Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

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

Št. naloge: 01643/2010
Datum: 15.03.2010

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

Kandidat: **ANDREJ ROVAN**

Naslov: **NUNIT: ORODJE ZA TESTIRANJE ENOT V OKOLJU .NET**
NUNIT: A UNIT-TESTING TOOL FOR MICROSOFT .NET

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Proučite značilnosti agilnih metod za razvoj programske opreme s posebnim poudarkom na ekstremnem programiranju in testno vodenem razvoju. Opišite koncept testiranja enot in možnosti, ki jih za razvoj v okolju .NET nudi orodje NUnit. Uporabo tega orodja prikažite na primeru razvoja preproste spletne aplikacije.

Mentor:
prof. dr. Viljan Mahnič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Franc Solina

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Andrej Rovan

NUnit: Orodje za testiranje enot v okolju .NET

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2010

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Andrej Rovan,

z vpisno številko 63040147,

sem avtor/-ica diplomskega dela z naslovom:

NUnit: Orodje za testiranje enot v okolju .NET

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom
  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, dne 15.06.2010 Podpis avtorja/-ice:

Zahvala

Zahvalil bi se mojemu mentorju za uporabne nasvete in popravke pri diplomskem delu, mojim staršem, ki so mi stali ob strani tekom študija in vsem, ki so mi kakorkoli pomagali, da sem uspešno končal študij.

Kazalo

Povzetek

Testno voden razvoj programske opreme je eden izmed načinov razvoja programske opreme. Uvrščamo ga med prakse ekstremnega programiranja, temelji pa na ponavljanju kratkih razvojnih ciklov, kjer je prvi korak vedno pisanje testa in šele nato implementacija funkcionalnosti. Pri takem razvoju imamo prisotno preoblikovanje, ki je ena izmed značilnosti ekstremnega programiranja.

Diplomsko delo na začetku opisuje agilen pristop k razvoju programske opreme, ekstremno programiranje in testno voden razvoj. Sledi bolj podroben opis orodja NUnit, ki ga uporabljamo pri avtomatiziranih testih. Skozi praktičen primer na koncu bomo poskušali uveljaviti prakse, ki smo jih v drugem poglavju opisali.

Ključne besede:

testno voden razvoj, ekstremno programiranje, testiranje, agilen pristop, NUnit, ASP.NET

Abstract

Test driven development is a software development technique that relies on a repetition of very short development cycles: first the developer writes an automated test case, then implements functionality. Developers also use technique called refactoring, which is one of the features of extreme programming.

At the beginning, thesis describes agile software development, extreme programming and test driven development, followed by a more detailed description of NUnit, which is used for automated tests. Through a practical example, in the end we will try to implement practices that were described in second chapter.

Key words:

test driven development, extreme programming, testing, agile software development, NUnit, ASP.NET

Poglavje 1

Uvod

Razvoj programske opreme se skozi čas spreminja in prilagaja. Pojavljajo se nove tehnologije, ki prinašajo boljše načine za razvoj programske opreme. Eden izmed glavnih problemov so naročnikove zahteve, ki se skozi čas spreminjajo, kar predstavlja razvijalcem veliko težavo. V ta namen so se pojavile različne nove metodologije razvoja programske opreme.

Na tem mestu bomo omenili ekstremno programiranje ter testno voden razvoj, ki je ena izmed praks ekstremnega programiranja. Ekstremno programiranje spada med agilne metode, ki temeljijo na prilagodljivosti, poudarku na ljudeh in ne na tehnologiji, sodelovanju s stranko ter hitrim odzivom na spremembe.

Lastnosti ekstremnega programiranja pa so še programiranje v parih, preprosta in jasna programska koda, sprotno preoblikovanje, igra planiranja itn.

Cilj diplomskega dela je preučiti orodje NUnit in po principih testno vodenega razvoja implementirati krajšo aplikacijo ter podati zaključke. Na osnovi praktičnega primera bomo poskušali opisati prednosti in slabosti, predvsem pa ilustrirati testno voden razvoj in njegov cikel.

Poglavje 2

Testno voden razvoj programske opreme

2.1 Agilen pristop k razvoju programske opreme

2.1.1 Uvod

Agilen razvoj programske opreme je nov pristop k razvoju programske opreme [1], njena osnova pa je iterativni razvoj, kjer se zahteve in rešitve razvijajo s sodelovanjem med samo-organiziranimi ekipami.

Izraz je bil oblikovan leta 2001, ko je bil objavljen *manifest agilnosti* [8].

Agilne metode v splošnem predstavljajo disciplinirano vodenje projektov, kjer je v ospredju pogost pregled in prilagajanje programske kode. Filozofija vodstva temelji na opogumljanju skupine, da je odgovorna in z množico najboljših praks hitro in kvalitetno dostavi programsko opremo. Poslovni pristop, ki vodi razvoj programske opreme, je uravnan s potrebami stranke in cilji podjetja.

2.1.2 Zgodovina

Moderna definicija agilnega razvoja programske opreme se je razvila v srednjih-1990 kot del reakcije proti "težkim" metodam. Slednje se strogo držijo "waterfall" modela razvoja (veliko dokumentacije, zamrznitev strankinih zahtev...). Slednji način razvoja se je mnogim zdel prepočasen in nekonsistenten v primerjavi z učinkovitim delom razvijalcev. Lahko bi rekli, da se agilne metode vračajo k začetkom razvoja programske opreme. Imenujemo jih tudi "lahke" metode.

V letu 2001 so se zagovorniki agilnih metod zbrali v smučarskem letovišču v mestu Utah z namenom, da bi ustvarili lažje, hitrejše načine ustvarjanja programske opreme, ki so bolj osredotočeni na ljudeh in ne na tehnologiji. Osnovali so izraza *agilen razvoj programske opreme* in *agilne metode*. Posledica je tudi *manifest agilnosti*, ki je bil izdan leta 2001.

2.1.3 Manifest agilnosti

Agilne metode so družina razvojnih procesov programske opreme, ne samo enega pristopa k razvoju programske opreme. Manifest agilnosti navaja:

> Odkrivamo boljše načine razvoja programske opreme - z našim udejstvovanjem in s pomočjo drugim. Z našim delom smo prišli do naslednjih točk:
>
> - **Posamezniki in interakcije** so pomembnejši od procesov in orodij
> - **Delujoča programska oprema** je pomembnejša od obsežne dokumentacije
> - **Sodelovanje s stranko** je pomembnejše od pogajanj o pogodbi
> - **Odzivanje na spremembe** je pomembnejše od sledenja načrtu
>
> Vrednote na levi strani so za nas bolj cenjene od tistih na desni.

Nekateri najbolj pomembni principi, ki jih manifest agilnosti vključuje:

- zadovoljstvo stranke s hitro in neprekinjeno dostavo kvalitetne programske opreme,
- delujoča programska oprema je dostavljena pogosto (raje tedni, kot meseci),
- delujoča programska oprema je glavno merilo napredka,
- tudi pozne spremembe v zahtevah so dobrodošle,
- dnevno sodelovanje med programerji in vodstvenim kadrom,
- pogovor v živo je najboljša oblika komunikacije,
- samo-organizirane ekipe.

2.1.4 Posamezniki in interakcije so pomembnejše od procesov in orodij

Ljudje so najpomembnejša sestavina uspeha. Dober proces ne bo rešil propadlega projekta, če ekipa nima dobrih igralcev. V primeru slabega procesa pa lahko tudi najboljši igralci postanejo neučinkoviti. Tudi skupini najboljših igralcev lahko spodleti, če ne delujejo kot celotna ekipa.

Dober igralec ni nujno najboljši programer. Lahko je čisto povprečen, vendar dobro dela in sodeluje z ostalimi. To je dosti bolj pomembno od čistega programerskega talenta. Skupina povprečnih programerjev, ki dobro komunicirajo, ima večje možnosti za uspeh, kot skupina zvezd, ki ne delujejo skupaj kot ekipa.

Prava orodja so lahko zelo pomembna za uspešnost. Prevajalniki, interaktivna razvojna orodja (IDE), kontrolni sistem razvoja kode itn. pripomorejo k pravilnemu delovanju razvojne ekipe. Pogosto pa jih precenimo. Prevelika uporaba nepriročnih orodij je lahko prav tako škodljiva kot premajhna.

Včasih je bolje začeti z zastonjskimi orodji, se poglobiti v njih in postopoma nadgraditi razvojni cikel z boljšimi. Zmotno je razmišljanje, da bodo boljša orodja samodejno naredila boljše delo.

Gradnja ekipe je veliko bolj pomembna od gradnje razvojnega okolja. Veliko programerskih hiš naredi to napako, da začne najprej z gradnjo razvojnega okolja in pričakuje, da bodo dobri programerji prišli kar sami od sebe. Zato je potrebno delo na oblikovanju ekipe in na osnovi ljudi zgraditi razvojno okolje.

2.1.5 Delujoča programska oprema je pomembnejša od obsežne dokumentacije

Programska oprema brez dokumentacije je zelo slaba. Programska koda ni idealen medij za utemeljitev in strukturiranje sistema. Ekipa namesto tega potrebuje berljive dokumente, ki opisujejo sistem in utemeljitve za arhitekturne odločitve.

Vseeno pa velja, da je preveč dokumentacije slabše kot premalo. Pisanje obsežne dokumentacije nam vzame veliko časa, še več časa pa porabimo za sinhronizacijo med programsko kodo in dokumentacijo. Če dokumentacija ni sinhronizirana s programsko kodo, se spremeni v ogromno zbirko nerazumljivih strani, ki je vzrok za mnoge nesporazume.

Za ekipe je vedno dobra ideja, da piše in vzdržuje kratek in jedrnat dokument, ki vsebuje zgolj bistveno arhitekturno zasnovno in principe. Kratko v tem primeru pomeni manj od 30 strani. Znanje novim članom ekipe se prenaša

tako, da izkušen programer sedi pri njih in jim pomaga.

Najboljša dokumenta, ki jih lahko predamo novim članom projekta sta programska koda in ekipa. Koda ne laže o tem, kaj dejansko počne. Včasih je težko razbrati njeno delovanje, vendar je to edini nedvoumen vir informacij. Najhitrejši prenos znanja pa se zagotovo prenaša preko interakcije človek-človek.

Obstaja preprosto pravilo, da naš razvoj ne preide v brezumno pisanje dokumentacije (*Martin's First Law of Documentation*) [5]:

> Ne napiši dokumenta, razen če je potreba po njem takojšnja in pomembna.

2.1.6 Sodelovanje s stranko je pomembnejše od pogajanj o pogodbi

Programske opreme ni moč naročiti kot navadno blago. Stranka ne more podati samo opisa sistema, ki ga želi, in nato zahtevati od nekoga, da ga razvije v točno določenem času za točno določeno ceno. Mnogi projekti, ki so bili osnovani na slednji način, so propadli [5].

Uspešni projekti vključujejo reden in pogost odziv strank. Raje, kot da smo odvisni samo od pogodbe, naj stranka tesno sodeluje z razvojno ekipo in tako zagotavlja povraten odziv na njena prizadevanja.

Pogodba, ki določa zahteve, načrt in ceno projekta je že v osnovi pomankljiva. V večini primerov pogodbeni pogoji postanejo brezpredmetni, še preden je projekt končan, včasih celo veliko pred podpisom pogodbe. Najboljše pogodbe vsebujejo tudi definicijo, kako bo potekalo sodelovanje med stranko in razvojno ekipo.

2.1.7 Odzivanje na spremembe je pomembnejše od sledenja načrtu

Sposobnost odzivanja na spremembe pogosto določa uspeh ali neuspeh projekta. Ko načrtujemo projekt, se moramo zavedati, da so naši načrti prilagodljivi in pripravljeni za poslovne in tehnološke spremembe.

Poteka projekta ni mogoče načrtovati zelo daleč v prihodnost. Zelo velika verjetnost je, da se bo poslovno okolje tekom projekta spreminjalo. Druga stvar je spreminjanje zahtev naše stranke, ko sistem začne delovati. Na koncu je dejstvo tudi to, da je časovna ocena znanih zahtev zelo težavno opravilo, ki ga je težko točno napovedati.

