Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Knaflič

Vedenjsko voden razvoj programske opreme

diplomsko delo na visokošolskem strokovnem študiju

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Vedenjsko voden razvoj je nadgradnja testno vodenega razvoja, ki v ospredje postavlja uporabnika, kar ima za posledico kvalitetnejši programski izdelek, ki bolje naslavlja dejanske potrebe uporabnika.

V diplomski nalogi preučite in predstavite osnovne prvine vedenjsko vodenega razvoja programske opreme, pri čemer izhajajte iz testno vodenega razvoja. Na konkretnem primeru aplikacije za vodenje osebnih financ predstavite dejansko implementacijo teh prvin. Nalogo zaključite z analizo učinka in primernosti uporabe pristopa v praksi.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Gregor Knaflič, z vpisno številko **63970243**, sem avtor diplomskega dela z naslovom:

Vedenjsko voden razvoj programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 31. avgust 2016 Podpis avtorja:

Zahvaljujem se mentorju viš. pred. dr. Igorju Rožancu za pomoč in svetovanje pri izdelavi diplomske naloge.

Hvaležen sem moji družini, mojim staršem, prijateljem in vsem, ki so me podpirali in spodbujali pri študiju.

Moji družini.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Testiranje enot **3**

2.1 Avtomatizirano testiranje 3

2.2 Koristi testiranja 7

2.3 Testno voden razvoj 9

3 Vedenjsko voden razvoj **13**

3.1 Težave klasičnih projektov 13

3.2 Principi vedenjsko vodenega razvoja 15

3.3 Agilne metode druge generacije 17

3.4 Izvedba vedenjsko vodenega razvoja 19

3.4.1 SMART izid 20

3.4.2 Uporabniške zgodbe 20

3.4.3 BDD cikel 23

4 Primer uporabe vedenjsko vodenega razvoja **25**

4.1 Predstavitev 25

4.2 Zaledni sistem 27

4.3 Testni podatki 30

4.4 Izvedba testov 35

4.5 Izdelava kode 37
4.6 Odjemalec 40

5 Analiza vedenjsko vodenega razvoja **45**
5.1 Prednosti 45
5.2 Slabosti 45
5.3 Priložnosti 46
5.4 Nevarnosti 46

6 Sklepne ugotovitve **47**

Literatura **49**

Seznam uporabljenih kratic

kratica	angleško	slovensko
TDD	Test Driven Development	testno voden razvoj
BDD	Behaviour Driven Development	vedenjsko voden razvoj
AAA	Arrange Act Assert	nastavi, izvedi, potrdi
DRY	Don't Repeat Yourself	ne ponavljaj se
GWT	Given When Then	imamo, ko, potem
SMART	Specific Measurable Achievable	določen, merljiv, dosegljiv
	Relevant Timeboxed	pomemben, časovno omejen
NPM	Node Package Manager	upravljalec paketov za okolje Node
E2E	End-To-End	od začetka do konca
API	Application Programming interface	aplikacijski programski vmesnik
HTML	Hyper Text Markup Language	jezik za označevanje nadbesedila

Povzetek

V diplomi je predstavljen razvoj programske opreme z uporabo vedenjsko vodenega razvoja. Vedenjsko voden razvoj je nadgradnja testno vodenega razvoja in prinaša precejšen zasuk v načinu razmišljanja med razvojem aplikacije v primerjavi s klasičnim pristopom. V ospredje je postavljen uporabnik ter njegov pogled na uporabo aplikacije, zato je končni produkt vedenjsko vodenega razvoja visoko kvalitetna koda in aplikacija, ki služi namenu uporabnika in omogoča poceni vzdrževanje in nadaljni razvoj novih funkcionalnosti. V osrednjem delu je prikazana uporaba vedenjsko vodenega razvoja na praktičnem primeru razvoja zalednega sistema in odjemalca za zaledni sistem aplikacije za vodenje osebnih financ BBSSolvent. V zadnjem delu pa je SWOT analiza vedenjsko vodenega razvoja, njegove prednosti in slabosti.

Ključne besede: agilne metodologije, vedenjsko voden razvoj, testno voden razvoj, C#, AngularJS.

Abstract

This thesis presents application development using Behavior Driven Development. Behavior driven development upgrades the Test Driven Development and requires substantial shift in the way the application is developed. Shift is also required in the way of thinking during the development process in comparison to classical development. Focus is on the user and his view on the application usage, and the end result is high quality code and application that serves the user and enables cheap maintanance and easy implementation of new features. In the main part Behavior Driven Development is demonstrated on example of backend and frontend system of application for personal finances BBSSolvent. In the last part SWOT analysis, advantages and disadvantages of Behavir Driven Development are presented.

Keywords: agile methods, behaviour-driven development, test-driven development, C#, AngularJS.

Poglavje 1

Uvod

Cilj diplomskega dela je predstaviti vedenjsko voden razvoj [1] (angl. Behavior Driven Development - BDD) in prikazati uporabo vedenjsko vodenega razvoja pri razvoju spletne aplikacije. Vedenjsko voden razvoj je zanimiv zaradi tega, ker poizkuša rešiti problem v komunikaciji med razvijalci in interesno skupino s pomočjo komunikacijskih vzorcev, kot so primeri. Primeri so opisani v obliki uporabniških zgodb in scenarijev, ki so razumljivi vsem udeležencem.

Vedenjsko voden razvoj je nadgradnja testno vodenega razvoja, ki je opisan v drugem poglavju, prav tako njegove pomanjkljivosti oziroma problemi, ki jih vedenjsko voden razvoj odpravlja.

Ta vrsta razvoja je bila uporabljena pri razvoju aplikacije za upravljanje osebnih financ BBSSolvent - Become Be Stay Solvent - Postani, bodi, ostani solventen. Aplikacija implementira inovativno, preprosto in učinkovito metodo upravljanja osebnih financ, ki se pri izdatkih osredotoča na dve kategoriji, pri katerih dejansko lahko vplivamo, koliko bomo v posameznem obdobju porabili. Prva kategorija so izdatki za hrano, druga pa so nenujni izdatki, kot so oblačila, obutev, nakit, knjige, kino, koncerti, izleti, obiski kavarn, barov in restavracij ... Izdatki, kot so recimo gorivo, elektrika, telefon, krediti, stanovanjski stroški, vrtec, šola, pri tej metodi ne igrajo nobene vloge, čeprav lahko poberejo precejšnji del naših prihodkov. Na njih v bistvu ne moremo

vplivati, oziroma se ne moremo se odločit, da ne bomo plačali elektrike ali obroka kredita.

Vedenjsko voden razvoj se vodi skozi tako imenovane uporabniške zgodbe in scenarije, ki nam v človeku prijazni obliki z natančno določenim in lahko razumljivim formatom za vse interesne skupine sporočata, kaj naj aplikacija oziroma sistem počne oziroma kako naj se obnaša. Te uporabniške zgodbe in scenariji so v obliki avtomatiziranih testov. Skozi teste zagotovimo, da aplikacija oziroma sistem deluje tako, kot je opisano v uporabniških zgodbah in scenarijih. Ob izvajanju avtomatiziranih vedenjskih testov se v obliki poročila generira tudi živa dokumentacija, ki nam opisuje, kako se uporablja aplikacija oziroma sistem.

Pri vedenjsko vodenem razvoju si pomagamo z orodji in ogrodji kot so BDDfy, Moq, NUnit, NCrunch za zaledni sistem (C Sharp) ter Protractor in Jasmine za odjemalca (AngularJS). Uporabljamo tudi konceptualne vzorce Objekt mati, Graditelj testnih podatkov. Ta orodja, ogrodja in konceptualni vzorci so opisani in prikazani v četrtem poglavju, kjer je predstavljena implementacija z uporabo vedenjsko vodenega razvoja.

Poglavje 2

Testiranje enot

V tem poglavju bomo najprej pogledali podobnosti med ročnim testiranjem in med strukturiranimi, ponovljivimi oziroma avtomatiziranimi testi enot.

Ko bomo pogledali koristi testiranja enot, bomo videli, da so testi enot investicija, kako nam prihranijo čas na dolgi rok, in kako nam pomagajo pri regresijskih testih. Zanesljivo preoblikovanje pa je težko, če ne nemogoče brez testov. Če imamo teste napisane pred preoblikovanjem, občutno zmanjšamo tveganje za napake, testi nam tudi močno olajšajo navzkrižno testiranje brskalnikov. To nas vodi do principov testno vodenega razvoja, ki je zasnovan za proizvajanje čiste kode (angl. clean code) [2], kateri lahko zaupamo.

2.1 Avtomatizirano testiranje

Spletni razvijalci se velikokrat znajdemo v situaciji, kjer porabimo preveč časa z gumbom `Refresh` v brskalniku: napišemo nekaj kode v urejevalnik besedila, nato preklopimo v brskalnik in pritisnemo `F5`. In to ponavljamo v nedogled.

Tovrstno ročno testiranje je časovno potratno, ni ponovljivo, obstaja velika je možnost napake. Ker se za spletne aplikacije pričakuje, da se izvajajo na raznih kombinacijah brskalnikov in operacijskih sistemov, testirati vse skupaj ročno postane nemogoče opravilo.

Avtomatizirano testiranje [3] ponuja rešitev za proces ročnega testiranja. Namesto ponovnega izpolnjevanja tistega obrazca in klikanja `Submit` gumba, da vidimo, če validacija na klientu deluje pravilno, lahko naročimo programski opremi, da izvede ta test namesto nas.

Prednosti so očitne: lahko testiramo več brskalnikov, teste lahko poženemo kasneje ali jih poganjamo periodično brez naše interakcije.

Osnovni način za avtomatiziranje testiranja je uporaba testov enote. V nadaljevanju bomo pogledali, kaj je test enote, kako jih strukturiramo oziroma oblikujemo, kaj je njihovo bistvo, kako preverjamo njihovo uspešnost oziroma neuspešnost.

Test enote

Test enote [4] je del kode, ki testira del produkcijske kode. To stori z vzpostavitvijo enega ali več objektov v znanem stanju, nad njimi nekaj izvede (npr. kliče metodo), in nato pregleda rezultat, ki ga primerja s pričakovanim izidom. Testi enot morajo biti shranjeni na disku in morajo biti enostavno in hitro izvedljivi. Če so testi težko ali počasi izvedljivi, je manjša verjetnost, da jih bodo razvijalci poganjali. Testi enot morajo testirati programske komponente v izolaciji. Tudi teči morajo izolirano - noben test ne sme biti odvisen od drugega testa, tako da testi lahko tečejo istočasno in v poljubnem vrstnem redu. Da bi testirali komponente v izolaciji, je včasih treba objekte, od katerih je komponenta odvisna, nadomestiti z njeno imitacijo (angl. mock object).

Zato imamo teste enot shranjene na disku, ponavadi v sistemu za upravljanje verzij skupaj s produkcijsko kodo. Teste lahko poganjamo kadarkoli:

- ko je implementacija končana, da preverimo pravilno vedenje ali
- ko se spremeni implementacija, da preverimo, da je vedenje ostalo nespremenjeno.

Testi enot morajo biti avtomatizirani in ponovljivi.

