TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andrei Reinus, 162788 IABM

SPECIFY, BDD TÖÖVAHEND

Magistritöö

Juhendaja: Ants Torim
PhD

Tallinn 2018

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Andrei Reinus

07.05.2018

Annotatsioon

Antud magistritöö raames toob autor välja *Behaviour driven development*`i kitsaskohad ning põhjused, miks antud arenduspraktika pole laialt levinud. Töö tulemusena on autor loonud tarkvara, mida saavad analüütikud kasutada tarkvaraprojekti nõuete kirjeldamisel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 43 leheküljel, 5 peatükki, 15 joonist.

Abstract

Specify, BDD tool

In the thesis author explores the reasons why Behaviour driven development is lacking in popularity and finds a way to reduce some of the bottlenecks. As a result, author has developed a software-based tool for software analytics to use in their projects to describe software requirements in more convenient way.

The thesis is in Estonian language and contains 43 pages of text, 5 chapters, 15 figures.

Lühendite ja mõistete sõnastik

BDD	Behaviour driven development
TDD	Test driven development
Git	Versioonihaldustarkvara
Gherkin	Ärinõuete kirjelduskeel
CD	Continuous deployment
CI	Continuous integration
SOLID	Objektiorienteeritud tarkvara disaini põhimõtete kogum

Sisukord

1 Sissejuhatus 8
2 Taust 9
2.1 Eesmärk 9
3 Teooria 11
3.1 Behaviour driven development 11
3.2 Tavapärane arendusprotsess *versus* BDD arendusprotsess 12
3.3 BDD ja TDD samaaegne rakendamine 13
3.4 Takistused BDD rakendamisel 15
3.5 Gherkini lühiülevaade 16
3.6 BDD piirangud 17
4 Spetsifikatsioon 18
4.1 Nõuded 18
4.2 Arenduspõhimõtted 25
4.3 BDD arenduse näide ühe stsenaariumi põhjal 25
4.4 Arhitektuur ja tehniline lahendus 31
4.5 Andmemudel 33
4.6 Detailne Specify struktuur 34
4.7 Lahenduse verifitseerimine 36
5 Kokkuvõte 37
Lisa 1 – Ekraanivaated 41

Jooniste loetelu

Joonis 1. Tavapärane arendusprotsess 12
Joonis 2. BDD arendusprotsess 13
Joonis 3. TDD arendustsükkel 14
Joonis 4. BDD ja TDD arendustsükkel 14
Joonis 5. Gherkini süntaks inglise keeles 16
Joonis 6. Gherkini süntaks eesti keeles 17
Joonis 7. Komponentide diagramm 31
Joonis 8. Andmebaasimudel 34
Joonis 9. Projektide omavahelised seosed 36
Joonis 10. Ülevaade nõuete täitmisest 41
Joonis 11. Faili lisamine 41
Joonis 12. Faili muutmine 42
Joonis 13. Muudatuste kinnitusvorm 42
Joonis 14. Sünkroniseerimisvorm 43
Joonis 15. Seadete vorm 43

1 Sissejuhatus

Tarkvara, eriti äritarkvara aluseks on nõuded, mida on vaja koguda, hallata ning hilisemalt ka verifitseerida. Ebatäpsed või vasturääkivad nõuded on aluseks hilisematele tarkvaras esinevatele vigadele, mis omakorda mõjutavad arenduse hinda ja ajakava.

Behaviour driven development arenduspraktika keskendub rakenduse nõuetele ning võimaldab kirjeldada tarkvara toimimist kasutades näitestsenaariume. Struktureeritud formaat võimaldab spetsiaalsel tarkvaral neid nõudeid kontrollida.

Agiilsed arendusmeetodid on populariseerinud automatiseeritud testimist koodi funktsionaalsel tasemel, kuid nõuete vastavuse automatiseerimise praktikad on vähem levinud. Kasutades *behaviour driven development* arenduspraktikat on üks viis kuidas automatiseerida nõuete verifitseerimist, kuid mille praktilist kasutamist takistab vajalike töövahendite puudumine.

Antud töö raames on valminud töövahend, mis võimaldab vähemate tehniliste teadmistega inimestel osaleda projektides, kus kasutatakse BDD arenduspraktikat. Tarkvara meeskonnad, kes viljelevad ühiktestimisega seotud arenduspraktikaid saavad nüüd kergemini automatiseerida nõuete verifitseerimist.

Rakenduse arenduse aluseks on oodatavad nõuded, mida loodav töövahend peab suutma täita ning seejärel on võimalik alustada rakenduse arendusega, kus järgitakse nii BDD kui ka teisi projekti sobivaid arenduspraktikaid ning põhimõtteid. Käesoleva töö tulemusena valmiv rakendus peab olema kasutatav tavakasutaja poolt ning avatud võimalikele edasiarendustele tulevikus.

2 Taust

Uurimused on näidanud, et kuni 25% nõuetest tarnitakse tellijale vigastena [1] ning mida hilisemas projekti faasis viga avastatakse, seda kallim on tema parandamine [2].

Vigastest nõuetest tekkinud probleeme on raske tuvastada kuna nad on tihtipeale leitavad alles siis, kui tarkvara on juba kasutusele võetud ja inimesed töötavad rakendusega [3].

Lisaks toovad Shelly Park ja Frank Maurer välja, et „Tarkvaranõuded jõuavad tavapäraselt arendusmeeskonnani dokumentidena, mis sisaldavad kasutuslugusid või siis lihtsalt kirjeldavat teksti. Oma igapäevatöös on analüütikutel kasutada piiratud kogus tarkvaralisi töövahendeid võrreldes arendajatele mõeldud töövahenditega. Tarkvaraprojekti keerukuse kasvades kulub analüütikutel aina rohkem aega tellija soovide kaardistamisele ning heade töövahendite puudumine raskendab projekti nõuete haldamist.“

2.1 Eesmärk

Tarkvaraarenduse protsessi mitmetes etappides on võimalik kasutada automatiseerimist. Näiteks on olemas vahendid programmi lihtsaks kompileerimiseks, pakendamiseks, rakenduse paigaldamiseks, jne. Mida rohkem samme on automatiseeritud, seda vähem kulub arendusele inimressurssi ning samuti väheneb tõenäosus inimfaktorist tingitud vigade tekkeks rakenduse arenduse ja hooldamise käigus. Infotehnoloogia spetsialistid on üha kallinev ressurss ning seega peab otsima võimalusi, kuidas muuta tarkvaraarenduse protsessi efektiivsemaks automatiseerides manuaalseid töölõike.

Töö eesmärgiks on luua analüütikutele mõeldud tarkvararakendus, mis võimaldab projektides edukalt kasutada BDD praktikat. Antud praktika kasutamine on eriti tulemuslik pikaajalistes projektides, kus nõuded muutuvad ajas ning automatiseeritud nõuete kontroll on oluliselt kuluefektiivsem kui manuaalne testimine. Loodud tarkvara võimaldab äripoole esindajatel lisada ning muuta rakenduse spetsifikatsiooni ja nõudeid, hoida spetsifikatsiooni rakenduse koodiga samas versioonihalduses ning saada pidevat ülevaadet rakenduse vastavusest nõuetele.

Tarkvara arendamisel kasutatakse BDD arenduspraktikat ning kombineeritakse seda teiste arenduspraktikate ja –põhimõtetega. Tarkvaraarenduse tulemusena peab olema rakendus sellises seisus, et seda oleks võimalik kasutusele võtta mõnes tarkvaraprojektis ning tulevikus arendada edasi lisades juurde uut funktsionaalsust.

3 Teooria

Agiilse tarkvaraarenduse manifest [16], mis on tänapäevase tarkvaraarenduse põhimõtete üks alusdokumentidest, ütleb selgelt, et tarkvaraarenduses on oluline asetada rõhk koostööle kliendiga ning inimeste omavahelisele suhtlusele. Äri ja arendus peavad rääkima sama keelt, mõistma ühtemoodi muudatuste sisu ja liikuma sama eesmärgi suunas. Agiilsed arenduspraktikad soosivad tihedat tarnegraafikut ning väiksemate ning hoomatavate muudatustena tarbijale tarnimist. Iga tarne peaks läbima vastuvõtutestid, kus äriosakond või tellija teostab vastuvõtutestid, kinnitamaks tarkvara vastavust nõuetele. Hoolimata sellest, kas vastuvõtutestimist viib läbi tellija isiklikult või on delegeerinud vastutuse testimismeeskonnale, peab tarkvara iga tarnega vastama nõuetele.

Tarkvaraprojektid võivad kesta aastaid ning iga lisandunud muudatussoov mõjutab rohkemal või vähemal määral varasemaid nõudeid. Kuidas olla kindel, et uue arve väljatrüki kujundus ei ole muutnud arve koostamist? Täieliku kindluse annaks ainult kogu rakenduse uuesti testimine, mis sõltuvalt rakenduse suurusest on kallis ja pole kuluefektiivne, eriti olukorras, kus laiapõhjalist testimist oleks vaja korrata iga kahe nädala tagant. Inimeste tööjõule lootma jäädes kaotab ettevõte, kas tellija või siis täitja poolelt, oma konkurentsivõime ja peab leidma viise, kuidas korduvat ja ühetaolist tööd automatiseerida. Arendajalt saaks nõuda ühiktestimist (*unit testing*), mis aitab kaasa programmikoodi õigele funktsioneerimisele ja aitab leida vastuseid küsimusele "kas tarkvara töötab?". Vastust ei saaks küsimusele, kas rakendus teeb seda, mida tellija on oodanud ehk kas rakendus vastab nõuetele.

Nõuetele vastavusega tegeleb arenduspraktika, mida nimetatakse kas *Acceptance test driven development* või *Behaviour driven development*. BDD arenduspraktika näeb ette rakenduse nõuete kirjeldamist sellisel moel, mis võimaldaks töövahendil automaatselt verifitseerida rakenduse vastavust nõuetele.

3.1 Behaviour driven development

Behaviour driven development on Dan Northi [4] poolt kasutusele võetud termin kirjeldamaks tarkvara arenduspraktikat, mis sisaldab endas ideid *Domain driven design'*ist (DDD) [2] ja *Test driven development'*ist (TDD) [3].

DDDs on kasutusel põhimõte, et tarkvara peegeldab kasutusala terminoloogiat. Kui rakenduse kasutajad viitavad kliendile (*customer*) siis andmebaasis ning programmikoodis on ka samanimelised olemid, mitte kunde (*client*) või midagi kolmandat. Tellija ja arendusmeeskonna vaheline suhtlus on lihtsustatud, kui nad räägivad omavahel samade mõistetega. Programmi tasandil olevad olemid peegeldavad kasutusel olevat terminoloogiat, siis peaksid ka rakenduse nõuded olema pandud kirja kasutades sama sõnavara.

