormatter.java**

```
package si.fri.processlog;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;
/**
 * Formats a log string into format "yyy.mm.dd hh:mm:ss LOGMESSAGE"
 *
 * @author matjaz
 */
class ProcessLogFormatter extends Formatter {
        // This method is called for every log records
        public String format(LogRecord rec) {
                StringBuffer buf = new StringBuffer(200);
                buf.append((new SimpleDateFormat("yyyy.MM.dd HH:mm:ss ")).format(new
Date()));
                buf.append(formatMessage(rec));
                buf.append(System.getProperty("line.separator"));
                return buf.toString();
        }
}
```

Datoteka **si.fri.processlog.ProcessLogMessage.java**

```
package si.fri.processlog;

/**
 * String messages for actions.
 */
final public class ProcessLogMessage {

        public static final String CODE_1 = "C1";
        public static final String REFACTOR_1 = "R1";
        public static final String TEST_1 = "T1";

        public static final String CODE_2 = "C2";
        public static final String REFACTOR_2 = "R2";
        public static final String TEST_2 = "T2";

        public static final String PAUSE = "PA";
        public static final String PRE_LOGNOTE = "LO ";

        public static final String STARTUP = "LO STARTUP";
        public static final String SHUTDOWN = "LO SHUTDOWN";
}
```

Datoteka **si.fri.processlog.ProcessLogPlugin.java**

```
package si.fri.processlog;

import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Logger;

import org.eclipse.core.resources.IWorkspace;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IPluginDescriptor;
import org.eclipse.core.runtime.Platform;
```

```
import org.eclipse.ui.IStartup;
import org.eclipse.ui.plugin.AbstractUIPlugin;

/**
 * The main plugin class to be used in the desktop.
 */
public class ProcessLogPlugin extends AbstractUIPlugin implements IStartup {

	//The shared instance.
	private static ProcessLogPlugin plugin;

	// Logger
	static Logger logger;
	static FileHandler fh;
	private static final String LOGGER_FILE_NAME = "/.metadata/process.log";
	private static final String LOGGER = "si.fri.ProcessLog";

	/**
	 * The constructor.
	 */
	public ProcessLogPlugin(IPluginDescriptor descriptor) {
		super(descriptor);
		plugin = this;

		logger = Logger.getLogger(LOGGER);
		try {
			fh =
				new FileHandler(
					Platform.getLocation().toOSString() +
LOGGER_FILE_NAME,
					true);
		} catch (SecurityException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

		fh.setFormatter(new ProcessLogFormatter());
		logger.addHandler(fh);
	}

	/**
	 * Returns the shared instance.
	 */
	public static ProcessLogPlugin getDefault() {
		return plugin;
	}

	/**
	 * Returns the workspace instance.
	 */
	public static IWorkspace getWorkspace() {
		return ResourcesPlugin.getWorkspace();
	}

	/** Logs logString to the logger.
	 *
	 * @param logString String message to log
	 */
	public static void log(String logString) {
		logger.info(logString);
	}

	/**
	 * At plug-in startup log STARTUP message to the log.
	 * @see org.eclipse.ui.IStartup#earlyStartup()
	 */
	public void earlyStartup() {
		log(ProcessLogMessage.STARTUP);
	}

	/**
	 * At plug-in shutdown log SHUTDOWN message to the log.
	 * @see org.eclipse.core.runtime.Plugin#shutdown()
```

```
	 */
	public void shutdown() throws CoreException {
		super.shutdown();
		log(ProcessLogMessage.SHUTDOWN);
	}

}
```

