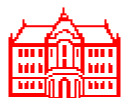

Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

BORUT TERPINC

Sprejemni testi v procesu agilnega razvoja programske opreme

MAGISTRSKO DELO

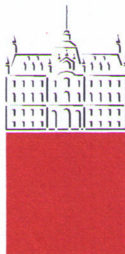
Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana, 2011

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.: 116-MAG-ISO/2011
Datum: 24. 06. 2011

Borut TERPINC, univ. dipl. org.

Ljubljana

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **Sprejemni testi v procesu agilnega razvoja programske opreme**

Acceptance tests in the process of agile software development

Tematika naloge:

Agilne metode za razvoj programske opreme (npr. Scrum in ekstremno programiranje) uporabljajo za specifikacijo zahtev ti. uporabniške zgodbe (angl. user stories), ki vsebujejo tudi sprejemne teste, s katerimi pokažemo, da je zgodba ustrezno realizirana. Avtomatsko izvajanje sprejemnih testov lahko bistveno pripomore k hitrejšemu razvoju in večji kakovosti programske opreme. Medtem ko je avtomatsko testiranje posameznih programskih enot že dodobra uveljavljeno, je avtomatizacija sprejemnega testiranja še predmet raziskav.

V svoji magistrski nalogi proučite agilni pristop k razvoju programske opreme, izvedite primerjavo testov enot in sprejemnih testov ter prikažite možnosti za avtomatsko izvajanje sprejemnih testov. Na tej osnovi predlagajte proces razvoja programske opreme, ki bo združil koncept sprejemnih testov in testno vodenega razvoja v celoto. Proces naj bo na voljo kot alternativna oblika razvoja programske opreme. Predlagani proces uporabite na primeru razvoja spletne strani za potrebe Statističnega urada Republike Slovenije. Spletna stran naj prikazuje različne statistične podatke o posameznih slovenskih občinah.

Mentor:

prof. dr. Viljan Mahnič

Dekan:

prof. dr. Nikolaj Zimic

Original tema magistrske naloge

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

magistrskega dela

Spodaj podpisani **Borut Terpinc**,

z vpisno številko **63060500**,

sem avtor magistrskega dela z naslovom:

Sprejemni testi v procesu agilnega razvoja programske opreme

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod vodstvom mentorja **prof. dr. Viljana Mahniča**,
- so elektronska oblika magistrskega dela, naslova (slov., angl.), povzetka (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela
- in soglašam z javno objavo elektronske oblike magistrskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____________________   Podpis avtorja: ________________________

Zahvala

Zahvaljujem se prof. dr. Viljanu Mahniču za strokovne usmeritve, nasvete, priporočila ter pomoč pri iskanju literature za pripravo magistrske naloge. Prav tako se zahvaljujem Statističnemu uradu Republike Slovenije, ki mi je omogočil praktično realizacijo magistrske naloge.

KAZALO

POVZETEK ____ 1

ABSTRACT ____ 2

KRATICE ____ 4

1 UVOD ____ 5

1.1 Cilji magistrske naloge ____ 7

1.2 Metode dela ____ 7

1.3 Zgradba nadaljnjega besedila ____ 7

2 PROCES RAZVOJA PROGRAMSKE OPREME ____ 9

2.1 Klasičen proces slapa ____ 9

2.2 Uporabniške zgodbe ____ 11

2.3 Agilni pristop ____ 13

2.3.1 Ekstremno programiranje ____ 15

2.4 Testiranje programske opreme ____ 19

2.4.1 Testiranje po metodi bele škatle ____ 20

2.4.2 Testiranje po metodi črne škatle ____ 20

2.5 Prevzemanje programske opreme ____ 22

2.5.1 Osnovni proces prevzemanja ____ 22

2.5.2 Zahteve programske opreme s stališča sprejemanja ____ 22

3 PRIMERJAVA SPREJEMNIH TESTOV IN TESTOV ENOT ____ 26

3.1 Življenjski cikel testov ____ 26

3.2 Sprejemni Testi ____ 27

3.3 Testi enot ____ 31

3.3.1 Testiranje enot v štirih fazah ____ 34

3.3.2 Upravljanje odvisnosti ____ 35

3.3.3 Testni dvojniki ____ 39

3.4 Podobnosti in razlike med sprejemni testi in testi enot ____ 43

4 TESTNO VODEN RAZVOJ S SPREJEMNIMI TESTI ____ 47

4.1 Testno voden razvoj ____ 47

4.2 Sprejemni testi v kontekstu ekstremnega programiranja ____ 50

4.2.1 Testno voden razvoj s sprejemnimi testi ____ 53

5 UPORABA TESTNO VODENEGA RAZVOJA PRI RAZVOJU SPLETNE APLIKACIJE ______ 58

5.1 Splošen Pregled orodij ______ 58

5.1.1 Fitnesse ______ 59

5.1.2 Visual Studio ______ 61

5.2 Razvoj spletne strani občine v številkah ______ 64

5.2.1 Uporabniške zgodbe ______ 64

5.2.2 Uporabniške zgodbe po iteracijah ______ 68

5.2.3 Sprejemni testi ______ 69

5.2.4 Uporaba orodja Fitnesse ______ 70

5.2.5 Izdelava testne povezave in slepih funkcionalnosti ______ 72

6 SKLEPNE UGOTOVITE ______ 80

7 LITERATURA IN VIRI ______ 82

7.1 Literatura ______ 82

7.2 Viri ______ 84

Kazalo slik

Slika 1: Klasičen proces slapa 9
Slika 2: Primer uporabniške zgodbe 12
Slika 3: Agilni pristop 14
Slika 4: Proces razvoja programske opreme po metodologiji ekstremnega programiranja 16
Slika 5: Klasičen proces sprejemanja programske opreme 22
Slika 6: Funkcionalne in nefunkcionalne zahteve 23
Slika 7: Primer testa enote – koda 32
Slika 8: Testiranje v štirih fazah 35
Slika 9: Princip uporabe testnih dvojnikov 36
Slika 10: Posredni izhodi in vhodi iz testiranega subjekta 37
Slika 11: Nastavljiv in ročno kodiran testni dvojnik 40
Slika 12: Interakcija z lažnim objektom 42
Slika 13: Delitev testa večje enote na več testov manjše enote 46
Slika 14: Proces testno vodenega razvoja – rdeče/zeleno/preoblikuj 49
Slika 15: Testno voden razvoj s sprejemnimi testi 54
Slika 16: Testno voden razvoj s sprejemnimi testi in slepimi funkcionalnostimi 56
Slika 17: Tipični Fitnesse test 60
Slika 18: Prikaz testov v orodju Visual Studio 62
Slika 19: Generacija testov s pomočjo orodja Visual Studio 64
Slika 20: Uporabniške zgodbe, ki se nanašajo na splošnega uporabnika 65
Slika 21: Uporabniške zgodbe, ki se nanašajo na upravljavca 66
Slika 22: Uporabniške zgodbe, ki jih lahko implementiramo kasneje 66
Slika 23: Definirane omejitve pri razvoju 67
Slika 24: Primer Fitnesse testa – test pravilnih podatkov o občini 72
Slika 25: Programska koda slepe funkcionalnosti, ki vrača podatke o občini 74
Slika 26: Programska koda slepe funkcionalnosti, ki vrača absolutno vrednost razlike 75
Slika 27: Programska koda testa enote, ki vrača absolutno vrednost razlike 76
Slika 28: Primer kode, kjer je test sestavljen iz več manjših testov enote 76
Slika 29: Primera testov funkcionalnosti, ki testirata metodo »VrniAbsolutnoVrednostRazlike« 77
Slika 30: Implementacijska koda, ki vrača absolutno vrednost razlike podatkov 78

Kazalo tabel

Tabela 1: Primer sprejemnega testa 29
Tabela 2: Primerjava testov enot in sprejemnih testov 43
Tabela 3: Atributi, s katerimi označimo razrede in metode kot testne 62
Tabela 4: Razredi in metode za preverjanje vrednosti, definirani v orodju za testiranje Visual Studio 63
Tabela 5: Povezava Fitnesse testa in testnega okolja 73

POVZETEK

Ključna problema za neuspešnost projektov v informacijski tehnologiji (IT) sta nejasna slika naročnika in spreminjajoče se zahteve poslovnega sveta. Naročnik ima na začetku naročila le bledo sliko o tem, kakšen bo končni izdelek projekta. Ker so IT projekti po naravi kompleksni in dolgotrajni, se skozi proces razvoja velikokrat pojavijo nove zahteve. Pri uporabi klasičnega načina razvoja se na začetku procesa analizira in popiše zahteve IT sistema, nato pa se razvoj programske opreme izvaja v skladu s podrobno dokumentacijo.

Agilni modeli razvoja programske opreme poizkušajo spremeniti to togost in približati razvoj programske opreme hitro se spreminjajočim zahtevam uporabnika. Testno voden razvoj in uporaba sprejemnih testov sta praksi ekstremnega programiranja, ki uporabljata model agilnega razvoja programske opreme.

Doslej je bila pozornost testno vodenega razvoja usmerjena predvsem v testiranje enot. V okviru magistrske naloge pa več pozornosti namenjamo sprejemnim testom. Sprejemne teste najprej primerjamo s testi enot in ugotovimo, da se razlike nanašajo predvsem na lastnosti, ki se večinoma dotikajo sodelujočih oseb, hitrosti izvedbe in integracije v razvojna orodja. Če na oba testa gledamo kot na testa določenega dela programa, lahko ugotovimo, da je z vidika programerja sprejemni test pravzaprav test enote, ki ni najmanjši možni del programa, ampak se nahaja na najvišjem nivoju hierarhije. Sprejemni test tako testira več med seboj povezanih enot programa.

Sprejemne teste v kontekstu ekstremnega programiranja premaknemo v začetni cikel razvoja programske opreme in jih uporabimo kot vodilo za nadaljnji razvoj in dokumentacijo. V magistrski nalogi predstavimo testno voden razvoj s sprejemnimi testi kot proces razvoja programske opreme. Predstavljen proces uporablja principe testno vodenega razvoja, kot vodilo za razvoj pa uporablja sprejemne teste, ki so definirani na podlagi uporabniških zgodb.

Predlagan proces nato uporabimo za razvoj spletne strani za potrebe Statističnega urada Republike Slovenije. V nalogi najprej predstavimo razvojna orodja, ki omogočajo izvajanje predlaganega procesa, nato po principih ekstremnega programiranja in testno vodenega razvoja s sprejemnimi testi razvijemo spletno stran Slovenske občine v številkah.

Ključne besede: Agilni modeli, ekstremno programiranje, testno voden razvoj, testno okolje, testi enot, sprejemni testi, uporabniške zgodbe, Fitnesse.

ABSTRACT

One of the key reasons for failures of information technology (IT) projects are the customer's unclear picture of the product and the constant changes of demands in the business world. Usually, in the early phases of project development the customer sees only a pale image of the desired final product. As the IT projects are inherently complex and time-consuming, new requirements often show up during the development process. Conventional development process starts with requirements analysis. When requirements are analyzed and written down, the software development is done in exact accordance with the detailed documentation.

Agile models try to respond to the lack of flexibility in such cases and move software development closer to the rapidly changing costumer's requirements. Test driven development and acceptance tests are the practices of extreme programming, an agile model of software development.

The attention in test driven development has been mainly focused on the unit tests. This master thesis focuses to the acceptance tests. Acceptance tests are first compared to the unit tests. It has been concluded that differences relate to the properties that mainly concern people, who participate in the tests, the speed of execution and the integration with development tools. If both types of tests are considered as a test of certain program part, it can be concluded, that from a programmers perspective, acceptance test is actually a unit test without the term "smallest part" in its definition. In this regard, acceptance test are the "big" unit tests used to check on several interconnected units of the program at the highest level of test hierarchy.

When it comes to the extreme programming, acceptance tests are moved to the initial stage of software development cycle and can therefore be used to drive the software development process and documentation. Test driven development with acceptance tests is regarded as one type of the software development process. The process presented uses test-driven development principles and acceptance tests to drive the development of requirements based on the user stories.

The process presented here is being used in the development of website for the Statistical Office of Slovenia. First, the development tools for the implementation of the proposed process are presented. Later on, the principles of extreme programming and test-driven development with acceptance tests are used, in order to develop the website named *Slovenian municipalities in numbers.*

Key words: Agile software development, extreme programming, test driven development, acceptance tests, unit tests, user stories, Fitnesse.

Kratice

Kratica	Pomen
ADO.NET	Okolje za povezovanje programskih objektov s podatkovnimi viri (ActiveX Data Object for .NET)
API	Aplikativni programski vmesnik (application programming interface)
EP	Ekstremno programiranje
FIT	Framework for integrated testing – okolje za izvajanje povezanih testov
HTML	HyperText Markup Language – označevalni jezik za oblikovanje večpredstavnostnih dokumentov
IT	Informacijska tehnologija
OK	Odvisna komponenta
TD	Testni dvojnik
TS	Testirani subjekt
TVR	Testno voden razvoj

1 UVOD

Čeprav je računalništvo že dodobra spremenilo način življenja človeškega rodu in je gonilo razvoja ter dostopa do informacij in povezanosti, pa je pot do realizacije projektov, povezanih z uporabo informacijskih tehnologij (IT), negotova in zahtevna. Po podatkih Standish Group, ki izdeluje letne raziskave o uspešnosti IT projektov, je bilo leta 2009 uspešnih samo 32 % vseh začetih IT projektov [37].

Razloge za neuspešnost lahko najdemo v več dejavnikih, ki pa dandanes niso več povezani s tehnološkimi omejitvami, saj današnja tehnologija omogoča praktično vse. Problemi se dotikajo predvsem organizacijskih strani in strani poslovnega procesa. Vsakdo, ki je sodeloval pri procesu izdelave programske opreme ali večjega računalniškega sistema, je zagotovo negodoval nad spreminjajočimi se zahtevami uporabnika.

Eden ključnih problemov za neuspešnost IT projektov je nejasna slika naročnika. Naročnik ima na začetku naročila le bledo sliko o tem, kakšen bo končni izdelek projekta. Ker so IT projekti po naravi kompleksni in dolgotrajni, se skozi proces razvoja velikokrat pojavijo nove zahteve. Pri uporabi klasičnega načina razvoja se na začetku procesa analizira in popiše zahteve IT sistema. Glede na potrjene zahteve se potem izvedejo faze tehničnega načrtovanja, programiranja, končne realizacije in produkcije. Končni produkt velikokrat ne ustreza naročniku, saj si je v fazi analize drugače predstavljal njegovo sliko. K neuspešnosti pripomorejo še nezaupanje uporabnikov do novih sistemov, kompleksnost programske opreme in hitro se spreminjajoče tehnologije.

V analizi programskih razvojnih projektov podjetja QSM so raziskovalci pri poskusu merjenja napredka razvoja ugotovili, da se je zaradi spreminjanja zahtev pri polovici projektov plan tako spremenil, da niso več razbrali prvotne razvojne in finančne dokumentacije projekta. Hitro spreminjajoči se svet informacijskih tehnologij zahteva hitro odzivnost na zahteve stranke, zato slepo sledenje pripravljenemu načrtu več ne zadovoljuje potreb strank [35].

Agilni modeli razvoja programske opreme poizkušajo spremeniti to togost in približati razvoj programske opreme hitro se spreminjajočim zahtevam uporabnika. V kontekstu agilnega razvoja se pojavljajo razvojne prakse, ki poizkušajo skrajšati razvojni cikel in približati končni produkt zahtevam uporabnika. Ker se prakse nanašajo na nenehno sodelovanje razvojne ekipe z naročnikom, se proces lažje prilagaja spremembam in specifični situaciji naročnika.

Ekstremno programiranje je agilna disciplina razvoja programske opreme, ki kot ključne vrednote poudarja enostavnost, komunikacijo in povratno informacijo. Osredotoča se na pravice in dolžnosti vlog, ki jih nosijo stranke, managerji in programerji [13]. Ekstremno programiranje uporablja pogoste izdaje različic in kratke razvojne cikle z namenom

izboljšanja produktivnosti in kvalitete programske opreme. Pomembni praksi ekstremnega programiranja sta tudi uporaba sprejemnih testov in testno voden razvoj.

Doslej je bila pozornost testno vodenega razvoja usmerjena predvsem v testiranje enot. Testiranje enot je metoda, pri kateri testiramo posamezne enote programske kode. Enota je najmanjši del programske kode, ki ga lahko testiramo, kar navadno predstavlja funkcijo ali proceduro. Tako se je razvoj s pomočjo testno vodenega razvoja dotikal predvsem tistega dela procesa razvoja programske opreme, v katerem sodelujejo predvsem programerji.

V nasprotju z najmanjšimi možnimi tehničnimi enotami programske opreme se sprejemni testi dotikajo dejanskih funkcij, ki jih mora programska oprema zagotoviti. Sprejemni testi pravzaprav semantično – tekstovno opisujejo, kakšno delo naj bi programska oprema opravljala. Klasično se sprejemni testi pišejo in izvajajo potem, ko je programska oprema že razvita, saj določajo kriterije, po katerih naročnik sprejme ali zavrne produkt.

V magistrskem delu smo poizkušali povezati koncepte sprejemnih testov in testov enot ter razviti proces razvoja programske opreme, ki temelji na vnaprejšnjem pisanju sprejemnih testov. Razvoj programske opreme smo naslonili na sprejemne teste, ki predstavljajo začetek in konec cikla razvoja, ter proces poimenovali testno voden razvoj s sprejemnimi testi. Proces smo tudi praktično preverili, saj smo za Statistični urad Republike Slovenije na osnovi predlaganega procesa razvili spletišče Občine v številkah.

1.1 CILJI MAGISTRSKE NALOGE

Motivacijo za pisanje magistrske naloge so predstavljali naslednji cilji:

- Raziskati področje testov enot in sprejemnih testov ter preučiti in povezati koncepte iz obeh področij.
- Raziskati področje testno vodenega razvoja in sprejemnih testov v okviru področja ekstremnega programiranja.
- Predstaviti testno voden razvoj s sprejemnimi testi kot proces razvoja programske opreme.
- Razviti spletno stran po principih testno vodenega razvoja s sprejemnimi testi.

1.2 METODE DELA

V okviru magistrske naloge so uporabljene naslednje metode dela:

- Študij teorije na področju metodologij testno vodenega razvoja, testov enot in sprejemnih testov.
- Izgradnja procesa razvoja programske opreme – testno vodenega razvoja s sprejemnimi testi.
- Razvoj delujoče programske rešitve po principih testno vodenega razvoja s sprejemnimi testi.

1.3 ZGRADBA NADALJNJEGA BESEDILA

V prvem poglavju so definirani cilji magistrskega dela in metode dela. V drugem poglavju so predstavljeni osnovni pojmi, ki se dotikajo procesa razvoja programske opreme. Opisana sta klasičen in agilni pristop k razvoju programske opreme. Nadalje so opisane različne metode testiranja in uporabniške zgodbe, prav tako je opisan tudi pojem prevzemanja kot kriterija za končno ustreznost programske opreme.

Tretje poglavje se dotika primerjave sprejemnih testov in testov enot. V posameznem podpoglavju so najprej podrobno opisani sprejemni testi, nato pa še testi enot. Poglavje se zaključi s študijo povezanosti obeh konceptov.

V četrtem poglavju je izdelan proces razvoja – testno voden razvoj s sprejemnimi testi. Najprej je predstavljen koncept testno vodenega razvoja. Nato so sprejemni testi umeščeni v kontekst ekstremnega programiranja. Poglavje se zaključi z opisom testno vodenega razvoja s sprejemnimi testi.

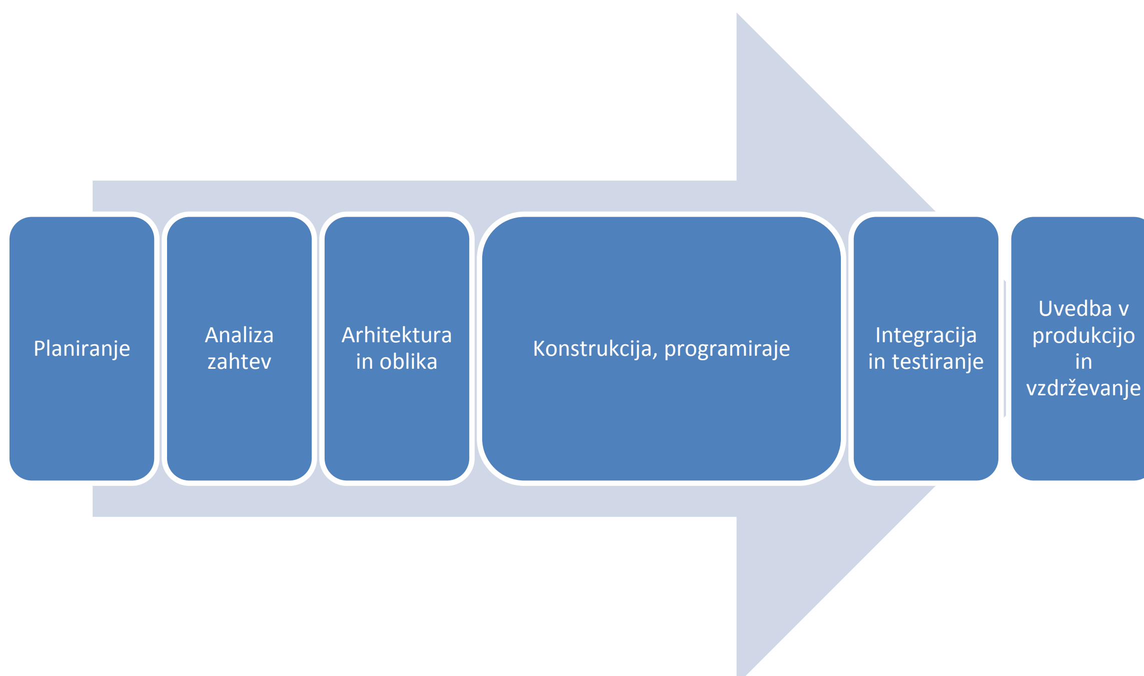
Peto poglavje zajema praktično realizacijo. Najprej so opisana orodja, ki smo jih uporabili za razvoj rešitve, nato je opisan praktični postopek uporabe procesa pri razvoju spletne strani.

Magistrsko nalogo zaključimo v šestem poglavju, kjer navedemo sklepne ugotovitve.

2 PROCES RAZVOJA PROGRAMSKE OPREME

2.1 KLASIČEN PROCES SLAPA

Klasičen proces slapa sloni na organiziranju in razdelitvi projekta v ločene faze. V vsaki izmed faz je točno določena vrsta dela. Prav tako ima faza točno določene vhodne in izhodne zahteve, posamezne faze pa naj se med sabo ne bi prekrivale. Med vhodom trenutne faze in izhodom predhodne faze so aktivnosti porazdeljene tako, da med fazami ni zamika. Vsaka faza se obravnava kot enota z odločitvami o napredovanju v naslednjo fazo. Klasičen proces slapa je prikazan na Sliki 1.



Slika 1: Klasičen proces slapa

Vsaka faza je razdeljena v posamezne delovne aktivnosti, ki se nanašajo na delo v fazi. Vzemimo za primer fazo analize zahtev. V tej fazi lahko delo razdelimo med analitike glede na temo zahtev. Ko se delo nadaljuje v fazi konstrukcije, delo razdelimo med razvijalce po posameznih modulih. Razen predaje med konstrukcijo in testiranjem, ki vsebuje tudi programsko kodo, se informacije med posameznimi fazami navadno predajajo v obliki dokumentov. Pri klasičnem procesu testiranja se testiranje izvede na koncu celotnega procesa, preden produkt vpeljemo v produkcijo. V posamezne faze so vključene različne osebe, ki celotnega procesa ne poznajo oz. ga poznajo le bežno. V posameznih fazah pa se izvajajo naslednje aktivnosti [36]:

1. Faza planiranja
    a. Sestanek z naročnikom, definicija zahtev.
    b. Usklajevanje in razumevanje zahtev.

2. Faza analize zahtev
    a. Določanje problema skupaj z izbranimi cilji produkta ali storitve.
    b. Identifikacija omejitev.
    c. Podrobna specifikacija sistema.
    d. Specifikacija funkcij sistema.

3. Faza arhitekture in oblike
    a. Sprememba specifikacij v arhitekturne podrobnosti: Struktura podatkov, arhitektura programske opreme, podrobnosti posameznih algoritmov, predstavitve vmesnikov.
    b. Definicija relacij med strojno, programsko opremo in pripadajočimi vmesniki.
    c. Izdelava podrobne specifikacije brez napak.

4. Faza konstrukcije in programiranja
    a. Pretvorba izdelanih specifikacij v programsko okolje.
    b. Kodiranje in preverjanje programske kode glede na specifikacije.
    c. Testiranje posameznih delov programske kode.

5. Faza integracije in testiranja
    a. Združevanje posameznih delov sistema v celoto.
    b. Testiranje delovanja sistema kot celote.
    c. Odpravljanje napak.

6. Faza uvedbe v produkcijo in vzdrževanje
    a. Zadovoljevanje spreminjajočih se potreb uporabnika.
    b. Prilagajanje spremembam v zunanjem okolju.
    c. Popravki napak, ki so bile spregledane v času testiranja.
    d. Izboljševanje učinkovitosti sistema.

Nekatere prednosti modela slapa [19]:

- Preverjanje ustreznosti izhoda se izvaja v vsaki fazi modela.
- Disciplina je pomemben del vseh faz.
- Dokumentacija je prisotna v vsaki fazi modela.

Model slapa je najstarejša in najbolj uporabljena paradigma, ki prinaša probleme predvsem zaradi togega formata. Nekatere slabosti modela slapa so [19]:

- Iteracije uporablja indirektno, zato lahko spremembe v posameznem delu predstavljajo veliko zmedo v nadaljevanju projekta.

- Naročnik velikokrat nima jasne ideje, kaj točno želi od produkta, zato je težko že v naprej določiti vse zahteve, ki jih bo sistem podpiral.
- Naročnik vidi delujočo verzijo produkta šele potem, ko je bil produkt narejen, kar lahko privede do problemov, saj so nekatere nerazumljivosti in napake odkrite šele na koncu.

2.2 UPORABNIŠKE ZGODBE

Uporabniška zgodba predstavlja enega ali več stavkov, napisanih v poslovnem jeziku, ki zajemajo bistvo tistega, kar želi uporabnik doseči s programom ali računalniškim sistemom. Posamezna uporabniška zgodba je omejena tako, da jo lahko napišemo na majhen listek ali kartico. Majhna velikost listka oz. kartice omejuje dolžino napisanega besedila na kartici. Bistvo uporabniških zgodb je vpliv na razvoj računalniškega sistema. Uporabniške zgodbe morajo napisati naročniki in jih uporabiti kot instrument za posredovanje zahtev pri razvoju programske opreme. Uporabniška zgodba v računalniški obliki je predstavljena na Sliki 2.