Vzorec 'AAA' oz. '3A'

Nastavi-izvedi-potrdi (angl. Arrange-Act-Assert) [5] na sliki 2.1 je vzorec za ureditev in obliko kode v testu enote. Z uporabo tega vzorca je jasno ločeno, kaj se testira od nastavitev oziroma verifikacije. Razkriva nam tudi, če testna metoda poizkuša testirati preveč stvari na enkrat. Vsak test enote naj združi te funkcionalne enote, ki so ločene s prazno vrstico:

- nastavi vse potrebne predpogoje in vhode,
- izvedi nekaj nad objektom oz. metodo in
- potrdi pričakovane rezultate.

```
138         [Test]
139         public void GetCurrentMonthFromDate_Today_FirstOfCurrentMonthReturned()
140         {
141             //Arrange
142             var firstOfCurrentMonth = new DateTime(_today.Year, _today.Month, 1);
143
144             //Act
145             DateTime result = _timeRangeCalculation.GetCurrentMonthFromDate();
146
147             //Assert
148             Assert.AreEqual(firstOfCurrentMonth, result);
149         }
```

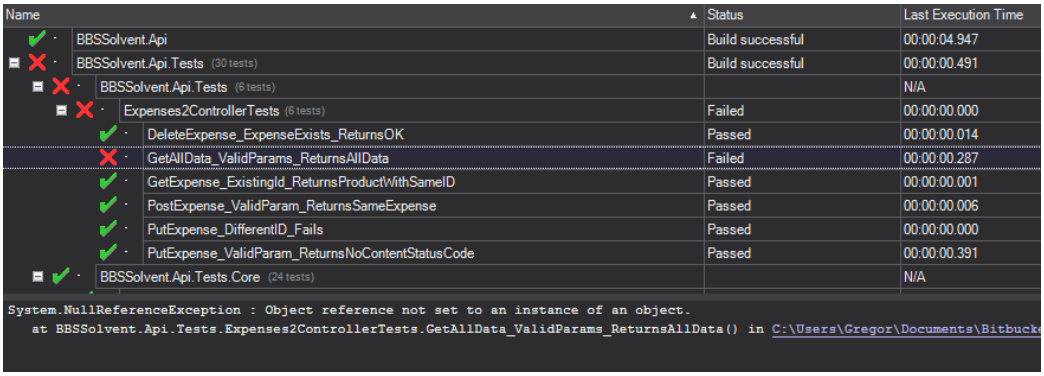
Slika 2.1: Arrange - Act -Assert

Trditev

Bistvo testa enote je trditev (angl. assert). Trditev je predpostavka, ki predvideva razvijalčevo ciljno stanje sistema. Trditve so uporabljene v testih enot za avtomatsko preverjanje teh predpostavk. Kadar trditev ne uspe, se test enote prekine in nas obvesti o neuspehu.

Rdeče in zeleno

V svetu testiranja enot se rdeče in zeleno pogostokrat uporabljata namesto neuspelega in uspelega testa. Če je test izpisan rdeče ali zeleno, je izid jasnejši za interpretacijo, kot nam prikazuje slika 2.2.



Slika 2.2: Rdeče in zeleno

Nastavitev in razstavitev

Ogrodja za testiranje enot običajno vsebujejo metodi za nastavitev (angl. setup) [6] in razstavitev (angl. teardown [7]. Kličeta se pred oz. po izvajanju vsakega testa, in omogočata centralizirano nastavitev testnih podatkov.

Integracijski testi

Integracijske teste in teste enot velikokrat mešamo med sabo. Ponavadi jih lahko ločimo po tem, da preverimo, ali testirajo komponente v izolaciji ali ne.

Predstavljajmo si proizvodnjo avtomobilov. Test enote ustreza preverjanju vsakega posameznega sestavnega dela avtomobila: volan, kolesa, električna stekla, in tako naprej. Integracijski test [8] ustreza preverjanju, da

izdelan avto deluje kot celota, oziroma da manjše skupine enot delujejo po pričakovanjih, kot na primer, da se kolesa obrnejo, ko zasučemo volan. Integracijski test testira skupek enot. V idealnih pogojih so te enote testirane in vemo, da delujejo pravilno v izolaciji. Integracijski testi na višjem nivoju ponavadi potrebujejo posebna orodja, kot so recimo simulatorji brskalnikov. Možno pa je napisati integracijske teste tudi z ogrodji za testiranje enot. V najpreprostejši obliki integracijski test testira dve ali več posamezni komponenti. Najpreprostejši integracijski testi so lahko tako podobni testom enot, da so pogostokrat pomotoma zamenjani z njimi.

2.2 Koristi testiranja

Najbolj pogost očitek testiranju enot je, da vzame preveč časa. Seveda za testiranje naše aplikacije potrebujemo čas. Alternativa avtomatiziranemu testiranju ponavadi ni popolno izogibanje testiranja. Brez testov smo razvijalci prepuščeni ročnemu procesu testiranja, ki je zelo neučinkovit. Avtomatizirano testiranje nam omogoča, da napišemo test enkrat in ga poganjamo tolikokrat, kot je potrebno.

Regresijsko testiranje

Včasih storimo kakšno napako v kodi. Te napake lahko vodijo v hrošče, ki včasih najdejo pot v produkcijo. Še huje, včasih popravimo nekega hrošča, ki se nam ta isti hrošč zopet priplazi na površje v produkciji. Regresijski testi (angl. regression tests) [9] nam pomagajo, da se temu izognemo. S tem, ko ujamemo hrošča v testu, nas bo naš skupek testov obvestil, če se ta hrošč ponovno pojavi. Ker so testi avtomatični in ponovljivi, lahko poženemo teste, preden damo kodo v produkcijo. S tem zagotovimo, da pretekle napake ostanejo v preteklosti. Ko sistem raste v velikosti in zahtevnosti, ročno testiranje nazadovanja hitro postane nemogoče opravilo.

Preoblikovanje kode

Preoblikovanje kode [10] je proces spreminjanja aplikacije na tak način, da se ne spremeni zunanje obnašanje kode, izboljša pa se notranja struktura.

Preoblikovati kodo pomeni spremeniti njeno izvedbo, medtem ko zunanje obnašanje ostane nespremenjeno. Tako kot s testi enot, smo najverjetneje to že počeli, ne glede na to, ali smo temu rekli preoblikovanje kode ali ne. Če smo že kdaj izločili pomožno metodo iz neke druge metode, da bi jo ponovno uporabili v drugih metodah, smo naredili preoblikovanje kode. Tudi preimenovanje objektov ali funkcij je preoblikovanje kode. Preoblikovanje kode je bistveno pri rasti aplikacije, da se ohranja dobra zasnova, da se koda ne ponavlja, in da se je zmožna prilagoditi se spreminjajočim se zahtevam.

Možnih točk okvar pri preoblikovanju kode je precej. Ko se preimenuje metoda, se je potrebno prepričati, da so spremenjene vse reference na to metodo. Če se kopira koda iz metode v pomožno metodo, je treba biti pozoren na podrobnosti, kot so lokalne spremenljivke v originalni implementaciji.

Prvi korak za uspešno preoblikovanje kode so obstoječi testi enot za del kode, ki se preoblikuje. Testi so zanesljiva metrika, ki pove, ali je bilo preoblikovanje uspešno ali ne, ter da v procesu preoblikovanja niso nastali novi hrošči. Kode, ki ni pokrita s testi enot, naj se ne bi preoblikovalo.

Preoblikovanje kode pa je zelo težko, če ne nemogoče, izvesti zanesljivo brez testov.

Navzkrižno testiranje brskalnikov

Razvijalci spletnih aplikacij razvijajo kodo, za katero se pričakuje, da bo tekla na raznih kombinacijah programskih okolij in uporabniških posrednikov. Z uporabo testov enot lahko zelo zmanjšamo trud, potreben za zagotovitev pravilnosti delovanja v različnih okoljih. Testiranje s testi enot pomeni zgolj, da se obstoječi testi poženejo na vseh ciljnih okoljih. Z naprednim izvajalcem testov enot, pri katerem so uporabniški posredniki že pripravljeni, je to enostavna operacija.

Ostale koristi

Dobro napisani testi lahko služijo tudi kot dobra dokumentacija za enote, katere pokrivajo. Kratki in osredotočeni testi enot lahko pomagajo novim razvijalcem, da s preučevanjem testov hitro spoznajo sistem, ki se razvija. Testi enot tudi pomagajo razvijalcem pisati bolj čiste vmesnike, ker jih zaradi testov uporabijo takoj, ko jih napišejo, in takoj dobijo povratno informacijo.

Predpogoji za koristi

Če želimo pisati zares dobre teste enot, mora biti koda primerna za testiranje [11]. Če se kdaj znajdemo v situaciji, da moramo napisati teste enot za obstoječo aplikacijo, ki ni bila napisana z mislijo na testiranje, bomo nenehno odkrivali, da so deli aplikacije velik, če ne celo nemogoč izziv za testiranje. Testiranje enot v izolaciji pomaga razkriti preveč tesno povezano kodo in spodbuja ločevanje zadev. Ko vidimo koristi testiranja enot, vidimo, da je to investicija, da nam testi prihranijo čas na dolgi rok, pomagajo nam pri regresijskih testih.

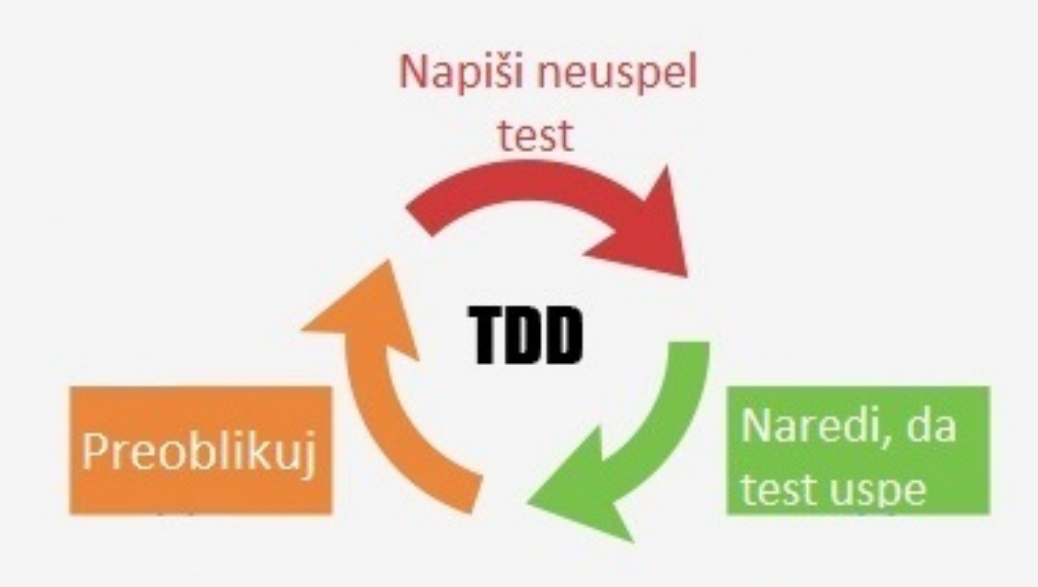
2.3 Testno voden razvoj

Postopek

Testno voden razvoj (angl. Test Driven Development - TDD) je praksa razvijalcev, pri kateri so testi enot napisani pred izdelavo kode, ki se testira [12]. Začnemo z zelo majhnim testom za kodo, ki še ne obstaja, nato napišemo ravno dovolj kode, da test uspe. Ko test uspe, pregledamo nastalo zasnovo, in preoblikujemo kodo, če opazimo kakšno podvojenost. Potem dokumentiramo naslednjo zahtevo v obliki naslednjega testa. Ko test poženemo, ne uspe, zato napišemo ravno dovolj kode, da uspe, pregledamo

zasnovo, odstranimo podvojenosti. Nadaljujemo na enak način. Postopek je prikazan na sliki 2.3.