Boljša načrtovalna strategija je izdelava natančnih načrtov za teden naprej in bolj grobih za tri mesece naprej. Programerji bi morali vedeti, kakšne naloge jih čakajo tekom naslednjega tedna in približno kakšne naloge jim bodo dodeljenje v naslednjih treh mesecih.

Takšna rešitev načrtovanja pomeni, da se v podrobnem načrtu upoštevajo samo tiste naloge, ki so takojšnje in najbolj pomembne. Podroben načrt, ko je enkrat narejen, je težko spreminjati. Zaradi podrobnega načrta, ki ne sega predolgo v prihodnost, je lahko razvoj dosti bolj prilagodljiv.

2.1.8 Metode agilnega razvoja progamske opreme

Nekatere od dobro poznanih agilnih metod so:

- **ekstremno programiranje**,
- agilno modeliranje,
- Agile Unified Process (AUP),
- DSDM,
- Essential Unified Process (EssUP),
- Feature Driven Development (FDD),
- Open Unified Process (OpenUP),
- Scrum.

2.1.9 Prakse agilnega razvoja programske opreme

Nekatere prakse agilnega razvoja programske opreme so:

- **testno voden razvoj**,
- preoblikovanje programske kode,
- stalna integracija,
- programiranje v parih,
- metoda RITE.

2.2 Ekstremno programiranje

Ekstremno programiranje [2] je ena izmed metod agilnega razvoja programske opreme. Njen namen je izboljšanje kvalitete programske opreme in odzivnost na strankine zahteve. Kot predstavnica agilnih metod poudarja pogoste izdaje novih verzij programske opreme v kratkih razvojnih ciklih.

Tipične prakse ekstremnega programiranja so: programiranje v parih, **testi enot, ki pokrivajo celotno kodo**, izogibanje programiranju funkcionalnosti, dokler jih zares ne potrebujemo, enostavost in jasnost programske kode, prilagodljivost na strankine zahteve, pogosta komunikacija med programerji in stranko, uporabniške zgodbe, kratki cikli, skupno lastništvo nad kodo, stalna integracija, 8-urni delovni čas, načrtovalna igra, preoblikovanje.

Spodaj so podani bolj podrobni opisi.

2.2.1 Uporabniške zgodbe

Za načrtovanje projekta potrebujemo informacije o zahtevah, pa vendar nam ni treba vedeti vsega. Za načrtovalne namene je dovolj, da grobo poznamo zahtevo in tako podamo oceno. V splošnem bi mislili, da moramo za oceno časa poznati vse podrobnosti, vendar po metodologiji ektremnega programiranja to ni potrebno.

Podrobnosti zahteve se bodo s časom verjetno spreminjale, še posebej ko stranka začne sistem prvič uporabljati ali testirati. V slednjem primeru se tudi zahteve bolje definirajo, tako da prezgodnje pridobivanje in specifikacija podrobnosti ne pride do izraza.

Po metodologiji ekstremnega programiranja se podrobnosti definirajo preko pogovorov s stranko, ki se ne zapišejo v nobeno obsežno dokumentacijo ipd. Namesto tega stranka napiše nekaj besed na list, ki so zgolj povzetek celotnega pogovora. Razvijalci ocenijo zahtevo skoraj v istem času, ko stranka piše povzetek in jo zapišejo na list.

Uporabniška zgodba je sopomenka za zabeležko o neki zahtevi s strani stranke. Je načrtovalno orodje, ki ga stranka uporablja za vrstni red razvoja zahtev, ki temelji na prioritetah in približni ceni.

2.2.2 Kratki cikli

Ekstremno programiranje predvideva zagotavljanje nove delujoče verzije programske opreme vsaka dva tedna. Vsaka od teh dvo-tedenskih iteracij prinaša

nove funkcionalnosti. Na koncu vsake iteracije se sistem demonstrira ter tako pridobi odziv stranke in odgovornih ljudi na projektu.

Načrtovanje iteracije

Iteracija je ponavadi dolga dva tedna in predstavlja manjše število novih funkcionalnosti, ki se jih lahko ali pa ne sme dodati v produkcijsko okolje. Načrt iteracije je zbirka uporabniških zgodb, izbranih s strani stranke glede na količino sredstev, namenjenim razvijalcem.

Razvijalci določijo vire za vsako iteracijo z merjenjem, koliko so naredili v prejšnji iteraciji. Uporabnik lahko izbere poljubno število uporabniških zgodb, pogoj je le, da skupna cena ne preseže dodeljenih virov.

Pred začetkom iteracije se vodje na obeh straneh strinjajo o zamrznitvi definicij in prioritet uporabniških zgodb, ki pripadajo trenutni iteraciji. Med iteracijo lahko razvijalci uporabniške zgodbe razdelijo na naloge in začnejo z njihovo implementacijo.

Načrt izdaje nove verzije

Ekipe pogosto načrtujejo izdajo nove verzije programske opreme približno vsakih šest iteracij. Izdaja nove verzije obsega tri mesece dela celotne ekipe. Predstavlja večji doprinos k končnemu produktu, ki se ga ponavadi doda v produkcijsko verzijo. Načrt vsebuje zbirko uporabniških zgodb, ki so izbrane glede na dodeljene vire programerjev.

Vsebina nove verzije ni dokončno določena, lahko se spremeni poljubno. Prekličemo lahko uporabniške zgodbe, napišemo nove, ali pa spremenimo prioritete. Pomembeno je le, da se spremembe ne odvijajo med iteracijo.

2.2.3 Uporabniški testi

Podrobnosti o uporabniških zgodbah so zajete v obliki uporabniških testov, ki jih napiše stranka oziroma uporabnik. Le ti so napisani že pred, ali hkrati z implementacijo uporabniške zgodbe. Njihov jezik je skriptni in jim omogoča avtomatizirano izvajanje, s katerim stranka preveri, ali se sistem obnaša, kot je predvideno.

Uporabniške teste pišejo poslovni analitiki, specialisti za zagotavljanje kvalitete in testerji med iteracijo. Njihov jezik je lahko razumljiv programerjem, strankam in poslovnim vodjem. Iz teh testov so programerjem razvidne podrobnosti uporabniških zgodb, ki jih implementirajo. V njih je opisana vsaka

funkcionalnost in imajo zadnjo besedo pri tem, ali je funkcionalnost končana in pravilna.

Ko uporabniški test uspešno prestane testiranje, je dodan v zbirko uspešno prestalih uporabniških testov, ki se morajo ob vsakem testiranju uspešno izvesti. Zbirka testov se požene večkrat na dan, vsakič ko se zgradi nova verzija sistema. Če eden od testov ne uspe, se novi sistem označi kot neuspešno zgrajen.

2.2.4 Programiranje v parih

Par programerjev, ki delata za istim računalnikom, piše programsko kodo. Eden od njiju uporablja tipkovnico in miško, drugi pa opazuje vnešeno programsko kodo, išče napake in izboljšave. Oba sta v celoti vključena v proces pisanja programske kode.

Vloge programerjev v paru, se spreminjajo pogosto. Če je nekdo utrujen, se za tipkovnico vsede drugi programer in nadaljuje z delom. S tem se produktivnost poveča. Poleg tega sta za končni rezultat enako zaslužna oba.

Razumljiv cilj je menjava partnerja vsaj enkrat na dan, tako da lahko vsak programer dela v dveh različnih parih vsak dan. Tekom iteracije bi moral vsak programer delati z vsakim članom ekipe, skupaj pa bi morala delati na vseh zahtevah, ki so bile predvidene v tej iteraciji.

Programiranje v paru dramatično povečuje širjenje znanja v celotni ekipi. Čeprav so posebne naloge dodeljene ustreznim specialistom, bodo ti specialisti delali v paru s skoraj vsemi člani ekipe.

2.2.5 Testno voden razvoj

Vsa produkcijska koda je napisana z namenom, da neuspeli test uspešno prestane testiranje. Prvi korak je vedno pisanje testa enote, ki se ne izvede uspešno, saj testirana funkcionalnost še ne obstaja. Zatem se napiše programska koda z namenom, da se test enote uspešno izvede.

Preklapljanje med pisanjem testnih primerov in programske kode je zelo hitra. Testi in programska koda so med seboj povezani in se postopoma razvijajo.

Število testov raste skupaj s številom vrstic programske kode. Testi enot omogočajo programerjem, da lahko v vsakem trenutku preverijo delovanje programa. Ko nekaj skodiramo, lahko to takoj preverimo s testi in se prepričamo, da nismo pokvarili kakšne funkcionalnosti.

Ko kodiramo neko funkcionalnost, da bi se test uspešno izvedel, je naša programska koda po definiciji zmožna testiranja. Obstaja tudi motivacija, da so posamezni moduli ločeni in jih je možno testirati neodvisno. Pri tem nam močno pomagajo principi objektno usmerjenega programiranja.

2.2.6 Skupno lastništvo nad programsko kodo

Vsak razvijalec ima pravico do pregleda in izboljšave poljubnega modula. Nobeden programer ni odgovoren za konkreten modul ali tehnologijo kot posameznik. Vsi delajo na grafičnemu vmesniku, poslovni plasti ali kateremu drugemu modulu. Razvijalci so med seboj enakovredni.

To ne pomeni, da ekstremno programiranje zanika posebna znanja. Če je nekdo specializiran za grafični vmesnik, bo po vsej verjetnosti delal na takih nalogah, vendar bo vključen tudi v druge naloge (podatkovna baza, poslovna plast...). Če bi se nekdo rad naučil druga znanja, se lahko prijavi za delo v paru in dela s specialistom. To omogoča hitro širitev znanja po celi ekipi.

2.2.7 Stalna integracija

Razvijalci preverijo programsko kodo in jo integrirajo večkrat na dan. Pravilo je preprosto. Prvi, ki doda kodo v sistem za kontrolo programske kode, je svoje delo opravil. Vsi ostali morajo kodo združevati.

Ekipe, ki delujejo po principu ekstremnega programiranja, uporabljajo neblokirajoč sistem za kontrolo programske kode. To pomeni, da lahko kadarkoli pišejo kodo v katerem koli modulu, neodvisno od tega, če že nekdo drug razvija ta modul. Ko programer doda novo verzijo modula v sistem za kontrolo programske kode, mora biti pripravljen na združevanje kode z ostalimi spremembami drugih razvijalcev. Temu se v veliki meri izognemo, da spremembe dodajamo pogosto v sistem za kontrolo programske kode.

Programerski par bo za posamezno nalogo porabil od 1-2 uri. Pisanju testov sledi kodiranje produkcijske kode. Na neki točki, ko naloga še ni končana, programerja dodata spremebe v sistem. Najprej se prepričata, da se testi enot uspešno izvedejo in nato naredita integracijo nove kode v obstoječo. Ko je integracija končana, se sistem ponovno zgradi in požene se teste nad celotnim novim sistemom.

2.2.8 8-urni delovni čas

Programski projekt ni tek na kratke proge, ampak ga lahko primerjamo z maratonom. Ekipa, ki začne hitro teči že na samem začetku, bo energijo porabila že pred ciljem. Potrebno je prihraniti energijo do konca, to pa dosežemo z enakomernim tekom.

Pravilo ekstremnega programiranja pravi, da so nadure prepovedane. Izjema je edino zadnji teden pred končno izdajo produkcijske verzije.

2.2.9 Igra planiranja

Bistvo načrtovalne igre je delitev odgovornosti med razvojem in vodstvom. Vodstveni kader se odloči, kako pomembna je neka funkcionalnost, razvijalci pa določijo, kakšna bo cena implementacije.

Razvijalci na začetku vsake iteracije podajo stranki število točk, ki jih lahko realizirajo. Ta pojem imenujemo hitrost (*velocity*). Stranka izbere uporabniške zgodbe, ki ne smejo presegati dodeljenih točk. Obseg se določi na podlagi hitrosti v prejšnih iteracijah.