Na ta način uporabniške zgodbe predstavljajo hiter način rokovanja z zahtevami uporabnika, brez uporabe preobsežnih administrativnih nalog, ki se tičejo pisanja in administracije formalizirane dokumentacije zahtev. Tako je glavni namen uporabniških zgodb hiter odziv na hitro se spreminjajoče zahteve uporabnika.

Uporabniška zgodba opisuje funkcionalnosti, ki bodo predstavljale neko končno vrednost za uporabnika. Uporabniška zgodba predstavlja tri vidike [7]:

- Zapisan opis zgodbe, ki jo uporabimo kot osnovo za planiranje računalniškega programa. Posamezen zapis posredno opravlja tudi funkcijo opomnika.
- Pogovore, ki jih o posamezni zgodbi opravimo z naročnikom, uporabimo za natančno opredelitev podrobnosti o posamezni zgodbi.
- Teste, ki posredujejo podatke in podrobnosti, na podlagi katerih lahko preverimo, ali je izbrana uporabniška zgodba zaključena.

Ron Jefferis je poimenoval te tri vidike čudovita sestavljenka kartice, pogovora in potrditve (ang. Card, Conversation and Confirmation), saj so uporabniške zgodbe tradicionalno napisane na majhne papirnate kartice [12]. Čeprav je kartica najbolj vidna manifestacija uporabniške zgodbe, pa ni najbolj pomembna. Kartica pravzaprav predstavlja zahteve uporabnika. Tako si je uporabniško zgodbo najlaže predstavljati na sledeč način: »Medtem ko kartica vsebuje tekst uporabniške zgodbe, se podrobnosti dogovorimo v pogovoru, ki jih nato še potrdimo.« [12]

Uporabnik lahko iz spletišča prebere tematske članke, ki se povezujejo na spletno stran s podatki.

Slika 2: Primer uporabniške zgodbe

Ker uporabniške zgodbe predstavljajo funkcionalnost, ki mora predstavljati neko vrednost za uporabnika, je potrebno implementacijske podrobnosti izpustiti iz zgodb. Primera slabih uporabniških zgodb:

- Program bo napisan v programskem jeziku C#.
- Program bo uporabljal ADO.NET komponente za povezovanje s podatkovno bazo.

Na prvi primer lahko sicer pogledamo z dveh vidikov. Če uporabniška zgodba predstavlja del večjega sistema, ki ga bo uporabljal klasični uporabnik, potem je ta podrobnost nepotrebna. V primeru, da je ta uporabniška zgodba napisana za programski vmesnik (API), pa ta podrobnost predstavlja ključno informacijo.

Druga uporabniška zgodba končnemu uporabniku ne predstavlja nobene vrednosti, saj opisuje način, kako se program povezuje s podatkovno bazo. Končnega uporabnika mogoče zanima, kakšen bazni sistem bo program uporabljal, zagotovo pa ga ne zanima, na kakšen način se bo programska koda povezovala s sistemom za upravljanje z bazami podatkov.

Vzemimo za primer uporabniško zgodbo, ki opisuje, da bo uporabnik lahko s spletne strani prebral tematske članke, ki opisujejo posamezno občino. S strani s tematskimi članki pa bodo določene povezave na spletne strani z numeričnimi podatki o posamezni občini.

Od tega, da se z naročnikom dogovorimo, da bo uporabnik lahko s spletišča prebral tematske članke, ki se povezujejo na spletno stran s podatki, do takrat, ko zares začnemo programersko delo, moramo razrešiti še veliko vprašanj, kot so na primer:

- Kakšni teksti se bodo prikazovali na straneh s tematskimi članki?
- Na kakšen način se bodo članki povezovali s podatki o občini?
- Ali bodo članki na kakršen koli način združeni?
- Kako bomo članke vpisovali v sistem?

Posamezne podrobnosti lahko izrazimo kot dodatne uporabniške zgodbe. Tako velja pravilo, da je bolje imeti več krajših uporabniških zgodb kot malo število dolgih. Kadar je zgodba predloga jo lahko poimenujemo epska [7]. Epske zgodbe lahko razdelimo v več krajših zgodb, vendar moramo biti pri razčlenjevanju zgodb previdni, da zgodb ne razčlenimo preveč. Predvsem pri izražanju podrobnosti posameznih zgodb je bolje, da pustimo podrobnosti odprte in prepustimo razvojni ekipi, da uskladi te podrobnosti z naročnikom. S tem uporabniški zgodbi dodamo nekaj manjših komentarjev, vendar pri tem ne smemo pozabiti, da je ključ pogovor in ne komentarji na uporabniški zgodbi. Tako uporabniki kot razvojniki

po končanem razvoju nimajo pravice do sklicevanja na komentarje k uporabniškim zgodbam.

Prav tako uporabniške zgodbe ne predstavljajo direktnih obveznosti glede razvoja programske opreme. Te obveznosti so definirane v sprejemnih testih. Uporabniške zgodbe naj bodo kratke in jedrnate. Priporočljivo je, da se na zadnjo stran kartic napiše pričakovanja uporabnikov projekta. Pričakovanja so najbolje zajeta v obliki sprejemnih testov. Testi na drugi strani kartic uporabniških zgodb morajo biti kratki in odprti za posodobitve. Teste lahko kadarkoli odstranimo ali dodamo. Glavni cilj pisanja testov je pripenjanje dodatnih informacij o zgodbi, ki bodo pomagale razvojniku oceniti končanost razvoja uporabniške zgodbe.

2.3 AGILNI PRISTOP

Februarja 2001 se je 17 svetovalcev za razvoj programske opreme zbralo na zborovanju z namenom, da poiščejo odgovor na problem, ki ga postavljajo hitro se spreminjajoče zahteve poslovnega sveta. Kot odgovor so predstavili agilne metodologije, ki so opisale načine za hiter in fleksibilen odziv na spremembe [27]. Pomemben izdelek zborovanja je agilni manifest, ki je izpostavil osnovne vrednote agilnega razvoja. Te vrednote so:

- Posamezniki in interakcije imajo večjo vrednost kot procesi in orodja.
- Delujoča programska oprema ima večjo vrednost kot izčrpna dokumentacija.
- Sodelovanje z naročnikom ima večjo vrednost kot pogajanja o pogodbenih zahtevah.
- Odgovarjanje na spremembe ima večjo vrednost kot sledenje planu.

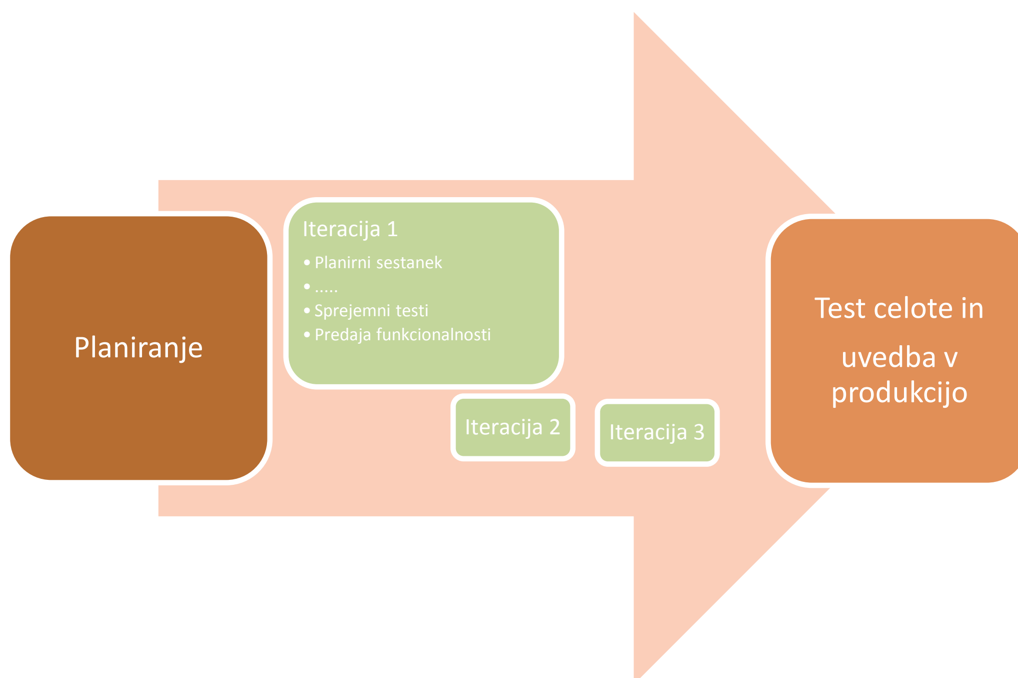
Agilni pristop ponuja drugačen pogled na razvoj inovativnih produktov. V nasprotju s procesom slapa, kjer se odločitve o prehodu v naslednjo fazo sprejemajo na koncu posamezne faze, agilne metodologije uporabljajo iterativni pristop. Faze agilnega pristopa uporabljajo znanja, ki so bila pridobljena v prejšnjih fazah. Agilni pristop ta znanja uporablja za razvoj naslednje zbirke funkcionalnosti [10].

Agilni razvoj je paradigma, ki jo vodi upravljanje, načrtovanje in dostava kreativnih, inovativnih, rizičnih in kompleksnih projektov. Osnove agilnega razvoja so zapisane v agilnem manifestu. V manifestu najbolj opazno izstopa [32]:

- Uporaba 1–4 tedenskih iteracij, ki dostavijo delujoče in testirane dele funkcionalnosti.
- Kratke in pogoste izdaje.
- Vzdrževanje sistema v neprestanem delujočem stanju. Stanje preverjamo s pomočjo avtomatskih testov.
- Organizacija sloni na večfunkcijskih in kompetentnih ekipah (definicija, testiranje, kodiranje in grajenje).
- Serijsko izdelovanje projektov s pomočjo rednega določanja prednosti in ovrednotenja nalog.

- Raziskovanje in prilagajanje procesa in produkta v vsaki stopnji razvoja.

Večina agilnih metod torej uporablja iterativni in postopen pristop k razvoju. Potem, ko je bilo izvedeno začetno planiranje, se projekt razdeli na posamezne razvojne iteracije. Vsaka končana iteracija prinese delček delujoče programske kode [19].



Slika 3: Agilni pristop

Slika 3 prikazuje preprost agilni pristop s tremi iteracijami. V realnih projektih je teh iteracij veliko več. Vsaka iteracija se začne s planirnim sestankom in konča s sprejemnim testiranjem. Posamezne iteracije se najprej oblikuje glede na značilnosti uporabniških zgodb, potem pa se za vsako od njih opravi celoten razvojni proces. Naročnik je med razvojem vedno na voljo in je odgovoren za opis zahtev in podrobnosti, prav tako pa je naročnik odgovoren za definicijo sprejemnih testov posamezne uporabniške zgodbe. V procesu agilnega razvoja naročniki razvijalcem podajajo teste v obliki podrobne definicije zahtev, kar omogoča razvijalcem uporabo testov v času razvoja programske opreme. Razvijalci takoj potem, ko vsi testi določene uporabniške zgodbe preživijo, funkcionalnosti predajo naročniku in skupaj izvedejo vmesne sprejemne teste. V primeru, da se pri testiranju pojavi hrošč, je prva prioriteta razvijalca odprava tega, po možnosti še v isti iteraciji.

Večinoma se agilni produkti naslanjajo na rek »dostavljaj zgodaj in pogosto«. V teoriji lahko definiramo, da ima vsaka iteracija na koncu samostojen rezultat, ki ga lahko vpeljemo v produkcijsko okolje. V praksi navadno potrebujemo več iteracij, da pridemo do izdaje (ang. release), ki jo lahko vpeljemo v produkcijsko okolje. Izdaja različice predstavlja najmanjši

nabor najbolj pomembnih uporabniških zgodb, ki za stranko predstavljajo neko smiselno celoto [5].

Agilni način razvoja se vodi in meri glede na vrednosti, ki jih razvoj prinese k viziji in vrednosti projekta. Podobno kot metoda slapa tudi agilni način zahteva določeno mero discipline, če hoče biti uspešen. Pomembni projekti zahtevajo prilagodljivo in učinkovito medfunkcijsko sodelovanje in obljubljajo delujočo kodo ne glede na majhne vire in kratke časovne okvirje. Prvi rezultati agilnega razvoja so vidni že zelo hitro, pravzaprav takoj, ko sta vizija in vrednost projekta določeni.

Agilni razvoj kaže dobre rezultate predvsem pri projektih, kjer vsebina projekta še nastaja in so riziki visoki. Preko preprostih empiričnih metod agilni razvoj omogoča boljše, cenejše in konstantne rezultate [32]. V nasprotju s klasičnimi metodologijami, ki obljubljajo predvidljivost, stabilnost in visoko stopnjo varnosti, pa agilni pristopi obljubljajo večje zadovoljstvo strank, manjše pojavljanje napak in hitrejši razvoj kot rešitev za hitro se spreminjajoče zahteve poslovnega okolja. Agilni razvoj je pravzaprav paradigma, ki s pomočjo povratnih informacij obvladuje razvoj kreativnih, inovativnih, kompleksnih in stroškovno tveganih projektov [6].

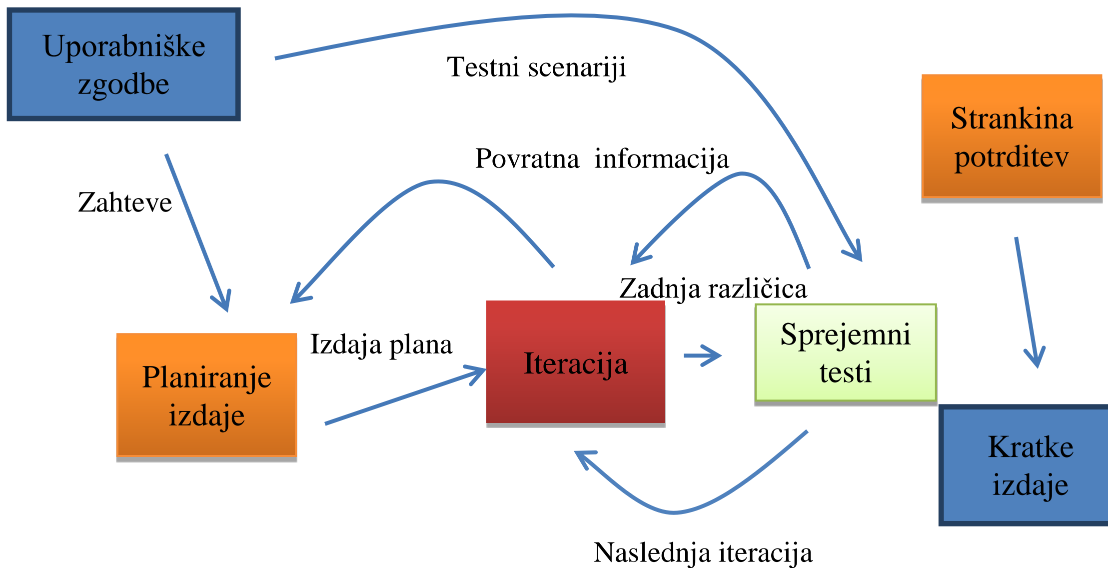
2.3.1 Ekstremno programiranje

Ekstremno programiranje (EP) je agilna disciplina razvoja programske opreme, ki kot ključne vrednote poudarja enostavnost, komunikacijo in povratno informacijo. Osredotoča se na pravice in dolžnosti vlog, ki jih nosijo stranke, menedžerji in programerji [13].

Metodologija ekstremnega programiranja se je razvila kot odgovor na težave, ki so nastale zaradi dolgih razvojnih ciklov, kot so jih poznali običajni razvojni modeli [5].

Ekstremno programiranje je torej metoda agilnega programiranja, ki uporablja pogoste izdaje različic in kratke razvojne cikle z namenom izboljšanja produktivnosti in kvalitete programske opreme. EP predstavlja razvoj programske opreme kot delo razvojne ekipe, ki ne zajema samo razvojnega kadra, ampak vključuje tudi naročnika in menedžerje. EP je pravzaprav preprost proces, ki združuje te ljudi in jim pomaga do končnega skupnega uspeha. Proces razvoja programske opreme s pomočjo EP je predstavljen na Sliki 4.

Metodologija ekstremnega programiranja je namenjena majhnim in srednje velikim skupinam. Beck predlaga, naj bo delovna skupina velika do 20 članov [4]. Za uspeh EP je pomembna komunikacija in koordinacija med posameznimi člani. Ta mora biti neprestano omogočena, pomembno pa je tudi, da so člani v istem prostoru. Prav tako je za uspeh pomembno, da skupina nima odpora do takega načina dela in te prakse sprejme.



Slika 4: Proces razvoja programske opreme po metodologiji ekstremnega programiranja

Ekstremno programiranje je namenjeno razvojnim ekipam, ki se nahajajo na eni lokaciji. Principi ekstremnega programiranja so tako namenjeni srednje velikim projektom, ki razvijajo programsko opremo hitro in fleksibilno. EP se torej dotika treh tipov udeležencev [13]:

- Strank: stranke so osebe, ki imajo zahteve za programsko opremo, ki jo je potrebno razviti. Naloga strank je, da se naučijo preprostega in učinkovitega načina izražanja potreb ter vodenja projekta do končnega uspeha.

- Programerjev: naloge programerjev so definiranje arhitekture, načrtovanje sistema, pisanje testov in končne programske kode. Poleg tega se morajo programerji naučiti, kako obvladati spreminjajoče se zahteve in graditi zaupanje stranke v njihovo delo.

- Menedžerjev: menedžerji so tisti, ki nadzirajo vire za projekt. Prav tako morajo menedžerji znati spremljati proces in meriti kvaliteto produkta. Pomembna naloga menedžerjev je tudi oceniti, uravnotežiti in usmeriti delo programerjev na tak način, da lahko predvidijo datum zaključka.

Ekstremno programiranje je pravzaprav zbirka obstoječih idej in praks, ki so v pomoč udeležencem pri razvoju projektov. Te prakse so [13]:

- Stranka je vedno na voljo (ang. on-site customer): akterji iz poslovnega sveta so vedno na voljo za vprašanja in oceno trenutnega stanja razvoja. S tem je dosežen hiter odziv na strankine zahteve.

- Kratke izdaje (ang. short releases): posamezna izdaja naj bo tako majhna, kot je

minimalno potrebno, da še predstavlja zaključeno celoto. Izdaje naj bodo pogoste. Vidni zaključeni deli celote naj se izdajo vsakih nekaj tednov.

- Igra načrtovanja (ang. planning game): razvoj programske opreme je proces, ki zajema neprestani dialog med naročnikom in razvojno ekipo. Prav tako pa razvoj zajema tako odločitve poslovnega dela kot tudi tehničnega osebja. V igri načrtovanja sodelovanje med poslovnim in tehničnim osebjem prinese pomembne odločitve in smernice dela. Določijo se prioritete posameznih delov, datumi in obsegi izdaj, ocene časa potrebnega za implementacijo posameznih delov sistema, način organiziranosti razvojne skupine, podroben urnik razvoja itd.

- Ocena zgodb (ang. story estimation): EP planira in gradi programsko opremo s pomočjo uporabniških zgodb. Zgodbe morajo biti dovolj podrobne, da lahko programer oceni težavnost implementacije. Zgodbe morajo biti napisane tako, da jih lahko testiramo. Naloga stranke pri oceni zgodb je ta, da izbere zgodbe, ki bodo v izbranem času ponudile največjo poslovno vrednost v okviru razvite programske opreme.

- Stranka definira izdajo (ang. customer defines release): za zadovoljstvo stranke je pomembno, da posamezna izdana različica programske opreme realizira to, kar si je stranka ob definiciji predstavljala. V izogib nesporazumom prepustimo stranki, da določi, kaj bo v posamezni izdaji realizirano.

- Planiranje in usmerjanje iteracij (ang. planning and stirring iteration): naloga posamezne iteracije je, da ob zaključku predstavi nabor novih, delujočih in stestiranih uporabniških zgodb, ki so razvite in pripravljene za uporabo. Razvojna pot do končnega produkta vodi preko več različnih iteracij [5]. Kdaj, katere in v kakšnem zaporedju naj bodo posamezne iteracije realizirane, določi stranka.

- Metafora (ang. metaphor): s pomočjo metafor poimenujemo posamezne tehnične elemente sistema. Metafore služijo za boljšo komunikacijo med tehničnim in poslovnim osebjem, ki sodeluje v projektu.

- Preprosta arhitektura (ang. simple design): program je zgrajen na osnovi preproste in čiste arhitekture. Tak način omogoča hitro grajenje kvalitetne programske kode.

- Kvaliteta kode (ang. code quality): kvaliteta programske kode se odraža v posameznih lastnostih programske kode. Kvaliteta je večja, če je koda napisana tako, da omogoča preprosto branje logike in namena posameznih delov programske kode. Dobro napisana koda ne vsebuje podvojevanj logike, uporablja čim manjše število razredov in metod, ideje in namere programerja pa so dobro označene in izražene.

- Hitri načrtovalni sestanki (ang. quick design session): pred začetkom programiranja posameznih delov se razvojna ekipa sestane na kratkem sestanku. Na sestanku je prisoten tudi naročnik. Skupaj z naročnikom razvojna ekipa poišče odgovore na posamezna odprta vprašanja, ki so pomembna za nemoteno nadaljevanje razvoja.

- Programiranje v parih (ang. pair programming): vso programsko kodo napišejo programerji v parih. Programerja se izmenjujeta pri programiranju. Medtem ko prvi programer piše programsko kodo, drugi napisano kodo preverja. Nato se programerja zamenjata.

- Skupno lastništvo programske kode (ang. collective code ownership): razvojna skupina si deli lastništvo programske kode, kar pomeni, da lahko vsakdo spreminja kodo od vsakogar. Skupina mora skrbeti za kvaliteto kode in nepréstano komunikacijo.

- Neprestana integracija (ang. continuous integration): sistem je neprestano integriran, kar pomeni, da je vedno na voljo delujoča različica sistema. Zaključene dele je potrebno pogosto testirati in povezati v delujočo celoto. Tak način omogoča hiter razvoj brez problemov, ki nastanejo zaradi združljivosti posameznih elementov.

- Standardi kodiranja (ang. coding standard): razvojna ekipa mora imeti jasno postavljene standarde programske kode. Ti standardi morajo biti jasni vsem programerjem. Pomembna je sporočilnost programske kode in poznavanje vzorcev programiranja.

- Sprejemni testi (ang. acceptance tests): s pomočjo sprejemnih testov preverimo, če je bila programska oprema pravilno razvita. Sprejemne teste lahko napišejo programerji, neodvisni testerji ali stranke same. Testi omogočajo povečanje zaupanja v dejstvo, da sistem deluje tako, kot si je stranka zamislila.

- Testi enot (ang. unit tests): s pomočjo neprestanega avtomatskega poganjanja testov enot, programsko kodo držimo v delujočem stanju. Testi omogočajo hitre spremembe in podporo skupnemu lastništvu kode, saj preprečujejo napake v kodi.

- Preoblikovanje programske kode (ang. refactoring): preoblikovanje programske kode je način programiranja, ki omogoča grajenje kvalitetne programske kode. Varnost pri preoblikovanju nam omogočajo testi enot.

- 40-urni delavnik (ang. 40 hour week): projektno delo je način realizacije v EP. Tako delo zahteva velike napore vseh sodelujočih. EP zato že pri načrtovanju razvoja upošteva 8-urni dnevni delavnik. Daljši dnevni delavniki se ne obnesejo, ker razvojna

ekipa s časom izgubi motivacijo za delo.

Posamezne prakse predstavljajo jedro ekstremnega programiranja. Prakse niso pravila, ampak predstavljajo vodila, na katera se naslanjamo pri razvojnih projektih. Uporabo posameznih praks prilagodimo razvojnemu projektu in organizaciji, pri tem pa pazimo, da se držimo preprostosti, komunikacije in povratne informacije [21].

Končno lahko ekstremno programiranje označimo kot pristop k razvoju programske opreme, ki omogoča vsem udeležencem v procesu, da delajo tisto, kar najbolje znajo in potrebujejo. Tako programerji izdelajo programe, od katerih stranke pridobijo želeno poslovno vrednost [13].

2.4 TESTIRANJE PROGRAMSKE OPREME

Če želimo definirati pojem testiranja programske opreme, je najbolje, da izhajamo iz njegovega namena. Zanima nas torej, kaj bomo s pomočjo testiranja programske opreme dosegli oz. kakšen je naš končni cilj.

Glavni namen testiranja programske opreme je pravzaprav opraviti raziskavo, ki lastnikom produkta ali storitve omogoča pregled kvalitete testiranega subjekta. Testiranje programske opreme prikazuje torej objektiven in neodvisen pogled na programsko opremo ter tako omogoča lastnikom, da razumejo in cenijo tveganja implementacije programske opreme. Testne tehnike so v prvi vrsti namenjene iskanju napak v programski opremi, vendar to ni njihov glavni namen [28].

Na testiranje programske opreme lahko gledamo tudi s stališča potrjevanja in preverjanja sledečih lastnosti programske opreme:

- Programska oprema izpolnjuje poslovne in tehnične zahteve, ki so bile določene preko inženiringa in razvoja.
- Programska oprema deluje tako kot je predvideno.
- Programsko opremo lahko zgradimo (implementiramo) z istimi karakteristikami, kot jih kažejo testirani elementi.

Izvedba testiranja programske opreme je neodvisna od stanja v procesu razvoja, saj ga lahko izvedemo v katerikoli fazi. Različni modeli razvoja programske opreme določajo testiranje v različnih stopnjah stanja razvoja programske opreme. Pri tradicionalnih modelih razvoja se večina testiranja opravi potem, ko so bile zahteve definirane in je bil proces kodiranja zaključen. Novejši modeli, ki se naslanjajo na agilne metodologije, pa raje uporabljajo testno voden razvoj, ki nalaga veliko več testiranja razvijalcu, ki testiranje uporablja v procesu razvoja, še preden je programska koda napisana in je poslana posebnim testnim skupinam, ki

so za testiranje specializirane.

Metode testiranja tradicionalno delimo na testiranje po metodi črne in bele škatle. Ta dva načina se pravzaprav razlikujeta glede na vidik, ki ga testni inženir uporablja, ko načrtuje testne primere.