V večini sistemov za testiranje vidimo neuspel test izpisan z rdečo barvo, ko pa test uspe, je izpisan z zeleno. Zaradi tega se ta cikel pogosto imenuje rdeče-zeleno-preoblikuj (angl. Red-Green-Refactor) [13].



Slika 2.3: TDD Cikel

Nastajajoča zasnova

Ko količina kode narašča, se vedno več pozornosti namenja koraku preoblikovanja kode. Zasnova se neprestano razvija in je podvržena stalni reviziji, čeprav ni vnaprej določena. Nastajajoča zasnova je eden najpomembnejših stranskih produktov testno vodenega razvoja.

Težava testno vodenega razvoja

Ideja testiranja enot pogosto zapelje razvijalce, da preverjajo interno strukturo oz. implementacijo objekta, kar ustvari nepotrebno odvisnost. Ta odvisnost pomeni, če se bo spremenila struktura, bo test neuspešen, čeprav se vedenje objekta ni spremenilo. Taka krhkost lahko naredi skupke testov precej dražje za vzdrževanje, in je glavni razlog, da se skupki testov začno ignorirati in se nazadnje celo zavržejo.

Poglavje 3

Vedenjsko voden razvoj

Ker testiranje interne strukture objektov ni učinkovito na dolgi rok, se moramo osredotočiti na nekaj drugega. Problem pri testiranju interne strukture objekta je, da testiramo, kaj objekt je, namesto da bi testirali, kaj počne. Slednje je namreč veliko bolj pomembno.

Podobno je na nivoju aplikacije. Interesnih skupin (angl. stakeholders) [14] ponavadi ne zanima, da so podatki shranjeni v relacijski bazi. Njim je važno, da so "v bazi", kar za njih pomeni, da so varno shranjeni nekje, kjer lahko učinkovito pridejo od njih.

Vedenjsko voden razvoj, ki se je začel kot nadgradnja testno vodenega razvoja, je zrasel v svojo metodologijo. V tem poglavju bomo pogledali osnove vedenjsko vodenega razvoja, njegove principe ter uporabo s pomočjo SMART izidov, uporabniških zgodb in BDD cikla.

3.1 Težave klasičnih projektov

Tradicionalni projekti ne uspejo zaradi različnih razlogov. V nadaljevanju bomo pogledali nekaj razlogov, kako in zakaj ti projekti ne uspejo.

Zamuda pri dobavi ali dobava preko predvidenega proračuna

Ocenjujemo, planiramo in računamo na nepredvidene dogodke, in potem se na naše razočaranje zgodi resnično življenje. Ko prvič zamudimo rok, ni nikomur preveč mar. Če pa se zamujanje nadaljuje dovolj dolgo, iz tedna v teden, iz meseca v mesec, vmes ljudje odhajajo in prihajajo, lahko projekt pokopljemo. Leto in pol do dveh let je ponavadi dovolj [15].

Dobava nepravih stvari

Večina uporablja programsko opremo, ki je bila narejena z zamudo in preko predvidenega proračuna. Deloma so to razlogi, da se sistemi neprestano samodejno posodabljajo s popravki in novimi funkcionalnostmi v obliki servisnih paketov in sistemskih posodobitev, na spletna mesta, ki dodajajo nove funkcionalnosti skozi čas.

Ampak nihče od nas ne uporablja programske opreme, ki ne rešuje problema, ki ga imamo.

Presenetljivo je, koliko napora pri vodenju projekta je posvečeno temu, da bo projekt končan pravočasno in znotraj proračuna, čeprav je programska oprema narejeno z zamudo neizmerno bolj uporabno kot neprava programska oprema [15].

Nestabilnost v produkciji

Projekt je bil končan pravočasno in znotraj proračuna, uporabnikom je všeč, zato ga damo v produkcijo. Problem pa je, ker se sesuje dvakrat dnevno. Mislimo, da je težava povezana z memorijo ali konfiguracijo ali infrastrukturo. Ne vemo, kaj povzroča težave, razen da nam je neprijetno in da nas stane veliko denarja. Če bi le več časa posvetili testiranju. Ljudje ga bodo preizkusili enkrat in obupali, če se bo sesuval [15].

Drago vzdrževanje

Na veliko stvari ni treba biti pozoren, če pišemo programsko opremo za enkratno uporabo. Vzdrževanje je ena od teh. Če pričakujemo, da bo prvi izdaji sledila druga, tretja in tako naprej, potem moramo paziti, da ne pridemo v situacijo, da bodo razvijalci imeli več dela z vzdrževanjem kot z razvijanjem novih funkcionalnosti. V neki točki bo prišlo do tega, da je večji strošek vzdrževanja kot pa prihodek.

3.2 Principi vedenjsko vodenega razvoja

Vse je obnašanje

Vedenjsko voden razvoj (angl. behavior driven development) [1] prestavi poudarek na obnašanje namesto na strukturo, in to naredi na vseh nivojih razvoja. Objekt, ki računa razdaljo med dvema mestoma, nek drug objekt, ki delegira iskanje zunanjemu servisu in zaslon, ki nudi povratno informacijo ob nepravilnem vnosu, vse to je obnašanje.

Ko se začnemo zavedati tega, se spremeni naš pogled na vodenje razvoja. Začnemo razmišljati bolj o interakcijah med osebami in sistemi ali objekti, kot pa o strukturi teh objektov.

Prave besede

Večina težav, s katerimi se sooča razvijalska ekipa, je povezanih s komunikacijo. Vedenjsko voden razvoj poskuša s poenostavljenim besednjakom opisati scenarije, v katerih se bo programje uporabljalo: `Dan je` (angl. given) nek kontekst, `Ko` (angl. when) se nekaj zgodi, `Potem` (angl. then) pričakujem nek rezultat.

`Dan je, Ko, Potem`, BDD trojček, so preproste besede, ki jih uporabljamo, ko govorimo o vedenju aplikacije ali vedenju objekta. Zlahka jih razumejo poslovni analitiki, testerji in razvijalci.

Preoblikovanje

Preoblikovanje je proces spreminjanja programskega sistema na tak način, da se ne spremeni zunanje obnašanje kode, izboljša pa se notranja struktura [10].

Da bi to zagotovili, poganjamo vedenjske teste med vsako spremembo. Če se izvedejo uspešno, pomeni, da smo preoblikovanje uspešno opravili. Če je kakšen test neuspešen, vemo, da je zadnja sprememba, ki smo jo naredili, povzročila to težavo. Tako lahko hitro prepoznamo in rešimo nastalo težavo da pridemo nazaj v zeleno, in nato poskusimo znova.

Ne preveč in ne premalo

Vnaprejšnje planiranje, analiziranje in načrtovanje, vse to nam na nek način zmanjšuje končni rezultat. Ne bi smeli narediti manj kot je potrebno za začetek, ampak karkoli več od tega je tudi zapravljen trud. To velja prav tako za avtomatiziranje procesov. Imejmo avtomatizirano prevajanje, grajenje in postavitev, ne avtomatizirajmo pa vsega.

Doprinos dodane vrednosti interesnim skupinam

Če delamo nekaj, kar ne prinaša dodane vrednosti ali povečuje naše sposobnosti za doprinos dodane vrednosti, prenehamo s tem in začnimo delati nekaj drugega.

Vse je vedenje

Če smo na nivoju kode, na nivoju aplikacije ali višje, vedno lahko uporabimo isto razmišljanje in iste jezikovne konstrukte za opis vedenja. Tako kot lahko opišemo vedenje aplikacije s perspektive interesnih skupin, tako lahko opišemo vedenje kode na nižjem nivoju s perspektive druge kode, ki jo uporablja.

3.3 Agilne metode druge generacije

Vedenjsko voden razvoj je dejansko ena izmed agilnih metod, natančneje agilna metoda druge generacije. Skupina strokovnjakov je napisala manifest agilnosti [16], po katerem agilne metode rešujejo tradicionalna projektna tveganja. Posledice so opisane v tem poglavju.

Manifest agilnosti

Odkrivamo boljše načine razvoja programske opreme, tako, da jo razvijamo, in pri tem pomagamo tudi drugim. Naše vrednote so ob tem postale:

Posamezniki in interakcije pred procesi in orodji

Delujoča programska oprema pred vseobsežno dokumentacijo

Sodelovanje s stranko pred pogodbenimi pogajanji

Odziv na spremembe pred togim sledenjem načrtom

Z drugimi besedami, četudi cenimo dejavnike na desni, vseeno bolj cenimo tiste na levi.

Nič več zamud pri dobavi in prekoračenega proračuna

Ker dobavljamo sistem v majhnih, eno ali dvotedenskih ponovitvah oz. mini projektih z majhno ekipo, je preprosto izračunati proračun projekta.

Če začnemo s smiselno domnevo o celotni velikosti projekta, to pomeni, koliko smo pripravljeni investirati v rešitev problema in uredimo funkcionalnosti po pomembnosti. Tako ekipa lahko dobavi zares pomembne stvari v zgodnjih ponovitvah. Ko pridemo do točke, ko nam zmanjka denarja, bi že morali delati na manj pomembnih funkcionalnostih. Prav tako lahko merimo, koliko proizvedemo v vsaki ponovitvi. To lahko uporabimo za napoved, kdaj bomo končali. Ko se približujemo roku za dostavo končnega izdelka, ima naročnik vedno nove ideje za funkcionalnosti in vidi, da se dogajajo velike stvari, se lahko odloči in financira še nadaljnje ponovitve. Prav tako se lahko odloči že pred rokom, da je dovolj funkcionalnosti in da zaključi predčasno in pohiti z izdajo.

Nič več dobav nepravih stvari

Naročniku dobavljamo delujoče programje vsaka dva tedna, kar pomeni, da dobavljamo funkcionalnosti, ki jih lahko prikažemo.

Po koncu vsake iteracije naročniku prikažemo nove funkcionalnosti, on pa lahko poda svoje pripombe, ter pove, če so potrebne še kakšne spremembe. Lahko se še razjasni kakšen nesporazum, dokler je delo še sveže v glavah razvojne ekipe. Ti redni mali popravki zagotovijo, da po nekaj mesecih dostavimo program, ki počne to, kar je naročnik želel.

Na začetku naslednje iteracije skupaj z naročnikom ponovno določimo prioritete novih funkcionalnosti. Lahko se upoštevajo nove ideje, predlogi in zahteve.