Datoteka **si.fri.processlog.actions.AddNoteToLogAction.java**

```
package si.fri.processlog.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.InputDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.window.Window;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import si.fri.processlog.ProcessLogMessage;
import si.fri.processlog.ProcessLogPlugin;

/**
 * "Adding note to log" action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class AddNoteToLogAction implements IWorkbenchWindowActionDelegate {
	private IWorkbenchWindow window;
	/**
	 * The constructor.
	 */
	public AddNoteToLogAction() {
	}

	 /**
	 * The action "Adding note to log" has been activated. The argument of the
	 * method represents the 'real' action sitting
	 * in the workbench UI.
	 * @see IWorkbenchWindowActionDelegate#run
	 */
	public void run(IAction action) {
		InputDialog dialog =
			new InputDialog(window.getShell(), "Log", "Opomba:", null, null);
		dialog.setBlockOnOpen(true);
		dialog.open();
		String dialogValue = dialog.getValue();
		if (dialog.getReturnCode() == Window.OK && dialogValue != "") {
			ProcessLogPlugin.log(ProcessLogMessage.PRE_LOGNOTE + dialogValue);
		}

	}

	/**
	 * Selection in the workbench has been changed. We
	 * can change the state of the 'real' action here
	 * if we want, but this can only happen after
	 * the delegate has been created.
	 * @see IWorkbenchWindowActionDelegate#selectionChanged
	 */
	public void selectionChanged(IAction action, ISelection selection) {
	}

	/**
	 * We can use this method to dispose of any system
	 * resources we previously allocated.
	 * @see IWorkbenchWindowActionDelegate#dispose
	 */
	public void dispose() {
	}
```

```
	/**
	 * We will cache window object in order to
	 * be able to provide parent shell for the message dialog.
	 * @see IWorkbenchWindowActionDelegate#init
	 */
	public void init(IWorkbenchWindow window) {
		this.window = window;
	}
}
```

Datoteka **si.fri.processlog.actions.Code1Action.java**

```
package si.fri.processlog.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import si.fri.processlog.ProcessLogMessage;
import si.fri.processlog.ProcessLogPlugin;

/**
 * "Coding 1st developer" action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class Code1Action implements IWorkbenchWindowActionDelegate {
	private IWorkbenchWindow window;
	/**
	 * The constructor.
	 */
	public Code1Action() {
	}

	/**
	 * The action "Coding 1st developer" has been activated. The argument of the
	 * method represents the 'real' action sitting
	 * in the workbench UI.
	 * @see IWorkbenchWindowActionDelegate#run
	 */
	public void run(IAction action) {
		if (action.isChecked()){
			ProcessLogPlugin.log(ProcessLogMessage.CODE_1);
		}
	}

	/**
	 * Selection in the workbench has been changed. We
	 * can change the state of the 'real' action here
	 * if we want, but this can only happen after
	 * the delegate has been created.
	 * @see IWorkbenchWindowActionDelegate#selectionChanged
	 */
	public void selectionChanged(IAction action, ISelection selection) {
	}

	/**
	 * We can use this method to dispose of any system
	 * resources we previously allocated.
	 * @see IWorkbenchWindowActionDelegate#dispose
	 */
	public void dispose() {
	}

	/**
	 * We will cache window object in order to
```

```
	 * be able to provide parent shell for the message dialog.
	 * @see IWorkbenchWindowActionDelegate#init
	 */
	public void init(IWorkbenchWindow window) {
		this.window = window;
	}
}
```

Datoteka **si.fri.processlog.actions.Refactor1Action.java**

```
package si.fri.processlog.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import si.fri.processlog.ProcessLogMessage;
import si.fri.processlog.ProcessLogPlugin;

/**
 * "Refactor 1st developer" action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class Refactor1Action implements IWorkbenchWindowActionDelegate {
	private IWorkbenchWindow window;
	/**
	 * The constructor.
	 */
	public Refactor1Action() {
	}

	/**
	 * The action "Refactor 1st developer" has been activated. The argument of the
	 * method represents the 'real' action sitting
	 * in the workbench UI.
	 * @see IWorkbenchWindowActionDelegate#run
	 */
	public void run(IAction action) {
		if (action.isChecked()){
			ProcessLogPlugin.log(ProcessLogMessage.REFACTOR_1);
		}
	}

	/**
	 * Selection in the workbench has been changed. We
	 * can change the state of the 'real' action here
	 * if we want, but this can only happen after
	 * the delegate has been created.
	 * @see IWorkbenchWindowActionDelegate#selectionChanged
	 */
	public void selectionChanged(IAction action, ISelection selection) {
	}

	/**
	 * We can use this method to dispose of any system
	 * resources we previously allocated.
	 * @see IWorkbenchWindowActionDelegate#dispose
	 */
	public void dispose() {
	}

	/**
	 * We will cache window object in order to
	 * be able to provide parent shell for the message dialog.
	 * @see IWorkbenchWindowActionDelegate#init
	 */
```

```
	public void init(IWorkbenchWindow window) {
		this.window = window;
	}
}
```

