Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matija Balantič

Testno voden razvoj programske opreme v Javi EE

diplomsko delo

univerzitetni študijski program fakultete za računalništvo in informatiko

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01972 / 2013
Datum: 5.11.2013

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

Kandidat: **MATIJA BALANTIČ**

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

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

Proučite postopek testno vodenega razvoja programske opreme ter analizirajte njegove prednosti in slabosti. Predstavite orodja, ki se uporabljajo za testno voden razvoj in sprotno integracijo pri razvoju aplikacij v programskem jeziku Java EE, ter opišite uporabo testov na različnih ravneh testiranja. Opisane postopke prikažite na praktičnem primeru razvoja preproste aplikacije in analizirajte kakovost tako dobljene programske kode.

Mentor:
izr. prof. dr. Viljan Mahnič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matija Balantič, z vpisno številko **63070026**, sem avtor diplomskega dela z naslovom:

Testno voden razvoj programske opreme v Javi EE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, 20. novembra 2013 Podpis avtorja:

Posebna zahvala gre izr. prof. dr. Viljanu Mahniču za mentorstvo, hitre in izčrpne odgovore in svetovanje pri pisanju magistrske naloge.

Petru Brajaku iz podjetja Medius se zahvaljujem za vso podporo in koristne nasvete ter možnost pisanja diplome v podjetju. S tem mi je zelo olajšal delo in brez njega naloga ne bi bila takšna, kot je.

Zahvaljujem se Martinu Bolčini za potrpežljive odgovore na vsa moja vprašanja.

Iskrena hvala punci Špeli, ki je verjela vame in mi stala ob strani. Hvala tudi za vse nasvete in pomoč.

Hvala tudi Majdi Deržič, ki mi je nalogo zlektorirala.

Zahvaljujem se tudi svoji družini, ki me je podpirala, spodbujala in mi stala ob strani med celotnim študijem.

Kazalo

Povzetek

Testno voden razvoj je tehnika, katere glavna ideja je, da se pred implementacijo nekega delčka kode zanj napiše test, ki bo ta delček testiral. Razvoj poteka v kratkih iteracijah, ki so sestavljene iz treh osnovnih korakov — *testiraj-kodiraj-preoblikuj*.

Diplomska naloga prikazuje razvoj spletne aplikacije Java EE — FerApp s pomočjo testno vodenega razvoja in sprotne integracije. Testom enot, s katerimi se vodi razvoj aplikacije, se dodajo integracijski in funkcionalni testi. Z njimi se prepričamo, da aplikacija deluje pravilno tudi znotraj aplikacijskega strežnika in da se različni deli aplikacije pravilno povezujejo med seboj. Z uporabo tehnike sprotne integracije se zagotovi pogosto izvajanje vseh vrst testov, saj to opravilo vzame preveč časa, da bi to počel programer sam. Strežnik za sprotno integracijo se poveže z orodjem za preverjanje kakovosti kode, kar omogoči, da se med razvojem neprestano preverja kvaliteta kode. Naloga tako predstavi celoten cikel razvoja poslovne aplikacije s tehnologijo Java EE in prikaže programiranje aplikacije na način, s katerim se zagotovijo močni temelji, ki olajšajo nadgrajevanje in vzdrževanje aplikacije v prihodnosti.

Ključne besede: testno voden razvoj, Java EE, sprotna integracija, testiranje, razvoj programske opreme

Abstract

Test driven development (TDD) is a technique with the main idea of writing a failing test first, which is then made to pass by implementing a particular snippet of code. Development is done in short iterations which consist of three basic steps, namely test-code-refactor.

The thesis shows the development of Java EE web applications FerApp using test driven development and continuous integration. The application development was driven with unit tests and complemented with integration and functional tests. Integration and functional tests tested that the application works properly even within the application server and that different parts of the application are properly wired with each other. The technique of continuous integration allows running all types of tests on a regular basis. The task of running all the tests would otherwise take too much time for a programmer to do it by himself. We integrated our continuous integration server with quality assurance tool Sonar. Sonar is run every time an automated build was triggered which gives a constant feedback about the quality of the project. This way the work concludes the whole cycle of development process of business application with Java EE and presents a way to create a strong foundation to facilitate the upgrading and maintenance of the application in the future.

Keywords: test driven development, Java EE, continuous integration, testing, software development

Poglavje 1

Uvod

Cilj programerjev je pisati lepo, lahko berljivo in kvalitetno kodo, ki ne vsebuje hroščev. Z večanjem števila zahtev in kompleksnostjo programske opreme se to izkaže za težko nalogo. Poleg tega se pri razvoju programske opreme razvijalci pogosto srečujemo s kratkimi roki in nenadnimi spremembami funkcionalnosti programov oz. aplikacij, ki jih razvijamo. Posledica tega je, da koda v naših programih postane nepregledna, pride do pojava hroščev, funkcionalnost, ki je že preverjeno delovala, ne deluje več ... Ravno take težave in situacije so razvijalce vodile v iskanje boljšega načina razvoja. Razvili so testno voden razvoj (ang. Test Driven Development — TDD), ki postaja danes vse bolj popularen način razvoja programske opreme.

O TDD je bilo narejenih veliko raziskav [4, 16, 22], ki kažejo, da tak način razvoja programske opreme izboljša kvaliteto kode in zmanjša število hroščev v kodi. Kompleksnost aplikacij JEE[1] (integracija spletnih storitev, večnivojska arhitektura, sinhroni in asinhroni poslovni procesi, zahtevni uporabniški vmesniki in vsak dan višja pričakovanja uporabnikov o čim prijetnejši uporabniški izkušnji, enkapsulirana poslovna pravila itd.) ter potrebe po večjih razvojnih skupinah so glavni razlogi za uvajanje TDD.

Cilj naloge je preizkusiti tehniko TDD in različna orodja za testiranje ter

[1] Aplikacije, ki so napisane v programskem jeziku Java in temeljijo na platformi Java Enterprise Edition (Java EE).

učinkovitost tovrstnega načina razvoja v kontekstu programskega jezika Java, različice Java EE. Aplikacije, napisane v programskem jeziku Java EE, tečejo znotraj aplikacijskega strežnika, kar predstavlja drugačen izziv za testiranje kot testiranje aplikacij, napisanih v drugih programskih jezikih, kot so PHP, Python, Java SE ...

V prvem delu diplomske naloge bomo predstavili različne vrste testov, tehniko TDD in glavna načela razvoja aplikacij s to tehniko. Opisali bomo tudi prednosti in slabosti tehnike TDD. V drugem delu bomo tehniko in orodja preizkusili na preprostem primeru. Aplikacijo bomo razdelili na več plasti v skladu z arhitekturno zasnovo realnih aplikacij, napisanih v programskem jeziku Java EE, in predstavili načine testiranja vsake plasti posebej. Ker je sprotna integracija eden izmed temeljev za avtomatizacijo izvajanja testov, jo bomo v tretjem delu podrobneje opisali in prikazali uporabo strežnika za sprotno integracijo Jenkins. Z njim in orodjem Sonar bomo nato naredili analizo kode naše aplikacije.

Poglavje 2

Testiranje programske opreme

Načinov testiranja programske opreme in vrst testov je veliko. Način in vrsta testov, ki jih uporabimo, se razlikujeta glede na metodologijo razvoja programske opreme, ki jo uporabimo. Pri tradicionalnih metodologijah (npr. slapovni model) se testiranje programske opreme izvede šele, ko je korak programiranja in implementacije zahtev že zaključen. V nasprotju s tradicionalnimi pristopi agilne metodologije temeljijo na sprotnem testiranju [35].

2.1 Testi enot — temelj TDD

Testi enot se osredotočajo na testiranje posameznih enot programske opreme [11]. Enota je najmanjši zaokrožen del kode programa, ki ga lahko stestiramo. V primeru objektno orientirane programske kode je to pogosto razred, lahko tudi samo metoda. Testne metode znotraj testa enot preverjajo posamezne delčke enote in zagotavljajo, da koda deluje tako, kot si je zamislil razvijalec. Teste enot ponavadi programirajo razvijalci sami in z njimi preverjajo delovanje programa z vidika programerja. Gre za testiranje vrste "bela skrinjica", pri katerem se programer zaveda in upošteva notranjo strukturo enote, ki jo testira. To omogoča celovito testiranje izoliranih delov enote.

Testi enot so med seboj neodvisni. Namenjeni so pogostemu izvajanju, saj želimo ves čas preverjati, da koda, ki smo jo že stestirali, še vedno deluje

kljub našim spremembam. To, da so neodvisni, pomeni, da niso odvisni od dosegljivosti in podatkov zunanjih sistemov, npr. od podatkovne baze. Da dosežemo neodvisnosti, si lahko pomagamo z uporabo navideznih metod (ang. stub) in objektov (ang. mock objects).

Ker testi enot preverjajo delovanje z vidika programerja, predstavljajo temelj TDD. Razlog je v tem, da mora biti vsa koda, ki jo želimo stestirati, napisana tako, da jo lahko stestiramo. Pri testno vodenem razvoju test napišemo, še preden napišemo kodo, ki implementira funkcionalnost. Ker najprej napišemo test, to avtomatsko "prisili" programerja v pisanje kode, ki jo bo lahko stestiral. Posledično je koda bolj preprosta in pregledna. O prednostih, ki jih prinese testiranje enot, bomo več povedali v poglavju 3.

2.2 Integracijski testi

Testi enot so pomembni za validacijo poslovne logike aplikacije, ne zagotavljajo pa, da aplikacijo Java EE lahko namestimo (ang. deploy) na strežnik. Medtem ko so testi enot hitri in testirajo majhne dele enote, so integracijski testi počasnejši in testirajo povezave med posameznimi enotami. Z njimi se prepričamo, da je naša implementacija vmesnikov pravilna in da lahko komunicira z ostalimi enotami, ki so del programskega sistema. Napake, ki jih odkrijemo z integracijskimi testi, s testi enot ne moremo odkriti.

Testiranje integracije sistema je pomembna stopnja v razvoju aplikacije in ponavadi sledi testiranju enot [5].

2.3 Ročno testiranje

Ročno testiranje je eden izmed najpogostejših načinov testiranja. Pogosto ga uporabljajo že programerji in poteka tako, da programer sprogramira del aplikacije in ga nato preizkusi s pomočjo uporabniškega vmesnika. Problem pri ročnem testiranju je, da testiranje neke funkcionalnosti vedno vzame enako veliko časa, rezultati pa so težko ponovljivi (drugi tester ne ve, katere

vrednosti je vnesel prvi tester) [11].

Uporabljajo ga tudi testerji, ki igrajo vlogo končnega uporabnika. Pri zahtevnejših programih ali aplikacijah testerji ponavadi sledijo testnemu načrtu, ki zagotavlja, da s testi pokrijejo vse možne scenarije in s tem zagotovijo celovitost testiranja [33].

2.4 Funkcionalni testi

Težki ponovljivosti ročnih testov se lahko izognemo z uporabo avtomatiziranih testov. Testi so namenjeni validaciji programske opreme in s programom komunicirajo "od zunaj" preko uporabniškega vmesnika, s klicem njegovih spletnih servisov, s simuliranjem zunanjih sistemov s pošiljanjem sporočil ... Tovrstne teste imenujemo funkcionalni testi in jih uvrščamo med vrste testov "črne skrinjice". Za njih je značilno, da z njimi preverjamo le funkcionalnost programa. Zanima nas, če program za določen vhod vrne pravilen izhod, ne zanima pa nas njegovo notranje delovanje. Ta njihova lastnost predstavlja prednost, saj simulirajo uporabo sistema, kot ga bo uporabljal končni uporabnik. Dejstvo, da so avtomatizirani, omogoča njihovo ponovno uporabo [11].

2.5 Regresijsko testiranje

Regresija v splošnem pomeni vračanje na nižjo razvojno stopnjo, nazadovanje. V primeru programske opreme to pomeni, da funkcionalnost, ki je že delovala, ne deluje več. Povzročamo jo programerji, v primeru TDD v 3. koraku cikla "testiraj-kodiraj-preoblikuj" (glej poglavje 3), ko preoblikujemo kodo, ki je že prestala test. In ker to počnemo namerno, želimo biti obveščeni čim prej, ko kaj pokvarimo. To najlažje dosežemo s pomočjo množice testov, ki jih redno izvajamo in nas obvestijo o napakah. To tehniko imenujemo regresijsko testiranje in jo lahko izvajamo na katerikoli stopnji testiranja. Prednost avtomatizacije testiranja je tu očitna. Avtomatsko regresijsko testiranje je

koristno med razvojem programske opreme, še bolj pa v fazi vzdrževanja, ko je sistem v uporabi in ga je treba spreminjati, tako da so spremembe za uporabnike čim manj moteče [17, 7].

2.6 Stresni testi

Stresni testi testirajo obnašanje programske opreme pod velikimi obremenitvami. Z njimi testiramo robustnost programske opreme tako, da jo obremenimo bolj, kot je zanjo to pričakovano. Tovrstno testiranje je še posebej pomembno pri kritičnih projektih [29].