Nič več nestabilnosti v produkciji

Programski sistem dobavljamo vsako iteracijo, kar pomeni, da moramo postati dobri pri prevajanju in postavitvi programskega sistema. Te procese se trudimo čim bolj avtomatizirati.

Izdaja za produkcijo ali za testiranje je tako samo neko prevajanje za neko okolje. Aplikacijski strežniki so avtomatsko skonfigurirani in inicializirani, sheme podatkovnih baz so avtomatsko posodobljene, koda je avtomatsko prevedena, zgrajena in postavljena, vsi testi so avtomatsko izvedeni.

Nič več dragega vzdrževanja

To je ena najbolj oprijemljivih koristi agilnega procesa. Takoj po prvi iteraciji je ekipa praktično že v vzdrževalni fazi. Funkcionalnosti se dodajajo na sistem, ki že deluje, torej je potrebno biti pazljiv.

Nič nenavadnega ni za agilno razvojno ekipo, da dela hkrati na več verzijah aplikacije hkrati. Dodajajo se nove funkcionalnosti na novi verziji, zagotavlja se podpora za pred kratkim izdano verzijo, in zagotavlja se tudi odpravljanje hroščev na starejši verziji.

3.4 Izvedba vedenjsko vodenega razvoja

Na začetku projekta izvedemo začetne aktivnosti, da razumemo namen dela, ki ga počnemo, in da ustvarimo skupno vizijo. Nato sproti razgrajujemo zahteve (opisane s SMART lastnostmi) v funkcionalnosti in te naprej v uporabniške zgodbe in scenarije, katere skozi BDD cikel avtomatiziramo, da nas lahko vodijo in nam držijo fokus na tem, kar moramo dostaviti. Ti avtomatizirani scenariji postanejo testi sprejemljivosti in zagotavljajo, da aplikacija počne vse tisto, kar od nje pričakujemo.

Zgodbe in scenariji so posebej zasnovani za podporo takemu modelu dela, bolj natančno da se lahko avtomatizirajo in so jasno razumljivi interesnim skupinam.

3.4.1 SMART izid

Preden se lotimo dela, moramo razumeti, kaj bi radi naredili. Zato moramo skupaj z interesno skupino določiti cilje oziroma izid. Tako bomo vedeli, da je projekt dosegel svoj namen, kaj bodo sedaj lahko naredili, kar prej niso mogli. Ciljev ne sme biti preveč, drugače lahko izgubimo fokus. Cilje lahko opišemo s SMART lastnostmi.

Kratica SMART

Kratica SMART [17] nam v angleškem jeziku opiše izid ali cilj z sledečimi lastnostmi: določen (angl. specific), merljiv (angl. measurable), dosegljiv (angl achievable), pomemben (angl. relevant), časovno omejen (angl. timeboxed).

Določen pomeni, da je izid dovolj natančno določen, da vemo, kdaj je nekaj narejeno.

Merljiv pomeni, da lahko izmerimo, ali je cilj dosežen ali ne oziroma do kakšne mere je narejen.

Dosegljiv pomaga zmanjšati nerealistična pričakovanja.

Pomemben preprečuje poizkus vključtve vse možne funkcionalnosti, kar tako, za vsak slučaj.

Časovno omejen nam pove, da preprosto zaključimo, če nismo uspeli doseči cilja. V nasprotnem primeru lahko traja, dokler nam je nekdo pripravljen plačevati.

3.4.2 Uporabniške zgodbe

Za dosego ciljev moramo opisati stvari, ki jih mora programska oprema narediti. Opišemo jih z uporabniškimi zgodbami. Tako dobimo tudi sledljivost nazaj do izvorne potrebe člana interesne skupine.

Uporabniška zgodba je sestavljena iz večih komponent:

Naslov

Naslov nam pove, o kateri zgodbi govorimo.

Pripoved

Pripoved nam pove, kaj je namen zgodbe. Obstaja kar nekaj splošnih oblik pripovedi, važno pa je, da se zajame bistvo. Identificirati mora interesno skupino za to zgodbo, opis funkcionalnosti, ki jo ta želi, ter razlog, zakaj jo želi (kaj je korist, ki jo pričakujejo).

Oblika

Najbolj znana oblika za to je znana kot Connextra oblika. Imenuje se po podjetju, ki jo je prvo uporabilo [18].

```
as a [stakeholder],
I want [feature],
so that [benefit].
```

```
kot [zainteresirani],
želim [funkcionalnost],
zato da bi [korist].
```

Novejša varianta, ki pridobiva na popularnosti, pa izgleda takole:

```
in order to [benefit],
as a [stakeholder],
I want a [feature].
```

```
zato da bi [korist],
kot [zainteresirani],
želim [funkcionalnost].
```

Vsebina je popolnoma enaka, razlika je samo v poudarku, ki pomaga obdržati fokus na izidu in ne na podrobnostih o funkcionalnosti.

Kriterij sprejemljivosti

Kriterij sprejemljivosti (angl. acceptance criteria) nam pove, kdaj smo končali. Pri vedenjsko vodenem razvoju ima obliko scenarijev, sestavljenih iz posameznih korakov.

Scenarij

Vsak scenarij ima naslov, oblika pa je opisana z angleško kratico GWT [19]:

```
given [initial context],        imamo [začetno stanje],
when [event occurs],            ko [se nekaj zgodi],
then [ensure some outcomes]     potem [so zagotovljeni rezultati]
```

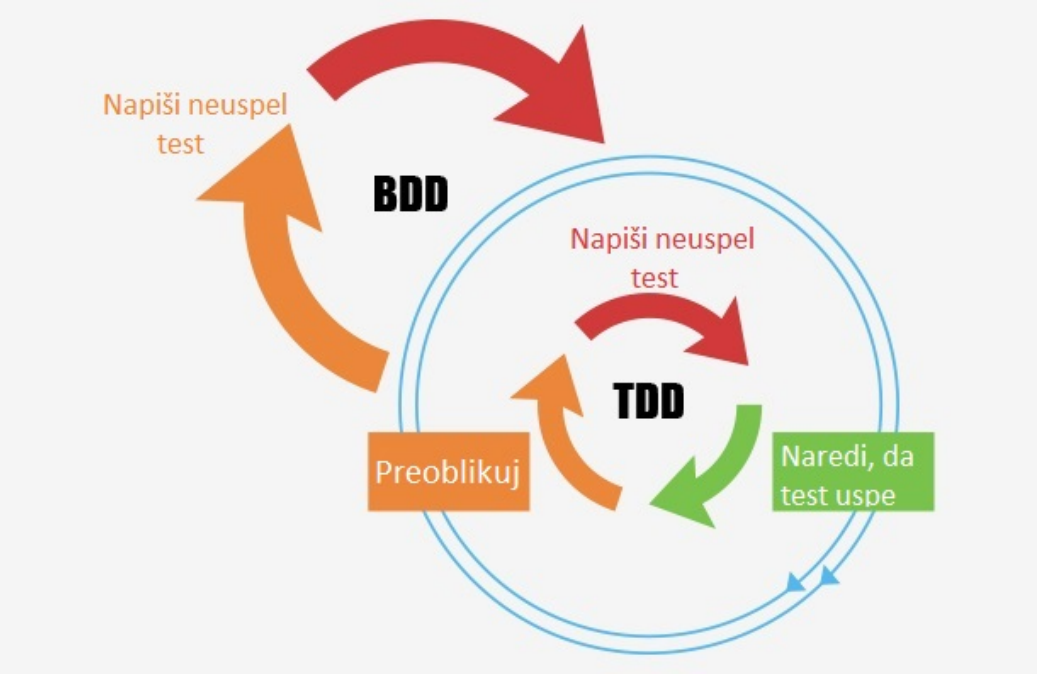
To ne pomeni, da mora imeti vsak scenarij natanko en `imamo`, `ko` in `potem` v tem vrstnem redu. Pomeni pa, da vsak korak bodisi nekaj nastavi v znano stanje (imamo), bodisi izvede neko vedenje (dogodek) bodisi preverja, da se je nekaj zgodilo (rezultat). Če poskušamo narediti več kot eno stvar v enem koraku, ponavadi povzročimo zmedo.

Taka ločitev je uporabna zato, ker je samo dogodek tisti, ki nas zanima. Za vzpostavitev začetnega stanja `imamo` ni pomembno, kako pridemo do njega. Lahko imamo vrednosti v podatkovni bazi, uporabimo uporabniški vmesnik, preberemo vrednosti iz datoteke. Važno je samo, da se koraki `ko` izvedejo z aplikacijo natanko tako, kot bi želela interesna skupina. Prav tako ni pomembno, kako se preverijo rezultati, važno je le, da se. Torej se lahko preverja DOM, ali pa se preverjajo vrednosti v podatkovni bazi, ali pa se izvede kakšno drugo preverjanje.

3.4.3 BDD cikel

Teste vedenja uporabimo, da opišemo vedenje aplikacije, se pravi obnašanje na višjem nivoju. Teste enot uporabimo, da opišemo vedenje objektov, se pravi obnašanje na bolj osnovnih nivojih. BDD cikel je vodenje razvoja od zgoraj navzdol, začenši z uporabniškimi scenariji, nato sledijo testi enot.

Iz TDD-ja poznamo cikel: rdeče-zeleno-preoblikuj (angl. Red-Green-Refactor) [13]. Z dodatkom BDD pa dobimo še en tak cikel, tako da imamo potem dva med seboj povezana cikla, kot nam kaže tudi slika 3.1.



Slika 3.1: BDD Cikel

Oba kroga vsebujeta majhne korake z opazovanjem povratne informacije naših orodij.

Začnemo z neuspešnim korakom (rdeče) v zunanjem BDD krogu. Da bi ta korak uspel, se spustimo v notranji TDD krog. Po vsakem uspešnem TDD ciklu, preverimo BDD cikel. Če je ta še vedno neuspešen (rdeč), nas to vodi

v nov TDD cikel. Ko pa BDD cikel postane uspešen (zelen), se lahko lotimo preoblikovanja, če je le-to potrebno. Nato pa začnemo z novim neuspešnim korakom.

Poglavje 4

Primer uporabe vedenjsko vodenega razvoja

4.1 Predstavitev

Uvod

Sodoben programski sistem je lahko sestavljen iz namiznih in mobilnih aplikacij na različnih platformah, ponavadi pa ima tudi spletno aplikacijo, ki teče v brskalniku. Tak pristop smo uporabili tudi pri razvoju naše aplikacije za vodenje osebnih financ BBSSolvent.

Primerna arhitektura za tak sistem je centralen API, ki vsebuje vso poslovno logiko. Vsi klienti uporabljajo isti API za pridobivanje, posodabljanje in manipulacijo podatkov. Vsi klienti imajo isti nabor funkcionalnosti, in ko je potrebno narediti spremembo, se naredi samo na enem mestu v skladu z DRY [20] principom. Arhitektura je prikazana na sliki 4.1.