Datoteka **si.fri.processlog.actions.Test1Action.java**

```
package si.fri.processlog.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import si.fri.processlog.ProcessLogMessage;
import si.fri.processlog.ProcessLogPlugin;

/**
 * "Testing 1st developer" action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class Test1Action implements IWorkbenchWindowActionDelegate {
	private IWorkbenchWindow window;
	/**
	 * The constructor.
	 */
	public Test1Action() {
	}

	/**
	 * "Testing 1st developer" action has been activated. The argument of the
	 * method represents the 'real' action sitting
	 * in the workbench UI.
	 * @see IWorkbenchWindowActionDelegate#run
	 */
	public void run(IAction action) {
		if (action.isChecked()){
			ProcessLogPlugin.log(ProcessLogMessage.TEST_1);
		}
	}

	/**
	 * Selection in the workbench has been changed. We
	 * can change the state of the 'real' action here
	 * if we want, but this can only happen after
	 * the delegate has been created.
	 * @see IWorkbenchWindowActionDelegate#selectionChanged
	 */
	public void selectionChanged(IAction action, ISelection selection) {
	}

	/**
	 * We can use this method to dispose of any system
	 * resources we previously allocated.
	 * @see IWorkbenchWindowActionDelegate#dispose
	 */
	public void dispose() {
	}

	/**
	 * We will cache window object in order to
	 * be able to provide parent shell for the message dialog.
	 * @see IWorkbenchWindowActionDelegate#init
	 */
	public void init(IWorkbenchWindow window) {
		this.window = window;
	}
```

```
}
```

Datoteka **si.fri.processlog.actions.PauseAction.java**

```
package si.fri.processlog.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import si.fri.processlog.ProcessLogMessage;
import si.fri.processlog.ProcessLogPlugin;

/**
 * "Pressing pause button" action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class PauseAction implements IWorkbenchWindowActionDelegate {
	private IWorkbenchWindow window;
	/**
	 * The constructor.
	 */
	public PauseAction() {
	}

	/**
	 * "Pressing pause button" action has been activated. The argument of the
	 * method represents the 'real' action sitting
	 * in the workbench UI.
	 * @see IWorkbenchWindowActionDelegate#run
	 */
	public void run(IAction action) {
		if (action.isChecked()){
			ProcessLogPlugin.log(ProcessLogMessage.PAUSE);
		}
	}

	/**
	 * Selection in the workbench has been changed. We
	 * can change the state of the 'real' action here
	 * if we want, but this can only happen after
	 * the delegate has been created.
	 * @see IWorkbenchWindowActionDelegate#selectionChanged
	 */
	public void selectionChanged(IAction action, ISelection selection) {
	}

	/**
	 * We can use this method to dispose of any system
	 * resources we previously allocated.
	 * @see IWorkbenchWindowActionDelegate#dispose
	 */
	public void dispose() {
	}

	/**
	 * We will cache window object in order to
	 * be able to provide parent shell for the message dialog.
	 * @see IWorkbenchWindowActionDelegate#init
	 */
	public void init(IWorkbenchWindow window) {
		this.window = window;
	}
}
```

Datoteke v paketu si.fri.processlog.actions **Test2Action.java, Refactor2Action.java** in **Code2Action.java** se razlikujejo od akcij za prvega razvijalca le po nizu, ki ga izpišejo v dnevniško datoteko (v run() metodi).

PRILOGA E – ANKETE IN REZULTATI

Subjekti so izpolnili dve anketi. Prvo (poimensko) na začetku eksperimenta, drugo (anonimno) na koncu. Anketi sta bili sicer obsežnejši, vendar so ostala vprašanja manj pomembna za našo raziskavo. Namenjena so bila bolj pregledu dosedanje prakse pri programiranju (kako so do sedaj razhroščevali in testirali programe, kakšna razvojna okolja so uporabljali, itd.), zato smo jih tu spustili.