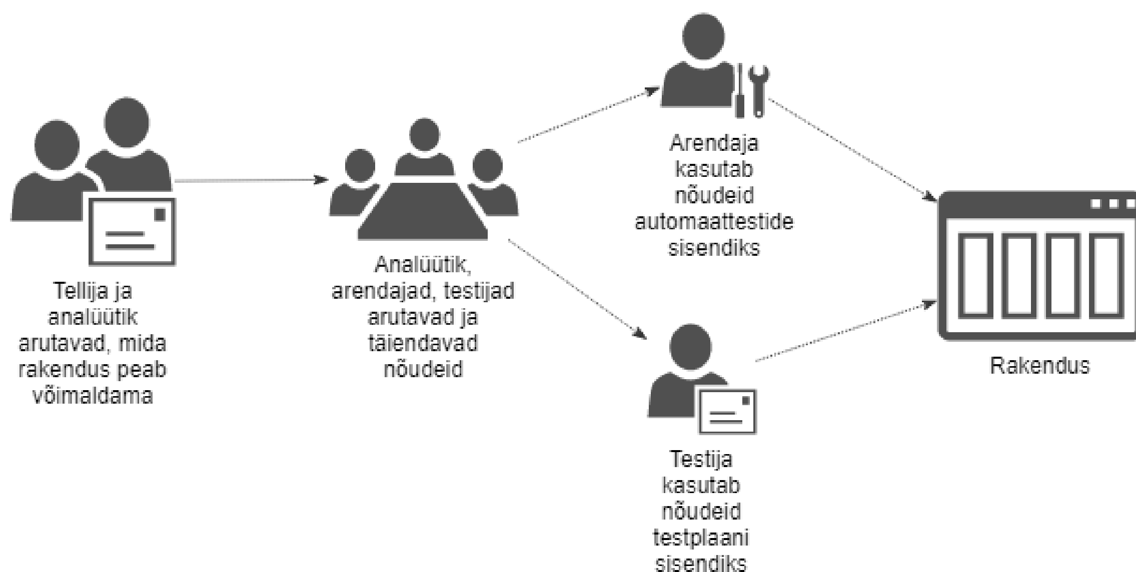
BDD kasutab nõuete kirjeldamiseks kasutuslugusid ja stsenaariume, mis kirjutatakse üles kasutades Gherkini [4] kirjelduskeelt (*domain specific language*). Erinevad arendusvahendid JBehave [5], RSpec [6], SpecFlow [7], BeHat [8] suudavad sisendfaili töödelda ning anda stsenaariumi käske edasi testkoodile, mis omakorda testib rakenduse vastavust nõuetele. Gherkini süntaksit ja toimimist on pikemalt kirjeldatud punktis 3.5.

3.2 Tavapärane arendusprotsess *versus* BDD arendusprotsess

Tavapärases arenduses suhtleb tellijaga analüütik, kes koostab nõuete dokumendi, mis on sisendiks arendusmeeskonnale rakenduse arenduseks ja testijatele testplaani koostamiseks. Arenduse tulemusena teostatud rakendus testitakse testmeeskonna poolt ja tarnitakse tellijale. Siinkohal ei ole erinevust, kas meeskond töötab kose või agiilsete meetoditega. Erinevus seisneb ainult selles, kui suur ühik rakendust tarnitakse ning kui tihti tarne toimub.

Joonis 1. Tavapärane arendusprotsess

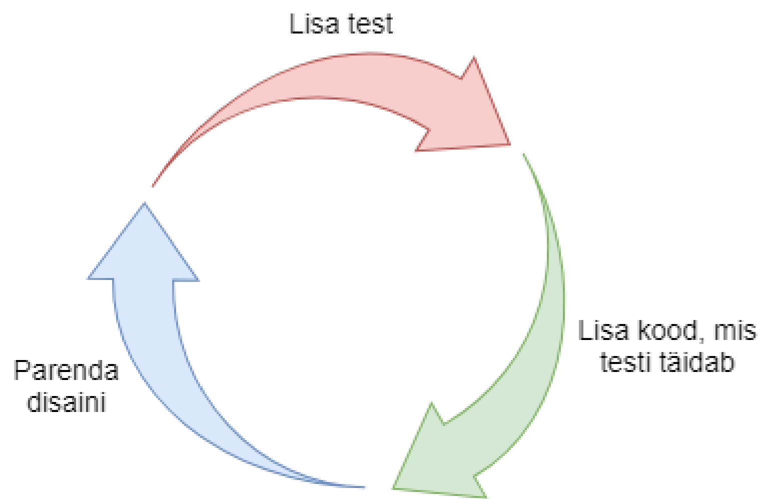
Kasutades *Behaviour driven development* praktikat, suhtlevad tellija ja analüütik omavahel nagu varasemalt, kuid nüüd kirjeldatakse nõudeid kasutuslugudena ja lisatakse juurde nõuet kirjeldavad stsenaariumid. Kirjeldatud nõudeid saab lugeda ja kinnitada tellija, mis aitab kindlustada loodava rakenduse vastavust tellija ootustele. Analüütik, arendus- ja testmeeskond arutavad üheskoos nõudeid ning võivad täiendada erinevaid stsenaariume. Antud arutelusse on soovitatav koheselt kaasata ka tellija, mis ühelt poolt kiirendab tekkinud küsimustele vastuste saamist ning samuti vähendab erinevate osapoolte võimalikust möödarääkimisest tulenevat riski. Arutluse tulemusena tekib täpsustatud ja kõikide osapoolte poolt üheselt mõistetav kasutuslugu, mida arendaja saab kasutada sisendiks oma töös ning testijal on olemas testplaan millest juhinduda.



Joonis 2. BDD arendusprotsess

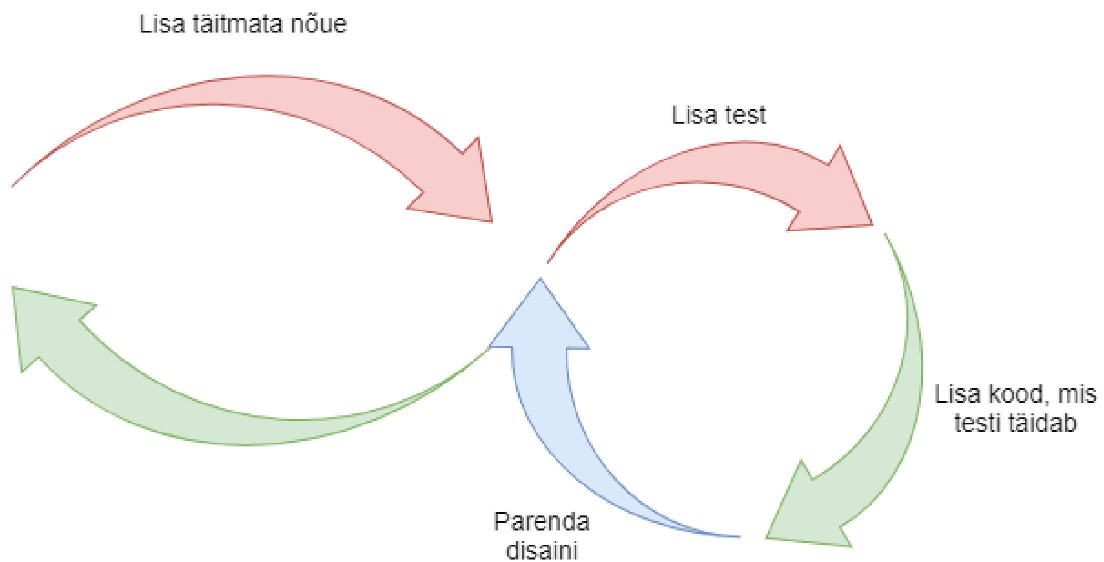
3.3 BDD ja TDD samaaegne rakendamine

Extreme programming (XP), mis sai tuntuks Kent Becki 1999. aastal avaldatud raamatuga „Extreme Programming Explained“ [9], viljeleb praktikat, mille kohaselt enne tarkvarakoodi kirjutamist on vaja kirjutada test, mis hakkab tulevast funktsionaalsust testima. Kirjeldatud praktika võimaldab kiiremini toota parema kvaliteediga koodi ning samaaegselt koguneb rakenduse testimiseks regressiooni testpakett. Selle praktika igapäevasel kasutamisel jaguneb töö kolmeks etapiks. Kõigepealt lisatakse test, mis uut funktsionaalsust testib, seejärel täiendatakse programmikoodi niikaua kuni testi tulemus on positiivne ning seejärel saab täiendada või kohendada rakenduse koodidisaini. Selliseid lühikesi iteratsioone korratakse kuni töölõik on valmis.



Joonis 3. TDD arendustsükkel

BDD arenduses järgitakse sarnast praktikat. Alustatakse nõude kirjeldamisest, seejärel arendatakse vajalikku funktsionaalsust ja teste, kuni kirjeldatud nõue on täidetud. Funktsionaalsuse arenduses saab BDD praktikat kombineerida TDD praktikaga, mille tulemusena näeb arendustsükkel välja nagu on kujutatud Joonis 4. *Test driven development* tsüklit korratakse kuni nõue on täidetud ning programmikood vastab ootustele.



Joonis 4. BDD ja TDD arendustsükkel

3.4 Takistused BDD rakendamisel

Äritarkvara loomisel on rakenduse peamisteks kasutajateks väiksemate infotehnoloogiliste teadmistega inimesed, kes samas määravad suures mahus nõuded, millele rakendus peab vastama. Seega on rakenduse spetsifitseerimisel oluline roll analüütikul, kes intervjuude kaudu või muul sarnasel moel kogub kokku ning kirjeldab rakenduse nõudeid.

Hoolimata sellest, et Gherkini kirjelduskeel on kindla formaadiga lihtne tekstifail, puudub analüütikul või tellijal mugav töövahend, millega igapäevaselt Gherkini faile hallata. Arendajatele mõeldud töövahendid saavad selle ülesandega hakkama, kuid on mõeldud ikkagi peamiselt programmikoodi kirjutamiseks ning samuti on nende töövahendite hinnatase 21 (HipTest [10]) ja 45 (Visual Studio [11]) euro vahel. Tellija jaoks on antud igakuine lisakulutus ebamõistlik arvestades projekti kestvust ning asjaolu, et kasutatakse vaid murdosa töövahendi kogu funktsionaalsusest. Tasuta alternatiive on võimalik rakendada kasutades Notepad++ [12] rakendust koos lisateegiga [13], kuid see ei elimineeri järgnevalt kirjeldatud takistust.

