View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

INWORKS-TUOTTEEN AUTOMAATIOTESTAUS

Dataa vertaileva ohjelmistokehys ja sovellutus Inpulsen laskutukseen

TEKIJÄ/T: Mika Juntunen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala			
Koulutusohjelma Tietotekniikan koulutusohjelma			
Työn tekijä(t) Mika Juntunen			
Työn nimi inWorks-tuotteen automaattitestaus			
Päiväys	21.5.2018	Sivumäärä/Liitteet	41
Ohjaaja(t) Lehtori Jussi Koistinen, Lehtori Keijo Kuosmanen			
Toimeksiantaja/Yhteistyökumppani(t) InPulse Works Oy, Solteq Oyj			

Tiivistelmä

Opinnäytetyön tarkoituksena oli kehittää ratkaisu inWorks-tuotteen laskutuksen automaatio testaukseen. Työn toimeksiantajana aloitti InPulse Works Oy ja kesäkuussa 2017 työn oikeudet siirtyivät Solteq Oyj:lle.

Työn tavoitteena oli kehittää ohjelmistokehys, jonka avulla voidaan toteuttaa testitapauksia, jotka vertailevat koodimuutosten välistä generoitua tai tuotua dataa. Muut tavoitteet olivat toteutttaa testitapaus inWorks-tuotteen laskutukseen käyttämällä tuotettua ohjelmistokehystä sekä käyttöliittymä helpompaan testituloksien tarkasteluun.

Työ kehitettiin Visual Studio ohjelmistokehittimellä ja tärkeimmät tekniikat olivat Microsoftin .NET, ASP.NET MVC, EntityFramework. Tietokantana ratkaisussa toimi Microsoftin SQLServer.

Opinnäytetyön tuloksena saatiin toimiva ohjelmistokehys, jossa testejä voi ajaa ja tulokset kerättyä tietokantaan. Tuloksien selaamiseen onnistuttiin myös toteuttamaan toimiva käyttöliittymäratkaisu. Työtä ei ole vielä otettu käyttöön inWorks-tuotteessa.

Avainsanat
Automaatiotestaus, Ohjelmistotestaus, Asiakastietojärjestelmä, .NET, ASP.NET, C#, EntityFramework
Visual Studio

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Mika Juntunen			
Title of Thesis Automated Testing of inWorks product			
Date	21 May 2018	Pages/Appendices	41
Supervisor(s) Mr Jussi Koistinen, Senior Lecturer and Mr Keijo Kuosmanen, Senior Lecturer			
Client Organisation /Partners InPulse Works Oy, Solteq Oyj			

Abstract

The purpose of this thesis was to develop an automated testing solution for the invoicing of inWorks product. The project was commissioned by InPulse Works Oy and the rights to this project were transferred to Solteq Oyj in June 2017.

The aim of the thesis was to develop a framework where developers can build data comparative test cases for generated or imported data. Other goals of this project were to implement a test case with the created framework for the invoicing of inWorks product as well as a user interface for easier test result viewing.

The end product was developed with Microsoft's Visual Studio and the most important technologies were Microsoft .NET, ASP.NET MVC and EntityFramework. Microsoft's SQLServer was used as the database.

As a result of this thesis, a functional software framework was created, where tests can be run and results collected in the database. Successful browsing of the results was also achieved by implementing a viable user interface solution. The end product has not yet been implemented in the inWorks product.

Keywords
Automated testing, Software testing, Customer information system, .NET, ASP.NET, C#, EntityFramework

Visual Studio

SISÄLTÖ

1 JOHDANTO 7
2 INWORKS 8
2.1 InWorks laskutus 9
2.2 inWorks arkkitehtuuri lyhyesti 10
3 KÄYTETYT TEKNIIKAT, TYÖKALUT JA MÄÄRITELMÄT 11
3.1 .NET Framework 11
3.1.1 C# 11
3.1.2 Rajapinta (Interface) 11
3.1.3 Perintä (Inheritance) 11
3.1.4 ADO.NET 12
3.2 EntityFramework 12
3.2.1 EntityFramework Code-First 12
3.2.2 DbContext 12
3.3 LINQ (Language Integrated Query) 12
3.4 ASP.NET MVC 13
3.4.1 MVC 13
3.4.2 Web API Controller 13
3.5 TypeScript 13
3.6 Visual Studio 13
3.7 SQL Server 14
3.7.1 T-SQL 14
3.8 JQuery 14
3.9 NUnit 14
3.10 NLog 14
3.11 Azure 14
3.11.1 Blob storage 15
3.11.2 Azure Functions 15
3.12 Bootstrap 15
3.13 Git 15
4 OHJELMISTOTESTAUS 16
4.1 Historia 16

4.2 Testauksen perusmääritelmät 17
4.2.1 Virhe (Error, Mistake) 17
4.2.2 Bugi (Bug) 17
4.2.3 Defekti (Defect) 17
4.2.4 Epäonnistuminen (Failure) 18
4.3 Testaustavat 18
4.3.1 Staattinen- ja dynaaminen testaus 18
4.3.2 Musta- ja valkolaatikkotestaus 18
4.4 Testauksen tasot 19
4.4.1 Yksikkötestaus 19
4.4.2 Integraatiotestaus 20
4.4.3 Järjestelmätestaus 20
4.5 Testaustyypit 20
4.5.1 Manuaalinen testaus 20
4.5.2 Automaatiotestaus 21
4.6 Käyttäjätestaus 21
5 TESTIOHJELMISTOKEHYS 22
5.1 Rajapinta 22
5.2 Kantaluokka 23
5.2.1 Testiajon alustus 24
5.2.2 Alkuperäisen datan talteenotto 24
5.2.3 Testiajon aloitus 24
5.2.4 Testidatan vertailu 25
5.2.5 Solukohtaisten datojen vertailu 26
5.2.6 Korjattujen bugien käsittely 27
5.3 Tietokanta 29
5.3.1 EntityFramework datamodel 31
5.3.2 TestFrameWorkDbContext luokka 31
5.3.3 TestFrameWorkRepository 32
6 OHJELMISTOKEHYKSEN SOVELLUTUS INWORKSIN LASKUTUKSEEN 32
6.1 *ITestObject* rajapinnan toteutus laskun generointi testitapauksessa 32
6.2 Testattavan taulun lähtötilanteen talteenotto ja palautus 32
6.3 Laskutustestin ajaminen 33

6.3.1 Vertailtavan datan alustus 33

7 KÄYTTÖLIITTYMÄ 35

7.1 Testiajot 35

7.2 Virheelliset rivit ja solut 36

7.3 Käyttöliittymän rakenne 36

7.3.1 Näkymät 37

7.3.2 Typescript ja DataTables 37

7.3.3 ApiController 38

8 POHDINTA 39

9 JATKOKEHITYSIDEOITA 40

LÄHTEET 41

1 JOHDANTO

Työn toimeksiantajana aloitti InPulse Works Oy, jonka Solteq Oy osti 12.6.2017 3.5 miljoonalla eurolla. InPulse on vuonna 2010 perustettu, henkilöstön omistama sähköisen asionnin ja IT-ratkaisujen asiantuntija- ja ohjelmistoyritys. InPulsessa toimipisteiteet sijaitsivat Jyväskylässä, Kuopiossa ja Seinäjoella. Vuonna 2016 inPulsen liikevaihto oli noin 5 miljoonaa euroa. InPulsen liiketoiminta on keskittynyt kahdelle osa-alueelle. Yhtiö tarjoaa asiakaskohtaisia toteutuksia ja tuoteratkaisuja Energia-toimialalla, sekä BI- ja analytiikkaratkaisuja, jotka ovat toimialariippumattomia. InPulsen toteutukset pohjautuvat Microsoftin tuotteisiin ja palveluihin. (Solteq Oyj, 2017.)

Solteq on vuonna 1982 perustettu ohjelmistoyritys, jonka tuottaa asiakaskohtaamiseen tarkoitettuja liiketoiminnan teknologiasia ratkasuja, palveluita sekä liiketoiminnan tukea. Solteqin työllistää yli 500 asiantuntijaa Suomessa, Ruotsissa ja Puolassa. Solteqin markkina-alueisiin kuuluvat Eurooppa, Pohjois-Amerikka, Aasia ja Australia. Solteqilla on suomessa vahva markkina-asema johtavien kaupan alan toimijoiden keskuudessa. Solteqin lähivuoden tavoitteisiin kuulu strateginen kasvu eritoten Pohjoismaissa. (Solteq Oyj.)

Alkuperäisenä opinnäytetyön tarkoituksena oli kehittää inWorks-tuotteeseen kokonaisvaltainen laskutuksen automaatiotestaus, mutta laskutuksen monimuotoisuuden ja asiakaskohtaisten ratkaisujen määrän takia päädyttiin toteuttamaan ohjelmistokehys (Framework), jonka avulla voidaan toteuttaa testitapauksia, jotka vertailevat koodimuutosten välistä generoitua tai tuotua dataa. Työhön toteutettiin testitapaus inWorks-tuotteen laskutukseen käyttämällä tuotettua ohjelmistokehystä.

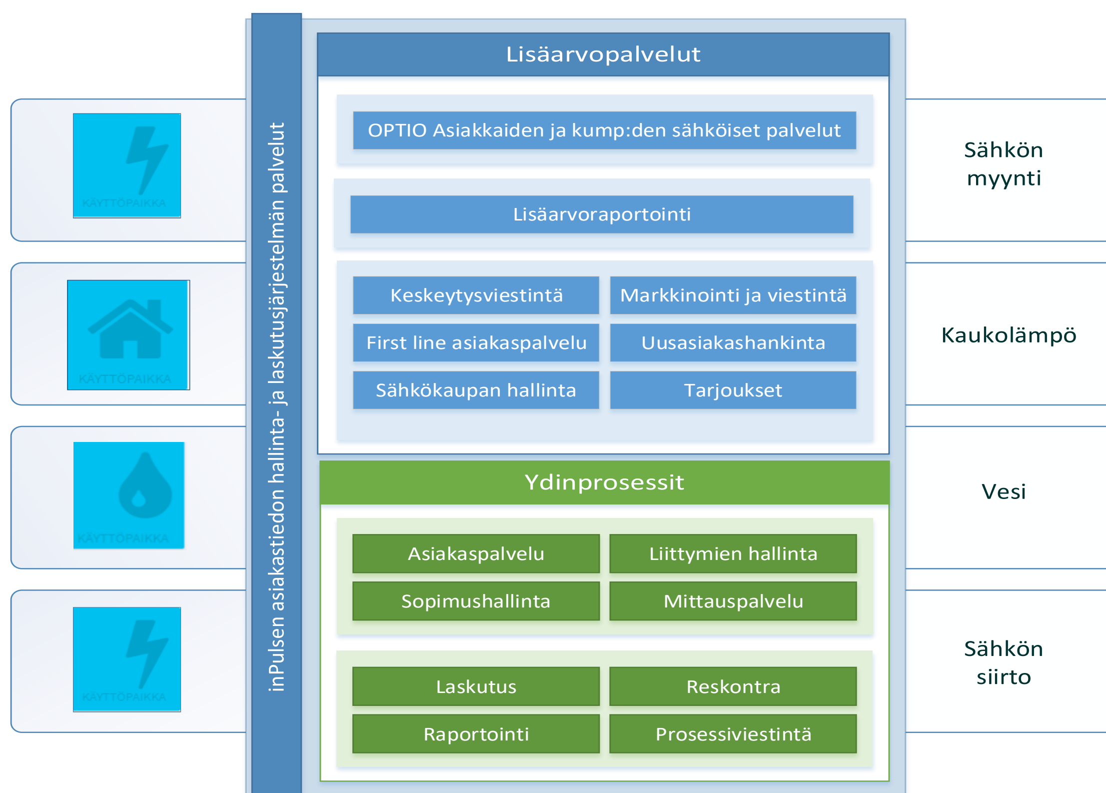
Työssä käsitellään lyhyesti inWorks-tuotetta, käytettyjä tekniikoita, käsitteitä sekä käydään läpi testauksen teoriaa lyhyesti. Testiohjelmistokehys osiossa käydään läpi kehyksen toteutus, laskutukseen toteutettu testitapaus sekä tuloksien tarkasteluun kehitetty käyttöliittymä.