S temi preprostimi pravili, kratkimi iteracijami in pogostimi izdajami novih verzij se stranka in razvijalci hitro navadijo na tak način dela. Stranka hitro dobi občutek, kakšna bo cena in trajanje njihovega projekta.

2.2.10 Preprosta arhitektura in zasnova

Cilj vsake ekipe je preprosta zasnova. Razvijalci namenijo pozornost samo tistim uporabniškim zgodbam, ki so predvidene za tekočo iteracijo, in ne skrbijo za tiste, ki prihajajo šele v naslednjih iteracijah. Zasnova se preko novih iteracij spreminja in se podreja uporabniškim zgodbam v trenutni iteraciji.

To pomeni, da razvijalci verjetno ne bodo začeli projekta z infrastrukturo, ki npr. vključuje podatkovno bazo ali podatkovno plast. Namesto tega bodo skušali pridobiti uporabniške zgodbe in jih implementirati na najbolj enostaven način. Ekipa bo dodala infrastrukturo, ko bo uporabniša zgodba to zahtevala.

Dve vodili ektremnega programiranja za razvijalce [5]:

- **Potrebno si je zamisliti najenostavnejšo stvar, ki deluje.** Ekipe, ki delujejo po principu ekstremnega programiranja, poskušajo zmeraj poiskati preprosto rešitev. Namesto podatkovne baze, lahko uporabimo datoteke ipd. Zmeraj najprej pomislimo na najhitrejšo in enostavnejšo implementacijo uporabniške zgodbe.

- **Enkrat in samo enkrat.** Ekstremno programiranje ne dopušča podvojene programske kode.

2.2.11 Preoblikovanje

Programska koda se s časom "pokvari" in je vse manj uporabna. Ko dodamo novo funkcionalnost in popravimo napake, se struktura kode razgrajuje. Rezultat je "nered" v programski kodi, katere vzdrževanje s časom postane zelo težavno.

Ta pojav rešujemo s pogostim preoblikovanjem. Preoblikovanje je serija manjših sprememb, s katerimi izboljšamo strukturo sistema in s tem ohranimo njegovo obnašanje. Vsaka sprememba je trivialna, skupaj pa tvorijo občutno spremembo zasnove in arhitekture sistema.

Po vsaki manjši spremembi poženemo teste enot in se prepričamo, da programska koda deluje pravilno. S takim načinom ohranimo delovanje sistema.

Preoblikovanje se izvaja tekom celotnega razvoja projekta, včasih celo vsako uro ali še manj. S tem ohranjamo programsko kodo "čisto" in preprosto.

2.2.12 Metafora

Metafora je edina abstraktna praksa ekstremnega programiranja in je zato najmanj razumljiva.

Lahko jo primerjamo s sestavljanko, kjer sestavljamo manjše dele v celoto. Pri tem nas vodi ujemanje sosednjih elementov. Glavno vodilo pa vendarle niso ujemanja manjših delov, ampak celotna slika, ki nastane, ko sestavimo sestavljanko. Če sosednja elementa nimata komplementarnih oblik, vemo, da je sestavljalec naredil napako.

To je lep primer metafore, ki jo lahko označimo za sliko, ki poveže celoten sistem skupaj. Je vizija sistema, ki naredi lokacijo in obliko vseh posameznih modulov očitno. Če oblika modula ni konsistentna z metaforo, potem vemo, da je modul napačen.

Metafora je v praksi oznaka za sistem imen. Npr. bloke podatkov lahko imenujemo "rezine", ki jih program "skuha" in ga zato imenujemo "pekač". Z metaforo pa ne označujemo samo imen, ampak je vizija za celoten sistem. Je vodilo razvijalcem, kako naj izbirajo primerna imena in lokacije za funkcije, kreirajo nove razrede in metode ipd.

2.2.13 Zaključki

Ekstremno programiranje je množica preprostih in konkretnih praks, ki se vključujejo v agilen razvojni proces. Ekstremno programiranje je dobra splošnonamenska metoda za razvoj programske opreme.

2.3 Testno voden razvoj

Testno voden razvoj [3] je ena izmed praks agilnega razvoja programske opreme. Njena osnova je zelo kratek razvojni cikel:

- Razvijalec najprej napiše testni primer, ki ne uspe, saj definira neobstoječo funkcijo.
- Sledi kreiranje programske kode, zaradi katere se test uspešno izvede.
- Na koncu se koda preoblikuje.

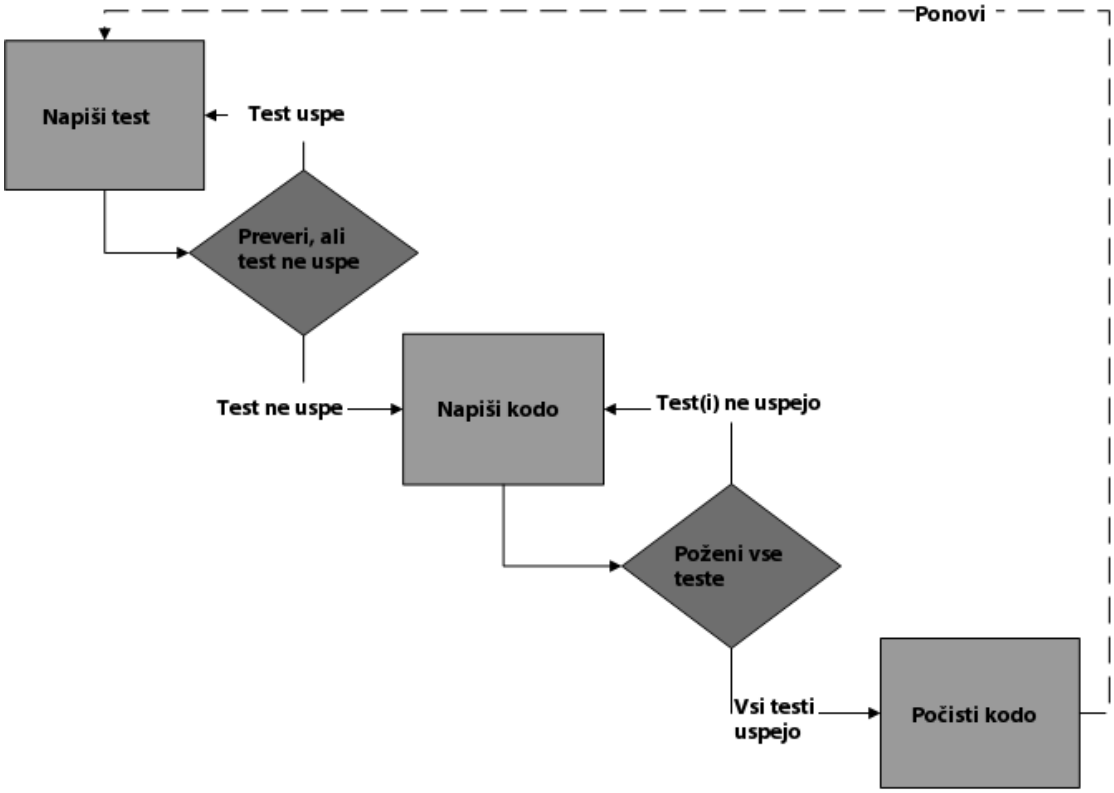
2.3.1 Zahteve

Testno voden razvoj zahteva od razvijalcev pisanje avtomatiziranih testov, ki definirajo zasnovo sistema. Testi enot vsebujejo ukaze za preverjanja - *trditve*, le ti pa vračajo vrednosti *true* ali $false$. Uspešno izvedeni testi potrjujejo pravilnost programske kode, ki se s kodiranjem novih funkcionalnosti ali preoblikovanjem spreminja. Razvijalcem so v veliko pomoč ogrodja, kot je npr. xUnit, s katerimi pišemo in poganjamo avtomatizirane teste enot programske kode.

2.3.2 Cikel testno vodenega razvoja

1. **Dodaj nov test**
   V testno vodenenem razvoju se kodiranje nove funkcionalnosti vedno začne s pisanjem testa. Test mora spodleteti, ker je napisan še prej, kot smo začeli z implementacijo funkcionalnosti. Za pisanje testa moramo jasno razumeti specifikacijo in zahteve, ki so nam razvidne iz uporabniških zgodb in uporabniških primerov.

2. **Poženi teste in preveri, če na novo dodani test spodleti**
   S tem korakom preverimo, da test deluje pravilno, in da ni po nesreči uspel, ne da bi implementirali novo funkcionalnost. Test testira tudi sam



Slika 2.1: Cikel pri testno vodenem razvoju.

sebe, saj izključuje možnost, da bo vedno uspel in bil s tem neuporaben. Novi test ne sme uspeti tudi zaradi pričakovanega razloga, kar zviša samozavest v pisanje testov, ki se uspešno izvedejo samo v pričakovanih primerih.

3. **Napiši programsko kodo**
   Naslednji korak je pisanje programske kode, zaradi katere se bo test uspešno izvedel. Trenutna programska koda mogoče ne bo idealna, vendar jo bomo z naslednjimi iteracijami izboljšali. Pomembno je, da test pokriva vso novo kodo, saj smo s tem prepričani, da ne obstajajo netestirani bloki programske kode.

4. **Poženi avtomatizirane teste in preveri, če uspejo**
   Če sedaj vsi testni primeri uspejo, smo lahko prepričani, da programska koda ustreza zahtevam testov. Na tej točki lahko začnemo s končnim

korakom cikla.

5. **Preoblikuj programsko kodo**
   Sedaj lahko kodo preoblikujemo, v kolikor je to potrebno. S poganjanjem testov smo lahko samozavestni, da preoblikovanje ni poškodovalo obstoječih funkcionalnosti. Pomemben je tudi koncept odstranjevanja podvojene kode - v našem primeru je to podvojena koda med testno in produkcijsko kodo.

2.3.3 Način razvijanja programske kode

V testno vodenem razvoju obstaja več različnih pogledov, npr. "*keep it simple, stupid* ($KISS$)"in "$You\ ain't\ gonna\ need\ it$ ($YAGNI$)". Osredotočenje na pisanje zgolj kode, ki je potrebna, da se testi izvedejo, nam prinese bolj čisto in jasno zasnovo, kakor bi jo dosegli z drugimi metodami.

Testno voden razvoj stalno ponavlja korake dodajanja testnih primerov, ki ne uspejo, implementacije funkcionalnosti z namenom, da se testi uspešno izvedejo in preoblikovanja.

2.3.4 Prednosti

Različne študije so pokazale, da nam testno voden razvoj prinaša večjo produktivnost [9]. Avtomatizirani testi zmanjšajo uporabo razhroščevalnika in s tem prihranijo mnogo časa.

Testno voden razvoj ponuja veliko več kot samo preprosto preverjanje pravilnosti programske kode - preko testov oblikujemo zasnovo aplikacije. Testni primeri nam pomagajo razumeti, kako bo stranka uporabljala posamezno funkcionalnost.

Obenem nam tak način omogoča delati majhne korake, kadar je to potrebno. Razvijalci so osredotočeni na dejstvo, da se morajo testi uspešno izvesti. Tak način zagotavlja, da je vsa programska koda pokrita z vsaj enim testom, kar daje ekipi večjo samozavest v pravilnost kode.

Čeprav imamo skupaj z avtomatiziranimi testi večjo količino programske kode, je čas implementacije aplikacije tipično krajši. Testi nam pomagajo omejiti število napak. Zgodnje in pogosto testiranje zgodaj odkriva napake in s tem prepreči, da bi morali na koncu opravljati veliko število napak, kar je drago in počasno.

Rezultat testno vodenega razvoja je modularna, prilagodljiva in razširljiva programska koda. Do tega pojava privede način razmišljanja, da na programsko opremo gledamo v smislu majhnih testov enot, ki jih neodvisno kodiramo

in testiramo, ter kasneje integriramo. K temu prispeva tudi uporaba *mock* objektov.