Rakenduse spetsifikatsioon peaks olema seotud rakenduse programmikoodiga, et oleks võimalik liikuda ajateljel ning saada teada, mis seisus oli rakendus igal ajahetkel. Seepärast on oluline hoida spetsifikatsiooni samas versioonihalduses koos rakenduse koodiga. Versioonihalduse kasutamine ei ole alati üheselt lihtne ega probleemivaba ning oskamatul käsitlemisel võib tekkida andmekadu. Rakenduse kasutajale eraldatakse limiteeritud funktsionaalsus versioonihaldusega suhtlemisel, mis lihtsustab kasutaja tööd ning vähendab võimalike tehtavate vigade riski.

Tavaliselt soovib tellija saada jooksvalt ülevaadet projekti arenduse seisust, kuid tavapraktikas saab perioodilist tagasisidet projektijuhilt. Nõuete automatiseeritud verifitseerimisel saab tellijale tulemustest teavitada jooksvalt. Lisaks õnnestuks tagasisidetsüklit oluliselt lühendada, kui stsenaariumi lisamisel suudaks *continuous integration* [14] juba muutunud nõuetega rakenduse läbitestida ja tagastada testimise tulemused. Erijuhtude verifitseerimine toimuks niimoodi kiiremini kui tavapärases arendustsüklis.

3.5 Gherkini lühiülevaade

Gherkin on kindla süntaksiga kirjelduskeel, millega kirjeldatakse rakenduse nõudeid. Iga nõue kirjeldatakse eraldi failis, mis on kindla struktuuriga koosnedes nõude pealkirjast, kasutusloo kirjeldusest ning ühest kuni mitmest stsenaariumist. Iga stsenaarium kirjeldatakse sammudena: eeltingimus (*Given*, *Eeldades*), sündmus (*When*, *Kui*) ja kontroll (*Then*, *Siis*).

```
# Nõude pealkiri
Feature: Withdraw money from ATM
# Kasutuslugu
    As a user with an account at the bank I would like to withdraw cash from ATM

# Stsenaarium 1
Scenario: Account has sufficient funds
    Given I have an account with 400 euros
    When I try to withdraw 100 euros from ATM
    Then I have 100 euros in cash
    And I have 300 euros on account

# Stsenaarium 2
Scenario: Insufficient funds
    Given I have an account with 40 euros
    When I try to withdraw 100 euros from ATM
    Then I should get an error message
    And Account has 40 euros
```

Joonis 5. Gherkini süntaks inglise keeles

Gherkin ei ole limiteeritud ainult inglise keelse süntaksiga, vaid spetsifikatsiooni on võimalik kirjutada enam kui kuuekümnes erinevas keeles [15], kaasa arvatud eesti keeles. Joonis 6 on näide eesti keelsest nõudest, mis oli inglise keelsena Joonis 5.

```
#language: et

# Nõude pealkiri
Omadus: Pangaautomaadist sularaha väljavõtmine
# Kasutuslugu
    Kasutajana kellel on pangas konto ma tahan
    kasutada pangaautomaati, et võtta välja sularaha

# Stsenaarium 1
Stsenaarium: Kontol on piisavalt raha
    Eeldades Mul on konto, millel on 400 eurot
    Kui Ma proovin välja võtta 100 eurot
    Siis Mul on 100 eurot sularaha
    Ja Mul on 300 eurot pangaarvel

# Stsenaarium 2
Stsenaarium: Kontol pole piisavalt raha
    Eeldades Mul on on konto millel on 40 eurot
    Kui Ma proovin välja võtta 100 eurot
    Siis Pangaautomaat näitab veateadet
    Ja Mu kontol on 40 eurot
```

Joonis 6. Gherkini süntaks eesti keeles

Mõlemas keeles kirjeldatud nõuded on üheselt mõistetavad sündmuste jadad, mida saab tellija soovi korral ise kirjeldada. Projektides, kus spetsifikatsiooni kirjutab analüütik saab tellija Gherkinis kirjeldatud stsenaariumid üle vaadata, täiendada ning kinnitada.

3.6 BDD piirangud

Behaviour driven development ei sobi igaks tarkvaraprojektiks. Nõuete kirjutamine ja koodi või testide arendamine on seotud lisa ajakuluga, mis ei ole alati äriliselt õigustatud. Näiteks alustavatel ettevõtetel, kes alles otsivad kinnitust oma äriidee toimimisele ei ole mõistlik panustada ülemäära palju ressurssi tarkvara kvaliteeti, kuna nõuded võivad igapäevaselt muutuda.

Samuti tehniliste programmide nõuete verifitseerimist nagu kopeerimise käsk terminalis, on mõistlikum lahendada ühiktestidega või käsitsi testides kuna rakenduse tellijad on üldjuhul ise tehniliselt piisavalt pädevad.

4 Spetsifikatsioon

BDD annab võimaluse arendusorganisatsioonil parendada suhtlust tellijaga ning hallata nõudeid viisil, mis annab hea ülevaate rakenduse seisukorrast. Nõuete muutumine ei too kaasa käsitsi kogu rakenduse üle testimist ning infosüsteemi töökvaliteet on oluliselt parema kontrolli all. Punkt 3.4 kirjeldab erinevaid takistusi, mis pidurdavad BDD laialdasemat levikut.

Antud töö oluliseks tulemuseks on tarkvara, mida saab edukalt rakendada BDD arenduspraktikat kasutavates tarkvaraprojektides ning mis võimaldab aktiivselt kaasata kõiki projekti osapooli rakenduse nõuete koostamisel.

Kuna edaspidine töö räägib nii BDD arenduseks mõeldud tarkvara arendusest, kui ka loodavast rakendusest tellijale, siis sõnastuse lihtsustamiseks nimetame BDD töövahendit edaspidi nimega Specify.

Specify rakendusel on kaks peamist ülesannet. See võimaldab hallata rakenduse nõudeid ning saada ülevaadet tarkvaraarenduse järgust andes vastuseid küsimustele, millised nõuded on täidetud ning millised veel tegemata.

Tellija ja arendusüksus ei pruugi töötada samas asukohas ning nende osapoolte riist- ja tarkvaraline kasutus võivad oluliselt erineda. Näiteks võivad probleemiks osutuda erinevad operatsioonisüsteemid ning juhul kui Specify arendada paksu kliendina Windowsi platvormil, siis Linuxi ja OSXi kasutajatel on kasutamine raskendatud või lausa võimatu. Kõige mõistlikum lahendus tarkvara kasutajaliidesele on veebipõhine rakendus, mis võimaldab paremat kättesaadavust ja lihtsustab Specify kasutuselevõttu.

Viimase takistusena toob autor välja integratsiooni versioonihaldusega. Lihtsustatud kasutajaliides muudatuste salvestamiseks kesksesse versioonihaldusesse on oluline funktsionaalsus, mida Specify peab suutma lahendada.

4.1 Nõuded

Enne Specify arenduse algust on vaja kirjeldada tulevase rakenduse nõuded. Kuigi saaks BDD'd järgides arendada rakendust lisades nõudeid ükshaaval ning realiseerida

programmis koodis, siis antud rakenduse suurust arvestades on mõistlikum kirjeldada nõuded ette ära.

Specify nõuded on kirjeldatud eesti keele süntaksit kasutavas Gherkinis. Gherkini võtmesõnad on märgitud rasvases kirjas, et lihtsustada nõuete lugemist. Specify rakenduse siseselt on võtmesõnad eraldatud teise värviga.

4.1.1 Seaded

Selleks, et kirjeldatud nõuded jõuaksid arendajani, peab neid säilitama ning vastavalt punktis 3.4 antud soovitusele on mõistlik nõudeid ja programmikoodi hoida ühtses versioonihaldussüsteemis. Antud töö raames realiseeritakse ainult *git*iga [17] liidestus.

```
Omadus: Seaded
       Rakenduse kasutajana
       tahan ma muuda projekti sätteid,
       et saaksin nõuetes tehtud muudatused sünkroniseerida
versioonihaldusega


Stsenaarium: Seadete vormi avamine
       Kui Ma avan seadete vormi
       Siis Vormil on väli 'Name'
       Ja Vormil on väli 'GitUrl'
       Ja Vormil on väli 'GitUsername'
       Ja Vormil on väli 'GitPassword'


Stsenaarium: Seadete vormi salvestamine
       Eeldades Ma olen avanud seadete vormi
       Ja Ma sisestan väljale 'Name' 'Specify'
       Kui Ma salvestan seaded
       Siis Seaded salvestatakse andmebaasis
```

4.1.2 Faili lisamine

Specify kasutajal on vajadus lisada uusi faile, mida arendusmeeskond tarkvaraarenduse käigus järgib. Kasutusloona näeb nõue välja järgnevalt.

```
Omadus: Uue faili lisamine
	Rakenduse kasutajana
	tahan lisada spetsifikatsiooni uue nõude,
	et täiendada rakenduse nõudeid
```

Faili lisamiseks peab kasutajal olema võimalus avada lisamise vorm, täita faili nimi ning valida kataloog, kuhu uus fail koos muudatustega salvestada. Rakendus peab kontrollima, et antud kataloogis samanimelist faili ei eksisteeriks, et vältida ohtu juhuslikkust üle kirjutamisest.

Nõude täitmiseks peab rakendus läbima järgnevad stsenaariumid.

```
Stsenaarium: Uue faili lisamise vorm
	Eeldades Ma olen rakenduse avanud
	Kui Ma avan faili lisamise vormi
	Siis Ma näen vormi, milles on kaks välja
	Ja Ma näen failinime tekstivälja
	Ja Ma näen kataloogi valikvälja


Stsenaarium: failinime väli on kohustuslik
	Eeldades Ma olen faili avamise vormil
	Kui Ma vajutan salvesta
	Siis Uue faili salvestamist ei toimu
	Ja Ma näen veateadet 'The Filename field is required.'


Stsenaarium: fail eksisteerib kataloogis
	Eeldades Ma olen faili avamise vormil
	Ja Fail nimega 'Specify.Specification\FailiLisamine.feature' on
versioonihalduses
	Ja Ma sisestan failinimeks 'FailiLisamine.feature'
	Ja Ma valin kataloogiks 'Specify.Specification'
	Kui Ma vajutan salvesta
	Siis Uue faili salvestamist ei toimu
	Ja Ma näen veateadet 'File already exists'
```

```
Stsenaarium: Uue faili salvestamine
	Eeldades Ma olen faili avamise vormil
	Ja Ma sisestan failinimeks 'NewFeature.feature'
	Ja Ma valin kataloogiks 'Specify.Specification'
	Kui Ma vajutan salvesta
	Siis Fail salvestatakse
	Ja Mind suunatakse edasi faili muutmise vormile
```

4.1.3 Faili muutmine

Failide sisu muutmiseks kuvatakse kasutajale eraldi vorm, kus tekstiredaktoris on mugav sisestada nõudeid. Antud töö raames on kasutajal võimalik avada ja muuta stsenaariume ning täpsustada kasutuslugu, kuid edasised arendused võiksid võimaldada rohkemat, näiteks salvestamisel kontrollida süntaksi korrektsust.

```
Omadus: Faili muutmine
	Kasutajana
	Ma tahan muuta olemasolevaid faile,
	et vajadusel täiendada rakenduse spetsifikatsiooni
Stsenaarium: Faili avamine
	Eeldades Mu rakenduses järgnevad on nõudefailid
	| Kataloog              | Fail                   |
	| Specify.Specification | FailiLisamine.feature |
	| Specify.Specification | FailiMuutmine.feature |
	Kui Ma valin 'FailiMuutmine.feature' vasakpoolsest menüüst
	Siis Ma näen faili muutmise vormi

Stsenaarium: Faili salvestamine
	Eeldades Mu rakenduses järgnevad on nõudefailid
	| Kataloog              | Fail                   |
	| Specify.Specification | FailiLisamine.feature |
	| Specify.Specification | FailiMuutmine.feature |
	Ja Ma valin 'FailiMuutmine.feature' vasakpoolsest menüüst
	Kui Ma vajutan salvesta nupule
	Siis Muudatused salvestatakse
	Ja Ma näen salvestamise teadet
```

4.1.4 Versioonihaldus

Specify on liidestatud *git* versioonihaldusega, mis on mõeldud hajustööks. Failidest on koopiad kohalikul kõvakettal ning muudatused on seotud lokaalse haruga (*branch*).