Poglavje 3

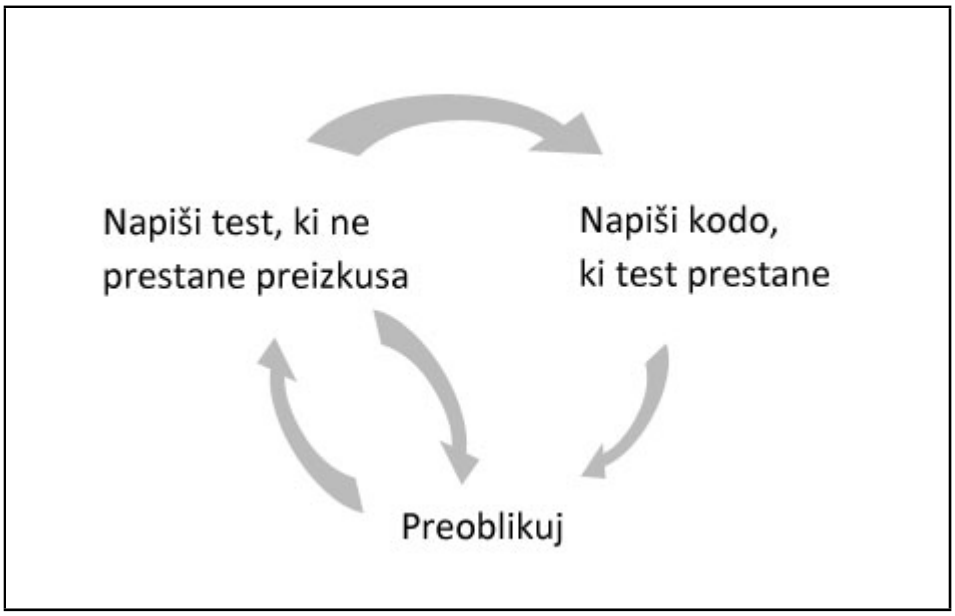
Testno voden razvoj (TDD)

Testno voden razvoj je ena izmed temeljnih tehnik ekstremnega programiranja (ang. Extreme programming). Je pristop k razvoju programske opreme, katerega glavna ideja je, da pred implementacijo neke funkcionalnosti zanjo napišemo test, ki bo to funkcionalnost testiral. Razvoj aplikacije poteka v majhnih iteracijah. Vsaka nova iteracija se začne z novim testom, nato napišemo kodo, ki test prestane, in jo v naslednjem koraku čim bolj poenostavimo. Ta cikel, prikazan na sliki 3.1, imenujemo *testiraj-kodiraj-preoblikuj* (ang. test-code-refactor) oz. *rdeča-zelena-preoblikuj* (ang. red-green-refactor).

- Rdeča — najprej napišemo test, ki sam po sebi ne more uspeti, ker funkcionalnost še ni implementirana. V nekaterih razvojnih okoljih to pomeni, da se nam prikaže rdeče okence. Od tu beseda *rdeča* v imenu cikla.

- Zelena — napišemo kodo, ki test prestane. Ko zaženemo teste, morajo poleg novega testa test prestati tudi vsi, ki smo jih napisali pred tem. Ob tem se prikaže zeleno okence, zato *zelena*.

- Preoblikuj — ko koda deluje, jo preoblikujemo, da postane preprostejša in preglednejša ter odstranimo podvojeno kodo. To, da s preoblikovanjem nismo pokvarili delovanja, nam zagotavljajo testi.



Slika 3.1: Cikel testno vodenega razvoja

Najprej napiši test

Ko v prvem koraku cikla TDD napišemo test, v resnici počnemo veliko več. Oblikujemo programski vmesnik (API) za dostop do enote, ki jo testiramo. S tem ko najprej napišemo test, razmišljamo o tem, kako želimo našo kodo uporabljati. Na ta način testi vodijo razvoj in strukturo naše kode.

Nato napiši ravno dovolj kode ...

V naslednjem koraku napišemo le toliko kode, da je test uspešno izpeljan. To je namreč naš cilj: zadostiti zahtevi testa. Ena izmed temeljnih idej TDD je, da naši testi narekujejo, kaj bomo implementirali v naslednjem trenutku in tako napredovali v razvoju. Ko napišemo čim manj kode, je naš glavni cilj, da test prestanemo čim hitreje. Pogosto naša implementacija ni najboljša, vendar za to poskrbimo v naslednjem koraku, ko bo naša zahteva že implementirana in cilj izpolnjen.

In preoblikuj

V zadnjem koraku cikla kodo preoblikujemo tako, da je čim bolj berljiva in odstranimo podvojeno kodo. Preoblikovanje kode je tehnika, s pomočjo katere preoblikujemo strukturo kode, ne da bi spremenili njeno funkcionalnost. Ali z besedami Martina Fowlerja, avtorja knjige Refactoring: Improving the Design of Existing Code [10] - preoblikovanje je *"a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."*

Ta korak je pomemben, saj z njim ohranjamo našo kodo "čisto" in obvladljivo, to pa vpliva na našo produktivnost. Testi, ki smo jih sprogramirali v 1. koraku cikla, nam omogočajo preoblikovanje brez skrbi, da bomo med tem nevede kaj "pokvarili" [17, 11].

3.1 Prednosti TDD

Zagovorniki testno vodenega razvoja trdijo, da nudi veliko prednosti pred drugimi načini razvoja. Nekatere prednosti so bolj očitne kot druge, vendar skupaj prispevajo k lažjemu in učinkovitejšemu razvoju.

Poglejmo si nekatere izmed prednosti, ki jih prinese TDD.

- **Preprost, inkrementalen razvoj**: Razvoj pri TDD poteka v kratkih iteracijah. Glavna prednost tega je, da imamo skoraj vsak trenutek delujoč program, četudi še nima implementiranih veliko funkcionalnosti [11].

- **Osredotočenost**: Programerji so produktivnejši, saj se morajo osredotočiti le na to, da sprogramirajo majhen košček kode, ki bo prestal test, nato pa začnejo z novo iteracijo oz. testom [3].

- **Neprestano regresijsko testiranje**: Prepreči simptom domin, da ima sprememba v enem modulu nepredvidljive posledice drugje v projektu. Ker teste neprestano izvajamo, napake opazimo zelo hitro in jih zato lažje odpravimo [11].

- **Dokumentacija**: Včasih imamo težave z razumevanjem kode, ki so jo napisali drugi, čeprav je "čista" in dobro dokumentirana. Testi vsebujejo del informacije o strukturi kode in nam nudijo pogled na enoto z drugega vidika. Sestavljajo namreč seznam zahtev enote. Z boljšim razumevanjem tako lažje spreminjamo kodo, ne da bi povzročili hrošče [11].

- **Zanesljivost delovanja programa**: V študiji *Realizing quality improvement through test driven development: results and experiences of four industrial teams* [22] so avtorji v primeru ekipe podjetja IBM ugotovili, da je ekipa, ki je uporabljala TDD, "proizvedla"kar 40 % manj hroščev, kot ekipa, ki je za testiranje uporabljala "ad-hoc" pristop. Bistvene razlike v porabljenem času niso opazili. Tudi ostale 3 ekipe, ki so jih primerjali, so dosegle malo slabše, a podobne rezultate.

- **Zaupanje**: V članku *Software Architecture Improvement through Test-Driven Development* [16] trdijo, da se zaradi pogostega izvajanja testov programerji počutijo samozavestnejše in lažje posegajo po večjih spremembah v kodi. Razlog za to je v zaupanju, da jih bodo testi opozorili, če bodo med spreminjanjem naredili kakšno napako.

Čeprav so vse te pozitivne lastnosti pomembne, pa je najpomembnejša izmed njih kvalitetnejša koda [3]. Namen TDD je v prvi vrsti ta, da pomaga pri pisanju "čiste", delujoče kode. V nadaljevanju si bomo ogledali, kako in zakaj pripomore k temu cilju.

3.2 Vpliv TDD na kvaliteto kode

TDD zahteva discipliniran proces razvoja programske opreme, s katerim se lažje izognemo "pastem"razvoja, kot je npr. "špagetasta koda" (ang. spaghetti code). Vzrok je v ideji, da pišemo kratke, avtomatizirane teste, s pomočjo katerih počasi zgradimo močan alarmni sistem, ki našo kodo varuje pred regresijo. Mnenja o tem, kaj je kvalitetna koda, so deljena. Nekateri

pravijo, da je ta odvisna od števila hroščev, najdenih med uporabo programskega sistema, drugi menijo, da se kvaliteta kode izraža v tem, kako dobra je uporabniška izkušnja, spet tretji, da je poleg uporabniške izkušnje pomembna tudi notranja zgradba, saj se ta posredno preslika v ceno razvoja, ceno vzdrževanja [17] ...

V našem primeru si želimo, da bi bile metrike čim bolj objektivne, zato smo jih izbrali tako, da jih bomo lahko izmerili. Pri razvoju naše aplikacije smo se odločili, da bomo za merjenje kvalitete kode uporabili orodje Sonar [27], ki v kodi preverja 7 ravni kvalitete kode:

- arhitekturo in strukturo kode,
- podvojenost kode,
- teste enot,
- kompleksnost kode,
- komentarje v kodi,
- število potencialnih hroščev,
- obliko kode (ang. coding rules).

3.3 Slabosti TDD

Kljub številnim prednostim ima TDD tudi nekaj pomanjkljivosti.

- Večina nasprotnikov in zagovornikov TDD priznava, da TDD zahteva veliko učenja in znanja, kar mnoge razvijalce programske opreme odvrača od uporabe [23].
- TDD upočasni razvoj. V študiji, ki sta jo opravila IBM in Microsoft, so ugotovili, da se je čas razvoja podaljšal od 15 % do 35 % [22].

- O učinkovitosti TDD obstaja preveč naznank in nasprotujočih si mnenj. [23].
- Začetni vložek časa in denarja je relativno visok [23].
- Jamie Zawinski, Lisp heker in Netscapeov razvijalec, pravi, da testi niso ključni za razvoj projekta in se jim lahko v časovni stiski odpovemo, saj nas pri razvoju programa le upočasnjujejo [25].
- Joshua Bloch, glavni arhitekt Jave pri podjetju Google, meni, da testi nikakor niso sprejemljiva zamenjava dokumentacije. Trdi, da ko želiš uporabljati kodo, ki jo je napisal nekdo drug, potrebuješ natančno specifikacijo, testi pa naj testirajo, da koda res deluje tako, kot to zahteva specifikacija [25].
- Nujno je, da vodstva podpira TDD. Če vanj ne verjame, se mu bo čas, porabljen za pisanje testov, zdel zapravljen [36].
- David Heinemeier Hansson, izumitelj ogrodja (ang. framework) Ruby On Rails, opozarja, da je pomembno, da se programer zaveda, katere dele kode je vredno testirati, drugače lahko preveč časa izgubimo s testiranjem kode [14].

Poglavje 4

Okolje in orodja

Sedaj, ko vemo, kaj je TDD, si poglejmo, kako ga vključimo v razvoj. Ključni del TDD je avtomatizacija procesa prevajanja in testiranja kode z uporabo ustreznih orodij. Brez njih lahko postaneta implementacija in vzdrževanje procesa TDD časovno zahtevna in težavna. Uporaba prevajalskih skript (ang. build script) je pogosta, vendar pa dober TDD-proces z uporabo skript, ki vključijo proces prevajanja in testiranja v proces razvoja, avtomatizacijo pelje še korak dlje. Po tehtnem premisleku smo se odločili za uporabo naslednjih orodij.

4.1 Java Enterprise Edition

Razvijalci se vedno bolj zavedamo potreb po porazdeljenih sistemih in transakcijah, prenosljivih aplikacijah, ki se zanašajo na hitrost, varnost in zanesljivost strežniških tehnologij. Poslovne aplikacije pogosto komunicirajo z drugimi poslovnimi aplikacijami. Od njih se pričakuje, da so hitre, stabilne, varne in potrebujejo čim manj virov. Java Enterprise Edition (Java EE) [24] platforma olajša razvoj takih aplikacij. Njen cilj je, da razvijalcem ponudi močan nabor programskih vmesnikov (ang. API) in skrajša čas razvoja, zmanjša kompleksnost aplikacij in izboljša zmogljivost. Ponuja sistemsko neodvisnost, kar ponazarja tudi njihov moto "Write once, run anywhere".

Aplikacije Java EE se izvajajo znotraj aplikacijskega strežnika. Ta omogoča varnost, transakcije, JNDI (ang. Java Naming and Directory Interface), JDBC (ang. Java Database Connectivity), zagotavlja življenjski cikel EJB-jem in servletom ...

Namesto datotek XML lahko uporabljamo anotacije, s katerimi lahko vstavimo potrebne informacije neposredno v izvorno kodo aplikacije. Strežnik Java EE nato določi konfiguracijo komponent glede na informacijo, ki jo vsebujejo anotacije.

4.1.1 Arhitektura in plasti aplikacij Java EE

Kompleksnost aplikacij Java EE je privedla do organizacije aplikacij na več logičnih, lahko pa tudi fizičnih plasti. Razdelitev na plasti je ena izmed najpogostejših tehnik, ki jih programerji uporabljamo. Plasti, ki so med seboj logično ločene, so lahko ločene tudi fizično. Ena izmed najbolj pogosto uporabljenih arhitektur aplikacij Java EE je 4-plastna arhitektura (ang. 4-tier[1] arhitecture):

- **plast podatkovne baze** — najnižjo plast aplikacije predstavlja podatkovna baza, s pomočjo katere hranimo podatke;
- **podatkovna plast** — odgovorna za komunikacijo s podatkovno bazo, sporočilnimi sistemi itd.;
- **plast poslovne logike** — vsebuje poslovno logiko aplikacije;
- **predstavitvena plast** — odgovorna za prikaz informacij, upravljanje uporabnikovih zahtev (npr. klikov z miško, pritiskov tipk na tipkovnici ...).

S tako arhitekturno zasnovo je testiranje naše kode lažje [9].