2.4.1 Testiranje po metodi bele škatle

Kadar ima tester dostop do notranjih podatkovnih struktur in algoritmov, ki te implementirajo, potem tako testiranje imenujemo testiranje po metodi bele škatle [2]. Načini testiranja po metodi bele škatle so :

- Testiranje API-ja (application programming interface) – testiranje aplikacije s pomočjo javnih in privatnih API-jev.
- Kodna pokritost – testiranje pokritosti kode s pomočjo testov, ki vsaj enkrat izvedejo vse stavke v kodi.
- Metode vstavljanja napak – omogočajo izboljšanje pokritosti kode s pomočjo uvedbe napak pri testiranju kode.
- Metode testiranja mutacij – pri metodi namensko spremenimo delček kode, nato pa preverjamo odziv celotnega sistema.
- Statično testiranje – preverjanje smiselnosti kode in algoritmov, predvsem preverjanje sintaktičnih napak.

2.4.2 Testiranje po metodi črne škatle

Testiranje po metodi črne škatle pomeni testiranje zaključene celote brez znanja o tem, kaj je v notranjosti te celote (škatle), torej, na kakšne način je narejena implementacija [2]. Med metode testiranja po metodi črne škatle spadajo:

- Razdeljevanje enakovrednosti – temelji na delitvi vhodnih podatkov v razdelke, na osnovi katerih lahko pridobimo testne primere.
- Analiza mejne vrednosti – teste načrtujemo tako, da vsebujejo značilne mejne vrednosti.
- Testiranje vseh parov – metoda za vsak par vhodnih parametrov testira vse možne diskretne kombinacije teh parametrov.
- Naključno testiranje – metoda temelji na uporabi naključnih, napačnih ali nepravilnih vhodnih podatkov. V primeru, da test ne preživi, lahko zabeležimo napake.
- Matrika sledljivosti – predstavlja dokument, ki preverja, če se zahteve odražajo tudi v podrobni arhitekturi.
- Raziskovalno testiranje – metoda, pri kateri ima tester proste roke pri načrtovanju testov. Od testerja se pričakuje, da bo z učenjem teste neprestano izboljševal.

2.5 PREVZEMANJE PROGRAMSKE OPREME

2.5.1 Osnovni proces prevzemanja

Slika 5: Klasičen proces sprejemanja programske opreme

Na Sliki 5 je prikazan klasičen proces prevzemanja, po katerem vsaka stopnja v procesu vsebuje veliko aktivnosti in odločitev. Čeprav je stopnja *Prevzemi* na sliki predstavljena kot en dogodek, je ta pravzaprav sestavljen iz več aktivnosti prevzemanja. Idealno naj bi se te aktivnosti izvajale samo enkrat, oceno pa bi v celoti izvedel naročnik, vendar bi v praksi to lahko predstavljalo slabe rezultate. Netestirana programska koda vsebuje veliko napak, saj navadno proces potrebuje veliko preizkušanj, preden je stranka zadovoljna s produktom, zato se proces prevzemanja navadno odvija v več ponovitvah. Odločitev o tem, ali bo uporabnik uporabljal produkt, potem ko je ta že narejen, imenujemo odločitev o uporabi. Odločitev je lahko pozitivna ali negativna [19].

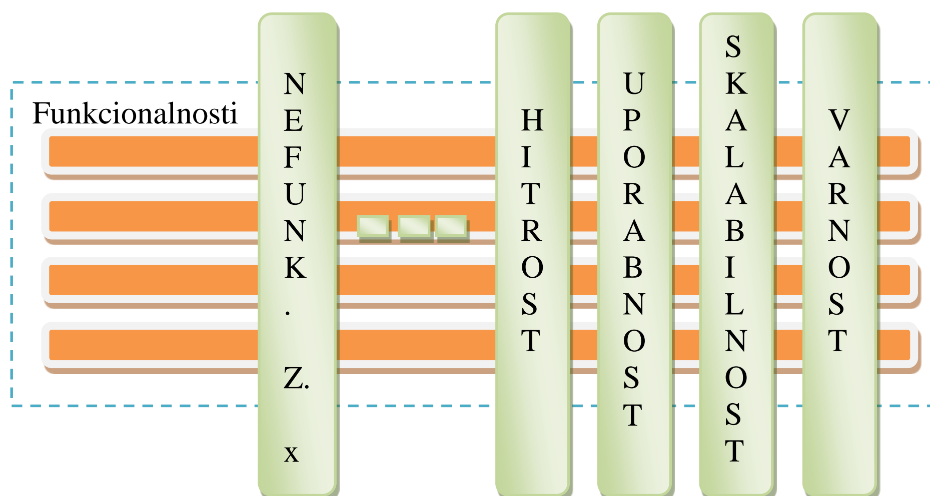
2.5.2 Zahteve programske opreme s stališča sprejemanja

Zahteve programske opreme lahko tipično razdelimo v dve veliki kategoriji:

- funkcionalne zahteve in
- nefunkcionalne zahteve.

Pokrivanje funkcionalnih in nefunkcionalnih zahtev je prikazano na Sliki 6.

Funkcionalne zahteve opisujejo funkcionalnosti, ki so namenjene uporabnikom in administratorjem programskih sistemov. Funkcionalne zahteve direktno odsevajo funkcionalnost, ki jo od programske opreme pričakujemo. Za opisovanje in preverjanje nabora funkcionalnosti lahko uporabimo različne tehnike.



Slika 6: Funkcionalne in nefunkcionalne zahteve

Funkcionalne zahteve opisujejo tisto, kar različni tipi uporabnikov od produkta pričakujejo. Produkt mora delovati tako, da predstavlja pomoč pri njihovem delu. Funkcionalne zahteve lahko določimo direktno iz načrta produkta ali pa jih iz načrta izpeljemo. Pri organiziranju in sporočanju funkcionalnih zahtev so nam v pomoč naslednji načini [19]:

- primeri uporabe,
- poslovni procesi,
- uporabniške zgodbe,
- scenariji,
- seznam zahtev,
- specifikacije protokolov,
- specifikacije funkcij,
- diagrami stanj.

Nefunkcionalne zahteve opisujejo splošne kvalitete ali splošno obnašanje sistema, ki uresničuje različne uporabniške scenarije. V nasprotju s funkcionalnimi zahtevami, ki so od sistema do sistema zelo različne, so nefunkcionalne zahteve večinoma splošne za vse sisteme.

Te zahteve opisujejo karakteristike, ki jih mora celoten sistem podpirati v času delovanja in morajo zadovoljiti potrebe naslednjih zahtev [19]:

- Razpoložljivost – kdaj mora biti sistem razpoložljiv.
- Integriteta podatkov – podatki morajo biti shranjeni in procesirani na tak način, da jih lahko pridobimo brez bojazni, da bi spremenili prejšnje vrednosti.
- Varnost – sistem ne sme fizično ali emocionalno škodovati uporabniku ali lastniku.
- Obnova sistema – v primeru, da sistem pade, se mora ponovno zagnati v enakem stanju, kot se je nahajal pred padcem.
- Dostopnost – uporabniki s posebnimi potrebami morajo imeti možnost uporabe sistema.
- Podpora – obstajati mora služba za pomoč uporabnikom.
- Zanesljivost – sistem mora delovati dobro v vseh situacijah.
- Robustnost – sistem mora kljub zlorabam izvrševati zahtevane funkcije pravilno in konstantno.
- Uporabnost – sistem mora biti preprost za uporabo.
- Varnost – sistem mora biti narejen tako, da je zaščiten pred nezaželenimi vdori.
- Skalabilnost – sistem mora imeti možnost, da zveča ali zmanjša kapacitete, glede na število uporabnikov in zagnanih transakcij.
- Hitrost – kakšne hitrostne kazalnike mora sistem zadovoljiti.
- Namestitvenost – produkt mora imeti možnost enostavne namestitve.
- Skladnost – sistem mora biti narejen tako, da je skladen z operacijskim sistemom in morebitnimi zunanjimi komponentami.

Na celotno ceno produkta pa vplivajo še:

- Testiranje – kako in v katerih korakih bo sistem testiran.
- Prenosljivost – koliko truda je potrebnega, da produkt prestavimo na drug operacijski sistem ali napravo.
- Vzdrževanje – ali je sistem preprost za vzdrževanje.
- Lokalizacija – ali lahko produkt preprosto prevedemo v drug nacionalni jezik.
- Ponovna uporabnost – ali lahko programsko kodo produkta uporabimo v drugih produktih.
- Razširljivost – ali lahko na preprost način produkt nadgradimo.
- Nastavljivost – ali ima sistem možnost nastavljanja različnih parametrov.

Vse naštete nefunkcionalne zahteve niso univerzalne in se ne dotikajo vseh sistemov. V procesu načrtovanja produkta je potrebno določiti, katere nefunkcionalne zahteve so pomembne in katere ne. Tiste, ki so pomembne, je potrebno določiti in jih dodati v testne načrte.

Večina nefunkcionalnih zahtev se dotika večine uporabniških zgodb. Nekatere nefunkcionalne zahteve lahko vsaj delno opišemo z funkcionalnimi termini. Pri varnosti

sistema lahko določimo, da določena uporabniška vloga ne bo smela spreminjati določene vrednosti, medtem ko bo druga uporabniška vloga imela možnost spremembe.

Ključ do končnega zadovoljstva lastnikov z naročeno programsko opremo je določitev pomembnosti posamezne nefunkcionalne zahteve. Varnost je denimo pri internetnih bančnih aplikacijah zelo pomembna, saj lahko varnostna napaka oškoduje tako uporabnika kot banko, ki je lastnica sistema. Pri preprostih namiznih igrah, kjer uporabnik ne uporablja zunanje povezave in gre samo za interakcijo med uporabnikom in namizjem, pa varnost ni tako pomembna.

3 PRIMERJAVA SPREJEMNIH TESTOV IN TESTOV ENOT

3.1 ŽIVLJENJSKI CIKEL TESTOV

Posamezni testi morajo, ne glede na njihovo sestavo ali način izvajanja, iti skozi različna življenjska stanja [19]:

- Sestava: test sestavimo tako, da rešuje določen specifičen problem in zadovolji izbran cilj.
- Določitev in arhitektura: test napišemo v obliki podrobnih korakov, ki določajo tisto, kar mora biti narejeno. Določeno pa mora biti tudi, kdo, kdaj, kje in kako bo to naredil.
- Planiranje izvedbe: izvajanje testov je časovno planirano v okviru izbranega časovnega okvirja in z razpoložljivimi viri.
- Izvajanje: test se izvede v testiranem sistemu.
- Ocena rezultatov: dobljene rezultate primerjamo s pričakovanimi rezultati.
- Poročanje: združitev rezultatov in poročanje nadrejenim.
- Nadaljnji postopki: glede na stanje testov se odločimo, ali bomo test izvajali še naprej ali pa bomo prijavili napako v sistemu.
- Vzdrževanje: če želimo, da testi ohranjajo svojo zrelost, jih moramo vzdrževati.
- Smrt: test je preživel svojo uporabnost, zato je opuščen in ga ni več potrebno vzdrževati.

V prvem stanju – sestava testa, postavimo pogoje testiranja glede na različna obnašanja sistema, ki ga testiramo. Test najprej predstavlja mentalni oris, nato pa neposredne zahteve napišemo bolj točno in podrobno. Dokument testa je navadno sestavljen iz testnih pogojev, ki so povezani s posamezno lastnostjo, zahtevo ali uporabniško zgodbo. V tej fazi je test samo konceptualno ogrodje pripravljeno za vstop v nadaljnje faze.

V stanju določitve in arhitekture testa izvedemo transformacijo konceptualnega ogrodja v konkretna dejanja. Naloge v tem ciklu zajemajo tudi dejanja organiziranja testov na tak način, da si koraki izvajanja sledijo v logičnem zaporedju. Fazo lahko izvedemo skoraj sočasno z izvedbo testa ali pa tudi prej.

Ko je testni primer identificiran in avtoriziran, sledi korak časovnega planiranja izvedbe testov. Urnik izvajanja navadno zajema frekvenco izvajanj ali točno določen datum in čas, ko bomo test izvedli. V fazi določimo tudi primerne mehanizme proženja, beleženja in izvajanja testov.

Tako pripravljen test nato izvedemo. Če so testi dinamični, izvajanje pravzaprav pomeni poganjanje testov v testiranem sistemu. Izvajanje statičnih testov pomeni preverjanje dejstev, ki opisujejo testiran sistem. V tem primeru kode ne izvajamo. Teste lahko izvajamo ročno ali pa s pomočjo orodja, ki omogoča avtomatsko poganjanje testov.

Status izvajanja testov lahko preverjamo sproti v teku poganjanja testov ali pa na koncu, ko so se vsi testi že izvedli. Glede na trajanje poganjanja testov in orodja, ki jih uporabljamo, izberemo način pregledovanja testov. Glede na rezultate, ki nam povejo, ali je test preživel ali padel, se odločimo, ali lahko testiran subjekt sprejmemo ali ne.

Poročanje o izvedbi testiranja predstavlja naslednje stanje v ciklu testiranja. Poglavitna stvar pri poročanju o testih je razumljivost poročila. Poročila morajo biti napisana tako, da lastniki točno razumejo, v kakšnem stanju se nahaja projekt. Poročila pa zajemajo poleg točnega stanja testov tudi količino preostalega dela, ki je potrebna za dokončanje.

Glavni razlog izvajanja testov je pravzaprav učenje o kvaliteti produkta, ki nam je v pomoč pri sprejemanju pomembnih odločitev o tem, ali je produkt pripravljen za uporabo ali potrebuje še dodaten razvoj in testiranje. Bistvo procesa sprejemanja produkta je pravzaprav v tem, da preverimo, če je produkt pripravljen za prenos oz. prodajo ali pa potrebuje še dodatne popravke.

V procesu razvoja bomo nekatere teste poganjali samo enkrat, večinoma pa bomo teste izvajali konstantno, uporabljali pa jih bomo tudi potem, ko bo prva različica produkta že na trgu. Testi, ki jih bomo izvajali večkrat, navadno potrebujejo določeno mero vzdrževanja. Kadar v sistemu ali produktu naredimo večje spremembe, moramo teste, ki so namenjeni ročnemu izvajanju, posodobiti.

Vsak test v določenem obdobju preživi svoj življenjski status in tako ni več uporaben. Razlogov za to je več. Lahko da smo funkcionalnost, ki jo test testira, že odstranili iz sistema ali pa smo določeno funkcionalnost dovolj dobro pokrili z drugimi testi. V tej fazi test preide v končno stanje oz. smrt in ga ne uporabljamo več.

3.2 SPREJEMNI TESTI

Sprejemni test je način preverjanja programske opreme, ki nam pove, če se je sistem pod različnimi pogoji obnašal tako, kot smo od njega pričakovali [1]. Sprejemni testi so testi, ki testirajo celoten sistem. Teste običajno pišejo naročniki ali zunanje testne skupine [29].
Glavni cilj sprejemnih testov je način prikaza delovanja sistema končnemu uporabniku. S pomočjo sprejemnih testov uporabnika prepričamo, da sistem deluje tako, kot je zahteval oziroma pričakoval.

Sprejemni testi temeljijo na testiranju po metodi črne škatle in so namenjeni predvsem preverjanju obnašanja sistema.

Tipičen problem razvoja programske opreme je v tem, da stranka navadno nima točne predstave o tem, kakšen proces naj bi sistem točno podpiral. S pomočjo pisanja uporabniških zgodb in sprejemnih testov, ki stojijo za uporabniškimi zgodbami, izboljšamo komunikacijo med razvojno ekipo in naročnikom. Na tak način se poveča razumljivost osnove problema in posredno se poveča tudi kvaliteta aplikacije. Sprejemni testi so v pomoč pri komunikaciji znotraj razvojne ekipe, saj določajo tipične besede ali jezik področja, ki se ga uporablja v razpravah.

Sprejemne teste lahko vzamemo kot popoln odločitveni kriterij, ki nam pove, kdaj je del programske opreme, s katerim je realizirana neka zgodba, končan. Padec testa sprejemanja pomeni, da zgodba ni razvita na tak način, da bi bila naročniku sprejemljiva. Enako prakso lahko uporabimo pri razvoju na mikro ravni, kjer testiramo posamezne enote programske opreme. Kadar razvijalec najprej napiše test enote in šele potem programsko kodo, ki test realizira, lahko napiše samo toliko kode, da test preživi. Tako točno ve, kdaj je delo končano.

Glavni cilj pri pisanju testov je sestava učinkovite specifikacije zahtev. Sprejemne teste pišemo v jeziku, ki je razumljiv tako tehnični osebi kot naročniku – opisnem jeziku, saj od naročnika ne moremo pričakovati, da bo pisal programsko kodo [31]. Na ta način testi zajemajo bistvo poslovnih pravil in procesov. Bistvo pisanja razumljivih testov je v tem, da lahko tisti, ki berejo teste, lažje razumejo probleme in bolj točno vidijo celotno sliko problema. Testi morajo biti točni in popolni, saj predstavljajo osnovo za specifikacije programske opreme [1].

Ker se testi uporabljajo kot osnovna definicija za razvoj programske opreme, morajo biti ti predvsem preprosti za pisanje in urejanje. Posebna pazljivost je potrebna pri točnem izražanju poslovnih zahtev, ki jih programska oprema uresničuje. Testov pa ne bodo uporabljali samo programerji, zato mora biti jezik stranki hitro razumljiv. Iz strankinega stališča je najbolj uporaben naravni jezik, saj je zelo prilagodljiv in ekspresiven, vendar pa ima uporaba naravnega jezika določene pomanjkljivosti [8]:

- Naravi jezik je preveč nerazumljiv in precizen za dobro definicijo testov.
- Naravni jezik je preveč dolgovezen, saj za izražanje preprostih zadev potrebujemo veliko besed.
- Naravni pisni jezik uporablja kompleksno slovnico, ki je nepotrebna za testiranje.

Za pisanje sprejemnih testov potrebujemo jezik, ki bo definiral zahteve in bo združeval lagodnost in prilagodljivost naravnega jezika in obenem združil strukturo in preciznost programskega jezika.

Ker so stranke navadno iz različnih področij, je potrebno z vsako stranko delno preoblikovati jezik. Splošno je najlaže začeti s preprostim jezikom in mu prepustiti, da se s časoma razvija. Pri tem stranki ne smemo dati že razvitega jezika, vendar moramo pustiti, da jezik skupaj z razvojno ekipo gradi sama.

Jezik mora biti [8]:

- dovolj preprost, da ga zlahka razumemo,
- dovolj generičen, da zajame zahteve in
- dovolj abstrakten, da ga lahko vzdržujemo.

Dovolj veliko abstrakcijo dosežemo z uporabo jezika, ki ga uporablja stranka v svojem procesu. Jezik ne sme uporabljati podrobnosti implementacije in se mora osredotočiti predvsem na poslovne zahteve.

Preprost primer sprejemnega testa:

- Odpri: stran za vstop
- Vnesi uporabniško ime: Terpinc
- Vnesi geslo: terpinc15
- Pritisni: Vstopi
- Preveri stran: Ali je bil vstop uspešen?

Primer podrobnega sprejemnega testa je prikazan v Tabeli 1.

Primerjava vrednosti absolutnih podatkov o občinah			
Vrsta podatka	Vrednost podatka referenčna občina	Vrednost podatka izbrana občina	Absolutna razlika podatkov?
Površina km2	146	151	5
Število prebivalcev	22667	54188	31521
Število moških	11160	26631	15471
Število žensk	11507	27557	16050
Naravni prirast	82	213	131
Skupni prirast	235	652	417

Tabela 1: Primer sprejemnega testa

Sprejemni testi morajo biti preprosti za branje in lahko razumljivi nekomu, ki sicer nima tehničnega znanja, vendar ima dokaj poglobljeno znanje o poslovnem problemu, ki ga aplikacija rešuje. Slovnica in struktura testov morata biti zelo preprosti, vsaka vrstica v testu pa predstavlja posamezno akcijo ali preverjanje.

Testi so navadno sestavljeni iz nabora preprostih ključnih besed, ki jih lahko ponovno uporabimo in kombiniramo z različnimi parametri ter tako dobimo nove teste. Prav tako

morajo biti testi ločeni od implementacije ter tako omogočati, da niso občutljivi za določene spremembe na uporabniškem vmesniku.

Stranke so navadno vajene vloge analitika in poznajo način pisanja podrobnih dokumentov. Ti dokumenti so navadno dobra začetna točka za prepoznavanje potencialnih sprejemnih testov, saj vsebujejo kriterije, ki definirajo zgodbo. Ti kriteriji so za razvijalca navadno zelo pomembni, saj posredno opisujejo dela, ki morajo biti opravljena.

Sprejemni test mora biti razumljiv in lahko berljiv, saj je njegov namen ta, da je enkrat napisan, vendar večkrat prebran. V življenjskem ciklu aplikacije dokumentacija potuje med več osebami z različnimi vlogami. Vse osebe, ki se srečajo s testom, ga morajo hitro in točno razumeti, zato mora biti berljivost testov glavno gonilo pri njihovi sestavi. Za dobro berljivost pa je pomembna predvsem čista sestava testa. Testi so lahko glede na tip in kompleksnost problema predstavljeni v različnih oblikah (tekstovni, tabelarični, uporabniške zgodbe, grafični ...). V idealnem primeru bi bil test napisan v univerzalnem jeziku, iz katerega bi lahko bralec izbral pogled, s pomočjo katerega bi lahko test prebral v izbranem jeziku [14].

V osnovi morajo biti sprejemni testi kratki, razumljivi in preprosto berljivi, vendar pa morajo imeti še druge pomembne lastnosti [17]:

- lastnik testov je stranka;
- teste napišejo skupaj stranka, razvijalec in tester;
- bistvo testov je, da opisujejo želeno delo in ne kako oz. na kakšen način bomo delo izvedli;
- testi so napisani v jeziku, ki je blizu poslovnemu problemu;
- testi so jedrnati, precizni in nedvoumni.

Pomembna lastnost sprejemnih testov je tudi možnost preprostega iskanja. Posamezna aplikacija ima lahko tudi več sto sprejemnih testov, zato je pomembno da teste uredimo in opremimo z metapodatki tako, da jih bo lahko nekdo drug preprosto našel. Iskanje mora omogočati naslednje tipe iskanj [14]:

- po imenu testa,
- po metapodatkih,
- po funkcijah, ki jih testi preverjajo.

Teste se v procesu razvoja nenehno poganja, zato je pomembno, da je okolje, ki omogoča izvajanje testov, prilagojeno tako, da je poganjanje testov preprosto. Prav tako mora okolje dopuščati možnost poganjanja več testov in podpirati hkratno delo več testerjev. Pomembna lastnost testov, ki jih poganjamo v ciklu razvoja, je tudi možnost razhroščevanja testov. Razhroščevanje mora biti integrirano in enostavno.

Za pisanje testov potrebujemo učinkovito razvojno okolje. Moderna orodja za pisanje

programske kode so se razvila do te mere, da omogočajo razvijalcem zelo učinkovito pisanje kode. Okolja za pisanje sprejemnih testov v tej meri še niso napredovala in so zelo okorna. Večinoma teste pišemo s pomočjo preprostih urejevalnikov teksta, poganjamo pa jih preko ukazne vrstice. Velik napredek v tej smeri predstavlja orodje Fitnesse [29], saj predstavlja okolje na osnovi Wikipedije, ki omogoča lažje pisanje in poganjanje testov. V primeru, da sprejemne teste povežemo, s praksami ekstremnega programiranja pridobimo nove možnosti uporabe sprejemnih testov. Sprejemni testi predstavljajo veliko komunikacijsko pomoč znotraj razvojne ekipe.

3.3 TESTI ENOT

Testiranje enot je praksa ekstremnega programiranja, ki se osredotoča na testiranje najmanjših možnih enot programa. Tako testiranje pravzaprav predstavlja testiranje posameznih enot ali enot povezanih v skupino. Testiranje posameznih enot programske opreme je tako strateško in praktično domena programerja, saj upravljanje in kvaliteta testiranja ne vplivata neposredno na strategije in prakse testiranja enot [25].

Testiranje enot je metoda, pri kateri testiramo posamezne enote programske kode. Enota je najmanjši del programske kode, ki ga lahko testiramo, kar navadno predstavlja funkcijo ali proceduro. Teste enot največkrat napišejo programerji, občasno pa jih pišejo tudi testerji. V idealnem primeru so testi med seboj neodvisni. Za lažje testiranje se uporabljajo metode zamenjave, kjer namesto prave programske kode uporabimo dvojnike.

Testi so strukturni in temeljijo na testiranju po metodi bele škatle, razvijalci pa merijo njihovo popolnost glede na strukturno pokritost.

Kent Beck v definiciji testov te preprosto razdeli na dva tipa [4]:

- programerski testi in
- uporabniški testi.

Teste, ki so bolj osredotočeni na tehnologijo in jih pišejo programerji, imenujemo programerski testi. Za te tipe testov je pomembno, da so napisani v jeziku, ki ga razumejo programerji.

Teste, katerih funkcionalnosti definirajo uporabniki, pa imenujemo uporabniški testi. Na te teste se navado sklicujemo kot na sprejemne teste in morajo biti napisani v jeziku, ki ga razumejo uporabniki [4].

Tim Koomen and Martin Pol definirata testiranje enot kot test, ki ga razvijalec poganja v

laboratorijskem okolju in naj bi pokazal, ali program izpolnjuje spisek zahtev in načrtovalske zahteve [16].

Testi definirajo vhodno domeno tistih enot, ki so vprašljive, in ob tem ne upoštevajo ostalega sistema. Testiranje enot občasno zahteva konstrukcijo dvojnikov funkcionalnosti in kode, ki bo zavržena. Testiranje enot se velikokrat izvaja s pomočjo razhroščevalca.