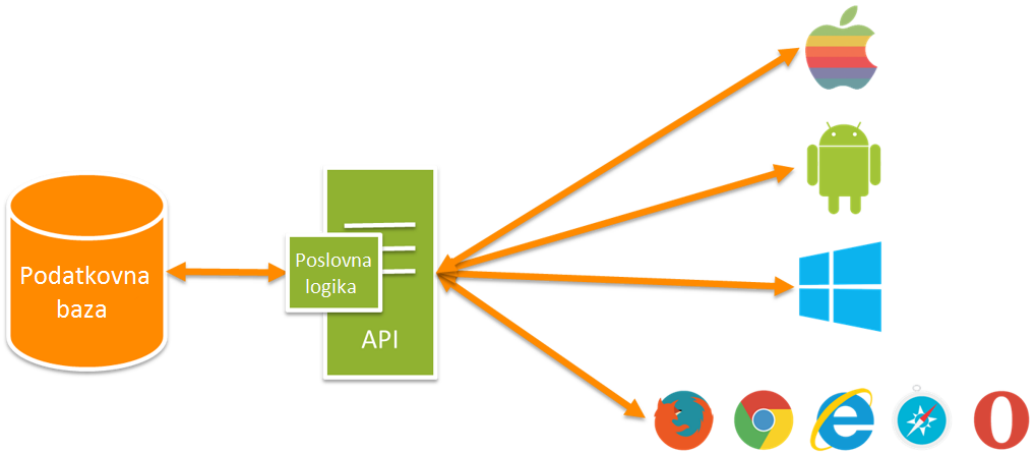
Aplikacija BBSSolvent nam omogoči, da si določimo proračun za izdatke za hrano in za ostale izdatke za tedensko obdobje. Tedensko obdobje je primernejše od mesečnega, ki je običajen pri ostalih tovrstnih aplikacijah, ker so izdatki na tedenskem nivoju precej bolj konstantni kot pa na mesečnem

nivoju. Vsak teden ima namreč natanko pet delovnih dni, in natanko en vikend, medtem ko imajo meseci od osemindvajset do enaintrideset dni, nekateri imajo štiri konce tedna, drugi pet, zaradi česar so neprimerljivi med sabo.

Aplikacija nam torej omogoča vnos dveh tipov izdatkov. V pregledu za trenutni teden lahko vidimo, koliko smo že porabili oziroma koliko imamo še na razpolago v posamezni kategoriji, ter koliko dni je še do konca obdobja.

Spletna aplikacija je sestavljena iz dveh delov, odjemalca in zalednega sistema. Vsak del se lahko razvija neodvisno od drugega. Vsak del ima svojega uporabnika, pri odjemalcu je to oseba, ki uporablja aplikacijo, uporabnik zalednega sistema pa je odjemalec.

V tem poglavju bomo pogledali uporabo vedenjsko vodenega razvoja na primeru obeh delovrazvoja spletne aplikacije za upravljanje osebnih financ.



Slika 4.1: API arhitektura

4.2 Zaledni sistem

Prestavitev

Najprej si bomo pogledali uporabo vedenjsko vodenega razvoja na primeru razvoja zalednega sistema za upravljanje osebnih financ. Za API je uporabljeno ogrodje .Net Web Api [21] in programski jezik C Sharp [22].

Pogledali bomo konceptualne vzorce in orodja za pripravo testnih podatkov, ter programska orodja in ogrodja, ki jih potrebujemo za izvedbo celotnega BDD postopka.

Uporabniška zgodba

Prva stvar, ki jo potrebujemo, je uporabniška zgodba. Primer take zgodbe je naslednji:

`Naslov: API metoda stroški`

`zato da bi` [manipuliral s stroški]
`kot` [klient]
`želim` [ustvariti, prebrati, posodobiti, zbrisati strošek]

Scenarij

Naslednja stvar, ki jo naredimo, so scenariji.

`Naslov: Preberi strošek`
`imamo` [nekaj stroškov shranjenih]
`ko` [kličemo get metodo s parametrom id]
`potem` [nam vrne pravilen strošek]

`Naslov:` `Zbriši strošek`

`imamo` [nekaj stroškov shranjenih]

`ko` [kličemo delete metodo s parametrom id]

`potem` [je strošek izbrisan]

BDDfy

Ker želimo, da so uporabniške zgodbe zapisane v obliki izvedljivih avtomatiziranih testov, si pomagamo z ogrodjem BDDfy.

BDDfy [23] je preprosto BBD ogrodje, ki nam omogoča, da uporabniške zgodbe in scenarije zapišemo v C Sharp kodi. Deluje s katerimkoli testnim ogrodjem in poganjalcem testov.

```
8   [TestFixture]
9   [InOrderToStory(
10      InOrderTo = "In order to manipulate expenses",
11      AsA = "As a client app",
12      IWant = "I want to CRUD expenses")]
13  public class ApiMethodExpensesTestsV1
14  {
15      [Test]
16      public void GetExpense()
17      {
18          this.Given(_ => _.FewExpensesAreStored())
19              .When(_ => _.GetWithIdIsCalled())
20              .Then(_ => _.CorrecExpenseIsReturned())
21          .BDDfy();
22      }
23
24      [Test]
25      public void DeleteExpense()
26      {
27          this.Given(_ => _.FewExpensesAreStored())
28              .When(_ => _.DeleteWithIdIsCalled())
29              .Then(_ => _.ExpenseIsDeleted())
30          .BDDfy();
31      }
```

Slika 4.2: BDDfy, NUnit, NCrunch

NUnit

Potrebujemo tudi testno ogrodje. Uporabili bomo NUnit. NUnit [24] je testno ogrodje za vse .NET jezike.

NCrunch

Kot poganjalec testov pa bomo uporabili orodje NCrunch.

NCrunch [25] je avtomatizirano hkratno orodje za testiraje za Visual Studio. Teste poganja samodejno, ko shranimo kodo oziroma teste in nam sproti prikazuje rezultate testiranja. Z uporabo tega orodja lahko takoj ugotovimo, če smo s spreminjanjem kode pokvarili kakšen test.

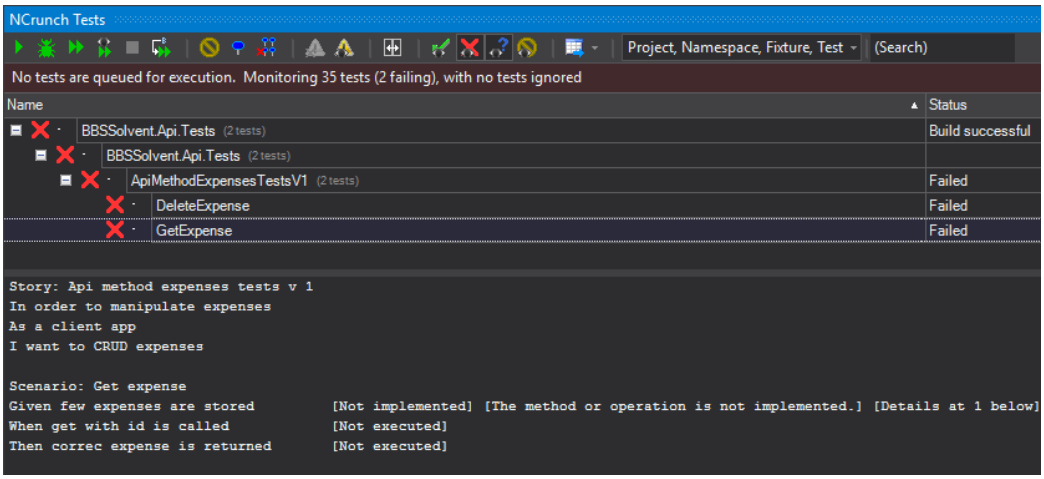
Postopek

Uporabniško zgodbo in scenarije zapišemo v obliki testov, kot vidimo na sliki 4.2. Ker nimamo napisane še nobene kode, testi seveda ne uspejo, kar vidimo na sliki 4.2 z NCrunchevim kazalcem pokritosti kode s testi (rdeč x). Neuspele teste vidimo tudi v NCrunchevem oknu na sliki 4.3.

Sedaj imamo zgodbe in scenarije napisane v ogrodju BDDfy, in ko jih NCrunch požene, vidimo, da ne uspejo.

Tako smo začeli z neuspešnim korakom v BDD ciklu. Sedaj lahko začnemo z implementacijo. Začnemo pri `Given` koraku, v katerem pripravimo kontekst, v tem primeru so to testni podatki.

Generiranje testnih podatkov s pomočjo konceptualnih vzorcev si bomo pogledali v naslednjem poglavju.



Slika 4.3: NCrunch okno

4.3 Testni podatki

Generiranje testnih podatkov

V implementaciji koraka `Given` pripravimo podatke, ki jih bomo potrebovali v testu.

Generiraje testnih podatkov za avtomatizirano testiranje lahko postane zelo dolgočasno opravilo, ko aplikacija postaja vedno bolj zahtevna. Posledica je, da testi postajajo zahtevni za vzdrževanje. Če je testna koda enako pomembno kot produkcijska koda, potem želimo imeti logiko v testih preprosto, lahko za vzdrževanje in brez ponavljanj - DRY (angl. don't repeat yourself) [20].

Med razvojem, ko avtomatizirani skupki testov organsko rastejo in objektni model pridobiva na zahtevnosti, in ne razmišljamo, kako bomo generirali testne podatke, potem zelo verjetno pridemo do situacije, v kateri ima veliko testov del *Nastavi* polno instanciranja objektov, ki postajajo vedno bolj zahtevni.

Zaradi vsega tega se pojavijo problemi, kot so:

- ko se spremeni konstruktor nekega razreda, je potrebno popraviti veliko referenc tega konstruktorja čez vse testne projekte, in tako preoblikovanje kode postane boleče oziroma pripelje do uporabe bližnjic.

- veliko ponavljanja v različnih testnih razredih, kjer se kreirajo objekti.

- grajenje seznamov je zelo obširno z veliko ponavljanja oziroma težko berljivo kodo.

- generirane testnih podatkov je nekonsistentno po posameznih testnih razredih, zaradi tega je koda težja za razumevanje in vzdrževanje.

- ni nemudoma jasno, kateri od parametrov konstruktorja je tam zaradi testa in kateri zaradi potreb konstruktorja, zaradi česar je namen testa lahko nejasen.

Objekt mati

Dober pristop za rešitev nastalega problema je konceptualni vzorec Objekt mati (angl. Object Mother) [26] [27]. Ta nam omogoča hitro in jedrnato ustvarjanje predefiniranih objektov z opisnim imenom. Namesto zahtevne verige instanciranih objektov imamo eno vrstico kode, kar je precej bolj opisno in posledično so testi preprostejši. Zagotovi nam tudi, da je pristop k generiranju podatkov usklajen in zaradi tega lažji za vzdrževanje. Primer razreda Objekt mati vidimo na sliki 4.4.

Če uporabljamo samo konceptualni vzorec Objekt mati, žal opazimo sledeče probleme:

- čeprav so klici konstruktorjev razredov sedaj v eni datoteki, je vseeno še lahko veliko klicev, zato je še vedno izziv spreminjati konstruktorje.

- ko potrebujemo minimalno drugačne podatke, preprosto naredimo novo lastnost Objekta mati.