2 INWORKS

KUVA 1. InWorksin etusivu.

InWorks on asikastietojärjestelmä, joka tarjoaa käyttäjälleen työkalut muunmuassa asiakkuushallintaan, raportointiin, laskutukseen, reskontraan ja kirjanpitoon. InWorks tuoteperheeseen kuuluu seuraavat jokaista hyödykettä vastaavat osakokonaisuudet:

- inWorks Water (vesi)
- inWorks Heat (kaukolämpö)
- inWorks Power (sähkö)
- inWorks Gas (maakaasu).



KUVA 2. InWorksin Ydinprosessit ja Lisäarvopalvelut kuvattuna (Solteq Oyj, 2017.)

2.1 InWorks laskutus

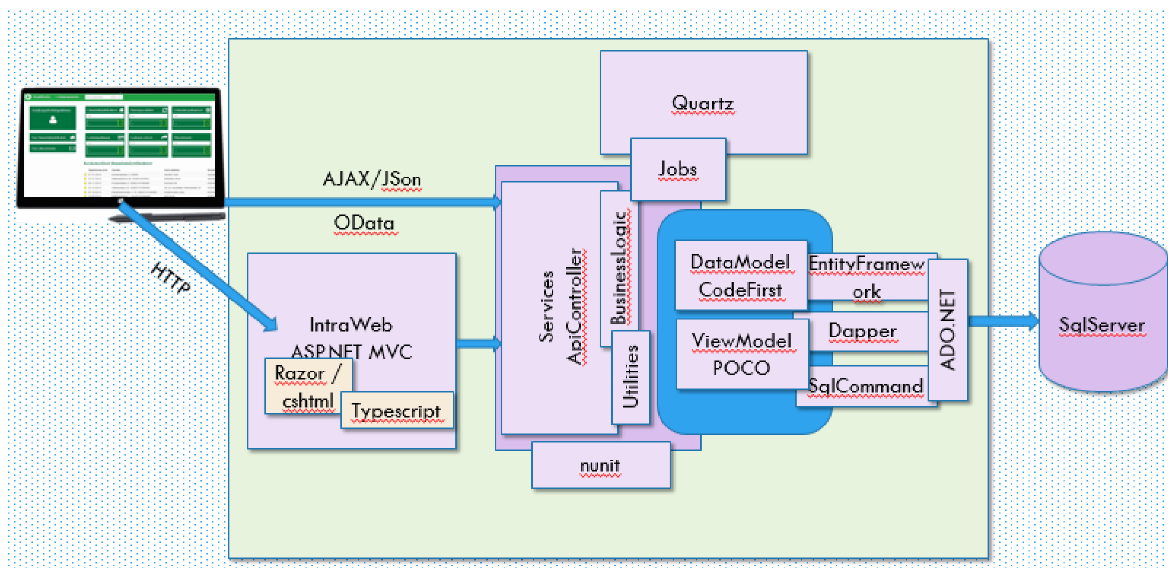
Laskutuksen tärkeimmät prosessit ovat laskutussyklin mukainen massalaskutus, yksittäinen laskutus sekä laskujen mitätöinti. InWorksin massalaskutus on automatisoitu prosessi, jossa kullakin laskutusjaksolla vuoroon osuville sopimuksille muodostetaan laskut. Tämän jälkeen laskuja peilataan inWorksin ylläpitotoiminnoissa määrittetyjä tarkistuksia vasten ja lopuksi laskut jaellaan eri kanavia pitkin, kuten e-laskuna, tulostusoperaattorin tai ulkopuolisen kumppanin kautta. Verkkolaskut ja sähköpostilaskut voidaan toimittaa myös suoraan järjestelmästä. Laskutusprosessia seurataan massalaskutuksen seurantakäyttöliittymän kautta. (Kulmala, 2017.)

KUVA 3. inWorksin massalaskutuksen seurantakäyttöliittymä.

Laskutus tukee monien hyödykkeiden laskuttamisen samalla laskulla, kuten esimerkiksi sähköllä myynin ja siirron tai esimerkiksi kaukolämmön ja veden laskut voi yhdistää yhdelle laskulle. Koontilaskutuksella voi laskuttaa esimerksi kaikki yhden asiakkaan kohteet yhdellä laskulla. Kaikki laskut ovat esikatseltavissa pdf-muodossa. (Kulmala, 2017.)

2.2 inWorks arkkitehtuuri lyhyesti

InWorksin suunnittelu ja toteutus pohjautuu Microsoftin tuotteisiin ja palveluihin. *IntraWeb* eli käyttöliittymä on toteutettu ASP.NET MVC suunnittelumallin mukaan. Näkymät on toteutettu Razor/cshtml kielillä ja asiakasohjelman (client) toiminnallisuus on hoidettu käyttämällä TypeScript ohjelmointikieltä. Näytettävä ja tallennettava data siirtyy asiakasohjelman ja palvelimen välillä JSon tai OData tyyppisenä käyttämällä Jqueryn AJAX(Asynchronous JavaScript and XML) kutsuja. *Servicet* sisältävät asiakasohjelman palvelinkutsuja vastaanottavat ja hoitavat Api- tai OdataControllerit. *BusinessLogic* projekti sisältää inWorksin sovelluslogiikka luokat ja toiminnallisuudet muunmuassa laskutukseen. *Utilities* projekti sisältää hyödykekohtaiset repositoryt ja toiminnallisuudet muunmuassa kulutuksien laskentaan ja validointiin. Asiakaskohtaiset tietokannat luodaan DataModelin pohjalta käyttämällä EntityFrameworkin CodeFirst lähestymistapaa. Kutsut tietokantaan hoidetaan EntityFrameworkin contextilla, Dapperillä tai suorilla sql komennoilla. Tietokanta on Microsoftin SQLServer.



KUVA 4. InWorksin arkkitehtuuri kuvattuna. (Hyvärinen, 2014.)

3 KÄYTETYT TEKNIIKAT, TYÖKALUT JA MÄÄRITELMÄT

Toteutus on pyritty hoitamaan käyttämällä samoja työkaluja, tekniikoita ja määrtelmiä kuin inWorks-tuotekin. Toteutus pohjautuu Microsoftin tuotteisiin tai ratkaisuihin kuten .NETiin, ASP.NET MVChen, EntityFrameworkiin ja SQLServeriin

3.1 .NET Framework

.NET Framework on Microsoftin kehittämä alusta Windows- ja Web sovelluksille, Web palveluille ja hajautetuille järjestelmille. Se sisältää perus luokka kirjaston BCL:n (Base Class Library), ohjelmien ajon mahdollistavan "virtuaalimoottori" CLR:n(Common Language Runtime). CLR tarjoaa monia palveluja kuten turvallisuuden, automaattisen muistinhallinnan, poikkeuskäsittelyn ja resurssien hallinnan. (Sinha, 2015.)

3.1.1 C#

C# on olioperustainen ohjelmointikieli, joka mahdollistaa ohjelmistokehittäjien rakentaa monipuolisia ja turvallisia sovelluksia, joita ajetaan .NET Frameworkissä. C# syntaksi yksinkertaistaa monia C++ monimutkaisuuksia ja tarjoaa monia tehokkaita ominaisuuksia kuten tyhjättävät arvo tyypit (nullable value types), enumeraatiot, delegaatit, lambda expressiot ja oikosiirron (direct memory access). (Microsoft, 2017.)

3.1.2 Rajapinta (Interface)

Rajapinta sisältää ainostaan metodien (Method), ominaisuuksien (Property), tapahtumien (Event) tai indeksien esittelyt. Rajapinnan implentoivan luokan tai structin on myös implementoitava kaikki rajapinnan kuvauksessa esitellyt jäsenet. (Microsoft, 2015.)

3.1.3 Perintä (Inheritance)

Yhdessä kapseloinnin (encapsulation) ja muunneltavuuden (polymorphism) kanssa, perintä on yksi olio-ohjelmoinnin pääpiirteistä. Perintä mahdollistaa uusien luokkien luomisen, jotka käyttävät, laajentavat ja muokkaavat perityn luokan toimintaa muissa luokissa. Luokkaa, jonka jäseniä peritään muihin luokkiin kutsutaan kantaluokaksi (base class) ja luokkia, jotka perivät kantaluokan jäseniä kutsutaan lapsiluokiksi (derived class). Lapsiluokalla voi olla vain yksi suora kantaluokka, mutta perittävyys on siirtyvää esimerkiksi, jos luokka C perii luokan B ja luokka B perii luokan A niin luokalla C on käytössä kaikki luokan B ja A esitellyt jäsenet. (Microsoft, 2015.)

3.1.4 ADO.NET

ADO.NET tarjoaa yhtenäisen sisäänpääsyn datalähteisiin kuten SQL Serveriin ja XMLään sekä datalähteisiin, jotka ovat näkyvissä OLE DB:n ja ODBC:n lävitse. Dataa jakavat kuluttaja sovellukset voivat käyttää ADO.NETiä näihin datalähteisiin yhdistämiseen sekä niiden sisältämän datan hakemiseen, käsittelyyn ja päivittämiseen. (Microsoft, 2017.)

3.2 EntityFramework

EntityFramework on nippu teknologioita ADO.NETissä, jotka tukevat oliopohjaisten sovellusten kehitystä. Se mahdollistaa kehittäjien työstää dataa paikallisina olioina ja ominaisuuksina ilman, että kehittäjien täytyisi huolehtia taustalla toimivista tietokannan tauluista ja sarakkeista. EntityFrameworkin avulla kehittäjät voivat luoda ja ylläpitää datapainotteisia sovelluksia vähemmällä koodimäärällä kuin perinteisempiä sovelluksia. (Microsoft, 2017.)