Testiranje enot torej predstavlja tehnično orientirano testiranje najmanjših enot v sistemu s pomočjo vhodnih in izhodnih parametrov. V osnovi je cilj testiranja enot v tem, da vzamemo majhen del kode, ki je v razvijajočem programu odgovoren za točno določeno funkcionalnost, in pod izbranimi pogoji testiramo njegovo obnašanje. Testiranje enot tako omogoča testiranje posameznih enot programske opreme, ki tipično niso izpostavljene uporabniku. Način testiranja daje razvojnim ekipam hitro povratno informacijo, ki kaže, če razvoj poteka v pravo smer, in omogoča takojšnje manjše popravke.

```
[TestMethod]
        public void TestEnote_VrniAbsolutnoVrednostRazlikeNegativnaStevila()
        {
            PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik instancaDvojnika = new
PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik();
            int vrednost1 = -12;
            int vrednost2 = -15;
            int pricakovanaVrednost = 3;
            int pravaVrednost =
instancaDvojnika.VrniAbsolutnoVrednostRazlike(vrednost1, vrednost2);
            Assert.AreEqual(pricakovanaVrednost, pravaVrednost);
        }
```

Slika 7: Primer testa enote – koda

Slika 7 prikazuje tipičen test enote napisan v programskem jeziku C# s pomočjo integriranega Microsoftovega ogrodja za testiranje. Test preverja pravilno delovanje metode, ki vrača absolutno vrednost razlike dveh števil. Metodi vnesemo dve števili, nato pa preverimo, če sta pričakovana vrednost, ki jo ročno nastavimo, in izračunana vrednost, ki jo vrne metoda, enaki. V primeru, da sta enaki, je test preživel, v nasprotnem primeru pa je padel.

Pri klasičnem testiranju se preverjanje funkcionalnosti navadno izvaja šele potem, ko je bila programska oprema že razvita. Tak način testiranja otežuje popravke kritičnih napak in napačne arhitekture. S pomočjo testiranja enot je delo programerja preverjeno praktično vzporedno z razvojem programske kode, kar omogoča hitre popravke in spremembe v arhitekturi.

Ena izmed pomembnih lastnosti testiranja enot je ta, da omogoča programerju varno

preoblikovanje kode. S pomočjo preoblikovanja kode je koda lažje razumljiva, bolj berljiva in predvsem krajša, saj je glavni pomen preoblikovanja kode odstranjevanje podvojene programske kode. S pomočjo testov lahko programer hitro preveri, če preoblikovana koda še deluje tako, kot je delovala pred preoblikovanjem [34].

Pomembna lastnost testov enot je razmejitev vmesnika od implementacije. Ker imajo nekateri razredi reference na druge razrede, se lahko testiranje posameznega razreda hitro premakne čez svoje meje in zavzame tudi testiranje kode, ki se nahaja v ostalih odvisnih razredih. Nazoren primer take odvisnosti je odvisnost od podatkovne baze, saj tester velikokrat napiše kodo, ki je odvisna od baze podatkov. To pa predstavlja napako, saj test enote praviloma ne sme presegati mej razreda. Preseganje mej razreda spremeni test enote v integracijski test. Praviloma se testi enot kreirajo s pomočjo abstraktnega vmesnika, ki zaobide poizvedovanja v bazo. Namesto odvisnosti potem vmesnik implementiramo v testni dvojnik, ki ga uporabimo namesto odvisnega razreda. Na tak način lahko posamezno enoto testiramo neodvisno od ostalih razredov [20].

Seveda ne moremo od testiranja enot pričakovati, da bo zajelo popolnoma vsako napako v programu, saj testiranje enot zajema samo testiranje funkcionalnosti posamezne enote. Testiranje ne bo zajelo integracijskih napak, širših napak sistema ali napak procesa. Tako kot vse oblike testiranja programske opreme lahko testi enot pokažejo samo prisotnost napak, ne morejo pa prikazati izostanka napak.

V primeru uporabe testov enot je v procesu razvoja potrebna močna disciplina razvojnikov, saj je zelo pomembno, da beležimo vse dogodke o rezultatih in spremembah v izvorni kodi testov. Prav tako je zelo pomembna uporaba sistema za kontrolo različic, saj nam omogoča preprost pregled nad spremembami v programski kodi.

Pomemben del uporabe testov enot je tudi dnevno beleženje posameznih napak v testih. Napake je potrebno nato kar najhitreje odpraviti. V primeru, da takega procesa ne implementiramo v delovni tok ekipe, bo razvita aplikacija sčasoma zgubljala sinhronizacijo s testi enot, kar privede do povečevanja napak in zmanjševanja učinkovitosti testov.

Testiranje se navadno izvaja v avtomatiziranem okolju, lahko pa se ga še vedno izvaja ročno. Ročno testiranje enot je lahko zajeto v dokumentu, kjer so točno opisani postopki (po posameznih korakih), kako testirati posamezne enote. Avtomatizirano testiranje predstavlja učinkovit način, saj samo v primeru avtomatiziranega testiranja enot zajamemo vse prednosti uporabe testov enot. Če želimo realizirati vse prednosti izolacije enot pri uporabi avtomatiziranega testiranja, potem izvajamo testiranje znotraj okolja za testiranje, ki se nahaja izven naravnega okolja. Tak način testiranja tudi hitro razkrije vse odvisnosti med posameznimi enotami.

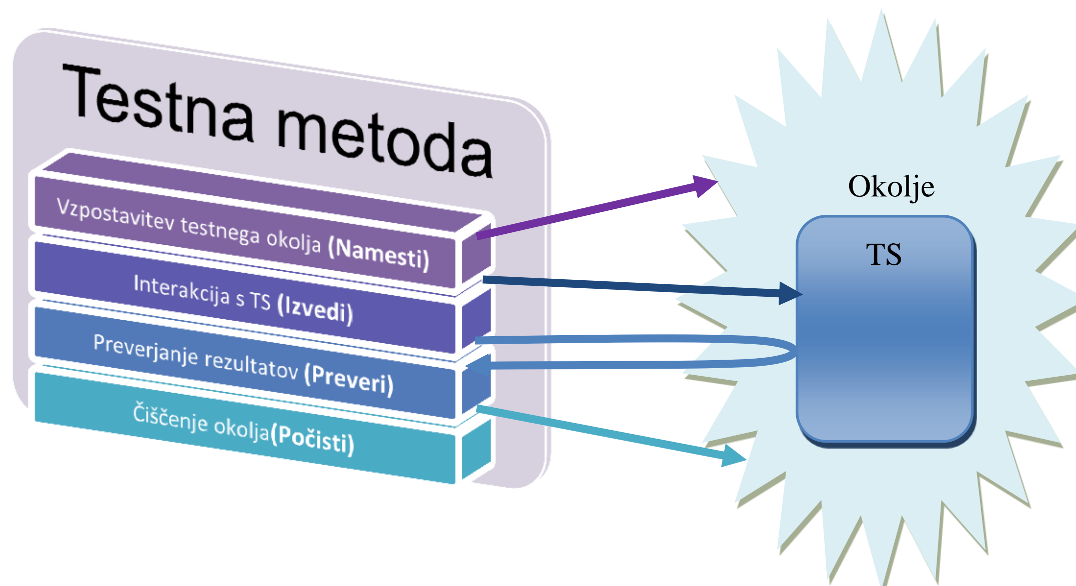
S pomočjo avtomatiziranega testiranja se posamezni dogodki o testu avtomatsko beležijo, tako ima razvijalec pregled nad vsemi testi, ki se nahajajo znotraj avtomatiziranega testnega okolja. S pomočjo prikazov o stanju posameznih testov lahko ugotovi, ali je test preživel ali padel in v primeru padca tudi vzrok padca. Veliko okolij to beleži in shranjuje v dnevniške zapise posameznih testiranj.

Testiranje enot programerjem predstavlja motivacijo za kreiranje programske kode, ki je izolirana in ni vezana na ostale dele programa. Taka praksa predstavlja, skupaj z uporabo primernih razvojnih vzorcev in preoblikovanjem kode, dober način razvoja programske opreme in prednosti za celotno razvojno ekipo.

3.3.1 Testiranje enot v štirih fazah

Proces testiranja posamezne enote je predstavljen na Sliki 8. Proces je razdeljen na štiri faze oz. štiri okvirje [18]:

- Vzpostavitev testnega okolja (ang. test fixture): v prvi fazi vzpostavimo okolje, ki je potrebno, da testirani subjekt (TS) prikaže zahtevano obnašanje. Prav tako v tej fazi vzpostavimo vse stvari, ki jih moramo postaviti, da bomo lahko opazovali dejansko izvajanje (na primer: testnega dvojnika).
- Interakcija s testiranim subjektom: v drugi fazi izvedemo interakcijo s testiranim subjektom in izvedemo izbrano funkcionalnost.
- Preverjanje rezultatov: v tretji fazi izvedemo vse aktivnosti, kjer preverimo, če se načrtovani izid sklada z dobljenim.
- Čiščenje okolja: v četrti fazi počistimo testno okolje in tako vzpostavimo stanje pred izvedbo testne metode.



Slika 8: Testiranje v štirih fazah

3.3.2 Upravljanje odvisnosti

Čeprav je testiranje enot osnovna praksa ekstremnega programiranja, je večino kompleksnih programov težko testirati, saj potrebujemo dovolj veliko stopnjo izolacije posameznih enot. Ključen faktor testiranja enot je enkratno testiranje posamezne funkcionalnosti. Razvijalec mora točno vedeti, kaj testira in kje ležijo problemi. V primeru testiranja posamezne funkcionalnosti mora biti prepričan, da bo obveščen takoj, ko katerikoli od možnih problemov nastopi. Testiranje večjih sklopov pa je problematično, saj kodo testiramo z zunanje strani, tako ni nujno, da primarna koda izpostavi stanje, ki ga potrebujemo za test [4].

Osnovna terminologija testiranja predvideva, da je testirani subjekt (TS) zasnovan tako, da je preprost za testiranje brez odvisnosti od drugih objektov. V tem primeru je testiranje točno določeno in preprosto [18].

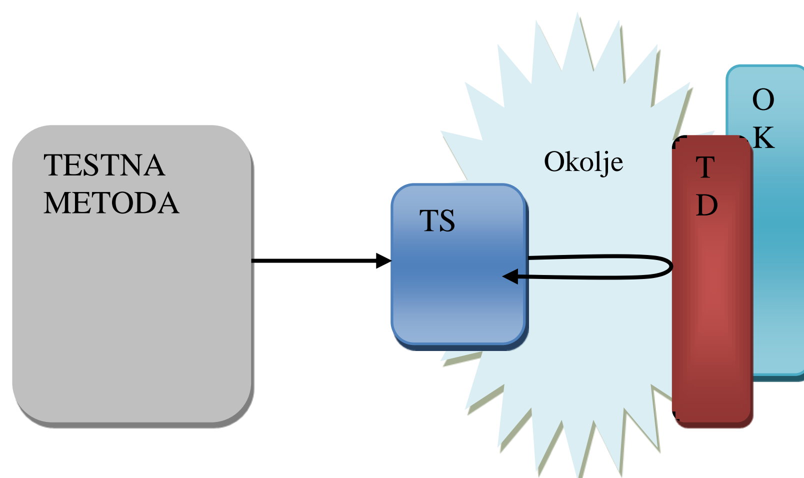
Kadar je testirani subjekt odvisen od ostalih komponent v programu, ga težko uporabimo v testnem okolju. Tak položaj lahko nastane zaradi različnih vzrokov:

- komponente ni na voljo;
- komponenta ne vrača vrednosti, ki jih za test potrebujemo;
- zagon odvisne komponente lahko povzroči neželene učinke.

Pri testiranju posameznih enot kompleksnih programov se tako pojavi problem odvisnosti, ki

jih posamezna enota lahko vsebuje. Odvisnosti onemogočajo izolacijo posamezne enote. Rešitev tega problema predstavlja tehnika uporabe nadomestnih objektov, kjer nadomestimo primarno kodo z lažnimi implementacijami, ki oponašajo vedenje prave kode. Nadomestni objekt je pravzaprav nadomestna implementacija, ki oponaša delovanje primarne kode. Ti objekti so preprosti in nam predstavljajo lažje delo pri testiranju. Pravzaprav je največji poudarek lažnih implementacij (v nasprotju s kompleksnostjo) njihova preprostost. V praksi so se za te zamenjave uveljavile besede navidezni objekti (ang. mocks) ali testna nadomestila (ang. stubs), vendar je terminologija okoli teh objektov zelo nekonsistentna.

Gerard Meszaros poimenuje te objekte testni dvojniki (ang. test doubles), kot splošno ime za vse objekte, ki so za namene testiranja uporabljeni namesto pravih komponent [20]. Tako neformalno standardizira poimenovanje teh objektov. Glavna lastnost testnih dvojnikov je, da njihovo obnašanje ni nujno enako obnašanju odvisnih komponent. Dvojniki morajo le zagotoviti enak vmesnik – API kot prava odvisna komponenta in s tem prepričati testirani subjekt, da je pravi. Slika 9 predstavlja način zamenjave odvisne komponente (OK) s testnim dvojnikom (TD).



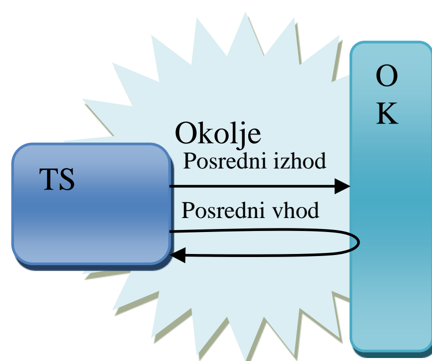
Slika 9: Princip uporabe testnih dvojnikov

3.3.2.1 Posredni vhodi in izhodi

Pokrivanje vseh delov kode je glavni problem testiranja skupinskih razredov. Odvisna komponenta lahko testiranemu subjektu vrača vrednosti ali programske izjeme. V praksi se izkaže, da je določene primere težko ali nemogoče priklicati. To privede do uporabe netestirane kode in produkcijskih hroščev.

Posredni vhodi, ki jih od odvisne komponente prejme testirani subjekt, so lahko nepredvidljivi (npr.: sistemska ura, koledar). Lahko se zgodi, da je odvisna komponenta nedostopna ali pa ne obstaja. Prav tako je potrebno preveriti, če so se določeni stranski učinki, ki jih povzroči

izvajanje testiranega subjekta, zares izvedli. Klici do odvisne komponente večinoma vračajo vrednosti, objekte ali programske izjeme. Ročno preverjanje teh posrednih izhodov je lahko zelo časovno in finančno potratno, prav tako pa nam poruši efektivnost testiranja. Rešitev teh problemov predstavlja vpeljava testnih dvojnikov, ki nam pomagajo testirati posredne vhode in izhode. Slika 10 prikazuje odnose med odvisno komponento in testiranim subjektom.



Slika 10: Posredni izhodi in vhodi iz testiranega subjekta

Test komunicira s programsko kodo preko različnih vmesnikov ali interakcijskih točk. S testerjevega pogleda so lahko te točke namenjene kontroli ali pa opazovanju. Kontrolna točka nadzira način, ki ga test uporabi ob prošnji programske kode za določeno delovanje, kot je npr. spreminjanje stanja spremenljivk v programski kodi [20]. Določene kontrolne točke so namenjene samo za namene testiranja in jih ne smemo uporabiti v produkcijski kodi, ker obidejo preverjanje vhodov ali pa skrajšujejo normalni življenjski cikel testiranega objekta.

Opazovalna točka nadzira način, ki ga test uporabi pri nadzoru testiranega objekta med fazo preverjanja rezultatov. S pomočjo opazovalnih točk lahko dostopamo do stanja testiranega subjekta ali odvisne komponente po izvršitvi testa. Prav tako pa jo lahko uporabimo za pregledovanje interakcij med testiranim subjektom in komponento [20].

Klici odvisnih komponent lahko vračajo vrednosti, objekte ali pa podajajo programske izjeme. Večino v testirani subjekt speljanih izvedbenih poti se ukvarja z vrnjenimi vrednostmi ter izjemami. Če teh poti ne testiramo, pridemo do netestirane kode, zato je pokrivanje teh poti največji izziv pri testiranju kode in zagotavljanju stabilne produkcijske kode.

Testiranje posrednih vhodov je nekoliko lažje kot testiranje posrednih izhodov, saj so tehnike, ki jih uporabljamo za testiranje izhodov, narejene na osnovi tehnik za testiranje vhodov. Če želimo testirati testirani subjekt s posrednimi vhodi, moramo tako imeti kontrolo nad odvisno komponento, da nam vrne katerokoli možno vrednost. To kaže možnost uporabe primerne kontrolne točke. Primeri posrednih vhodov, ki jih želimo napeljati preko kontrolne točke

vsebujejo:

- vrnjene vrednosti metod/funkcij,
- vrednosti argumentov, ki jih lahko posodobimo in
- programske izjeme, ki jih lahko metode vrnejo.

Pogosto test komunicira z odvisno komponento tako, da ji nastavi vrednosti, ki jih bo le-ta vračala. Vzemimo za primer komponento, katere glavni namen je preskrba podatkov iz baze. Preko zadnjih vrat vnesemo specifične vrednosti v bazo podatkov, te pa nam potem nastavijo komponento tako, da se bo odzivala na želen način. V tem primeru je baza podatkov naša kontrolna točka.

V večini primerov tak pristop ni mogoč ali pa ni praktičen, saj prave komponente ne moremo uporabiti. Za to navadno obstaja več razlogov [20]:

- s pravo komponento ne moremo manipulirati na tak način, da bi nam proizvajala želeni posredni vhod. Samo notranja napaka komponente na pravi programski opremi bi nam proizvajala želeni vhod v testirani subjekt;
- pravo komponento bi lahko zmanipulirali tako, da bi se pojavil pravi vhod, vendar bi bilo to predrago;
- pravo komponento bi lahko zmanipulirali tako, da bi se pojavil pravi vhod, vendar bi tako početje prineslo neželene stranske učinke;
- prava komponenta še ni na voljo.

Če za kontrolno točko ne moremo uporabiti prave komponente, potem moramo le-to zamenjati s tako, ki jo lahko kontroliramo. Zamenjava je možna na več načinov s pomočjo testnega dvojnika. Najbolj pogost način je uporabna testnega nadomestila (ang. test stub), ki ima definiran set vrednosti, katere lahko vrne. Med izvajanjem testiranega subjekta, testno nadomestilo prejme klice in vrne nastavljene vrednosti.

Pri normalni uporabi se ob izvedbi testa izvajajo naravne interakcije testiranega subjekta z odvisnimi komponentami. Za testiranje posrednih izhodov moramo imeti možnost pregleda klicev, ki jih testirani subjekt pošilja odvisni komponenti. Prav tako pa moramo imeti pregled nad napredkom testiranja za kontrolno točko in navsezadnje tudi možnost kontrole vrnjenih vrednosti.

3.3.2.2 Tehnike zamenjav odvisnosti

Kadar izvajamo testiranje enot v skladu s pravili, moramo enoto osvoboditi vseh odvisnosti. Izolacija pa ni preprosta, kadar so odvisnosti napisane v obliki razredov, katerih imena niso avtomatsko generirana, ampak so jih ročno poimenovali razvijalci. Tehnika *injekcije odvisnosti* (ang. dependency injection) predstavlja način, ki nam omogoča spajanje testiranih subjektov in odvisnosti, ki bi bile pretrgane med avtomatiziranim testiranjem.

Ročnemu poimenovanju imen razredov, ki predstavljajo odvisnost v kodi, se moramo izogniti. To naredimo tako, da odjemalcu ali sistemski konfiguraciji priskrbimo drugačne načine, ki bodo testiranemu subjektu povedali, katere objekte naj izvede za posamezno odvisnost. Testni subjekt načrtujemo tako, da odvisnosti plasiramo skozi vhodna vrata, kar pomeni, da specificiramo odvisnosti tako, da postanejo del aplikacijskega vmesnika testiranega subjekta. Odvisnost lahko vključimo kot argument s klicem metode, lahko jo vključimo v konstruktor objekta ali pa jo definiramo kot lastnost objekta.

Tehnika »injekcije odvisnosti« predstavlja dobro prakso za komuniciranje z razredom, kadar smo v začetni fazi načrtovanja programske kode. Testne dvojnike namreč vstavimo v fazi načrtovanja. Kadar pa nimamo popolne kontrole nad testirano kodo, pa raje uporabimo tehniko iskanja odvisnosti.

Tehnika iskanja odvisnosti predstavlja način, ki omogoča normalno združevanje testiranega subjekta in vseh njegovih odvisnosti med avtomatiziranim testiranjem. Prav tako kot pri tehniki injekcije odvisnosti se tudi pri tej tehniki izognemo ročnemu kodiranju imen razredov. Statično povezovanje nam namreč močno omeji opcije, ki določajo način konfiguracije med izvajanjem programske opreme. Pri tehniki ročno kodiramo ime komponentnega posrednika, ki nam vrne pripravljen objekt. Komponenti posrednik nam pripravi načine za programsko opremo odjemalca ali managerja sistemske konfiguracije, ki pove testiranemu subjektu, katere objekte naj uporabi kot odgovor na zahteve komponent.

Tehniko je lažje vključiti v obstoječo programsko opremo, saj vpliva le na tista mesta, kjer se konstrukcija objekta pravzaprav izvede, zato ni potrebno spreminjati vsakega posrednega objekta ali metode, kot bi naredili pri tehniki »injekcije odvisnosti«. Tehniko prav tako lažje vključimo v obstoječe teste.

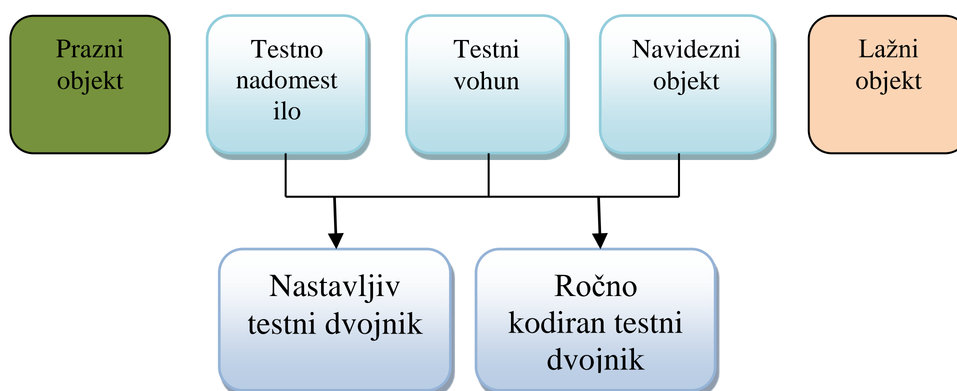
3.3.3 Testni dvojniki

Terminologija okoli testnih dvojnikov ni formalno definirana, zato je zelo nejasna in nekonsistentna. Mark Seemann zbere različna poimenovanja in različne tipe formalizira. Med testne dvojnike razvrsti naslednje objekte [26]:

- prazni objekti (ang. dummies),
- lažni objekti (ang. fakes),
- testni vohuni (ang. spies),
- testna nadomestila (ang. stubs),
- navidezni objekti (ang. mocks).

Vedenje testnih dvojnikov je odvisno od implemetacije in vrste dvojnika. Na Sliki 11 sta predstavljeni dve kategoriji: ročno kodirani testni dvojniki in nastavljivi testni dvojniki.

Nastavimo ali ročno kodiramo lahko le testna nadomestila, testne vohune in navidezne objekte. Lažni objekti so preproste implementacije, ki jih test ne kontrolira, prazni objekti pa nimajo implementacije.



Slika 11: Nastavljiv in ročno kodiran testni dvojnik

Praznih in lažnih objektov ni potrebno nastaviti, zato jih imenujemo nenastavljivi testni dvojniki. Praznih objektov prejemnik nikoli ne uporablja, zato nimajo prave implementacije, medtem ko lažni objekti potrebujejo pravo implementacijo, vendar je ta zelo preprosta in predstavlja zelo poenostavljano obliko pravega objekta. Nenastavljivi testni dvojnik samo namestimo, konfiguracija pa ni potrebna.

V primerih, ko v posameznem testu uporabljamo specifični testni dvojnik, je velikokrat preprosteje ročno zakodirati testni dvojnik na tak način, da nam vrača želene vrednosti ali specifične klice metod. Dvojnik, ki ga zakodiramo za posamezno uporabo, imenujemo ročno kodirani testni dvojnik.

Nastavljivi testni dvojnik uporabljamo, kadar potrebujemo dvojnik v več različnih testih. Mnoge različice testnih okolij vsebujejo orodja za generiranje testnih dvojnikov, lahko pa jih naredimo tudi ročno.

Prazni objekti so najbolj preprosti in primitivni tipi testnih dvojnikov. Objekte večinoma uporabljamo samo, kadar jih rabimo kot vrednosti parametrov. Kot prazne objekte bi lahko definirali nične objekte, vendar so pravi nični objekti izpeljave vmesnikov ali osnovnih razredov brez kakršnekoli implementacije, prazni objekti pa imajo toliko implementacije, da lahko hranijo vrednosti parametrov.

Testno nadomestilo uporabimo takrat, kadar naletimo na določen del netestirane kode, ki jo povzroči nezmožnost kontrole posrednih vhodov v subjekt. Nadomestilo lahko uporabimo kot kontrolno točko, ki nam omogoča kontrolo obnašanja subjekta pri različnih vhodih, vendar brez potrebe preverjanja posrednih izhodov. Testno nadomestilo lahko uporabimo tudi za metodo injekcije vrednosti, ki nam omogoča preiti določeno točko programske opreme, kjer testirani subjekt kliče dele programske kode, ki ni dosegljiva.

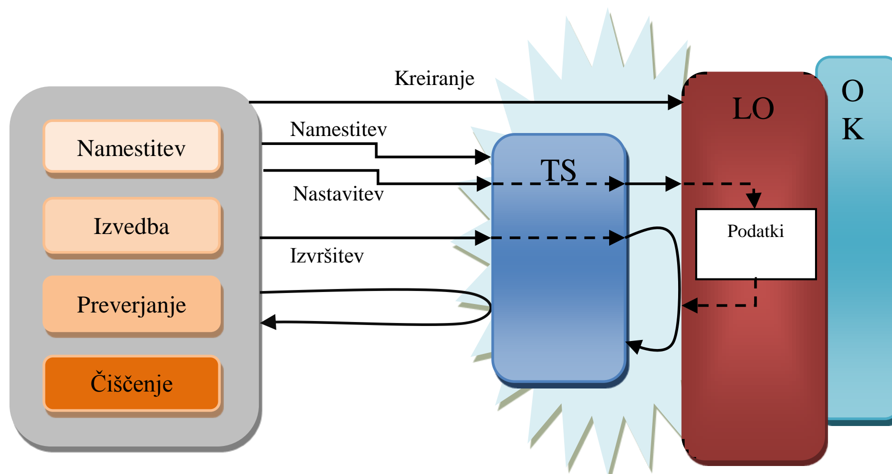
Najbolj standarden pristop konfiguracije testnega nadomestila je s pomočjo nabora vrednosti, ki naj jih nadomestilo vrne iz svojih funkcij in potem posreduje testiranemu subjektu. Med izvedbo testa testno nadomestilo prestreže klic testiranega subjekta, kot odziv pa vrne vrednost, ki je definirana v naboru vrednosti, ki jih testno nadomestilo vrača.