Selleks, et teised osapooled saaksid kasutaja tehtud muudatusi kasutada on kõigepealt vaja muudatused kinnitada (*git commit*) ja alles siis saab versiooniharu sünkroniseerida keskse repositooriumiga (*git push*). Tulenevalt *git*'i eripärast on kasutusloo stsenaariumid kohandatud vastavalt.

```
Omadus: Versioonihaldus
       Rakenduse kasutajana
       Soovin salvestada enda muudatused versioonihalduses,
       et teised osapooled saaksid näha muutunud nõudeid


Stsenaarium: Muudatuse kinnitusvormi avamine
       Eeldades Ma olen teinud järgnevaid muudatusi failides
       | Kataloog               | Fail                       | Muudatus |
       | Specify.Specification | Versioonihaldus.feature | Muutunud |
       Kui Ma avan kinnitamise lehe
       Siis Ma näen kinnitusvormi
       Ja Ma saan kinnitada muudatused


Stsenaarium: Muudatuste puudumisel ei saa kinnitada
       Eeldades Ma olen teinud järgnevaid muudatusi failides
       | Kataloog               | Fail                       | Muudetud |
       Kui Ma avan kinnitamise lehe
       Siis Ma näen kinnitusvormi
       Ja Ma ei saa kinnitada muudatusi


Stsenaarium: Muudatuste kinnitamine
       Eeldades Ma olen teinud järgnevaid muudatusi failides
       | Kataloog               | Fail                       | Muudatus |
       | Specify.Specification | Versioonihaldus.feature | Muutunud |
       Ja Ma avan kinnitamise lehe
       Ja Ma sisestan kirjelduse 'Täiendused versioonihalduses'
       Kui Ma kinnitan muudatused
       Siis Muudatused on kinnitatud
       Ja Ma mind suunatakse sünkroniseerimise vormile
```

```
Stsenaarium: Kinnitatud muudatuste sünkroniseerimine
    Eeldades Ma olen kinnitanud järgnevad muudatused
    | CommitId | Message   |
    | a        | Muudatus1 |
    | b        | Muudatus2 |
    Ja Ma avan sünkroniseerimise vormi
    Kui Ma sünkroniseerin muudatused
    Siis Muudatused kajastuvad keskses versioonihalduses
    Ja Ma näen positiivset teadet


Stsenaarium: Sünkroniseerimine ei ole võimalik, kui muudatusi pole
    Eeldades Ma olen kinnitanud järgnevad muudatused
    | CommitId | Message   |
    Kui Ma avan sünkroniseerimise vormi
    Siis Ma ei saa sünkroniseerida
```

4.1.5 Testitulemuste vastuvõtmine

Specify rakendus, hoolimata ligipääsust tarkvara programmikoodile, ei saa kontrollida rakenduse vastavust spetsifikatsioonile. Selline ülesanne oleks keeruline ning ei lisaks mingit erilist väärtust, kuna rakenduse kompileerimiseks, seadistamiseks ja paigaldamiseks on olemas eraldi vahendid. Testimise tulemused, mis tekivad CI/CD töövoo käigus peavad jõudma Specify andmebaasi ja seejärel võimalik täita ka nõuet 4.1.6.

```
Omadus: Testitulemuste vastuvõtmine
    Süsteemina,
    tahan saada testimise tulemusi,
    et säilitada neid hilisemaks analüüsiks.
```

```
Stsenaarium: Testitulemuste vastuvõtt
       Eeldades Rakenduses on süsteemne liides
       Kui liidesele postitatakse testide tulemused
       Siis tulemused salvestatakse andmebaasis
```

4.1.6 Ülevaade nõuete täitmisest

Kasutaja soovib igal ajahetkel saada ülevaadet rakenduse seisust. Vajalik informatsioon on jagunenud kahe andmeallika vahel, versioonihalduses on kirjas kõik rakenduse nõuded ning andmebaasis on teada viimase testimise tulemused. Ülevaate saamiseks on vaja kuvada tulemustabel.

```
Omadus: Tulemuste ülevaade
       Rakenduse kasutajana
       Soovin saada ülevaadet nõuete täitmisest,
       et planeerida edasist arendust


Taust:
  Eeldades Süsteemis on järgnevad testitulemused
    | Feature        | Scenario                        | Result | StartTime  |
    | Uus fail       | Uue faili lisamise vorm         | Passed | 2018-04-26 |
    | Uus fail       | failinime väli on kohustuslik   | Passed | 2018-04-26 |
    | Faili muutmine | Faili avamine                   | Failed | 2018-04-26 |
    | Seaded         | Seadete vormi salvestamine      | Failed | 2018-04-26 |
    | Seaded         | Seadete taaslaadimine           | Passed | 2018-04-26 |
  Ja Versioonihalduses on järgnevad nõuded
    | Omadus         | Stsenaarium                     | Muutja     |
    | Uus fail       | Uue faili lisamise vorm         | John Smith |
    | Uus fail       | failinime väli on kohustuslik   | John Smith |
    | Faili muutmine | Faili avamine                   | John Smith |
    | Seaded         | Seadete vormi avamine           | John Smith |
    | Seaded         | Seadete vormi salvestamine      | John Smith |
```

Andmeallikate kirjeldamisel saab lühendatult kirja panna ka edasised stsenaariumid, mida Specify peab täitma.

```
Stsenaarium: Testitud stsenaarium on näha
       Kui Ma avan ülevaate lehekülje
       Siis Ma näen ülevaadet
       Ja Omadus 'Uue faili lisamine', stenaarium 'Uue faili lisamise vorm'
on näha

Stsenaarium: Testiimata stsenaarium on näha
       Kui Ma avan ülevaate lehekülje
       Siis Ma näen ülevaadet
       Ja Omadus 'Seaded', stenaarium 'Seadete vormi avamine' on näha

Stsenaarium: Kustutatud stsenaariumit pole näha
       Kui Ma avan ülevaate lehekülje
       Siis Ma näen ülevaadet
       Ja Omadus 'Seaded', stenaarium 'Seadete taaslaadimine' ei ole näha
```

4.2 Arenduspõhimõtted

BDD töövahendi arendamisel on lähtutud tarkvaraarenduse headest praktikatest rakenduse ülesehituse ja programmikoodi struktureerimisel. Esmane arenduspraktika on loomulikult BDD ning ühtegi uut funktsionaalsust ei lisata rakendusele ennem, kui on olemas nõue koos stsenaariumitega. BDD arenduspraktika kasutamine projektis ei takista TDD kasutuselevõttu. Täpsemalt on kombineeritud arenduspraktikaid kirjeldatud punktis 4.3.

Selleks, et edukalt kasutusele võtta TDD peab rakenduse kood olema testitav. Iga arendatav komponent peab olema eraldatud muudest komponentidest sellisel viisil, et oleks võimalik väliseid komponente anda edasi testides. Siinkohal on kasutusel objektorienteeritud programmeerimise disaini põhimõte nimetusega SOLID [18]. SOLID on viie erineva disainipõhimõtte kooslus ning ühiktestimise hõlbustamiseks on oluline jälgida *Dependency inversion principle* printsiipi, mis sunnib kõik klassi toimimiseks vajalikud sõltuvused asendama abstraktsioonidega.

4.3 BDD arenduse näide ühe stsenaariumi põhjal

Specify arendamisel kasutab autor BDD arenduspraktikat koos TDDga. Kuna BDD keskendub rakenduse käitumise kirjeldamisele lõppkasutaja vaatepunktist, siis jääb

tahaplaanile programmi tehniline disain. Antud puudust adresseerib TDD arenduspraktika ning nende kombineerimine on autori arvates mõistlik viis tarkvara arendada. Punkt 3.3 kirjeldab üldistatult BDD ja TDD kooskasutust ning antud punktis kirjeldab autor nende kasutust täpsemalt ühe stsenaariumi näitel.

Näitena on kasutatud versioonihalduse omaduse (punkt 4.1.4) stsenaariumit „Muudatuste kinnitamine“. Stsenaarium algab sellest, et kasutaja on teinud lokaalse muudatuse ning soovib seda kinnitada. Kinnitamisel salvestatakse muudatus kohalikus harus, misjärel rakendus suunab kasutaja sünkroniseerimise leheküljele.

Lokaalse muudatuse kinnitamist viib läbi *ProjectController* ning kogu toodangu kood on minimaalne, jagatuna kolmeks osaks: väljad ja nende initsialiseerimine konstruktoris, kinnitamislehe kuvamine ja kinnitamine.

```
public class ProjectController : Controller
{
    private readonly IMapper _mapper;
    private readonly ISourceControl _sourceControl;

    public ProjectController(IMapper mapper, ISourceControl sourceControl)
    {
        _mapper = mapper;
        _sourceControl = sourceControl;
    }

    public ActionResult Status()
    {
        var status = _sourceControl.GetStatus();

        return View(_mapper.Map<StatusViewModel>(status));
    }

    [HttpPost]
    public ActionResult Commit(StatusViewModel model)
    {
        _sourceControl.Commit(_mapper.Map<CommitRequest>(model));
        return RedirectToAction("Sync", "SourceControl");
    }
}
```

Stsenaariumi käsustiku saab jagada eraldi osadeks, mis võimaldab verifitseerida stsenaariumis kirjeldatud nõude täitmist. Microsoft .NETi [19] platvormil arendades on BDD arendusteegiks SpecFlow [7], kus Gherkini sisendfail ja testikood ühendatakse omavahel kasutades sammude definitsioone (*step definitions*) ning testikood peab olema dekoreeritud vastava atribuudiga. Sama atribuudi nõue kehtib ka kõigile stsenaariumi sammudele ning mis on nähtavad edasistest koodinäidetest.