[1] V angleški literaturi namesto izraza layer pogosto zasledimo tudi izraz tier. Pojavlja se veliko zmede glede izrazov layer in tier. Nekateri trdijo, da sta izraza sinonima, drugi, da se razlikujeta. Prvi naj bi pomenil logično ločitev, drugi pa fizično. V diplomski nalogi sem v skladu z [9] privzel izraz plast, ki pomeni, da je plast logično, lahko pa tudi fizično ločena od ostalih plasti.

4.2 Eclipse

Eclipse [8] je odprtokodno razvojno orodje (urejevalnik), ki razvijalcu na različne načine olajša razvijanje programske opreme. Med drugim podpira tudi preoblikovanje in razhroščevanje. Brez tovrstnega orodja bi bil razvoj velikih projektov praktično nemogoč.

4.3 Subversion — SVN

Odprtokodni sistem za nadzor verzij SVN [30] skrbi za nadzor in pregled različnih verzij datotek. V svojem repozitoriju hrani trenutno verzijo in vse prejšnje verzije datotek, ki smo jih vanj shranili. Omogoča, da brskamo po verzijah in pregled nad metapodatki, kot so npr. kdo je dodal neko verzijo v repozitorij, kakšne spremembe je dodal ...

4.4 Maven

Apache Maven [19] je orodje za upravljanje projekta. Preko datoteke pom.xml (ang. project object model) upravlja zaganjanje projekta, generiranje poročil in dokumentacije.

Mavenov primarni cilj je razvijalcu na preprost način omogočiti nadzor nad stanjem projekta, ki ga razvija. Obstaja več področij, ki jih poskuša reševati:

- poenostaviti grajenje in izvajanje projekta,
- zagotoviti enoten sistem za upravljanje projekta,
- zagotoviti kvalitetne informacije o projektu,
- ponuja smernice za najboljše prakse ...

4.5 JBoss AS7

JBoss aplikacijski strežnik (ang. JBoss Application Server) [31] je certificirana platforma za izvajanje aplikacij Java EE. Uporabljali bomo verzijo 7.1. Strežnik podpira vrsto funkcij, ki jih nudi JEE. Nekatere izmed njih so:

- Enterprise JavaBeans,
- Hibernate integracijo,
- Java Naming and Directory Interface (JNDI),
- Java Server Pages (JSP),
- JBossWS (JBoss Web Services) za Java EE spletne servise, npr. JAX-WS,
- Java Database Connectivity (JDBC) ...

4.6 Podatkovna baza H2

H2 [13] je odprtokodna relacijska podatkovna baza SQL[2], napisana v Javi. Podatkovna baza H2 lahko deluje v obstojnem načinu (angl. persistent) ali "v spominu"(angl. in-memory). Poleg tega podpira različne načine povezovanja:

- **Vgrajen način** - z uporabo lokalne JDBC-povezave. Ta način je najpreprostejši in najhitrejši. V njem aplikacija odpre povezavo do baze v istem Java navideznem stroju (ang. Java Virtual Machine). Zaradi preprostosti smo ga v kombinaciji z načinom "v spominu"uporabili v testnem okolju.

[2]Strukturirani povpraševalni jezik za delo s podatkovnimi bazami (angl. Structured Query Language) je najbolj razširjen in standardiziran povpraševalni jezik za delo s podatkovnimi zbirkami, s programskimi stavki, ki posnemajo ukaze v naravnem jeziku [2].

- **Strežniški način** - z uporabo oddaljene (ang. remote connection) povezave JDBC ali ODBC preko TCP/IP.
- **Mešani način** - z uporabo lokalne in oddaljene povezave.

4.7 Podatkovna baza MySQL

Prav tako kot H2 je tudi MySQL [21] relacijska podatkovna baza. Za dostop do baze oz. za povpraševanje uporablja jezik SQL in je ena izmed najbolj popularnih podatkovnih baz, ki se uporabljajo za spletne aplikacije. Zanjo smo se odločili, ker je brezplačna in jo je preprosto izvajati na osebnem računalniku.

4.8 Hibernate

Hibernate [2] je odprtokodna implementacija programskega vmesnika JPA (ang. Java Persistence API), ki ga določa specifikacija Java EE. Omogoča, da naša aplikacija upravlja s podatki, ki jih hranimo v relacijski podatkovni bazi.

Hibernate uporablja jezik za pisanje poizvedb, imenovan HQL (ang. Hibernate Query Language), ki je podoben jeziku SQL. V primerjavi s SQL je HQL objektno orientiran in omogoča dedovanje, polimorfizem ...

4.9 JUnit

JUnit [17] je ogrodje za testiranje enot v Javi. Predstavlja *de facto* orodje za testiranje in zaradi podobnosti z ogrodji v drugih programskih jezikih sodi v skupino *xUnit* ogrodij. Vsa ogrodja, ki sodijo v to skupino, podajajo kodo, ki omogoča pisanje testov za testiranje enot, njihovo izvajanje in poročanje rezultatov. JUnit ogrodje nudi to podporo s pomočjo množice osnovnih razredov, ki jih razvijalci razširijo (ang. extend), in množico razredov in vmesnikov za izvajanje nalog, ki se pogosto uporabljajo. Za izvajanje

testov enot, napisanih z uporabo ogrodja JUnit, ogrodje ponuja razrede za zaganjanje, ki znajo zbrati množico testov enot, jih zagnati, izluščiti njihove rezultate in jih razvijalcu prikazati v obliki grafičnega ali tekstovnega poročila.

4.10 Mockito

Kot smo že omenili v poglavju 2.1, pri testiranju enot testiramo izolirane dele kode. Ker večina enot sodeluje z drugimi enotami, moramo v testu simulirati njihovo komunikacijo z drugimi enotami, da jih lahko testiramo v izolaciji.

Navidezni objekt je testno orientirana zamenjava za enoto, s katero testirana enota komunicira. Njegova naloga je, da simulira objekt, s katerim smo ga zamenjali.

Mockito [20] je odprtokodna knjižnica, ki omogoča preprosto uporabo navideznih objektov. Ti simulirajo obnašanje drugih delov kode in hkrati preverjajo, da se njihova uporaba sklada z njihovo definicijo. Mockito navidezne objekte generira dinamično in s tem programerju olajša delo ter zmanjša možnost napak pri testiranju. Ker objekte generira dinamično, ne generira kode in eliminira potrebo po ročnem pisanju navideznih objektov.

4.11 Arquillian

Kot smo omenili v poglavju 4.1, aplikacije JEE "živijo" znotraj aplikacijskega strežnika. Ker pri testiranju enot preverjamo delovanje kode v izolaciji, ne vemo, če bo pravilno delovala v interakciji z njegovimi (in drugimi) servisi. Poleg tega aplikacijski strežnik nekatere funkcionalnosti, kot so vbrizg odvisnosti (ang. dependency injection oz. DI), nadzor transakcij itd., zagotovi šele ob zagonu aplikacije (ang. at runtime).

Za testiranje tovrstnih odvisnosti uporabljamo integracijske teste. Arquillian [1] omogoča pisanje integracijskih testov, ki se izvedejo znotraj aplikacijskega strežnika. Omogoča nam, da prav tako kot teste enot tudi inte-

gracijske teste programiramo z uporabo knjižnice JUnit in jih tudi izvajamo kot navadne teste enot.

4.12 Selenium

Selenium [28] je orodje za avtomatizacijo brskalnikov in se uporablja za pisanje funkcionalnih testov spletnih aplikacij. Njegova naloga je, da avtomatizira nadzor nad brskalnikom. S tem omogoči izvajanje ponavljajočih se nalog, npr. izvajanje testov. Sestavljajo ga tri orodja:

- Selenium IDE: Je dodatek za brskalnik Firefox, ki uporabniku omogoča, da posname in kasneje izvaja posnete teste. Zaradi preprostosti, ki jo ponujata snemanje in izvajanje testov, je njegova uporabnost omejena in pogosto ne zadošča potrebam zahtevnih uporabnikov.

- Selenium WebDriver: Je drugo orodje, ki ponuja programski vmesnik (ang. API) v več programskih jezikih in omogoča več nadzora in uporabo v standardnih procesih razvoja programske opreme.

- Selenium Grid: Omogoča uporabo Seleniumovih programskih vmesnikov, ki so porazdeljeni po več računalnikih. Tako lahko vzporedno izvajamo več testov naenkrat.

4.13 Jenkins

Jenkins [26, 32] je odprtokodno orodje za podporo sprotni integraciji, napisano v Javi. Je strežniško temelječ sistem, ki ga poganjajo aplikacijski strežniki, npr. Tomcat ali JBoss. Podpira različne programske jezike, med njimi tudi Javo. Osnovno različico lahko razširimo z različnimi dodatki, npr. z dodatki za podporo sistemov za nadzor verzij izvorne kode, z orodji za zagotavljanje kakovosti kode, z integracijo z zunanjimi sistemi. Z njim lahko izvajamo Apache ANT in Apache Maven skripte. Sprotna integracija predstavlja enega

izmed temeljev TDD, zato bo Jenkins nepogrešljivo orodje v našem razvoju. Več o sprotni integraciji bomo povedali v poglavju 6.

4.14 Sonar

Sonar [34] je odprtokodno orodje za zagotavljanje kakovosti kode. Uporablja različna statična orodja za analizo kode, kot sta Checkstyle in FindBugs. Z njihovo pomočjo nato Sonar pridobi metrike o kvaliteti izvorne kode, ki jih lahko programerji uporabimo za izboljšanje kakovosti kode.

Sonar ponuja poročila o podvojeni kodi, upoštevanju pravil kodiranja (ang. coding standards), pokritosti kode s testi enot (ang. code coverage), kompleksni kodi, potencialnih hroščih, komentarjih, arhitekturi in strukturi kode.

Poglavje 5

Tehnike testiranj in uporaba testov na različnih nivojih aplikacije

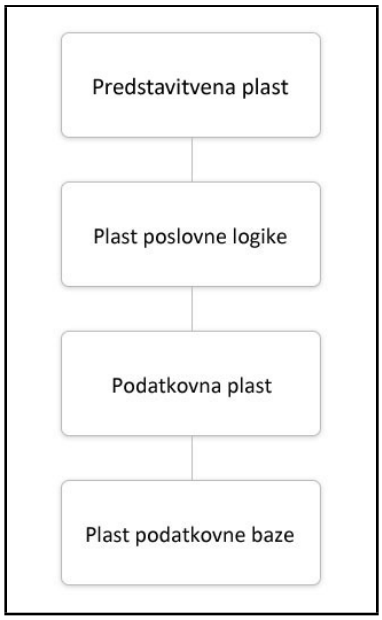
Kot smo omenili že v poglavju 4.1.1, kompleksne aplikacije programerji pogosto razdelimo na več plasti oz. nivojev. V tem poglavju bomo na primeru preproste Java EE-aplikacije FerApp prikazali, kako z uporabo različnih orodij in pristopom testno vodenega razvoja testiramo različne plasti.

FerApp je aplikacija, ki je namenjena reševanju problema *"kdo je na vrsti"*. Omogoča, da se uporabniki registrirajo in prijavijo v aplikacijo ter se združijo v skupine. Uporabljali naj bi jo ljudje, ki nekaj pogosto počnejo skupaj (npr. sodelavci se skupaj vozijo na malico in se morajo dogovoriti, kdo je na vrsti za vožnjo). Člani skupine nato s pomočjo aplikacije beležijo statistiko (npr. kolikokrat je kdo šel na malico in kolikokrat je bil v vlogi voznika). Aplikacija nato s pomočjo pristopa statistike, ki jo hrani v podatkovni bazi MySQL, predlaga, kdo je na vrsti za določeno akcijo (npr. kdo je na vrsti za vožnjo na malico).

Čeprav je FerApp sestavljena iz več modulov, bomo zaradi lažjega razumevanja v nalogi prikazali načine testiranja le na poenostavljenih odsekih kode iz modula *Uporabniki*. Tehnike se namreč ponavljajo tudi v ostalih modulih

in menimo, da bo bralec lažje sledil nalogi, če se bodo primeri kode navezovali eden na drugega.

Na sliki 5.1 je prikazana osnovna arhitekturna zasnova aplikacije.



Slika 5.1: Štiriplastna zasnova aplikacije Java EE

5.1 Plast podatkovne baze

Najnižjo plast aplikacije predstavlja relacijska podatkovna baza (ang. Relational Database Management System oz. RDBMS), ki jo bomo testirali posredno z uporabo integracijskih testov. Te bomo sprogramirali za testiranje višjih nivojev v arhitekturi aplikacije. Nekateri izmed bolj popularnih RDBMS-sistemov so Oracle, MySQL, MS SQL ...

5.2 Podatkovna plast

Druga plast predstavlja visokonivojsko abstrakcijo persistenčnih podatkov. Na tem nivoju nam Java EE nudi vmesnik JPA. Z njim omogoča, da lahko

z zelo malo konfiguracije preprosto zamenjamo RDBMS-sistem. Ta nivo je neposredna preslikava podatkovne baze in njenih relacij, zato nam lahko kodo na tem nivoju zgenerirajo napredna integrirana razvojna okolja (ang. Integrated Development Enviroment oz. IDE) kar sama. Razrede na tem nivoju imenujemo entitete in želimo, da koda na tej plasti ostane brez dodatne logike. S tem dosežemo, da entitete vsebujejo le lastnosti (ang. properties) in metode, s katerimi dostopamo do njih (ang. getters/setters), ter anotacije. Zaradi enostavnosti entitet jih zato ni potrebno testirati, saj bi s tem testirali kodo, ki jo je generiralo razvojno okolje.