Navidezni objekt je zelo zmogljiva implementacija preverjanja obnašanja, ki ga uporabimo v izogib podvajanju testne kode v istih testih. Pri navideznem objektu se aktivnost preverjanja posrednih izhodov izvaja direktno na testnem dvojniku.

Navidezni objekt je dinamičen objekt, ki ga kreiramo s pomočjo knjižnice za podporo navideznim objektom. Navidezne objekte lahko kreiramo tudi ročno, vendar razvijalec nikoli ne vidi prave implementacijske kode navideznega objekta. Te objekte lahko nastavimo tako, da vračajo želene vrednosti glede na klice ostalih komponent ali pa, da se obnašajo bodisi kot prazni objekti, testna nadomestila ali testni vohuni.

Testni vohun je podoben testnemu nadomestilu, vendar ostalim komponentam vrača poleg instance objekta tudi njegovo stanje. Vohun si bo zapomnil, kateri objekti so bili klicani, ter tako omogočil testnim enotam možnost preverjanja pravilnosti klicanih objektov.

Kadar so interakcije z odvisnimi komponentami potrebne, stranski učinki teh interakcij (kot bi jih izvajala prava komponenta) pa so nepotrebni ali pa celo škodljivi, se srečujemo z lažnimi objekti. To so pravzaprav zelo preproste implementacije pravega objekta, s preprostimi funkcionalnostmi in brez tistih stranskih učinkov, ki jih nočemo imeti.



Slika 12: Interakcija z lažnim objektom

Objekt deluje torej tako, da lahko uporablja implementacijo istih funkcionalnosti kot prava odvisna komponenta, od katere je testirani subjekt odvisen. Subjektu ukažemo, naj uporablja lažni objekt namesto prave komponente. Slika 12 prikazuje uporabo lažnega objekta kot nadomestilo za odvisno komponento.

Lažni objekt uporabljamo predvsem v primerih, kjer drugih komponent nimamo na voljo ali pa te delujejo zelo počasi. Prav tako pa lažni objekt uporabljamo, kadar je lažje narediti preprosto implementacijo, kot pa uporabiti testno nadomestilo ali navidezni objekt [20].

3.4 PODOBNOSTI IN RAZLIKE MED SPREJEMNI TESTI IN TESTI ENOT

Testi enot in sprejemni testi so si v mnogih pogledih podobni. V okviru magistrske naloge smo raziskali podobnosti in razlike med obema vrstama testov. Te smo raziskali glede na izbrane lastnosti posameznih testov. Rezultati so predstavljeni v Tabeli 2.

Vrsta testa / Lastnost	Test enote	Sprejemni test
Kdo piše teste?	Razvijalec.	Naročnik ali naročnik ob pomoči razvijalca.
Ali lahko teste poganjamo samostojno?	Da.	Za poganjanje potrebujemo povezavo s testiranim subjektom.
Kakšno je testno okolje?	Preprosto.	Kompleksno.
Kakšen jezik uporabljamo za pisanje testov?	Programski jezik.	Opisni, domenski jezik.
Kakšna je razvitost okolij za poganjanje avtomatiziranih testov?	Obstaja več dobro izpopolnjenih okolij.	Obstaja nekaj okolij, ki so še v začetni fazi razvoja.
Ali je mogoča integracija testov v razvojna orodja?	Dobra, teste je možno poganjati neposredno iz razvojnega okolja.	Orodja še ne omogočajo integracije.
Kakšna je hitrost izvajanja avtomatiziranih testov?	Avtomatizirani testi se izvedejo sorazmerno hitro.	Izvajanje avtomatiziranega testiranja je počasno.
Ali testi omogočajo tiho poročanje?	Orodja nas obvestijo samo takrat, ko test pade.	Testna okolja še ne omogočajo tihega poročanja.
Kako kompleksne so odvisnosti v okolju?	Odvisnosti v okolju so preproste.	Odvisnosti v okolju so kompleksne.

Tabela 2: Primerjava testov enot in sprejemnih testov

Ker so si testi dokaj podobni je smiselno podrobneje pogledati dejanske razlike. Največja razlika je zagotovo v tem, da sprejemne teste v osnovi piše uporabnik. Testi na ta način predstavljajo osnovno komunikacijsko vodilo med naročnikom in razvojno ekipo. Temu primeren mora biti tudi jezik pisanja testov. Ta mora biti razumljiv tako stranki kot uporabniku.

Sprejemne teste torej praviloma piše naročnik v opisnem jeziku, medtem ko teste enot praviloma piše programer. Če na teste pogledamo z vidika tistega, ki teste piše, opazimo, da teste enot navadno piše programer v programerskem jeziku, medtem ko sprejemne teste v osnovi piše naročnik v opisnem domenskem jeziku. Sprejemni testi so torej napisani v opisnem jeziku, ki je blizu naročniku. Kadar želimo sprejemne teste povezati s programsko kodo, je potrebno vzpostaviti okolje, ki povezuje opisni in programerski jezik [24]. Naročnik ne pozna programerskega jezika, zato je naloga programerja, da to okolje napiše. Pisanje sprejemnih testov bi tako lahko izvajal naročnik sam, vendar mora programer za avtomatsko poganjanje sprejemnih testov napisati še povezavo sprejemnih testov s programom. Programer praviloma tudi bolje pozna logiko in tehnologijo računalniških programov, zato je sodelovanje programerja pri pisanju opisnih testov priporočljivo. V primeru, da gledamo na teste z vidika tiste osebe, ki teste piše, potem vidimo, da pri testih enot nastopa programer, medtem ko pri sprejemnih testih nastopata programer in naročnik skupaj.

Testi enot so narejeni v istem programskem jeziku kot aplikacija, ki jo testiramo, kar pomeni, da lahko teste kličemo direktno iz kode, ki jo testiramo. Test enote kreira objekt razreda, ki ga testiramo, nato pa sproži metode in preveri, če so odzivi pravi. Sprejemni testi sami po sebi ne delujejo samostojno, vendar so samo specifikacije o tem, kaj je potrebno narediti. Če želimo, da testi komunicirajo s testiranim subjektom, moramo za tak način delovanja testom dodati nekaj programske kode. Najlažja pot je direktno povezovanje ključne besede v svojo metodo. Za pomoč pri povezovanju lahko zgradimo neke vrste prevajalnik, ki skrbi za povezovanje ključnih besed v metode. Za pomoč pri povezovanju obstaja nekaj generičnih okolij, primer takih okolij sta okolji FIT in Fitnesse.

Razvijalci za razvoj programske kode in testov enot uporabljajo že pripravljena razvojna orodja. Naročniki navadno pišejo sprejemne teste brez uporabe določenega posebnega orodja, saj bi učenje uporabe orodja vzelo preveč časa. Naročniki morajo zato uporabljati orodja, ki so jim domača in lahko dostopna, kot denimo Word, Excel ali Wikipedia. Trenutno na trgu ni orodja, ki bi omogočalo preprosto povezovanje Worda ali Excela s testiranim subjektom. Najbolj razširjeni sta okolji FIT in Fitnesse, ki za vnašanje in izvajanje testov uporabljata Wikijev vmesnik. Okolji sta dokaj preprosti, vendar ne omogočata integracije v najbolj razširjena razvijalska orodja.

Glavni namen testov enot je preverjanje, če enota deluje v skladu s pričakovanji razvijalca. Ker imajo razvijalci vpogled tako v testno kot v razvojno kodo, lahko preprosto preverijo

pravilnost izvajanja testov. Vsa preverjanja, ki se vršijo v kodi, morajo biti zajeta v stavkih preverjanj. Zaradi tega testi tečejo »tiho«, kar pomeni, da testi med izvajanjem ne izpisujejo podatkov na izhod, oziroma izpisujejo podatke samo, če testi padejo [24]. Orodja za poganjanje sprejemnih testov v še niso razvita do te mere, da bi omogočala tiho delovanje.

Če želimo neodvisno testirati enote, morajo biti testi enot preprosti, vsak test pa testira samo eno enoto testiranega razreda. Testi enot so torej načrtovani tako, da testirajo posamezno izolirano enoto. Tak način testiranja omogoča neodvisno testiranje posameznih enot, kar pomeni, da bo v v primeru problemov z eno določeno enoto, padel samo test za to enoto, ostali pa bodo preživeli. Tako lahko hitro in enostavno ugotovimo, kateri test je padel in zakaj.

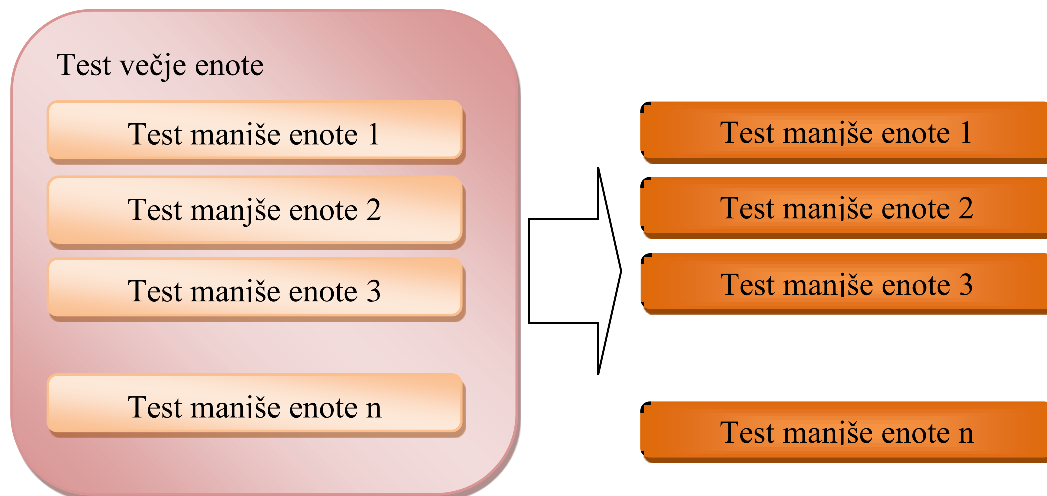
Za razliko od testov enot so sprejemni testi napisani na višjem nivoju, nivoju uporabniške zgodbe. Vsak test opisuje določen scenarij, po katerem uporabnik deluje s sistemom in definira, kako se bo sistem odzival. Testi testirajo delovanje tako med različnimi plastmi aplikacije kot tudi sodelovanje z zunanjimi odvisnostnimi. Zunanje odvisnosti morajo biti za pravilno testiranje delujoče in dosegljive, kar pomeni, da je potrebno za uporabo sprejemnih testov vložiti dobršno mero dodatnega dela za postavitev in čiščenje testnega okolja.

Postavitev in čiščenje testnega okolja pri testih enot je sorazmerno lahko opravilo, saj okolje predstavlja samo način za ponovno uporabo kode pri testih. Pri sprejemnih testih je lahko postavitev napeljave zelo kompleksno opravilo, ki je lahko porazdeljeno preko več različnih okolij sprejemnih testov. S stališča kompleksnosti testnega okolja lahko torej ugotovimo, da je okolje pri sprejemnih testih veliko bolj kompleksno kot okolje pri testih enot.

Večja kompleksnost sprejemnih testov se kaže tudi v dolžini izvajanja posameznih testov. Test enote testira eno posamezno majhno programsko enoto, ki naj praviloma ne bi vsebovala odvisnosti. Prav povezave in odvisnosti med posameznimi programskimi enotami pa tvorijo integracijsko celoto določenega dela programske opreme. Sprejemni testi so namenjeni testiranju takih celot. Sprejemni test tako testira večje število enot, zato je izvajanje posameznega testa veliko bolj počasno, kot izvajanje posameznega testa enote. Ker testiramo večje dele sistema, pa je število sprejemnih testov v okviru posamezne uporabniške zgodbe veliko manjše kot število testov enot.

Če povzamemo razlike med sprejemnimi testi in testi enot lahko ugotovimo, da se te nanašajo predvsem na lastnosti, ki se večinoma dotikajo udeležencev, piscev, hitrosti izvedbe in integracije v razvojna orodja. Če iz testov enot izvzamemo del, ki govori o tem, da je enota najmanjši možni del programa, in iz sprejemnih testov izvzamemo naročnika kot pisca testov in namesto njega postavimo programerja, pa lahko na obe vrsti testov gledamo enako. V okviru takega razmišljanja lahko na sprejemni test gledamo kot na test velike enote.

Test velike enote sicer testira več enot hkrati, vendar lahko te združimo v eno celoto, ki testira določeno funkcionalnost. Torej lahko tak test predstavimo kot test enote na višjem nivoju. Delitev testa večje enote na več testov manjših enot je predstavljena na Sliki 13.



Slika 13: Delitev testa večje enote na več testov manjše enote

Test večje enote predstavlja miselno ogrodje za pisanje testov manjše enote. Ker je dejanska funkcionalnost preverjena s pomočjo manjših enot, implementacija večjega testa ni nujna, nam pa večji test lahko predstavlja osnovo za grajenje testov manjših enot. Implementacijo testov večjih enot lahko vzamemo tudi kot vodilo za boljšo preglednost in lažjo organizacijo testov enot.

Pri preverjanju testov se lahko nanašamo na pravilo, ki pravi, da v primeru, da pade eden izmed testov manjše enote, ki je zajet v testu večje enote, pade tudi test večje enote. To lahko preslikamo tudi v kontekst sprejemnih testov, saj v primeru, da pade test enote, ki testira del, ki je zajet v sprejemnem testu, pade tudi ta sprejemni test. V primeru upoštevanja zgornjega pravila, lahko testiramo samo teste večji enot, torej teste na najvišji ravni.

V primeru, da se pomikamo s testov najmanjših enot po hierarhiji do testov višjih enot, končno pridemo do testa, ki je najvišje v hierarhiji in predstavlja test celotne funkcionalnosti. Tak test enote je pravzaprav sprejemni test napisan v programerskem jeziku.

4 TESTNO VODEN RAZVOJ S SPREJEMNIMI TESTI

4.1 TESTNO VODEN RAZVOJ

Testno voden razvoj (TVR) se je začel razvijati s prihodom agilnih modelov razvoja, ki imajo korenine v postopkovnih iterativnih in evolucijskih procesnih modelih, ki so se uporabljali že v petdesetih letih 20. stoletja.

Pred predstavitvijo praks ekstremnega programiranja leta 1998 ni bilo veliko pisnega gradiva, ki bi omenjalo koncept uporabe majhnih postopnih avtomatiziranih testov enot za vodenje razvoja in arhitekture procesa. Kljub pomanjkanju dokumentacije je veliko razvojnih ekip že uporabljajo način predhodnega pisanja testov.

Eden pionirjev testno vodenega razvoja Kent Beck trdi, da se je naučil programiranja na osnovi testov kot otrok, ko je bral knjigo o programiranju. Knjiga je predstavljala razmišljanja o programiranju na tak način, da programiraš tako, da vzameš vhodni trak in natipkaš, kaj pričakuješ na izhodnem traku. Potem programiraš toliko časa, da dobiš pričakovan izhodni trak [4].

Testno voden razvoj je disciplina načrtovanja in programiranja, kjer se vsak delček kode napiše kot odgovor testu, ki ga je programer napisal tik pred začetkom programiranja.

Osnovno načelo testno vodenega razvoja je pisanje testov, preden se začne razvijati funkcionalna koda. Teste pišemo v kratkih in hitrih iteracijah. Čeprav se princip testno vodenega razvoja uporablja že nekaj desetletij, pa je strategija v zadnjem času pridobila na uporabnosti. Testno voden razvoj v začetku ni imel takega imena, vendar je bil predstavljen kot praksa EP, ki je pomembna za analizo, načrtovanje in testiranje, obenem pa je omogočala razvoj s pomočjo preoblikovanja, skupnega lastništva in neprestane integracije. Skupaj s programiranjem v parih in preoblikovanjem je testno voden razvoj postal pomembna praksa EP [23]. Razvila so se orodja, ki podpirajo TVR v različnih programskih jezikih, prav tako je bilo napisanih kar nekaj knjig, ki opisujejo načine uporabe TVR.

Čeprav ime testno vodenega razvoja namiguje na metodo testiranja, ima sam izraz veliko večji pomen. Razvoj s pomočjo TVR usmerja programerja tako, da ta najprej napiše teste enot in šele potem kodo, ki jo testi testirajo. V takem primeru lahko razvijalec izvaja teste takoj zatem, ko so ti napisani. Smisel testno vodenega razvoja je konstrukcija programske opreme v zelo majhnih iteracijah z minimalno začetno arhitekturo [22].

Ključno pri razumevanju TVR je dejstvo, da tu ne gre (le) za testiranje, ampak tudi za načrtovanje. Vnaprejšnje definiranje testov razvijalcu pomaga pri razumevanju,

definiranju in modeliranju problema. TVR bi lahko definirali tudi kot stil razvoja, kjer razvijalec najprej napiše avtomatizirane teste in šele potem kodo, ki jo testi preverjajo, nato pa še odstrani podvojeno kodo in teste. Testi so napisani v istem jeziku kot programska koda, ki jo testiramo, in tako služijo tudi kot dokumentacija te programske kode.

Kent Beck, testno voden razvoj definira z dvema praviloma [4]:

- nikoli ne napiši niti delčka kode, če nimaš vnaprej napisanega testa in
- odstrani vso podvojeno kodo.

Prvo pravilo zahteva, da najprej napišemo teste in z njimi točno definiramo funkcionalnost, ki jo od kode želimo. Šele ko imamo napisane teste, se lotimo programiranja produkcijske kode. Pravilo preprečuje, da bi v kodo vključili funkcionalnost, ki je ne potrebujemo za našo rešitev.

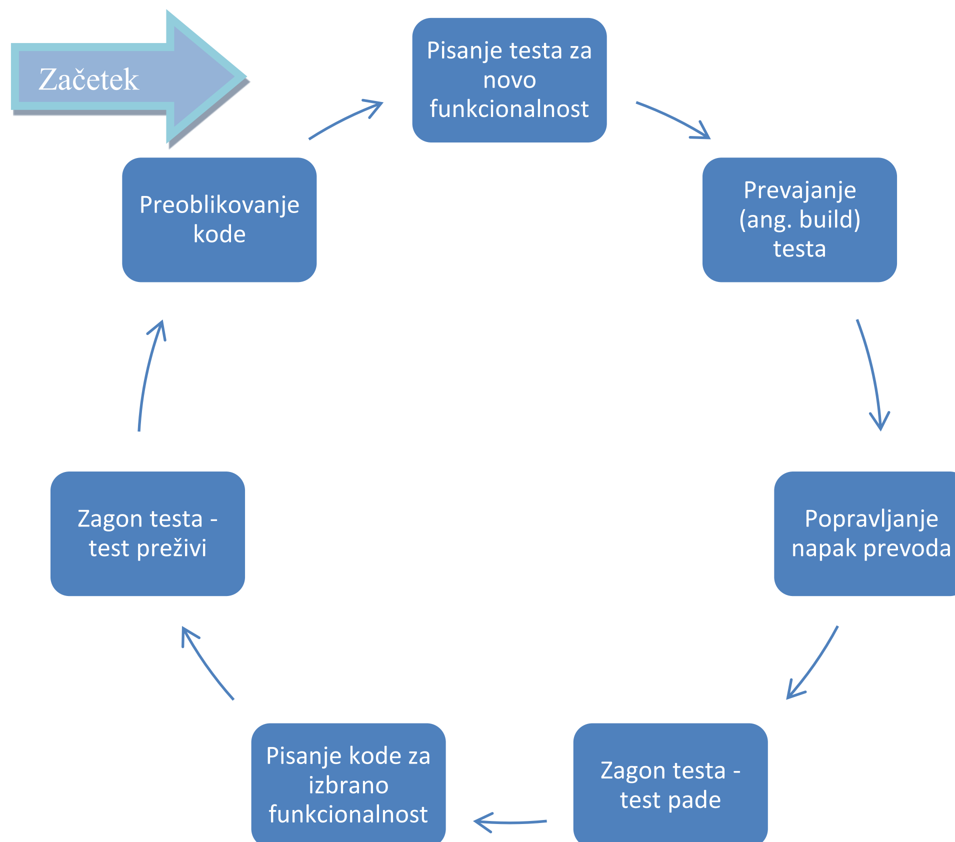
Drugo pravilo se dotika podvajanja kode, ki je glavni krivec za slabo arhitekturo. Podvajanje vodi k nekonsistentnosti, obenem pa zmanjšuje stopnjo zaupanja v kodo, saj s časoma pozabimo, kje se podvajanja pojavljajo. Pravilo od nas zahteva, da ob vsakem zaznanem podvajanju le-to takoj odstranimo.

Ker testi definirajo zahteve, je naša naloga, da napišemo kodo, ki bo ravno pravšnja, da bo zadovoljila naše teste. Pri pisanju je torej potrebno upoštevati pravilo – čim manj kode, obenem pa se držati naslednjih krakov:

- koda je namenjena izbrani publiki,
- koda preživi vse teste,
- koda nam sama po sebi pove vse, kar mora,
- koda ima najmanjše možno število razredov in
- koda ima najmanjše možno število metod.

Glavni vidik testno vodenega razvoja je pisanje avtomatiziranih testov posameznih programskih enot na najmanjši ravni, ki jo je še mogoče testirati.

TVR od nas zahteva, da naši testi najprej padejo. Ta ideja zagotavlja, da test resnično deluje in da lahko ujamemo napake. Ko to dosežemo, lahko začnemo z navadnim ciklom razvoja. Tak princip imenujemo rdeče/zeleno/preoblikuj, kjer rdeče pomeni, da test ne uspe (pade) in zeleno, da test uspe (preživi). Ko napišemo dovolj programske kode, da test preživi, se lotimo preoblikovanja kode. S pomočjo preoblikovanja pridobimo čisto in kratko programsko kodo. TVR predvideva, da je razvoj programske opreme vedno nepopoln in odprt za spremembe [9]. Tipičen proces testno vodnega razvoja programske opreme je prikazan na Sliki 14.



Slika 14: Proces testno vodenega razvoja – rdeče/zeleno/preoblikuj

TVR poleg testiranja uvaja tudi pisanje avtomatiziranih testov majhnih individualnih enot programske opreme. Enota predstavlja najmanjšo možno enoto programske opreme, ki jo je mogoče testirati. Stroka ni čisto enotna, kako predstaviti najmanjšo programsko enoto. V objektno orientiranem programiranju sta tako razred kot metoda predstavljena kot prava enota, čeprav je splošno metoda ali procedura najmanjša programska enota, ki jo je mogoče testirati.

Tradicionalno se je testiranje enot izvajalo potem, ko so razvijalci že napisali kodo enote. Časovno je to lahko predstavljajo kratek ali zelo dolg časovni okvir, teste pa so pisali testerji ali razvijalci. Pri uporabi TVR pa programer najprej napiše teste, šele potem napiše dejansko programsko kodo in lahko tako poganja teste, takoj ko je koda napisana.

Praksa predvideva, da so testi pomembna komponenta procesa razvoja programske opreme, ki omogoča hitro povratno informacijo in spremembe sistema. V primeru, da pride do spremembe programske kode, ki povzroči padec testa, razvijalec to takoj opazi in lahko kodo popravi. Ena izmed slabih strani TVR je skrb razvijalca tako za produkcijsko kodo kot za teste.

Ker so testi neodvisni med seboj, je zaporedje izvajanja testov neodvisno. Okolje programerju

sporoči število uspešno in neuspešno prestanih testov. Testi so napisani v enakem jeziku kot programska koda, poleg funkcije testiranja pa izvajajo tudi funkcijo dokumentacije. Uporaba enakega jezika pripomore k preprostosti, razumljivosti in fleksibilnosti.

Raziskave učinkovitosti TVR so pokazale zmanjšano število napak in povečanje kvalitete, tako na akademski kot profesionalni ravni. Profesorji so začeli raziskovati vpeljavo testno vodenega razvoj v računalniško znanost in pedagogiko računalniškega inženiringa. Nekaj teh prizadevanj je bilo vpeljanih v kontekst projektov ekstremnega programiranja [11].

Uporaba pravih razvojnih orodij je postala pomembna praksa pri razvoju modernih računalniških sistemov. Različna orodja, kot so okolja za pisanje in razvoj programske opreme, prevajalniki in razhroščevalci, so z razvojem povečala produktivnost razvijalcev. Brez pravih podpornih orodij se TVR zagotovo ne bi mogel razviti do te mere. TVR predvideva obstoj okolja za avtomatsko testiranje, saj to omogoča preprosto kreiranje in izvajanje testov enot.

Avtomatsko testiranje vključuje pisanje testov in umeščanje kode v testno okolje. Okolja za avtomatsko testiranje so namenjena minimiziranju napora pri testiranju. Za lažje izvajanje avtomatskega testiranja uporabimo testne dvojnike. V podporo testno vodenemu razvoju sta Erich Gamma in Kent Beck razvila JUnit, orodje za pomoč avtomatskemu testiranju v Javi. Orodje se je potem prevedlo tako, da ga je bilo mogoče uporabljati tudi v drugih programskih jezikih. Skupino orodij so poimenovali xUnit. Orodje omogoča programerju tak način dela, da napiše skupek avtomatiziranih testov, jih izvede in preveri pravilno delovanje kode [30].

Za pomoč pri uporabi TVR se je sčasoma razvilo še več orodij, ki omogočajo tudi kreiranje testnih dvojnikov. Testni dvojniki služijo kot zamenjava pravih objektov, vendar s to razliko, da samo simulirajo njihovo pravo delovanje. Na ta način se znebimo odvisnosti med objekti, saj objektov ni potrebno združevati. Tako lahko testiramo posamezne objekte neodvisno od ostalih.

4.2 SPREJEMNI TESTI V KONTEKSTU EKSTREMNEGA PROGRAMIRANJA

Tradicionalno se sprejemne teste izvaja potem, ko je programska koda že razvita. V kontekstu EP je bila doslej večja pozornost usmerjena v teste enot kot vodila za razvoj programske opreme. Manjša pozornost je bila namenjena sprejemnim testom. V okviru magistrske naloge želimo večjo pozornost nameniti vodenju razvoja s pomočjo sprejemnih testov.

Ena izmed klasičnih kritik EP je podedovano pomanjkanje dokumentacije. Pri EP so dokumenti napisani, vendar imajo veliko manjšo vrednost kot delujoča programska oprema [5].