3.2.1 EntityFramework Code-First

EntityFramework Code-First lähestymistavalla kehittäjä voi keskittyä sovelluksen paikalliseen suunnitteluun ja aloittaa luomaan olioita paikallisten vaatimusten mukaan sen sijaan, että tietokanta pitäisi suunnitella ensin ja luoda oliot vastaamaan luotua tietokantaa. Code-First lähestymistavassa ensin luodaan tai muokataan toimiala kohtaiset luokat (domain class), mitkä seuraavassa kehitysvaiheessa konfiguroidaan käyttämällä Fluent-APIa tai datan kommentaari attribuutteja (data annotation attributes). Konfiguroinnin jälkeen tietokannan skeema luodaan tai päivitetään käyttämällä automaattista migraatiota tai koodiperustaista migraatiota. (EntityFrameworkTutorial.net, 2016.)

3.2.2 DbContext

EntityFramework mahdollistaa kehittäjän hakea, lisätä, päivittää ja poistaa dataa käyttämällä CLR:n objekteja, entiteettejä. EntityFramework kartoittaa entiteetit ja niiden suhteet, jotka ovat kuvattuna kehittäjän datamallissa. EntityFramework mahdollistaa kannasta palautuvan datan materialisoinnin entiteeteiksi, muutoksien seurannan ja hallinnan, entiteettien muutoksien heijastamisen takaisin tietokantaan ja entiteettien sitomisen käyttöliittymän käsittelijöille (Controller). (Microsoft, 2016.)

Pääluokka, joka vastaa entiteettien hallinnasta on DbContext. Context luokka hallitsee entiteettejä ajon aikana. Hallinta sisältää entiteettien populoinnin tietokannan datalla, muutoksien tarkkailun ja datan takaisin lähetyksen tietokantaan. (Microsoft, 2016.)

3.3 LINQ (Language Integrated Query)

LINQ on vahvasti tyypitetty kysely kieli, jonka voi upottaa suoraan C# ohjelmointikieleen. Sillä voidaan kirjoittaa tyyppivarmoja kyselyitä osana ohjelmointikieltä, jota .NET tukee. LINQ mahdollistaa datan helpon poimimisen erilaisista tietolähteistä kuten esimerkiksi XML tiedostoista, SQL kyselyistä

tai olioista. Se mahdollistaa myös kyselyt muistissa oleviin kokoelmiin, kuten esimerkiksi tauluihin ja listoihin. (Harwani, 2015.)

3.4 ASP.NET MVC

ASP.NET on ilmainen web-ohjelmistokehys web-sivujen- ja sovellusten kehittämiseen HTML, CSS/LESS ja JavaScript/TypeScript ohjelmointikielillä. Sillä voi myös luoda Web Ohjelmistorajapintoja(API) ja käyttää reaaliaika teknologioita kuten Web Socketeja. ASP.NET MVC on yksi ASP.NET ohjelmistokehyksistä ja siinä käytetään MVC (Model-View-Contoller) suunnittelumallia. (Microsoft.)

3.4.1 MVC

MVC erottelee sovelluksen kolmeen pää komponenttiryhmään: malli (Model), näkymä (View) ja käsittelijä (Controller). MVC:ssä käyttäjän komennot reititetään näkymältä käsittelijälle, joka työskentelee yhdessä mallin kanssa toteuttaakseen käyttäjän komentoja tai hakeakseen kyselyjen tuloksia. Käsittelijä valitsee, minkä näkymän käyttäjä näkee ja palauttaa minkä tahansa mallin datan, jonka näkymä vaatii. (Microsoft, 2016.)

3.4.2 Web API Controller

ASP.NET:in tapauksessa web API on tapa toteuttaa RESTful tyyppisiä web palveluja käyttämällä .NET frameworkiä. Web API kuvataan ohjelmistokehyksenä, joka mahdollistaa HTTP-palveluiden kehittämisen tavoittaa asiakasyhteisöt, kuten selaimet ja älylaitteet. Web API:a voidaan käyttää yhdessä MVC suunnittelumallin kanssa, missä tahansa sovelluksessa. (Brainvire Infotech Inc, 2016.)

3.5 TypeScript

TypeScript on Microsoftin kehittämä ja ylläpitämä ohjelmointikieli. TypeScript on JavaScript kieli, mihin on lisätty muuttujien tyypitykset. TypeScript kääntäjällä on kaksi pääominaisuutta: Se on transpiler ja tyyppitarkastelija (type checker). Transpiler on eräänlainen kääntäjä, joka tuottaa TypeScript koodista JavaScript koodia. Tyyppitarkastelija etsii ristiriitaisuuksia koodista. Esimerkiksi, jos muuttujan tyypiksi määrätään merkkijono ja sen jälkeen sitä yritetään käyttää numeron tavoin TypeScript ilmoittaa tyypitysvirheestä. (Wolff, 2016.)

3.6 Visual Studio

Visual Studio on Microsoftin kehittämä ohjelmankehitysympäristö (IDE). Visual Studiolla voi katsellä ja muokata melkein mitä tahansa ohjelmakoodia sekä debugata (debug), kasata (build) ja julkaista sovelluksia Androidille, iOSille, Windowsille, verkkoon ja pilveen. (Microsoft, 2018)

3.7 SQL Server

Microsoft SQL Server on tietokanta alusta, joka on kehitetty laajojen online transactioiden prosessointiin, datan tietovarastointiin ja verkkokauppa sovelluksiin. Se on myös alusta liiketoiminnallisten tietojen hallintaan (Business Intelligence) kuten data integraatioon, analytiikkaan ja raportointiin. (Microsoft, 2017.)

3.7.1 T-SQL

Transact-SQL (T-SQL) on Microsoftin oma versio ASNI SQL:stä Microsoftin Sql Serverille. Structured Query language (SQL) on eniten käytetty relaationaalisten tietokantojen kyselykieli ja sen strandisoitun version on suunnittellut American National Standards Institute (ANSI). (Techopedia.)

3.8 JQuery

Jquery on avoimen lähdekoodin nopea, kevyt ja kattava JavaScript kirjasto. Se on HTML dokumenttien läpikäyntiä, manipulointia, event käsittelyä ja Ajax kutsuja helpottava yksinkertainen ohjelmistorajapinta, joka toimii monilla eri selaimilla (The JQuery Foundation, 2017.)

3.9 NUnit

Nunit on yksikkötestaus ohjelmistokehys kaikille .NET ohjelmointikielille. Se on alunperin käännetty .NET ympäristöön JUnit ohjelmistokehyksestä, joka on tarkoitettu Java kielen yksikkötestaamiseen. (Poole;ym., 2017.)

3.10 NLog

Nlog on ilmainen lokitus alusta .NETille, .NET standardille, Xamarinille, Silverlightille ja Windows Phonelle. Nlog tarjoaa hyvät ja helposti konfiguroitavat työkalut lokien hallinnointiin ja reititykseen. NLog mahdollistaa sovelluksen lokien kirjoituksen tiedostoihin, konsoliin, tapahtumalokiin (event log), sähköpostiin, tietokantoihin tai suoraan ASP.NETin diagnostiikkaan (ASP.NET Tracing). (NLog.)

3.11 Azure

Microsoft Azure on laajeneva nippu pilvipalveluita, jotka auttavat organisaatioita vastaamaan liiketoiminnallisiin haasteisiin. Azure antaa vapauden rakentaa, hallita ja ottaa käyttöön sovelluksia massiivisessa maailmanlaajuisessa verkossa käyttämällä suosittuja työkaluja ja ohjelmistokehyksiä. (Microsoft, 2018.)

3.11.1 Blob storage

Azure blob storage on Microsoftin pilvessä toimiva objektien varastointi ratkaisu. Blob storage on optimoitu suurten epämuodollisten data massojen kuten tekstin tai binääri datan varastointiin. Blob storage on ideaali muunmuassa kuvien tai dokumenttien tarjoiluun suoraan selaimelle, tiedostojen varastointiin hallinnoiduilla oikeuksilla, videon ja audion streamaamiseen tai lokiin kirjoitukseen. (Microsoft, 2018.)

3.11.2 Azure Functions

Azure funktiot ovat ratkaisu, jolla voi helposti ajaa pieniä osia koodista tai funktioista pilvessä. Kehittäjä voi Azure funktioilla ajaa tietyn ohjelman osan ilman, että täytyy olla koko kehitysympäristö pystytettynä sen ajamiseksi. Ohjelmointikielet kuten C#, F#, Node.js, Java ja PHP ovat tuettuja Azure funktioissa. (Microsoft, 2017.)

3.12 Bootstrap

Twitter Bluebrint nimellä alkunsa saanut Bootstrap on avoimen lähdekoodin ohjelmistokehys websivustojen- ja sovellusten kehittämiseen HTML ja CSS kielillä. (Johanan;ym., 2016.)

3.13 Git

Git on versionhallinnointijärjestelmä. Git pitää kirjaa projektin historiasta sallien pääsyn mihin tahansa version menneisyydessä. Se sallii monien ihmisten työskentelyn samassa projektissa auttaen välttämään sekaannuksia, kun monet ihmiset yrittävät muokata samoja tiedostoja. Gitin loi myös linux käyttöjärjestelmän luonut Linus Torvalds ja Junio Hamano, joka on myös gitin pääkehittäjä. Gittiä kuvaillaan lähdekoodin hallintatyökaluna, mutta tosiasiassa se on vain erään tyyppinen versionhallinnointityökalu. (Daityari, 2015.)

4 OHJELMISTOTESTAUS

Testaaminen on prosessi, jossa arvioidaan systeemiä tai sen komponentteja tarkoituksellisesti, jotta saadaan selville vastaako se määriteltyjä vaatimuksia vai ei. Testauksessa järjestelmää ajetaan, jotta tunnistetaan aukkoja, virheitä tai puutteita alkuperäisistä vaatimuksista. (Tutorialspoint, 2018.)

4.1 Historia

1950-luvulla ohjelmistotestaus kuvattiin, mitä ohjelmoijat tekivät löytääkseen ja korjatakseen ohjelmistossa ilmenneitä vikoja. 1960-luvun alussa testauksen määritelmä sai päivityksen ja alettiin puhua tyhjentävästä testaamisesta (Exhaustive Testing) eli yritettiin löytää kaikki mahdolliset syötettävän tiedon variaatiot ja eliminoida sitä kautta viat. Tyhjentävä testaus hylättin, koska sen todettiin olevan teoriassa mahdotonta toteuttaa syötettävän tiedon variaatioden määrän, monimutkaisuuden tai liian monien mahdollisten syötevariaatioiden takia. (Lewis, 2017.)