- zaradi tega razred Objekt mati hitro postane nepriročen in zahteven za razumevanje in vzdrževanje, postane Razred bog (ang. God Class) [28].

```
3   public static class ObjectMother
4   {
5       public static class Expenses
6       {
7           public static ExpenseBuilder Default
8           {
9               get
10              {
11                  return new ExpenseBuilder();
12              }
13          }
14
15          public static ExpenseBuilder Food1
16          {
17              get
18              {
19                  return new ExpenseBuilder()
20                      .WithAmount(5)
21                      .WithType(ExpenseType.Food);
22              }
23          }
24
25          public static ExpenseBuilder Other1
26          {
27              get
28              {
29                  return new ExpenseBuilder()
30                      .WithAmount(5)
31                      .WithType(ExpenseType.Other);
32              }
33
34          }
35
```

Slika 4.4: Razred Objekt mati

Graditelj testnih podatkov

Alternativa Objektu mati je konceptualni vzorec Graditelj testnih podatkov (angl. Test Data Builder) [29], kateri je varianta Graditeljskega vzorca (ang. Builder Pattern) [30].

Ustvari se graditeljski razred (slika 4.5), ki je odgovoren za kreiranje objekta preko tekočega vmesnika (angl. Fluent Interface) [31]. Ta poskrbi, da je objekt v željenem stanju, preden se dejansko skreira.

```
public class ExpenseBuilder : TestDataBuilder<Expense, ExpenseBuilder>
{
    private Expense _expense;
    public ExpenseBuilder()
    {
        _expense = new Expense();

        Set(_ => _.Timestamp, DateTime.Now);
    }

    public ExpenseBuilder WithId(int id)
    {
        Set(_ => _.ID, id);
        return this;
    }

    public ExpenseBuilder WithComment(string comment)
    {
        Set(_ => _.Comment, comment);
        return this;
    }

    public ExpenseBuilder WithAmount(int amount)
    {
        Set(_ => _.Amount, amount);
        return this;
    }

    public ExpenseBuilder WithType(ExpenseType type)
    {
        Set(_ => _.Type, (int)type);
        return this;
    }
```

Slika 4.5: Graditelj testnih podatkov

To prinaša številne prednosti:

- enkraten klic konstruktorja v testnem projektu, kar omogoča enostavno spreminjanje,

- tekoči vmesnik za generiranje objektov, zaradi česa so testi preprosti za branje, pisanje in vzdrževanje,

- usklajenost pri generiranju testnih podatkov,

- živo dokumentacijo (ang. Living Documentation) [32], kako se ravna oziroma uporablja domenske objekte,

- dovolj prilagodljiv, da omogoča izvajanje akcij nad objektom, potem ko je že skreiran,

- graditelj testnih podatkov lahko vsebuje začetne vrednosti lastnosti objekta, tako da se v testu nastavijo samo vrednosti, ki so potrebne.

Kombinacija

Ena boljših stvari pri Objektu mati je, da se zmanjšajo ponavljanja v testih s ponovno uporabo predefiniranih objektov.

Lahko pa združimo fleksibilnost, ki nam jo daje Graditelj testnih podatkov z jedrnatostjo, ki nam jo omogoča Objekt mati, in sicer tako, da nam Objekt mati vrača Graditelja testnih podatkov. Na tak način se izognemo vsem slabostim konceptualnega vzorca Objekta mati, ki smo jih navedli prej.

```
38      [SetUp]
39      public void Setup()
40      {
41          _timeRangeCalculation = new Mock<ITimeRangeCalculation>();
42          _context = new TestBBSSolventApiContext();
43          _controller = new Expenses2Controller(_context, _timeRangeCalculation.Object);
44      }
45
46      [Test]
47      public void GetExpense()
48      {
49          this.Given(_ => _.FewExpensesAreStored())
50              .When(_ => _.GetWithIdIsCalled())
51              .Then(_ => _.CorrecExpenseIsReturned())
52          .BDDfy();
53      }
54
55      private void FewExpensesAreStored()
56      {
57          _foodExpense1 = ObjectMother.Expenses.Food1.Build();
58          _foodExpense2 = ObjectMother.Expenses.Food2.Build();
59          _otherExpense1 = ObjectMother.Expenses.Other1.Build();
60          _otherExpense2 = ObjectMother.Expenses.Other2.WithId(42).Build();
61          _context.Expenses.Add(_foodExpense1);
62          _context.Expenses.Add(_foodExpense2);
63          _context.Expenses.Add(_otherExpense1);
64          _context.Expenses.Add(_otherExpense2);
65      }
66
67      private void GetWithIdIsCalled()
68      {
69          _actionresult3 = _controller.GetExpense(42) as OkNegotiatedContentResult<Expense>;
70      }
71
72      private void CorrecExpenseIsReturned()
73      {
74          Assert.IsNotNull(_actionresult3);
75          Assert.AreEqual(42, _actionresult3.Content.ID);
76      }
77
```

Slika 4.6: izvedba korakov scenarija

Da si olajšamo iskanje objektov, uporabimo statičen parcialni razred za Objekt mati. Naredimo Objekt mati mapo oziroma imenski prostor v testnem projektu, v katerem je datoteka za vsako entiteto, ki jo generiramo s statičnim gnezdenim razredom znotraj korenskega Objekt mati razreda. S tem si zagotovimo, da do vseh testnih podatkov pridemo tako, da najprej

vpišemo `ObjectMother`. Tako imamo zagotovljeno konsistenco in preprosto dostopnost brez Razredov bogov.

Če naš test zahteva malenkostno spremembo na predefiniranem objektu, nam ni potrebno dodajati nove lastnosti v razred Objekt mati, ampak lahko uporabimo graditeljevo metodo.

Na sliki 4.6 v metodi `FewExpensesAreStored` koraka `Given` vidimo preprosto in pregledno zgrajene testne podatke, ki jih potrebujemo v scenariju, implementirane s konceptualnima vzorcema Objekt mati in Graditelj testnih podatkov, pri enem primeru pa vidimo tudi, kako z graditeljevo metodo `WithId(42)` dosežemo malenkostno spremembo na predefiniranem objektu.

4.4 Izvedba testov

When

Naslednja stvar, ki jo vidimo na sliki 4.6, je korak `When`, implementiran v metodi `GetWithIdIsCalled`. Tukaj se zgodi akcija, pokliče se metoda `GetExpense` s parametrom `Id`, ki smo ga v koraku `Given` z graditeljevo metodo nastavili enemu od testnih podatkov.

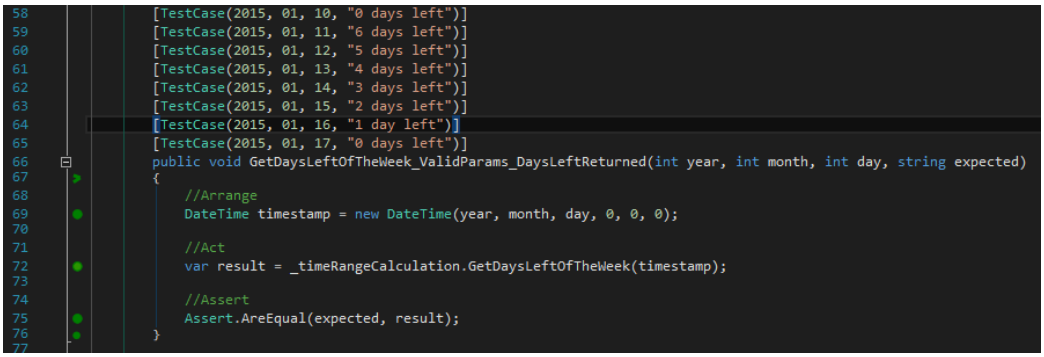
Then

V zadnjem koraku `Then` preverimo, če rezultat, ki smo ga dobili v prejšnjem koraku, ustreza pričakovanemu.

NUnit

Na sliki 4.6 vidimo še testno ogrodje NUnit [24], testno ogrodje za vse .NET jezike. Ena glavnih prednosti NUnita je atribut `[TestCase]`, ki nam omogoča poganjati teste s parametri, kar vidimo na sliki 4.7.

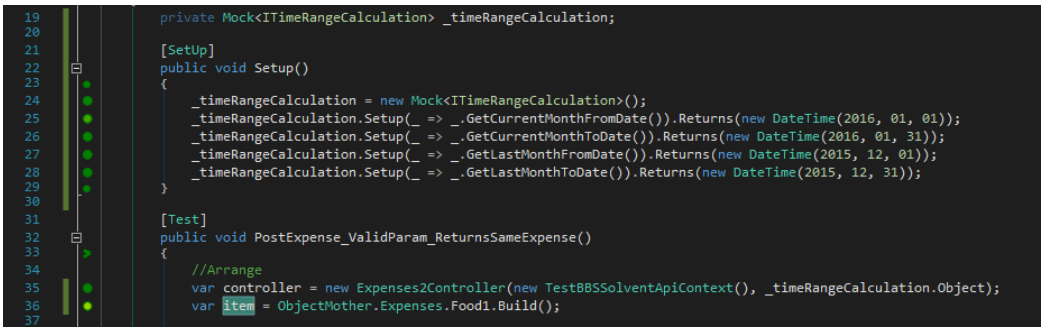
NUnitova metoda za nastavitev (angl. setup) se kliče pred izvajanjem vsakega testa v testnem razredu. Prepoznamo jo po atributu `[SetUp]`. V tej metodi pa vidimo še orodje Moq.



Slika 4.7: NUnit

Moq

Moq [33] je najpopularnejše in najbolj prijazno ogrodje za testne dvojnike (angl. mocking framework) za .NET. Izkorišča prednosti .NET Linq izrazov, zaradi česar omogoča večjo produktivnost. Podpira nadomeščanje vmesnikov in razredov, ima enostaven API.



Slika 4.8: Moq

Na sliki 4.8 vidimo, kako metodam nadomestnega vmesnika določimo, kakšne vrednosti bodo vračale ob klicu.

V metodi `[SetUp]` na sliki 4.6 je Moq uporabljen za nadomeščanje vmesnika, ki je injeciran v kontroler. Tako dosežemo, da so metode kontrolerja testirane v izolaciji.

