UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Aleš Šarkanj

Primerjava orodij za testiranje enot JUnit in TestNG

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: dr. Igor Rožanc

Ljubljana, 2011

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: 00544/2011
Datum: 04.04.2011

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

Kandidat: **ALEŠ ŠARKANJ**

Naslov: **PRIMERJAVA ORODIJ ZA TESTIRANJE ENOT JUNIT IN TESTNG**
THE COMPARISON OF THE UNIT TEST TOOLS JUNIT AND TESTNG

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

V diplomski nalogi predstavite sistematično primerjavo dveh znanih orodij za izvajanje testiranja posameznih enot javanske programske kode: Junit in TestNG. Najprej predstavite področje testiranja programske opreme s poudarkom na testiranju enot in omenjenih orodij. Pred izvedbo opredelite učinkovite kriterije za primerjavo z vidika razvijalca programske opreme, ob izvedbi pa predstavite ugotovitve in končno izbiro boljšega orodja.

Mentor:

viš. pred. dr. Igor Rožanc

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani **Aleš Šarkanj**, z vpisno številko **63030296**, sem avtor diplomskega dela z naslovom:

Primerjava orodij za testiranje enot JUnit in TestNG

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom **dr. Igorja Rožanca**
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 15.9. 2011 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju dr. Igorju Rožancu, za strokovno vodstvo in pomoč pri izdelavi diplomskega dela. Posebno zahvalo namenjam svoji družini za materialno in moralno podporo v času študija. Hvala tudi dekletu Tamari, ki me je podpirala in bodrila pri izdelavi diplomske naloge.

Hvala!

Moji mami

Kazalo vsebine

Povzetek

Testiranje ima zelo pomembno vlogo pri razvoju programskega opreme. Je proces, ki nam zagotavlja, da programska oprema deluje po pričakovanjih. Pomemben del testiranja predstavlja testiranje enot kot najnižji nivo testiranja.
Namen diplomskega dela je primerjava orodij za testiranje enot JUnit in TestNG, ki sta najpopularnejši odprtokodni orodji v programskem jeziku Java.
V prvem delu bomo predstavili področje testiranja in njegov pomen v okviru razvoja programske opreme. Predstavili bomo osnovno terminologijo in nivoje testiranja. V nadaljevanju bomo prikazali področje testiranja enot, v okviru katerega bomo tudi na kratko predstavili orodji JUnit in TestNG.
Glavni del diplomskega dela je namenjen primerjavi obeh orodij. Najprej bomo zajeli zahteve, ki jih pričakujemo od ogrodja za testiranje enot. Na podlagi teh zahtev bomo definirali kriterije, na osnovi katerih bomo izvedli primerjavo. Na koncu bodo predstavljeni rezultati primerjave in izbira najboljšega orodja za potrebe testiranja enot.

Ključne besede:

testiranje, testiranje enot, orodje za testiranje enot, ogrodje za testiranje enot, JUnit, TestNG

Abstract

Testing holds an important role in developing software. It is a process ensuring software works as expected. An important part of testing is unit testing as the basic testing level.
The purpose of this thesis is comparing JUnit and TestNG, two of the most popular unit testing tools in Java.
In the first part we will present testing and its purpose within software development. We will also present the basic terminology and levels of testing. Further on the area of unit testing will be presented, as well as JUnit and TestNG.
The main part of this thesis is dedicated to comparing the two tools. First we will consider the requirements, expected in a unit testing tools, from which a set of comparison criteria will be established. Finally the comparison results and choice of the best tool will be presented.

Keywords:
testing, unit testing, unit testing tool, unit testing framework, JUnit, TestNG

1 Uvod

Razvoj programske opreme postaja vedno bolj kompleksen. Od programske opreme se vse bolj pričakuje, da bo kakovostna in zanesljiva. Ena izmed aktivnosti, s katero zagotavljamo njeno kakovost, se imenuje testiranje. Testiranje je sistematičen pristop k zagotavljanju kakovosti programske opreme.

Pomemben del testiranja je testiranje enot. Testiranje enot je nivo testiranja, na katerem testiramo najmanjše dele programske opreme. Enota je najmanjša zaokrožena celota programske kode, kar je v objektnem programiranju razred. Za izvedbo testiranja enot se uporabljajo t.i. orodja za testiranje enot.
V nadaljevanju bomo namesto termina *orodje za testiranje enot* uporabljali termin *ogrodje za testiranje enot*, saj je beseda oz. izraz *ogrodje* v kontekstu testiranja enot bolj pravilna in splošno sprejeta.

Glavni del diplomske naloge se ukvarja s primerjavo dveh ogrodij za testiranje enot. To sta ogrodji v JUnit in TestNG, ki sta najbolj popularni v programskem jeziku Java. Ogrodji bomo primerjali z vidika njune uporabnosti. Izbira določenega ogrodja ni vselej lahka, saj je odvisna od velikega števila faktorjev. Najprej bomo postavili zahteve, ki jih mora izpolnjevati ogrodje za testiranje enot. Na podlagi teh zahtev bomo oblikovali omejeno število kriterijev, na podlagi katerih bomo primerjali obe ogrodji ter izpostavili njune prednosti in slabosti. Na podlagi rezultatov primerjave bomo predstavili ugotovitve in izbrali najbolj primerno ogrodje.

2 Testiranje programske opreme

Vsak programer med razvijanjem programske opreme lahko stori napako. Razlika med dobrim in slabim programerjem pa je ta, da dober programer uporablja testiranje za odkrivanje oziroma preprečevanje teh napak.
Testiranje (*ang. testing*) je proces, ki nam zagotavlja, da programska oprema dela to, kar je njen namen in ne dela tistega, kar ni njen namen. Kakovostno testiranje nam daje zaupanje v programsko opremo, da je le-ta zanesljiva in s čim manj pomanjkljivostmi. Nobena programska oprema ni brez napak. Utopično bi bilo pričakovati, da lahko naredimo program, ki bo brez napak. Lahko se pa temu zelo približamo, in sicer z dobrim testiranjem.

Ko ljudje slišijo pojem *testiranje programske opreme*, po navadi najprej pomislijo na testiranje uporabniškega vmesnika. V vnosna polja vpišemo nekaj vrednosti in preverimo, ali dobimo pričakovan odziv. Ko naredimo spremembe v programski kodi, ponovimo postopek. Zaradi takšnega preverjanja velja, da je testiranje dolgočasno, monotono in ponavljajoče delo, ki ga lahko izvaja vsak, brez posebnega znanja. Vendar ni tako. Pojem testiranja je mnogo širši od testiranja uporabniškega vmesnika [4].

2.1 Terminologija

Za pravilno razumevanje testiranja je pomembno ločevanje osnovnih pojmov *okvara*, *napaka* in *odpoved*.

Napaka *(ang. fault)* je pomanjkljivost, pomota v programu. Napake naredijo razvijalci z napačnim razumevanjem zahtev, zaradi neznanja, kar posledično privede do napačne implementacije.

Okvara *(ang. error)* je neveljavno notranje stanje, ki je posledica napake. Do okvare pride, ko se izvede del programa, ki vsebuje napako.

Odpoved *(ang. failure)* je odstopanje od pričakovanega zunanjega obnašanja programa. Je nezmožnost sistema, da opravi svojo funkcijo oziroma namen. Do odpovedi pride zaradi okvar. Okvara preide v odpoved samo v primeru, če program navzven ne deluje po pričakovanjih, oziroma ne vrača pričakovanega rezultata. Program lahko v določenih primerih deluje po pričakovanjih tudi, če vsebuje okvare, saj vsaka okvara ne vodi v odpoved programa. Program je tedaj sicer v napačnem notranjem stanju, vendar je zunanje obnašanje programa po naključju enako pričakovanemu. Ko pa so izpolnjeni določeni pogoj, se pokaže okvara v obliki odpovedi.

Zelo pomembna je izbira vhodnih podatkov, s katerimi testiramo program, saj tako razkrijemo čim več odpovedi programa.

Testni primer (*ang. test case*) je skupek vrednosti in pogojev, na podlagi katerih določimo, če program deluje tako kot se to od njega pričakuje. Testni primer je sestavljen iz zbirke vhodnih podatkov, pričakovanih izhodnih podatkov ter pogojev pod katerimi je test uspešen ali neuspešen. Testne primere pišemo na podlagi testnih zahtev, ki so formalni zapis zahtev delovanja programske opreme.

Testna zahteva (*ang. test requirement*) je zahteva, ki jo testni primer mora zadovoljiti. Testne zahteve lahko napišemo na podlagi specifikacije, izvorne kode, itd. Ponavadi se jih uporablja v množicah testnih zahtev (*ang. set of test requirements*).

2.2 Definicija testiranja

Posebej pomembno je tudi ločevanje med pojmoma testiranje in razhroščevanje.

Testiranje (*ang. testing*) je vrednotenje kakovosti programske opreme na podlagi zahtev. S testiranjem ugotavljamo odstopanje od zahtevanega delovanja programa.
Osnovna ideja testiranja je, da izvedemo program in opazujemo, če deluje po pričakovanjih. Če opazimo odstopanje od zahtevanega delovanja, poiščemo vzrok in ga odpravimo.

Razhroščevanje *(ang. debugging)* je iskanje napak na podlagi odpovedi programa. V primeru, da pride do odpovedi programa, moramo najprej najti izvor odpovedi, oziroma napako, nato lahko najdeno napako tudi odpravimo.

2.3 Pogled na testiranje

Najpogostejši vzrok slabega testiranja in posledično slabe programske opreme je napačen pogled na testiranje. Najosnovnejša razlaga je, da je testiranje dokazovanje pravilnosti programa. Ta pogled ima veliko pomanjkljivost, saj se niti teoretično niti praktično ne da dokazati pravilnost delovanja programa. Da bi dokazali, da program deluje pravilno, bi morali preveriti odziv programa na vse možne vhodne podatke. Za večino programov pa obstaja zelo veliko oziroma velikokrat neskončno število možnih vhodnih podatkov, vseh pa ne moremo uporabiti pri testiranju. Precej enostavni programi, imajo lahko na tisoče možnih vhodnih podatkov in rezultatov.
S tega stališča delimo testiranje na več nivojev:

Nivo 0

Je najnižji nivo, pri katerem ni razlike med nepravilnim obnašanjem programa in napako v programu. Posledično tudi ni razlike med testiranjem in razhroščevanjem.

Nivo 1

Na tem nivoju skušamo dokazati, da program deluje pravilno. Kot že omenjeno, se pravilnosti programa ne da dokazati. Takšno testiranje ima poleg tega še eno pomanjkljivost. V primeru, da testi ne najdejo nobenih odpovedi, ne vemo, če program deluje po pričakovanjih, ali pa imamo slabe teste. Ker se na tem nivoju išče potrditev delovanja programske opreme, se pogosto testira z testnimi primeri, ki najverjetneje ne bodo našli napak.

Nivo 2

Tukaj je namen testiranja dokazati, da v programu obstajajo odpovedi. Pomemben preskok v mišljenju je, da se zavedamo, da vsak program vsebuje odpovedi. Nastane pa isti problem kot na prejšnjem nivoju, saj v primeru da ne najdemo nobenih odpovedi, ne vemo ali program deluje po pričakovanjih ali pa imamo le slabe teste.

Nivo 3

Nam da spoznanje, da lahko testiranje pokaže le prisotnost odpovedi, ne more pa pokazati njenih odsotnosti. To pomeni, da smo pri uporabi programske opreme zmeraj izpostavljeni določenemu tveganju. Na tem nivoju pridemo do spoznanja, da namen testiranja ni dokazovanje prisotnosti ali odsotnosti odpovedi, temveč zmanjšati tveganje, ki smo mu izpostavljeni ob uporabi programske opreme.

Nivo 4

Je zadnji nivo, kjer namen testiranja ni dokazovanje, ali program deluje ali ne, temveč razvijanje programske opreme, ki bo kakovostna.

Večina razvijalcev začne s pogledom na testiranje na nivoju 0 in se počasi pomika po nivojih navzgor. Pravi namen testiranja je seveda razvoj kakovostne programske opreme [10].
Dobra analogija je primer črkovalnika. Na črkovalnik lahko gledamo kot na orodje, ki popravlja napačno zapisane besede. Lahko pa gledamo nanj tudi kot pomoč, ki nam omogoča, da se na podlagi napačno zapisanih besed naučimo, kako se besede pravilno črkujejo, da naslednjič pravilno zapišemo besede.

2.4 Nivoji testiranja

Razvoj programske opreme poteka tako, da razvijamo najmanjše dele in jih nato sestavljamo v vedno večje enote, vse dokler ne dobimo celotne aplikacije. Podobno izvajamo tudi testiranje, saj ni smiselno najprej testirati aplikacije kot celote, ampak moramo najprej preveriti delovanje manjših delov.