2.3.5 Slabosti

Tesno voden razvoj je zelo težko uporabiti v primeru, ko so za preverjanje delovanja aplikacije potrebni funkcionalni ali uporabniški test. To so npr. vmesniki za delo s podatkovno bazo, programi, ki uporabljajo prenos podatkov po omrežju. V takih primerih je pomembno, da razvijalci minimizirajo količino programske kode in maksimizirajo kodo, ki jo je mogoče testirati s pomočjo *mock* objektov.

Nujno je tudi, da imamo podporo vodstvenega kadra. V testno voden način razvoja mora verjeti celotna organizacija, sicer bo vodstvo mislilo, da je dodaten čas za teste nepotreben.

Testi morajo biti del vzdrževanja celotne aplikacije, za kar je potreben dodaten čas in denar. Vzdrževanje slabo napisanih testov je drago, obstaja tudi možnost, da takih testov ne upoštevamo tudi v primeru, ko pride do resnične napake.

Stopnjo pokritosti programske kode, ki jo dosežemo s cikli testno vodenega razvoja, je zelo težko doseči kasneje. Zato postanejo prvotni testi, ki jih pišemo od samega začetka, s časom zelo dragoceni.

Nepokritost programske kode se lahko zgodi zaradi večih razlogov. Prvi je lahko ta, da nekdo od razvijalcev ni predan takemu načinu razvoja in ne piše testov pravilno. Med razvojem lahko po nesreči ali namenoma pobrišemo skupino testov in s tem dosežemo manjšo pokritost. Posledica je lahko neodkrivanje napak, saj ne testiramo vseh funkcionalnosti.

Teste enot običajno pišejo razvijalci, ki testirano funkcionalnost tudi kasneje implementirajo. Če razvijalec npr. pozabi na preverjanje nekega parametra, potem s testom tega ne bo preveril, posledično pa programska koda ne bo delovala pravilno. Nevarnost je tudi nerazumevanje specifikacij, saj bo rezultat kodiranja nepravilna programska koda in nepravilni testi

Visoko število uspešno izvedenih testov lahko privede do lažnega zaupanja v pravilnost delovanja aplikacije in s tem do manjšega števila kasnejših testiranj, kot so npr. integracijski testi.

2.3.6 Objekti Mocks in Stubs

Teste enot imenujemo tako, ker testirajo enoto programske kode. Število testov na posameznem modulu pri tem ni pomembno. Testi enot ne smejo vsebovati

mejnih procesov aplikacije, kot so npr. dostop do podatkovne baze. Prav tako je potrebno omejiti dostope do mrežnih povezav ipd. Vzrok temu so počasna izvajanja testov enot, ki s tem odvračajo programerje od pisanja in poganjanja testov. Pridemo do pojma zunanja odvisnost.

Zunanja odvisnost je objekt v našem sistemu, s katerim komunicira naš test in nad tem nimamo kontrole. (Tipični primeri so datotečni sistemi, niti, pomnilnik, čas, itn.) [10]

Zunanje odvisnost so prisotne v integracijskih testih, nikakor pa ne v testih enot.

Ko je naša programska koda odvisna od podatkovne baze ali mrežnega servisa, se moramo take odvisnosti rešiti. Potreba sta dva koraka:

- Definirati moramo vmesnik, ki specificira obnašanje neke zunanje odvisnosti, kot je npr. mrežni servis, ali dostop do podatkovne baze
- Testirani programski kodi moramo podati navidezen objekt, ki implementira vmesnik, ki smo ga prej definirali.

Na tem mestu uvedemo pojma objektov *mock* in *stub* in podamo njuni definiciji.

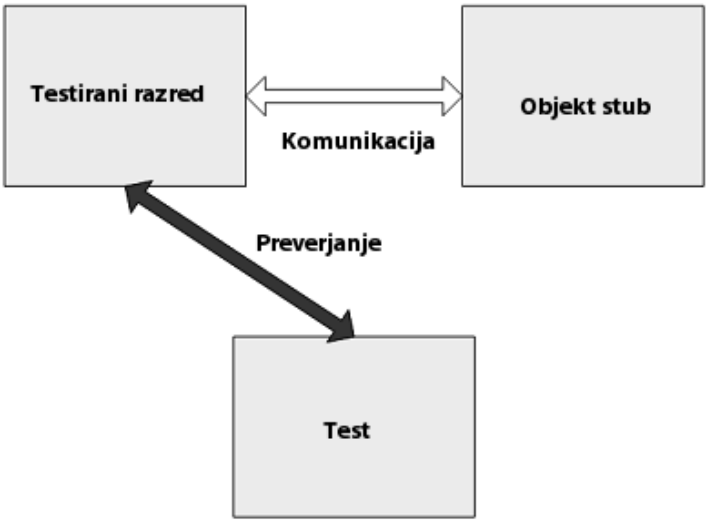
Objekt stub je zamenjava za obstoječo odvisnost, ki jo lahko kontroliramo. Z njegovo uporabo se lahko pri testiranju izognemo zunanjim odvisnostim. [10]

Objekt mock je lažen objekt, od katerega je odvisna uspešnost testa. Test se izvede uspešno, če je interakcija med testiranim objektom in mock objektom taka, kot jo test predvideva. [10]

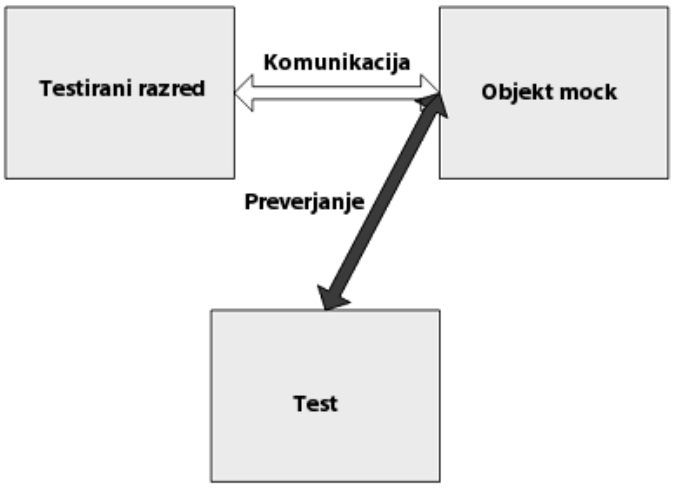
Omenili bi še, da so tovrstni objekti lahko statični, ali pa dinamični. Statične implementira programer ročno in s tem simulira obnašanje resničnega objekta.

Dinamičnih objektov ni potrebno implementirati, saj jih generiramo s pomočjo ogrodij, kot so Moq, Typemock, NMock, Rhino Mocks. Na ta način lahko simuliramo zelo kompleksna obnašanja resničnih objektov.

Bolj podroben primer bomo videli kasneje v poglavju 4.



Slika 2.2: Pri uporabi objekta stub se preverjanje izvrši nad testiranim razredom.



Slika 2.3: Komunikacija poteka med testiranim razredom in objektom mock. Za preverjanje uspešnosti testa se uporabi objekt mock.

Poglavje 3

Orodje NUnit

Orodje NUnit je del družine ogrodij xUnit, ki omogočajo avtomatizirano izvajanje testov enot in ponujajo knjižnice za hitrejše in bolj učinkovito pisanje testov ter enostaven pregled rezultatov.

V tem poglavju si bomo ogledali delovanje orodja NUnit, njegovo sintakso in pregled rezultatov izvajanja testov. Primere bomo prikazali na manjšem testnem projektu, s katerim prehajamo od teoretičnega k praktičnemu delu.

3.1 Prvi koraki z orodjem NUnit

3.1.1 Namestitev

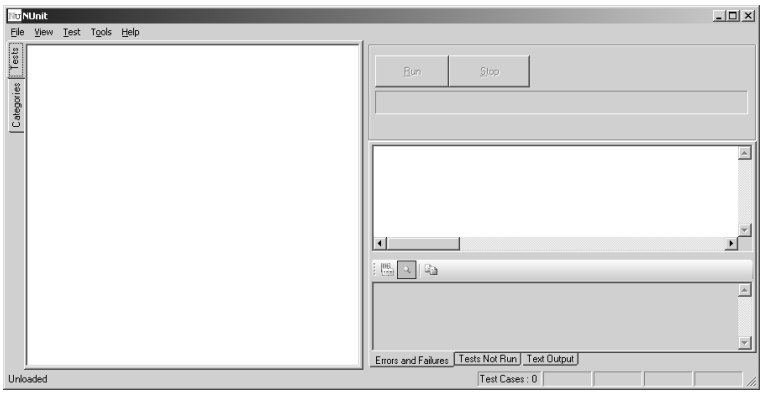
Prvi korak je prav gotovo namestitev orodja, ki ga lahko brezplačno dobimo na medmrežju. Namestitvene datoteke je mogoče najti na naslovih www.NUnit.org in www.NUnit.com. Orodje lahko uporabljamo brez omejitev in je odprtokodni produkt. Zraven dobimo izvorno kodo, ki lahko prevajamo in poljubno spreminjamo.

Po namestitvi lahko odpremo grafični vmesnik ogrodja, ki je eden od načinov poganjanja testov in pregleda rezultatov.

3.1.2 Nalaganje rešitve

Preko grafičnega vmesnika je najprej potrebno naložiti rešitev, ki jo želimo testirati. NUnit podpira več vrst projektov - lahko odpremo zbirne datoteke (.dll, .exe), kakor tudi celoten projekt (.nunit, .sln, .csproj...).

Z orodjem Microsoft Visual Studio kreiramo nov projekt in mu dodamo razred s funkcijo, ki jo bomo testirali. Naša funkcija je zelo preprosta in vrača



Slika 3.1: Grafični vmesnik je razdeljen na tri glavne dele: drevesni seznam testov na levi, sporočila in napake na zgornji desni in informacija o sledi sklada na spodnji desni

true, če je parameter *name* enak nizu "nunit", v nasprotnem primeru vrne *false*.

```
namespace NunitTutorial
{
  public class TestProject
  {
    public bool CheckName(string name)
    {
      if (name == "nunit")
      {
        return true;
      }
      return false;
    }
  }
}
```

Funkcija se mogoče ne zdi zapletena, vendar jo bomo kljub temu testirali in s tem potrdili njeno delovanje. Njena logika preveri parameter name in vrne ustrezno vrednost.

Začeli bomo s pisanjem testa za funkcijo *CheckName*:

- Rešitvi bomo dodali nov projekt, ki bo vseboval testne razrede.
- Projektu bomo dodali nov testni razred, ki bo vseboval teste.
- Znotraj testnega razreda bomo definirali test, ki bo testiral našo funkcijo *CheckName*.

3.1.3 Uporaba NUnit atributov v izvorni kodi

Orodje NUnit uporablja atributno shemo za prepoznavo in nalaganje testov. Ogrodje s pomočjo atributov prepozna pomembne dele v naloženi zbirni datoteki ("assembly") in kateri deli so testi, ki jih je potrebno poklicati.

NUnit nam v namestitvi prilaga zbirno datoteko, ki vsebuje te posebne atribute. Testnemu projektu je potrebno dodati referenco nunit.framework, da lahko začnemo s pisanjem testov.

Potrebno je definirati vsaj dva atributa, da lahko NUnit prepozna teste:

- Atribut [*TestFixture*] označuje razred, ki vsebuje avtomatizirane teste.
- Atribut [*Test*] podamo nad metodo in jo s tem označimo kot avtomatiziran test.

Sedaj definiramo nov testni razred in znotraj njega dodamo opisana atributa:

```
using NUnit.Framework;

namespace NUnitTutorialTests
{
  [TestFixture]
  public class TestProjectTests
  {
    [Test]
    public void CheckName_correctName_ReturnsTrue()
    {

    }
  }
}
```

Na tem mestu smo označili naš razred in metodo, za katero želimo, da se obnaša kot avtomatiziran test.

3.2 Pisanje prvega testa

Test enote je ponavadi sestavljen iz treh glavnih delov:

- prirejanje objektom in ustrezna nastavitev njihovih lastnosti,
- klicanje ustrezne funkcije, metode, lahko tudi večih metod oziroma funkcij,
- preverjanje rezultata.