5.3 Plast poslovne logike

Tretji nivo, nivo poslovne logike, je najpomembnejši. Velja, da aplikacija obstaja samo zaradi poslovne logike, ki jo ta plast implementira. Za testiranje tega nivoja smo uporabili knjižnico JUnit za testiranje enot. Aplikacijo smo testirali na dva načina. S pomočjo testov enot smo vodili njen razvoj, ko pa smo določen del funkcionalnosti sprogramirali, smo sprogramirali še integracijske teste.

Eden izmed modulov aplikacije je modul uporabnikov, ki omogoča registracijo, prijavo, pregled uporabnikov ... V podatkovni bazi smo ustvarili potrebne tabele in iz njih, s pomočjo integriranega razvojnega okolja Eclipse, generirali entitete.

5.3.1 Testi enot

Želeli smo razviti metodo `findAllOrderedByName()`, s katero bomo iz baze prebrali vse uporabnike, urejene po abecednem vrstnem redu. Najprej smo sprogramirali test, ki preverja njeno delovanje. Ker teste vedno pišemo za najmanjši zaokrožen del kode programa, je test v odseku 5.1 preprost, vendar pa je za uporabo navideznih objektov potrebnih nekaj dodatnih vrstic kode.

Navidezne objekte uporabimo, kadar znotraj enote, ki jo testiramo, uporabimo metode drugih razredov. V testu predpostavimo, da so metode razre-

dov, ki jih uporabljamo znotraj naše metode, že preverjene, zato njihovega delovanja ne testiramo, ampak ga simuliramo z uporabo navideznih objektov. Glavna ideja testiranja je, da izvedemo neko metodo in vrednost, ki jo vrne, primerjamo s pričakovano vrednostjo. V primerih, kjer uporabimo navidezne objekte in s tem sami definiramo vrnjeno vrednost, se prepričamo, da se tista metoda res izvede. V testu metode `findAllOrderedByName()` smo s pomočjo ogrodja Mockito definirali, da navidezni objekt `Query` ob klicu metode `query.getResultList()` vrne seznam, ki vsebuje dva objekta tipa `User`. Ta seznam predstavlja pričakovano vrednost, ki jo kasneje primerjamo z dobljeno vrednostjo.

Ker gre za test enote, želimo z njim preveriti, da je metoda `findAllOrderedByName()` implementirana pravilno. Preveriti moramo, da se ob njenem klicu znotraj nje kličeta metodi `entityManager.createNamedQuery()` in `query.getResultList()` ter da metoda `entityManager.createNamedQuery()` kot argument prejme pravi stavek SQL - `User.ALL_ORDERED_BY_NAME`.

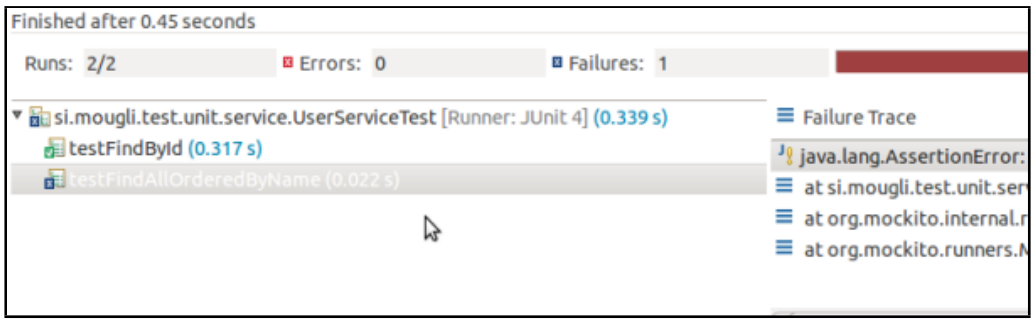
```
1
2  @RunWith(MockitoJUnitRunner.class)
3  public class UserServiceTest
4  {
5    // Z anotacijo @InjectMock obvestimo Mockito,
6    // da je to objekt, za katerega bo generiral navidezne objekte.
7    @InjectMocks
8    private UserService userService;
9
10   // To je objekt, ki ga bo potrebno "ponarediti" - (@Mock).
11   @Mock
12   private EntityManager entityManagerMock;
13
14   @Mock
15   private Query queryMock;
16
17   private List<User> users;
18   private User user;
19
```

```
20  // Zaradi @Before anotacije se metoda setUp pozene vsakic,
21  // ko se pozene nov test.
22  @Before
23  public void setUp()
24  {
25    // Rezultati, ki jih vracajo navidezni objekti.
26    user = EntitiesGeneratorHelper.generateUser(1, "Janez", "Novak");
27    User user2 = EntitiesGeneratorHelper.generateUser(2, "Toncek", "
      Baloncek");
28
29    users = new ArrayList<User>();
30    users.add(user);
31    users.add(user2);
32
33    // ... manjka koda - mock nastavitve za ostale metode ...
34
35    // Definiramo pravilo ob klicu createNamedQuery
36    Mockito.when(entityManagerMock
37      .createNamedQuery(User.ALL_ORDERED_BY_NAME))
38      .thenReturn(queryMock);
39    Mockito.when(queryMock.getResultList()).thenReturn(users);
40  }
41
42    // ... manjka koda - ostale metode
43
44  /**
45   * Testiraj, da metoda UserService.findAllOrderedByName()
46   * klice metodo EntitiManager.createNamedQuery() s pravim stavkom HQL
47   * in vrne seznam entitet User.
48   */
49  @Test
50  public void testFindAllOrderedByName()
51  {
52    List<User> result = userService.findAllOrderedByName();
53
54    // Prepricajmo se, da se klice prava metoda
55    assertEquals(users, result);
56
```

```
57      // Preverimo, da se klice pravi stavek HQL in da se metoda
58      // getResultList() klice natancno enkrat.
59      Mockito.verify(entityManagerMock)
60        .createNamedQuery(
61          Mockito.eq(User.ALL_ORDERED_BY_NAME));
62      Mockito.verify(queryMock, Mockito.times(1))
63      .getResultList();
64    }
65  }
```

Izvorna koda 5.1: Test metode findAllOrderedByName()

Ko smo napisali test, ga zaženemo, da se prepričamo, ali test vrne napako. Na sliki 5.2 vidimo, da test ni uspel.



Slika 5.2: Rezultat testa, ko metoda `findAllOrderedByName()` še ni implementirana

Ker test kode ni bil uspešen, smo lahko nadaljevali z implementacijo kode, ki bo test prestala. Ker smo za metodo že napisali test, smo vedeli, da moramo znotraj metode `findAllOrderedByName()` klicati metodo `EntityManager.createNamedQuery()` in `Query.getResultList()`, zato je bila implementacija preprosta. Napisali smo kodo, ki je prikazana v odseku 5.2.

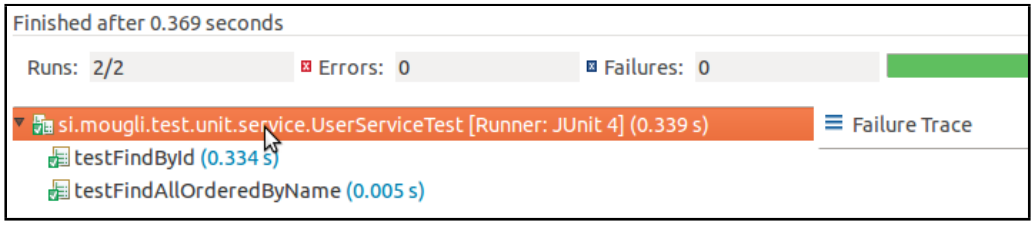
```
1  @Entity
2  @Table('users')
3  // Dodamo stavek HQL.
```

```
4  @NamedQueries(value = { @NamedQuery(name = User.ALL_ORDERED_BY_NAME, query
       = "SELECT u FROM User u ORDER BY u.name ASC") })
5  public class User implements Serializable
6  {
7    // ... manjka koda entitete User
8  }
9
10 @Stateless
11 public class UserService
12 {
13   // ... manjka koda - ostale metode razreda
14
15   public List<User> findAllOrderedByName()
16   {
17     @SuppressWarnings("unchecked")
18     List<User> users = em.createNamedQuery(User.ALL_ORDERED_BY_NAME).
       getResultList();
19
20     return users;
21   }
22
23   // ... manjka koda - ostale metode razreda
24 }
```

Izvorna koda 5.2: Implementacija metode findAllOrderedByName()

Koda v odseku 5.2 je zelo preprosta, zato je v tretjem koraku cikla TDD (preoblikovanje) nismo poenostavili. Preden smo se lahko lotili programiranja ostalih funkcionalnosti, smo preverili, če je naša implementacija pravilna in če prestane test.

Primer, ki smo ga prikazali zgoraj, je preprost, v realnosti pa koda, s tem pa tudi testiranje, hitro postane bolj zapletena. V odseku 5.3 je prikazan test metode `register()`, ki je v osnovi dokaj preprosta, vendar pa ne vrača ničesar. Kot smo videli v primeru 5.1, smo predvideli, kakšen rezultat bo metoda `findAllOrderedByName()` vrnila, jo izvedli in primerjali s predvi-



Slika 5.3: Rezultat testa metode `findAllOrderedByName()`

deno vrednostjo. Pri metodah, ki ne vračajo ničesar, tega ne moremo narediti. Take metode pogosto spreminjajo lastnosti objektov. V našem primeru metodi podamo objekt User. Ko objekt shranimo v podatkovno bazo, pridobi unikatni identifikator (njegova lastnost se spremeni). Knjižnica Mockito omogoča tudi testiranje tovrstnih metod. Za reševanje takih problemov ponuja metodo `Mockito.doAnswer()`, s pomočjo katere lahko spremenimo stanje objektu, kot bi ga spremenila izzvana metoda.

```
1  public class UserServiceTest
2  {
3    // ... manjka koda - ostali testi
4
5    @Test
6    public void testRegister()
7    {
8      User usr = EntitiesGeneratorHelper.generateUser("Janez", "Novak");
9
10     // Pripravi odgovor.
11     Answer<Object> answer = createRegisterAnswer();
12     Mockito.doAnswer(answer).when(entityManagerMock).persist(usr);
13
14     // Pred klicem je id 0, po klicu 1 -> pravilna metoda se je izvedla.
15     assertEquals(0, usr.getId());
16     userService.register(usr);
17     assertEquals(1, usr.getId());
18
19     Mockito.verify(entityManagerMock, VerificationModeFactory.times(1))
```

```
20        .persist(usr);
21    }
22
23    /**
24     * Generira Answer objekt, ki ga Mockito vrne kot
25     * odgovor na klic metode register in simulira
26     * spremenjeno lastnost id objekta User.
27     *
28     * @return
29     */
30    private Answer<Object> createRegisterAnswer()
31    {
32      return new Answer<Object>()
33      {
34        @Override
35        public Object answer(InvocationOnMock invocation)
36        {
37          Object[] args = invocation.getArguments();
38          User usr = (User) args[0];
39          usr.setId(1);
40          return null;
41        }
42      };
43    }
44
45    // ... manjka koda - ostali testi
46  }
```

Izvorna koda 5.3: Testiranje "void"metode

Ko smo s funkcionalnostjo zaključili, smo sprogramirali še integracijske teste, da smo se prepričali, da naša koda deluje v okolju z ostalimi komponentami.

5.3.2 Integracijski testi

Pisanje integracijskih testov je zelo podobno pisanju testov enot, le da ne potrebujemo navideznih objektov, saj orodje Arquillian kodo izvede znotraj aplikacijskega strežnika. To omogoča, da v testih lahko komuniciramo s podatkovno bazo, uporabljamo anotacije ... Od navadnih testov enot se Arquillianovi testi razlikujejo v tem, da vsebujejo metodo, ki vrača objekt tipa `ShrinkWrap`, npr. `JavaArchive` ali `WebArchive`, in je anotirana z anotacijo `@Deployment`. V `ShrinkWrap` objekt zapakiramo vse datoteke, ki jih potrebujemo za zagon testa.

Java EE ponuja ločene nastavitve za teste in aplikacijo. Tako se lahko v testnem okolju povezujemo na drugo podatkovno bazo kot v aplikaciji. Zato smo se odločili, da bomo za testno okolje uporabili podatkovno bazo H2. Dodatna možnost, ki smo jo izkoristili, je sposobnost ogrodja Hibernate, da se ob zagonu v podatkovno bazo uvozijo vsi stavki SQL, ki se nahajajo v datoteki import.sql. Vse to olajša pripravo testnega okolja.

Da bo primerjava integracijskih testov s testi enot najlažja, smo v odseku 5.4 prikazali test za isto metodo kot pri prikazu testov enot - `findAllOrderedByName()`. Za testiranje metode `testFindAllOrderedByName()` smo potrebovali še manj vrstic kode kot za test enot. Poleg tega smo se s tem testom lahko prepričali, da vrne uporabnike v pravilnem vrstnem redu. Tega s testi enot nismo mogli preveriti, saj smo tam sami definirali, kakšen rezultat bo ob klicu vrnila določena metoda, z integracijskimi testi pa lahko podatke preberemo iz baze in tako testiramo tudi pravilnost stavka SQL.

```
1 @RunWith(Arquillian.class)
2 public class UserServiceTestIT
3 {
4   @EJB
5   private UserService userService;
6
7   @Deployment
8   public static JavaArchive createDeployment()
9   {
```

```
10      JavaArchive jar = ShrinkWrap.create(JavaArchive.class)
11          .addClasses(UserService.class, User.class, Resources.class)
12          .addClasses(Stat.class, Group.class, Role.class)
13          .addAsResource("META-INF/test-persistence.xml", "META-INF/
        persistence.xml")
14          .addAsResource("import.sql")
15          .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
16
17      // Izpisi vsebino jara.
18      System.out.println(jar.toString(true));
19      return jar;
20    }
21
22    @Test
23    public void testFindAllOrderedByName()
24    {
25      // iz baze preberi uporabnike, ki smo jih uvozili ob zacetku
26      List<User> users = userService.findAllOrderedByName();
27
28      assertEquals(2, users.size());
29      assertEquals(true, isOrderedByName(users));
30    }
```

Izvorna koda 5.4: Integracijsko testiranje metode findAllOrderedByName()

V primerjavi s testi enot je glavna slabost integracijskih testov ta, da za izvedbo potrebujejo več časa kot testi enot.