4.5 Izdelava kode

Prikazali smo, kako se pripravi testni del vedenjsko vodenega razvoja, konceptualne vzorce, ogrodja in orodja, ki nam pri tem pomagajo.

```
110         // GET: api/Expenses2/5
111         [ResponseType(typeof(Expense))]
112         public IHttpActionResult GetExpense(int id)
113         {
114             Expense expense = _dataSource.GetExpense(id);
115             if (expense == null)
116             {
117                 return NotFound();
118             }
119
120             return Ok(expense);
121         }
122
```

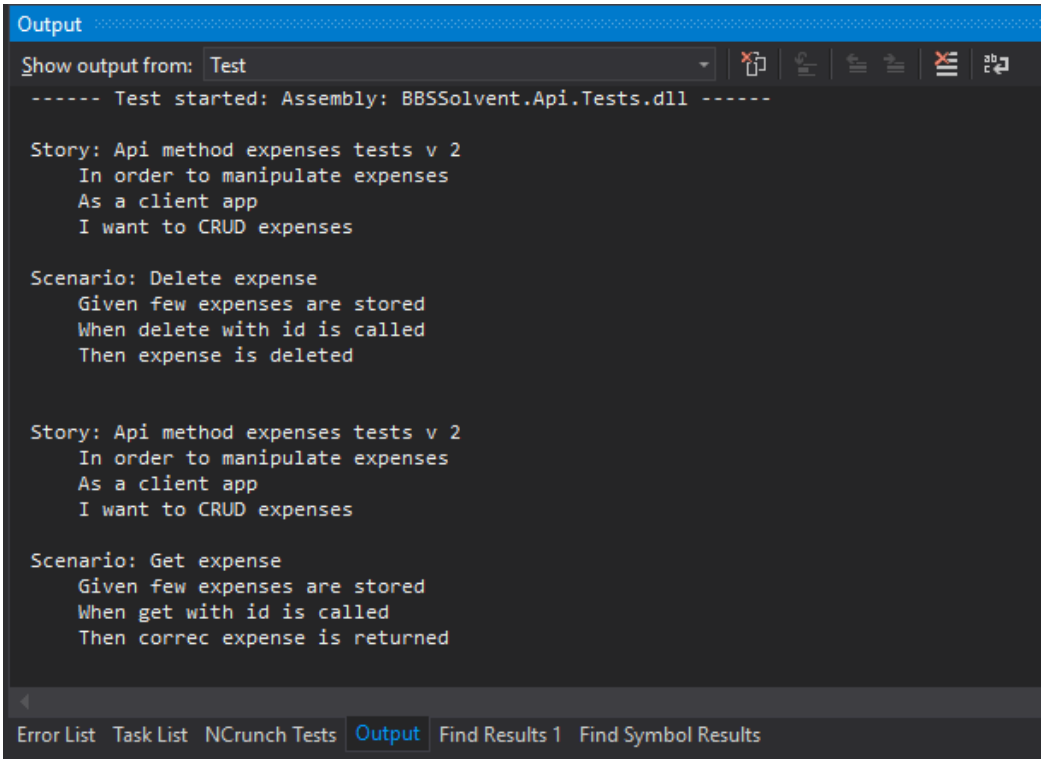
Slika 4.9: Izvedba metode

Preostane nam samo še implementacija kode. Orodje za testiranje NCrunch nam preko samodejnega poganjanja testov pokaže, kdaj smo pri implementaciji uspešni. Na sliki 4.6 vidimo, da so vsi kazalci pokritosti kode s testi obarvani zeleno, kar pomeni, da je naša implementacija, ki jo vidimo na sliki 4.9 uspešna. Tako se lahko lotimo naslednjega koraka, to je preoblikovanje kode, odstranimo morebitno podvojenost kode, kjer je to potrebno.

Tako smo zaključili cikel `rdeče-zeleno-preoblikuj` (angl. red-green-refactor) [13] in lahko začnemo nov cikel z novim testom.

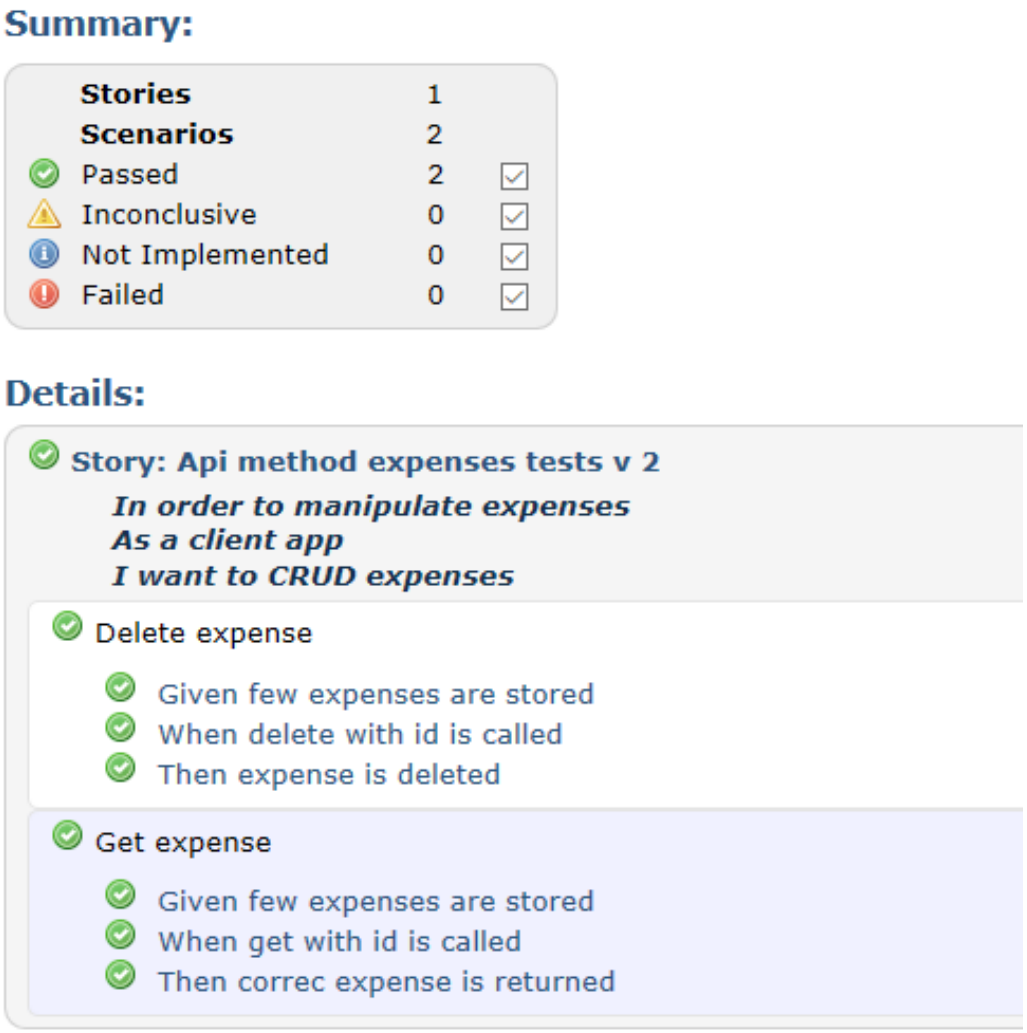
Testno poročilo

Kadar poženemo teste v razvojnem okolju, se nam v konzolnem oknu (slika 4.10) izpišejo rezultati testov v obliki zgodb in scenarijev v formatu, ki je razumljiv vsem interesnim skupinam, ne samo razvijalcem.



Slika 4.10: Poročilo v konzoli

Ker pa interesne skupine nimajo dostopa do razvojnega okolja, jim lahko ponudimo rezultate v HTML dokumentu, kot vidimo na sliki 4.11, do katerega interesne skupine lahko dostopajo preko omrežja ali preko strežnika za zvezno integracijo (angl. continuous integration).



Slika 4.11: Poročilo v formatu HTML

Tako lahko ves čas spremljajo napredek razvojne ekipe in vidijo, katere funkcionalnosti so že razvite.

4.6 Odjemalec

Uvod

Sedaj smo pogledali vedenjsko voden razvoj na zalednem sistemu, ki služi kot strežnik različnim odjemalcem, prestavili smo poudarek na obnašanje med dvema deloma sistema. V nadaljevanju pa si bomo za bolj celovit prikaz pogledali še postopek vedenjsko vodenega razvoja na primeru odjemalca zalednega sistema, in sicer spletnega odjemalca. Tukaj bomo prikazali uporabo vedenjsko vodenega razvoja na drug način, z uporabo drugih tehnologij in tudi na drugem nivoju, in sicer med končnim uporabnikom in spletnim klientom.

Za izvedbo testov bomo uporabili ogrodji Protractor in Jasmine, za izdelavo kode pa JavaScript ogrodje AngularJS.

Predstavitev

JavaScript ogrodje AngularJS [34] nam omogoča testiranje enot in testiranje od začetka do konca (angl. end-to-end), ki pa sta med seboj precej različna. Testiranje enot deluje na manjših enotah kode, testiranje od začetka do konca pa deluje na celih področjih aplikacije, tako da poganja teste preko posebnega HTTP strežnika.

Uporabniška zgodba

Zopet je uporabniška zgodba prva stvar, ki jo potrebujemo. Primer te je recimo:

`Naslov: Pregled stroškov`

`zato da bi` [imel pregled nad stroški]

`kot` [uporabnik]
`želim` [videti stroške za hrano]

Scenarij

Potem je na vrsti scenarij.

```
Naslov:  Pregled shranjenih stroškov
imamo [nekaj stroškov shranjenih]
ko [odprem stran Expenses]
potem [so prikazani stroški za hrano]
```

Testiranje 'od začetka do konca'

Testiranje od začetka do konca si lahko predstavljamo, kot da bi robot uporabljal brskalnik za poganjanje testov in preverjanje trditev, potem ko je spletna stran naložena. Programsko orodje za poganjanje testov je Protractor. Protractor [35] uporablja ogrodje Jasmine [36] za testno ogrodje.

Jasmine

Ker želimo, da so uporabniške zgodbe zapisane v obliki izvedljivih avtomatiziranih testov, si pomagamo z ogrodjem Jasmine.

Jasmine je BDD ogrodje za testiranje JavaScript kode. Ni odvisno od brskalnikov, DOM-a ali kakšnega drugega JavaScript ogrodja. Primerno je za spletne aplikacije, Node.js projekte oziroma kjerkoli se lahko poganja JavaScript.

Protractor

Kot poganjalec testov pa bomo uporabili ogrodje Protractor.

Protractor je ogrodje za testiranje AngularJS aplikacij. Poganja teste za aplikacijo v pravih brskalnikih, tako kot bi aplikacijo uporabljal uporabnik. S tem nam omogoča tudi navzkrižno testiranje brskalnikov.

```
/*
Scenario: View stored Expenses

Given 3 Expenses are stored
When I open expense page
Then 3 Expenses should be displayed
*/
describe("E2E - SCENARIO: View stored Expenses", function () {

    beforeEach(function () {
        browser.get('#/expenses');
        browser.waitForAngular();
    });

    describe("GIVEN 6 Expenses are stored", function () {
        var expenses = element.all(by.repeater('expense in vm.expensedata'));
        describe("WHEN I open expense page", function () {

            it('THEN 3 food expenses should be displayed', function () {
                expect(expenses.count()).toEqual(3);
            });
        });
    });

    describe("GIVEN 6 Expenses are stored", function () {
        var expenses = element.all(by.repeater('expense in vm.expensedata'));
        describe("WHEN I open expense page", function () {
            it('THEN 3 other expenses should be displayed', function () {
                expect(expenses.count()).toEqual(3);
            });
        });
    });
});

```

Slika 4.12: Jasmine test

Postopek

Scenarije zapišemo v obliki testov, kot vidimo na sliki 4.12. Jasmine ima za opis testa na voljo funkciji `describe` in `it`. Funkcija `describe` ima dva parametra, prvi je tekst, v katerem je opisano, kaj se testira. Drugi parameter

pa je funkcija, ki implementira test. Funkcija `it` pa je namenjena trditvi. Prav tako ima dva parametra, tekst z opisom trditve in funkcijo, v kateri se preveri, ali je trditev resnična ali ne.