Spodaj je podan tipičen test, ki vsebuje vse tri glavne dele.

```
using NUnit.Framework;
using NunitTutorial;

namespace NUnitTutorialTests
{
  [TestFixture]
  public class TestProjectTests
  {
    [Test]
    public void CheckName_correctName_ReturnsTrue()
    {
      //prirejanje, oziroma nastavitev
      TestProject testProject = new TestProject();

      //klicanje funkcije
      bool result = testProject.CheckName("NUnit");

      //preverjanje rezultata
      Assert.IsTrue(result, "Result for string NUnit" +
        " should be true!");
    }
  }
}
```

Za preverjanje rezultata se uporablja razred *Assert*, zato ga bomo na tem mestu bolj podrobno opisali.

3.2.1 Razred Assert

Razred Assert vsebuje statične metode, ki so most med našo kodo in ogrodjem NUnit. Njegov namen je deklarirati, da neka specifična predpostavka obstaja. Če se argumenti, ki jih podamo neki metodi tega razreda, izkažejo, da so različni od podanih, nas NUnit obvesti o neuspešnosti testa. Opcijsko lahko tudi povemo, kakšno sporočilo naj se izpiše v primeru, da test spodleti.

Razred Assert ima veliko metod, ena izmed glavnih pa je *Assert.IsTrue*(*someBooleanexpression*), ki preveri pogoj. Primeri drugih metod so še:

AreEqual

```
Assert.AreEqual(expected, actual [, string message])
```

Preveri, ali se vrednosti *expected* in *actual* ujemata.

Less / Greater

```
Assert.Less(x, y)
Assert.Greater(x,y)
```

Preveri, ali velja $x<y$ (oziroma $x>y$) za numerične tipe, ali katerikoli tip, ki implementira *IComparable*.

GreaterOrEqual / LessOrEqual

```
Assert.GreaterOrEqual(x, y)
Assert.LessOrEqual(x,y)
```

Preveri, ali velja $x \geqslant y$ (oziroma $x \leqslant y$) za numerične tipe, ali katerikoli tip, ki implementira *IComparable*.

IsNull / IsNotNull

```
Assert.IsNull(object [, string message])
Assert.IsNotNull(object [, string message])
```

Preveri, ali je dan objekt enak *null* (oziroma je različen od *null*). Sporočilo je opcijsko.

AreSame

```
Assert.AreSame(expected, actual [, string message])
```

Preveri, ali se parametra *expected* in *actual* nanašata na isti objekt. Sporočilo je opcijsko.

IsTrue

```
Assert.IsTrue(bool condition [, string message])
```

Preveri, ali je pogoj tipa *boolean* resničen. Spročilo je opcijsko.

Fail

```
Assert.Fail([string message])
```

Takoj spodleti test z opcijskim sporočilom. V praksi se ne uporablja pogosto.

Is.EqualTo

```
Assert.That(actual, Is.EqualTo(expected))
```

To je enakovredno ukazu *Assert.AreEqual*(). Metoda *Is.EqualTo*() je statična metoda, ki vrača objekt tipa *EqualConstraint*.

Is.Not.EqualTo

```
Assert.That(actual, Is.Not.EqualTo(expected))
```

To je primer omejitvene sintakse in je enakovredno ukazu *Assert.AreNotEqual*().

Is.AtMost

```
Assert.That(actual, Is.AtMost(expected))
```

Enkovreden ukazu *Assert.LessOrEqual*().

Is.Null

```
Assert.That(expected, Is.Null);
```

Preveri, ali je argument *expected* enak *null*.

Is.Empty

```
Assert.That(expected, Is.Empty);
```

Preveri, ali je argument *expected* prazna zbirka ali prazen niz.

Is.AtLeast

```
Assert.That(actual, Is.AtLeast(expected));
```

Enakovreden ukazu *Is.GreaterThanOrEqualTo*(), ki preveri $actual \geqslant y$ (oziroma $expected \leqslant y$).

Is.InstanceOfType

```
Assert.That(actual, Is.InstanceOfType(expected));
```

Preveri, da je tip parametra *actual* enak *expected*.

Has.Length

```
Assert.That(actual, Has.Length(expected));
```

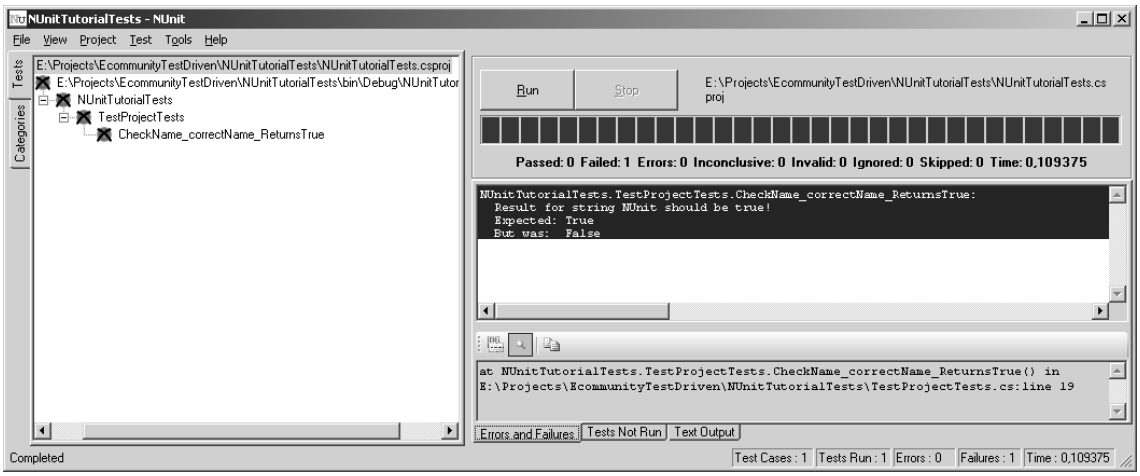
Preveri, ali je dolžina parametra *actual* enaka *expected*. To je možno testirati samo na objektu, ki ima lastonst *Length* in ni nujno da je tipa *string* ali *Collection*.

3.2.2 Poganjanje našega prvega NUnit testa

Za zagon testov moramo orodju NUnit dostaviti zbirno datoteko (v našem primeru je to kar dinamični *dll*). Najprej prevedemo projekt, ki vsebuje teste in v orodju NUnit odpremo zbirno datoteko.

To naredimo tako, da izberemo meni *File > OpenProject...*, zatem pa izberemo želeno zbirno datoteko. V našem primeru vidimo na levi strani en sam test. Za zagon testa je potreben klik na gumb *Run*. Testi so avtomatsko razdeljeni glede na *imenski prostor* - *namespace*, zato lahko poženemo teste, ki pripadajo samo določenemu imenskemu prostoru.

Kot prikazuje slika 3.2 nam je prvi test spodletel, zato sklepamo, da imamo v kodi napake. Kodo je potrebno popraviti in s testom preveriti njeno delovanje.



Slika 3.2: Če nam test spodleti, to lahko vidimo na treh mestih: testna hiearhija na levi strani postane rdeča, prikazovalnik napredka postane rdeč, vidimo tudi rdeče napake na desni strani.

3.2.3 Odprava napak in uspešen test

Hiter pogled na našo kodo nam pokaže, da smo pozabili upoštevati velike in male črke pri preverjanju niza. Želimo, da nam funkcija upošteva tudi npr. niz "NUnit". Test bi lahko prav tako spodletel, če bi imeli znotraj njega kakršnokoli izjemo, razen če pričakujemo, da nam koda v posebnih okoliščinah povzroči izjemo.

S popravkom funkcije nam test uspe:

```
if (name.ToLower() == "nunit")
```

3.2.4 Od rdeče do zelene luči

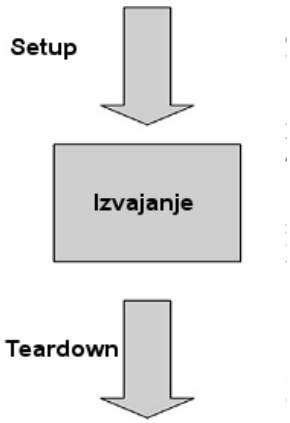
Grafični vmesnik orodja NUnit je zgrajen na preprosti ideji: vsi testi se morajo uspešno izvesti in nam dati zeleno luč, da lahko nadaljujemo z delom. Če eden izmed testov spodleti, dobimo rdečo luč, ki nas opozarja, da nekaj ni v redu z našo kodo (ali testi). Tak koncept je razširjen v svetu testiranja, posebno v testno vodenem razvoju.

3.3 Več NUnit atributov

Test enote ima več stanj v življenskem ciklu, ki jih lahko kontroliramo. Poganjanje testov je le eno izmed njih - obstajajo še posebne nastavitvene metode, ki se izvedejo pred vsakim testom in tudi za njim.

3.3.1 Setup in teardown

V NUnit poznamo posebne atribute, ki omogočajo lažjo kontrolo inicializacije in vrnitve v prvotno stanje pred in za vsakim testom. To sta atributa $[SetUp]$ in $[TearDown]$.



Slika 3.3: NUnit izvede akciji setup in teardown pred in za vsako testno metodo

Slika 3.3 prikazuje proces poganjanja testa z akcijami $[SetUp]$ in $[TearDown]$. Spodnja koda nazorno prikazuje praktičen primer uporabe teh dveh atributov:

```
using NUnit.Framework;
using NunitTutorial;

namespace NUnitTutorialTests
{
  [TestFixture]
  public class TestProjectTests
  {
    private TestProject testProject;

    [SetUp]
    public void SetUp()
    {
      testProject = new TestProject();
    }

    [TearDown]
    public void TearDown()
    {
      testProject = null;
    }

    [Test]
    public void CheckName_correctName_ReturnsTrue()
    {
      //klicanje funkcije
      bool result = testProject.CheckName("NUnit");

      //preverjanje rezultata
      Assert.IsTrue(result, "Result for string NUnit" +
        " should be true!");
    }
  }
}
```

Na metodi *setup* in *teardown* lahko gledamo kot na konstruktorje in destruktorje za teste v testnem razredu. Testni razred lahko vsebuje le po eno tako metodo, in vsaka se bo izvedla le enkrat za vsak test.

Ogrodje NUnit vsebuje tudi nekatere druge atribute, ki so uporabni pri inicializaciji in vrnitvi stanja za vse teste v nekem testnem razredu. To sta atributa [*TestFixtureSetUp*] in [*TestFixtureTearDown*], ki ju uporabljamo pri poganjanju testov nekega testnega razreda. Ta način je uporaben takrat, ko inicializacija in vzpostavitev prvotnega stanja trajata dolgo in bi radi to storili enkrat za vse teste.

3.3.2 Preverjanje pričakovanih izjem

Eden izmed pogostih testnih scenarijev je testiranje, ali nam metoda vrne pravilno izjemo, ko to pričakujemo.

Praktičen primer bi bil naš prejšnji test, ki bi moral vrniti izjemo tipa *ArgumentException*, ko mu podamo prazno ime. Če nam test ne vrne izjeme, mora spodleteti. V našo funkcijo dodamo ustrezno logiko:

```
using System;
namespace NunitTutorial
{
  public class TestProject
  {
    public bool CheckName(string name)
    {
      if ( String.IsNullOrEmpty(name) )
          throw new ArgumentException("No name provided!");

      if (name.ToLower() == "nunit")
      {
        return true;
      }
      return false;
    }
  }
}
```

NUnit ponuja poseben atribut, ki nam pomaga testirati izjeme - to je [*ExpectedException*]. Definirali bomo še test, ki bo testiral, ali nam metoda vrne pravilno izjemo:

```
[Test]
[ExpectedException(typeof (ArgumentException),
  ExpectedMessage="No name provided!")]
public void CheckName_EmptyName_Throws_Exception()
{
  testProject.CheckName(String.Empty);
}
```

3.3.3 Ignoriranje testov

Včasih se nam zgodi, da imamo napake v samih testih in lahko izvajanje takih testov preprosto prezremo. To naredimo tako, da nad test dodamo atribut [*Ignore*].