Testiranje delimo na nivoje, od testiranja najmanjših enot preko testiranja večjih enot, vse do testiranja celotnega sistema (slika 1). Vsak nivo vsebuje drugačne vrste napak.

Slika 1: Potek testiranja po nivojih

2.4.1 Testiranje enot

Testiranje enot (*ang. unit testing*) je najnižji nivo testiranja, kjer testiramo najmanjše enote. Pri objektno usmerjenem programiranju te enote imenujemo objekti. Metode objektov testiramo z vnaprej pripravljenimi vhodnimi podatki. Dobljene rezultate nato preverimo s predvidenimi rezultati. S testiranjem enot dosežemo, da objekti delujejo po pričakovanjih v izolaciji. To pomeni, da nas na tem nivoju ne zanima širši kontekst, v katerem so te enote uporabljene znotraj aplikacije.

Ponavadi mora vsak razvijalec sam napisati teste za enote, ki jih razvija, saj sam najbolj pozna njihovo notranje delovanje. Ko posamezne enote delujejo po pričakovanjih, lahko napredujemo na naslednji nivo testiranja.

2.4.2 Integracijsko testiranje

Ko posamezne enote v izolaciji delujejo v skladu z zahtevami, moramo preveriti, če pravilno komunicirajo med sabo. Enote same po sebi delujejo po pričakovanjih, vendar to še ne pomeni, da med sabo tudi pravilno komunicirajo. Vedno obstaja možnost, da med interakcijo enot pride do nepravilnega delovanja. Integracijsko testiranje je torej testiranje kompatibilnosti med vmesniki enot in pričakovanega delovanja pri interakciji med njimi. Je vmesna stopnja med testiranjem posameznih enot in testiranjem celotnega sistema. Enote združujemo oziroma integriramo v večje celote oziroma podsisteme. Začnemo z majhnimi podsistemi, nato dodajamo enote in testiramo vedno večje podsisteme.

Nekompatibilni vmesniki so lahko posledica napake pri načrtovanju, slabe dokumentacije ali slabe komunikacije med razvijalci posameznih enot. Primer nekompatibilnih vmesnikov je zahteva, da ena enota vrača rezultat v metrih, druga enota pa zahteva podatek v centimetrih. V tem primeru pride do tipične integracijske napake.

2.4.3 Sistemsko testiranje

Sistemsko testiranje (*ang. system testing*) preveri, če programska oprema deluje kot celota. Namen sistemskega testiranja je vrednotenje v celoti integrirane aplikacije. Preverimo, če programska oprema dosega zahteve, ki so zapisane v specifikaciji.
Predpogoj za sistemsko testiranje je uspešno opravljeno integracijsko testiranje. Sistemsko testiranje se izvaja v testnem okolju, ki je reproducirano produkcijsko okolje, v katerem bo programska oprema dejansko delovala. To pomeni, da mora imeti testno okolje identične karakteristike kot produkcijsko okolje (enako strojno opremo, enak operacijski sistem, enak strežnik, enako podatkovno bazo, itd.). Sistemsko testiranje po navadi zajema:

- varnostno testiranje,
- obremenitveno testiranje,
- stresno testiranje,
- testiranje grafičnega vmesnika,
- testiranje okrevanja.

2.4.4 Sprejemno testiranje

Pri sprejemnem testiranju (*ang. acceptance testing*) preverimo, če aplikacija zadostuje zahtevam naročnika. Testiranje poteka z uporabniškega vidika in ga izvaja končni uporabnik oziroma naročnik. Tudi v tem primeru se testiranje izvaja v testnem okolju. Sprejemno testiranje je po navadi zadnji korak preden se aplikacija preda v produkcijsko delovanje.
Ko aplikacija uspešno prestane sprejemno testiranje, se preda v produkcijo. Vendar se s tem testiranje še ne konča. Zaradi ugotovljenih odpovedi, ki se pokažejo med uporabo ali zaradi implementacije novih funkcionalnosti, prihaja pogosto do zahtev po spremembah programske opreme. Ob vsaki spremembi programske opreme obstaja možnost, da v kodo vnesemo nove napake. Temu se ne moremo izogniti, lahko pa novo nastale napake pravočasno odkrijemo.

Regresijsko testiranje je vnovično izvajanje že obstoječih testov. Izvajamo jih vsakič, ko spremenimo aplikacijo. Tako ugotavljamo, če spremembe niso povzročile odpovedi. Regresijsko testiranje se izvaja na vseh nivojih testiranja, najpomembnejše pa je na nivoju testiranja enot. Pri regresijskem testiranju pride najbolj do izraza avtomatizacija testiranja, saj zaradi množice testov in časovne dinamike testiranja ni primerno za ročno izvajanje. Z regresijskimi testi zagotavljamo, da spremenjen program ohranja kakovost, ki jo je imel pred spremembami.

2.5 Aktivnosti znotraj procesa testiranja

Testiranje na vsakem posameznem nivoju je sestavljeno iz več aktivnosti. Te aktivnosti lahko razdelimo na tri splošne sklope, ki si sledijo kronološko [4]:

Priprava in načrtovanje testiranja

Je prvi in najpomembnejši del procesa testiranja. Brez pravilnega načrtovanja testiranja ne moremo zagotoviti kakovostnega izvajanja testiranja in posledično kvalitetne programske opreme. Tu določimo cilje in strategijo testiranja. Ugotovimo testne zahteve in na njihovi podlagi ustvarimo testne primere. Določimo tudi časovne okvire testiranja.

Izvajanje testiranja

Ta aktivnost predvideva izvajanje pripravljenih testnih primerov in opazovanje obnašanja aplikacije.

Analiza testiranja

Vsako testiranje je neuporabno brez analize testiranja. V tem koraku ugotavljamo, če je bila med izvajanjem testiranja najdena kakšna odpoved. Če je bila najdena odpoved, poskušamo najti vzrok in zagotoviti, da bo odpravljena. Analiza se uporablja kot povratna informacija.

Dostikrat razvijalci omejijo proces testiranja samo na izvajanje testiranja in izpustijo ostali dve aktivnosti. Takšno testiranje ni smotrno, saj brez pravega načrtovanja testiranja ne moremo zagotoviti kakovostnega izvajanja testiranja. Prav tako testiranje brez analize nima prave vrednosti.

2.6 Metode testiranja

Metode testiranja delimo na dve vrsti: *metoda bele skrinjice* in *metoda črne skrinjice*. Ločita se glede na to, na kakšen način izbiramo testne primere. Izbiramo jih lahko na podlagi zunanjega obnašanja ali notranje strukture programa.

Metoda bele skrinjice

Pri metodi bele skrinjice (strukturnem testiranju) izbiramo testne primere na podlagi notranje strukture aplikacije. Najbolj pogosto se uporablja na nivoju testiranja enot. Namen te metode je, da s testnimi primeri pokrijemo čim več notranjih struktur (poti, veje, zanke). Ker poznamo notranje delovanje, lahko na podlagi notranjih struktur izberemo primerne testne primere in podatke.

Metoda črne škatle

Metodo črne škatle imenujemo tudi funkcionalno testiranje. V primerjavi z metodo bele škatle tu testiramo zunanje obnašanje programa na podlagi vhodnih in izhodnih podatkov. Ker v tem primeru ne vemo nič o notranji zgradbi in algoritmih, testne zahteve pišemo na podlagi specifikacije, v kateri je opisano delovanje. Primer funkcionalnega testiranja je integracijsko testiranje.

2.7 Testiranje kot del razvoja programske opreme

Ker testiranje zagotavlja kakovost programske opreme mora biti del razvoja že od samega začetka. Veliko razvijalcev začne s testiranjem zelo pozno, po navadi šele po končani fazi implementacije ali pa celo po prenosu sistema v ciljno okolje. S takšnim pristopom ne moremo zagotoviti kakovostne programske opreme, saj se kakovosti ne da kar ustvariti, ko je sistem končan. Poleg tega pogosto ni dovolj časovnih in finančnih virov za testiranje. Zaradi omejenih resursov po navadi ni možno izvesti vse aktivnosti znotraj procesa testiranja. Tako po navadi testiranje zajema samo izvajanje testiranja, brez predhodnega načrtovanja in priprave testiranja. Vse to pripomore k temu, da programska oprema na koncu vsebuje ogromno neodkritih napak, ki se potem pokažejo z odpovedjo programske opreme med njeno uporabo. Napake, narejene v posamezni fazi razvoja, se širijo v naslednje faze, kjer jih je težje izslediti.

Če se zgodi napaka v zgodnji fazi razvoja (kot je na primer specifikacija sistema), jo moramo najti in čim prej odpraviti, saj bo v nasprotnem primeru prišlo do napačne implementacije in do napake v končnem izdelku. Napaka v specifikaciji, ki jo ugotovimo, ko je programska oprema že prenesena v ciljno okolje, je zelo težko odpravljiva. V tem primeru moramo odpraviti posledice napake od faze, v kateri se je napaka zgodila, vse do faze, v kateri smo jo ugotovili. Pri sprotnem testiranju programsko opremo izboljšujemo skozi celoten življenjski cikel. S tem preprečimo, da bi se napake širile skozi posamezne faze vse do končnega izdelka.

Življenjski cikel programske opreme

Življenjski cikel programske opreme (*ang. Software development life cycle*), je konceptualni model, ki opisuje faze v razvoju programske opreme. Obstaja več delitev življenjskega cikla, ampak v splošnem vse vsebujejo tradicionalne osnovne faze:

- analiza zahtev in specifikacija sistema,
- načrtovanje sistema in komponent,
- implementacija,
- integracija sistema,
- prenos sistema v ciljno okolje,
- vzdrževanje.

Testiranje lahko implementiramo v vsaki posamezni fazi življenjskega cikla razvoja programske opreme. Dobra praksa je, da teste načrtujemo in pripravimo med vsako fazo, četudi testi ne bodo izvršljivi pred določeno fazo. Že sama priprava testov lahko ugotovi dosti napak v zahtevah in načrtu programske opreme. Na posameznih fazah ima testiranje različen pomen, saj iščemo tudi različne vrste napak.

2.8 Avtomatizacija testiranja

Testiranje programske opreme lahko vzame do 50% časa razvoja programske opreme, po nekaterih ocenah tudi do 70%, kar je seveda povezano z velikimi stroški. Zaradi tega je pomembno, da avtomatiziramo proces testiranja, kolikor se le da. Glavni namen avtomatizacije je, da se poveča produktivnost in zmanjša stroške, ki so povezani s testiranjem. S tem izničimo možnost napak pri testiranju zaradi človeškega faktorja. Po navadi so prve stvari, ki se jih avtomatizira ponavljajoče se naloge, ki jih ni smotrno izvajati ročno. Te naloge lahko avtomatiziramo s posebnimi programskimi orodji. Celotnega procesa testiranja se ne da avtomatizirati, lahko pa avtomatiziramo posamezne aktivnosti testiranja. Aktivnost, ki je najbolj primerna za avtomatizacijo, je aktivnost izvajanja testiranja oziroma izvrševanje testnih primerov. S tem najbolj zmanjšamo čas, namenjen regresijskemu testiranju. Aktivnost priprave in načrtovanja testiranja po navadi ni primerna za avtomatizacijo, lahko pa se avtomatizirajo nekatere podaktivnosti. Tudi analiza testiranja v veliki meri ni primerna za avtomatizacijo.

3 Testiranje enot

Testiranje enot je nivo testiranja, kjer testiramo najmanjše zaokrožene celote programske opreme. V objektno usmerjenem programiranju te zaokrožene celote imenujemo objekti. S testiranjem enot preverjamo, če posamezni objekti delujejo po pričakovanjih. Omogoča nam testiranje notranjih delov aplikacije, ki zagotavljajo delček funkcionalnosti in niso direktno izpostavljeni uporabniku. S testiranjem enot zmanjšujemo tveganje uporabe posameznih objektov. Osnovne zahteve pri testiranju enot so:

- testiranje enot mora biti ponovljivo,
- enote testiramo v izolaciji,
- testi sami preverjajo delovanje enote, ki jo testirajo.

3.1 Prednosti

Testiranje enot ima številne prednosti in koristi. Daje nam povratne informacije o delovanju enot, omogoča nam, da spreminjajmo kodo znotraj enot in služi kot dokumentacija. Uspešno testiranje enot je podlaga za integracijsko testiranje.

Povratne informacije

Najpomembnejša prednost je, da testne primere pišemo na podlagi specifikacije. To pomeni, da lahko napišemo test in izvajamo testiranje hkrati z implementacijo enote. Tako lahko zelo zgodaj med implementacijo izvemo, če enota deluje po pričakovanjih in tako tudi preprečimo, da se napake vnesejo v kodo.
Druga možnost je, da napišemo teste, še preden je enota sploh implementirana. Tak pristop uporablja testno voden razvoj programske opreme (*ang. test driven development*). Pri tem pristopu napišemo najprej teste enote, šele nato začnemo z njeno implementacijo. Za izvedbo testiranja enote rabimo samo informacije o njenem vmesniku. S tem pristopom dobivamo povratne informacije o delovanju enot že med samo implementacijo enote. Tak način pomaga pri implementaciji enote in pospešuje razvoj.