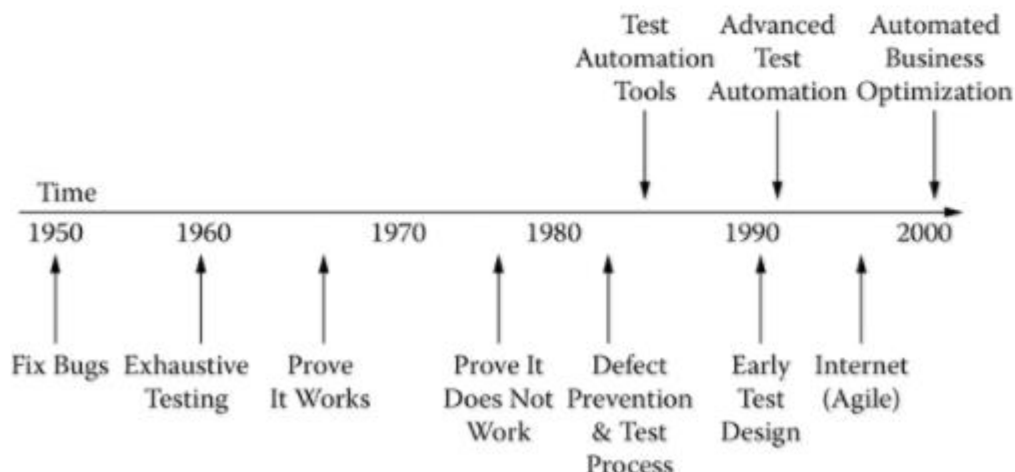
Ohjelmistokehitys aikuistui läpi 1960- ja 1970-lukujen ja ohjelmistokehityksessä alettiin käyttämään termiä tietokonetiede (Computer science) ja ohjelmistotestaus kuvattiin tapana demonstroida ohjelman toimivuus tai varmistaa, että järjestelmä tekee sen minkä se on suunniteltu tekemään. 1970-luvun lopussa todettiin, että testaus on prosessi, jossa ohjelmaa ajetaan siten, että pyritään löytämään virheet eikä todistamaan, että ohjelma toimii. Onnistunut testi oli se, joka löysi virheen. (Lewis, 2017.)

1980-luvulla testauksen määritelmää laajenettiin sisältämään vian ehkäisy ja ehdotettiin, että testausta täytyi katselmoida koko ohjelmiston kehityskaaren ajan sekä testauksen täytyi olla hallittu prosessi. Korotettiin myös, että testausta pitäisi lisätä myös vaatimuksiin, suunnitteluun, ohjelmakoodiin ja testeihin itseensä myös itse ohjelman lisäksi. (Lewis, 2017.)

1980-luvun puolenvälin jälkeen automaattiset testausvälineet ilmaantuivat automatisoimaan käsin tehtävän testauksen vaivaa, sekä kasvattamaan testauksen tehokkuutta sekä kohde sovelluksen laatua. Arvioitiin, että tietokone pystyy suorittamaan enemmän ja luotettavammin ohjelman testejä kuin ihminen, mutta tämän ajan automaatiotestaus työkalut olivat alkukantaisia eivätkä ne sisältäneet kehittyneitä ohjelmakoodin kielipalveluita (advanced scripting language facilities) kuin vasta 1990-luvun alkupuolella, jolloin automaatiotestaus alkoi yleistyä. 1990-luvun puolessa välissä internetin suosion kasvaessa ohjelmien testejä suoritettiin ilman tiettyä testausmallia tehden testauksesta paljon vaikeampaa. Tällöin syntyi ajatusmalli, että testejä voidaan suorittaa ilman, että kaikkea testattavaa määritellään etukäteen. Tälläistä testauksen lähetymistapaa alettiin kutsua ketteräksi testaukseksi (Agile testing). (Lewis, 2017.)

2000-luvun alussa Mercury Interactive esitteli vielä laajemman testauksen määritelmän, kun he julkaisivat BTOn (Business technology optimization) konseptin. BTO linjaa IT strategian ja suorituksen

liiketoiminnallisten tavoitteiden kanssa ja se auttaa prioriteettien, ihmisten ja IT prosessien hallinnoinnissa. (Lewis, 2017.)



KUVA 5. Ohjelmistotestauksen historia. (Lewis, 2017.)

4.2 Testauksen perusmääritelmät

Suurta osaa testauksen kirjallisuuden termistöstä pidetään sekavana johtuen todennäköisesti siitä, että teknologia on kehittynyt vuosikymmenten vaihtuessa sekä kirjoittajien vaihtuessa. ISTQB (The International Software Testing Qualification Board) pitää yllä laajaa termistöä testaukselle, jotka ovat IEEE (Institute of Electronics and Electrical Engineers) standardien mukaisia. Testauksen perusmääritelmiin kuuluu muunmuassa virhe, bugi, defekti ja epäonnistuminen (Jorgensen, 2016.)

4.2.1 Virhe (Error, Mistake)

Kehittäjän vahinkoa kutsutaan virheeksi. Virheet voivat johtua esimerkiksi väärin ymmärretystä ohjelmistovaatimuksesta, arvojen väärin laskemisesta tai arvon väärintulkinnasta. (Sharma, 2016.)

4.2.2 Bugi (Bug)

Bugi on koodi virheen tai vian tulos ohjelmassa, jonka takia ohjelma käyttäytyy tahattomasti väärin tai ennalta-arvaamattomasti. Bugit johtuvat kehittäjien virheistä ja ne esiintyvät ohjelman lähdekoodissa tai sen suunnitelmassa. Bugeja esiintyy kaikissa käytettävissä ohjelmissa, mutta hyvin kirjoitetut ohjelmat sisältävät suhteellisen vähän bugeja ja tyypillisesti ne eivät estä ohjelman toimintaa. (Sharma, 2016.)

4.2.3 Defekti (Defect)

Defekti on tila ohjelmistotuotteessa, joka ei vastaa ohjelmiston vaatimusmäärittelyä tai loppu käyttäjän odottamuksia. Toisinsanottuna defekti on virhe koodissa tai logiikassa, joka aiheuttaa ohjelman toimintahäiriön tai tuottaa vääriä tuloksia. (Sharma, 2016.)

4.2.4 Epäonnistuminen (Failure)

Epäonnistuminen on ohjemiston poikkeama sen tarkoitetusta päämäärästä. Se on systeemin tai komponentin kyvyttömyys suorittaa vaadittuja funktioita määriteltyjen suorituskyky vaatimusten mukaisesti. Epäonnistuminen tapahtuu, kun virhe suoritetaan. (Sharma, 2016.)

4.3 Testaustavat

Ohjelmistotestauksen toteuttamiseen on monia eri tapoja, joista tunnetuimmat ja käytetyimmät on kuvattu seuraavissa alikappaleissa.

4.3.1 Staattinen- ja dynaaminen testaus

Vahvistus ja validointi ovat kaksi käytettyä tapaa, kun tarkastellaan vastaako toteutettu ohjelma vaatimusmäärittelyä. Staattinen testaus sisältää vahvistukset, kun taas dynaaminen testaus sisältää validoinnin. Yhdessä ne auttavat parantamaan ohjelman laatua. (Guru99, 2018.)

Taulukko 1. Staattisen- ja dynaamisen testauksen eroja (Guru99, 2018.)

Staattinen testaus	**Dynaaminen testaus**
Testi hoidetaan ohjelmaa suorittamatta	Testi hoidetaan ohjelmaa ajamalla
Testaus hoitaa vahvistus prosessia	Testaus hoitaa validointi prosessia
Ennalta ehkäisee defektejä	Löytää ja korjaa defektejä
Antaa arvioita koodista ja dokumentaatiosta	Ilmiantaa bugeja ja pullonkauloja
Sisältää tarkistuslistan ja prosessin, jota noudatetaan	Sisälttää testitapaukset suorittamiseen
Voidaan suorittaa ennen kokoamista(Compilation)	Ei voida suorittaa ennen kokoamista
Defektien haun ja korjaamisen kustannukset pienempiä	Defektien haun ja korjaamisen kustannukset korkeampia
Vaatii paljon tapaamisia	Ei vaadi tapaamisia

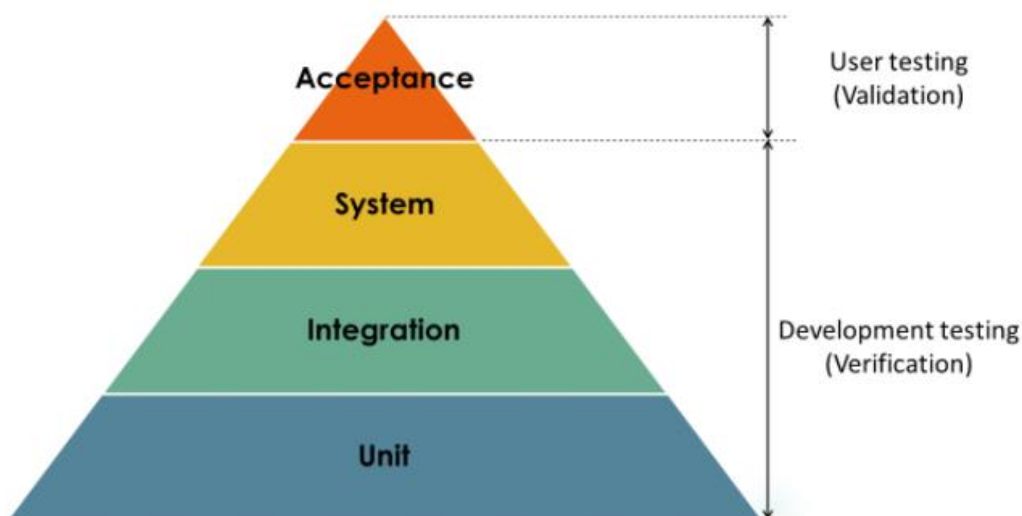
4.3.2 Musta- ja valkolaatikkotestaus

Mustalaatikkotestaus (Black-box testing) perustuu siihen, että testaajalla ei ole tietämystä sisäisestä ohjelmarakenteesta tai datasta. Mustalaatikkotestaus on riippuvainen järjestelmän tai komponenttien määrittelystä, jotta voidaan muodostaa oikeanlaiset testitapaukset. Mustalaatikkotestauksessa järjestelmä ajatellaan laatikkona, jonka toimintaa voidaan päätellä pelkästään tutkimalla syötettä ja siihen liittyvää tulostetta. Systemaattinen testaus (Systematic testing), satunnainen testaus (Random testing), graafisen käyttöliittymän testaus (Graphic User Interface testing), mallipohjainen testaus (Model-based testing), savutestaus (smoke testing) ja järkitestaus (Sanity testing) ovat yksiä parhaiten tunnettuja mustalaatikkotestauksen tekniikoita. (García, 2017.)

Valkolaatikkotestaus (White-box testing) perustuu testaajan ohjelmakoodin sisäisen logiikan tuntemukseen. Se tutkii, onko ohjelman rakenne ja logiikka virheellinen. Valkolaatikkotestitapaukset ovat tarkkoja vain, jos testaaja tietää, mitä ohjelman on tarkoitus tehdä. Valkolaatikkotestaus käyttää ohjelman lähdekoodia testitapausten perustana. Koodireportaasi(Code coverage), tahallisen virheen syöttäminen ja mutaatiotestaus ovat tärkeimpiä valkolaatikkotestauksen tapoja. (García, 2017.)