Testi definitsioonis seadistatakse testitav komponent (*ProjectController* _projectController) ning kontrolleriga seotud komponendid (*IMapper* _mapper, *ISourceControl* _sourceControl). Testimise tulemuste verifitseerimiseks seatakse ka üles mõningad abimuutujad (*ActionResult* _result, *StatusViewModel* _statusViewModel), mida üks samm täidab ja järgnev samm verifitseerib. Seadistamine toimub iga stsenaariumi alguses, et vältida eelmiste stsenaariumite mõju või järgnevate

stsenaariumite mõjutamist. Kuna kõik komponendid on üksteisest eraldatud abstraktsioonidega, siis kontrolleri testimises ei kasutata *ISourceControli* implementatsiooni klassina vaid kasutusel on Moq [20] teek. Moq lihtsustab seotud komponentide erinevate käitumiste imiteerimist (meetod *Setup*) ja kontrollimist (meetod *Verify*).

```
[Binding]
public class VersioonihaldusSteps
{
  private SourceControlController _vcController;
  private ProjectController _projectController;
  private IMapper _mapper;
  private Mock<ISourceControlMediator> _mediator;
  private ActionResult _result;
  private StatusViewModel _statusViewModel;

  [Before]
  public void Before()
  {
    _mapper =
Infrastructure.MapperConfiguration.BuildConfiguration().CreateMapper();
    _mediator = new Mock<ISourceControlMediator>();
    _projectController = new ProjectController(_mapper, _mediator.Object);
  }
```

Stsenaariumi esimene samm eeldab, et rakenduse spetsifikatsioonis on tehtud lokaalseid muudatusi. Gherkin võimaldab sammu sisendit edastada ka tabeli kujul ning sammu testikoodis on võimalik seda tabelit lugeda.

```
[Given(@"Ma olen teinud järgnevaid muudatusi failides")]
public void EeldadesMaOlenTeinudJargnevaidMuudatusiFailides(Table table)
{
  var statusItems = table.Rows.Select(row => new StatusItem {
    Type = StatusType.Modified,
    FileItem = new FileItem {
      Name = row["Fail"],
      Path = Path.Combine(row["Kataloog"], row["Fail"])
    }
  }).ToArray();
  _sourceControl.Setup(a => a.GetStatus()).Returns(() => statusItems);
}
```

Antud sammu realisatsioonis on välja toodud kuidas SpecFlow's näeb välja sammu defineerimine (esimesel real olev meetodi atribuut). Samuti on näha kuidas omistatakse seotud komponendile käitumine kasutades stsenaariumi parameetrina tulnud tabelit.

Järgneva sammuna läheb kasutaja muudatuse kinnitamise lehele ning siin saab juba kontrollida rakenduse käitumist. Kontrollitakse, kas kinnitamise lehe avamine tõi oodatud tulemuse (*ViewResult*), kas tulemuses on õige andmemudel ja kas andmed tulid väliselt komponendilt.

```
[Given(@"Ma avan kinnitamise lehe")]
public void KuiMaAvanKinnitamiseLehe()
{
  _result = _projectController.Status();
  _result.As<ViewResult>().Model.As<StatusViewModel>().Should().NotBeNull();
  _sourceControl.Verify(a => a.GetStatus(), Times.Once);
}
```

Stsenaariumi järgi on kasutaja järgmine tegevus kinnitusvormile sisestada kommentaar, mis testkoodis omistab päringu väärtused. Kommentaari väärtus jõuab testkoodi meetodisse parameetrina.

```
[Given(@"Ma sisestan kirjelduse '(.*)'")]
public void EeldadesMaSisestanKirjelduse(string message)
{
  _statusViewModel = new StatusViewModel
  {
    Message = message
  };
}
```

Stsenaariumi kinnitamise samm postitab eelmises sammus koostatud päringu *ProjectController'*i meetodisse ning omistab tulemuse vahemuutujasse.

```
[When(@"Ma kinnitan muudatused")]
public void KuiMaKinnitanMuudatused()
{
  _result = _projectController.Commit(_statusViewModel);
}
```

Järgnevad sammud kontrollivad, et kinnitamise protsess on sooritatud ning kasutaja suunatakse edasi sünkroniseerimise vormile.

```
[Then(@"Muudatused on kinnitatud")]
public void SiisMuudatusedOnKinnitatud()
{
  _sourceControl.Verify(a=>a.Commit(It.IsAny<CommitRequest>()), Times.Once);
}

[Then(@"Ma mind suunatakse sünkroniseerimise vormile")]
public void SiisMaMindSuunatakseSunkroniseerimiseVormile()
{
  var redirect = _result.As<RedirectToRouteResult>().RouteValues;
  redirect["Action"].Should().Be("Sync");
  redirect["Controller"].Should().Be("SourceControl");
}
```

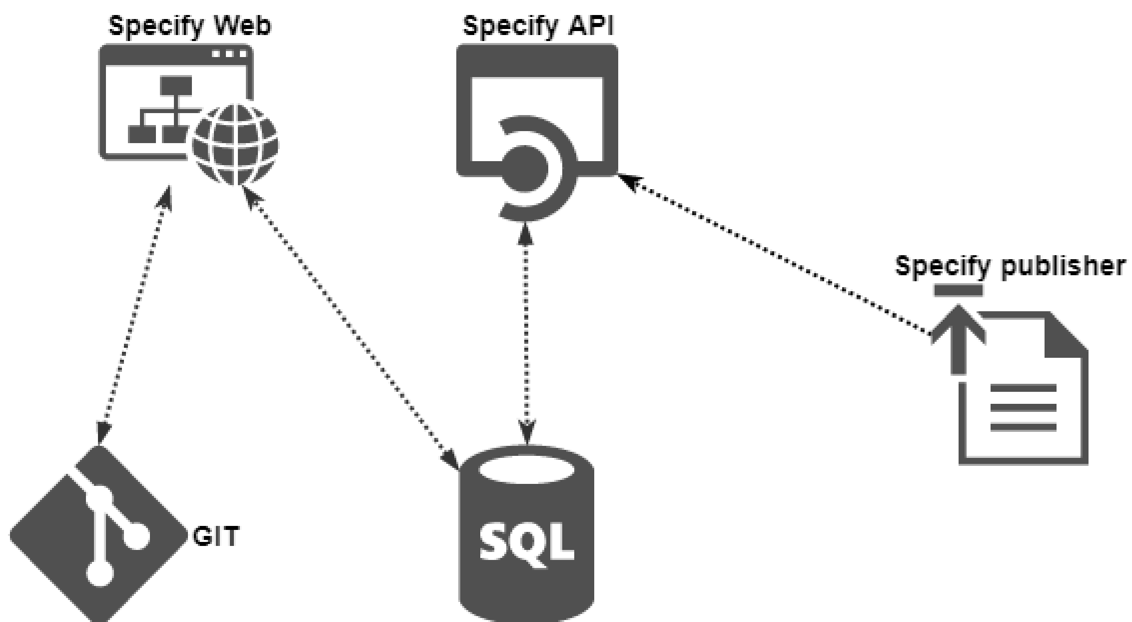
Kogu stsenaarium on nüüd testitav ja saame verifitseerida rakenduse käitumist vastavalt spetsifikatsioonile. Reaalsuses sellest programmikoodist ei piisa, et tarnida tellijale

nõutud funktsionaalsus, kuna versioonihalduse komponendi (*ISourceControl*) käitumisloogika on arendamata ja ka testimata. Versioonihalduse komponendi arendust ja testimist saab teostada kasutades TDD praktikat, kirjutades ühikteste.

4.4 Arhitektuur ja tehniline lahendus

Kogu rakenduse realisatsiooniks vajalikud komponendid on identifitseeritud ning jagunevad kõrgemal tasemel järgnevalt:

1. Specify Web – Veebipõhine kasutajaliides, mis on kasutajate peamine töövahend.
2. Specify Publisher – käsurea programm, mis edastab testimise tulemused kesksesse andmebaasi.
3. Specify API – REST API veebiliides, mis võtab vastu Publisherilt tulnud testimise tulemused.
4. Andmebaas – Testitulemuste ja muude lisaandmete säilitamiseks.
5. Versioonihaldus – Nõuete säilitamiseks.



Joonis 7. Komponentide diagramm

Arendusvahenditena on kasutusel Microsofti tehnoloogiatel põhinevad vahendid. Alusraamistikuks on Microsofti .NET [16] ja programmeerimiskeeleks C# [17], mis on

staatiline tüübikontrolliga objektorienteeritud keel ja võimaldab arendada erinevat suurusjärku tarkvaralahendusi mikrokontrolleritest kuni suurte pilvelahendusteni.

Peamiseks töövahendiks on Microsoft Visual Studio [18], mis on enimlevinud arendajatele mõeldud töövahend Microsofti .NET platvormil töötamiseks. Erinevad lisad, mida on võimalik paigaldada Visual Studiole lihtsustavad tööd veel. Autoril oli töö kirjutamisel ajal kasutuses ReSharper [19] (enimlevinud produktiivsuse lisa) ja SpecFlow [7].

Veebipõhine kasutajaliides baseerub Microsoft ASP.NET MVC [20] raamistikul, kasutusel olev versioon on 5.2.4. ASP.NET MVC omab ka uuemat versiooni nimetusega ASP.NET MVC Core, mis on oluliselt kergekaalulisem ja sobiks antud ülesande lahendamiseks paremini. Kuna SpecFlow raamistik ei ole veel kohandatud töötamaks .NET Core platvormil, siis üldise keerukuse vähendamiseks on kasutusel vanema põlvkonna veebiraamistik. Kasutajaliidese kujunduses on kasutatud CoreUI [21] malli, mis baseerub Bootstrap [22] raamistikul.

Integratsioon Specify ja *continuous integrationiga* teostatakse läbi REST API veebiteenuste, mis baseerub Microsoft ASP.NET Web API raamistikul ning kasutusel olev versioon on 5.2.4.

Andmevahetuskiht kasutab andmebaasiga suhtluseks Entity Framework [23] teeki. Andmebaasiks on valitud Microsoft SQL [24] versioon 2017, mis sobib hästi valitud teekidega. Andmebaasis hoitakse ainult andmeid ning andmebaas ei tea midagi äriloogikast, välja arvatud olemite omavahelisi seoseid.

Versioonihaldus, millega integreeritakse on git [25] ja .NETis on gitiga suhtlemiseks olema teek nimetusega libgit2sharp [26].