Spreminjanje kode

Vsaka programska oprema se spreminja skozi čas. Spreminjanje kode (*ang. refactoring*) je rekonstrukcija programske opreme na način, ko naredimo notranje spremembe, ki ne spremenijo zunanjega delovanja [7].
Vsakič, ko naredimo majhno spremembo v kodi, obstaja možnost, da naredimo napako. Testiranje enot nam omogoča, da varno spreminjamo in popravljamo kodo. Testi enot delujejo kot varnostna mreža. Ob vsaki spremembi kode s testi preverimo, če ob spremembi

kode ni prišlo do spremembe v funkcionalnosti. Če je prišlo do spremembe v funkcionalnosti, pomeni, da smo naredili napako.

Dokumentacija

Testi enot služijo kot dokumentacija za razvijalce. Drugi razvijalci lahko zelo hitro na podlagi testov ugotovijo, kako enote delujejo.

3.2 Omejitve

Kot vsako testiranje, ima tudi testiranje enot določene omejitve:

- ne moremo najti vseh odpovedi,
- ne moremo prikazati odsotnost napak, ampak le njihovo prisotnost.

Ob pomanjkljivem testiranju enot se neodkrite napake pokažejo kot odpovedi na ostalih nivojih testiranja.

3.3 Zgradba testa

Osnovi princip vsakega testnega primera je, da na podlagi določene akcije (dejanja) nad testiranim objektom, preverjamo njegov odziv. Če objektu, ki ga testiramo, podamo točno določene vhodne podatke, moramo dobiti točno določene izhodne podatke (odziv).
Testni primer lahko zaokrožimo s štirimi zaporednimi fazami [4]:

- postavitvena faza,
- vadbena faza,
- verifikacijska faza,
- zaključna faza.

Postavitvena faza

Prva faza je postavitvena faza (*ang. setup*). Tu kreiramo enoto, ki jo želimo testirati. Po potrebi jo postavimo v določeno stanje. Če enota uporablja odvisne objekte, jih ustvarimo in nastavimo testirani enoti.

Vadbena faza

Ko je enota, ki jo testiramo v zahtevanem stanju, sledi vadbena faza (*ang. exercise*). V tej fazi poteka interakcija z enoto. To po navadi pomeni klicanje metode, katere funkcionalnost želimo preveriti. Potrebno je poudariti, da lahko funkcionalnost posamezne testne metode testiramo iz različnih aspektov, zato lahko ima ena metoda več testov.

Verifikacijska faza

Rezultate, ki jih dobimo v vadbeni fazi, preverimo v verifikacijski fazi. Poznamo dve glavni vrsti verifikacije [11]:

- verifikacija stanja,
- verifikacija obnašanja.

Pri verifikaciji stanja preverimo, če je stanje, v katerem se testirana enota nahaja, v skladu s pričakovanji. Stanje lahko preverimo tako, da pokličemo metodo, in preverimo njen odziv.
Pri verifikaciji obnašanja, pa preverjamo, če se je med vadbeno fazo enota pravilno obnašala. To pomeni, da preverjamo, če so bili izvedeni pravilni klici metod do odvisnih objektov.
V verifikacijski fazi se odloči, ali je bil test uspešen ali ne.

Zaključna faza

Na koncu sledi še zaključna faza (*ang. teardown*), ki pa se pri testiranju enot ne uporablja. Uporablja se predvsem pri integracijskem testiranju, kjer se uporabljajo viri, ki jih je potrebno na koncu zapreti oziroma postaviti v začetno stanje.

Na tem osnovnem zaporedju sloni vsak test. Najprej kreiramo enoto, ki jo želimo testirati. Nato kreiramo odvisne objekte in jih nastavimo enoti. Po potrebi enoto postavimo v zahtevano stanje. V naslednjem koraku kličemo metodo, katere funkcionalnost želimo preveriti. Na koncu vsakega testa naredimo verifikacijo, s katero preverimo, če se enota obnaša po pričakovanjih.

3.4 Izolacija enot

Iz definicije testiranja enot izhaja, da testiramo objekte v popolni izolaciji. Zahteva po popolni izolaciji testiranja je najpomembnejši koncept testiranja enot, ki zagotavlja kakovostno testiranje.
Popolna izolacija pomeni, da v vsakem testu vedno testiramo samo en objekt. Problem nastane, ker je po navadi vsak objekt tako ali drugače odvisen od funkcionalnosti drugih objektov. Tem objektom pravimo odvisni objekti (*ang. dependencies*).
V primeru, da testiramo enoto skupaj z odvisnimi objekti, lahko pride do odpovedi v katerem izmed odvisnih objektov. Zaradi tega ne vemo nikdar natančno, če je prišlo do napake v enoti ali v katerem izmed odvisnih objektov.
Ideja izolacije je, da se razmejimo od teh odvisnih objektov. Na nivoju testirana enot namesto pravih implementacij odvisnih objektov, uporabljamo t.i. nadomestne oziroma simulirane objekte. S tem omogočimo testiranje primarne enote v popolni izolaciji [12].

3.4.1 Nadomestni objekti

Nadomestni objekti so posebni objekti, ki jih uporabljamo namesto pravih objektov. Simulirajo delovanje pravih objektov, vendar tako, da mi kontroliramo njihovo delovanje.
Najpogosteje uporabljeni nadomestni objekti so: lažni objekti (*ang. dummy objects*), ponaredki (*ang. fake objects*), nastavki (*ang. stub objects*), imitirani objekti (*ang. mock objects*) [8].

Lažni objekti

Ti objekti ponavadi niso nikdar uporabljeni. To pomeni, da se nikdar ne kliče njihovih metod. Zato tudi nimajo delujoče implementacije. Uporabljajo se samo za to, da zapolnijo zahtevane parametre metod.

Ponaredki

Ponaredki imajo delujoče implementacije vmesnika, vendar je funkcionalnost dosti bolj poenostavljena kot pri pravih objektih. Po navadi se uporabljajo na nivoju integracijskega testiranja, v primeru ko določeni viri niso na voljo ali pa rabimo enostavnejšo implementacijo. To so na primer podatkovna baza, datotečni sistem, zunanji sistem ali vir podatkov.

Nastavki

So kontrolirani objekti, ki simulirajo delovanje pravega objekta. Ti objekti se odzivajo na klicane metode z vnaprej pripravljenimi odzivi (podatki). Njihovo delovanje je po navadi omejeno zgolj na funkcionalnost, ki se bo uporabljala znotraj testa.

Imitirani objekti

Ti objekti so vnaprej programirani objekti s pričakovanji. Pričakovanja so zaporedja interakcij, ki ga testirana enota izvede nad imitiranim objektom. Gre za pričakovanja testirane enote, katere metode imitiranega objekta mora klicati.
Pri testiranju z imitiranimi objekti se izvaja verifikacija obnašanja, kar pomeni, da preverimo, če se je objekt uporabljal v skladu z predhodno določenimi pričakovanji.

V okviru testiranja enot se po navadi uporabljajo nastavki in imitirani objekti.

4 Ogrodja za testiranje enot

Ogrodje (*ang. framework*) je zbirka kode oziroma knjižnic, ki omogoča funkcionalnost določeni vrsti aplikacij.
Medtem ko običajne knjižnice omogočajo neko specifično funkcionalnost, ogrodje ponuja širšo paleto funkcionalnosti. Glavna razlika med knjižnico in ogrodjem je v nadzoru. Pri knjižnici ima nadzor aplikacija, ki uporablja funkcionalnost knjižnice. Ogrodje pa uporablja princip obrnjenega nadzora (*ang. inversion of control*). To pomeni, da ima ogrodje nadzor nad funkcionalnostjo aplikacije in njenim izvajanjem [9].
Testno ogrodje standardizira strukturo testov, omogoča avtomatizirano izvajanje testov in vrača povratne informacije o poteku testiranja. Ogrodja za testiranje enot (*ang. unit testing frameworks*) so v prvi vrsti nastala kot odgovor na zahtevo po avtomatizaciji testiranja. Avtomatizacija testiranja pomeni, da uporabimo orodje, ki omogoča:

- kontroliranje izvajanja testov,
- verifikacijo rezultatov (primerjava pričakovanih rezultatov z dobljenimi),
- povratne informacije o poteku testiranja,
- neodvisnost posameznih testov,
- predstavitev rezultatov testiranja.

Ogrodje za testiranje enot torej postavlja okolje, ki omogoča pisanje testov in izvajanje testiranja. Ima nadzor nad izvajanjem enot in od zunaj preverja njihovo delovanje. Predstavlja celovito rešitev izvajanja testiranja in organizacije testiranja enot.

4.1 Zgodovina

Ogrodja za testiranje enot so znana pod splošnim imenom xUnit. Najdemo jih pri večini modernih objektno usmerjenih programskih jezikih. Prvo ogrodje se je imenovalo SUnit in je bilo namenjeno avtomatiziranemu testiranju enot v programskem jeziku Smalltalk. Avtor ogrodja SUnit je bil Kent Beck [13].
Principi, ki so bili združeni v tem ogrodju, so se prenesli v programski jezik Java in nastalo je ogrodje JUnit [2]. Začetki ogrodja segajo v leto 1997. Imelo je izjemen vpliv na področje testiranja enot, saj je bilo prvo splošno znano ogrodje za testiranje enot, na podlagi katerega so nastala ogrodja tudi v drugih programskih jezikih [6]. JUnit je dolgo časa bilo edino splošno znano ogrodje za testiranje enot v Javi. Kot odgovor na nekatere pomanjkljivosti in omejitve, je leta 2004 nastalo ogrodje TestNG.

4.2 Osnovni pojmi

Testni razred

Testni razred je zaokrožena celota testov, ki testirajo isti objekt (enoto). Testni razred je Java razred, ki vsebuje vsaj eno testno metodo, ki je označena z javansko oznako (*ang. annotation*) [5]. Testne metode znotraj posameznega testnega razreda testirajo isto enoto. Posamezne testne razrede lahko združujemo v večje množice z namenom, da jih poganjamo kot celoto.

Testna metoda

Testna metoda je implementacija testnega primera. Znotraj testne metode testiramo delček funkcionalnosti enote.

Trditve

Najpomembnejši del znotraj testne metode se zgodi na koncu, ko preverimo delovanje enote. Preverjanje delovanja enote poteka preko posebnih metod ogrodja imenovanih trditve (*ang. assert*). Trditve so vnaprej definirane pogojne metode ogrodja, s katerimi preverimo, če se testiran objekt obnaša po pričakovanjih. Najpogostejša trditev je, da morata biti dva podatka enaka. Pričakovan rezultat mota biti torej enak rezultatu, ki ga vrača metoda testiranega objekta.

4.3 JUnit

Je odprtokodno ogrodje za avtomatizirano testiranje enot v programskem jeziku Java. Avtorja ogrodja sta Erich Gamma in Kent Beck. Nastalo je leta 1997 na podlagi ogrodja SUnit in bilo prvo splošno znano ogrodje za testiranje enot [6]. Ogrodje je imelo ogromen vpliv na dojemanje testiranja enot. Dolgo časa je postavljalo smernice testiranja enot v Javi in tudi širše, saj so na njegovi osnovi nastala ogrodja v drugih programskih jezikih. Vse do današnjega dne velja kot *de-facto* standard na področju testiranja enot.

Testni razred

Celotna konfiguracija testnih razredov poteka preko javanskih oznak. Testne metode označujemo z `@Test`. Tako ogrodju povemo, da je določena metoda test (slika 2). Pri definiciji testnih metod smo omejeni z dvema zahtevama. Testne metode ne smejo:

- vračati vrednosti (vračajo `void`),
- sprejemati parametrov.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void divideWithNegativeNumber() {
        Calculator calculator = new Calculator();
        long result = calculator.divide(10, -5);

        assertEquals(result, -20);
    }

    @Test
    public void addPositiveNumbers() {
        Calculator calculator = new Calculator();
        long result = calculator.add(10, 20);

        assertEquals(result, 30);
    }

}
```

Slika 2: Primer osnovnega testnega razreda v ogrodju JUnit

Življenjski cikel testnega razreda

Testiranje poteka tako, da ogrodje znotraj testnega razreda poišče vse testne metode in jih začne izvajati. Izvedba testne metode v osnovi zajema cikel treh zaporednih akcij:

1. kreiranje instance testnega razreda,
2. klicanje testne metode,
3. uničenje instance testnega razreda.

Bistven poudarek pri tem je, da se vsaka testna metoda izvede v okviru lastne instance. To pomeni, da se po vsaki izvedbi testne metode zavrže trenutna instanca testnega razreda in se ustvari nova za izvedbo naslednje testne metode. To obnašanje sloni na zahtevi po neodvisnosti posameznih testnih metod. S tem se onemogoči prenašanje stanj med posameznimi testnimi metodami.