4.4 Testauksen tasot

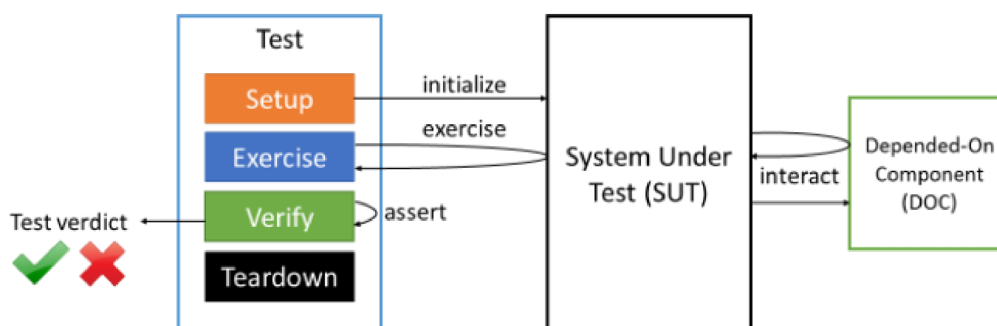
Riippuen testattavan järjestelmän koosta, testausta voidaan toteuttaa monilla eri tasoilla.



KUVA 6. Testauksen tasot ja niiden suuhteet validointiin ja vahvistukseen (García, 2017.)

Yksikkö- (Unit), integraatio (Intergarion)- ja järjestelmätestausta (System) suoritetaan tyypillisesti sovelluksen kehitysvaiheessa ja niitä suorittavat erilaisissa rooleissa toimivat ohjelmistoinsinöörit. Näiden testien tarkoitus on järjestelmän toimivuuden vahvistus. Hyväksymistestaus (Acceptance testing) on erään tyyppinen loppukäyttäjätestaus, jossa potentiaaliset tai oikeat käyttäjät ovat osallisena. (García, 2017.)

4.4.1 Yksikkötestaus



KUVA 7. Yksikkötestin neljä eri vaihetta. (García, 2017.)

Yksikkötestauksessa lähdekoodin yksittäisiä paloja testataan, jotta saadaan varmistus, että niiden suunnitelma ja toteutus kyseiselle yksikölle on oikeanlainen. Testitapaus suoritetaan neljässä eri vaiheessa:

- Asennus: Testitapaus asentaa ajettavan testin tavalla, jolla testattavan järjestelmänkin on oletettu toimivan.
- Ajo: Testitapaus vuorovaikuttaa testattavan järjestelmän kanssa ja saa järjestelmältä lopputuleman tuloksena. Testattava järjestelmä tekee usein hakuja toiselle komponenteille, joita kutsutaan riippuvaisiksi komponenteiksi (Depended-On Component).
- Varmistus: Testitapaus päättelee vastasiko tulos odotettua.
- Testitapaus purkaa testin ja asettaa testattavan järjestelmän alkuperäiseen tilaansa. (García, 2017.)

4.4.2 Integraatiotestaus

Integraatiotestauksen tarkoituksena on paljastaa defektejä rajapinnoissa ja integroitujen komponenttien tai moduulien välisessä vuorovaikutuksessa. On olemassa erilaisia strategioita integraatiotestauksen suorittamiseksi, joista tunnetuimpia ovat ylhäältä-alas integraatio (Top-down integration), pohjalta-ylös (Bottom-up integration), Ad hoc integraatio ja selkäranka integraatio (Backbone integration). (García, 2017.)

4.4.3 Järjestelmätestaus

Järjestelmätestaus varmistaa, että liitetyt komponentit ovat yhteensopia, kommunikoivat oikein ja siirtävät oikeaa dataa oikeaan aikaan tyypillisesti järjestelmän käyttäjärajapintojen yli. Vaikka järjestelmätestaus ja integraatiotestaus toimivatkin osittain samoilla periaatteilla, järjestelmätestaukseen pitäisi sisällyttää kaikki järjestelmän komponentit yhdessä loppukäyttäjän kanssa. On myös olemassa erikoistyyppinen järjestelmän testaus tapa, jota kutsutaan lopusta-loppuun(end-to-end) testaukseksi. Lopusta-loppuun lähestymistavassa järjestelmän loppukäyttäjää imitoidaan käyttämällä automaatiotestauksen tekniikoita. (García, 2017.)

4.5 Testaustyypit

Manuaalinen- ja automaattinen testaaminen ovat kaksi päätyyppiä, joilla sovellustestaus voidaan hoitaa. Nämä kaksi päätyyppiä avattu seuraavissa alikappaleissa.

4.5.1 Manuaalinen testaus

Manuaalinen testaus on prosessi, jossa testattavan järjestelmän arvioinnin hoitaa ihminen, tyypillisesti ohjelmistoinsinööri tai järjestelmän loppukäyttäjä. Manuaalisessa testauksessa ihmistestaajat arvioivat järjestelmää tutkimalla ja käyttämällä omaa havainnointikykyään. (García, 2017.)

4.5.2 Automaatiotestaus

Automaatiotestauksen prosessissa testattavan järjestelmän kussakin testiprosessissa hoitaa siihen tarkoitettu erikoisohjelmisto tai testaukseen tarkoittettu infastruktuuri. Järjestelmän automaattisen testaamisen päähyödyt ovat; säästöt kustannuksissa, lyhemmät testien kestoajat, testien perinpohjaisuus, tarkkuus sekä tuloksien raportointi. Automaatiotestaus on tehokkain, kun se on implementoitu rajapinnan sisälle. Testausrajapintoja voidaan kuvata kokoelmaksi abstrakteja konsepteja, prosesseja, proseduureja ja ympäristöjä, joissa automaatiotestejä tullaan suunnittelemaan, luomaan ja toteuttamaan. (García, 2017.)

4.6 Käyttäjätestaus

Käyttäjätestaus on vaihe testausprosessissa, jossa käyttäjät tai asiakkaat antavat palautetta ja neuvoja järjestelmän testaukseen. Hyväksymistestaus on käyttäjätestaamisen eräs tapa, mutta on olemassa myös erilaisia käyttäjätestaustyyppejä

- Alpha testaus: Tapahtuu kehittäjien sijainnissa yhdessä sovelluksen kuluttajien kanssa, ennen kuin sovellus julkaistaan ulkopuolisille käyttäjille tai asiakkaille.
- Beta testaus: Sisältää testausta asiakasryhmän kanssa, jotka käyttävät järjestelmää omissa sijainneissaan ja antavat palautetta ennen kuin järjestelmä julkaistaan lopuille asiakkaille.
- Operatiivinen testaus: Tapahtuu loppukäyttäjän toimesta lopullisessa järjestelmässä. (García, 2017.)

5 TESTIOHJELMISTOKEHYS

Testiohjelmistokehys on toteutettu koodimuutosten väliseen generoitavan tai tuotavan datan tarkasteluun. Perusperiaatteeltaan testit toimivat siten, että vanha data kerätään inWorksin tietokannasta talteen, jonka jälkeen data generoidaan kulloisenkin testattavan tapauksen mukaan. Esimerkiksi testattavana kohteena ollessa laskutus, vanhat laskut kerätään talteen, laskun perustiedot sisältävät kantaobjektit asetetaan sellaiseen tilaan, että ne voidaan generoida inWorksin työkaluilla uudestaan. Vanhojen alkuperäisten laskun muodostavien objektien tila kerätään talteen, jotta tilanne voidaan palauttaa alkuperäiseksi testiajon jälkeen.

Tämän prosessin jälkeen testattavat objektit generoidaan tai tuodaan inWorksin metodeilla uudestaan ja uusia muodostuneita objekteja esimerkiksi laskuja, peilataan vanhoja talteen otettuja objekteita vasten ja eroavaisuudet kerätään ohjelmistokehyksen omaan tietokantaan. Tulokset sisältävät aina testauksen tilan eli onko se epäonnistunut, onnistunut vai keskeneräinen eli onko testi kaatunut kesken ajon. Epäonnistuneista testeistä kerätään talteen alkuperäisistä poikkeavat objektit ja niiden alkuperäiset, sekä uudet generoituneet arvot. Ohjelmistokehyksen päälle on myös kehitetty käyttöliittymä, josta käyttäjän tai kehittäjän on helpompi tarkastella kunkin testiajon tuloksia. Tämän tyyppisen testauksen huono puoli on siinä, että se ei osaa tunnistaa vielä olemassa olevia virheitä eli tässä tapauksessa testi on yhtä hyvä kuin datan alkuperäinen tila ja korjausten oikeellisuus.

5.1 Rajapinta

Ohjelmistokehyksen rajapinnat ovat hyvin yksinkertaisia ja pakottavat kehittäjän toteuttamaan perivälle luokalle pakolliset ominaisuudet(property) ja metodit.

```
public interface ITestCase
{
    string Table { get; }
    string TempTableName { get; }
    void Execute();
}

public interface ITestObject
{
    string Identifier();
}
```
Ohjelmakoodiote 1. Ohjelmistokehyksen rajapinnat

ITestCase-rajapinta pakottaa käyttäjän tai kehittäjän implementoimaan seuraavat ominaisuudet omaan testitapaukseensa:

Taulukko 2. *ITestCase* rajapinnan ominaisuuksien selitteet

Ominaisuus	Selite
string Table { get; }	Tietokannan kohdetaulu, johon testaus suoritetaan.
string TempTableName {get;}	Taulun nimi, joka luodaan testitapausta ajettaessa ja sinne varastoidaan alkuperäinen data, jota vasten uutta luotua dataa peilataan.
void Execute();	Metodi, joka aloittaa testin suorituksen ohjelmoidussa järjestyksessä.

ITestObject-rajapinta pakottaa toteuttamaan testattavan luokan toteutukseen *Identifier* metodin, jolla kasataan uniikki avain, mitä vasten testiobjekteja verrataan keskenään.

5.2 Kantaluokka

Kantaluokka *TestCaseBase* toteuttaa aiemmin kuvatun *ITestCase* rajapinnan ja sisältää toteutuksen *ITestCase* rajapinnan määrityksille. Testiajon aloittavat metodit on esitelty kantaluokassa. Testejä ei voi suorittaa kantaluokalla vaan tapauskohtainen testi täytyy vielä toteuttaa kantaluokan perivässä lapsiluokassa, jossa toteutetaan testikohtainen suoritusjärjestys sekä alustetaan testikohtaiset muuttujat, kuten esimerkiksi testin kohdetaulu sekä väliaikainen taulu alkuperäisen datan keräämiseen.

```
public virtual string Table { get; }
public string TempTableName { get; }
public void Execute()
        {
            var watch = Stopwatch.StartNew();
            try
            {
                Logger.Debug(string.Format("TestFrameworkRun started : {0}", DateTime.UtcNow));
                ExecuteTest();
                Logger.Debug(string.Format("TestRun {0} finished. Execution time : {1} minutes",  _testRun.Test-
CaseName, watch.Elapsed.TotalMinutes));
                watch.Stop();
            }
            catch (Exception ex)
            {
                Logger.Error(string.Format("TestRun {0} could not finish. Execution time : {1} minutes",
_testRun.TestCaseName, watch.Elapsed.TotalMinutes));
                Logger.Error(ex.Message, ex);
                watch.Stop();
                _testRun.TestResult = TestResultEnum.NotComplete;
                throw;
            }
            _testRepository.SaveChanges();
        }
```

Ohjelmakoodiote 2. *ITestCase* rajapinnan implementointi *TestCaseBase* luokassa.