V tem kontekstu lahko vidimo sprejemne teste kot delno nadomestilo za specifikacijo, posebno za dokumentacijo zahtev programske opreme. Kritika tradicionalnih dokumentov z zahtevami je ta, da lahko dokumente hitro preberemo in preletimo ter tako problema ne razumemo natanko, ampak samo okvirno. Če dokumentacijo zahtev nadomestimo z uporabniškimi zgodbami in sprejemnimi testi, ki jih podpirajo, zahteve programske opreme bolje opredelimo in razumemo, način dela pa nam ne dopusti, da bi zahteve videli samo okvirno.

Sprejemni testi v sistemu v nasprotju z dokumentacijo zahtev omogočajo podrobno povratno informacijo o stanju implementacije uporabniških zgodb. Prav tako hitro pokažejo na manjkajoča znanja razvojne ekipe in premalo temeljit razmislek naročnika. V primeru razvoja s sprejemnimi testi teže zaidemo v neusklajenost med zahtevami in zahtevano aplikacijo, prav tako pa nam predstavlja pomoč pri razumevanju in spremembi zahtev.

Cikli takega razvoja premikajo definicije specifikacij sprejemnih testov v začetno fazo razvoja programske opreme. Sprejemni testi tako dobijo novo vlogo, saj niso namenjeni samo merjenju kvalitete, ampak tudi razvoju kvalitetne programske opreme.

V takem primeru morajo biti sprejemni testi izvedljivi in uporabni skozi celoten cikel produkcijske kode. Glede na produkcijsko kodo opredelimo zahteve sprejemnih testov, ki morajo biti [14]:

- Preprosti za pisanje: ker sprejemni testi predstavljajo osnovo za pisanje produkcijske kode, morajo biti preprosti, saj bi v nasprotnem primeru postali nepopolni in zastareli.
- Točni: v primeru, da imajo že sprejemni testi napake, potem bo tudi produkcijska koda napačna.
- Razumljivi tudi netehničnim osebam: testi morajo biti napisani v takem jeziku, da jih razume naročnik.
- Preprosti za vzdrževanje: način urejanja in vzdrževanja testov mora biti preprost.
- Ažurni: testi morajo biti napisani in preverjeni ter popravljeni preden se lotimo pisanja oziroma popravkov v produkcijski kodi.

V primeru uporabe sprejemnih testov za vodenje razvoja mora razvojna ekipa sodelovati z uporabnikom in mu pokazati napredovanje razvoja izdelka. S tem, ko uporabnik oz. naročnik med razvojem preveri delovanje delov produkta s sprejemnimi testi, prevzame tudi del odgovornosti za pravilno delovanje celotnega produkta. Njegova odgovornost je tudi specifikacija sprejemnega testa.

Vsak sistem gre pred uporabo skozi fazo testiranja, kjer naročnik ugotovi, ali sistem deluje tako, kot je pričakoval. V praksi se navadno takega testiranja ne izvaja, dokler uporabnik ne odkrije napake v sistemu. Tak način privede do nezadovoljstva naročnika in slabega slovesa računalniškega podjetja.

Testiranje na koncu razvojnega cikla v praksi velikokrat pade v čas, ki že presega rok končanja projekta, zato se testiranju ne posveča dovolj pozornosti. Problem predstavlja tudi pozabljivost programerjev. Programerji, ki so napisali program danes in ga testirali teden kasneje, bodo v tem času že pozabili podrobnosti programa, kar bo oteževalo hitro odpravo napak. Testiranje takoj potem, ko je del programa razvit, je zato ključnega pomena za kakovosten razvoj programske opreme.

Ena izmed vrednot ekstremnega programiranja je povratna informacija. Pri razvoju je zelo pomembna čimprejšnja informacija o tem, kako dobro program deluje. Take povratne informacije lahko dobimo s pomočjo sprejemnih testov. Določanje sprejemnih testov je odgovornost naročnika. Če te dostavimo programerjem na koncu vsake iteracije, pospešimo projekt, prav tako pa projekt na koncu posamezne iteracije predstavlja neko poslovno vrednost.

Praksa ekstremnega programiranja pravi, da moramo testirati samo tiste dele, za katere hočemo da delujejo, torej dele, za katere se splača, da jih programerji naredijo, in to naredijo pravilno. Veliko projektov ima v ozadju starejše sisteme. Iz starejših sistemov lahko razberemo vhode in izhode in na ta način naredimo ogrodje za testiranje. Pametni ekstremni programerji bodo iz starih sistemov prebrali informacije in jih vnesli v razvijajoči se sistem.

Ena izmed slabih praks preverjanja delovanja sistema je preverjanje izhoda tako, da ga iz sistema ročno preberemo. Ročno preverjanje računalniških izhodov je zelo slaba ideja, saj smo ljudje nagnjenji k napačnemu in površnemu branju podatkov. Ekstremno programiranje zato predvideva, da morajo biti vsi testi avtomatizirani.

Ednini način, da hitro in samozavestno programiramo, je s pomočjo avtomatiziranega sistema sprejemnih testov in testov enot. Način nam omogoča, da takoj odkrijemo napako, tudi če smo preko odvisnosti spremenili neželen del programa.
Avtomatizacija testov predstavlja ključni del, vendar se v nasprotju s testi enot, zaradi narave sprejemnih testov, pri avtomatizaciji pogosto pojavijo težave [15]. Obstaja več načinov za avtomatizacijo testov:

- avtomatizacija s pomočjo poganjanja skriptih programov, ki preberejo vhodne podatke, jih sprocesirajo in oddajo na izhod;
- uporaba xUnit ogrodij za testiranje, kjer napišemo sprejemne teste kot programe. Uporabo preko xUnit orodij lahko razvijemo do te mere, da jih lahko preprosto uporabljajo naročniki;
- prepustimo uporabnikom, da napišejo teste v programu za urejanje besedila ali preglednic, ki jim je najbližje (npr. MS Excel). S pomočjo skript potem preberemo teste iz programa, jih poženemo in preverimo delovanje;
- uporaba snemalca vhodov, ki preverja vse vhodne akcije, ki jih uporabnik izvaja. S pomočjo posnetka potem definiramo teste, ki jih avtomatsko preverjamo;

- s pomočjo integriranega okolja za izvajanje testov, kot je okolje Fitnesse, ki ima predpisan način pisanja in izvajanja testov.

Sprejemni testi morajo biti zgrajeni tako, da jih lahko avtomatsko testiramo, način avtomatskega testiranja pa mora biti preprost in intuitiven. Kadar sami gradimo okolje za testiranje, moramo paziti, da v to ne vložimo preveč časa. Edini način, da preverimo, če razvoj poteka v pravi smeri, je, da testiramo razvit program že na koncu vsake iteracije [1]. Naročniki imajo pravico, da preverijo napredovanje razvoja sistema. To najlaže preverijo s pomočjo poganjanja avtomatiziranih testov.

4.2.1 Testno voden razvoj s sprejemnimi testi

Doslej je bila pozornost testno vodenega razvoja usmerjena predvsem v uporabo testov enot. Vnaprejšnje pisanje testov enot je predstavljalo izhodišče za vodenje procesa razvoja programske opreme. V kontekstu ekstremnega programiranja prestavljamo sprejemne teste v začetni cikel razvoja programske opreme.

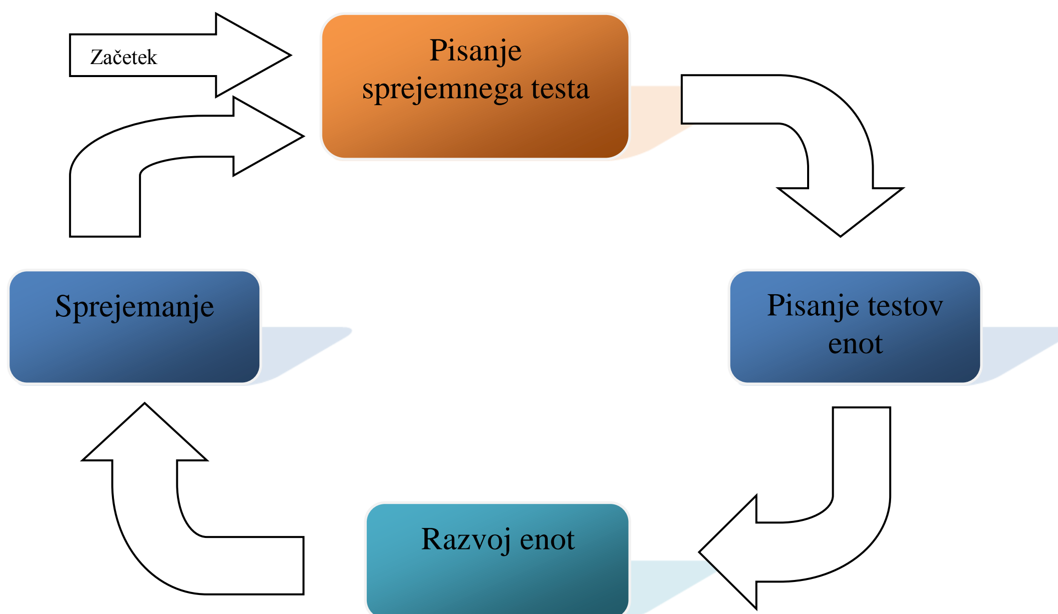
Testno voden razvoj s sprejemnimi testi postavimo nad klasičen testno voden razvoj s testi enot in poiščemo vzporednice med obema vrstama razvoja. Tako kot je osnovno načelo TVR s testi enot pisanje testov enot pred pisanjem programske kode, je osnovno načelo TVR s sprejemnimi testi pisanje sprejemnih testov pred razvojem programa ali računalniškega sistema.

Če gre pri TVR s testi enot za pisanje testov na najnižjem nivoju programa, torej na nivoju najmanjše enote, ki jo predstavlja metoda ali procedura, gre pri TVR s sprejemnimi testi za pisanje testov na najvišjem nivoju, torej na nivoju uporabniške zgodbe. Idejo TVR s testi enot, da v naprej napišemo test, ki nam je potem v pomoč pri vodenju in odločanju, hkrati pa predstavlja tehnično dokumentacijo, lahko prav tako uporabimo v kontekstu sprejemnih testov. Sprejemni testi v kontekstu TVR širijo dokumentacijo in k programerski dodajajo tudi naročniško dokumentacijo. Pri TVR s testi enot mejo za nadaljevanje procesa predstavlja posamezna razvita, testirana in preoblikovana enota programa. V primeru TVR s sprejemnimi testi tako mejo predstavlja zaključena funkcionalnost, ki zadovolji izbran sprejemni test.

Proces testno vodenega razvoja s sprejemnimi testi je prikazan na Sliki 15 in je sestavljen iz naslednjih korakov:

- Pisanje sprejemnega testa: s pomočjo uporabniških zgodb napišemo sprejemni test, ki opisuje delovanje posamezne funkcionalnosti programa.
- Pisanje testov enot: sprejemni test je vodilo za pisanje testov enot.

- Razvoj enot: razvoj enot po principu testno vodenega razvoja s testi enot.
- Sprejemanje: sprejemne teste poganjamo šele, ko je razvita prva manjša funkcionalnost na ravni testa sprejemanja. Ko je prva funkcionalnost narejena, razvijemo okolje, s pomočjo katerega povežemo sprejemne teste in razvito programsko kodo.



Slika 15: Testno voden razvoj s sprejemnimi testi

Proces izvajamo toliko časa, da sprejemni test preživi. Ko posamezen sprejemni test preživi, lahko nadaljujemo s pisanjem naslednjega sprejemnega testa v okviru izbrane uporabniške zgodbe. Ko vsi sprejemni testi izbrane uporabniške zgodbe preživijo, nadaljujemo z naslednjo uporabniško zgodbo.

Pri izvajanju procesa se srečamo s problemom testiranja pravilnosti delovanja sprejemnih testov. Sprejemni testi so za naročnika opredeljeni kot testi črne škatle, kar pomeni, da naročnik ne more direktno preveriti, če so testni subjekti s sprejemnimi testi povezani tako, kot od njih pričakuje. Eden od glavnih namenov testiranja sprejemanja je povečanje zaupanja naročnika, da sistem deluje pravilno. Okolje za testiranje mora zato imeti nek način, ki omogoča testerju vpogled v delovanje. Ena od možnih rešitev tega problema je prikaz dnevnika, ki beleži vse akcije, ki so povezane s testiranjem [24].

Sprejemni test je celota šele, ko je napisan tako opisni del testa kot tudi programski del testa. Pri vsakem testu mora torej sodelovati tudi programer, ki glede na opisni del testa napiše povezavo, ki poveže sprejemni test z dejansko funkcionalnostjo, ki jo testira.

Če želimo rešiti problem pravilnega delovanja sprejemnih testov, potem lahko v fazi pisanja povezave med sprejemnimi testi in programsko kodo napišemo tudi preprosto komponento, ki definira funkcionalnost do te mere, da zadovolji sprejemni test. Taka implementacija nima polno razvite funkcionalnosti in ne vsebuje kompleksne programske kode. Implementacija je pravzaprav testni dvojnik, ki oponaša delovanje prave komponente. Ker komponenta ni namenjena premostitvi odvisnosti v programski kodi, ampak jo uporabljamo za testiranje pravilnosti delovanja sprejemih testov, jo imenujemo slepa funkcionalnost.

Osnova slepih funkcionalnosti je preprosta implementacija, zato slepi funkcionalnosti napišemo samo toliko programske kode, da zadovolji sprejemni test. Poleg tega, da slepa funkcionalnost omogoča preverjanje pravilnosti delovanja sprejemnih testov, pa programerju predstavlja tudi dokumentacijo napisano v programski kodi. Definicija je zato lahko programerju v pomoč pri pisanju kode. Ker slepe funkcionalnosti zrcalijo sprejemne teste v programersko kodo, jih lahko uporabimo za pomoč pri pisanju testov enot.

Na Sliki 16 je predstavljen proces TVR s sprejemnimi testi, ki vključuje tudi pisanje slepih funkcionalnosti. Proces vključuje naslednje korake:

1. Zajem in priprava uporabniških zgodb:
    a. Pisanje uporabniških zgodb.
    b. Ureditev in prioritizacija uporabniških zgodb.
    c. Priprava opisnih sprejemnih testov za izbrane uporabniške zgodbe.

2. Priprava sprejemnih testov:
    a. Izbira testov, ki jih je smiselno in možno implementirati v okviru izbranih uporabniških zgodb.
    b. Priprava in preoblikovanje testov v obliko primerno za ogrodje, ki omogoča podporo avtomatskemu testiranju sprejemnih testov.
    c. Izgradnja povezave, ki povezuje opisni in tekstovni del sprejemnih testov.

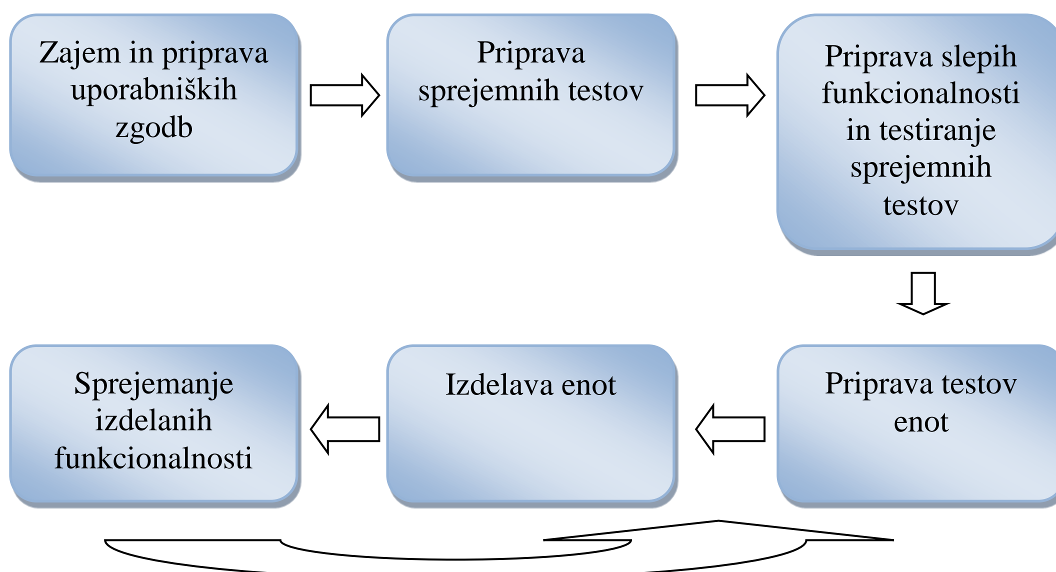
3. Priprava slepih funkcionalnosti in testiranje sprejemnih testov:
    a. Priprava slepih funkcionalnosti.
    b. Testiranje pravilnosti delovanja sprejemnih testov s pomočjo slepih funkcionalnosti
    c. Preoblikovanje slepih funkcionalnosti za boljšo berljivost kode.

4. Priprava testov enot:

   a. Pisanje testov enot s pomočjo sprejemnih testov in slepih funkcionalnosti.
   b. Preoblikovanje testov enot za boljšo berljivost kode.

5. Izdelava enot:
   a. Pisanje samo toliko programske kode, da zadovolji test enote.
   b. Testiranje posameznih enot s testi enot.
   c. Preoblikovanje programerske kode enot.

6. Sprejemanje izdelanih funkcionalnosti:
   a. Priklop sprejemnih testov na izdelane funkcionalnosti.
   b. Avtomatsko preverjanje sprejemnih testov.



Slika 16: Testno voden razvoj s sprejemnimi testi in slepimi funkcionalnostimi

Koraka priprava testov enot in izdelava enot ponavljamo toliko časa, da zadovoljimo sprejemni test za funkcionalnost, ki jo posamezne enote implementirajo. Ko zadovoljimo vse sprejemne teste določene uporabniške zgodbe, nadaljujemo z izvajanjem procesa za naslednjo uporabniško zgodbo.

Pomemben del izvajanja procesa je uporaba avtomatiziranih orodij za poganjanje tako sprejemnih testov kot tudi testov enot. Orodja omogočajo naročniku in programerjem hitro preverjanje delovanja vseh izdelanih komponent programa. Ogrodja na nivoju enot

programerju omogočajo neprestano in hitro testiranje vseh izbranih enot, ogrodja na nivoju sprejemnih testov pa testiranje delovanja celotne uporabniške zgodbe. Naročnik tako lahko s pomočjo avtomatskih testov hitro preveri, če izdelan program deluje v skladu z njegovimi pričakovanji. Primera takih orodij sta Fitnesse in Visual Studio.

5 UPORABA TESTNO VODENEGA RAZVOJA PRI RAZVOJU SPLETNE APLIKACIJE

5.1 SPLOŠEN PREGLED ORODIJ

Trenutno na trgu obstaja več orodij, ki so v pomoč pri posameznem delu testno vodenega razvoja. S pomočjo internetnega iskanja pridemo do naslednjih skupin orodij [14]:

- Orodja, ki omogočajo snemanje in predvajanje testov, ki smo jih naredili naknadno, ko je bila programska koda že razvita. Testne skripte, ki definirajo korake za testiranje in so narejene s temi orodji, so težko berljive za uporabnika, prav tako pa je take skripte težko vzdrževati.

- Prihod testno vodenega razvoja je prinesel razvoj različnih orodij za avtomatsko testiranje, ki so skupno poznana kot orodja xUnit (jUnit za Javo, nUnit za .Net, sUnit za Smalltalk). Ta orodja podpirajo teste, ki so jih napisali razvijalci z uporabo obstoječih programskih jezikov v obstoječih integriranih razvijalskih okoljih. Svoje testno okolje je v svoje orodje za razvoj programske opreme Visual Studio vpeljal tudi Microsoft.

- Orodja za integrirane teste ali FIT (Framework for Integrated Testing) orodja. Orodja so omogočila proces avtomatiziranega testiranja s pomočjo izraznih, berljivih in odstavčnih sprejemnih testov. FIT s pomočjo ločitve testov od izvajalske kode predstavlja veliko fleksibilnost v uporabi testov. S pomočjo orodja Fitnesse sprejemne teste približamo dejanskemu subjektu problema in olajšamo izvajanje le-teh, saj lahko teste poganjamo s pomočjo navadnega brskalnika. Čeprav so ta orodja napredna, pa je delo z njimi dokaj okorno, saj je za pisanje in vzdrževanje testov potrebno veliko dela, prav tako pa za pisanje ni dobrega podpornega razvojnega okolja.

- Poznamo tudi mnogo ostalih orodij, ki predstavljajo različne testne možnosti in jezike (spletni testi so navadno napisani v XML jeziku, Watir testi so napisani v Rubiju), vendar ti testi ostajajo na nivoju xUnit orodij.

- Vedenjsko voden razvoj predstavlja dobršen evolucijski korak proti določanju poslovno orientiranih potreb namesto testov. Vedenjsko voden razvoj se naslanja na nedvoumen domensko specifičen jezik, ki omogoča definicijo deklarativnih specifikacij. Orodja za pomoč vedenjsko vodenemu razvoju kot jBehave in rSpec predstavljajo nov način in kažejo velik potencial za nadaljnji razvoj.

Razlike med orodji pritiskajo na ekipe, ki se ukvarjajo z agilnim razvojem, k izboljšavi testnih orodij. Orodja se izboljšujejo predvsem v tej meri, da povečujejo njihovo berljivost in omogočajo lažje vzdrževanje.

Za potrebe magistrske naloge bomo uporabili orodje za pisanje testov enot Visual Studio in orodje za izvajanje sprejemnih testov Fitnesse.

5.1.1 Fitnesse

Ogrodje za integrirano testiranje (FIT – Framework for Integrated Testing) je namenjeno sprejemnemu testiranju. Ogrodje je razvil Ward Cunningham v programskem jeziku Java. Ena glavnih idej orodja FIT je promocija sodelovanja in omogočanje poslovnim analitikom, da pišejo in preverjajo teste. Čeprav je ogrodju FIT dodano orodje za izvajanje testov, pa ta nima nikakršne podpore pri pisanju testov. Prvotna ideja FIT je bila, da teste napišemo v MS Wordu ali Excelu oziroma kakršnem koli orodju, ki zna izvoziti test v HTML obliko.

Robert Martin in Micah Martin sta prišla na idejo, da bi osnovala skupno orodje za pisanje testov [3]. Glede na izkušnje s pisanjem testov sta kreirala Fitnesse, spletno masko za pisanje testov, ki je osnovana na sistemu Wikipedije. Fitnesse je pravzaprav integrirano okolje, s pomočjo katerega lahko pišemo in izvajamo teste. Orodje je namenjeno predvsem pospešitvi pisanja in izvedbe več testov. Čeprav je orodje razvito v Javi, je narejeno tako, da ga lahko s pomočjo vtičnikov za poganjanje testov uporabimo v različnih razvijalskih okoljih.

Pisanje testov s pomočjo orodja Fitnesse ne zahteva dodatnih tehničnih znanj ali znanj programskih jezikov. Testi so napisani v tabelarični obliki. V stolpcih tabele so specificirani tako testni vhodi kot tudi pričakovani rezultati. Rezultati so označeni s pomočjo znaka vprašaj v naslovu stolpca. Tabele predstavljajo pregleden in preprost način za testiranje in pregledovanje rezultatov, saj jih lahko napišemo v Excelu, Wordu ali kateremkoli HTML urejevalniku.

Številčni podatki o občini	
Vrsta podatka	Pričakovana vrednost?
Šifra občine	1223
Šifra regije	9
Šifra upravne enote	53
Ime občine	Škofja Loka
Ime regije	Gorenjska
Ime upravne enote	Škofja Loka

Slika 17: Tipični Fitnesse test

Tabele FIT z domensko kodo povežemo s pomočjo tanke plasti okolja, v kateri se nahaja povezava. FIT za testiranje ne potrebuje veliko programske kode, potrebuje jo samo toliko, da preveri smiselnost podrejenih vmesnikov. Velikokrat FIT povezave predstavljajo prvega odjemalca naše kode.

Fitnesse je spletno osnovan strežnik, ki omogoča preprosto uporabo za potrebe sodelovanja. Poslovni analitiki in druge netehnične osebe za poganjanje Fitnesse ne potrebujejo nobenih dodatnih namestitev programske opreme, saj lahko do sistema dostopajo preprosto preko spletnega brskalnika. Vso dodatno dokumentacijo, ki je v pomoč pri razumevanju testov in prepoznavanju poslovnega problema, lahko kot pomoč pri razvoju testov pripnemo stranem s tabelami.

Fitnesse je torej sodelovalno orodje za razvoj programske opreme, saj omogoča izboljševanje sodelovanja in komuniciranja. Fitnesse pravzaprav omogoča uporabnikom, testerjem in programerjem, da se naučijo, kako naj bi programska oprema delovala in obenem primerjajo pridobljene in zahtevane rezultate. Tipični Fitnesse test je prikazan na Sliki 17.

Pravilnost testov je prikazana s pomočjo barvanja celic. Če testi preživijo, se celice obarvajo zeleno, v nasprotnem primeru pa se celice obarvajo rdeče. V primeru, da test ni preživel, se poleg pričakovane izpiše tudi prava vrednost podatka. Fitnesse s pomočjo preprostega označevalnega jezika omogoča kreacijo naslovov krepkega, podčrtanega in poševnega teksta ter uporabo naštevanj in ostalih načinov preprostega oblikovanja teksta [29].

Fitnesse je pravzaprav preprosto odprtokodno orodje, ki ekipam, ki razvijajo programsko opremo, omogoča [29]:

- skupno definicijo sprejemnih testov,
- poganjanje testov in preverjanje rezultatov,
- kreiranje in urejanje testnih strani,
- uporabo orodja brez namestitve.

Tabele in programsko kodo FIT povezuje s pomočjo vnaprej definiranih okolij. Najbolj uporabna okolja so [3]:

- Stolpično okolje (ang. column fixture) – favorizira poslovno logiko, saj uporablja majhne funkcije, s katerimi preverjamo pravilnost podatkov, ki se nahajajo v stolpcih tabele.
- Vrstično okolje (ang. row fixture) – združuje, preverja in organizira domenske objekte. To je edino okolje, ki direktno zrcali domenske objekte, saj ostali zrcalijo sebe ali pa svojo vrsto.
- Izvajalno okolje (ang. action fixture) – izvaja se na grafičnem vmesniku problemskega modela, saj upravlja dodatna manjša okolja, ki so odgovorna za vsako pričakovano zaslonsko sliko modela.