```java
public class CounterTest {

    private int stevec = 0;

    @Test
    public void count1() {
        this.stevec++;
        assertEquals(1, stevec);
    }

    @Test
    public void count2() {
        this.stevec++;
        assertEquals(1, stevec);
    }
}
```

Slika 3: Primer omejitve prenašanja stanj

Na sliki 3 je primer testnega razreda v JUnit. V tem primeru povečanje atributa `stevec` iz vrednosti 0 na vrednost 1, znotraj metode `count1` nima nobenega vpliva na ostale metode. Ob izvedbi metode `count2`, je vrednost `stevec` spet enaka 0, saj se ustvari nova instanca. Tako je vrednost števca ob klicu katere koli metode enaka 0.

Izvajanje testnega razreda

Za izvajanje testnega razreda skrbijo testni sprožilci (*ang. test runners*). To vključuje klicanje in izvajanje testnih metod, preverjanje pravilnosti rezultata in zbiranje rezultatov. Če sprožilca ne navedemo, se testni razred izvede s privzetim sprožilcem `JUnit4`. Če hočemo razred zagnati s katerim drugim sprožilcem, uporabimo oznako `@RunWith` in podamo ime sprožilca. Testni razredi, ki uporabljajo podatkovno vodeno testiranje, uporabljajo sprožilec `Parameterized`. Testne zbirke pa uporabljajo sprožilec `Suite`.

Trditvene metode

Trditvene metode so zbrane v razredu `Assert`. Osnovna trditev je metoda `assertEquals`. S to metodo preverjamo, če sta dva objekta ali dva primitivna tipa enaka. Pri JUnit kot prvi parameter navedemo sporočilo, ki je opcijski. Drugi parameter je pričakovan rezultat, tretji pa dejanski rezultat. Znotraj razreda najdemo tudi dodatne trditvene metode. To so `assertNotEquals`, `assertTrue`, `assertFalse`, `assertNull`, `assertNotNull`.

Testiranje izjem

Pri testiranju enot preverjamo, če se enote odzivajo po pričakovanjih. To pa ne pomeni samo, da preverimo, če se metode pravilno odzivajo v primeru pravilnih vhodnih podatkov, ampak tudi, da se pravilno odzivajo v primeru neveljavnih vhodnih podatkov. V ogrodju JUnit lahko preverimo, če enote pod določenimi pogoji vračajo pričakovane izjeme. To naredimo tako, da oznaki `@Test` dodamo parameter `expected`, s katerim povemo katero izjemo pričakujemo (slika 4).

```
@Test(expected = ArithmeticException.class)
public void divideByZeroThrowsException() {
    Calculator calculator = new Calculator();
    calculator.divide(4, 0);
}
```

Slika 4: Testiranje izjem v ogrodju JUnit

V tem primeru na koncu testne metode ne uporabljamo trditev. Enota deluje po pričakovanjih v primeru, da se zgodi izjema, ki smo jo navedli v oznaki.

Podatkovno vodeno testiranje

Podatkovno vodeno testiranje (*ang. data driven testing*) pomeni, da posamezno testno metodo izvedemo z različnimi vhodnimi podatki. Za uporabo podatkovno vodenega testiranja moramo označiti testni razred z oznako `@RunWith(Parameterized.class)` (slika 5). Potrebujemo še:

- statično metodo, ki vrača testne podatke,
- konstruktor, prek katerega nastavljamo testne podatke in
- testno metodo, ki jo izvajamo s testnimi podatki.

Na ta način ločimo testno metodo od podatkov, s katerimi testiramo. Metodo, ki vrača testne podatke, označimo z `@Parameters`. Testni podatki se vračajo kot zbirka tabel. Posamezna tabela je sestavljena iz vhodnih podatkov in pripadajočih izhodnih podatkov, ki se uporabijo pri posamezni izvedbi testne metode. Definiramo lahko statične testne podatke, lahko pa implementiramo branje podatkov iz zunanjih virov, kot so podatkovna baza, datoteka.

```
@RunWith(value=Parameterized.class)
public class CalculatorDataTest {

    private long a;
    private long b;
    private long expectedResult;

    public CalculatorDataTest(long a, long b, long expectedResult) {
        this.a = a;
        this.b = b;
        this.expectedResult = expectedResult;
    }

    @Parameters
    public static Collection<Object[]> multiplyProvider() {
        Object[][] data = {
                {2,4,8},{3,2,6},{10,2,20}
        };
        return Arrays.asList(data);
    }

    @Test
    public void multiplyWithData() {
        Calculator calculator = new Calculator();
        long result = calculator.multiply(a, b);

        assertEquals(result, expectedResult);
    }
}
```

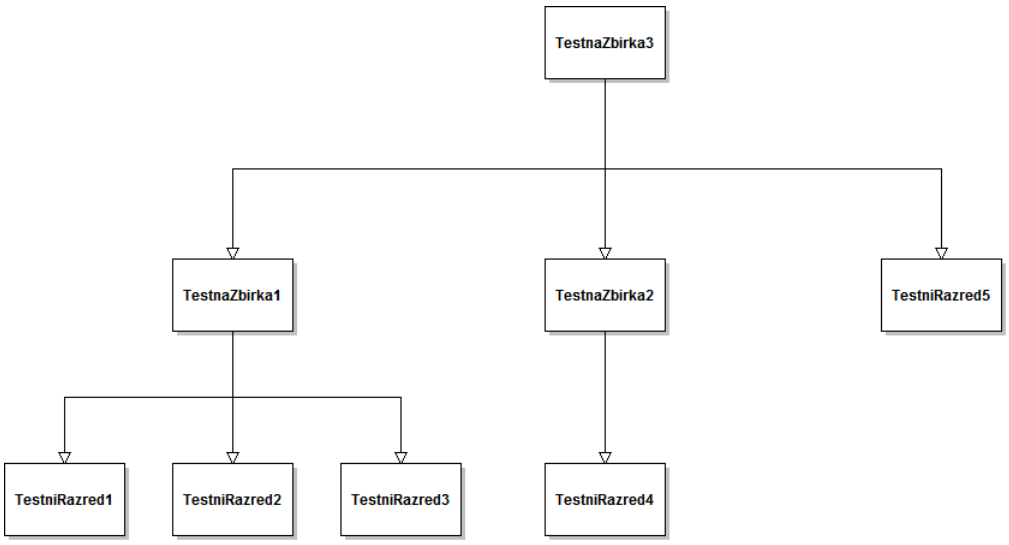
Slika 5: Podatkovno vodeno testiranje v JUnit

Testne metode v JUnit ne morejo imeti parametrov, zato se testni podatki nastavljajo prek konstruktorja razreda. V konstruktorju se zapišejo v atribute razreda, nato jih uporabi testna metoda (slika 5).

Pristop podatkovno vodenega testiranja nam daje določeno fleksibilnost, saj lahko posamezno testno metodo testiramo z več različnimi vhodnimi podatki.

Združevanje testnih razredov

Posamezne testne razrede lahko združujemo v večje množice. V JUnit tem množicam pravimo testne zbirke (*ang. test suites*). Testna zbirka je testni razred, ki združuje druge testne razrede ali zbirke. Testne zbirke lahko sestavljamo do poljubne globine in širine (slika 6).



Slika 6: Primer organizacije testnih zbirk v JUnit

Testno zbirko kreiramo kot prazen razred, ki nima nobenih metod, in jo označimo s pripadajočim sprožilcem `@RunWith(Suite.class)`. Poleg tega moramo uporabiti tudi oznako `@SuiteClasses`, znotraj katere navedemo seznam testnih razredov, ki jih testna zbirka združuje (slika 7). Ko izvedemo testno zbirko, se izvedejo vsi testni razredi, ki jih zajema.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses(value = {
        CalculatorDataTest.class,
        CalculatorTest.class
})
public class AllTests {

}
```

Slika 7: Primer definicije testne zbirke v JUnit

Izvajanje testiranja

Testne metode, testne razrede in testne zbirke lahko poganjamo na tri načine:

- prek ukazne vrstice,
- prek grafičnega vmesnika razvojnega okolja ali
- programsko.

Najbolj pogost način poganjanja testov je prek razvojnega okolja, kot sta npr. Eclipse ali NetBeans. JUnit je v ta razvojna okolja integriran kot vtičnik, prek katerega zelo enostavno kreiramo in poganjamo testne razrede, zbirke ali posamezne testne metode. Na koncu vsakega testiranja dobimo predstavitev rezultatov in opis morebitnih najdenih odpovedi.

4.4 TestNG

TestNG (*Test New Generation*) je ogrodje za testiranje, ki je v prvi vrsti namenjeno testiranju enot in integracijskemu testiranju. Njegovi začetki segajo v leto 2004. Osnovano je bilo na ogrodju JUnit (verzija 3) in je nastalo kot odgovor na omejitve, ki jih je takrat imel JUnit. Namen ogrodja TestNG je bil omogočiti funkcionalnosti, ki v JUnit niso bile na voljo [3]. Najpomembnejše funkcionalnosti s področja testiranje enot, ki jih je uvedel in takrat niso bile na voljo v JUnit [1]:

- konfiguracija z javanskimi oznakami,
- testiranje izjem,
- testne metode sprejemajo parametre in
- definicija organizacije testiranja v konfiguracijski datoteki.

Uvedel je tudi druge funkcionalnosti, ki pa se večinoma uporabljajo izven področja testiranja enot.

Testni razred

Celotna konfiguracija testnih razredov temelji na javanskih oznakah. Testna metoda mora biti označena z oznako `@Test` (slika 8). To oznako lahko prav tako uporabimo na nivoju testnega razreda. V tem primeru se kot testne metode smatrajo vse javne metode razreda, ki niso drugače označene.

```
import static org.testng.Assert.assertEquals;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class CalculatorTest {

    private Calculator calculator;

    @BeforeMethod
    private void init() {
        this.calculator = new Calculator();
    }

    @Test(expectedExceptions = ArithmeticException.class)
    public void divideByZeroThrowsException() {
        this.calculator.divide(4, 0);
    }

    @Test
    public void divideWithNegativeNumber() {
        long c = this.calculator.divide(10, -5);
        assertEquals(c, -20);
    }

    @Test
    public void multiplyWithZeroIsZero() {
        long c = this.calculator.multiply(100, 0);
        assertEquals(c, 0);
    }

}
```

Slika 8: Primer testnega razreda v ogrodju TestNG

Življenjski cikel testnega razreda

TestNG uporablja eno instanco za izvedbo vseh testnih metod znotraj testnega razreda. Najprej se ustvari instanca testnega razreda, nato pa se izvajajo testne metode ena za drugo. Ko se izvedejo vse testne metode, se instanca testnega razreda uniči. To pomeni, da testne metode uporabljajo isto instanco testnega razreda.

```
package com.inf;

import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class CalculatorInstanceTest {

    private Calculator calculator = new Calculator();

    @Test
    public void first() {
        calculator.add(40);
        int result = calculator.getResult();
        assertEquals(result, 40);
    }

    @Test
    public void second() {
        calculator.add(40);
        int result = calculator.getResult();
        assertEquals(result, 40);
    }

}
```

Slika 9: Primer prenašanja stanja v TestNG

Na primeru iz slike 9 bomo pokazali primer prenašanja stanj. Najprej se ustvari instanca testnega razreda in se inicializira atribut `calculator` z začetno vrednostjo 0. Ob izvedbi metode `first` se atributu `calculator` prišteje vrednost 40. Potem se preveri, če je rezultat 40 in test ne najde napak. V naslednjem koraku se izvede metoda `second`, ampak ker atribut `calculator` vsebuje vrednost iz prejšnje testne metode in ne začetno vrednost 0, bo rezultat (po prištevanju vrednosti 40) enak 80 in testna metoda bo vrnila napako.

```
public class CalculatorInstanceTest {

    private Calculator calculator;

    @BeforeMethod
    public void init() {
        calculator = new Calculator();
    }

    @Test
    public void first() {
        calculator.add(40);
        int result = calculator.getResult();
        assertEquals(result, 40);
    }

    @Test
    public void second() {
        calculator.add(40);
        int result = calculator.getResult();
        assertEquals(result, 40);
    }

}
```

Slika 10: Primer postavitvene metode v TestNG

Prenašanju stanj se izognemo tako, da ustvarimo postavitveno metodo in jo označimo z oznako `@BeforeMethod` (slika 10). Ta metoda se bo izvedla pred vsako testno metodo in bo atribut `calculator` vsakič na novo inicializirala.