Vsi subjekti so delali v Javi, zato smo pri prvi anketi pri drugem vprašanju upoštevali le odgovore za Javo (le vprašanje 2.a in 2.a (meseci izkušenj)). Za prvo anketo objavljamo odgovore le za subjekte, ki so sodelovali pri obeh eksperimentih v določenem letu (delu v parih in samostojnem delu). Rezultate eksperimenta SR03 smo zaradi suma pristranosti zaradi smrtnosti izločili, zato je potrebno tudi rezultate prve ankete v letu 2003 obravnavati temu primerno (glej Prilogo A).

V tabelah pri vprašanjih 3, 4 in 5 pomeni 1 odgovor "da", 0 odgovor "ne". Prazna celica pomeni, da ni bil podan odgovor. "p" pomeni verjetnost *p*, da razlika v povpr. vrednosti med skupinama TDD in ITL ni statistično značilna (Mann-Whitney U Exact test, α=0,1, dvosmerni test). Vprašanja 1, 2 in 3 so bolj subjektivna kot povprečna ocena izpitov prvih treh letnikov (označena v tabelah prve ankete kot "Povp. ocena"), ki smo jo pridobili sami (torej jo niso navajali študentje, s tem smo izločili možno napako).

Prva anketa

1. Ocenite vaše dosedanje izkušnje:

a) znanje programiranja _____ (1 slabo – 10 zelo dobro)
b) OO razvoj _____ (1 slabo – 10 zelo dobro)

2. Katere programske jezike obvladate (1 – slabo poznam, 10 – zelo dobro poznam)

			meseci izkušenj
a)	Java	_____	_____
b)	Pascal	_____	_____
c)	C/C++	_____	_____
d)	ostali: _____	_____	_____

3. Imate izkušnje pri programiranju v projektih zunaj fakultete?

DA NE

Druga anketa (anonimna)

4. Primerjava procesa razvoja TDD/ITL (opisujete tistega, ki ste ga uporabljali pri vajah in izpitu) s klasičnim razvojem (ki ste ga uporabljali prej).

a)	Raje uporabljam viden "green in red bar" pri vsakem zagonu testov	☐ da	☐ ne
b)	Uporaba TDD/ITL omogoča programerju, da bolje razume zahteve	☐ da	☐ ne
c)	Uporaba TDD/ITL zmanjša čas razhroščevanja	☐ da	☐ ne
d)	Uporaba TDD/ITL zmanjša čas razvoja programa	☐ da	☐ ne
e)	S TDD/ITL je koda boljše kvalitete	☐ da	☐ ne
f)	Z uporabo TDD/ITL je design programa preprostejši	☐ da	☐ ne
g)	S TDD/ITL je delo opazno bolj učinkovito	☐ da	☐ ne
h)	Osvojiti način TDD/ITL dela je težko	☐ da	☐ ne
i)	slabost TDD/ITL je pomanjkanje posebne faze načrtovanja na začetku	☐ da	☐ ne
j)	zaradi avtomatiziranega testov imam večje zaupanje v pravilnost rešitve	☐ da	☐ ne
k)	Sprejel bi delo v podjetju, kjer uporabljajo TDD/ITL	☐ da	☐ ne

5. Tudi v prihodnosti pri drugih programerskih nalogah imam še namen preizkušati in uporabljati:

a)	avtomatizirano testiranje	☐ da	☐ ne
b)	vnaprejšnje pisanje testov	☐ da	☐ ne
c)	izrazito iterativno reševanje problemov	☐ da	☐ ne
d)	JUnit	☐ da	☐ ne
e)	Eclipse	☐ da	☐ ne
f)	preoblikovanje	☐ da	☐ ne
g)	TDD	☐ da	☐ ne
h)	ITL	☐ da	☐ ne
i)	stalno programiranje v paru	☐ da	☐ ne

Anketa 1 – 2003 TDD

Vprašanje	Povp. ocena	1.a	1.b	2.a	2.a (mes. izk.)	3
	8,53	9	8	5	4	1
	7,65	8	6	6	2	1
	8,20	8	7	5	5	0
	9,63	7	8	8	11	1
	7,88	8	7	8	5	1
	7,03	8	8	2	0	1
	8,09	7		4	6	1
	8,86	10	10	10	30	1
	8,94	9	5	4	5	0
	7,77	8	7	6	3	0
	7,53	7	4	2	1	1
	8,02	7	6	7	6	1
	7,70	7	4	3	1	1
	7,22	6	2	2	1	1
	7,50	7	6	1	0	1
	7,13	3	3	1		0
	7,56	8	7	5	3	1
Povp. vr.	7,95	7,47	6,13	4,65	5,19	0,76
St. dev.	0,70	1,50	2,09	2,64	7,20	0,44
Mediana	7,77	8,00	6,50	5,00	3,50	1,00
p	0,38	0,34	0,35	0,81	0,56	0,67