FIT in Fitnesse sta orodji, ki sta predvsem dobri v tem, da v koraku definicije in verifikacije kriterija sprejemanja omogočata netehničnim osebam vpeljavo v proces testiranja. Orodji omogočata razvijalcem lahek prenos zahtev in pogovorov v teste. Poslovni analitiki in managerji lahko preprosto berejo teste in preverjajo rezultate ter tako spremljajo napredovanje projekta. Ker testna pravila niso neposredno spojena s kodo, se lahko testi razvijajo v skladu s poslovnimi pravili, ki so neodvisna od programske kode. Tak način nam omogoča, da napišemo teste najprej, še preden so definirani vmesniki, brez bojazni, da bi kakorkoli pokvarili zadnjo zgrajeno verzijo. Fitnesse testi so tudi dober način za posredovanje zahtev drugim (notranjim ali zunanjim) razvijalcem ali skupinam, saj se obnašajo kot tehnične specifikacije, ki morajo biti realizirane. Kljub vsemu pa Fitnesse ne predstavlja splošne rešitve za probleme, ki jih prinaša testiranje, saj ne omogoča vseh funkcionalnosti, kot so denimo podpora za operacije stila posnemi-predvajaj [9].

5.1.2 Visual Studio

Microsoftovo razvojno orodje Visual Studio, različica Team Test, poleg običajnih razvojnih funkcionalnosti ponuja tudi dodatke, ki omogočajo lažjo uporabo testno vodenega razvoja. Orodje omogoča:

- generacijo programske kode za testne dvojnike,
- poganjanje testov znotraj razvojnega okolja,
- vključevanje testnih podatkov in podatkovne baze,
- analize pokritja kode.

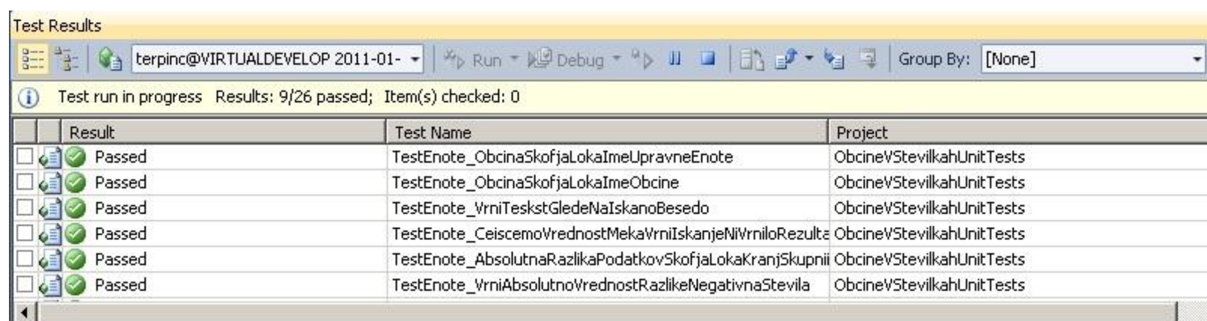
Prav tako kot v drugih orodjih za testiranje tudi v Visual Studiu posamezne atribute podajamo v oglatih oklepajih (npr. [Test]), kar nam prinaša fleksibilnost pri poimenovanju testnih metod. Vsa poimenovanja metod in razredov so povsem poljubna. Z uporabo imenskih prostorov (ang. namespaces) pa nam ogrodje povsem avtomatsko zgradi tudi hierarhijo testnih zaporedij. Spisek

atributov, ki jih orodje uporablja, je predstavljen v Tabeli 3.

Atribut	Opis
TestClass()	Atribut določa testno okolje.
TestMethod()	Atribut določa testni primer.
AssemblyInitialize()	Metode, ki jih določa ta atribut so uporabljene pred prvo testno metodo v prvem testnem razredu.
ClassInitialize()	Metode razreda označenega s tem atributom se izvedejo pred prvim testom.
TestInitialize()	Metode označene s tem atributom se izvedejo pred izvedbo vsake testne metode.
TestCleanup()	Metode označene s tem atributom se izvedejo po izvedbi vsake testne metode.
ClassCleanup()	Metode označene s tem atributom se izvedejo po izvedbi vseh testov.
AssemblyCleanup()	Metode označene s tem atributom se izvedejo po izvedbi zadnje testne metode v prvem testnem razred izbranem za izvajanje.
Description()	Opis testne metode.
Ignore()	Prezri testno metodo ali razred.

Tabela 3: Atributi, s katerimi označimo razrede in metode kot testne

Visual Studio omogoča poganjanje testov znotraj razvojnega orodja. Za boljšo fleksibilnost omogoča različne načine poganjanja (test v trenutnem kontekstu, vseh testov v razredu, vseh testov v imenskem prostoru). Pregled nad testi pa nam omogoča pregledni grafični vmesnik, ki je prikazan na Sliki 18.



Slika 18: Prikaz testov v orodju Visual Studio

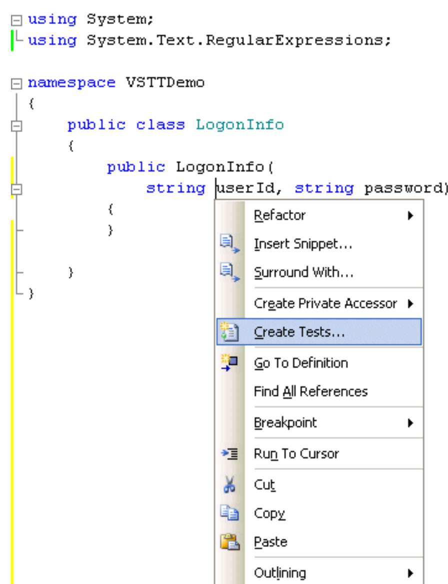
Osnovni gradniki za testiranje v orodju so trditve »Assertions«. Orodje ima na voljo široko paleto trditev, ki so implementirane kot statične metode v treh trditvenih razredih.

Metoda `AreEqual` je največkrat uporabljena metoda pri testiranju, zato smo v okviru magistrske naloge opisali samo to metodo. Metoda preverja, če sta vrednosti, ki jih preverjamo, enaki in vrne vrednost 1 – `true`, če sta vrednosti enaki oziroma vrednost 0 – `false`, če sta vrednosti različni. Metoda zahteva dva argumenta, lahko pa ji dodamo še enega, ki predstavlja dodatno sporočilo, ki nam ga bo metoda vrnila v primeru, da preverjanje pade. Metoda kot vhodni parameter sprejme kakršenkoli tip objekta: od celoštevilskih vrednosti pa do spremenljivke tipa objekt – `object`. Ostale metode so prikazane v Tabeli 4 [33].

Assert Class	StringAssert Class	CollectionAssert Class
AreEqual()	Contains()	AllItemsAreInstancesOfType()
AreNotEqual()	DoesNotMatch()	AllItemsAreNotNull()
AreNotSame()	EndsWith()	AllItemsAreUnique()
AreSame()	Matches()	AreEqual()
EqualsTests()	StartsWith()	AreEquivalent()
Fail()		AreNotEqual()
GetHashCodeTests()		AreNotEquivalent()
Inconclusive()		Contains()
IsFalse()		DoesNotContain()
IsInstanceOfType()		IsNotSubsetOf()
IsNotInstanceOfType()		IsSubsetOf()
IsNotNull()		
IsNull()		
IsTrue()		

Tabela 4: Razredi in metode za preverjanje vrednosti, definirani v orodju za testiranje Visual Studio

Visual Studio s pomočjo generatorjev kode omogoča tudi avtomatsko generacijo okvira programske kode testov ter testnih dvojnikov. Okvir testne metode generiramo direktno iz že izbrane implementirane metode. Orodje tako dobro podpira proces izdelave programske kode v testno vodenem razvoju, saj omogoča avtomatsko generacijo okvirjev testnih metod neposredno iz slepih funkcionalnosti. Slika 19 prikazuje vmesnik za generacijo testov.



Slika 19: Generacija testov s pomočjo orodja Visual Studio

5.2 RAZVOJ SPLETNE STRANI OBČINE V ŠTEVILKAH

Tiskana publikacija Občine v številkah je prvič izšla leta 2009 z namenom približati različne kazalnike o občinah širši javnosti s pomočjo vizualnih in numeričnih izkazov. Tiskana publikacija je naletela na dober odziv. Ker Statistični urad RS prehaja na digitalizacijo tiskanih medijev, je bila za leto 2010 predvidena digitalna oblika publikacije. Tako je bil sprejet sklep, da bo leta 2010 publikacija izšla v obliki spletne strani.

Za izvedbo projekta smo bili določeni: vodja projekta, dve vsebinski pomočnici in tehnična oseba. Vodja projekta je bila zadolžena za koordinacijo projekta in pripravo vsebin, vsebinski pomočnici sta skrbeli za pripravo in preverjanje vsebin. Tehnična oseba je bila zadolžena za razvoj in objavo spletne strani glede na pripravljeno vsebino.

Za razvoj in izvedbo spletne strani je tehnična oseba predlagala razvoj s pomočjo agilnih metodologij. Teoretično predstavljen proces smo želeli preveriti v praksi. Predlagan je bil proces testno vodenega razvoja s sprejemnimi testi in slepimi funkcionalnostmi. Projektna skupina se je strinjala s predlaganim procesom.

5.2.1 Uporabniške zgodbe

Na uvodnem sestanku smo najprej določili tipične uporabnike, ki bodo uporabljali spletno stran. Izpostavili smo dva tipa uporabnikov:

- Splošni uporabnik – tipični uporabnik spletne strani, ki bo na spletni strani iskal in bral različne podatke o občinah.
- Upravljavec vsebine – upravljavec vsebin na spletni strani, ki bo s pomočjo različnih orodij spreminjal oz. dodal podatke na spletno stran.

Glede na izbrane uporabnike smo se lotili pisanja uporabniških zgodb. Pri iskanju idej za pisanje uporabniških zgod smo si pomagali s pomočjo že obstoječih spletnih strani in obstoječe publikacije iz leta 2009. Napisali smo večje število uporabniških zgodb, ki smo jih potem kategorizirali. Nekatere uporabniške zgodbe smo združili, nekatere opustili, nekaj pa smo jih prepustili za kasnejšo implementacijo. Poleg uporabniških zgodb smo definirali tudi nekatere omejitve, ki so prikazane na Sliki 23.

Izluščili smo naslednje uporabniške zgodbe:

- uporabniške zgodbe, ki se nanašajo na splošnega uporabnika so prikazane na Sliki 20;
- uporabniške zgodbe, ki se nanašajo na upravljavca so prikazane na Sliki 21;
- uporabniške zgodbe, ki jih lahko implementiramo kasneje, so prikazane na Sliki 22.

Občine
Uporabnik lahko s spletišča prebere različne statistične podatke o občini (grafično, semantično, numerično).

Tematski članki
Uporabnik lahko s spletišča prebere tematske članke, ki se povezujejo na spletno stran s podatki.

Iskalnik
Uporabnik lahko posamezne podatke o občinah poišče s pomočjo iskalnika.

Navigacijsko drevo
Uporabnik lahko do posameznih strani dostopa preko navigacijskega drevesa.

Dinamična stran
Širina spletišča se spreminja dinamično.

Slika 20: Uporabniške zgodbe, ki se nanašajo na splošnega uporabnika

Upravljanje občin
Upravljavec lahko spreminja vse podatke o posamezni občini (povezave, numerične podatke, tekst).

Upravljanje tematskih člankov
Upravljavec lahko popravlja podatke tematskih člankov (povezave, tabele, numerične podatke, tekst).

Dodajanje tematskih člankov
Upravljavec lahko dodaja tematske članke.

Slika 21: Uporabniške zgodbe, ki se nanašajo na upravljavca

Absolutni podatki o regiji
Uporabnik lahko pridobi absolutne podatke o regiji, ki so seštevek vseh podatkov o občinah v regiji.

Tiskana oblika
Uporabnik lahko pridobi tiskano obliko spletne strani v PDF obliki.

Absolutni podatki med občinami
Uporabnik lahko primerja vrednosti absolutnih podatkov med posameznimi občinami.

Komentarji
Uporabnik lahko dodaja komentarje pod posamezne strani s podatki o občinah.

Ocena članka
Uporabnik lahko oceni izbran tematski članek.

Slika 22: Uporabniške zgodbe, ki jih lahko implementiramo kasneje

Prikaz
Spletišče se mora pravilno prikazovati v tistih verzijah brskalnikov, ki so bili izdani v roku zadnjih dveh let. **Omejitev**

Baza podatkov
Spletišče mora za ozadje uporabljati obstoječo bazo podatkov. **Omejitev**

Več jezikovnih različic
Spletišče mora biti izdelano tako, da je možno dodati novo različico jezika. **Omejitev**

Slika 23: Definirane omejitve pri razvoju

5.2.2 Uporabniške zgodbe po iteracijah

Glede na izbrane uporabniške zgodbe smo se odločili, da bomo zgodbe implementirali po iteracijah. Najprej smo implementirali tiste zgodbe, ki omogočajo osnovno prepoznavnost spletne strani, nato pa zgodbe, ki omogočajo dodatne funkcionalnosti na spletni strani. V posamezni iteraciji mora biti uporabniška zgodba dokončno razvita in pripravljena za produkcijo. Ko bodo vse iteracije končane in uporabniške zgodbe stestirane, lahko produkt predamo v produkcijo.

Uporabniške zgodbe po iteracijah do prve izdaje produkta v produkcijsko okolje:

- Prva iteracija:
  - Uporabnik lahko s spletišča prebere različne statistične podatke o občini (grafično, semantično, numerično).
  - Širina spletišča se spreminja dinamično.
  - Uporabnik lahko do posameznih strani dostopa preko navigacijskega drevesa.
  - Upravljavec lahko spreminja vse podatke o posamezni občini (povezave, numerične podatke, tekst).

- Druga iteracija:
  - Uporabnik lahko s spletišča prebere tematske članke, ki se povezujejo na spletno stran s podatki.
  - Uporabnik lahko posamezne podatke o občinah poišče s pomočjo iskalnika.
  - Upravljavec lahko popravlja podatke tematskih člankov (povezave, tabele, numerične podatke, tekst).
  - Upravljavec lahko dodaja tematske članke.

Ker smo bili časovno omejeni, obenem pa smo želeli izdati vidno delujočo spletno stran čim prej, smo se odločili, da bomo nekatere zgodbe prestavili v naslednjo izdajo. Na naslednjo izdajo so tako počakale naslednje uporabniške zgodbe:

  - Uporabnik lahko primerja absolutne podatke med posameznimi občinami.
  - Uporabnik lahko pridobi tiskano obliko spletne strani v PDF obliki.
  - Uporabnik lahko dodaja komentarje pod posamezne strani s podatki o občinah.
  - Uporabnik lahko oceni izbran tematski članek.

Po razdelitvi uporabniških zgodb v izdaje in iteracije smo določili rok za izvedbo prve izdaje. Dogovorili smo se, da je rok za izvedbo prve izdaje dva meseca. V drugi izdaji lahko glede na časovne omejitve in potrebe uporabnikov določene zgodbe odvzamemo ali dodamo. Predvideni rok za izvedbo druge izdaje je mesec dni po prvi izdaji.

Glede na omejitve smo se dogovorili, da bo uporabniški vmesnik uporabljal standarde, ki bodo skladni s standardi, ki so bili v veljavi zadnji dve leti. Prav tako bo zasnova spletne strani strukturirana tako, da bo omogočala preprosto dodajanje novih jezikovnih različic. Največjo težavo je predstavljala omejitev, ki zahteva uporabo obstoječe baze podatkov. Posamezni podatkovni tipi in njihova poimenovanja so tako že definirani, kar pa ne omogoča avtomatske definicije tipov glede na uporabniške zgodbe in sprejemne teste.

5.2.3 Sprejemni testi

Po določitvi iteracij in rokov za izvedbo smo se lotili pisanja sprejemnih testov za posamezno uporabniško zgodbo. S sprejemni testi smo poizkušali pokriti vse možne oblike prikazov in dogodkov na spletni strani. Različne uporabniške zgodbe so zahtevale različno število sprejemnih testov.

V okviru magistrske naloge smo zajeli samo razvoj za uporabniške zgodbe, ki se nanašajo na splošnega uporabnika in izpustili tiste, ki se nanašajo na upravljavca. Zaradi različne oblike testiranja uporabniške zgodbe pa smo zajeli tudi zgodbo »Absolutni podatki med občinami«, ki je bila sicer izbrana kot zgodba, ki bi jo lahko implementirali kasneje.

Za izbrane uporabniške zgodbe smo napisali naslednje sprejemne teste:

- **Sprejemni testi za uporabniško zgodbo »Občine«:**
  - Na strani s podatki o občinah sta prikazana šifra in ime občine.
  - Na strani s podatki o občinah sta prikazana šifra in ime regije, ki ji občina pripada.
  - Na strani s podatki o občinah sta prikazana šifra in ime upravne enote, ki ji občina pripada.
  - Na strani s podatki o občinah je prikazan krajši tekst z opisom občine.
  - Na strani s podatki o občinah je prikazan tekst s semantično razlago podatkov o občini.
  - Na strani s podatki o občinah je prikazana slika občine z opisom in virom.
  - Na strani s podatki o občinah je prikazana populacijska piramida.
  - Na strani s podatki o občinah so numerično prikazani absolutni podatki o občini.
  - Uporabnik lahko s pomočjo klika odpre okno, kjer so prikazani kazalniki o občini.
  - Uporabnik lahko dinamično spreminja velikost pisave.

- **Sprejemni testi za uporabniško zgodbo »Tematski članki«:**
  - Na strani, ki predstavlja tematski članek, je prikazana vsebina članka.
  - Na strani, ki predstavlja tematski članek, je lahko prikazana poljubna tabela s

podatki.
    - Na strani, ki predstavlja tematski članek, je lahko prikazan tematski zemljevid.
    - Na strani, ki predstavlja tematski članek, so vsebovani vsebinsko sorodni članki, ki so povezani preko tematskega področja.

- **Sprejemni testi za uporabniško zgodbo »Iskalnik«:**
    - Če v iskalnik vnesemo besedo, ki je zajeta v besedni zvezi občine, iskalnik vrže enega ali več zadetkov, ki zadostuje temu pogoju, npr: Loka.
    - Če v iskalnik napišemo besedo, ki je vsebovana v naslovu tematskega članka, iskalnik vrne »Iskanje ni vrnilo rezultatov«.
    - Če v iskalnik vnesemo besedo, ki ni zajeta v besedni zvezi občine, iskalnik vrne »Iskanje ni vrnilo rezultatov«.

- **Sprejemni testi za uporabniško zgodbo »Navigacijsko drevo«:**
    - Navigacijsko drevo združuje občine po regijah, posamezna občina lahko pripada samo eni regiji.

- **Sprejemni testi za uporabniško zgodbo »Dinamična stran«:**
    - Širina uporabniškega vmesnika se spreminja dinamično glede na širino okna brskalnika.

- **Sprejemni testi za uporabniško zgodbo »Uporabnik lahko primerja podatke med posameznimi občinami s pomočjo tabel«:**
    - Sistem vrne razliko absolutnih vrednosti med izbrano in referenčno občino za absolutne podatke o občinah.
    - Če sta občini enaki, sistem vrne vrednost 0.
    - Če primerjamo absolutne podatke o občini z absolutnimi podatki o Sloveniji, sistem vrne vrednost 0.

5.2.4 Uporaba orodja Fitnesse

Orodje Fitnesse omogoča avtomatizacijo testiranja uporabniških testov s pomočjo »Wiki« oblike, kjer so testi napisani v obliki, ki je razumljiva tudi netehničnim osebam. Poleg »Wiki« oblike testi za delovanje potrebujejo tudi povezavo opisnega dela testa s programsko kodo. Testna povezava je napisana v programskem jeziku in je tako razumljiva samo tehničnim osebam oziroma razvijalcem.

Večino sprejemnih testov smo želeli avtomatizirati, vendar nekaterih zaradi preveč vizualne narave nismo uspeli. Tipična predstavnika takih testov sta testa, ki pripadata uporabniškima

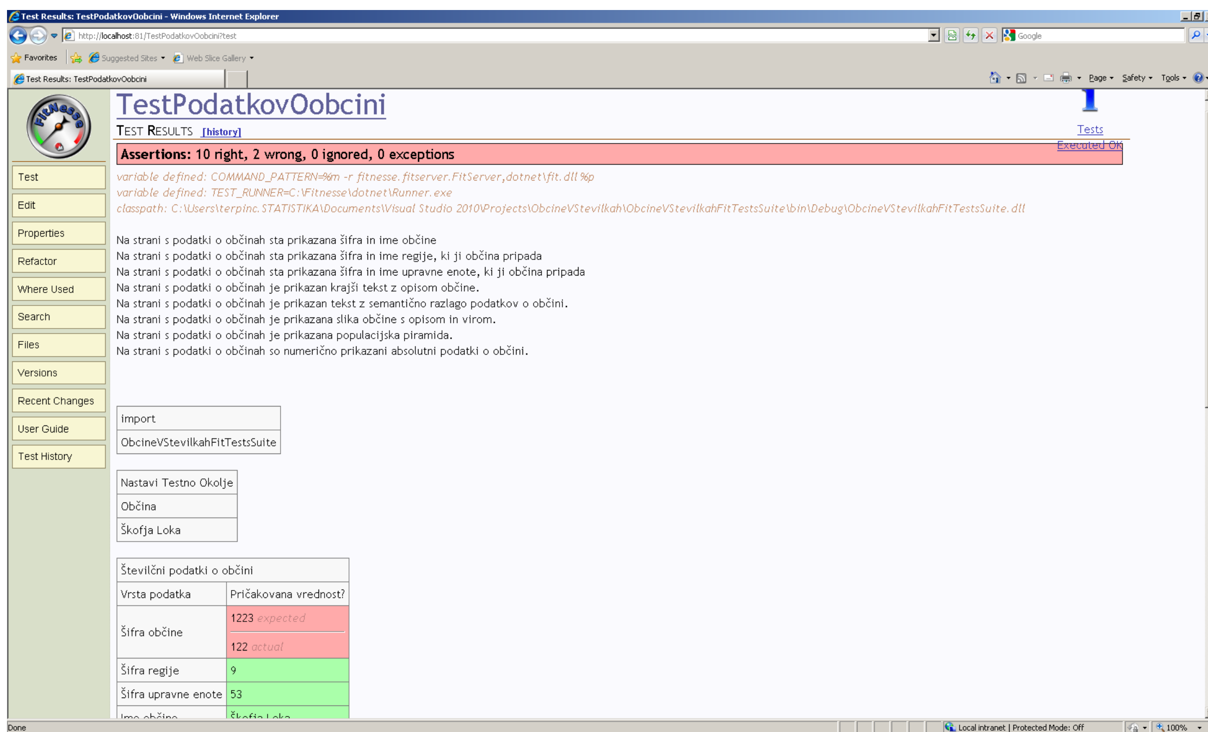
zgodbama »navigacijsko drevo« in »dinamična stran«. Testa bi bilo verjetno možno avtomatizirati s pomočjo orodij za avtomatizacijo testov uporabniškega vmesnika, vendar to presega okvire te magistrske naloge.

Sprejemne teste smo vpisali v Wiki vmesnik Fitnesse in jih tako pretvorili v tabelarično obliko. Primer testa je prikazan na Sliki 24. Na zgornjem nivoju smo definirali tri teste, ki jih ločimo glede na vsebino testiranja. Ti testi so:

- Test pravilnih podatkov o občini (TestPodatkovOobcini) – test je namenjen definiciji in preverjanju podatkov o posamezni občini. Ker so podatki vneseni v bazo ročno, je potrebno pred vnosom preveriti vsebinsko pravilnost podatkov. Preverja se pravilnost povezave med različnimi polji, npr.: šifra občine, ime občine, ime upravne enote, opis občine itd. Test bolj kot pravilnost vsebine testira pravilnost povezovanja med imeni podatkovnih spremenljivk programske kode in imeni podatkovnih polj v bazi podatkov. Ker ima aplikacija v ozadju veliko število podatkov – 210 občin, bi v primeru, da bi želeli testirati vsebinsko pravilnost vseh podatkov, morali napisati 210 testov, za vsako občino posebej. To pa predstavlja preveliko in nepotrebno porabo časa, zato smo napisali teste samo za izbrane občine.

- Test iskanja občin (TestIskanjeObcine) – test je namenjen definiciji in preverjanju funkcionalnosti ob iskanju posameznih občin. Ker bi bilo število testov za preverjanje vseh iskanj zelo veliko, smo definirani tri teste, ki zajemajo: iskanje po delu imena občine, iskanje po celotnem imenu občine in iskanje občine, ki ne obstaja. Na podlagi teh treh testov smatramo, da iskanje deluje pravilno.

- Test primerjanja absolutnih vrednosti (TestPrimerjanjeVrednosti) – test je namenjen definiciji in testiranju funkcionalnosti, ki nam omogoča primerjavo podatkov med dvema izbranima občinama. Test je implementiran izključno v okviru magistrske naloge. S testom preverjamo pravilnost razlike posameznih absolutnih podatkov o občinah (površine, števila žensk, števila moških itd.).

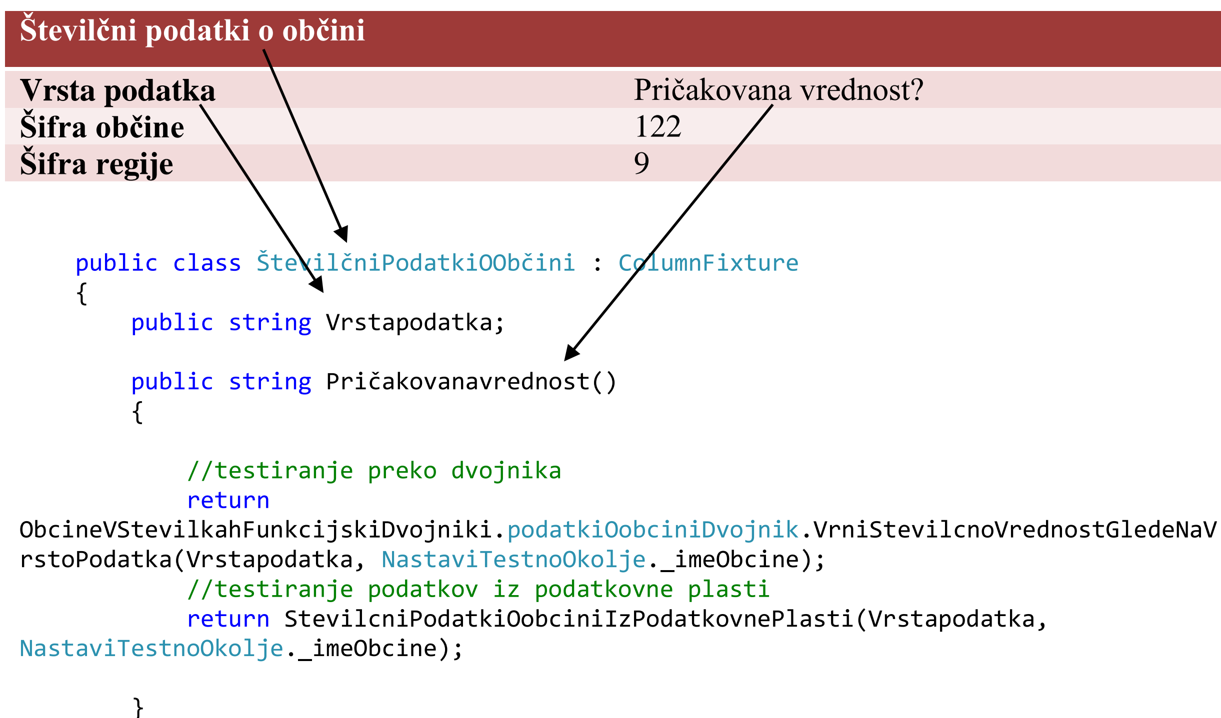


Slika 24: Primer Fitnesse testa – test pravilnih podatkov o občini

5.2.5 Izdelava testne povezave in slepih funkcionalnosti

Fitnesse poveže opisne dele testov s programersko kodo s pomočjo različnih načinov vzpostavitve okolja. V primeru testiranja številčnih podatkov o občini smo uporabili stolpčno okolje, ki omogoča povezavo s pomočjo podatkov, katerih vrednost se spreminja v okviru stolpca tabele.