Trditvene metode

Trditvene metode so zelo podobne kot v ogrodju JUnit. Največja razlika je, da sprejemajo parametre v drugačnem vrstnem redu. Osnovna trditvena metoda zgleda tako: `assertEquals(dobljenRezultat, pricakovanRezultat, sporocilo)`. Parametra dobljen rezultat in pričakovan rezultat sta zahtevana, parameter sporočilo pa je opcijski.

Testiranje izjem

Oznaki `@Test` dodamo parameter `expectedException`. Naštejemo eno ali več izjem, ki jih pričakujemo.

```
@Test(expectedExceptions = ArithmeticException.class)
public void divideByZeroThrowsException() {
    Calculator calc = new Calculator();
    calc.init(44);
    calc.divideBy(0);
}
```

Slika 11: Testiranje izjem v TestNG

Kot je razvidno iz slike 11, `calc.divideBy(0)` zaradi deljenja z vrednostjo 0 sproži izjemo tipa `ArithmeticException`. Enota v tem primeru deluje po pričakovanjih, saj se je zgodila izjema, ki smo jo navedli v oznaki.

Podatkovno vodeno testiranje

Pri podatkovno vodenem testiranju potrebujemo metodo, ki vrača vhodne in izhodne podatke. Ta metoda se imenuje podatkovni ponudnik (*ang. data provider*) in ima oznako `@DataProvider`.

```
public class CalculatorDataTest {

    @DataProvider(name="_multiplyData")
    public Object[][] multiplyProvider() {

        Object[][] data = new Object[][] {
            {1,2,2},
            {2,4,8},
            {10,2,20},
            {12,5,60}
        };

        return data;
    }

    @Test(dataProvider = "_multiplyData")
    public void multiplyRandomData(long a, long b, long expectedResult) {

        Calculator calculator = new Calculator();
        long result = calculator.multiply(a, b);

        assertEquals(result, expectedResult);
    }
}
```

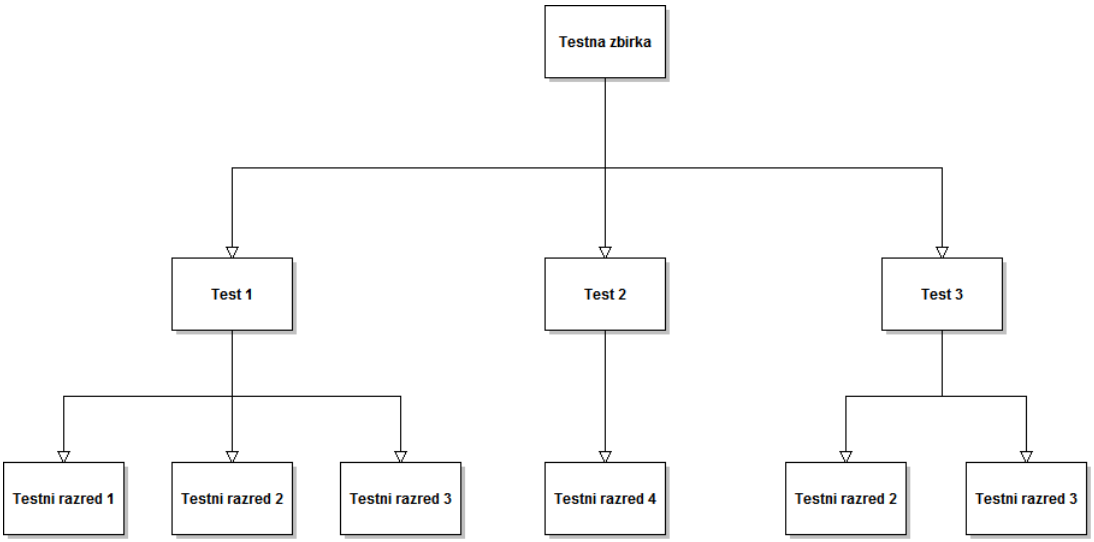
Slika 12: Podatkovno vodeno testiranje v TestNG

Metoda vrača podatke kot dvodimenzionalno tabelo tipa `Object[][]`. Vsak `Object[]` znotraj tabele predstavlja posamezen niz podatkov, ki jih ogrodje v vsaki iteraciji poda testni metodi preko parametrov. Pri testni metodi, v kateri želimo uporabiti zbirko podatkov, oznaki `@Test` dodamo parameter `dataProvider`. Prek njega se sklicujemo na podatkovnega ponudnika (slika 12). Tak način nam omogoča, da imamo več podatkovnih ponudnikov, saj lahko vsaka testna metoda uporablja svojega. Ogrodje pred izvedbo testne metode pokliče metodo s podatki, ki je navedena v oznaki. Nato z dobljenimi podatki iterativno kliče testno metodo in ji posreduje podatke preko parametrov.
Testne podatke lahko pridobimo iz zunanjega vira, lahko pa jih tudi definiramo v konfiguracijski datoteki, vendar smo v tem primeru omejeni na primitivne tipe.

Zbirke testnih razredov

TestNG ima malce specifično terminologijo za definiranje organizacije testiranja. Uporabljata se dva termina:

- `test` (*ang. test*), ki ga sestavlja en ali več testnih razredov,
- `testna zbirka` (*ang. test suite*), ki jo sestavlja en ali več testov.



Slika 13: Primer organizacije testiranja v TestNG

Testna zbirka je najvišji nivo, ki zajema celotno organizacijo testiranja (slika 13). Po navadi definiramo eno testno zbirko za en projekt. Celotno organizacijo definiramo v konfiguracijski datoteki.

Konfiguracijska datoteka

Datoteka `testng.xml`, je XML dokument, v katerem definiramo testno zbirko in nastavitve testiranja (slika 14). Definicija v XML datoteki nam omogoča enostavno definiranje testne zbirke in izvajanja testiranja, saj jih lahko priredimo lastnim potrebam in željam. V