Anketa 1 – 2003 ITL

Vprašanje	Povp. ocena	1.a	1.b	2.a	2.a (mes. izk.)	3
	7,85	9	6	4	12	1
	7,90	8	8	4	4	0
	8,44	8	2	2	0	0
	7,58	3	3	4	5	1
	8,57	7	8	6		0
	7,70	5	3	3	6	0
	8,11	8	8	8	3	1
	9,15	7	7	7	6	1
	8,00	6	2	5	1	1
	7,83	6	3	4	6	1
	7,63	8	6	6	3	1
Povp. vr.	8,07	6,82	5,09	4,82	4,60	0,64
St. dev.	0,47	1,72	2,51	1,78	3,34	0,50
Mediana	7,90	7,00	6,00	4,00	4,50	1,00
p	0,38	0,34	0,35	0,81	0,56	0,67

Anketa 1 – 2004 TDD

Vprašanje	Povp. ocena	1.a	1.b	2.a	2.a (mes. izk.)	3
	7,38	8	8	8	50	1
	7,93	8	8	8	36	1
	8,07	8	1	1	1	0
	7,05	7	2	2	0	1
	8,14	7	5	3	1	1
	8,27	6	4	5	3	0
	7,97	5	6	4	6	0
	7,51	7	6	4	2	0
	7,64	7	3	1	0	1
	7,64	7	4	2	0	0
	7,81	6	5	5	1	1
	7,68	5	5	5	12	0
	7,52	9	8	6	1	1
	9,45	10	10	10	108	1
Povp. vr.	7,86	7,14	5,36	4,57	15,79	0,57
St. dev.	0,56	1,41	2,53	2,74	30,59	0,51
Mediana	7,75	7,00	5,00	4,50	1,50	1,00
p	0,89	0,22	0,35	0,38	0,90	0,48

Anketa 1 – 2004 ITL

Vprašanje	Povp. ocena	1.a	1.b	2.a	2.a (mes. izk.)	3
	7,11	3	4	1	2	0
	7,81	5	5	1	2	0
	8,06	9	7	3	1	1
	8,07	7	3	5	1	0
	7,75	7	6	6	2	0
	7,30	7	8	4	1	1
	8,19	6	5	6	2	0
	8,16	10	7	5	10	1
	7,35	8	6	6	1	0
	7,17	7	1	2	12	1
	7,19	4	2	3	12	0
	8,08	3	3	1	3	1
	7,88	7	3	3	5	0
	7,15	4	3	2	1	0
	8,04	10	10	10	18	1
	8,36	4	3	3	1	0
	8,00	5	3	3	1	1
	8,71	7	4	2	1	0
Povp. vr.	7,80	6,28	4,61	3,67	4,22	0,39
St. dev.	0,48	2,19	2,33	2,33	5,14	0,50
Mediana	7,94	7,00	4,00	3,00	2,00	0,00
p	0,89	0,22	0,35	0,38	0,90	0,48

Anketa 2 – 2003 TDD

Vprašanje	4.a	4.b	4.c	4.d	4.e	4.f	4.g	4.h	4.i	4.j	4.k	5.a	5.b	5.c	5.d	5.e	5.f	5.g	5.h	5.i
	1	1	1	0	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	0
	1	1	0	0	1	1	0	1	0	1	1	1	0	1	1	1	1	0	1	0
	1	1	0	0	1	1	0	1	0	1	1	1	1	0	1	1	1	1		1
	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0
	1	1	1	0	1	0	0	1	0	1	1	1	0	0	1	1	1	0	1	0
	1	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1
	1	0	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1		1
	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	1	1	0	1	1
	1	0	1	0		0	0	1	0	1	1	1	0	0	1	1	1			0
	1	1	0	0	1	0	0	1	1	1	1	1	0	0	1	1	1	0	1	0
	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	1	1	0	0	1	0	0	1	1	1	1	1	0	1	0	1	1	1	0
	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1
	1	1	0	0	1	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0
	1	0	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0
	1	1	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0
	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	0	0	1	0	1	0	1	1	1	1	0	0	0	1	1	0
Odg. "da"	18	12	11	4	14	10	5	12	14	16	17	15	8	12	14	15	17	8	9	5
Vseh odg.	19	19	19	19	18	19	19	19	19	19	19	19	19	19	19	19	19	18	15	19