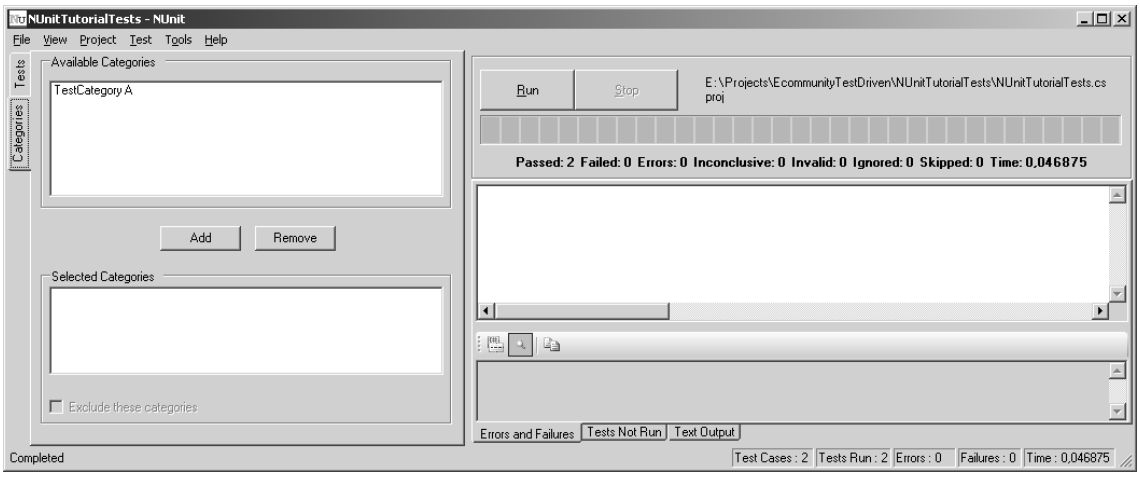
3.3.4 Nastavitev testnih kategorij

Teste lahko razdelimo po kategorijah. To nam omogoča atribut [*Category*].

```
[Test]
[Category("TestCategory A")]
[ExpectedException(typeof (ArgumentException),
  ExpectedMessage="No name provided!")]
public void CheckName_EmptyName_Throws_Exception()
{
  testProject.CheckName(String.Empty);
}

[Test]
[Category("TestCategory B")]
public void CheckName_correctName_ReturnsTrue()
{
  //klicanje funkcije
  bool result = testProject.CheckName("NUnit");

  //preverjanje rezultata
  Assert.IsTrue(result, "Result for string NUnit" +
    " should be true!");
}
```



Slika 3.4: V kodi testa lahko kategorije nastavimo. Grafični vmesnik nam omogoča poganjanje posamezenih kategorij testov.

Poglavje 4

Uporaba orodja NUnit na konkretnem primeru

Na tem primeru bomo pokazali, kako poteka testno voden razvoj v praksi. Bistvo testov enot ni zgolj samo v testiranju napisane kode - s testi neposredno definiramo arhitekturo aplikacije, saj je prvi korak vedno najprej pisanje testa, šele nato sledi programiranje posamezne funkcionalnosti. Ker so družbena omrežja (Facebook...) danes tako priljubljena, je tudi praktičen primer izbran ustrezno. Opisana koda na naslednjih straneh prikazuje programiranje nove funkcionalnosti na konkretni spletni aplikaciji, napisani v ASP.NET tehnologiji. Cilj je sprogramirati prikazovanje sporočil, ki jih lahko uporabniki pišejo na portalu. Vso kodo bomo podprli z ustreznimi testi, ki bodo tako narekovali ves razvoj.

4.1 Arhitektura

Celotna rešitev vsebuje več smiselno ločenih modulov, na katere je razdeljena aplikacija. Moduli so naslednji:

- ECDatasets - podatkovne strukture
- ECWeb - spletna stran
- ECBusiness - poslovna plast
- ECCommon - skupne funkcije
- ECDataacess - podatkovna plast

- Tests - testi

Modul s testi je namenoma definiran v svojem projektu, saj nočemo, da bi produkcijska koda vsebovala teste, ki jih bomo uporabljali pri razvoju.

4.2 Prikazovanje sporočil uporabniku

4.2.1 Uporabniška zgodba

"Uporabnik želi imeti pregled sporočil, ki jih je napisal sam, ali so mu jih napisali ostali uporabniki. Pri sporočilu naj bo prikazan avtor in vsebina spročila" Pri uporabniški zgodbi se navadno poda še oceno tveganja in ceno, kar pa za nas v tem trenutku ni bistveno.

4.2.2 Priprava podatkovnih množic

Iz uporabniške zgodbe lahko razberemo, da bomo potrebovali dva podatka: ime uporabnika in vsebino sporočilo. Imamo več možnosti, kako bomo do teh dveh podatkov dostopali - uporabil bomo kar podatkovne množice ("data sets"), ki jih ponuja Microsoftovo ogrodje .NET.

Slika 4.1: Podatkovna množica.

Podatkovne množice bomo uporabili tudi pri testih enot, kjer jih bomo napolnili s testnimi podatki. Glavni razlog je hitrost izvajanja testov, saj je dostop do pomnilnika precej hitrejši, kot do podatkovne baze oziroma diska. Pri majhni količini testov je to še sprejemljivo, pri večji pa lahko izvajanje testov traja več minut ali dlje, kar programerje odvrača od pisanja in poganjanja testov.

4.2.3 Podatkovna plast

Za dostop do podatkov bomo uporabili ločeno plast. V ta namen dodamo nov vmesnik $IWallMessage$, ki definira metode za dostop do sporočil. Lahko bi dodali samo nov razred, vendar se bo šele kasneje pri pisanju testov enot pokazala prava vrednost vmesnika.

```
using EC.ECDataaccess.Entities;

namespace EC.ECDataaccess.Abstract
{
    public interface IWallMessage
    {
        //method that returns messages
        DsEcomm.TBWALLDataTable WallMessages { get; }
    }
}
```

Sedaj ko imamo definiran vmesnik, lahko naredimo konkretne implementacije (uporaba podatkovne baze...). Na začetku uporabimo navidezno implementacijo, ki ima zbirko objektov (sporočil) kar v pomnilniku.

```
using EC.ECDataaccess.Abstract;
using EC.ECDataaccess.Entities;

namespace EC.ECDataaccess.Concrete
{
  public class DaFakeWallMessage : IWallMessage
  {
    public DsEcomm.TBWALLDataTable WallMessages
    {
      get
      {
        DsEcomm.TBWALLDataTable table = new DsEcomm.
          TBWALLDataTable();
        table.AddTBWALLRow("Andrej Rovan", "Sample Message 1");
        table.AddTBWALLRow("Andrej Rovan", "Sample Message 2");
        table.AddTBWALLRow("Miha Novak", "Sample Message 3");
        return table;
      }
    }
  }
}
```

Sporočila smo definirali ročno in jih dodali v tabelo podatkovne množice. Pripravljen imamo prvi osnutek podatkov.

4.2.4 Poslovna plast

Naša aplikacija je zasnovana na tri tirni arhitekturi - imamo podatkovno plast, poslovno plast in aplikacijsko (predstavitveno) plast. Sedaj bomo definirali poslovni razred sporočila, ki je vmesni člen med podatkovno in predstavitevno plastjo.

```
namespace EC.ECBusiness.Controllers
{
  public class BuWallMessage
  {
    private IWallMessage wallMessage;

    public BuWallMessage()
    {
      //just temporary
      wallMessage = new DaFakeWallMessage();
    }

    public List<DsEcomm.TBWALLRow> GetMessages()
    {
      return wallMessage.WallMessages.ToList();
    }
  }
}
```

4.2.5 Aplikacijska plast

Definirali smo že podatkovni in poslovni razred - manjka nam le še predstavitvena plast. Ker programiramo spletno aplikacijo, bomo podatke predstavili na spletni formi. Projektu dodamo novo formo in jo ustrezno opremimo s kontrolo, ki bo prikazala naše podatke.

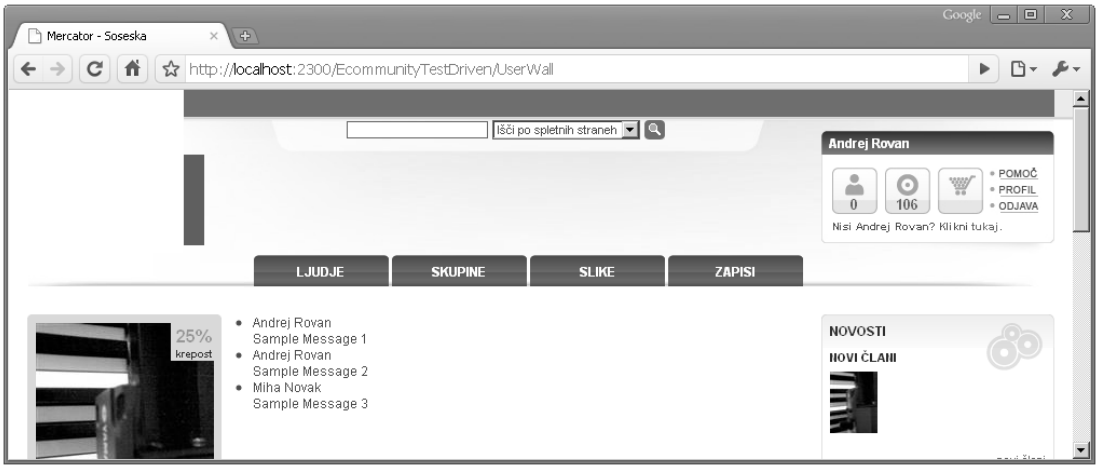
```
<asp:ListView ID="lvMessages" ItemPlaceholderID="itemContainer"
  runat="server">
  <LayoutTemplate>
    <ul>
      <asp:PlaceHolder ID="itemContainer" runat="server">
      </asp:PlaceHolder>
    </ul>
  </LayoutTemplate>

  <ItemTemplate>
    <li>
      <%# Eval("USER") %> <br />
      <%# Eval("MESSAGE") %>
    </li>
  </ItemTemplate>
</asp:ListView>
```

Zgornji kontroli moramo še nekako dostaviti podatke. To storimo s klicem metode, ki smo jo definirali v poslovnem razredu. Spodnja koda prikazuje ravno to.

```
public partial class TestWall : System.Web.UI.Page
{
  public  BuWallMessage buWallMessage = new BuWallMessage();
  protected void Page_Load(object sender, EventArgs e)
  {
    if (!IsPostBack)
    {
      BindData();
    }
  }

  private void BindData()
  {
    lvMessages.DataSource = buWallMessage.GetMessages();
    lvMessages.DataBind();
  }
}
```



Slika 4.2: Prikaz sporočil.

4.2.6 Konkretna implementacija podatkovne plasti

Do sedaj imamo implementiran samo navidezen razred podatkovne plasti, ki vsebuje sporočila v pomnilniku. Radi bi naredili konkretno implementacijo vmesnika *IWallMessage*, ki vrača prave podatke iz podatkovne baze. V ta namen dodamo nov razred.

```
namespace EC.ECDataaccess.Concrete
```

```
{
  public partial class DaSqlWallMessage : Component, IWallMessage
  {
    public DaSqlWallMessage()
    {
      InitializeComponent();
      conn.ConnectionString = DbConnectionHelper.Instance.
        ConnectionString;
      conn.DriverType = DbConnectionHelper.Instance.DbDriverType;
    }

    public DaSqlWallMessage(IContainer container)
    {
      container.Add(this);
      InitializeComponent();
    }

    private DsEcomm.TBWALLDataTable wallMessages;

    public DsEcomm.TBWALLDataTable WallMessages
    {
      get
      {
        DsEcomm dsEcomm = new DsEcomm();
        daWallMessage.Fill(dsEcomm.TBWALL);
        return dsEcomm.TBWALL;
      }
    }
  }
}
```

Zgornja implementacija uporablja objekt *DataAdapter*, ki napolni tabelo v podatkovni množici s podatki iz podatkovne baze. Zaradi poudarka na testih enot podrobnosti tu ne bomo opisovali. Bistvo je, da tu dobimo prave podatke.