posamezen test lahko vključimo ali izključimo posamezne testne metode, razrede ali celotne pakete.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Calculator Suite">
  <test name="Unit tests" preserve-order="false">
    <classes>
      <class name="com.inf.CalculatorTest"/>
      <class name="com.inf.CalculatorDataTest"/>
    </classes>
  </test>
  <test name="Integration tests">
    <classes>
        <class name="com.inf.integration.MyIntegrationTest" />
    </classes>
  </test>
  <listeners>
    <listener class-name="com.inf.testng.ext.MyTestListener"></listener>
  </listeners>
</suite>
```

Slika 14: Primer datoteke testng.xml

Zbirki lahko pripnemo tudi poslušalce (*ang. listeners*), preko katerih lahko spreminjamo potek testiranja.

Izvajanje testiranja

Testiranje lahko izvedemo na več načinov:

- prek ukazne vrstice,
- prek grafičnega vmesnika razvojnega okolja (IDE) ali
- programsko.

5 Primerjava orodij za testiranje enot

V tem poglavju bomo pozornost namenili primerjavi ogrodij **JUnit** in **TestNG**. Primerjava je narejena na podlagi zadnjih stabilnih verzij obeh ogrodij (JUnit 4.9 in TestNG 6.2).

Ogrodja za testiranje enot ponujajo enako splošno funkcionalnost, vendar vseeno prihaja do razlik med njimi, saj so funkcionalnosti v vsakem ogrodju implementirane drugače. Z enim ogrodjem lahko naredimo stvari, ki jih z drugim ne moremo, eno ogrodje poenostavlja proces testiranja, drugo ima razširjeno funkcionalnost, itd.

Namen primerjave je pokazati, katero ogrodje bolje pokriva zahteve na področju testiranja enot. Primerjava je narejena z zornega kota razvijalca, ki ima kot uporabnik ogrodja določene zahteve do ogrodja. Primerjava ni narejena z zornega kota določenega ogrodja, ampak kot nevtralen pogled na ogrodja za testiranje enot. Pri primerjavi je najpomembnejše vodilo uravnotežena selekcija zahtev in smiselno oblikovanje omejenega števila kriterijev.
Primerjavo bomo izvedli v treh korakih. Najprej bomo izvedli analizo zahtev, s katero bomo ugotovili zahteve, ki jih imamo do ogrodja. Kaj mora omogočati in na kakšen način. Na podlagi zahtev bomo oblikovali kriterije. Na podlagi posameznih kriterijev bomo naredili primerjavo med ogrodjema. Ugotovili bomo, kakšne razlike so med njima in v kakšni meri pokrivata zahteve v okviru kriterijev. Na koncu poglavja bomo predstavili povzetek primerjave in ugotovitve, do katerih smo prišli. Ugotovili bomo, katero ogrodje najboljše izpolnjuje naše kriterije.

5.1 Analiza zahtev

Najprej moramo ugotoviti katere zahteve imamo do ogrodja za testiranje enot. Osredotočili se bomo predvsem na vidik uporabnosti ogrodja. Glavni zahtevi sta po našem mnenju enostavnost uporabe in prilagajanje funkcionalnosti ogrodja.
Ti zahtevi se nanašata predvsem na:

- enostavnost pisanja testov,
- enostavnost izvajanja testiranja,
- hitrost osvojitev konceptov ogrodja,
- prilagajanje delovanja in
- razširitev funkcionalnosti.

To so po našem mnenju ključne zahteve, ki jih ogrodje za testiranje enot mora omogočati.

5.2 Kriteriji primerjave

Skozi kriterije, ki smo jih izbrali na podlagi zahtev, bomo izpostavili prednosti in slabosti posameznega ogrodja. Preverili bomo, kakšna je uporabnost določenih konceptov in funkcionalnosti znotraj ogrodij.

5.2.1 Življenjski cikel testnega razreda

Osrednja komponenta ogrodja je testni razred. Vsak testni razred ima določen življenjski cikel, na podlagi katerega se izvajajo njegove testne metode. V okviru tega kriterija si bomo pogledali, na kakšen način posamezno ogrodje implementira ta življenjski cikel. Ogrodji uporabljata različne pristope, ki temeljijo na različnih predpostavkah.

JUnit

Kot je bilo omenjeno v prejšnjem poglavju, je JUnit zasnovan z zahtevo po neodvisnosti testnih metod. Ogrodje naredi novo instanco testnega razreda za vsako testno metodo. S tem doseže popolno neodvisnost testnih metod. Tako ima vsaka metoda na voljo sveže stanje vseh spremenljivk in se ne more zgoditi, da bi prišlo do nepravilnega delovanja zaradi stanja spremenljivk v predhodno izvedenih metodah. Prednost je tudi v tem, da nam ni potrebno skrbeti, kaj se bo zgodilo na koncu testne metode, saj se instanca zavrže.

TestNG

TestNG uporablja drugačen pristop glede izvajanja testov. Tukaj se vse testne metode posameznega testnega razreda izvedejo znotraj iste instance. Tako mora vsak razvijalec sam poskrbeti za neodvisnost testnih metod. TestNG se poleg testiranja enot uporablja tudi za integracijsko testiranje. Pri integracijskem testiranju testne metode uporabljajo skupne vire, zato se uporablja ohranjanje stanj med posameznimi testnimi metodami. Ob nepazljivosti lahko hitro pride do prenašanja stanj, kar pa vodi v nepredvidljivo delovanje. Ohranjanju stanj pri testiranju enot se izognemo tako, da:

- znotraj testnih metod uporabljamo samo lokalne spremenljivke, ki niso dosegljive v ostalih testnih metodah,
- v pripravljalni metodi vse skupne objekte na novo inicializiramo,
- v zaključni metodi skupne objekte postavimo nazaj v prvotno stanje.

Testne metode znotraj testnega razreda se v obeh ogrodjih izvajajo nedeterministično. To pomeni, da ogrodje ne zagotavlja, da se bodo testne metode vedno izvedle v enakem zaporedju. Zaradi tega tudi ni smiselno načrtno prenašati vrednosti med testnimi metodami, saj lahko prihaja do nepredvidljivih težav.

Prednosti in slabosti:

- **JUnit**
  - ✓ popolna neodvisnost testnih metod.
- **TestNG**
  - × nepredvidljivi rezultati testiranja ob nepazljivosti.

Neodvisnost testnih metod je del načina delovanja ogrodja JUnit, medtem ko moramo pri TestNG za to sami poskrbeti. Znotraj področja testiranje enot vlada velik razkorak zaradi obeh pristopov. Zagovorniki izolacije poudarjajo, da mora biti ta zahteva že vgrajena v samo ogrodje, kot je pri JUnit. Nasprotniki pa zagovarjajo mišljenje, da ogrodje razvijalca ne sme omejevati, zato mora za neodvisnost metod poskrbeti razvijalec sam. Tak princip uporablja TestNG.
Po našem mnenju mora biti zagotavljanje neodvisnosti metod funkcionalnost ogrodja samega, saj je to temeljna zahteva pri testiranju enot. Zato menimo, da je boljša izbira ogrodje JUnit.

Izbira: JUnit

5.2.2 Konfiguracija testnih razredov

Obe testni ogrodji omogočata konfiguracijo testnih razredov z javanskimi oznakami. Z oznakami testnim razredom in metodam dodajamo metapodatke, prek katerih ogrodje dobi potrebne informacije za izvedbo testiranja.
Pred nastankom TestNG v ogrodju JUnit ni bilo konfiguracije z oznakami. Vsak testni razred je moral razširiti osnovni razred imenovan `TestCase`. Bili smo omejeni pri poimenovanju testnih metod, saj so se imena morala začeti s prefiksom `test`, da je ogrodje vedelo, katere metode so testi. V današnjih verzijah teh omejitev ni več, saj večino konfiguracije opravimo z oznakami. Najbolj osnovna konfiguracija znotraj testnega razreda zajema:

- oznaka testne metode,
- oznaka za ignoriranje testne metode,
- oznaka pričakovane izjeme.

JUnit

Testne metode označimo z oznako `@Test`. Oznako uporabljamo pred vsako metodo v razredu, ki jo želimo označiti kot test. Ogrodje nam omogoča, da testiramo pričakovan odziv v obliki izjeme. To storimo tako, da testno metodo označimo z `@Test(expected=PricakovanaIzjema.class)`. V tem primeru je test uspešen, če enota vrne izjemo, ki je zapisana znotraj oznake. Včasih hočemo kakšno testno metodo izključiti iz testiranja. Bodisi metoda, ki jo testiramo, ne obstaja več bodisi je nočemo testirati v tistem trenutku. To naredimo z oznako `@Ignore`. V tem primeru bo JUnit metodo preskočil.

TestNG

Tudi tukaj testne metode označimo z oznako `@Test`. Vendar TestNG omogoča širšo uporabo te oznake, saj jo lahko uporabimo tudi na nivoju testnega razreda. To pomeni, da vse javne metode, ki niso drugače označene, smatramo kot teste. Za testiranje izjem uporabimo oznako `@Test(expectedExceptions=PricakovanaIzjema.class)`. Medtem ko v JUnit lahko naštejemo samo eno pričakovano izjemo, jih lahko pri TestNG naštejemo več. Test je uspešen v primeru, da enota vrne katero koli od naštetih izjem. Za ignoriranje testa uporabimo oznako `@Test(enabled=false)`.

Pri osnovni konfiguraciji ni bistvenih razlik med ogrodjema. Edina razlika je, da TestNG pri oznaki za pričakovane izjeme omogoča, da lahko podamo več izjem hkrati. Ostale razlike so le v poimenovanju oznak, ki so večinoma stvar osebnega okusa. Pri tem kriteriju je vseeno katero ogrodje izberemo, saj so razlike zanemarljive.

Izbira: JUnit ali TestNG

5.2.3 Zbirke testnih razredov

Testne razrede združujemo v večje množice, da jih lahko izvajamo kot celoto. Ogrodje mora omogočati fleksibilno organizacijo testnih razredov, prilagojeno potrebam razvijalcev.

JUnit

Pri JUnit testne razrede združujemo v testne zbirke. Testne zbirke lahko združujemo tudi v višje nivojske zbirke. JUnit nam omogoča, da definiramo testno hierarhijo do poljubne globine in širine. Slaba lastnost je, da moramo navesti vsak posamezni testni razred, ki ga vključuje testna zbirka. Pri velikem številu testnih razredov to lahko predstavlja problem.

TestNG

Prednost TestNG je fleksibilna definicija testne zbirke v XML datoteki. Navajamo lahko posamezne razrede ali celotne pakete. Omogoča nam tudi, da izvzamemo posamezne metode ali razrede.
Prednosti in slabosti:

- **JUnit**
  - ✓ fleksibilna organizacija testnih zbirk,
  - × testno zbirko moramo napisati kot razred.
- **TestNG**
  - ✓ definicija zbirke testnih razredov v XML datoteki,
  - ✓ ko spreminjamo zbirke, ni potrebno spreminjati programske kode,
  - × težje berljiv XML.

Pri organizaciji testnih razredov je boljši TestNG. Glavna razlika je, da v TestNG zbirke testnih razredov v celoti definiramo v XML datoteki, medtem ko jih moramo v JUnit implementirati kot razrede. To nam omogoča večjo fleksibilnost, saj je organizacija testnih razredov ločena od samih testnih razredov in ni implementirana z Java kodo.

Izbira: TestNG

5.2.4 Poročila o testiranju

Ob koncu vsakega testiranja ogrodje vrača informacije o poteku in rezultatih testiranja. Razvijalec po navadi dobi informacije o poteku testiranja v okviru razvojnega okolja. Včasih pa je potrebno, da se te informacije shranijo v obliki poročil. Poročila o testiranju služijo kot dokumentacija testiranja. Priročna so za druge razvijalce, saj s pregledom poročila hitro dobijo informacije o testiranju. Pomembna pa so tudi za samega razvijalca, ki izvaja testiranje. Tako ima vedno na voljo informacije o tem, kako daleč je potek testiranja, kateri deli kode še ne delujejo po pričakovanjih, itd. Brez poročil nimamo dobrega pregleda nad celovitostjo testiranja, ali je bilo izvedeno kakovostno in ali so bile testirane vse funkcionalnosti. Zlasti ob koncu testiranja je pomembno, da ustvarimo poročilo, ki služi za hitro ovrednotenje testiranja.