Anketa 2 – 2003 ITL

Vprašanje	4.a	4.b	4.c	4.d	4.e	4.f	4.g	4.h	4.i	4.j	4.k	5.a	5.b	5.c	5.d	5.e	5.f	5.g	5.h	5.i
	1	1	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1
	1	1	1	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0
	1	1	1	0	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	0	0	1	1	1	0	1	1	1	1	0	1	1
	1	1	0	0	0	0	0	1	1	0	0		0	0	0	1	1	0	1	0
	1	0	1	0	1	0	1	0	1	1	1	1	0	1	1	0	1	0	1	0
	1	1	0	0	0	1	0	0	1	1	1	1	0	1	1	1	0	0	0	0
	1	1	0	1	1	0	0	0	1	1	1	1	0	1	1	1	1	0	1	0
	1	0	1	1	1	0	0	0	0	1	1	1	0	0	1	1	1	0	1	0
	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
	1	0	0	1	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1
	1	0	0	0	1	0	0	0	1	1	0	1	1	0	1	1	1	0	0	0
	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0
	1	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	0
	1	0	1	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	0	0
Odg. "da"	15	9	8	5	11	6	6	3	9	14	12	14	6	11	13	13	14	4	12	4
Vseh odg.	15	15	15	15	15	15	15	15	15	15	15	14	15	15	15	15	15	15	15	15

Anketa 2 – 2004 TDD

Vprašanje	4.a	4.b	4.c	4.d	4.e	4.f	4.g	4.h	4.i	4.j	4.k	5.a	5.b	5.c	5.d	5.e	5.f	5.g	5.h	5.i
	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0
	1	1	1	0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0
	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1	0
	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	0	1	0	1	0
	1	1	1	0	1	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0
	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0
	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0
	1	0	1		1	0	0	1	1	0	1	1	0	0	1	1	1	0	1	0
	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0	1	0	1	1
	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	1	1	1	0	0
	1	0	1	1	1	0	0	0	0	1	1	1		1	1	1	1			
	1	1	0	0	1	0	0	0	0	1	1	1	1	0	1	1	1	1		0
	1	0	1	0	1	1	0	1	1	1	1	1	0	1	1	0	1	0	1	0
Odg. "da"	**14**	**10**	**11**	**6**	**13**	**5**	**4**	**6**	**10**	**10**	**11**	**10**	**7**	**6**	**9**	**10**	**10**	**5**	**7**	**1**
Vseh odg.	**14**	**14**	**14**	**13**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**13**	**14**	**14**	**14**	**14**	**13**	**12**	**13**

Anketa 2 – 2004 ITL

Vprašanje	4.a	4.b	4.c	4.d	4.e	4.f	4.g	4.h	4.i	4.j	4.k	5.a	5.b	5.c	5.d	5.e	5.f	5.g	5.h	5.i
	1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1	1	0	0	0
	1	0	1	0	1	1	0	0	1	0	1	0	0	1	0	1	1		0	0
	1	1	1	1	1	0	1	1	0	1	1	1	0	1	1	1	0	0	1	1
	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	1	
	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	0
	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0
	1	0	1	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1
	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0
	1	1	1	1	1	0	0	1	1	1	0	1	0	1	1	1	1	0	1	1
	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1
	1	1	1	0	1	1	0	0	1	1	1	1	0	1	0	1	0	0	1	1
	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0
	1	1	1	0	1	1	1	0	1	0	1	0	0	1		0	1	0	1	0
	1	1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	1	0	1	0
	0	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
Odg. "da"	**16**	**14**	**15**	**10**	**13**	**11**	**9**	**4**	**13**	**15**	**16**	**14**	**8**	**13**	**12**	**14**	**15**	**5**	**14**	**8**
Vseh odg.	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**16**	**17**	**17**	**16**	**17**	**16**