5.4 Predstavitvena plast

Četrto plast imenujemo predstavitvena plast in je odgovorna za prikaz informacij in upravljanje uporabnikovih zahtev (npr. klikov z miško, pritiskov tipk na tipkovnici ...).

Ker je naša aplikacija spletna aplikacija, smo morali informacije izpisati v obliki HTML, da jo brskalniki lahko prikažejo. Najprej smo izdelali poseben

Javanski razred, ki predstavlja vmesnik (ang. Controller) med uporabnikom in plastjo poslovne logike. Za prikaz vsebin smo uporabili tehnologijo JSF 2.0.

5.4.1 Testiranje vmesnika

Za testiranje vmesnika smo uporabili enaka ogrodja in orodja kot za testiranje plasti poslovne logike. Ker uporabljamo enaka orodja, se tudi večina tehnik testiranja v odseku 5.5 ponovi. Sprememba, ki bi jo želeli izpostaviti, se nahaja v metodi `setUp()`. V njej smo uporabili programski vmesnik Reflection, s katerim lahko dostopamo do privatnih lastnosti drugih razredov, do katerih drugače ne bi mogli dostopati. Razred `UserController` namreč vsebuje privatno lastnost `user`, ki smo jo morali nastaviti preko programskega vmesnika Reflection, da smo lahko stestirali metodo `register()`, saj metoda predvideva, da je lastnost nastavljena. Testirati smo želeli dva primera. V testu `testRegister()` smo želeli stestirati primer, ko registracija uspe. Takrat se uporabnik s pomočjo metode `UserService.register()` shrani v bazo in metoda `userController.register()` nam vrne niz, v katerem je podan URL, na katerega se preusmerimo po uspešni registraciji. V testu `testRegisterShouldFail()` pa pričakujemo, da registracija ne bo uspela. Ob neuspešni registraciji se želimo prepričati, da metoda vrne `null`.

```
1  @RunWith(MockitoJUnitRunner.class)
2  public class UserControllerTest
3  {
4    @InjectMocks
5    private final UserController userController = new UserController();
6
7    @Mock
8    private UserService userService;
9
10   @Mock
11   private RoleService roleService;
12
```

```
13  private User user;
14
15  /**
16   * Metoda poskrbi za inicializacijo potrebnih nastavitev.
17   *
18   * @throws SecurityException
19   * @throws NoSuchFieldException
20   * @throws IllegalArgumentException
21   * @throws IllegalAccessException
22   */
23  @Before
24  public void setUp() throws SecurityException, NoSuchFieldException,
      IllegalArgumentException, IllegalAccessException
25  {
26    user = EntitiesGeneratorHelper.generateUser("Toncek", "Baloncek");
27    Class<? extends UserController> clazz = userController.getClass();
28    Field f = clazz.getDeclaredField("user");
29    f.setAccessible(true);
30    f.set(userController, user);
31
32    Role role = EntitiesGeneratorHelper.generateRole("user", 1);
33    Mockito.when(roleService.getRoleUser()).thenReturn(role);
34
35    Answer<Object> answer = EntitiesGeneratorHelper
36      .createRegisterUserAnswer();
37    Mockito.doAnswer(answer).when(userService).register(user);
38
39    // ... manjka koda - ostale nastavitve
40  }
41
42  @Test
43  public void testRegister()
44  {
45    String forward = userController.register();
46
47    assertNotNull(forward);
48    assertEquals(1, user.getRoles().size());
49    assertEquals(1, user.getId());
```

```
50    }
51
52    @Test
53    public void testRegisterShouldFail()
54    {
55      Mockito.doThrow(new RuntimeException()).when(userService).register(
      user);
56
57      String forward = userController.register();
58      assertEquals(null, forward);
59    }
60
61    // ... manjka koda - ostali testi
62  }
```

Izvorna koda 5.5: Testiranje metode register() na predstavitveni plasti

Poleg dostopanja do privatnih lastnosti razreda programski vmesnik Reflection ponuja veliko drugih možnosti za dinamično "preiskovanje" in uporabo razredov.

V odseku 5.6 je implementacija kode, ki prestane oba testa.

```
1  public String register()
2    {
3      try
4      {
5        Role role = roleService.getRoleUser();
6        user.getRoles().add(role);
7        userService.register(user);
8      } catch (Exception e)
9      {
10       String msg = "Prislo je do nepricakovane napake! Registracija ni
       uspela!";
11       facesContext.addMessage(null, new FacesMessage(msg));
12       return null;
13     }
```

```
14
15      return "index.jsf";
16    }
```

Izvorna koda 5.6: Implementacija metode register na predstavitveni plasti

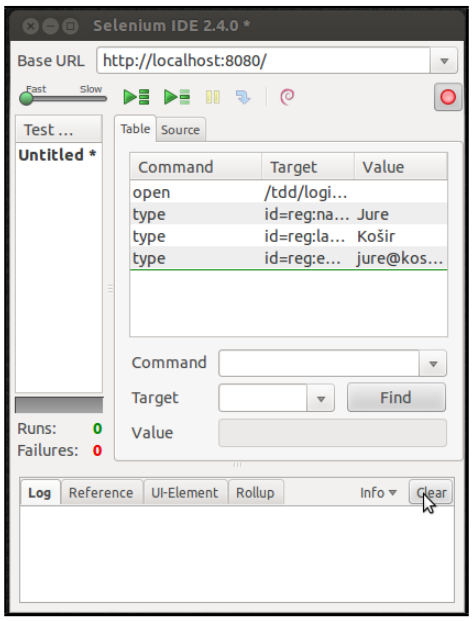
Tudi na tej plasti bi lahko izvedli integracijske teste z Arquillianom, vendar smo se odločili, da bomo za testiranje predstavitvene plasti raje uporabili orodje Selenium. Z njim opravljamo funkcionalne teste, kar nam omogoča, da hkrati testiramo obe komponenti predstavitvene plasti: prvo, ki je sprogramirana v Javi, in drugo, ki je realizirana z uporabo JSF 2.0.

5.4.2 Testiranje JSF 2.0

Za testiranje komponente JSF nismo uporabili načina razvoja TDD. Orodje Selenium omogoča, da teste posnamemo v brskalniku in jih izvozimo v kodo različnih programskih jezikov. Kodo lahko kasneje, če se izkaže, da je to potrebno, prilagodimo, saj Seleniumova komponenta WebDriver ponuja več funkcionalnosti kot komponenta Selenium IDE. Razlog, da smo se odločili za tak pristop, je, da bi v nasprotnem primeru, če bi teste pisali z uporabo tehnik TDD, morali predvideti sestavo dokumenta HTML (katere značke in atribute bo imel[1]), kar pa bi bilo preveč zamudno.

Na sliki 5.4 je prikazan uporabniški vmesnik komponente Selenium IDE, ki ga ponuja orodje Selenium, v odseku 5.7 pa koda, ki smo jo izvozili z njim in popravili, da bolj ustreza našim potrebam. Želeli smo stestirati, če naš uporabniški vmesnik dovoli registracijo le, če je uporabnik pravilno vnesel vse podatke in da v nasprotnem primeru prikaže prava obvestila o napakah. Najprej smo poskusili registrirati uporabnika, ne da bi vnesli podatke, nato z nepravilnimi podatki in na koncu uporabnika s pravilno vnešenimi podatki.

[1]Selenium izvaja akcije v brskalniku s pomočjo Javascripta tako, da se sprehaja po dokumentovem objektnem modelu (ang. Document Object Model oz. DOM), ki ga določajo značke in njihovi atributi [6].



Slika 5.4: Selenium IDE

```
1  public class TestRegisterIT
2  {
3    private WebDriver driver;
4    private final String baseUrl = "http://localhost:8080/tdd";
5
6    @Before
7    public void setUp() throws Exception
8    {
9      // Zazeni brskalnik Mozilla Firefox
10     driver = new FirefoxDriver();
11     driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
12
13     // in obisci naslov ...
14     driver.get(baseUrl + "/login.jsf");
15     driver.findElement(By.id("reg:register")).click();
16   }
17
18   @Test
```

```
19  public void testRegisterEmptyError() throws Exception
20  {
21    List<WebElement> errors = driver.findElements(
22      By.className("invalid"));
23    assertEquals(5, errors.size());
24
25    for (WebElement webElement : errors)
26    {
27      assertEquals("Polje je obvezno", webElement.getText());
28    }
29  }
30
31  @Test
32  public void testRegisterInvalidValues()
33  {
34    // Izpolnimo obrazec z napacnimi vrednostmi
35    // in pritisnemo gumb "Registriraj se!".
36    driver.findElement(By.id("reg:name")).clear();
37    driver.findElement(By.id("reg:name")).sendKeys("J");
38    driver.findElement(By.id("reg:lastname")).clear();
39    driver.findElement(By.id("reg:lastname")).sendKeys("K");
40    driver.findElement(By.id("reg:email")).clear();
41    driver.findElement(By.id("reg:email")).sendKeys("jure");
42    driver.findElement(By.id("reg:password")).clear();
43    driver.findElement(By.id("reg:password")).sendKeys("jure");
44    driver.findElement(By.id("reg:confirmPassword")).clear();
45    driver.findElement(By.id("reg:confirmPassword")).sendKeys("jure");
46    driver.findElement(By.id("reg:register")).click();
47
48    // Poisci elemente z napakami.
49    List<WebElement> errors = driver.findElements(
50      By.className("invalid"));
51
52    // Pricakujemo 4 napake.
53    assertEquals(4, errors.size());
54
55    // Preveri, ce so prave.
56    String nameError = "Dolzina imena mora biti med 3 in 20 znaki!";
```

```
57      String lastnameError = "Dolzina priimka mora biti med 3 in 20 znaki!";
58      String emailError = "Neveljaven e-postni naslov";
59      String passError = "Geslo mora biti dolgo vsaj 6 znakov in vsebovati
        crke in stevilke.";
60      assertEquals(nameError, errors.get(0).getText());
61      assertEquals(lastnameError, errors.get(1).getText());
62      assertEquals(emailError, errors.get(2).getText());
63      assertEquals(passError, errors.get(3).getText());
64    }
65
66    @Test
67    public void testRegister() throws Exception
68    {
69      int rand = 1000 + (int) (Math.random() * 9999);
70      String mail = "jure" + rand + "@kosir.si";
71      driver.findElement(By.id("reg:name")).sendKeys("Jure");
72      driver.findElement(By.id("reg:email")).clear();
73      driver.findElement(By.id("reg:lastname")).clear();
74      driver.findElement(By.id("reg:lastname")).sendKeys("Kosir");
75      driver.findElement(By.id("reg:email")).sendKeys(mail);
76      driver.findElement(By.id("reg:password")).clear();
77      driver.findElement(By.id("reg:password")).sendKeys("jure123");
78      driver.findElement(By.id("reg:confirmPassword")).clear();
79      driver.findElement(By.id("reg:confirmPassword")).sendKeys("jure123");
80      driver.findElement(By.id("reg:phoneNumber")).clear();
81      driver.findElement(By.id("reg:phoneNumber")).sendKeys("0409876543");
82      driver.findElement(By.id("reg:register")).click();
83
84      // ob oddaji obrazca se nalozi nova stran -> pocakaj
85      // da se nalozi do konca
86      new WebDriverWait(driver, 5)
87        .until(presenceOfElementLocated(By.id("greeting")));
88      WebElement element = driver.findElement(By.id("greeting"));
89
90      assertEquals("Pozdravljen, Jure!", element.getText());
91    }
92
93    @After
```

```
94   public void tearDown() throws Exception
95   {
96     driver.quit();
97   }
98 }
```

Izvorna koda 5.7: Selenium test registracije uporabnika

5.5 Preverjanje pravilnosti podatkov

Pomemben del vsake aplikacije je preverjanje pravilnosti podatkov. Predstavlja nalogo, ki se razteza čez vse plasti aplikacije, zato si zasluži, da ga opišemo ločeno. Java EE 6 ponuja programski vmesnik, imenovan Bean Validation, s katerim omogoča preprosto validacijo podatkov na različnih nivojih aplikacije. Z njim se izognemo nepotrebnemu ponavljanju in medsebojni odvisnosti (ang. tight coupling) kode. Omogoča, da pravila zapišemo v kodi v programskem jeziku Java in jih definiramo z anotacijami (meta podatki). Anotacije lahko nato uporabljamo na različnih mestih v aplikaciji [12].