Od ogrodja se zahteva, da omogoča generiranje poročil na podlagi opravljenega testiranja v katerem izmed popularnejših formatov. Poročila morajo biti jasna, pregledna, vsebovati morajo osnovne informacije o izvedenih testih. V primeru, da kateri izmed testov najde napake, mora biti to ustrezno označeno s pripadajočim opisom.

JUnit

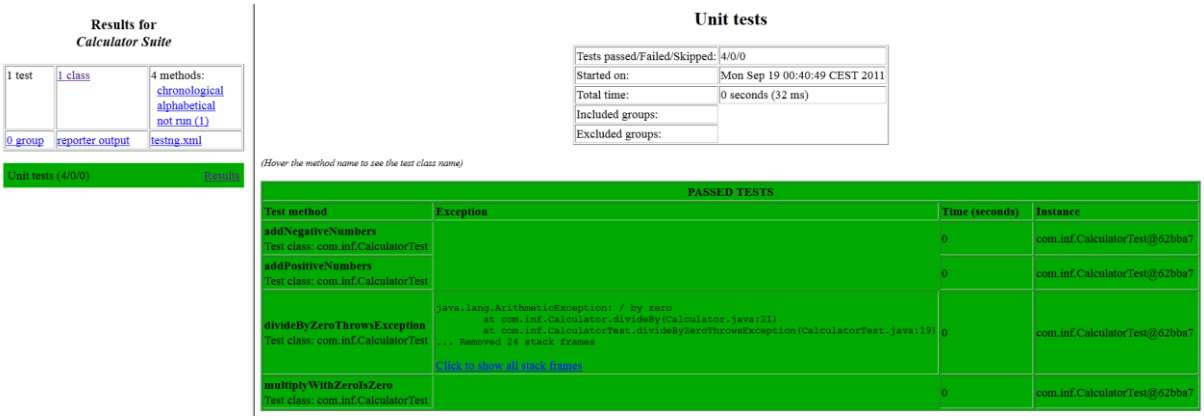
Generiranje poročil je sicer možno, vendar ga moramo sami implementirati tako, da razširimo vmesnik `TestRule`. Druga možnost je, da JUnit uporabljamo v kombinaciji z orodji za upravljanje projektov, kot sta npr. Ant in Maven. Privzeto generiranje poročil znotraj ogrodja ni možno.

TestNG

Omogoča avtomatsko generiranje poročila ob vsakem testnem zagonu. Poročila se ustvarjajo s pomočjo t.i. *poročevalcev* (*ang. reporters*). V ogrodju so že implemenitrani nekateri poročevalci:

- `EmailableReporter` – ustvari kompaktno poročilo, namenjeno pošiljanju po elektronski pošti.
- `FailedReporter` – ustvari datoteko `testng-failed.xml`. Vsebuje seznam testov, ki so našli napake.
- `JUnitReportReporter` – ustvari JUnit XML poročilo.

- `SuiteHTMLReporter` – ustvari standardno HTML poročilo za celotno zbirko.
- `TestHTMLReporter` – ustvari standardno HTML poročilo za skupino testov.
- `TextReporter` – ustvari preprosto tekstovno poročilo.
- `XMLReporter` – ustvari standardno XML poročilo.



Results for *Calculator Suite*

1 test	1 class	4 methods: chronological alphabetical not run (1)
0 group	reporter output	testng.xml

Unit tests (4/0/0) Result

Unit tests

Tests passed/Failed/Skipped:	4/0/0
Started on:	Mon Sep 19 00:40:49 CEST 2011
Total time:	0 seconds (32 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS

Test method	Exception	Time (seconds)	Instance
addNegativeNumbers Test class: com.inf.CalculatorTest		0	com.inf.CalculatorTest@62bba7
addPositiveNumbers Test class: com.inf.CalculatorTest		0	com.inf.CalculatorTest@62bba7
divideByZeroThrowsException Test class: com.inf.CalculatorTest	java.lang.ArithmeticException: / by zero at com.inf.Calculator.divideBy(Calculator.java:21) at com.inf.CalculatorTest.divideByZeroThrowsException(CalculatorTest.java:19) ... Removed 24 stack frames Click to show all stack frames	0	com.inf.CalculatorTest@62bba7
multiplyWithZeroIsZero Test class: com.inf.CalculatorTest		0	com.inf.CalculatorTest@62bba7

Slika 15: Primer standardnega HTML poročila v TestNG

Ob vsakem zagonu testne zbirke se ustvarita HTML (slika 15) in XML poročili, ki vsebujeta podatke o:

- testnih razredih in metodah,
- rezultatih testnih metod,
- času trajanja posameznih testnih metod in celotnega testiranja.

TestNG razvijalcem omogoča enostavno implementacijo poročevalcev po meri. Lastne poročevalce ustvarimo tako, da implementiramo vmesnik `IReporter`. Z njimi lahko generiramo poročila v drugih formatih, ali pa priredimo obstoječa poročila. Pripnemo jih ogrodju tako, da jih navedemo v datoteki `testng.xml`. Ogrodje jih potem uporabi med izvedbo testiranja.
Za morebitne neuspešne teste, se ustvari datoteka `testng-failed.xml`, v katero se zapišejo testi, ki so našli napake. Ta datoteka je v enakem formatu kot `testng.xml`, zato jo lahko enostavno ponovno izvedemo.

Prednosti in slabosti:

- **JUnit**
  - ✓ možnost implementacije lastnih poročil po meri,
  - ✗ ni privzetih poročil.
- **TestNG**
  - ✓ avtomatsko generiranje poročil v HTML in XML formatu,
  - ✓ implementacija lastnih poročil po meri,
  - ✓ XML datoteka z seznamom testov, ki so našli napake.

Privzeto generiranje poročil je možno samo v ogrodju TestNG. Ogrodje ponuja široko paleto poročil, poleg tega pa nam omogoča, da implementiramo poročevalce za lastna poročila po meri.

Izbira: TestNG

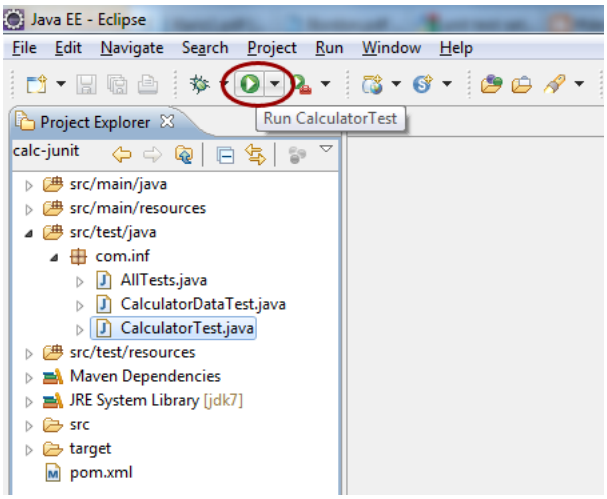
5.2.5 Integracija z razvojnimi okolji

Razvijalci uporabljajo različna orodja, s katerimi si pomagajo pri razvoju programske opreme. Priročno je, da so ta orodja zbrana v okviru razvojnega okolja (IDE). Tako je tudi pri testiranju. Za razvijalca je pomembno, da lahko testiranje enot enostavno izvaja znotraj razvojnega okolja. Pri tem kriteriju bomo primerjali integracijo obeh ogrodij z najbolj popularnim odprtokodnim razvojnim okoljem, in sicer z Eclipse. Obe ogrodji sta v razvojnem okolju Eclipse na voljo kot vtičnika in omogočata osnovne funkcionalnosti:

- pisanje testnih razredov,
- enostavno izvajanje testiranja in
- pregled rezultatov.

JUnit

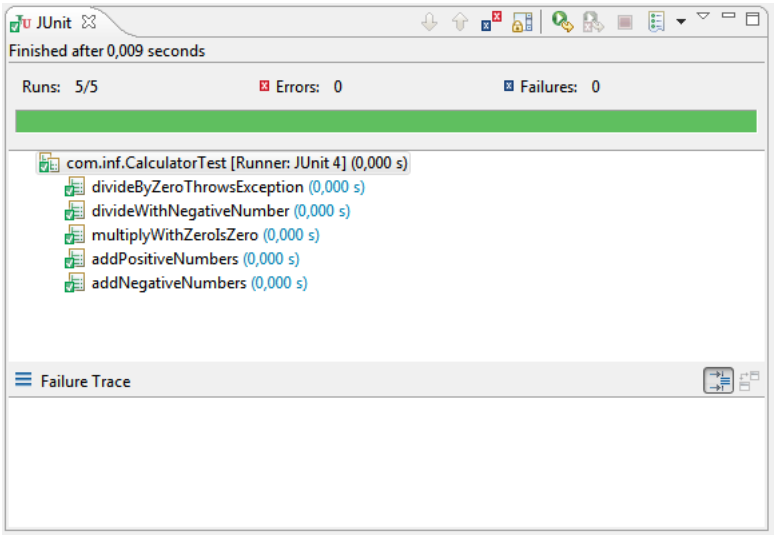
Vtičnik za JUnit je del standardne distribucije Eclipse. Omogoča enostavno kreiranje testnih razredov preko čarovnika, ki generira okostje testnega razreda.



Slika 16: Poganjanje JUnit testov v razvojnem okolju Eclipse

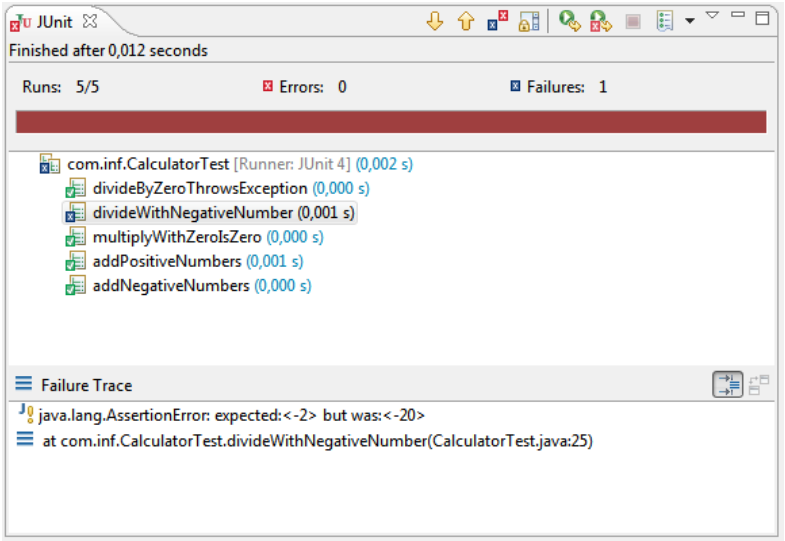
Teste enostavno poženemo prek orodne vrstice okolja, kjer lahko tudi nastavimo različne konfiguracije (slika 16). Poganjamo lahko testne razrede, posamezne testne metode ali testne zbirke.

Po izvedbi testiranja, dobimo grafični prikaz povzetka testiranja. Iz povzetka lahko razberemo, koliko testov je bilo izvedenih, koliko jih je našlo napake in kako dolgo so posamezni testi trajali (slika 17 in 18).



Slika 17: Rezultati testiranja z JUnit (primer 1)

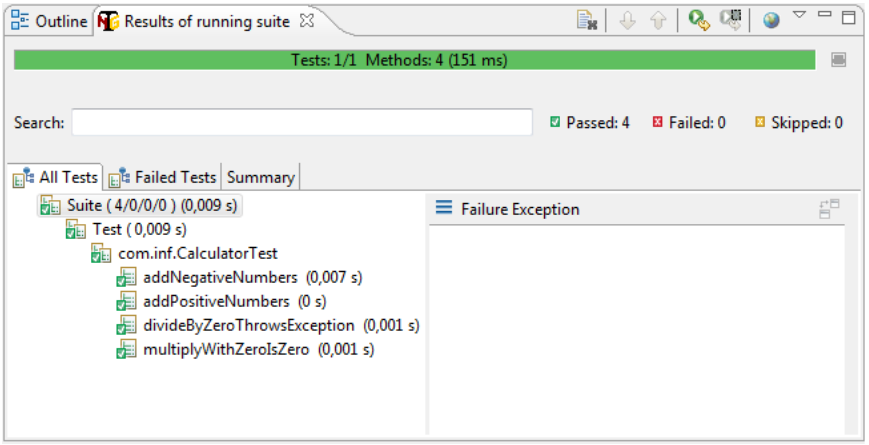
V primeru, da so bile najdene napake, dobimo opis o vzroku napake (slika 18). Testne metode, ki so našle napake, lahko zelo enostavno z nekaj kliki ponovno poženemo.



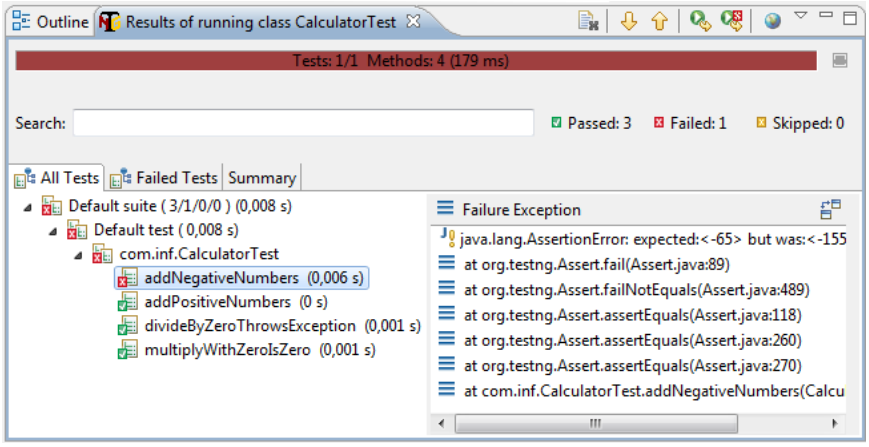
Slika 18: Rezultati testiranja z JUnit (primer 2)

TestNG

Je prav tako narejen kot vtičnik, razlika je le, da ga moramo ročno namestiti. Tudi tu je dobra podpora za kreiranje in izvajanje testov. Poženemo lahko testne razrede ali posamezne metode. Poleg tega pa lahko zaženemo tudi testne zbirke, ki so definirane v datoteki `testng.xml`, ali datoteko `testng-failed.xml`, ki vsebuje seznam testnih metod, ki so našli napake. Ob koncu vsakega testiranja dobimo podobno kot pri JUnit prikaz rezultatov testiranja, kjer vidimo uspešne, neuspešne teste in čas trajanja testov.



Slika 19: Rezultati testiranja z TestNG (primer 1)



Slika 20: Rezultati testiranja z TestNG (primer 2)

TestNG nam omogoča podoben prikaz rezultatov testiranja kot JUnit (slika 19 in 20). Vtičnik nam poleg naštetih funkcionalnosti omogoča tudi delno pretvorbo obstoječih JUnit testov v TestNG teste.

Prednosti in slabosti:

- **JUnit**
  - ✓ privzeta integracija z razvojnimi okolji.
- **TestNG**
  - ✗ potrebna dodatna namestitev.

Pri integraciji z razvojnim okoljem Eclipse ni večjih razlik. Obe ogrodji imata odlično podporo za osnovne funkcionalnosti. Manjša prednost JUnit je, da je že v osnovi vključen v Eclipse. Pri ostalih vodilnih odprtokodnih razvojnih okoljih, kot sta NetBeans in IntelliJ IDEA so funkcionalnosti podobne.

Izbira: JUnit ali TestNG

5.2.6 Popularnost ogrodja

Pomemben kriterij pri izbiri ogrodja je njegova popularnost. Popularnost pomeni, da je na voljo ustrezna literatura, kot so knjige, članki, vodiči, dokumentacija, iz katerih črpamo potrebne informacije. Pomembno je tudi, da obstaja dobro razvita spletna skupnost prek katere najdemo pomoč.

JUnit

Zaradi oznake de-facto standard na področju testiranja enot, obstaja veliko literature v katerih je testiranje enot predstavljeno skozi perspektivo JUnit. Zaradi tega večina razvijalcev povezuje pojem testiranja enot prav z ogrodjem JUnit. Zaradi svoje starosti in statusa ima ogromno spletno skupnost, tako da ob vprašanjih in težavah zelo hitro dobimo pomoč. Prav tako najdemo na internetu ogromno člankov in vodičev. Velik plus za ogrodje je tudi ogromno izdanih knjig. Ogrodje se uporablja tako pri enostavnih projektih, kakor tudi pri obširnih in kompleksnih projektih. Na popularnost JUnit zelo vpliva privzeta integracija z večino popularnih razvojnih okolij.
Slaba lastnost JUnit je, da za nekatere napredne funkcionalnosti ogrodja obstaja zelo malo gradiva. Kot primer lahko navedemo pravila (*ang. rules*). Na spletu razen dokumentacije ne najdemo skorajda nobenega gradiva o tem, kako se pravila uporabljajo.

TestNG

Od svojega nastanka pospešeno pridobiva na splošni popularnosti, vendar se glede tega ne more primerjati z JUnit, saj obstaja precej manj časa. V zadnjih letih je nastala velika spletna skupnost okoli ogrodja. Na internetu najdemo veliko člankov in vodičev na temo testiranja enot z TestNG. Ogrodje je zelo dobro dokumentirano. Obstaja tudi knjiga, ki jo je napisal avtor ogrodja, v kateri je zelo obširno razložena uporaba ogrodja.

Popularnost ogrodja si lahko pogledamo tudi v kontekstu spletnih iskalnikov. Na podlagi števila zadetkov na spletnih iskalnikih, lahko posredno sklepamo o popularnosti posameznega ogrodja na spletu. V tabeli 1 je prikazano število zadetkov v treh največjih spletnih iskalnikih za iskalna pojma `junit` ter `testng`.

	JUnit	TestNG
Google	8.470.000	810.000
Yahoo	552.000	60.200
Bing	518.000	58.700

Tabela 1: Približno število zadetkov v spletnih iskalnikih (na dan 27.8.2011)

JUnit je pri vseh večjih iskalnikih v zajetni prednosti glede števila omemb na spletu. Podatki do neke mere kažejo na popularnost posameznega ogrodja.

Prednosti in slabosti:

- **JUnit**
  - ✓ ogromna spletna skupnost,
  - ✓ ogromno knjig in člankov,
  - ✓ privzeto ogrodje za testiranje enot v razvojnih okoljih.
- **TestNG**
  - ✓ vztrajno pridobiva na popularnosti,
  - ✓ dobro dokumentiran.

Pri kriteriju popularnosti je nedvomno boljši JUnit. V veliki prednosti je tako po številu uporabnikov, kakor tudi po zastopanosti in omembah na spletu. Največja prednost je ogromno knjig na temo testiranja enot. V večini knjig je testiranje enot predstavljeno skozi perspektivo JUnit. Trenutno se TestNG glede popularnosti ne more primerjati z JUnit, vendar se razlika počasi zmanjšuje.

Izbira: JUnit

5.2.7 Podatkovno vodeno testiranje

Velikokrat se zahteva, da se določeno testno metodo enote preveri z večjim številom testnih podatkov. To nam omogoča podatkovno vodeno testiranje. Obe ogrodji omogočata podatkovno vodeno testiranje, vendar uporabljata različne pristope.

JUnit

JUnit ima zelo omejeno funkcionalnost podatkovno vodenega testiranja. Omejitve izhajajo iz same zasnove ogrodja. Prva omejitev je, da mora biti metoda, ki vrača zbirko podatkov, definirana statično. Druga omejitev izhaja iz zahteve, da testne metode ne smejo imeti parametrov. Zaradi tega se testni podatki lahko nastavljajo le prek konstruktorja testnega razreda. Velika pomanjkljivost ogrodja je tudi, da lahko imamo le eno zbirko testnih podatkov za celoten razred. Znotraj testnega razreda nikjer ne določimo, katere metode uporabljajo podatkovno vodeno testiranje. Zato JUnit vse testne metode izvede s podatkovno zbirko, tudi tiste, ki sploh ne uporabljajo podatkovno vodenega testiranja. Ta problem obidemo tako, da testne metode, ki ne uporabljajo podatkovno vodenega testiranja, prestavimo v drug testni razred.

TestNG