Table ja *TempTableName* on esitelty kantaluokassa sellaisenaan kuin ne on määritelty rajapinnassakin. Lisäksi *TempTableName* muuttuja on alustettu kantaluokassa "tempTable" merkkijonoksi (string). *Execute* metodin muuttuja *watch* on instanssi *StopWatch* luokasta, joka kuuluu .NET:in System.Diagnostics nimiavaruuteen (Namespace). Execute metodi sisältää myös instanssin *NLog* lokitusluokasta. Lokiin kirjataan testin aloitus- ja lopetusaika, sekä epäonnistumiset, jos testiajo kaatuu ennen kuin testiä ennätetään suorittaa loppuun.

5.2.1 Testiajon alustus

Testiajo alustetaan *TestCaseBase* luokan konstruktorissa (constructor), jossa ajolle asetetaan perustiedot kuten nimi, yksillöllinen *Guid* tyyppinen *BatchId* tunniste ja ajon aloituksen aikaleima. Testiajon perustiedot tallennetaan tietokantaan ennen varsinaisen testin ajamista.

5.2.2 Alkuperäisen datan talteenotto

```
private void CopyOriginalValues()
        {
            var repo = new ATJRepository();
            using (var transaction = repo.Database().BeginTransaction(IsolationLevel.ReadCommit-
ted))
            {
                try
                {
                    var cmd = $@"IF OBJECT_ID('{TempTableName}', 'U') IS NOT NULL DROP TABLE
{TempTableName}
                        SELECT * INTO {TempTableName} FROM {Table} {Condition}";
                    repo.GetContext().Database.ExecuteSqlCommand(cmd);
                    transaction.Commit();
                }
                catch
                {
                    transaction.Rollback();
                }
            }
        }
```

Ohjelmakoodiote 3. *CopyOriginalValues* metodi.

Testiajon aikana testin kohteena olevan tietokannan taulun tiedot otetaan talteen *CopyOriginalValues* metodissa. Tiedot kopioidaan talteen testin kohdetaulusta *Table* ja tallennetaan *TempTableName* muuttujassa määritettyyn tauluun. Condition muuttuja sisältää tiedon kopioinnissa määrätyistä ehdotista, joiden perusteella tieto kerätään kohde taulusta. Metodi käyttää T-SQL:n SELECT INTO lausetta ja komento ajetaan tietokantaan käyttämällä inWorksin ATJRepository luokkaa. Komento ajetaan transactionin sisällä, jotta taulua ei luoda turhaan, jos kirjoitetussa T-SQL komennossa onkin vikaa.

5.2.3 Testiajon aloitus

Testiajo on toteutettu siten, että sitä voidaan ajaa automaattisesti ajastettuna työnä (scheduled job) tai yksikkötestinä käyttäen Visual Studion C# kielelle toteutettua ulkopuolista testadapter liitännäistä NUnitia.

```
[TestFixture]
    public class UnitTests
    {
        [Test]
        public void RunTest()
        {
            var testCase = new InvoicingTestCase(UtilityTypeEnum.Water, 25, inWorksCreateIn-
voicesTest");
            testCase.Execute();
        }
    }
```

Ohjelmakoodiote 4. Esimerkki testiajon aloituksesta käyttäen *NUnit* test adapteria

RunTest metodissa ajetaan inWorksin uutta laskutustapaus testiä. *TestCaseBasen* perivä lapsiluokka *InvoicingTestCase* alustetaan parametrisoidussa kontstruktorissa testattavalla hyödykkeellä, testattavalla laskutusjaksolla ja testin nimellä.

5.2.4 Testidatan vertailu

Testidatan vertailu toteutetaan samassa metodissa, missä data ladataan ohjelmistokehyksen omaan tietokantaan. Vertailu ottaa parametrina kaksi eri listaa, joiden arvoja vertaillaan keskenään ja tehdään päätelmä, onko tulos oikea vai väärä. Tuloksen ollessa poikkeava inWorks tietokannan alkuperäisestä arvosta se ladataan *inWorksTestFramework* tietokantaan.

```
protected void LoadResultsToTestCaseDB<T>(List<T> newRows, List<T> oldRows) where T : ITestObject
        {
            if (newRows == null || oldRows == null)
                Logger.Error("new values or old values are null");
                throw new NotImplementedException();
            using (var transaction = _testRepository.GetContext().Database.BeginTransaction(Isolation-
Level.ReadCommitted))
            {
                int faultyRows = 0;
                CheckForNewlyCreatedRows(_testRun, oldRows, newRows);
                foreach (var oldRow in oldRows)
                {
                    var oldIdentifier = oldRow.Identifier();
                    var newRow = newRows.Where(nr => nr.Identifier() == oldIdentifier).SingleOrDefault();
                    var testObject = new TestObject
                    {
                        TestRun = _testRun,
                        NewValueNull = false,
                        OldValueNull = false
                    };
                    if (newRow == null)
                    {
                        testObject.NewValueNull = true;
                        testObject.Identifier = oldRow.Identifier();
                    }
                    else if (oldRow == null)
                    {
                        testObject.OldValueNull = true;
                        testObject.Identifier = newRow.Identifier();
                    }
                    else
                    {
                        testObject.PopulateTestObject(newRow, oldRow, 0.1m);
                    }
```

Ohjelmakoodiote 5. Testiobjektin alustus ja rivikohtaiset null-vertailut *LoadResultsToTestCaseDB* metodissa.

Ajon tullessa *LoadResultsToTestCaseDB* metodiin, tarkastetaan ettei kumpikaan tuotavista listoista ole tyhjiä. Tietoja tarkastellaan alkuperäisten datarivien näkökulmasta ja parametrina tuotavat listat

tulee olla dataltaan sellaisia, että samalla tunnisteella löytyy molemmista listoista datarivit. Kumman tahansa arvon puuttuessa tulos kirjataan kantaan virheellisenä *TestObject* tyyppiselle luokalle, joka kuvastaa testattavan taulun datariviä. Molempien datarivien ollessa mukana siirrytään tarkastelemaan tuloksia soluarvokohtaisesti *PopulateTestObject* metodiin.

5.2.5 Solukohtaisten datojen vertailu

Solukohtaisten arvojen vertailut hoidetaan *TestFrameworkUtils* luokan *PopulateTestObject* metodissa. Vertailu ottaa parametrina vanhan ja uuden rivin, jotka ovat *ITestObject* rajapinnan toteuttavia luokkia. Lisäksi parametrinä voidaan tuoda numeraalisille muuttujille tarkoitettu *decTolerance* muuttuja, jolla voidaan määrittää testattavalle arvolle toleranssi, minkä sisälle sen pitää osua, jotta vertailun tulos hyväksytään. *PopulateTestObject* metodi hakee testattavan luokan ominaisuudet .NET:in *System.Reflection* nimiavaaruden *PropertyInfo* luokan *GetProperties* metodilla. Luokan ominaisuuksista otetaan talteen ominaisuuden nimi ja sen sisältämä arvo. Ominaisuuden nimen arvo on tallennettu *PropertyInfo* luokassa *Name* merkkijonoon ja arvo haetaan käyttämällä *PropertyInfo.GetValue* metodia*.* Mikäli rivikohtainen kenttäarvo ei täsmää uudella ja alkuperäisellä rivillä, tarkastetaan, onko se määritetyn toleranssin sisällä. Uuden arvon ollessa erisuuri ja toleranssin ulkopuolella, alkuperäinen ja uusi solukohtainen arvo lisätään *TestObjectField* tyyppisenä rivikohtaiselle arvolle.

```
public static void PopulateTestObject<T>(this TestObject testObject, T newValues, T oldValues, decimal? dec-
Tolerance = null) where T : ITestObject
        {

            PropertyInfo[] fi = typeof(T).GetProperties();
            foreach (PropertyInfo f in fi)
            {
                var propName = f.Name;
                object valA = null;
                object valB = null;
                if (newValues != null)
                    valA = f.GetValue(newValues);
                if (oldValues != null)
                    valB = f.GetValue(oldValues);
                if (valA != null && valB != null && valA.Equals(valB))
                    continue;

                if (valA == null && valB == null)
                    continue;

                if (decTolerance != null)
                {
                    try
                    {
                        if (IsInsideTolerance(Convert.ToDecimal(valA), Convert.ToDecimal(valB), decToler-
ance.Value))
                            continue;
                    }
                    catch(Exception ignore)
                    {
                        //Decimal convert failed
                    }
                }

                var testObjectFieldNew = new TestObjectField
                {
                    TestObject = testObject,
                    FieldName = propName,
                    Value = valA?.ToString(),
                    TestObjectFieldType = TestobjectFieldTypeEnum.NewValue
                };
                var testObjectFieldOld = new TestObjectField
                {
                    TestObject = testObject,
                    FieldName = propName,
                    Value = valB?.ToString(),
                    TestObjectFieldType = TestobjectFieldTypeEnum.OriginalValue
                };
                if (testObject.ObjectFields == null)
                    testObject.ObjectFields = new List<TestObjectField>();
                testObject.ObjectFields.Add(testObjectFieldNew);
                testObject.ObjectFields.Add(testObjectFieldOld);
                if (string.IsNullOrWhiteSpace(testObject.Identifier))
                    testObject.Identifier = oldValues.Identifier();
            }
        }
```