Za validacijo smo poleg obstoječih anotacij, ki jih ponuja programski vmesnik Bean Validation, potrebovali pravilo za preverjanje pravilnosti gesla, ki ga uporabnik vpiše ob registraciji. Zanj smo sprogramirali lastno anotacijo. Želeli smo, da so gesla dolga vsaj 6 znakov in sestavljena vsaj iz črk in številk. Najprej smo napisali test v odseku 5.8, ki testira, da naša validacijska metoda sprejme pravilno tvorjeno geslo in zavrne geslo, ki ne ustreza pogojem:

```
1 @Test
2 public void testPasswordStrength()
3 {
4   PasswordStrengthValidator validator = new PasswordStrengthValidator();
5
6   boolean resultOK = validator.isValid(VALID_PASS, null);
7   boolean resultNotOK = validator.isValid(BAD_PASS, null);
8
```

```
9   assertTrue(resultOK);
10  assertTrue(!resultNotOK);
11 }
```

Izvorna koda 5.8: Test validatorja PasswordStrenghtValidator

Nato smo napisali še implementacijo kode, prikazane v odseku 5.9, in definirali anotacijo:

```
1  public class PasswordStrengthValidator implements ConstraintValidator<
       PasswordStrength, String>
2  {
3    @Override
4    public void initialize(PasswordStrength constraintAnnotation)
5    {
6    }
7
8    @Override
9    public boolean isValid(String pass, ConstraintValidatorContext context)
10   {
11     if (pass.length() < 6)
12     {
13       return false;
14     }
15
16     if (pass.matches(".*[a-zA-Z]+.*") && pass.matches(".*[\\d]+.*"))
17     {
18       return true;
19     }
20
21     return false;
22   }
23 }
24
25 @Constraint(validatedBy = PasswordStrengthValidator.class)
26 @Target({ ElementType.METHOD, ElementType.FIELD, ElementType.
       ANNOTATION_TYPE, ElementType.CONSTRUCTOR,
```

```
27      ElementType.PARAMETER })
28  @Retention(RetentionPolicy.RUNTIME)
29  public @interface PasswordStrength
30  {
31    String message() default "Geslo mora biti dolgo vsaj 6 znakov in
        vsebovati crke in stevilke.";
32
33    Class<?>[] groups() default {};
34
35    Class<? extends Payload>[] payload() default {};
36  }
```

Izvorna koda 5.9: Implementacija validatorja PasswordStrenghtValidator

Ker pravila na različnih mestih prirejamo v kodi preko anotacij, jih lahko neposredno testiramo le s pomočjo integracijskih testov. Za testiranje validacije metodi `validator.validate()` kot argumet podamo entiteto, nad katero želimo izvesti validacijo. Validator nam vrne množico napak, ki jo potem lahko preverimo z metodami ogrodja JUnit. V odseku 5.10 prikazujemo še metodo, s katero smo stestirali validacijska pravila entitete `User`:

```
1  @RunWith(Arquillian.class)
2  public class UserValidatorTest
3  {
4    // potrebujemo validator
5    @Inject
6    private Validator validator;
7
8    // ... manjka koda - @Deployment metoda
9
10   @Test
11   public void testConstraintsShouldFail()
12   {
13     User user = EntityGeneratorHelper.generateBadUser();
14     Set<ConstraintViolation<User>> violations = validator.validate(user);
15
```

```
16      // Pricakujemo 4 napake.
17      assertEquals(4, violations.size());
18    }
19
20    @Test
21    public void testPasswordStrenghtValidator()
22    {
23      User user = EntityGeneratorHelper.generateUser("Tina", "Maze");
24      user.setPassword("invalid");
25      Set<ConstraintViolation<User>> violations = validator.validate(user);
26      assertEquals(1, violations.size());
27
28      // Preveri, da gre res za napako gesla.
29      ConstraintViolation<User> constraint = violations.iterator().next();
30      assertEquals("invalid", constraint.getInvalidValue());
31      assertEquals("password", constraint.getPropertyPath().toString());
32    }
33
34    @Test
35    public void testConstraintsShouldPass()
36    {
37      User user = EntityGeneratorHelper.generateUser("Tina", "Maze");
38      Set<ConstraintViolation<User>> violations = validator.validate(user);
39
40      assertEquals(0, violations.size());
41    }
42  }
```

Izvorna koda 5.10: Integracijski test validatorjev

Ko smo se lotevali projekta, smo se zavedali, da bo izvajanje vseh testov trajalo predolgo. Želeli smo, da bi lahko teste izvajali avtomatsko in da bi po izvedbi testov prejeli obvestilo o morebitnih napakah. Sprotna integracija je namenjena ravno reševanju tovrstnih problemov.

Poglavje 6

Sprotna integracija (Continuous Integration — CI)

Prav tako kot TDD je tudi sprotna integracija ena izmed tehnik ekstremnega programiranja [15]. Pri razvoju programske opreme se pogosto dogaja, da programski sistem, ki ga razvijamo, večino časa razvoja ne deluje. Še posebej je to opazno pri obsežnih projektih, ki jih razvijajo velike ekipe. Pri njih razvijalci med stopnje razvoja pogosto vključijo tudi stopnjo integracije. V njej lahko pride do zapletov pri združevanju vej in popravljanja nepravilno implementiranih programskih vmesnikov. Zaradi nepredvidljivosti je težko oceniti, koliko časa bo trajala ta stopnja razvoja. Tem problemom se najlažje izognemo z uporabo tehnike, imenovane sprotna integracija. Ta zahteva, da se vsakič, ko nekdo odda spremembe v kodi v sistem za nadzor verzij izvorne kode, celotna aplikacija prevede in stestira s pomočjo množice avtomatskih testov. Če testi ugotovijo napako, jo razvijalska ekipa takoj odpravi. Cilj sprotne integracije je, da programski sistem ves čas deluje. Izhaja iz ideje, da če je koda, ki smo jo ravnokar napisali, pravilno integrirana z ostalo kodo, potem to lahko dokažemo s tem, da jo poženemo. Če ugotovimo napako, jo popravimo takoj, ker je to lažje, kot da jo popravljamo kasneje.

V primeru standardnega načina razvoja programa se o pravilnem delovanju našega programskega sistema prepričamo šele kasneje v fazi testiranja

ali integracije. Pri sprotni integraciji pa aplikacija dokazano ves čas deluje. Hrošči se ugotovijo hitro, kar zniža ceno odprave napak in prihrani čas. Ker smo s testi zagotavljali delovanje naše aplikacije, smo se odločili tudi za uporabo sprotne integracije [15].

6.1 Vključitev sprotne integracije

Tehniko sprotne integracije lahko uspešno integriramo v naš postopek razvoja le ob upoštevanju nekaterih ključnih zahtev [15].

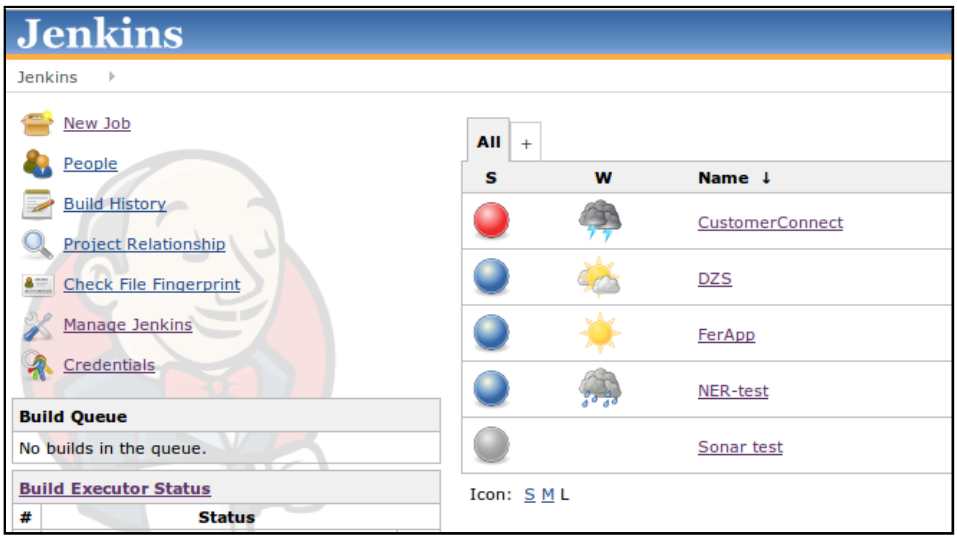
- **Sistem za kontrolo verzij**: Izbrali smo sistem za kontrolo verzij SVN. V repozitorij moramo vključiti vse datoteke, ki jih naš projekt vsebuje (kodo, teste, skripte za generiranje podatkovne baze, dokumentacijo ...).
- **Avtomatska gradnja projekta (ang. An Automated Build)**: Ni pomembno, ali uporabljamo le ukazno vrstico v terminalu ali kakšno bolj napredno orodje, pomembno je le, da omogoča, da lahko oseba ali računalnik prevede projekt in zažene teste ter projekt. Tej zahtevi smo zadostili že, ko smo se odločili, da bomo za izdelavo naše aplikacije uporabili Apache Maven.
- **Soglasje ekipe**: Kot smo že omenili na začetku tega poglavja, je sprotna integracija tehnika, ne orodje. Člani ekipe se morajo strinjati z njeno uporabo, saj za dosledno izvajanje zahteva določeno mero discipline in sodelovanje vseh.

Poleg zgoraj naštetih zahtev je zelo pomembno tudi, da datoteke v repozitorij shranjujemo dovolj pogosto - večkrat na dan. Tako dosežemo, da spremembe v kodi niso prevelike in lahko napake hitro ugotovimo in odpravimo. S tem tudi zmanjšamo verjetnost, da bi se spremembe, ki vključujejo veliko različnih datotek, prekrivale s spremembami ostalih razvijalcev, ki sodelujejo pri projektu. Pomembno je tudi, da je proces zaganjanja projekta čim krajši, saj se v nasprotnem primeru hitro zgodi, da se mu ljudje začnejo izogibati.

Danes obstajajo različne odprtokodne in plačljive različice orodij za podporo sprotni integraciji. Čeprav za uporabo tehnike ta orodja niso obvezna, je izvajanje sprotne integracije z njihovo uporabo lažje. Odločili smo se za uporabo strežnika za sprotno integracijo Jenkins, s pomočjo katerega smo implementirali sprotno integracijo v naš razvoj.

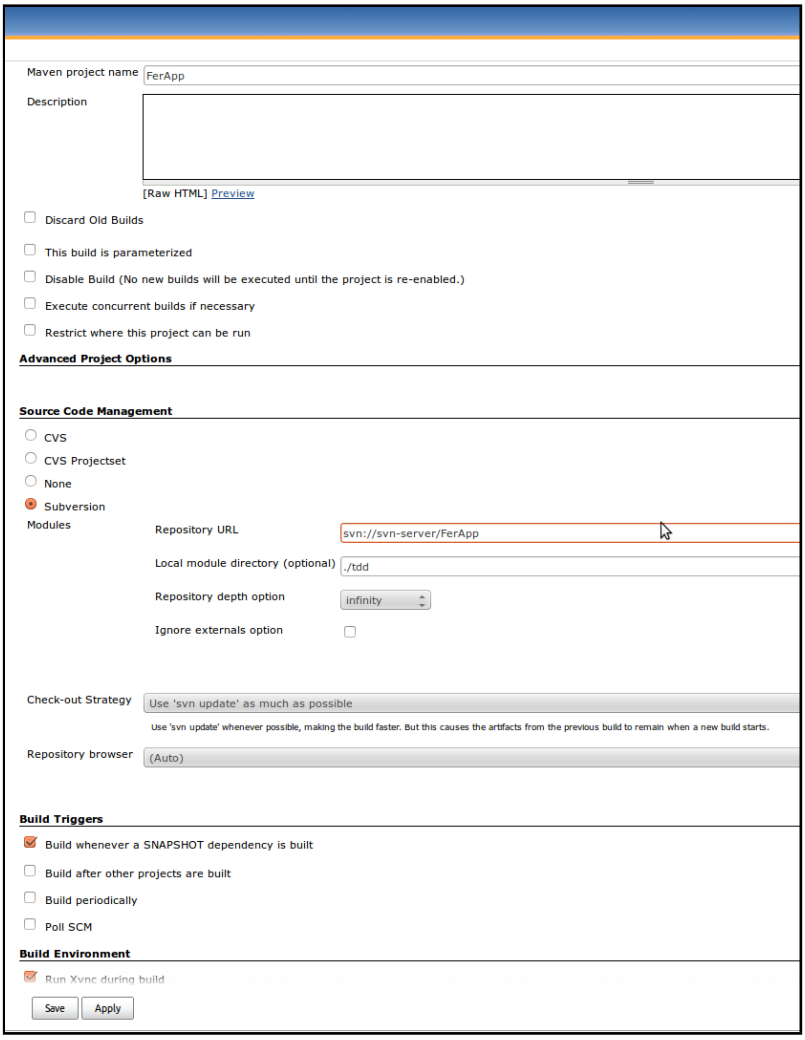
6.2 Jenkins

Kot smo že omenili v poglavju 4.13, je Jenkins odprtokodno orodje za podporo sprotni integraciji, napisano v Javi. V osnovi ima, tako kot vsak strežnik za podporo sprotni integraciji, dve komponenti. Prva komponenta je proces (ang. long-running process), ki lahko v intervalih izvaja preprosta zaporedja korakov (ang. workflow). Druga omogoča prikazovanje rezultatov projektov in testov, ki so bili zagnani, za obveščanje o uspešnem ali neuspešnem zaganjanju, generiranju poročil itd. Jenkins ponuja spletni uporabniški vmesnik, prikazan na sliki 6.1, preko katerega ga lahko konfiguriramo, dodajamo opravila (ang. jobs) za projekte, pregledujemo rezultate ...