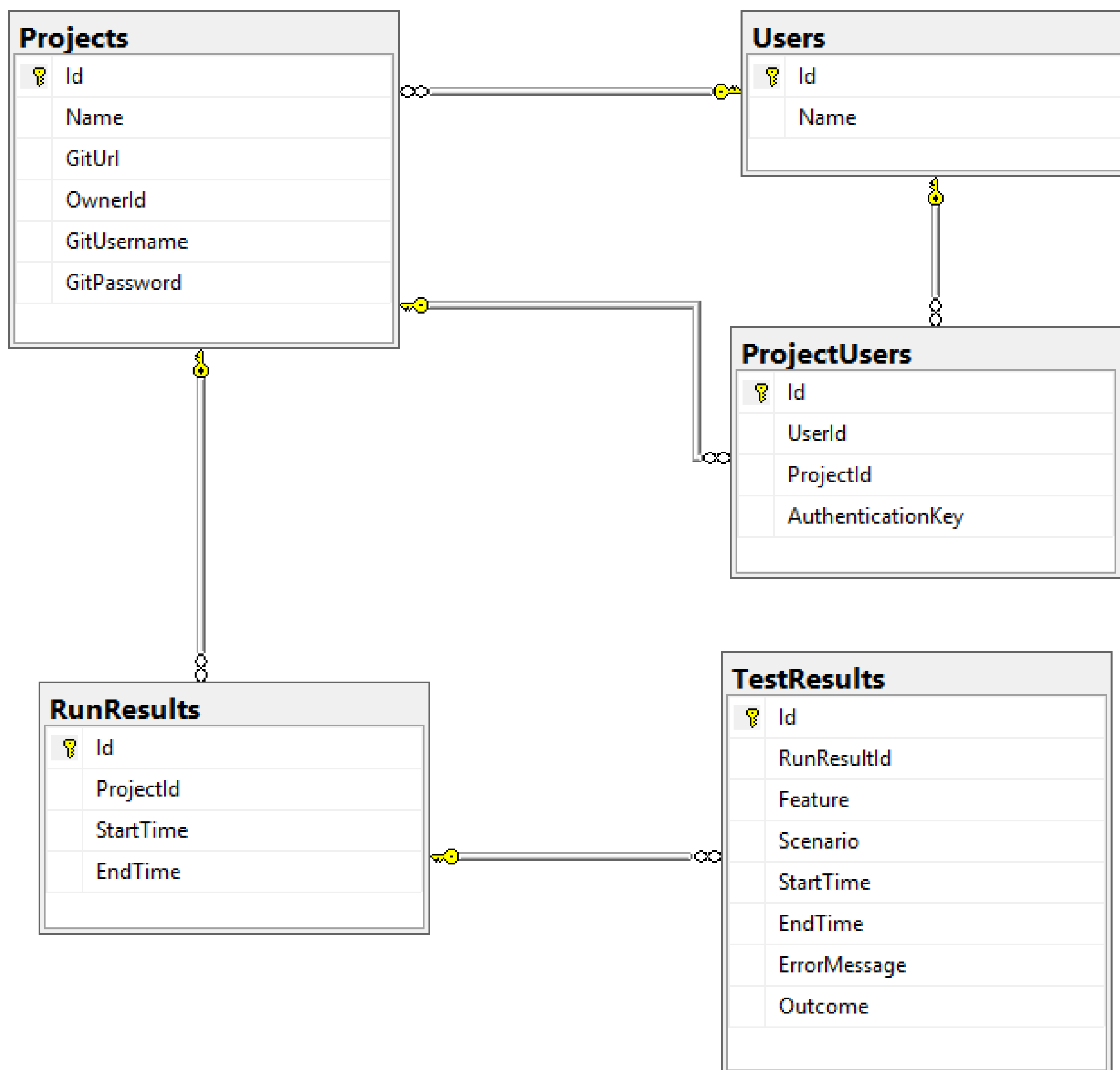
Kasutajaliidese oluliseks osaks on redigeerimisvahend, mis võimaldab kasutajal lisada ning muuta nõudeid Gherkini formaadis. Süntaksiga seotud vigade lihtsamaks leidmiseks on oluline, et kasutajale kuvatakse spetsifikatsiooni nii, et võtmesõnad, sisu ja struktuur oleksid kergesti eristatavad. Kasutusel on avatud lähtekoodiga Ace [27] redaktor, millel on tugi Gherkini keele süntaksile. Autor kaalus ka Microsoft Monaco Editori [28], kuid sellel redaktoril puudus Gherkini keele tugi, ning keele toe lisamist takistas puudulik informatsioonivahetus rakenduse ja redaktori vahel.

4.5 Andmemudel

Analüüsi käigus on tuvastatud olemid, mida on vaja salvestada andmebaasi, et hoida töö käigus tekkivate andmete säilimine ka päringute vahelisel ajal. Kõik andmebaasi olemid on nimetatud inglise keelsetena, et need oleksid kooskõlas programmikoodis kasutusel olevate olemitega.

Olemite primaarsed võtmed on andmetüübiga UUID (versioon 4), ka välisvõtmed on seetõttu sama andmetüübiga. UUIDde kasutamine võtmete väljadena toob kaasa jõudluskao suuremates andmemahtudes, kuid antud Specify versioonis ei ole andmete hulk probleemiks. Kuupäevaväljades on kasutusel datetimeoffset andmetüüp, mis salvestab endas kuupäeva, kellaaja ja ajatsooni informatsiooni.

Tabelites Project, User ja ProjectUser hoitakse projekti ning kasutajaga seotud andmeid, versioonihalduse ligipääsutunnuseid ja asukohta. RunResults ning TestResults tabelites säilitatakse testimise tulemusi.



Joonis 8. Andmebaasimudel

4.6 Detailne Specify struktuur

Rakendus on jaotatud erinevateks komponentideks, mis omakorda jagunevad kihtideks. .NETi rakendused paigutatakse lahendusse (*solution*) ning iga alamkomponent eraldi projekti. Kogu lahendus on jaotatud neljateistkümne projekti vahel ning igal projektil on oma kindel ülesanne.

Keskne äriloogika asub Specify.Core projektis ja pea kõik teised projektid viitavad sellele projektile aga Core projekt ise ei viita kuskile (va. Specify.Common, projekt kus on konstandid ja abifunktsioonid). Projektis olev äriloogika vajab sisemiseks toimimiseks komponente, mis suhtlevad versioonihalduse ja andmebaasiga, kuid ühiktestimise hõlbustamiseks on nad eraldatud tuumikprotsessides abstraktsioonidena.

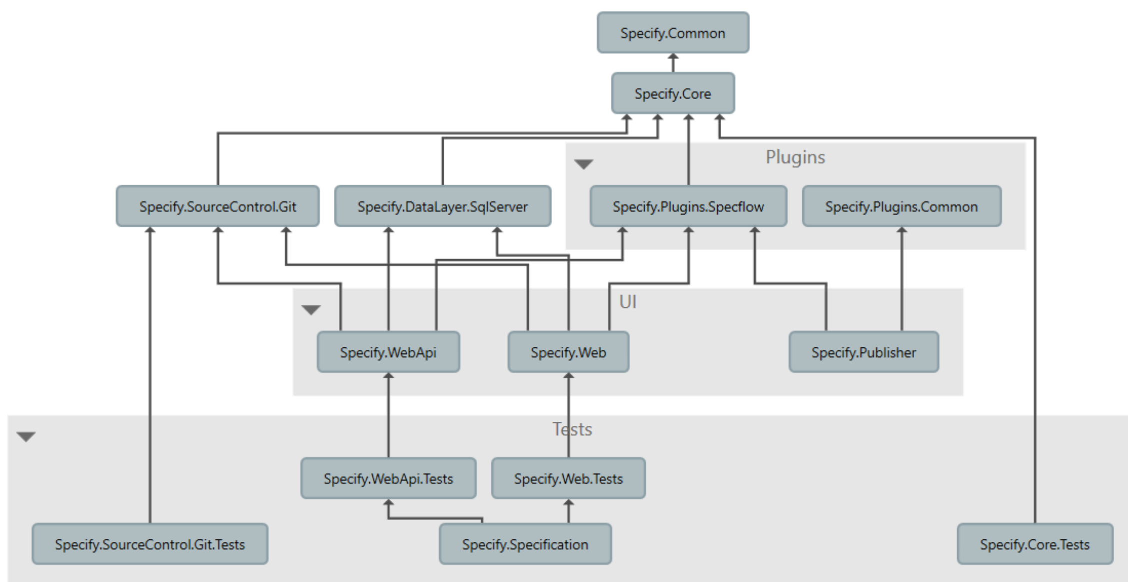
Git versioonihalduse komponent on eraldatud projekti Specify.SourceControl.Git, mis realiseerib kogu vajamineva funktsionaalsuse, mis on seotud versioonihaldusega. Komponendil on ka omad ühiktestid, mis tegelevad peamiselt Specify andmemudelite ja kasutatud teegi andmete teisendamise testimisega mõlemal suunal. Reaalse versioonihalduse testid on välja jäetud, kuna ei ole antud töö skoobis ning väliste komponentide testimine ei ole üldjuhul otstarbekas. Juhul kui on vaja välise komponendi töökindlust testida siis peaks kaaluma komponendi väljavahetamist mõne paremini töötava vastu.

Specify.DataLayer.SqlServer on andmebaasi komponent, mis vastutab projektis vajalike andmete säilitamisega andmebaasis. Kasutusel olev väline teek Entity Framework [27] võimaldab ka hallata andmebaasi struktuuri ja selle muutusi ning mõlemad omadused on ka Specify projekti raames ära kasutatud. Kui Specify rakendus käivitatakse esimest korda siis genereeritakse uude andmebaasi kõik vajaminevad tabelid ning täidetakse alusväärtustega.

Rakendus vajab oma tööks võimekust Gherkini faile töödelda sellisel moel, et omaduste ja stsenaariumite pealkirjad oleksid struktuursel moel kättesaadavad. Antud funktsionaalsus on eriti vajalik testitulemuste kuvamisel ülevaate lehel (punkt 4.1.6) ning realiseerimine on viidud projekti Specify.Plugins.SpecFlow ja Specify.Plugins.Common.

Rakenduse nõuded on kirjeldatud projektis Specify.Specification ning seotud testimise programmikood asub Specify.Web.Tests ja Specify.WebApi.Tests projektides. Kasutajaliidese ja äriloogika sidumine toimub vastavalt Specify.Web ja Specify.WebApi projektides, mis omakorda kutsuvad välja Specify.Core projekti komponendid.

Lõpptulemusena on lahenduse struktuur ja projektide omavahelised seosed kujutatud **Error! Reference source not found.Error! Reference source not found.**.



Joonis 9. Projektide omavahelised seosed

4.7 Lahenduse verifitseerimine

Specify, kui rakenduse verifitseerimine jäi peamiselt arendusfaasis lisatud nõuete testimisele koodi tasandil. Kuna rakendus saavutas oma praktilise kasutatavuse alles kõikide nõuete täitmisel, siis arenduse ajal käivitatud testide tulemused olid ainukesed verifitseerimise allikad.