4.2.7 Prvi test enote

Zgornji koraki so bili samo predpriprava za nadaljne delo, ki seveda vključuje testiranje. Naši rešitvi bomo dodali nov projekt Tests, ki bo vseboval teste enot. Radi bi napisali teste za funkcijo $GetMessages$, ki vrača seznam sporočil. Preden začnemo s testiranjem, bi se radi znebili vseh odvisnosti, ki jih ima naš testirani razred - $BuWallMessage$. Opazimo, da je $BuWallMessage$ odvisen od razreda, ki implementira naš $IWallMessage$. V testih bi lahko uporabili konkretno implementacijo $DaSqlWallMessage$, vendar tega zaradi počasnosti ne bomo storili. V takih primerih raje odvisnost nadomestimo

z objekti mock ali objekti stub. Ker si ne želimo še enkrat implementirati vmesnika *IWallMessage*, bomo raje uporabili dinamičen objekt mock. Na tem mestu imamo več alternativnih knjižnic, ki ponujajo objekte in metode za tovrstno testiranje. Ima jih NUnit, Rhino, Moq, TypeMockIsolator... Zaradi preprostosti sintakse smo se odločili za Moq. Spodaj je podana koda, ki vrača objekt, ki implementira vmesnik *IWallMessage*.

```
static IWallMessage MockDaWallMessage(TBWALLDataTable wallMessages)
{
  var mockDaWallMessage = new Moq.Mock<IWallMessage>();
  mockDaWallMessage.Setup(x => x.WallMessages).
    Returns(wallMessages);
  return mockDaWallMessage.Object;
}
```

Knjižnica nam generira želeni objekt, kateremu nastavimo, naj vrača tista sporočila, ki smo jih podali s parametrom funkciji *MockDaWallMessage*. Na tak način lahko zelo hitro ustvarimo objekte, ki nam simulirajo resnično obnašanje.

Sedaj se nam postavi vprašanje, kako naj ta objekt podamo poslovnemu razredu? Možnosti je zopet več, ena izmed njih pa se imenuje "vbrizg odvisnosti" ali "dependency injection". Poslovnemu razredu, ki vrača sporočila, bomo dodali še en konstruktor, katerega parameter bo tipa *IWallMessage*. To je bistven razlog, da smo najprej sprogramirali vmesnik.

```
public BuWallMessage(IWallMessage wallMessage)
{
  this.wallMessage = wallMessage;
}
```

S tem načinom lahko poslovnemu razredu podamo kakršenkoli objekt, ki implementira *IWallMessage* in simuliramo različne scenarije, ki jih bomo potrebovali pri testih enot.

Pripravili smo vse potrebno, da napišemo prvi test. Velja še enkrat poudariti, da se pri testno vodenem razvoju najprej napiše test, tudi, če npr. nimamo implementiranih funkcij in se posledično koda ne prevede.

```
//test that verifies if there is correct number of messages and
//that the message text is correct if we request second page
[Test]
public void GetMessages_RequestSecondPage_ReturnsTwoMessages()
{
  // Arrange: 5 wall messages in the data set
  DsEcomm.TBWALLDataTable wallMessages = new DsEcomm.
    TBWALLDataTable();
  wallMessages.AddTBWALLRow("Andrej Rovan", "sample message 1");
```

```
    wallMessages.AddTBWALLRow("Andrej Rovan", "sample message 2");
    wallMessages.AddTBWALLRow("Janez Novak", "sample message 3");
    wallMessages.AddTBWALLRow("Miha Končar", "sample message 4");
    wallMessages.AddTBWALLRow("Maja Novak", "sample message 5");

    IWallMessage daWallMessage = MockDaWallMessage(wallMessages);

    //constructor for dependency injection
    BuWallMessage buWallMessage = new BuWallMessage(daWallMessage);

    // This property doesn't yet exist, but by
    // accessing it, you're implicitly forming
    // a requirement for it to exist

    //we will use paging
    buWallMessage.PageSize = 3;

    //Act
    //request the second page
    var result = buWallMessage.GetMessages(2);

    //check the result
    Assert.IsNotNull(result, "Didn't render view");

    //Assert
    //make sure that correct objects were selected!
    DsEcomm.TBWALLRow rowOne = result[0];
    DsEcomm.TBWALLRow rowTwo = result[1];
    Assert.AreEqual("sample message 4", rowOne.MESSAGE );
    Assert.AreEqual("sample message 5", rowTwo.MESSAGE );
}
```

Vidimo, da test vsebuje naslednje logične dele:

- prirejanje (najprej pripravimo podatke, na katerih bi radi testirali funkcionalnost, v našem primeru smo kreirali instanco podatkovne množice in v tabelo dodali pet sporočil),
- klicanje ustrezne funkcije, metode (v našem primeru *GetMessages*, ki vrača seznam sporočil),
- preverjanje rezultata (v večini primerov s pomočjo razreda *Assert*, ki je del ogrodja NUnit, v našem primeru smo preverili, da nam funkcija za drugo stran vrne dve sporočili z ustreznimi podatki).

Pomemben del testov so tudi poimenovanja. V splošnem je vseeno, kako posamezen test poimenujemo, vendar je zaradi razumljivosti in lažje komu-

nikacije dobro smiselno poimenovati teste. Držali se bomo naslednje deklaracije:

```
[Test]
public void TestiranaFunkcionalnost_Scenarij_PričakovanRezultat()
```

Opazimo tudi, da v predhodnih korakih nismo definirali lastnosti *PageSize*, ki določa velikost strani. Prav tako ne obstaja funkcija *GetMessages*, ki sprejme številko strani kot parameter. Testov zaenkrat ne moremo zagnati, ker se koda ne prevede, zato nadaljujemo s preoblikovanjem.

V poslovnem razredu *buWallMessage* definiramo lastnost *PageSize*:

```
public class BuWallMessage
{
  private IWallMessage wallMessage;
  public int PageSize = 1;
  ...
}
```

Koda se še zmeraj ne prevede, zato popravimo še drugo napako - preoblikujemo funkcijo *GetMessages*, ki sedaj zgleda takole:

```
public List<DsEcomm.TBWALLRow> GetMessages(int page)
{
  return wallMessage.WallMessages
  .Skip((page - 1)*PageSize)
  .Take(PageSize)
  .ToList();
}
```

Sedaj poženemo teste, ki se uspešno izvedejo - naš prvi test enote je končan.

4.2.8 Uporabniku prijazne povezave

Naslednja stvar, ki bi jo radi dodali naši spletni aplikaciji, so uporabniku prijazni naslovi, ki jih vidi v brskalniku. Najprej naredimo test, šele nato bomo začeli z implementacijo funkcionalnosti. Hočemo, da povezava v brskalniku zgleda na ta način:

- /UserWall - vrne sporočila za prvo stran
- /UserWall/Page/PageNumber - vrne spročila za stran, ki je podana v naslovu
- /UserWall/Page/PageNumber - če stran ne obstaja, nam aplikacija preusmeri zahtevo na stran, ki izpiše uporabniku prijazno sporočilo, da stran ne obstaja.

Prvi korak je kreiranje novega razreda s testi, kjer bomo testirali prepisovanje uporabniku prijaznih povezav. Znotraj njega bomo definirali teste.

V poslovnem razredu *BuUrlRewriter* obstaja funkcija, ki nam vrne tabelo s prepisanim naslovom in vprašalnim nizom - v našem primeru je to kar številka strani do katere želimo dostopati. Kot parameter sprejeme objekt tipa *HttpContext*, ki predstavlja kontekst strani do katere dostopamo.

```
//first element in result is new url - strCustomPath
//second element in result is querry string
public string[] GetUrlMapping(HttpContext context)
{...}
```

Obenem pa uporabljamo znotraj funkcije *GetUrlMapping* še objekt tipa *HttpRuntime*, ki nam vrača dolžino navidezne poti naše aplikacije.

```
HttpRuntime.AppDomainAppVirtualPath.Length
```

Vidimo, da imamo dve odvisnosti:

- Objekt tipa *HttpContext*
- Objekt tipa *HttpRuntime*

Ker slednja razreda ne implementirata nobenega vmesnika in ne dedujeta metod in lastnosti iz nobenega abstraktnega razreda, nam ogrodje Moq tukaj žal ne more pomagati. V takem primeru uporabimo komercialno orodje TypeMockIsolator, s katerim bomo rešili odvisnost poslovnega razreda od objekta tipa *HttpRunTime*. *HttpContext* lahko simuliramo kar s kreiranjem nove instance in ustreznim prirejanjem lastnosti. Napisali bom štiri teste. Testirali bomo naslednje primere:

Naslov, ki je podan brez strani v povpraševalnem nizu - funkcija *GetUrlMapping* mora za vhod "*http://server/app/UserWall*" vrniti naslov "$\sim$ */TestWall.aspx*" in vprašalni niz "*page* = 1", kjer *server* pomeni ime strežnika, *app* pa ime aplikacije.

```
//requests first page without number in url
[Test]
public void GetUrlMapping_FirstPageWithoutNumber_GetCorrectUrl()
{
  string u = "http://localhost:2300/EcommunityTestDriven/UserWall";
  HttpRequest httpRequest = new HttpRequest("",u,"");
  StringWriter stringWriter = new StringWriter();
  HttpResponse httpResponse = new HttpResponse(stringWriter);
  HttpContext httpContext = new HttpContext(httpRequest,
    httpResponse);
```

```
  string p = "/EcommunityTestDriven";
  Isolate.WhenCalled(()=>HttpRuntime.AppDomainAppVirtualPath).
    WillReturn(p);

  string[] result = buUrlRewriter.GetUrlMapping(httpContext);
  string url = result[0];
  string querryString = result[1];

  Assert.AreEqual(url, "~/TestWall.aspx");
  Assert.AreEqual(querryString, "page=1");
}
```

Naslov, ki ima podano prvo stran v vprašalnem nizu - funkcija *GetUrlMapping* mora za vhod "*http* : *//server/app/UserWall/Page/*1" vrniti naslov " $\sim$ */TestWall.aspx*" in vprašalni niz "*page* = 1", kjer *server* pomeni ime strežnika, *app* pa ime aplikacije.

```
//requests first page with number in url
[Test]
public void GetUrlMapping_FirstPageWithNumber_GetCorrectUrl()
{
  string u = "http://localhost:2300/EcommunityTestDriven/UserWall/" +
    "Page/1";
  HttpRequest httpRequest = new HttpRequest("",u,"");
  StringWriter stringWriter = new StringWriter();
  HttpResponse httpResponse = new HttpResponse(stringWriter);
  HttpContext httpContext = new HttpContext(httpRequest,
    httpResponse);

  string p = "/EcommunityTestDriven";
  Isolate.WhenCalled(()=>HttpRuntime.AppDomainAppVirtualPath).
    WillReturn(p);

  string[] result = buUrlRewriter.GetUrlMapping(httpContext);
  string url = result[0];
  string querryString = result[1];

  Assert.AreEqual(url, "~/TestWall.aspx");
  Assert.AreEqual(querryString, "page=1");
}
```