Kot je prikazano v Tabeli 5, se zgornja vrstica »Številčni podatki o občini« preslika v ime razreda, »Vrsta podatka« v spremenljivko razreda, »Pričakovana vrednost?« pa v metodo razreda. Vsaka vrstica, ki sledi začetnim vrsticam, se preverja preko metode »Pričakovana vrednost«, ki ima spremenljivko »Vrsta podatka« kot vhodni parameter. V primeru, da je pričakovana vrednost enaka vrednosti, ki je definirana v vrstici stolpca »Pričakovana vrednost«, potem test preživi, v nasprotnem primeru pa test pade.



Številčni podatki o občini	
Vrsta podatka	Pričakovana vrednost?
Šifra občine	122
Šifra regije	9

```
    public class ŠtevilčniPodatkiOObčini : ColumnFixture
    {
        public string Vrstapodatka;

        public string Pričakovanavrednost()
        {

            //testiranje preko dvojnika
            return
ObcineVStevilkahFunkcijskiDvojniki.podatkiOobciniDvojnik.VrniStevilcnoVrednostGledeNaV
rstoPodatka(Vrstapodatka, NastaviTestnoOkolje._imeObcine);
            //testiranje podatkov iz podatkovne plasti
            return StevilcniPodatkiOobciniIzPodatkovnePlasti(Vrstapodatka,
NastaviTestnoOkolje._imeObcine);

        }
```

Tabela 5: Povezava Fitnesse testa in testnega okolja

Postopek smo nadaljevali s pisanjem povezav sprejemnih testov s programsko kodo za teste, ki smo jih definirali v prvi iteraciji. Za izbran sprejemni test smo napisali povezavo, vendar je v tej stopnji še nismo imeli kam povezati, saj funkcionalnost, ki bi jo testirali, še ni bila razvita. Zato smo se lotili naslednjega koraka pisanja slepih funkcionalnosti.

V definiciji testov je točno določeno, kaj naj bi posamezni test testiral, zato ni bilo težko napisati kode za slepe funkcionalnosti, saj so bile te dejansko že definirane v tekstovni obliki testa. Pri pisanju slepih funkcionalnosti, smo se držali pravila, da napišemo samo toliko kode, da sprejemni test preživi.

Slepa funkcionalnost dejansko izvaja to, kar je v testu določeno. Primer kode slepe funkcionalnosti, ki vrača podatke o občini, je predstavljena na Sliki 25. Koda vrača vrednosti posameznih vnosov glede na vrsto podatka. Funkcionalnost je implementirana samo za eno občino, zato je lažja kot končno implementirana funkcionalnost, ki bo vračala vrednosti za vseh 210 občin.

```
public class podatkiOobciniDvojnik
    {

        public static string VrniStevilcnoVrednostGledeNaVrstoPodatka(string
Vrstapodatka, string ImeObcine)
        {
            if (ImeObcine == "Škofja Loka")
            {
                switch (Vrstapodatka)
                {
                    case "Šifra občine":
                        return "122";
                    case "Šifra regije":
                        return "9";
                    case "Šifra upravne enote":
                        return "53";
                    case "Ime občine":
                        return "Škofja Loka";
                    case "Ime regije":
                        return "Gorenjska";
                    case "Ime upravne enote":
                        return "Škofja Loka";
                    default:
                        return String.Empty;
                }
            }
       }
```

Slika 25: Programska koda slepe funkcionalnosti, ki vrača podatke o občini

Slepa funkcionalnost, ki vrača razliko absolutnih vrednosti je predstavljena na Sliki 26. Funkcionalnost je sestavljena iz dveh metod. Prva metoda nam vrača dejansko absolutno vrednost razlike dveh števil, druga metoda pa nam, glede na vrsto podatka, referenčno ter iskano občino, vrne dejansko absolutno razliko podatkov.

```
public class PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik
    {

        public string referencnaObcina { get; set; }
        public string iskanaObcina { get; set; }

        public string VrniAbsolutnoVrednost(string VrstaPodatka, string
ReferencniPodatek, string IskaniPodatek)
        {
            String retString = String.Empty;
            List<Tip_PodatkiOObcini> podatki = new List<Tip_PodatkiOObcini>();
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Površina km2",
ReferencniPodatek = 146, IskaniPodatek = 151 });
```

```
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število prebivalcev",
ReferencniPodatek = 22667, IskaniPodatek = 54188 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število moških",
ReferencniPodatek = 11160, IskaniPodatek = 26631 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število žensk",
ReferencniPodatek = 11507, IskaniPodatek = 27557 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Naravni prirast",
ReferencniPodatek = 82, IskaniPodatek = 213 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Skupni prirast",
ReferencniPodatek = 235, IskaniPodatek = 652 });


            if (referencnaObcina == "Škofja Loka" && iskanaObcina == "Kranj")
            {
                var query = from v in podatki
                            where
v.VrstaPodatka.Trim().ToLowerInvariant().Contains(VrstaPodatka.Trim().ToLowerInvariant
())
                            select v;

                Tip_PodatkiOObcini podObcina = query.Single();

                int diff = VrniAbsolutnoVrednostRazlike(podObcina.IskaniPodatek,
podObcina.ReferencniPodatek);

                retString = diff.ToString();
            }
            return retString;
        }

        public int VrniAbsolutnoVrednostRazlike(int IskaniPodatek, int
ReferencniPodatek)
        {
            return  Math.Abs(IskaniPodatek - ReferencniPodatek);
        }
    }
```

Slika 26: Programska koda slepe funkcionalnosti, ki vrača absolutno vrednost razlike

Ko smo napisali vse slepe funkcionalnosti in so vsi sprejemni testi preživeli, smo slepe funkcionalnosti uporabili za pisanje testov enot. Ker testi enot dejansko testirajo to, kar slepe funkcionalnosti izvajajo, smo večino kode za teste prekopirali in preuredili tako, da preverjajo delovanje. Tako smo dobili teste enot na najvišjem nivoju. Teste na najvišjem nivoju smo najprej napisali tako, da so testirali celoto, potem pa smo jih razdelili na več manjših testov enot.

Primer testa enote na najvišjem nivoju, ki testira absolutno razliko podatkov med občinama Škofja Loka in Kranj je prikazan na Sliki 27.

```
[TestMethod]
        public void TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranj()
        {
```

```
            String retString = String.Empty;
            List<Tip_PodatkiOObcini> podatki = new List<Tip_PodatkiOObcini>();
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Površina km2",
ReferencniPodatek = 146, IskaniPodatek = 151 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število prebivalcev",
ReferencniPodatek = 22667, IskaniPodatek = 54188 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število moških",
ReferencniPodatek = 11160, IskaniPodatek = 26631 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Število žensk",
ReferencniPodatek = 11507, IskaniPodatek = 27557 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Naravni prirast",
ReferencniPodatek = 82, IskaniPodatek = 213 });
            podatki.Add(new Tip_PodatkiOObcini { VrstaPodatka = "Skupni prirast",
ReferencniPodatek = 235, IskaniPodatek = 652 });

            //testiranje testnega dvojnika
            PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik instancaDvojnika = new
PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik();
            instancaDvojnika.iskanaObcina = "Kranj";
            instancaDvojnika.referencnaObcina = "Škofja Loka";
            foreach (Tip_PodatkiOObcini obcinaPodatek in podatki)
            {
                int pricakovan = Math.Abs(obcinaPodatek.ReferencniPodatek -
obcinaPodatek.IskaniPodatek);
                string pravi =
instancaDvojnika.VrniAbsolutnoVrednost(obcinaPodatek.VrstaPodatka,
                    obcinaPodatek.ReferencniPodatek.ToString(),
obcinaPodatek.IskaniPodatek.ToString());
                Assert.AreEqual(pricakovan.ToString(), pravi);
            }
```

Slika 27: Programska koda testa enote, ki vrača absolutno vrednost razlike

Test skupaj v eni enoti testira vse vrste podatkov. Če eden od podatkov nima pričakovane vrednosti, potem test pade. Test smo potem razdelili na več manjših testov enot, kjer posamezen test predstavlja test za posamezno vrsto podatka. Manjše teste smo najprej testirali posebej, potem pa smo jih zaradi preglednosti združili v enega.

Primer kode, kjer je test sestavljen iz več manjših testov enote, je prikazan na Sliki 28. Test preverja vsebinsko pravilnost podatkov.

```
public void TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranj()
        {

            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjPovrsina();
            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjSteviloPrebivalcev();
            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjSteviloMoskih();
            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjSteviloZensk();
            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjNaravniPrirast();
            TestEnote_AbsolutnaRazlikaPodatkovSkofjaLokaKranjSkupniiPrirast();
        }
```

Slika 28: Primer kode, kjer je test sestavljen iz več manjših testov enote

Poleg testov, ki preverjajo vsebino, smo napisali tudi teste, ki preverjajo pravilnost delovanja metode »VrniAbsolutnoVrednostRazlike«. Za testiranje te metode smo napisali štiri teste:

- Test, ki testira pravilno delovanje v primeru, da sta obe vrednosti pozitivni.
- Test, ki testira pravilno delovanje v primeru, da sta obe vrednosti negativni.
- Test, ki testira pravilno delovanje v primeru, da sta je ena vrednost pozitivna, druga pa negativna.
- Test, ki testira pravilno delovanje v primeru, da sta obe vrednosti nič.

Primera testov enot, ki testirata metodo »VrniAbsolutnoVrednostRazlike«, sta predstavljena na Sliki 29.

```
        [TestMethod]
        public void TestEnote_VrniAbsolutnoVrednostRazlikePozitivnaStevila()
        {
            PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik instancaDvojnika = new
PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik();
            int vrednost1 = 12;
            int vrednost2 = 15;
            int pricakovanaVrednost = 3;
            int pravaVrednost =
instancaDvojnika.VrniAbsolutnoVrednostRazlike(vrednost1, vrednost2);
            Assert.AreEqual(pricakovanaVrednost, pricakovanaVrednost);
        }

[TestMethod]
        public void TestEnote_VrniAbsolutnoVrednostRazlikeNegativnaStevila()
        {
            PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik instancaDvojnika = new
PrimerjavaAbsolutnihStatisticnihPodatkovDvojnik();
            int vrednost1 = -12;
            int vrednost2 = -15;
            int pricakovanaVrednost = 3;
            int pravaVrednost =
instancaDvojnika.VrniAbsolutnoVrednostRazlike(vrednost1, vrednost2);
            Assert.AreEqual(pricakovanaVrednost, pravaVrednost);
        }
```

Slika 29: Primera testov funkcionalnosti, ki testirata metodo »VrniAbsolutnoVrednostRazlike«

Ko so bili vsi testi enot napisani in so preživeli, smo te teste enot odklopili od slepih funkcionalnosti in jih uporabili za pisanje dejanskih funkcionalnosti. Dejanske implementacije funkcionalnosti, ki so v našem projektu predstavljale predvsem povezave med bazo podatkov in plastjo poslovne logike, smo napisali do te mere, da so vsi testi enot preživeli.

Primer dejanske implementacije metode, ki vrača absolutno razliko podatkov za posamezno vrsto podatka je prikazan na Sliki 30.

```
        public int PridobiAbsolutnoRazlikoMedObcinamaZaVrstoPodatka(string
iskanaObcina, string referencnaObcina, string imePolja)
        {

            PodatkiOObcini PodatOObciniInst = new PodatkiOObcini();
            int idIskaneObcine =
((ObcineVStevilkah_Strani)PodatOObciniInst.PridobiPodatkeOObcini(iskanaObcina)).SifraO
bcine.Value;
            int idReferenceObcine =
((ObcineVStevilkah_Strani)PodatOObciniInst.PridobiPodatkeOObcini(referencnaObcina)).Si
fraObcine.Value;

            Dictionary<string, double> PodatkiIskanaObcina =
GetDataFromTableAbsolutni(idIskaneObcine);
            Dictionary<string, double> PodatkiRefObcina =
GetDataFromTableAbsolutni(idReferenceObcine);

            int retValue = 0 ;
            foreach ( KeyValuePair<string, double> podatek in PodatkiIskanaObcina)
            {
                if (podatek.Key.Contains(imePolja))
                {
                    string kljuc = podatek.Key;
                    retValue =
VrniAbsolutnoVrednostRazlike(Convert.ToInt32(PodatkiIskanaObcina[kljuc]),
Convert.ToInt32(PodatkiRefObcina[kljuc]));
                }
            }
            return retValue;

        }
```

Slika 30: Implementacijska koda, ki vrača absolutno vrednost razlike podatkov

Ko smo implementirali vse funkcionalnosti in so vsi testi enot preživeli, smo na implementacijo preklopili še sprejemne teste. Testi so preživeli, kar je pomenilo, da aplikacija deluje v skladu s tehničnimi in vsebinskimi zahtevami. Naslednji korak je bil izdelava uporabniškega vmesnika, ki je bil zaradi specifičnih zahtev oblikovanja časovno zelo potraten. Kljub veliki časovni obremenitvi z obliko uporabniškega vmesnika, smo prvo iteracijo uspeli dokončati v predvidenem roku. Podrobnosti izdelave uporabniškega vmesnika in implementacij ostalih iteracij presegajo okvire magistrskega dela.

Pri praktični izdelavi spletne strani smo prišli do nekaterih ugotovitev. Priporočljiva je uporaba standardnih vmesnikov. Iz sprejemnih testov najprej definiramo podatkovne tipe, ki bodo v sistemu nastopali. Določeni podatkovni tipi nam predstavljajo osnovo za izgradnjo vmesnikov, preko katerih gradimo razrede. Najbolje je, če se tipi med pisanjem ne spreminjajo.

Priporočljiva je tudi uporaba orodij za injekcijo odvisnosti (dependency injection), saj skozi proces spreminjamo odvisnosti med posameznimi objekti. Na primer testno povezavo sprejemnih testov najprej vežemo na slepo funkcionalnost, nato pa na dejansko

implementacijo funkcionalnosti. V primeru, da sledimo razvoju preko vmesnikov in pravilno definiramo tipe vmesnikov, lahko uporabimo možnost preslikave posameznih delov programske kode s pomočjo avtomatskih generatorjev.

Proces razvoja ne sledi natanko metodologiji testnega razvoja, saj so podatkovni tipi že definirani v podatkih, ki jih hrani obstoječa baza podatkov – uporaba obstoječe baze podatkov je definirana kot omejitev. Ob pravilni uporabi metodologije bi iz sprejemnih testov glede na uporabljene podatke definirali podatkovne tipe ter njim pripadajoče programske vmesnike in jih nato prenesli v podatkovne tipe baze podatkov. Ker so tipi baznih podatkov definirani že v naprej, prihaja v programski kodi do težav, saj skozi proces ne moremo uporabiti standardnih vmesnikov. Pravilno povezovanje podatkovnih tipov s tipi baznih podatkov je predstavljalo dodatno nepotrebno delo, programska koda pa je tako manj razumljiva in bolj nagnjena k napakam.

6 SKLEPNE UGOTOVITE

V magistrskem delu najprej podrobneje raziščemo, kakšne so podobnosti in razlike med sprejemnimi testi in testi enot. Na osnovi podobnosti med njimi definiramo pojem testno vodenega razvoja s sprejemnimi testi. Tak način razvoja temelji na vnaprejšnjem definiranju sprejemnih testov, ki predstavljajo osnovo za nadaljnji razvoj programske opreme s pomočjo principov testno vodenega razvoja. V kontekstu testno vodenega razvoja se je dosedanja praksa nanašala na uporabo testov enot. V magistrskem delu v ta kontekst dodajamo sprejemne teste. Testno voden razvoj s sprejemnimi testi predstavlja zgornjo plast testno vodenega razvoja, torej razvoj, pri katerem se za vodenje razvoja najprej opremo na sprejemne testne uporabniških zgodb.

V kontekst testno vodenega razvoja združimo teste enot in sprejemne teste. Za nastali proces opišemo posamezne korake in aktivnosti, ki jih izvajamo v posameznih korakih. Najprej s pomočjo uporabniških zgodb opišemo delovanje sistema, potem posamezne uporabniške zgodbe točneje opišemo s sprejemnimi testi. Za posamezen sprejemni test potem izvajamo razvoj programske opreme po principih testno vodenega razvoja. S pomočjo sprejemnih testov napišemo teste enot, potem pa na osnovi testov enot napišemo dejansko programsko kodo. Testno voden razvoj na nivoju enot izvajamo toliko časa, da vsi testi enot v okviru izbrane funkcionalnosti preživijo. Delovanje posamezne izbrane funkcionalnosti potem preverimo s sprejemnim testom. Razvoj nadaljujemo s pisanjem testov enot in programske kode za naslednji sprejemni test oziroma funkcionalnost. Postopek izvajamo toliko časa, da sprejemni testi v okviru uporabniške zgodbe preživijo.

Pri izvajanju procesa se soočimo s problemom preverjanja pravilnega delovanja testnega okolja, ki omogoča povezavo med opisnim delom testov in programsko kodo. V primeru, da je testno okolje napisano napačno, sprejemni testi napačno testirajo delovanje izbrane funkcionalnosti. Problem rešimo s pomočjo slepih funkcionalnosti, ki so preprosta implementacija prave kode. Slepe funkcionalnosti napišemo samo do te mere, da zadovoljijo sprejemni test. S pomočjo slepih funkcionalnosti delovanje sprejemnih testov preverimo na začetku razvojnega procesa. Poleg preverjanja delovanja testov pa slepe funkcionalnosti služijo tudi kot programerska dokumentacija. Slepe funkcionalnosti so pravzaprav zrcalni sprejemni testi v programerskem jeziku. S pomočjo tako definiranih slepih funkcionalnosti potem napišemo teste enot, ki nam služijo kot testi za končno pisanje enot programa. Ko so prave funkcionalnosti končane, sprejemne teste priklopimo na prave funkcionalnosti in tako dobimo končno sliko delovanja celote. V končni fazi morajo preživeti tako testi enot kot sprejemni testi. Ko je ta kriterij zadoščen, lahko preidemo na implementacijo naslednje uporabniške zgodbe.

Predstavljen proces razvoja programske opreme povezuje koncepte testno vodenega razvoja in sprejemnih testov ter tako omogoča podporo celotnemu razvojnemu ciklu – od definiranja

zahtev oz. uporabniških zgodb, do končne implementacije in testiranja. Proces razvoja vsebuje vse prednosti agilnega razvoja, saj uporablja kratke iteracije, ki že prikazujejo neko vrednost oz. vizualno sliko za končnega uporabnika. Prednost procesa je tudi dobra dokumentacija. Uporabniške zgodbe in sprejemni testi netehničnemu osebju predstavljajo dokumentacijo o tem, kaj naj bi program izvajal. Slepe funkcionalnosti pa to dokumentirajo s tehničnega vidika, ki je razumljiv programerju. Kot celota pa predlagan proces omogoča poleg avtomatske dokumentacije tudi večji nadzor nad pravilnostjo in skladnostjo razvoja programske opreme.

Slabost predlaganega se kaže predvsem v dodatnem delu, ki ga morajo programerji opraviti, da zadovoljijo potrebe procesa, saj morajo v procesu večkrat napisati podobno programsko kodo, kar za programerja predstavlja odvečno dodatno delo. Pomanjkljivost bi lahko izboljšali s pomočjo avtomatske generacije programske kode, ki bi izluščila podobnosti iz slepih funkcionalnosti in glede na te podrobnosti generirala ogrodje testov enot.

Praktično proces razvoja uporabimo za razvoj spletne strani za potrebe Statističnega urada Republike Slovenije. Spletno stran Občine v številkah razvijemo po principih agilnega razvoja in testno vodenega razvoja s sprejemnimi testi. Spletno stran razvijemo v razmeroma kratkem času. V času pisanja magistrske naloge se izvaja nadgradnja spletne strani, kar kaže, da je dokumentacija dovolj razumljiva, da omogoča preprosto dopolnjevanje tudi razvijalcem, ki pri projektu pred tem niso sodelovali.

7 LITERATURA IN VIRI

7.1 LITERATURA

[1] J. Aarniala, *Acceptance testing. In whitepaper*, University of Helsinki, 2006, 11 str., dostopno na [http://www.cs.helsinki.fi/u/jaarnial/jaarnial-testing.pdf].

[2] A. Abran, J. Moore, *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, IEEE Software, 2004, 204 str.

[3] G. Adzic, *Test Driven .NET Development with FitNesse second edition*, 2009, Neuri Limited, 224 str.

[4] K. Beck, *Test-Driven Development By Example*, Addison Wesley, 2002, 204 str.

[5] K. Beck, C. Andres*, Extreme Programming Explained: Embrace Change* (2nd Edition). Addison-Wesley Professional, 2004, str. 120–151.

[6] B. Boehm, R. Turner*, , Balancing Agility and Discipline: A Guide to the Perplexed.* Addison-Wesley, 2003, 237 str.

[7] M. Cohn, *User Stories Applied: For Agile Software Development*, Addison Wesley, 2004, 211 str.

[8] M. Fowler, *Domain specific languages*, Addison Wesley Professional, 2010, 640 str.

[9] B. Haugset, G. K. Hanssen, *Automated Acceptance testing: a literature review and Industrial test case study*, Agile 2008 Conference, str. 27–38.

[10] J. Highsmith, *Agile Project Management: Creating Innovative Products*, Addison Wesley, 2004, 329 str.

[11] D. Janzen, H. Saidian, *Test-Driven Development, Concepts, Taxonomy and Future Direction*, 2005, IEEE Software, September 2005, str. 43–50.

[12] R. Jeffries, *Essential XP: Card, Conversation, and Confirmation*. XP Magazine August 30, 2001.

[13] R. Jeffries, A. Anderson, C. Hendrickson, *Extreme programming installed*, Addison-Wesley Professional 2000, 265 str.

[14] A. Jennita, *Evnision teh next Generation of Functional testing tools*, 2007, IEEE Software, May/June 2007, str. 58–65.

[15] A. Jennita, *Generative Acceptance Testing For Difficult to test SW*, Extreme Programming and Agile Processes in Software Engineering, Proceedings of 5th International Conference XP 2004, str. 29–37.

[16] T. Koomen, M. Pol, *Test Process Improvement - A Practical Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999, str. 32–77.

[17] L. Koskela, *Test driven, practical TDD and aceptance TDD for Java Developers*, 2008, Manning, 470 str.

[18] T. Mackinnon, S. Freeman, P. Craig, *Endo-Testing*: *Unit Testing with Mock Objects,* eXtreme Programming and Flexible Processes in Software Engineering Conference, 2000.

[19] G. Melnik, G. Meszaros, J. Bach, K. Beck, *Acceptance Test Engineering Guide*, 2009 Microsoft Corporation, str. 33–53.

[20] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, Boston 2007, 833 str.

[21] J. Newkirk, R. C. Martin*: Extreme programming in practice*, Addison – Wesley, 2001, 204 str.

[22] M. Pančur, *Testno voden razvoj v okoljih Java in .Net*, Dnevi Slovenske informatike 2003, Portorož, str. 91–97.

[23] A. Rendell, *Effective and Pragmatic Test Driven Development*, Agile 2008, str. 298–303.

[24] R. O. Rogers, *Acceptance testing vs. unit testing*: A developer's per-spective. ThoughtWorks Technologies, 2004, str. 23–31.

[25] P. Runeson, *A Survey of Unit Testing Practices*, 2006, IEEE Software, July/Avgust 2006, str. 22–29 .

[26] M. Semmann, *Exploring The Continuum Of Test Doubles*, MSDN Magazine, September 2007.

[27] L. Williams, *Agile Software Development Methodologies and Practices*, Advances in Computer, vol. 80, 2010, str. 1–44.

7.2 VIRI

[28] CMS: *Selecting a development approach* [http://www.cms.gov/SystemLifecycleFramework/Downloads/SelectingDevelopmentApproach.pdf], 3. 9. 2010

[29] Fitnesse. Welcome to fitnesse. [http://fitnesse.org], 24. 5. 2011

[30] M. Fowler, Continuous Integration. [http://martinfowler.com/articles/originalContinuousIntegration.html], 29. 8. 2010

[31] S. Hanly, Building your own AT framework [http:// www.xpday.be/html/English/BuildYourOwnAcceptanceTestFramework.ppt], 21. 11. 2003

[32] R. Martens , Ready for Agile. An Introduction to Agile Development. Rally articles. [http://www.rallydev.com/articles/2006/10/ready_for_agile_an_intro.html], 15. 10. 2006

[33] Msdn. A Unit Testing Walkthrough with Visual Studio Team Test [http://msdn.microsoft.com/en-us/library/ms379625%28v=vs.80%29.aspx], 22. 7. 2010

[34] On Software. Unit Testing 101 For Non-Programmers [http://en.wikipedia.org/wiki/Unit_testing], 22. 2. 2011

[35] QSM Associates: Companies endure Delays in Order to obtain Better Results, News & Newsworthy. QSM Associates. [http:// http://www.qsma.com/news_companies_endure.htm], 13. 5. 2004

[36] Software Testing-Testing Life Cycles. [http://www.etestinghub.com/testing_lifecycles.php#2], 3. 3. 2009

[37] The Standish grup, The Chaos report 2009. [http://www.standishgroup.com/newsroom/chaos_2009.php], 24. 5. 2011