Valminud rakendus testiti ka käsitsi üle, kasutades nõuetes kirjeldatud stsenaariume ning tulemus oli samasugune kui automaattestimisel. BDD testides, mis on mõeldud veebirakendustena, võib kasutada ka automatiseeritud veebilehitsejaid, kuid antud töö raames piirduti kooditestidega.

Specify valideerimine kasutusmugavuse aspektist on väljaspool töö skoopi. Autori arvates oleks selline uuring väga huvitav ning annaks vastused, kas ja kuidas antud töövahendit analüütikut kasutaksid.

5 Kokkuvõte

Behaviour driven development on arenduspraktika, mis annab tarkvaraprojektis osalevatele pooltele ühise keele ning aitab vähendada valesti mõistmisest tekkivat kadu. Teisalt masinkäivitatavad vastuvõtutestid tõstavad arendatava rakenduse töökindlust ja parendavad kohanemist muutunud nõuetega.

Antud töö raames on välja toodud BDD puudused, mis takistavad arenduspraktika laiemat levikut ning kasutusele võtmist.

Üks vajakajäämistest on piisavalt hea töövahendi puudumine, mida analüütikud saaksid kasutada nõuete haldamisel. Antud puuduse leevendamiseks lõi autor vastavasisulise veebipõhise rakenduse Specify, mis annab kõigile projekti osapooltele võimaluse täiendada projekti nõudeid või saada operatiivset ülevaadet rakenduse vastavusest nõuetele. Rakendus on verifitseeritud vastamaks nõuetele nii automatiseeritud ühiktestidega kui ka rakendust kasutades.

BDD efektiivsust pole autori arvates piisavalt uuritud ning kindlasti on vaja kontrollida Specify töövahendi võimekust arendusprotsessi parendamisel. Ühe uurimissuunana pakub autor välja erinevate arenduspraktikate kõrvutamise samaväärsete meeskondade poolt, kes kasutaksid erinevaid testimis- ja arendusmetoodikaid (BDD, TDD, käsitsi). Sama ülesannet lahendades oleks võimalik võrrelda, mis on iga metoodika tugevused ja nõrkused ning kas automatiseeritud testimine hoiab kokku nii aja- kui finantsressursse.

Kasutatud kirjandus

[1] C. Jones, „Software benchmarking,“ *Computer,* pp. 102-103, October 1995.

[2] W. Baziuk, „BNR/NORTEL: path to improve product quality, reliability and customer satisfaction,“ *Software Reliability Engineering*, abc, 1995.

[3] S. S. Park ja F. Maurer, „The Benefits and Challenges of Executable Acceptance Testing,“ *Proceedings - International Conference on Software Engineering*, 2008.

[4] D. North, „Introducing BDD | Dan North & Associates,“ March 2006. [Võrgumaterjal]. Available: https://dannorth.net/introducing-bdd/. [Kasutatud 2018-04-25 April 2018].

[5] Wikipedia, „Domain-driven design,“ 16 April 2018. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Domain-driven_design. [Kasutatud 25 April 2018].

[6] Wikipedia, „Test-driven development,“ 10 April 2018. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Test-driven_development. [Kasutatud 25 April 2018].

[7] I. Bolic, „Ghekrin Wiki,“ 20 November 2017. [Võrgumaterjal]. Available: https://github.com/cucumber/cucumber/wiki/Gherkin. [Kasutatud 25 04 2018].

[8] „What is JBehave?,“ 20 March 2018. [Võrgumaterjal]. Available: http://jbehave.org/. [Kasutatud 20 March 2018].

[9] „RSpec: Behaviour Driven Development for Ruby,“ 20 March 2018. [Võrgumaterjal]. Available: http://rspec.info/. [Kasutatud 20 March 2018].

[10] „SpecFlow - Binding Business Requirements to .NET Code,“ 20 March 2018. [Võrgumaterjal]. Available: http://specflow.org/. [Kasutatud 20 March 2018].

[11] „Behat - a php framework for autotesting your business expectations.,“ 20 March 2018. [Võrgumaterjal]. Available: http://behat.org/en/latest/. [Kasutatud 20 March 2018].

[12] K. Beck ja C. Andres, Extreme Programming Explained, Addison-Wesley, 2004.

[13] Hiptest, „Hiptest pricing,“ [Võrgumaterjal]. Available: https://hiptest.com/pricing/. [Kasutatud 25 April 2018].

[14] Microsoft, „Pricing and Purchasing Options,“ Microsoft, [Võrgumaterjal]. Available: https://www.visualstudio.com/vs/pricing/. [Kasutatud 25 April 2018].

[15] D. Ho, „Notepad++," [Võrgumaterjal]. Available: https://notepad-plus-plus.org/. [Kasutatud 25 April 2018].

[16] „Syntax highlighting for Gherkin (.feature files) in Notepad++," 29 May 2013. [Võrgumaterjal]. Available: https://github.com/famished-tiger/gherkin-highlighting. [Kasutatud 25 April 2018].

[17] Wikipedia, 11 April 2018. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Continuous_integration. [Kasutatud 25 April 2018].

[18] „Spoken languages," 2 May 2017. [Võrgumaterjal]. Available: https://github.com/cucumber/cucumber/wiki/Spoken-languages. [Kasutatud 25 April 2018].

[19] K. Beck, M. Beedle ja jpt, „Agiilse tarkvaraarenduse manifest," 2001. [Võrgumaterjal]. Available: http://agilemanifesto.org/iso/et/manifesto.html. [Kasutatud 25 April 2018].

[20] S. Chacon, „Git," [Võrgumaterjal]. Available: https://git-scm.com/. [Kasutatud 25 April 2018].

[21] Wikipedia, „SOLID (object-oriented design)," 09 March 2018. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design). [Kasutatud 28 April 2018].

[22] Microsoft, „.NET," [Võrgumaterjal]. Available: https://www.microsoft.com/net/. [Kasutatud 25 April 2018].

[23] D. Cazzulino, „The most popular and friendly mocking framework for .NET," [Võrgumaterjal]. Available: https://github.com/moq/moq. [Kasutatud 29 April 2018].

[24] Microsoft, „C# Guide," Microsoft, 30 January 2018. [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/. [Kasutatud 25 April 2018].

[25] Microsoft, „Visual Studio IDE, Code Editor, VSTS, & App Center," [Võrgumaterjal]. Available: https://www.visualstudio.com/. [Kasutatud 25 April 2018].

[26] „ReSharper: Visual Studio Extension for .NET Developers by JetBrains," JetBrains, [Võrgumaterjal]. Available: https://www.jetbrains.com/resharper/. [Kasutatud 25 April 2018].

[27] Microsoft, „ASP.NET | The ASP.NET Site," [Võrgumaterjal]. Available: https://www.asp.net/. [Kasutatud 25 April 2017].

[28] Ł. Holeczek, „CoreUI - Open Source Boostrap Admin Template," [Võrgumaterjal]. Available: https://coreui.io. [Kasutatud 25 April 2018].

[29] J. T. a. B. c. Mark Otto, „Bootstrap · The most popular HTML, CSS, and JS library in the world.," [Võrgumaterjal]. Available: http://getbootstrap.com/. [Kasutatud 25 April 2018].

[30] Microsoft, „Entity Framework | Microsoft Docs," [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/ef/. [Kasutatud 25 April 2018].

[31] Microsoft, „SQL Server 2017 on Windows and Linux," [Võrgumaterjal]. Available: https://www.microsoft.com/en-us/sql-server/sql-server-2017. [Kasutatud 25 April 2018].

[32] G. contributors, „libgit2," [Võrgumaterjal]. Available: https://libgit2.github.com/. [Kasutatud 25 April 2018].

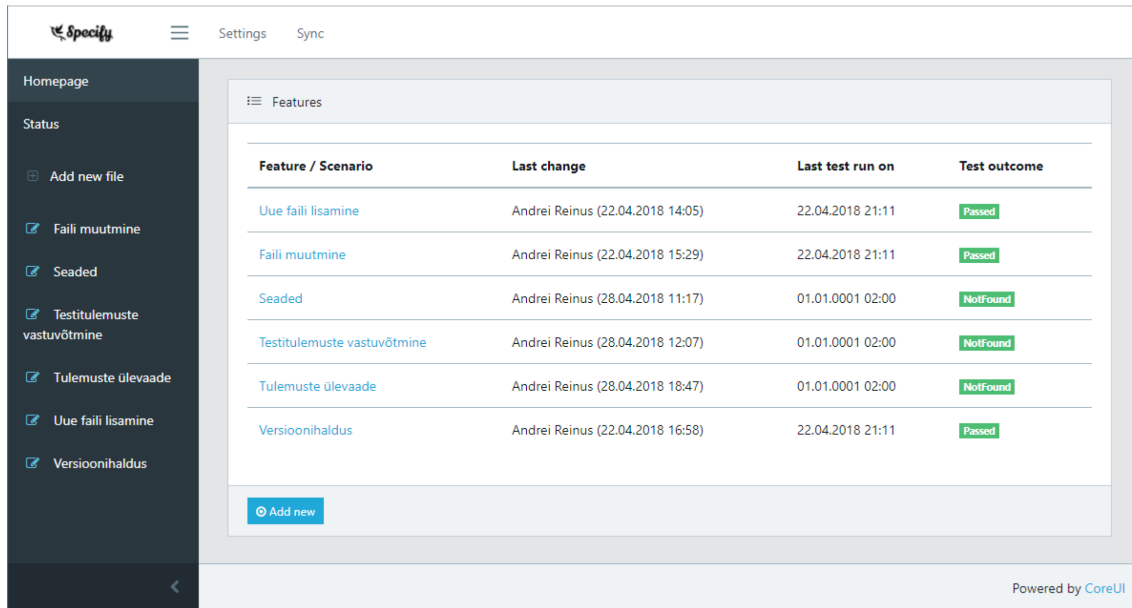
[33] „Ace - The High Performance Code Editor for the Web," [Võrgumaterjal]. Available: https://ace.c9.io. [Kasutatud 25 April 2018].

[34] Microsoft, „Monaco Editor," 2018. [Võrgumaterjal]. Available: https://microsoft.github.io/monaco-editor/. [Kasutatud 25 April 2018].