Za opis testa v obliki scenarija vgnezdimo več `describe` funkcij, vsaka od njih nam predstavlja določen element scenarija, naslov, korak `Given`, korak `When`. Za korak `Then` pa uporabimo funkcijo `it`.

Funkcija `beforeEach` nam služi za nastavitev in se izvede pred vsakim testom, v našem primeru brskalnik usmeri na ustrezen naslov.

Tukaj pa ne nastavljamo testnih podatkov, saj se ti testi poganjajo nad dejansko spletno aplikacijo, ki teče v brskalniku, aplikacija pa je povezana na zaledni sistem, kjer so podatki. Potrebujemo pa torej zaledni sistem z ustreznimi testnimi podatki. Kot smo povedali v prejšnjem poglavju, ni pomembno, kako pridemo do začetnega stanja.

```
Spec started
Started

  E2E - SCENARIO: View stored Expense

    GIVEN 6 Expenses are stored

      WHEN I open expense page
        ✓ THEN 3 food expenses should be displayed
.
    GIVEN 6 Expenses are stored

      WHEN I open expense page
        ✓ THEN 3 other expenses should be displayed
.
```

Slika 4.13: Testno poročilo

V tem poglavju smo pogledali arhitekturo sistema, prikazali smo vedenjsko voden razvoj na vseh nivojih razvoja, med končnim uporabnikom in klientom, ter med klientom in zalednim sistemom. Videli smo tudi, kakšne so razlike v testiranju različnih nivojev. S tem smo pokrili celoten spek-

ter razvoja aplikacije. Vmes smo pogledali še generiranje testnih podatkov, pogledali smo si ogrodja in orodja, videli smo tudi, kako se vedenjsko voden razvoj izvede v dveh popolnoma različnih razvojnih okoljih s popolnoma različnimi orodji in ogrodji. S tem smo pokazali, da vedensko voden razvoj ni omejen z določeno tehnologijo ali orodjem, ampak je neodvisen od tega.

Na sliki 4.13 vidimo izpis Protractorjevega testega poročila v konzolnem oknu.

Poglavje 5

Analiza vedenjsko vodenega razvoja

V tem poglavju bomo uporabili SWOT analizo [37] [38] za pregled, kaj smo opazili pri uporabi vedenjsko vodenega razvoja, katere so konkretne prednosti in slabosti vedenjsko vodenega razvoja, katere priložnosti se nam ponujajo, ter na katere nevarnosti moramo biti pozorni.

5.1 Prednosti

Vedenjsko voden razvoj rešuje problem komunikacije med razvijalci in interesnimi skupinami. To počne z uporabo komunikacijskih vzorcev kot so primeri. Tako oboji dobijo takojšno povratno informacijo, takoj se opazijo problemi, za katere smo pri klasičnih projektih potrebovali mesece ali leta, pri agilnih pa tedne.

5.2 Slabosti

Če je vedenjsko voden razvoj izveden pravilno, postane naša konkurenčna prednost, zmanjša nam stroške, predstavlja dodano vrednost. Če pa je izveden brez razumevanja, za kaj in kako bi moral bit uporabljan, je lahko

zapravljanje časa.

Ne moremo začeti razvijati funkcionalnosti, dokler ne vemo, kako se bo uporabljala, kakšnemu namenu bo služila, in kakšno dodano vrednost nam bo prinesla.

Če uporabniške zgodbe in scenariji niso narejeni z dovolj komunikacije in skupnega razumevanja o zahtevani funkcionalnosti, potem se bodo vračale nazaj k razvijalcem, ker to ni to, kar so želeli.

5.3 Priložnosti

Z uporabo vedenjsko vodenega razvoja se nam ponudi priložnost, da na formalen način izboljšamo komunikacijo med uporabniki in razvijalci.

Tako včasih razvijalci pridejo do boljših idej za rešitev poslovnih problemov kot interesne skupine, tako da razvijalci postanejo še razvijalci poslovnega projekta.

5.4 Nevarnosti

Nevarnost predstavljajo predvsem uporabniki, ki niso vešči uporabe tehnologij, ali pa niso pripravljeni sodelovati na način, kot bi bilo potrebno.

Poglavje 6

Sklepne ugotovitve

Vedenjsko voden razvoj se je izkazal za zelo primernega pri razvoju aplikacije, za katero so bile dobro poznane zahteve, tako da ni bilo težav pri pisanju uporabniških zgodb in scenarijev. Vedenjsko voden razvoj je manj primeren za razvoj slabše definiranih projektov in posledično manj kvalitetnih uporabniškh zgodb. Težava bi lahko bila tudi nepripravljenost interesnih skupin na tako obliko razvoja, kar bi pripeljalo do tega, da bi moral razvijalec sam pripraviti uporabniške zgodbe in scenarije.

Časovna komponenta je v uvajalnem obdobju daljša, vendar pa se, ko se razvijalci in interesne skupine prilagodijo, zmanjša, če pa upoštevamo še posledično manj časa porabljenega za vzdrževanje in odpravljanje hroščev, je dodatek časa za pisanje testov zanemarljiv.

Ni pa zanemarljiv prihranek, ki ga dobimo s tem, ker nam ni potrebno poganjati aplikacije vsakič, ko želimo preveriti pravilnost kode, saj nam orodja in ogrodja, ki jih uporabljamo pri vedenjsko vodenem razvoju, to sporočijo takoj, ko se koda spremeni.

Največji izziv pri vedenjsko vodenem razvoju je gotovo to, da je potreben zasuk v načinu razmišljanja razvijalca, podobno kot pri testno vodenem razvoju, zato imajo nekateri razvijalci lahko težave s prilagajanjem na tak način dela. Potrebno pa je imeti tudi podporo vodstva.

Naslednji korak pri zagotavljanju stabilnosti in kakovosti bi lahko bila

postavitev strežnika za zvezno integracijo, na katerem bi se ob prevajanju izvorne kode poganjali tudi testi in generirala poročila.

Tako bi prišli še korak bližje končnemu cilju, to pa je večja produktivnost in s tem boljše, hitrejše, cenejše dobavljanje programske opreme.

Literatura

[1] Behavior-driven development. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Behavior-driven_development. [Dostopano 12. 4. 2016].

[2] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008

[3] Test automation. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Test_automation. [Dostopano 12. 4. 2016].

[4] Unit Test [Online]. Dosegljivo: http://martinfowler.com/bliki/UnitTest.html. [Dostopano 1. 5. 2016].

[5] Portland Pattern Repository. [Online]. Dosegljivo: http://c2.com/cgi/wiki?ArrangeActAssert. [Dostopano 12. 4. 2016].

[6] NUnit SetUp Attribute. [Online]. Dosegljivo: https://github.com/nunit/docs/wiki/SetUp-Attribute. [Dostopano 23. 4. 2016].

[7] NUnit TearDown Attribute. [Online]. Dosegljivo: https://github.com/nunit/docs/wiki/TearDown-Attribute. [Dostopano 23. 4. 2016].

[8] Integration testing. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Integration_testing. [Dostopano 12. 4. 2016].

[9] Regression testing. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Regression_testing. [Dostopano 12. 4. 2016].

[10] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] Guide: Writing Testable Code. [Online]. Dosegljivo: http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf. [Dostopano 12. 4. 2016].

[12] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[13] Red-Green-Refactor. [Online]. Dosegljivo: http://www.jamesshore.com/Blog/Red-Green-Refactor.html. [Dostopano 12. 4. 2016].

[14] Project stakeholder. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Project_stakeholder. [Dostopano 12. 4. 2016].

[15] Broken Iron Triangle. [Online]. Dosegljivo: http://www.ambysoft.com/essays/brokenTriangle.html. [Dostopano 12. 4. 2016].

[16] Manifest agilnega razvoja programske opreme. [Online]. Dosegljivo: http://www.agilemanifesto.org/iso/sl. [Dostopano 25. 1. 2016].

[17] SMART criteria. [Online]. Dosegljivo: `https://en.wikipedia.org/wiki/SMART_criteria`. [Dostopano 12. 4. 2016].

[18] Connextra Story Card. [Online]. Dosegljivo: `http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html`. [Dostopano 12. 4. 2016].

[19] Given-When-Then. [Online]. Dosegljivo: `http://martinfowler.com/bliki/GivenWhenThen.html`. [Dostopano 12. 4. 2016].

[20] Don't Repeat Yourself [Online]. Dosegljivo: `http://c2.com/cgi/wiki?DontRepeatYourself`. [Dostopano 1. 5. 2016].

[21] ASP.NET Web API [Online]. Dosegljivo: `http://www.asp.net/web-api`. [Dostopano 1. 5. 2016].

[22] C Sharp [Online]. Dosegljivo: `https://en.wikipedia.org/wiki/C_Sharp_(programming_language)`. [Dostopano 1. 5. 2016].

[23] BDDfy [Online]. Dosegljivo: `http://bddfy.teststack.net`. [Dostopano 25. 1. 2016].

[24] NUnit [Online]. Dosegljivo: `http://www.nunit.org`. [Dostopano 25. 1. 2016].

[25] NCrunch [Online]. Dosegljivo: `http://www.ncrunch.net`. [Dostopano 25. 1. 2016].

[26] Object Mother [Online]. Dosegljivo: `http://martinfowler.com/bliki/ObjectMother.html`. [Dostopano 1. 5. 2016].

[27] Object Mother [Online]. Dosegljivo: http://www.c2.com/cgi/wiki?ObjectMother. [Dostopano 1. 5. 2016].

[28] God Class [Online]. Dosegljivo: http://www.c2.com/cgi/wiki?GodClass. [Dostopano 1. 5. 2016].

[29] Test Data Builder [Online]. Dosegljivo: http://www.c2.com/cgi/wiki?TestDataBuilder. [Dostopano 1. 5. 2016].

[30] Builder Pattern [Online]. Dosegljivo: http://www.c2.com/cgi/wiki?BuilderPattern. [Dostopano 1. 5. 2016].

[31] Fluent Interface [Online]. Dosegljivo: http://martinfowler.com/bliki/FluentInterface.html. [Dostopano 1. 5. 2016].

[32] Living Documentation [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Living_document. [Dostopano 1. 5. 2016].

[33] Moq [Online]. Dosegljivo: https://github.com/Moq/moq4. [Dostopano 25. 1. 2016].

[34] AngularJS. [Online]. Dosegljivo: https://angularjs.org/. [Dostopano 12. 4. 2016].

[35] Protractor [Online]. Dosegljivo: http://angular.github.io/protractor. [Dostopano 25. 1. 2016].

[36] Jasmine [Online]. Dosegljivo: http://jasmine.github.io. [Dostopano 25. 1. 2016].

[37] SWOT analysys [Online]. Dosegljivo: https://en.wikipedia.org/wiki/SWOT_analysis. [Dostopano 1. 5. 2016].

[38] Jeffrey P. Harrison, Wayne Gretzky. *Strategic Planning and SWOT Analysis* . Health Administration Press, 2010.