Naslov, ki ima podano tretjo stran v vprašalnem nizu - funkcija *GetUrlMapping* mora za vhod "*http* : *//server/app/UserWall/Page/*3" vrniti naslov " $\sim$ */TestWall.aspx*" in vprašalni niz "*page* = 3", kjer *server* pomeni ime strežnika, *app* pa ime aplikacije.

```
//requests third page with number in url
[Test]
```

```
public void GetUrlMapping_ThirdPageWithNumber_GetCorrectUrl()
{
  string u = "http://localhost:2300/EcommunityTestDriven/UserWall/"+
    "Page/3";
  HttpRequest httpRequest = new HttpRequest("",u,"");
  StringWriter stringWriter = new StringWriter();
  HttpResponse httpResponse = new HttpResponse(stringWriter);
  HttpContext httpContext = new HttpContext(httpRequest,
    httpResponse);

  string p = "/EcommunityTestDriven";
  Isolate.WhenCalled(()=>HttpRuntime.AppDomainAppVirtualPath).
    WillReturn(p);

  string[] result = buUrlRewriter.GetUrlMapping(httpContext);
  string url = result[0];
  string querryString = result[1];

  Assert.AreEqual(url, "~/TestWall.aspx");
  Assert.AreEqual(querryString, "page=3");
}
```

Naslov, ki ima podan nepravilen vprašalni niz namesto strani - funkcija *GetUrlMapping* mora za vhod "*http* : *//server/app/UserWall/Page/asd*" vrniti naslov " $\sim$ */404.aspx*", kjer *server* pomeni ime strežnika, *app* pa ime aplikacije.

```
//requests bad request - querry string is not the number
//returns 404 page
[Test]
public void GetUrlMapping_BadRequest_Get404Error()
{
  string u = "http://localhost:2300/EcommunityTestDriven/UserWall/" +
    "Page/asd";
  HttpRequest httpRequest = new HttpRequest("",u,"");
  StringWriter stringWriter = new StringWriter();
  HttpResponse httpResponse = new HttpResponse(stringWriter);
  HttpContext httpContext = new HttpContext(httpRequest,
    httpResponse);

  string p = "/EcommunityTestDriven";
  Isolate.WhenCalled(()=>HttpRuntime.AppDomainAppVirtualPath).
    WillReturn(p);

  string[] result = buUrlRewriter.GetUrlMapping(httpContext);
  string url = result[0];
  string querryString = result[1];
```

```
    Assert.AreEqual(url, "~/404.aspx");
    Assert.AreEqual(querryString, "");
}
```

Zatem poženemo vse teste, ki se seveda ne izvedejo. To je znak, da moramo začeti z implementacijo. Funkcijo *GetUrlMapping* je potrebno dopolniti, da bo generirala uporabniku prijazne naslove, ko bo dostopal do strani s prikazom sporočil.

```
...
//url rewriting code for userwall
else if (permalinkType.Equals("userwall"))
{
    strCustomPath = "~/TestWall.aspx";
    querryString = "";

    if (arr.Capacity == 2)
    {
        //redirect to page 1
        querryString = "page=1";
    }
    else if ((arr.Capacity == 4) && (arr[2].Equals("page")))
    {
        //redirect to page that is in the permalink
        try
        {
            querryString = "page=" + Convert.ToInt16(arr[3]);
        }
        catch (Exception)
        {
            strCustomPath = "~/404.aspx";
            querryString = "";
        }
    }
    else
    {
        strCustomPath = "~/404.aspx";
        querryString = "";
    }
}
...
//result
return new[] {strCustomPath, querryString};
```

Dopolnitev funkcije *GetUrlMapping* z zgornjo kodo povzroči, da se testi izvedejo. Tako smo prepričani, da naša koda deluje pravilno. To bo lahko

preveril tudi drug programer, ko bo spreminjal funkcijo *GetUrlMapping*. Testi enot so zelo hiter pokazatelj, ali nam npr. funkcija vrača pravilne rezultate in dajejo programerju samozavest pri spreminjanju stare kode.

4.2.9 Prikaz številk strani

Naslednja funkcionalnost, ki jo je potrebno implementirati, so številke strani, preko katerih dostopamo do posamezne strani s sporočili. Začnemo s pisanjem testov. Projektu s testi dodamo nov razred *PagingHelperTests*, v katerem bomo testirali funkcijo, ki nam generira HTML kodo s številkami in povezavami.

```
namespace Tests
{
[TestFixture]
public class PagingHelperTests
{
  [Test]
  public void PageLinks_MethodExists()
  {
    PagingHelpers.PageLinks(0, 0, null);
  }

  [Test]
  public void PageLinks_Generate3Pages_ReturnTags()
  {
    string links = PagingHelpers.
      PageLinks(2, 3, i => "Page/" + i);
    // This is how the tags should be formatted
    Assert.AreEqual(@"<a href=""Page/1"">1</a><a href=""Page/2""
      class=""selected"">2</a><a href=""Page/3"">3</a>", links);
  }
}
```

Prvi test samo preveri, ali funkcija *PageLinks* obstaja. Programerju daje vedeti, da je potrebno sprogramirati funkcijo, ki nam bo vračala HTML kodo. Drugi test je že bolj konkreten in preverja rezultat funkcije *PageLinks* ki smo jo definirali tako:

```
//currentPage - current page
//totalPages - total number of pages
//func<int, string> pageUrl - function that returns url link
public static string PageLinks(int currentPage, int totalPages,
  Func<int, string> pageUrl)
```

Funkcija nam mora za drugo trenutno stran, skupno tremi stranmi, ter navedeno funkcijo vrniti naslednji rezultat:

```
<a href="Page/1">1</a><a href="Page/2" class="selected">2</a>
<a href="Page/3">3</a>
```

Ker imamo izbrano drugo stran, je v povezavi na stran dva podan tudi CSS razred *selected*, s katerim bomo kasneje oblikovali vizualno podobo spletne strani.

Ko začnemo s kodiranjem, najprej opazimo, da poslovni razred za sporočila ne vsebuje nekaterih potrebnih lastnosti. Zaradi arhitekture hočemo, da se trenutna stran in skupno število strani določi v poslovnem razredu in ne na predstavitveni plasti.

Potrebno bo spremeniti prvi test in vključiti v testiranje tudi trenutno stran in število strani. Najprej preoblikujemo test:

```
//test that verifies if there is correct number of messages and
//that the message text is correct if we request second page
[Test]
public void GetMessages_RequestSecondPage_ReturnsTwoMessages()
{
  ...
  Assert.AreEqual(2, buWallMessage.CurrentPage);
  Assert.AreEqual(2, buWallMessage.TotalPages);
  Assert.AreEqual(2, result.Count);
  ...
  //make sure that correct objects were selected!
  DsEcomm.TBWALLRow rowOne = result[0];
  DsEcomm.TBWALLRow rowTwo = result[1];

  Assert.AreEqual("sample message 4", rowOne.MESSAGE );
  Assert.AreEqual("sample message 5", rowTwo.MESSAGE );
}
```

Če se spomnimo testa za nazaj:

- velikost strani je tri,
- funkcija nam za drugo stran vrne dve sporočili.

Iz tega je jasno, da nam mora poslovni razred vrniti skupno dve strani, število sporočil na drugi strani pa je dva, ker prva stran že vsebuje tri sporočila. Poslovnemu razredu dodamo lastnosti:

```
private int totalPages;
private int currentPage;

public int TotalPages
{
  get { return totalPages; }
```

```
}

public int CurrentPage
{
  get { return currentPage; }
}
```

Prav tako preoblikujemo funkcijo *GetMessages*:

```
public List<DsEcomm.TBWALLRow> GetMessages(int page)
{
  int numMessages = wallMessage.WallMessages.Rows.Count;
  //izračunamo število strani
  totalPages = (int)Math.Ceiling((double)numMessages / PageSize);
  //priredimo trenutno stran
  currentPage = page;

  return wallMessage.WallMessages
  .Skip((page - 1)*PageSize)
  .Take(PageSize)
  .ToList();
}
```

Zatem poženemo test in vidimo, da smo s preoblikovanjem poslovnega razreda uspešno zaključili. Vrnemo se nazaj k funkciji, ki nam generira HTML kodo za prikaz strani in jo dokončamo:

```
namespace EC.ECCommon.HtmlHelpers
{
  public static class PagingHelpers
  {
    public static string PageLinks(int currentPage, int totalPages,
      Func<int, string> pageUrl)
    {
      StringBuilder result = new StringBuilder();
      for (int i = 1; i <= totalPages; i++)
      {
        result.Append(@"<a href=""" + pageUrl(i) + @"""");
        if (i == currentPage)
        {
          result.Append(@" class=""selected""");
        }
        result.Append(">" + i + "</a>");
      }
      return result.ToString();
    }
  }
}
```

Z dodano funkcijo se testi uspešno izvedejo, potrebno je le še dodati klic funkcije na aplikacijski plasti:

```
<div class="pager">
  Stran: <%= PagingHelpers.PageLinks(buWallMessage.CurrentPage,
             buWallMessage.TotalPages,
             x => Request.Url.GetLeftPart(UriPartial.Authority) +
               Request.ApplicationPath + "/UserWall/Page/" + x)
      %>
</div>
```

Zaradi ponovne uporabe smo funkciji *PageLinks* kot tretji parameter dodali funkcijo, ki nam preslika številke strani v povezave. To je tudi eno izmed vodil agilnih metod.

Slika 4.3: Prikaz sporočil s stranmi.

Poglavje 5

Sklep

Testno voden razvoj je prav zagotovo uporaben način razvoja kvalitetne programske opreme, ki je testirana že med samim razvojem in ne šele potem, ko smo z njim zaključili.

Ekstremno programiranje zahteva drugačen način razmišljanja, saj je čisto nasprotje od klasičnega načina razvoja programske opreme. Isto velja za testno voden razvoj. To je lahko zelo težavno, če nimamo podpore vseh članov ekipe, v kateri bi radi uveljavili takšen način. Npr. spreminjanje zahtev s strani stranke kadarkoli med razvojem aplikacije je takšen primer, s katerim se zagotovo ne bi vsi strinjali. Težko je tudi spremeniti miselnost razvijalcev, da najprej začnejo s pisanjem testov in šele kasneje z implemetacijo funkcionalnosti.

Takšne probleme sem opazil tudi sam, ko sem implementiral praktičen del diplomskega dela. Na začetku sem porabil precej več časa, kot bi ga običajno, saj nisem bil navajen na takšen način dela. Pisanje testov se sprva zdi nepotrebno, vendar se njihova vrednost pokaže šele kasneje, ko lahko kadarkoli preverimo delovanje programske kode.

Prav tako je vzdrževanje stare kode, podprte s testi enot, veliko lažje. Ko spreminjamo del kode, lahko s testi enot takoj preverimo, ali nismo česa pokvarili. To nam vliva večjo samozavest in spodbuja preoblikovanje, ki je pomemben pojem pri ekstremnem programiranju.

Testno voden razvoj ima svoje prednosti in slabosti, vendar dobri rezultati kažejo na porast uporabe testno vodenega razvoja. To se odraža predvsem pri razvoju poslovnih aplikacij in raznih tečajev, na katerih se razvijalci učijo testno vodenega razvoja.

Slike

Literatura

[1] **David Cohen, Mikael Lindvall, Patricia Costa**, An Introduction to Agile Methods, Advances In Computers, Vol 62, 2004, strani 1-64

[2] **K. Beck**, Extreme Programming Explained, Embrace Change, Addison-Wesley, 2000

[3] **K. Beck**, Test-Driven Development by Example, Addison Wesley, 2003

[4] **Steven Sanderson**, Pro ASP.NET MVC Framework, Paperback, 2009

[5] **C. Martin, Micah Martin**, Agile Principles, Patterns, and Practices in C♯, Paperback, 2006

[6] **Wikipedia**, Extreme Programming, http://en.wikipedia.org/wiki/Extreme_programming

[7] **Wikipedia**, Test Driven Development, http://en.wikipedia.org/wiki/Test-driven_development

[8] **Agile Manifesto**, http://agilemanifesto.org/

[9] **Erdogmus, Hakan; Morisio, Torchiano**, "On the Effectiveness of Test-first Approach to Programming". Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). Retrieved 2008-01-14. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."

[10] **Roy Osherove**, The Art of Unit Testing with Examples in .NET, Paperback, 2009

[11] **Andy Hunt, Dave Thomas, Matt Hargett**, Pragmatic Unit Testing in C♯ with NUnit, 2nd Edition, Paperback, 2007