Slika 6.1: Jenkinsov uporabniški vmesnik - pregled projektov

Osnovno različico Jenkinsa smo razširili s pomočjo vtičnikov (ang. plugin), ki smo jih namestili preko uporabniškega vmesnika za urejanje vtičnikov. Z njimi smo dodali podporo za orodja Subversion, Apache Maven in Sonar. Po namestitvi smo dodali naš projekt, kot je to prikazano na sliki 6.2. Nastavili smo, da Jenkins vsako uro povprašuje (ang. pooling) v SVN-ju, če je kdo dodal spremembe v repozitorij. V primeru, da ugotovi, da je prišlo do sprememb, izvede avtomatsko zaganjanje projekta in testov in v primeru napak pošlje obvestilo na e-poštni naslov.



Slika 6.2: Jenkinsov uporabniški vmesnik - dodajanje projekta

6.3 Korist sprotne integracije v praksi

Med razvojem aplikacije smo se zavedali, da se naš projekt kljub podobni arhitekturni zasnovi po kompleksnosti problemov ne more primerjati s poslovnimi aplikacijami, s katerimi se srečujemo v resničnem svetu. Kljub temu smo že v primeru veliko manjšega projekta opazili, da s testi odkrijemo veliko napak, ki bi jih sicer spregledali.

Med programiranjem aplikacije FerApp smo najprej sprogramirali metodo `UserService.fetchAll()`, ki je iz podatkovne baze prebrala vse uporabnike in je prikazana v odseku 6.1.

```
1  @Entity
2  @Table('users')
3  @NamedQueries(value = {
4      @NamedQuery(name = User.FETCH_ALL, query = "SELECT u FROM User u")
5      })
6  public class User implements Serializable {
7    public static final String FETCH_ALL = "User.FETCH_ALL";
8
9    // ... manjka koda
10 }
11
12 @Stateless
13 public class UserService
14 {
15   // ... manjka koda
16
17   public List<User> fetchAll()
18   {
19     @SuppressWarnings("unchecked")
20     List<User> users = em.createNamedQuery(User.FETCH_ALL)
21       .getResultList();
22
23     return users;
24   }
25 }
```

Izvorna koda 6.1: Metoda UserService.getAll()

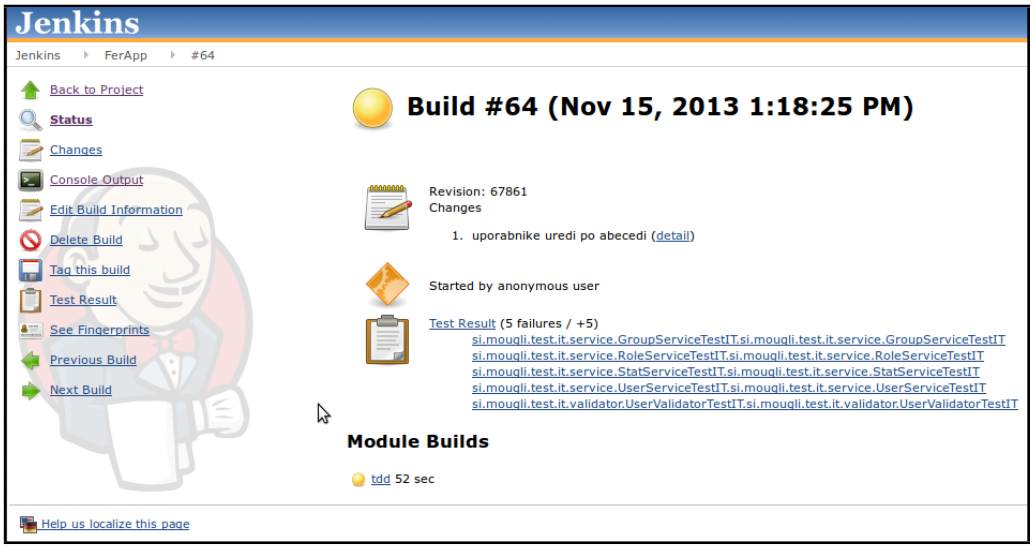
Kasneje smo se odločili, da bi bila uporabniku prijaznejša rešitev, če bi uporabnike prebrali iz baze, urejene po abecednem redu glede na ime. Sprememba bi bila brez naprednega razvojnega okolja Eclipse velika, saj se metoda uporablja v različnih razredih. Zato so bile potrebne spremembe v več datotekah. V kodi v odseku 6.2 smo spremenili stavek HQL in preoblikovali ime metode, da je namen metode razviden že iz imena. Problem je nastal, ker smo pri pisanju stavka HQL naredili napako.

```
1  @Entity
2  @Table('users')
3  @NamedQueries(value = {
4      @NamedQuery(name = User.ALL_ORDERED_BY_NAME, query = "SELECT u FROM
       User u ORDER BY u.name ASV")
5      })
6  public class User implements Serializable {
7    public static final String ALL_ORDERED_BY_NAME = "User.
       ALL_ORDERED_BY_NAME";
8
9    // ... manjka koda
10 }
11
12 @Stateless
13 public class UserService
14 {
15   // ... manjka koda
16
17   public List<User> fetchAllOrderedByName()
18   {
19     @SuppressWarnings("unchecked")
20     List<User> users = em.createNamedQuery(User.ALL_ORDERED_BY_NAME)
21       .getResultList();
22
```

```
23      return users;
24    }
25  }
```

Izvorna koda 6.2: Metoda UserService.getAll()

Zagnali smo teste enot, da smo se prepričali, če jih naše spremembe prestanejo. Testi so se uspešno zaključili. Ker izvajanje integracijskih testov zahteva preveč časa, jih nismo zagnali. Ko smo spremembe vnesli v SVN-repozitorij, je Jenkins začel z gradnjo in zaganjanjem projekta in testov. Integracijski testi so napako odkrili in build se je zaključil s statusom `UNSTABLE`. Na sliki 6.3 je prikazano kratko poročilo o neuspešni izvedbi testov.



Slika 6.3: Neuspešno izvajanje testov na Jenkinsu

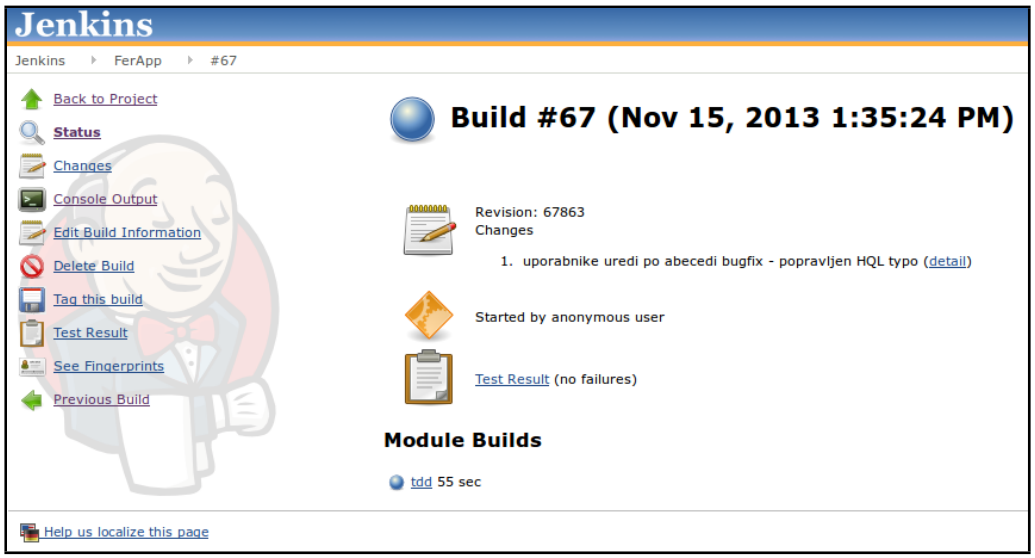
Jenkins nas je opozoril na napako in ker sprememb ni bilo veliko, smo ob pregledu kode napako hitro opazili in jo odpravili, kot je to prikazano v odseku 6.3.

```
1 @Entity
2 @Table('users')
```

```
3  @NamedQueries(value = {
4      @NamedQuery(name = User.ALL_ORDERED_BY_NAME, query = "SELECT u FROM
       User u ORDER BY u.name ASC")
5      })
6  public class User implements Serializable {
7    public static final String ALL_ORDERED_BY_NAME = "User.
       ALL_ORDERED_BY_NAME";
8
9    // ... manjka koda
10 }
```

Izvorna koda 6.3: Metoda UserService.getAll()

Ponovno smo zagnali prevajanje in zaganjanje projekta in tokrat bili uspešni, kar je razvidno s slike 6.4.



Slika 6.4: Uspešno končana gradnja projekta in izvedba testov na Jenkinsu

S pomočjo vtičnika za podporo orodij Apache Maven in Sonar smo nastavili, da Jenkins sproži analizo kode na Sonarju. Ta rezultate analize prikaže v obliki poročil. Več o Sonarju bomo povedali v nadaljevanju.

Poglavje 7

Kvaliteta kode in Sonar

Programerji se strinjamo, da je kvaliteta kode zelo pomembna, saj vsebuje manj hroščev, lažje jo je vzdrževati in dodajati nove funkcionalnosti. V praksi velja, da je ocena kvalitete kode pogosto subjektivna, še zlasti v primeru, ko ni ustreznih standardov za njeno preverjanje. Zato se člani ekipe med sabo dogovorijo za standarde, ki jim mora koda zadostiti. Standarde kode (ang. coding standards) predstavlja množica pravil. Ta ne določajo le oblike kode, poimenovanja spremenljivk in razredov, ampak tudi prakse slabega programiranja, ki se jim želimo izogniti. S tem ko se držimo dogovorjenih standardov, postane naša koda bolj berljiva in jasna tudi za ostale člane ekipe [26].

Sonar je orodje, ki združuje različna orodja za zagotavljanje kakovosti kode in njihove rezultate prikaže na spletni strani. Uporablja različne vtičnike Mavena, s katerimi analizira kodo projekta in prikaže množico različnih poročil z metrikami za zagotavljanje kakovosti kode. Poročila prikazujejo metrike o pokritosti kode s testi, upoštevanju dogovorjenih standardov, dokumentaciji, skupnem številu vrstic kode, kompleksnosti kode ...

Nastavitve in pravila, ki jih uporabljajo različna orodja, urejamo preko Sonarjevega spletnega uporabniškega vmesnika. Orodja, ki smo jih uporabili za analizo kode, so Checkstyle, PMD in FindBugs [26].

- **Checkstyle** je orodje za analizo kode Java. Pravila zanj lahko določimo

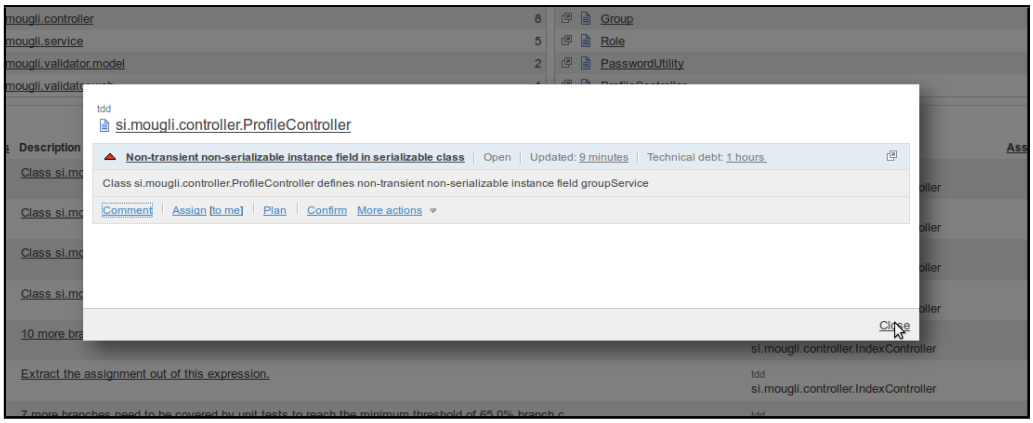
preko uporabniškega vmesnika v Eclipsu ali pa neposredno v datoteki XML. Tak način pisanja pravil je zelo prilagodljiv. Omogoča preverjanje standardov kode, ki smo jih nastavili, kompleksnost in podvojenost kode in znake slabega programiranja.

- **PMD** je še eno orodje za analizo kode. Njegovi glavni nalogi sta odkrivanje potencialnih težav z neučinkovitostjo kode, njeno velikostjo in kompleksnostjo ter preverjanje upoštevanja priporočenih tehnik programiranja. Primeri vzorcev kode, ki jih preverja, so npr. ali ukaz `switch` uporablja `default` stavek, prazen if stavek, globoko gnezdeni if stavki, ... Nekatera njegova pravila sovpadajo s pravili, ki jih določa Checkstyle.

- **FindBugs** analizira kodo in išče morebitne hrošče, vzorce slabega programiranja in potencialne težave s hitrostjo oz. zmogljivostjo programov. Sposoben je odkriti neskončne zanke, izjeme tipa NullPointerException ... Ponavadi odkrije manj napak kot Checkstyle in PMD, vendar pa so te bolj pomembne.

Med programiranjem projekta nas je povraten odziv, ki smo ga s pomočjo sprotne integracije dobili v obliki uspešno prestanih testov in rezultatov analize kode, neprestano opozarjal na napake, ki smo jih napravili, in tako prispeval k boljšemu končnemu izdelku. Na sliki 7.1 je prikazan primer opisa težave, ki jo prikaže Sonar.

Pri razvoju našega projekta smo hitro opazili, da nam je način programiranja projekta s tehniko TDD omogočil, da smo se lahko lotili spreminjanja kode brez strahu, da bi s tem povzročili hrošča na drugem mestu, saj smo se pri tem zanašali na teste. Vendar pa nas je poleg tega zanimalo tudi, če sta tehniki TDD in sprotna integracija kaj pripomogli k izboljšanju kvalitete kode.