Ohjelmakoodiote 6. *PopulateTestObject* metodi

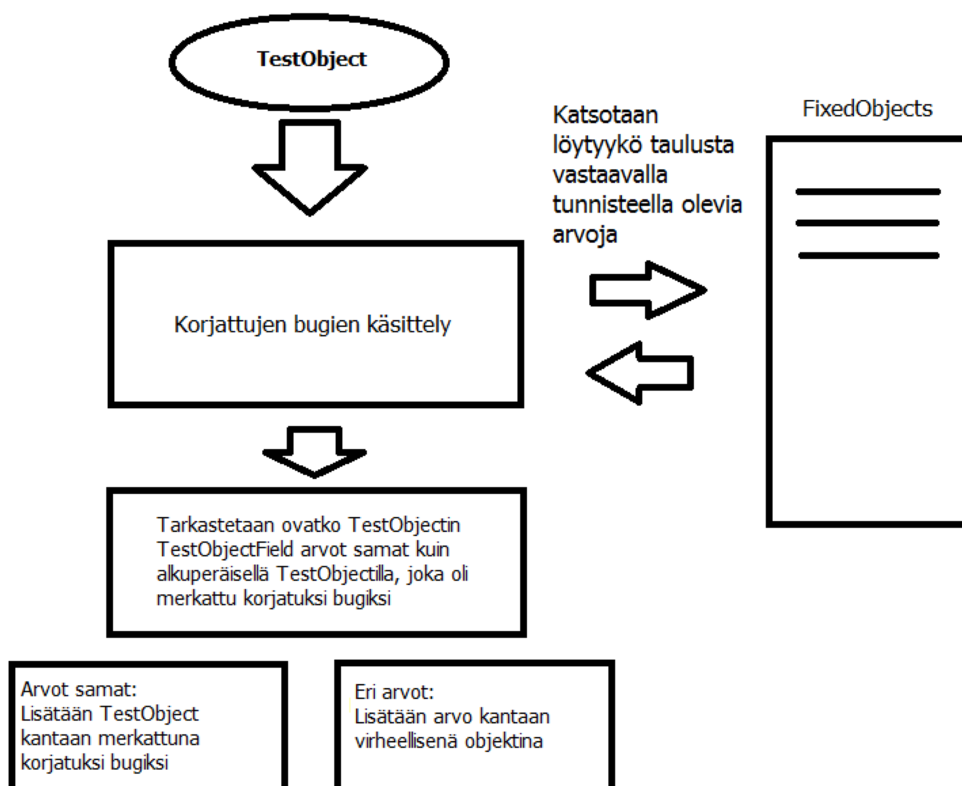
5.2.6 Korjattujen bugien käsittely

Testatessa ja tuloksia tutkiessa voi tulla vastaan tilanne, että uudempi versio ohjelmakoodista on korjannut jonkin vian ja silloin alkuperäinen data on virheellinen. Ohjelmistokehyksessä on tämänkaltaisille tilanteille rakennettu korjattujen bugien käsittely. Virheellisiä tuloksia voidaan kumota lisäämällä Ohjelmistokehyksen tietokannan *FixedObjects* tauluun rivejä. Rivit sisältävät saman tunnisteen(*Identifier*), jolla alkuperäisiä rivejä on vertailtu, sekä tiedon Testiajosta(*TestRun*) ja rivistä(*TestObject*), millä se on merkattu korjatuksi.

```
if (testObject.ObjectFields != null && testObject.ObjectFields.Any())
                    {
                        var fixedObjects = _testRepository.Find<FixedObject>(r => r.Identifier == tes-
tObject.Identifier).ToList();
                        if (fixedObjects != null && fixedObjects.Any())
                        {
                            foreach (var item in fixedObjects)
                            {
                                var values1 = _testRepository.Find<TestObjectField>(r => r.TestObjectId ==
item.OriginalTestObjectId && r.TestObjectFieldType == TestobjectFieldTypeEnum.NewValue)
                                    .Select(r => new { r.FieldName, r.Value })
                                    .ToList();
                                var values2 = testObject.ObjectFields.Where(r => r.TestObjectFieldType == Tes-
tobjectFieldTypeEnum.NewValue).Select(r => new { r.FieldName, r.Value }).ToList();
                                if (values1.Except(values2).ToList().Any() && values2.Except(val-
ues1).ToList().Any())
                                {
                                    testObject.IsFixedBug = false;
                                    _testRepository.InsertWithoutSaving(testObject);
                                    faultyRows++;
                                }
                                else
                                {
                                    testObject.IsFixedBug = true;
                                    _testRepository.InsertWithoutSaving(testObject);
                                }
                            }
                        }
                        else
                        {
                            testObject.IsFixedBug = false;
                            _testRepository.Insert(testObject);
                            faultyRows++;
                        }
                    }
```

Ohjelmakoodiote 7.Korjattujen bugien käsittely

Testiajossa jo virheelliseksi todettuja rivikohtaisia arvoja tarkastetaan *FixedObjects* taulun tunnisteita vasten ja tunnisteiden vastatessa toisiaan, tietokannasta tarkastetaan vielä uudelta rivikohtaiselta (*TestObject*) arvolta sen rivin solukohtaiset arvot (*TestObjectFields*), joita verrataan vielä alkuperäiseen tapaukseen, jolla rivikohtainen arvo on alunperin korjatuksi bugiksi.



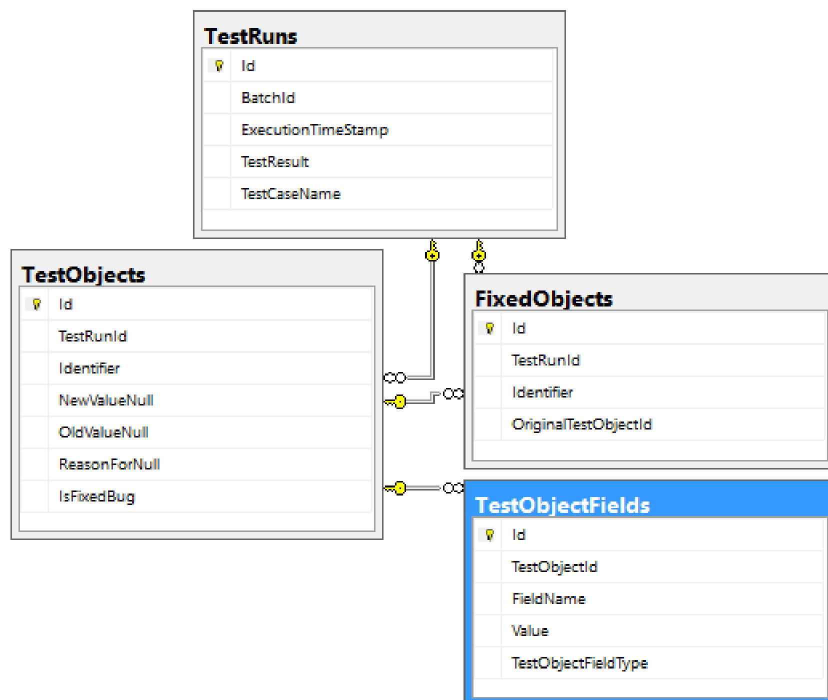
KUVA 8. Korjattujen bugien käsittely kuvattuna

5.3 Tietokanta

Ohjelmistokehyksen tietokanta on malliltaan yksinkertainen. *TestRuns* taulu sisältää perustiedot testiajosta kuten ajon alkuajan, tuloksen ja sen nimen. *TestObjects* taulu sisältää testattavien kohdetaulujen virheelliset riviarvot. Ne sisältävät viittauksen testiajoon, jossa se on luotu sekä lisäksi yksillöllisen merkkijono tunnisteen, joka luodaan testattavan objektin luokassa *ITestObject* rajapinnan määritysten mukaisesti *Identifier* metodissa. Lisäksi *TestObjects* taulu sisältää tiedon uuden tai alkuperäisen rivin puuttumisesta, puuttumisen syystä sekä tiedon siitä onko rivi korjattu bugi.

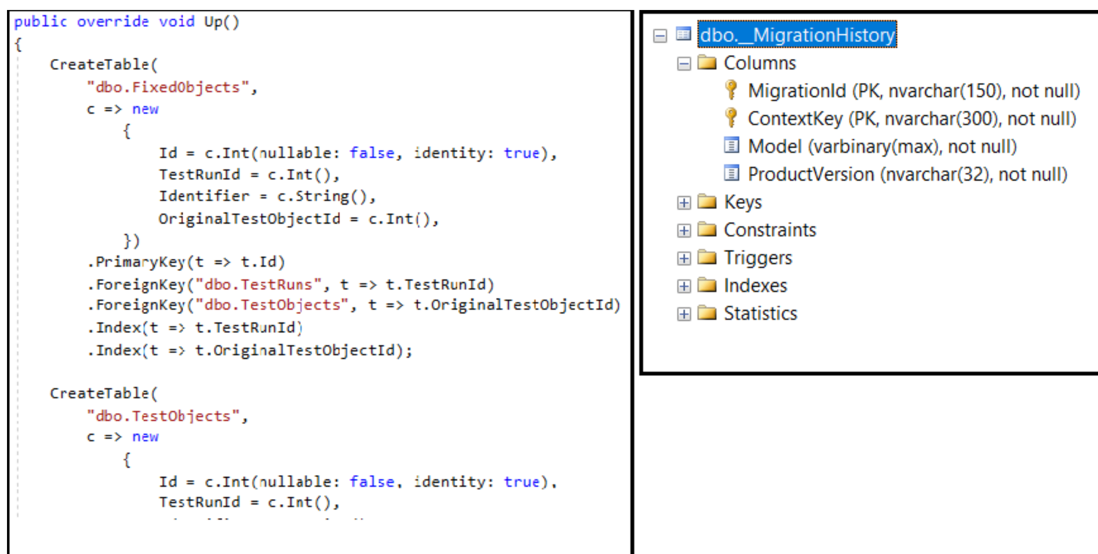
TestObjectFields sisältää *TestObject* rivin toisistaan eroavat solukohtaiset arvot, jotka on tarkistettu ja lisätty *PopulateTestObject* metodissa. *TestObjectFields* taulu sisältää tiedon kentän nimestä, arvosta ja siitä onko se uuden vai alkuperäisen datan arvo.

FixedObjects taulussa on eritelty korjatut bugit tunnisteella ja se sisältää viittaukset omaan alkuperäiseen testiajoonsa, jolla se on korjattu sekä alkuperäiseen rivikohtaiseen arvoon, jotta voidaan tarkastaa sen solukohtaiset arvot ja todeta onko se sama bugi, joka on merkattu korjatuksi.



KUVA 9. Ohjelmistokehyksen tietokanta *inWorksDb_TestFramework* kuvattuna

Tietokanta luotu käyttämällä EntityFrameworkin Code-First lähestymistapaa, eli taulut ja niiden väliset viittaukset sekä jaetut avaimet on ensin tehty datamodeliin luokkina, jonka jälkeen on luotu EntityFramework migraatio Visual Studion Package Manager konsolissa käyttämällä *Add-Migration* komentoa. Jokainen migraatio sisältää silloiset datamodeliin tehdyt muutokset. Migraatiot sisältävät tietokannan skeeman luonti- ja muutoslauseet sekä konfiguroinnit kuten relaatiot taulujen välillä. Tietokanta luodaan ja päivitetään käyttämällä *Update-Database* komentoa Package-Manager konsolissa. Update-database komento ajaa kaikkien migraatioiden *Up* metodin, joita ei löydy jo tietokannan *_MigrationHistory* taulusta. Migraatiossa on myös *Down* metodi, mikäli tietokannan skeema halutaan palauttaa aiemman migraation tilaan *–targetMigration* parametrillä.