Podatke testnim metodam podajamo direktno prek parametrov, kar je bolj logičen pristop. Znotraj testnega razreda lahko definiramo poljubno število metod, ki vračajo testne podatke in jih poljubno povežemo s posameznimi testnimi metodami. S tem dodamo veliko mero fleksibilnosti, saj nas ogrodje v nobenem pogledu ne omejuje.

Prednosti in slabosti:

- **JUnit**
  - ✗ podajanje podatkov prek konstruktorja,
  - ✗ omejitev ene zbirke testnih podatkov,
  - ✗ podatkovno vodeno testiranje se uporablja za vse metode znotraj tesnega razreda,
  - ✗ potrebna delitev na več testnih razredov.
- **TestNG**
  - ✓ definiranje testnih podatkov v konfiguracijski datoteki,
  - ✓ podajanje podatkov prek parametrov testnih metod,
  - ✓ več zbirk testnih podatkov znotraj testnega razreda.

Glavna prednost TestNG je popolnoma fleksibilna definicija podatkovno vodenega testiranja. Druga prednost je podajanje testnih podatkov testnim metodam prek parametrov in ne prek konstruktorja. Glavna slabost JUnit, zaradi katere je funkcionalnost zelo omejena, je, da lahko definiramo samo eno podatkovno zbirko za vse testne metode znotraj testnega razreda. Pri podatkovno vodenem testiranju je torej boljši TestNG zaradi razširjene funkcionalnosti.

Izbira: TestNG

5.2.8 Prilagoditev testiranja

Pomemben kriterij, po katerem primerjamo ogrodji, je tudi možnost prilagoditve procesa testiranja. Ogrodje nam mora omogočati, da razširimo delovanje ogrodja in da spremenimo privzeto delovanje tako, da ga prilagajamo lastnim potrebam.

JUnit

JUnit nam omogoča spreminjanje funkcionalnosti z uporabo pravil (*ang. rules*). S pravili določimo, kako se izvajajo testne metode, testni razredi ali testne zbirke. Pravila razširjajo vmesnik `TestRule`. Če pravila uporabljamo na nivoju testnih razredov ali zbirk, jih moramo označiti z `@ClassRule`. Če pa jih uporabljamo na nivoju testnih metod, uporabimo oznako `@Rule`. Pravila delujejo kot prestrezniki klicev testnih metod (*ang. interceptors*). Z njimi lahko spremenimo izvedbo testne metode. Znotraj posameznega pravila lahko določimo, kaj se bo zgodilo pred in po izvedbi testne metode, razreda ali zbirke. JUnit ima že nekatere implementirana pravila. Najpomembnejša so:

- `ExternalResource`: inicializiramo zunanji vir, kot je na primer strežnik.
- `TemporaryFolder`: ustvarimo začasne datoteke in mape, ki se po končanem testu samodejno zbrišejo.
- `Timeout`: določimo čas trajanja posamezne testne metode.

S pravili lahko sami implementiramo poročila, lahko naredimo dodatna preverjanja za testne metode, na podlagi katerih se lahko odločimo, če test vrne napako ali ne. Na žalost so pravila zelo slabo dokumentirana in obstaja zelo malo literature na to temo.

TestNG

TestNG ponuja različne vmesnike, prek katerih lahko spreminjamo obnašanje ogrodja. Ti vmesniki se imenujejo poslušalci (*ang. listeners*). Poslušalci se izvedejo na različnih mestih znotraj življenjskega cikla testnega razreda. Najpomembnejši so:

`IHookable` – se izvede tik pred izvedbo testne metode. Z njim lahko spremenimo ali preskočimo izvedbo testne metode.

`IMethodInterceptor` – se izvede, tik preden se začnejo izvajati testne metode. Lahko spreminjamo vrstni red izvajanja testnih metod.

`IReporter` – vmesnik, ki omogoča, da implementiramo generiranje lastnih poročil. Izvede se ob koncu testiranja.

`ISuiteListener` – izvede se pred začetkom in po koncu izvedbe testne zbirke.

`ITestListener` – izvede se pred začetkom in po koncu izvedbe testa.

V datoteki `testng.xml` pod značko `<listener>` navedemo poslušalce, ki jih želimo uporabiti v testni zbirki.

Prednosti in slabosti:

- **JUnit**
  - ✓ privzeta pravila,
  - × slabo dokumentirana funkcionalnost.
- **TestNG**
  - ✓ več vmesnikov poslušalcev,
  - ✓ navajanje poslušalcev v XML datoteki.

TestNG omogoča široko prilagoditev testiranja prek poslušalcev, saj nam omogoča, da implementiramo poslušalce na nivoju testne metode, razreda, testa, zbirke. V JUnit je funkcionalnost pravil zelo slabo dokumentirana in zelo zapletena za uporabo.

Izbira: TestNG

5.2.9 Razvoj ogrodja

Razvoj programske opreme postaja vse bolj kompleksen. Posledično to velja tudi za testiranje, saj se zahteva, da pokriva kompleksne programerske koncepte. Zato je pomembno, da ogrodja sledijo tem zahtevam. Pomembno je, da se uvajajo nove funkcionalnosti,

inovativni koncepti, ki poenostavijo celoten proces testiranja. Prav tako je pomembno, da se odpravljajo napake znotraj ogrodja in izboljšujejo obstoječe funkcionalnosti.

JUnit

JUnit je na začetku pomembno vplival na področje testiranja enot. Vendar je razvoj v zadnjih letih rahlo zastal. Večinoma se dela le na stabilizaciji ogrodja z dodajanjem nekaterih eksperimentalnih funkcionalnosti. Sprememb znotraj ogrodja skoraj ni. Največje spremembe so se zgodile z uvedbo JUnit verzije 4, ko so korenito spremenili funkcionalnost z uvedbo oznak in podatkovno vodenega testiranja. Oba koncepta so presneli z ogrodja TestNG. Izidi novih verzij so zelo redki. Med izidi verzij 4.8.2 in 4.9 je na primer preteklo skoraj 15 mesecev. Nove verzije po navadi ne prinašajo nobenih novih funkcionalnosti, ampak samo popravke. Dobra lastnost JUnit je kompatibilnost s prejšnjimi verzijami.

TestNG

Uvedba ogrodja TestNG je prinesla veliko inovacij. Ogrodje je nastalo prav z namenom, da omogoči funkcionalnosti, ki so se skozi leta pokazale kot nujne, vendar jih JUnit ni omogočal. Prav tako je uvedel določene poenostavitve. Najbolj pomembna inovacija, ki jo je uvedel TestNG, je bila konfiguracija testov z oznakami. Ta inovacija je zelo poenostavila pisanje testnih razredov in testnih metod. Druga pomembna inovacija je bila podpora podatkovno vodenemu testiranju. Razvoj ogrodja je zelo aktiven. Nove verzije izhajajo skoraj vsak mesec. Vsaka nova verzija prinaša nekatere popravke in majhne izboljšave.

Prednosti in slabosti:

- **JUnit**
  - ✓ kompatibilnost s starejšimi verzijami,
  - × rahlo zastal razvoj,
  - × počasno odpravljanje napak znotraj ogrodja.
- **TestNG**
  - ✓ redne posodobitve,
  - ✓ redno odpravljanje napak znotraj ogrodja.

Zaradi aktivnega razvoja, je TestNG boljša izbira.

Izbira: TestNG

5.3 Rezultati primerjave

S primerjavo smo ugotovili katero ogrodje je boljše z vidika uporabnosti. V obeh ogrodjih je možna kakovostna in enostavna izvedba testiranja enot. Pri večini kriterijev so razlike majhne, vendar do njih vseeno prihaja. Te razlike so vplivale na odločitev o izbiri ogrodja.

Kriterij	Izbira
1. Življenjski cikel testnega razreda	JUnit
2. Konfiguracija testnih razredov	JUnit ali TestNG
3. Zbirke testnih razredov	TestNG
4. Poročila o testiranju	TestNG
5. Integracija z razvojnimi okolji	JUnit ali TestNG
6. Popularnost ogrodja	JUnit
7. Podatkovno vodeno testiranje	TestNG
8. Prilagoditev testiranja	JUnit ali TestNG
9. Razvoj ogrodja	TestNG

Tabela 2: Povzetek rezultatov

JUnit je uveljavljeno ogrodje, ki mu zaupa ogromno razvijalcev po vsem svetu. Ima bogato zgodovino, saj velja za standard na področju testiranja enot že skoraj 15 let. Poleg svoje popularnosti je glavna prednost v primerjavi s TestNG implementacija življenjskega cikla, ki omogoča popolno izolacijo testnih metod. JUnit ima svoje prednosti, vendar v nekaterih primerih tudi omejitve, ki jih moramo obiti na različne načine. To se pokaže predvsem pri podatkovno vodenem testiranju in poročilih o testiranju.

Rezultati primerjave (tabela 2) prikazujejo, da je TestNG boljši po štirih kriterijih, po dveh kriterijih je boljši JUnit, pri ostalih treh kriterijih sta ogrodji zelo izenačeni. Glede na izbrane kriterije se torej bolje izkaže TestNG, ki v splošnem ustvari boljšo uporabniško izkušnjo. Izrazita prednost je pri funkcionalnosti generiranja poročil o testiranju. Razen te prednosti, nam TestNG ne ponuja nobene pomembne funkcionalnosti, ki ne bi bila tako ali drugače implementirana tudi v ogrodju JUnit. Ponuja pa nekatere prednosti, ki so z uporabnostnega vidika za razvijalca zelo priročne, saj poenostavijo testiranje. Najpomembnejši prednosti sta:

- konfiguracija testnih zbirk v XML datoteki in
- razširjeno podatkovno vodeno testiranje.

Čeprav so pri JUnit osnovni koncepti enostavni, obstaja ogromno literature in je ogrodje preverjeno na ogromnem številu projektov, je po naših ugotovitvah TestNG trenutno boljša izbira, saj omogoča enostavnejše testiranje in boljšo uporabniško izkušnjo. Zaradi tega smo se odločili za ogrodje TestNG.

6 Sklepne ugotovitve

Izdelava diplomskega dela je predstavljala svojevrsten izziv. Izkazalo se je, da je testiranje programske opreme zelo obširno področje. Najprej smo morali pridobiti temeljna znanja o procesu testiranja in celovit pogled na testiranje programske opreme. Prišli smo tudi do nekaterih novih spoznanj. Spoznali smo, kako velik pomen pri razvoju programske opreme ima testiranje enot oziroma testiranje nasploh. Prav tako smo pridobili nova znanja s področja testiranja enot in delovanja ogrodij za testiranje enot.

S primerjavo smo prišli do spoznanja, da je izbira pravega ogrodja pomembna, čeprav na nivoju testiranja enot ni velikih razlik med obema ogrodjema. Ugotovili smo, da sta obe ogrodji z vidika uporabnosti zelo dobri, vendar je TestNG trenutno boljši. Glavna prednost TestNG je poenostavitev in razširitev nekaterih funkcionalnosti.

Primerjava, ki smo jo izvedli pa ni celovita primerjava ogrodij za testiranje enot. Glede na to, da se ogrodja za testiranje enot uporabljajo tudi za druge vrste testiranj (npr. za integracijsko testiranje), bi lahko primerjavo razširili tudi s kriteriji, ki so pomembni za druge vrste testiranj. Prav tako bi lahko v celovito primerjavo vključili kriterij integracije z različnimi orodji, ki jih razvijalci uporabljajo skupaj z ogrodji za testiranje enot.

Dodatek A

Seznam slik

Seznam tabel

Literatura

[1] C. Beust, H. Suleiman, *Next Generation Java Testing: TestNG and Advanced Concepts*, Boston: Addison-Wesley, 2007.

[2] (2011) Domača stran JUnit. Dostopno na: http://www.junit.org

[3] (2011) Domača stran TestNG. Dostopno na: http://testng.org

[4] J. Tian, *Software quality engineering*, New Jersey: John Wiley & Sons, 2005.

[5] (2011) Javanske oznake. Dostopno na: http://en.wikipedia.org/wiki/Java_annotation

[6] (2011) JUnit zgodovina. Dostopno na: http://www.martinfowler.com/bliki/Xunit.html

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Upper Saddle River: Addison-Wesley, 2000.

[8] (2011) Martin Fowler: Mocks Aren't Stubs. Dostopno na: http://martinfowler.com/articles/mocksArentStubs.html

[9] (2011) Ogrodje. Dostopno na: http://docforge.com/wiki/Framework

[10] P. Amman, J. Offcutt, *Introduction to software testing*, New York: Cambridge University Press, 2008.

[11] P. Tahchiev, F. Leme, V. Massol, G. Gregory, *JUnit in Action, Second Edition*, Stamford: Manning Publications, 2010.

[12] R. Osherove, *The art of unit testing*, Greenwich: Manning Publications, 2009.

[13] (2011) xUnit. Dostopno na: http://en.wikipedia.org/wiki/XUnit