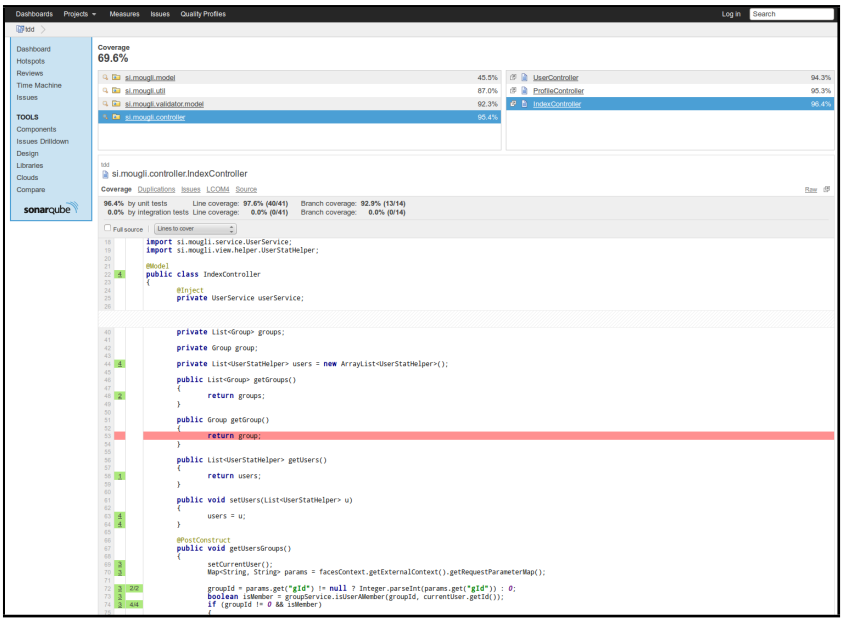
Slika 7.1: Sonarjev prikaz opozorila na problem

7.1 Metrike in rezultati

Za oceno uspešnosti izvajanja tehnike TDD smo uporabili metriko, ki meri pokritost kode s testi. Sonarjeva analiza je pokazala, da je s testi enot pokritih le 69.6 % kode. Ob površnem pregledu se zdi, da nam je v naši nameri, da aplikacijo sprogramiramo na način TDD, spodletelo, vendar pa je rezultat zavajajoč. Kot smo opisali v poglavju 5, smo podatkovno plast aplikacije zasnovali tako, da nam je ni treba testirati. Ker je naša aplikacija preprosta in ne vsebuje veliko poslovne logike, ta plast predstavlja velik del aplikacije.

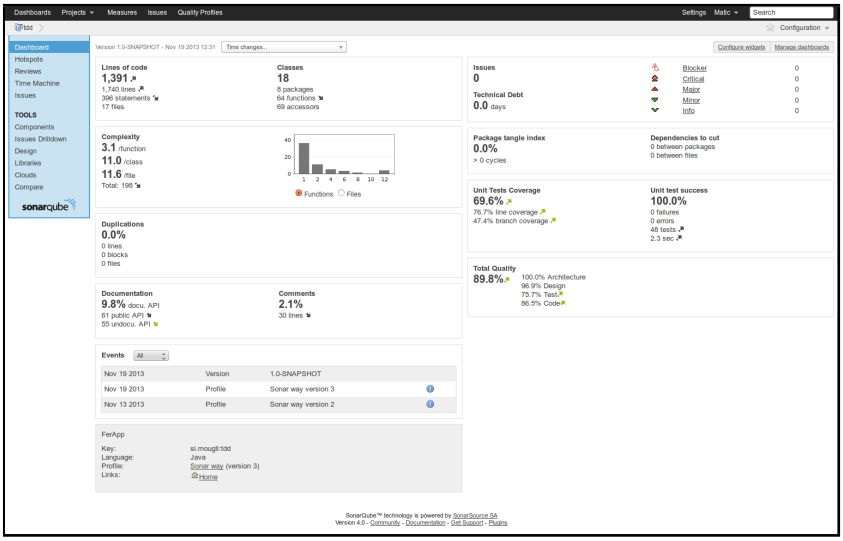
Sonar omogoča podrobnejši pregled pokritosti kode s testi preko uporabniškega vmesnika. Prikaže le tiste dele aplikacije, ki niso 100-odstotno pokriti s testi. S slike 7.2 je razvidno, da je pokritost poslovne plasti, ki se nahaja v paketu (ang. package) `si.mougli.tdd.service`, 100-odstotna, saj paketa ni na seznamu. Zelo visoka, 95,4-odstotna, je tudi pokritost paketa `si.mougli.tdd.controller`, ki prav tako vsebuje logiko za pravilno prikazovanje uporabniškega vmesnika.

Sprotna integracija nam je omogočila, da smo sproti popravljali napake standardov kode, ki jih je odkril Sonar. V nasprotnem primeru bi se skozi razvoj projekta nabralo preveč napak, da bi jih lahko hitro odpravili. Na sliki 7.3 je prikazana nadzorna plošča s poročili analize kode aplikacije FerApp. S



Slika 7.2: Sonarjev prikaz pokritosti kode s testi

slike je razvidno, da poročilo v okencu *Issues* ne vsebuje nobene napake, ki bi jo bilo potrebno odpraviti.



Slika 7.3: Prikaz različnih metrik orodja Sonar

Sonar išče tudi podvojene dele kode in o njih poroča v okencu *Duplications*, vendar jih naša aplikacija ne vsebuje. Metriki, ki smo ju merili, sta še podatek o kompleksnosti kode in številu komentarjev v kodi. Slednja je izmed vseh meritev najbolj subjektivna. Mnenja o primerni količini komentarjev v kodi so si med programerji nasprotujoča, zato njena vrednost neposredno ne vpliva na kvaliteto kode [18].

Bolj pomemben je podatek, izpisan v okencu *Complexity*, ki prikazuje kompleksnost kode. Ta namreč vpliva na zahtevnost, s tem pa tudi na ceno vzdrževanja. Na nivoju metode je kompleksnost kode določena s povprečnim številom vrstic, na nivoju razreda pa s povprečnim številom metod v razredu. Mi želimo, da naše metode ne vsebujejo prevelikega števila vrstic, saj jih je zaradi kompleksnosti težko testirati. Ker smo naš razvoj vodili s tehniko TDD, je kompleksnost naših metod nizka, saj povprečno metoda vsebuje 3,1 vrstice kode. Priporočilo v knjigi Clean Code pravi, da naj bo metoda čim krajša, dolga le nekaj vrstic [18].

Poglavje 8

Sklepne ugotovitve

Cilj diplomske naloge je bil prikazati postopek testno vodenega razvoja in sprotne integracije pri razvoju aplikacij v programskem jeziku Java EE. Zanimalo nas je, kako tak pristop k razvoju programske opreme vpliva na razvoj projekta in kvaliteto kode.

V prvem delu diplomske naloge smo predstavili različne vrste testov, tehniko TDD in opisali njene prednosti ter slabosti. V drugem delu smo s pomočjo testno vodenega razvoja razvili spletno aplikacijo Java EE — FerApp. Aplikacija omogoča registracijo novih uporabnikov, prijavo, dodajanje skupin, dodajanje uporabnikov skupinam in beleženje statistike. V tretjem delu smo opisali, kako smo tehniko TDD dopolnili z uporabo tehnike sprotne integracije in ostalimi orodji.

Aplikacijo smo v skladu z arhitekturno zasnovo realnih aplikacij razdelili na plasti in testirali vsako plast posebej. S pomočjo testov enot smo vodili razvoj aplikacije, z integracijskimi testi pa preverjali, če naša koda deluje pravilno tudi znotraj aplikacijskega strežnika in če se različni deli aplikacije med seboj pravilno povezujejo. Za testiranje uporabniškega vmesnika smo poleg testov enot uporabili funkcionalne teste.

Čeprav smo tehniko TDD uspešno uporabili pri razvoju projekta, je odstotek kode, ki je pokrit s testi, presenetljivo nizek. Vzrok je v tem, da smo aplikacijo zasnovali tako, da nam podatkovne plasti ni potrebno testi-

rati. Ker je aplikacija relativno majhna, ta plast obsega precejšen del kode. O uspešni uporabi tehnike TDD priča dejstvo, da sta pokritosti plasti poslovne logike in predstavitvene plasti zelo visoki, in sicer 100- oz. 95,4-odstotni. Ugotovili smo, da je trditev, da TDD-način razvoja zahteva veliko učenja, utemeljena. Težave smo imeli predvsem na začetku, preden smo se navadili, da je potrebno test napisati pred implementacijo kode. Opazili smo tudi, da je za uspešno izvajanje tehnike potrebno dobro znanje programiranja.

Pomanjkljivosti testov enot smo dopolnili z integracijskimi in funkcionalnimi testi. Izvajanje testov smo avtomatizirali s pomočjo strežnika za podporo sprotni integraciji Jenkins. Z njim smo ob vsaki posodobitvi repozitorija SVN aplikacijo zgradili, zagnali teste in prenesli na aplikacijski strežnik. Po namestitvi aplikacije na strežnik smo z orodjem Sonar analizirali kodo. Tako smo predstavili celoten cikel razvoja Java EE-aplikacije.

Čeprav se je uporabnost in učinkovitost tovrstnega načina razvoja pokazala že v našem primeru, se njegovi pozitivni učinki še bolj izrazijo v primerih, ko projekt razvija ekipa programerjev. S številčnejšo ekipo in z obsežnostjo projekta se pojavi več potencialnih težav, ki jih način razvoja, kot smo ga predstavili v diplomski nalogi, odpravlja takoj, ko se te pojavijo. S testi smo "spletli" učinkovito varnostno mrežo, na katero se bomo lahko zanašali med vzdrževanjem projekta v prihodnosti.

Kljub očitnim prednostim se testno voden razvoj še vedno ni povsem uveljavil. Čeprav so njegove pozitivne lastnosti pri večjih ekipah in projektih še bolj izrazite, so večje tudi težave na začetku uporabe. Ne glede na to se zaradi dobrih rezultatov število uporabnikov povečuje.

Literatura

[1] **D. Allen, A. Knutsen, P. Muir, A. Rubinger**, *Arquillian Refernce Guide*, http://docs.jboss.orgarquillian/reference/1.0.0.Alpha1/en-US/html_single/.

[2] **C. Bauer, G. King**, *Java Persistence with Hibernate*, Manning Publications Co., 2007.

[3] **K. Bent**, *Test-Driven Development By Example*, Three Rivers Institute, 2002.

[4] **T. Bhat, N Nagappan**, *Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies.*

[5] **R. Black**, *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, Wiley Publishing, Inc., 2002.

[6] **D. Burns** *Selenium 2 Testing Tools Beginner's Guide*, Packt Publishing Ltd., October 2012.

[7] **P. Čebokli**, *Avtomatizacija testiranja kot ključ agilnosti razvoja programse opreme*, Magistrsko delo, 2006.

[8] **Eclipse**, *What is Eclipse*, http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm.

[9] **M. Fowler**, *Patterns of Enterprise Application Arhitecture*, Addison-Wesley, 2006.

[10] **M. Fowler**, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2002, Preface, pp *XVI*.

[11] **R. Gold**, *Test Driven Development: A J2EE Example*, Apress, 2005.

[12] **A. Goncalves**, *Beginning Java EE 7*, Pact Publishing Ltd., Junij 2013.

[13] **H2**, *H2 Database Engine*, http://h2database.com/html/main.html.

[14] **D. H. Hansson**, *Testing like the TSA*, http://37signals.com/svn/posts/3159-testing-like-the-tsa.

[15] **J. Humble, D. Farley**, *Continuous Delivery*, Addison-Wesley, 2010.

[16] **D. S. Janzen**, *Software Architecture Improvement through TestDriven Development*, University of Kansas.

[17] **L. Koskela**, *Test driven: TDD and Acceptance TDD for Java Developers*, Manning Publications Co, 2007.

[18] **R. C. Martin**, *Clean Code: A Handbook Of Agile Software Craftmanship*, Prentice Hall, 2008.

[19] **Maven**, *What is Maven*, http://maven.apache.org/what-is-maven.html.

[20] **Mockito**, *What is Mockito*, http://code.google.com/p/mockito/.

[21] **MySQL** *What is MySQL*, http://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html.

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

[23] **A. Oram, G. Wilson**, *Making Software*, O'Reilly, 2010.

[24] **Oracle**, *The Java EE 6 Tutorial*, Oracle, January 2013.

[25] **P. Seibel**, *Coders at Work: Reflections on the Craft of Programming*, Apress, 2009.

[26] **J. F. Smart**, *Jenkins: The Definitive Guide*, O'Reilly Media, 2011.

[27] **Sonar**, *What is Sonar*
http://www.sonarsource.org/.

[28] **S. Stewart**, *Selenium Web Driver*,
http://www.aosabook.org/en/selenium.html.

[29] **Wikipedia**, *Stress testing* ,
http://en.wikipedia.org/wiki/Stress_testing_%28software%29.

[30] **Subversion**, *What is subversion*,
http://subversion.apache.org/.

[31] **Wikipedia**, *JBoss AS 7*,
http://en.wikipedia.org/wiki/JBoss.

[32] **Wikipedia**, *Jenkins*,
http://en.wikipedia.org/wiki/Jenkins_(software).

[33] **Wikipedia**, *Manual Testing*,
http://en.wikipedia.org/wiki/Manual_testing.

[34] **Wikipedia**, *Sonar software quality*,
http://en.wikipedia.org/wiki/Sonar_(software_quality).

[35] **Wikipedia**, *Software testing*,
http://en.wikipedia.org/wiki/Software_testing.

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