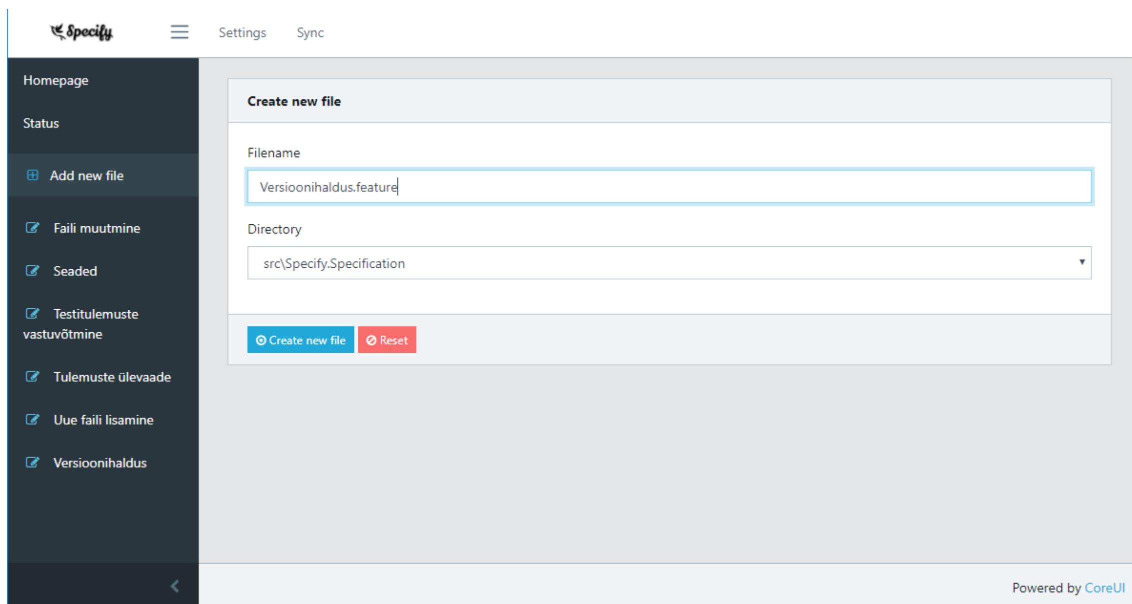
[35] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

[36] J. Bogard, C. Missal, L. Bargaoanu ja T. Carlson, „AutoMapper," [Võrgumaterjal]. Available: https://automapper.org/. [Kasutatud 29 April 2018].

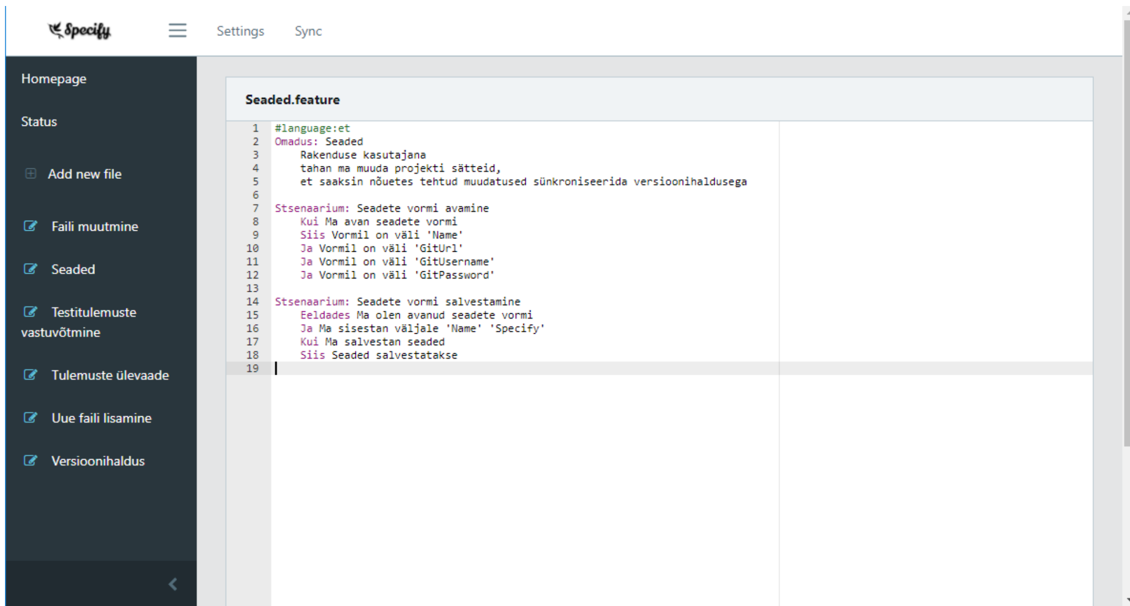
Lisa 1 – Ekraanivaated



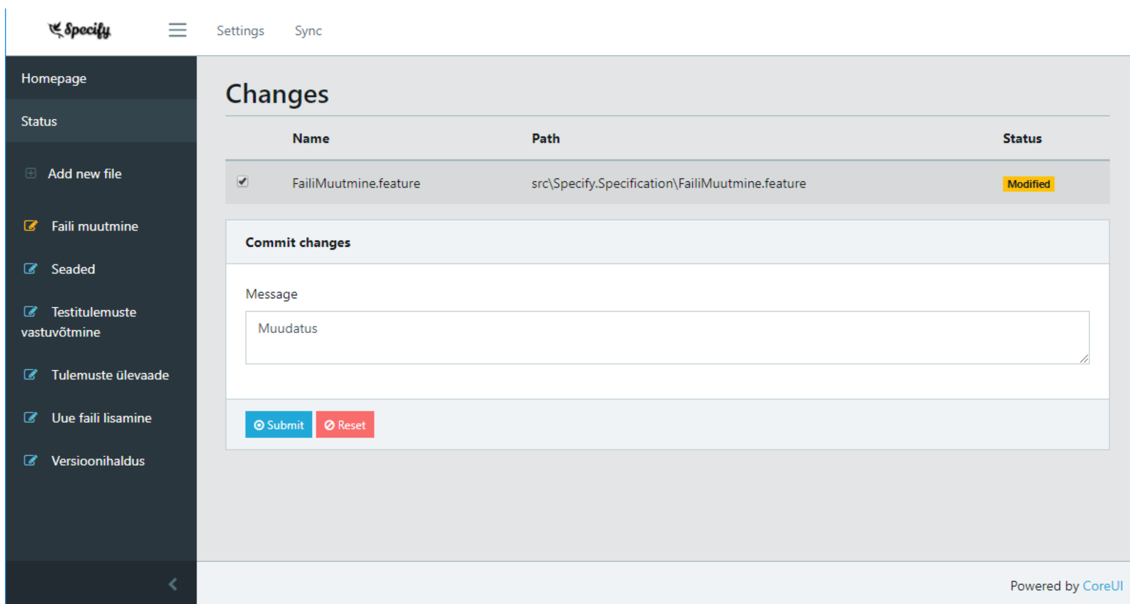
Joonis 10. Ülevaade nõuete täitmisest



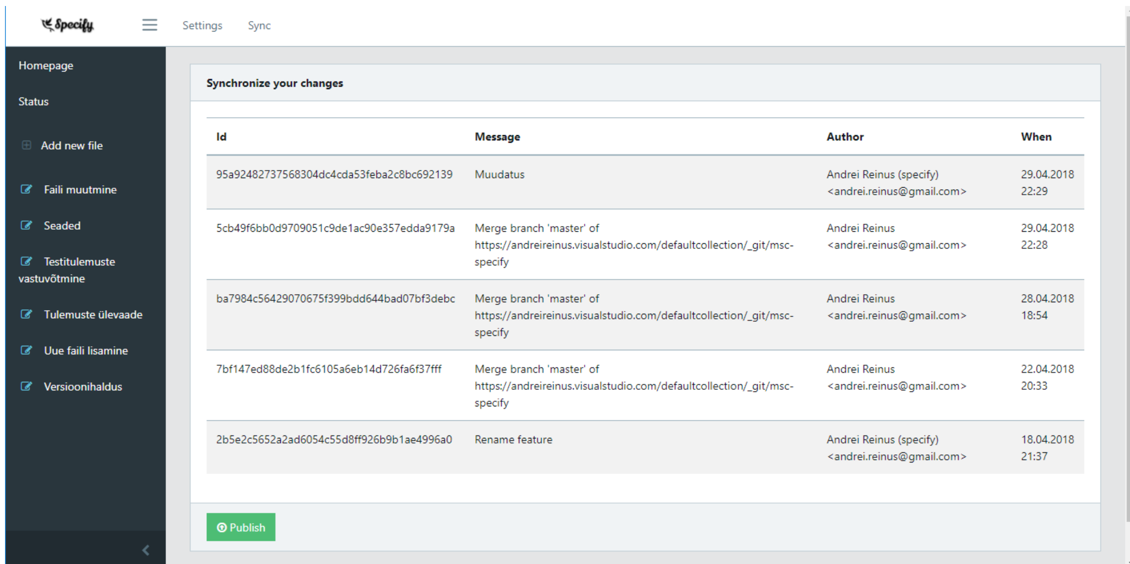
Joonis 11. Faili lisamine



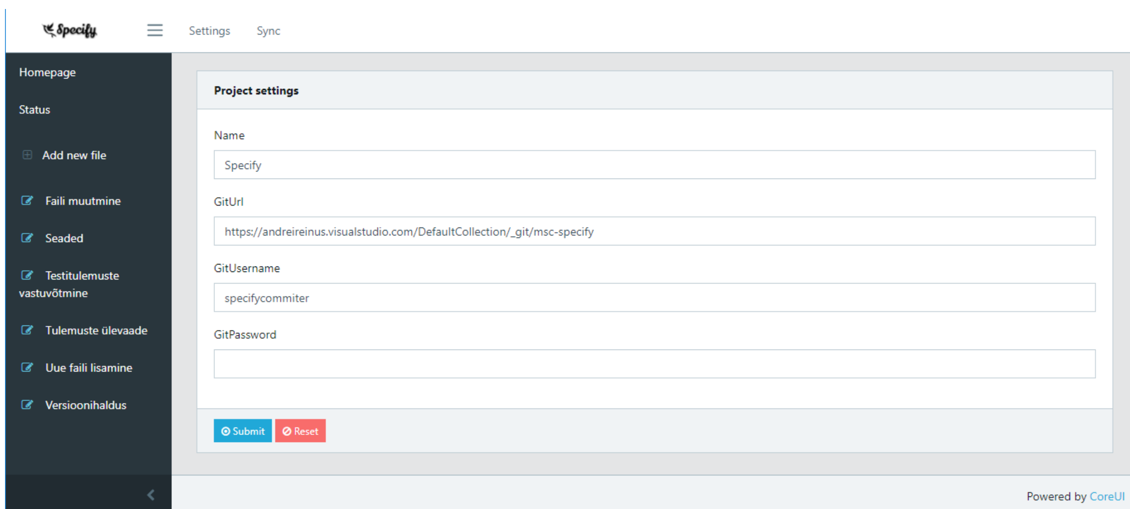
Joonis 12. Faili muutmine



Joonis 13. Muudatuste kinnitusvorm



Joonis 14. Sünkroniseerimisvorm



Joonis 15. Seadete vorm