KUVA 10. Esimerkki migraation *Up* metodista ja _MigrationHistory taulun rakenteesta

5.3.1 EntityFramework datamodel

Tietokanta luodaan ja sitä muokataan EntityFramework datamodelin muutosten perusteella. Datamodel sisältää luotavan tietokannan taulut luokkina ja kentät luokan ominaisuuksina. Ominaisuuksien tyypit määrittävät, mikä tietokannan kentän tyyppi tulee olemaan, kun tietokanta luodaan mallin pohjalta. Esimerkiksi string tyyppinen ominaisuus kääntyy tietokantaan varchar tyyppisenä kenttänä. Tietokannan relaatiot ja lisämääritykset on lisätty attribuutteina. Datamodelin luokat voivat pitää sisällään myös muita datamodelin objekteja tai objektikokoelmia olettaen, että taulujen väliset relaatiot ovat kunnossa.

```
public class TestObject
    {
        public int Id { get; set; }
        public int? TestRunId { get; set; }
        [ForeignKey("TestRunId")]
        public virtual TestRun TestRun { get; set; }
        public string Identifier { get; set; }
        public bool NewValueNull { get; set; }
        public bool OldValueNull { get; set; }
        public string ReasonForNull  { get; set; }
        public ICollection<TestObjectField> ObjectFields { get; set; }
        public bool IsFixedBug { get; set; }
    }
```

Ohjelmakoodiote 8. Esimerkki datamodelin luokasta *TestObject*. Sisältää viittauksen *TestRun* luokkaan sekä *TestObjectField* kokoelman, jolla on viittaus tähän objektiin.

5.3.2 TestFrameWorkDbContext luokka

DbContext luokka sisältää DbSet tyyppiset kokoelmat datamodelin objekteista. Context pitää kirjaa objekteiden tilasta ja aina kun objekteille tehdään toimintoja kuten päivitetään ominaisuuksien arvoa, lisätään uusia objekteja tai poistetaan vanhoja objekteja, päivittyvät ne tietokantaan *SaveChanges* metodia ajettaessa.

```
public class TestFrameWorkDbContext : DbContext
    {
        public TestFrameWorkDbContext()
            : base("TestFrameWorkDatabase")
        { }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            //modelBuilder.HasDefaultSchema("Identity");
        }

        public static TestFrameWorkDbContext Create()
        {
            return new TestFrameWorkDbContext();
        }

        public DbSet<TestRun> TestRuns { get; set; }
        public DbSet<TestObject> TestObjects { get; set; }
        public DbSet<TestObjectField> TestObjectFields { get; set; }
        public DbSet<FixedObject> FixedObjects { get; set; }
    }
```

Ohjelmakoodiote 9. *TestFrameWorkDbContext* luokka.

5.3.3 TestFrameWorkRepository

TestFrameWorkReporsitory on luokka, joka perii inWorksin GenericRepository luokan ja se sisältää metodeita *TestFrameWorkDbContext* luokan helpompaan ja tarkempaan tapauskohtaiseen käsittelyyn.

6 OHJELMISTOKEHYKSEN SOVELLUTUS INWORKSIN LASKUTUKSEEN

Ohjelmistokehyksestä toteutettiin sovellutus inWorksin laskujen muodostukseen. InWorks muodostaa laskut valituille hyödykkeille laskutuksen perustietojen koontitaulun perusteella. Laskujen muodostuksen testiajot on toteutettu vertailemaan määritetyllä laskutuskaudella tehtyjen laskujen ja samalla laskutuskaudella uudelleen generoitujen laskujen eroavaisuuksia valittavien osa-alueiden esimerkiksi laskun loppusumman, veromäärän ja kulutuksien osalta.

6.1 *ITestObject* rajapinnan toteutus laskun generointi testitapauksessa

Laskun muodostus testissä *ITestObject* rajapinnan toteuttavaan *InvoiceTestObject* luokkaan on kerätty kaikki yksilöivään tunnisteeseen tarvittavat ominaisuudet sekä ominaisuudet, joita ajettavassa testissä halutaan verrata solukohtaisesti. *Identifier* metodi palauttaa *InvoiceTestObject* tyyppisen luokan yksilöllisen tunnisteen.

```
public class InvoiceTestObject : ITestObject
    {
        public int ContractId { get; set; }
        public int MeteringPointId { get; set; }
        public int Payer_Id { get; set; }
        public int InvoicingPeriodId { get; set; }
        public int InvoicingCycleLength { get; set; }
        public string Payer_Code { get; set; }
        public string CustomerDetails_Code { get; set; }
        public string MeteringPointCode { get; set; }
        public decimal Total { get; set; }
        public decimal TotalWithoutVAT { get; set; }
        public decimal Vat { get; set; }

        public string Identifier() { return $"{ContractId}_{MeteringPointId}_{Payer_Id}_{InvoicingPerio-
dId}_{InvoicingCycleLength}"; }
    }
```

Ohjelmakoodiote 10. *ITestObject* rajapinnan toteuttava *InvoiceTestObject* luokka

6.2 Testattavan taulun lähtötilanteen talteenotto ja palautus

InWorksin laskut voidaan uudelleen generoida, kun tietokanta on saatettu sellaiseen tilaan, että inWorksin ohjelmakoodin validoinnit sallivat samojen laskujen uudelleenmuodostuksen. Alkuperäinen ajatus testien suorittamiseen oli ajaa ne transaktioiden sisässä, jotta tilanne olisi voitu palauttaa alkuperäiseen transaktionin *Rollback* metodilla. Tämä toteutus kaatui siihen, että transaktiot eivät sallineet sisäkkäisiä transaktioita, mitä inWorks laskun muodostus sisälsi ennestään. Tämän takia piti toteuttaa metodit, jotka keräävät tarvittavat tiedot inWorks kannasta väliaikaiseen tauluun, nollaavat

laskutuksen koontitaulujen arvot, jotta laskut voidaan muodostaa ja metodi, jolla inWorks kanta saadaan alkuperäiseen tilaansa. Metodit toteutettiin käyttämällä T-SQL kyselyjä *DbContextin SQLCommand* metodin avulla.

6.3 Laskutustestin ajaminen

Laskutustestin ajaminen aloitetetaan luomalla *InvoicingTestCase* luokasta uusi instanssi. *InvoicingTestCase* on luokka, joka perii *TestCaseBase* luokan ja kantaluokan konstruktori hoitaa testiajon alustuksen ja alkuperäisen vertailudatan ajamisen väliaikaistauluun. Kantaluokan *Execute* metodia kutsuttaessa toteuttavan luokan *ExecuteTest* metodissa määritelty testi aloittaa suorituksen.

```
public override void ExecuteTest()
        {
            foreach (var ip in _invoicingPeriodIds)
            {
                _invoicingPeriodId = ip;
                InitializeTest();
                var isprt = new InvoicingStartPointReturningTools(TempTableName, _repo);
                DeleteInvoicesByInvoicingPeriod(_invoicingPeriodId, (int)_utilityType);
                isprt.CopyOriginalInvoiceDataForReturningValues();
                isprt.UpdateInvoiceDataForGeneration();
                ReGenerateInvoices();
                LoadResultsToTable();
                CompareResults();
                isprt.ReturnOriginalValuesToDB();
            }
        }
```

Ohjelmakoodiote 11. *InvoiceTestCase* luokan toteutus *ExecuteTest* metodista

InvoiceTestCasen ExecuteTest metodissa on määritelty testin suoritusjärjestys ja se kutsuu muita testin hoitavia metodeja. Suoritettavat *InvoicingStartPointReturningTools* instanssin metodit ovat inWorksin kantaa kopioivia, muokkaavia ja palauttavia metodeita, joilla laskutuksen alkuperäinen tilanne pyritään pitämään kunnossa. *DeleteInvoicesByInvoicingPeriod* ja *ReGenerateInvoices* ovat metodeja, jotka kutsuvat inWorksin esikatseltavissa olevien laskujen poisto toimintoa ja laskujen muodostamiseen tarkoitettua metodia. *LoadResultsToTable* ja *CompareResults* ovat metodeita, jotka alustavat ja järjestävät datan listoihin, joita käytettään kantaluokan dataa vertailevassa *LoadResultsToTestCaseDB* metodissa.

6.3.1 Vertailtavan datan alustus

```
private void LoadResultsToTable()
        {
            var tableName = Table;

            var dropResultTable = $@"IF OBJECT_ID('{ResultTableName}', 'U') IS NOT NULL DROP TABLE {ResultTable-
Name}";
            _repo.GetContext().Database.ExecuteSqlCommand(dropResultTable);

            var cmd = $@"WITH result1 as
            (SELECT t.{Columns.Aggregate((current, next) => $"{current}, t.{next}")}
            FROM {tableName} t
            JOIN {TempTableName} tmp on t.Contractid = tmp.contractid WHERE t.InvoicingPeriodId={_invoic-
ingPeriodId} AND t.InvoiceState = 1
            EXCEPT
            SELECT {string.Join(",", Columns)} FROM {TempTableName}),
            result2 as
            (SELECT {string.Join(",", Columns)} FROM {TempTableName}
            EXCEPT
            SELECT t.{Columns.Aggregate((current, next) => $"{current}, t.{next}")}
            FROM {tableName} t
            JOIN {TempTableName} tmp on t.Contractid = tmp.contractid WHERE t.InvoicingPeriodId={_invoic-
ingPeriodId}  AND t.InvoiceState = 1)
            SELECT * INTO {ResultTableName} FROM
```

```
            (select ContractId, MeteringPointId, Payer_Id, Payer_Code, CustomerDetails_Code, MeteringPointCode, 
InvoicingPeriodId, InvoicingCycleLength from result1
            union
            select ContractId, MeteringPointId, Payer_Id, Payer_Code, CustomerDetails_Code, MeteringPointCode, 
InvoicingPeriodId, InvoicingCycleLength from result2) x";

            _repo.GetContext().Database.ExecuteSqlCommand(cmd);
        }
```

Ohjelmakoodiote 12. LoadResultsToTable metodi.

Laskujen uudelleen generoinnin jälkeen siirrytään datan alustukseen kantaluokassa tapahtuvaa vertailua varten. Alustus tapahtuu laskutustestin tapauksessa kahdessa eri metodissa; Ensimmäisessä metodissa *LoadResultsToTable* eroavaisuudet inWorksin laskutaulun ja alkuperäisten laskujen taulun, joka luotiin testin alustuksessa ladataan vielä kolmanteen väliaikaistauluun. Vertailu tapahtuu T-SQL:n EXCEPT lausetta käyttämällä ja rivejä vertaillaan molempiin suuntiin, jotta kaikki poikkeavuudet saadaan tarkastettua. Väliaikatauluun, jonka nimi on määritelty *ResultTableName* muuttujassa, tallennetaan poikkeavien rivien tunnisteet.

```
private void CompareResults()
        {
            var newRows = new List<InvoiceTestObject>();
            var oldRows = new List<InvoiceTestObject>();
            var contractIds = _repo.GetContext().Database.SqlQuery<int>($@"SELECT ContractId FROM {ResultTable-
Name}").ToList();
            var test = _repo.GetContext().Database.SqlQuery<string>($@"
            SELECT CONCAT(ContractId,'_',MeteringPointId,'_',Payer_Id,'_',InvoicingPeriodId,'_',Invoic-
ingCycleLength) as Identifier FROM {ResultTableName}").ToList();
            if (test.Any())
            {
                foreach (var tes in test)
                {
                    var newRowsSql = $@"SELECT {string.Join(",", Columns)} FROM {Table} WHERE CONCAT(Con-
tractId,'_',MeteringPointId,'_',Payer_Id,'_',InvoicingPeriodId,'_',InvoicingCycleLength) = @tes AND Invoic-
eState={(int)InvoiceStateEnum.WaitingForReview} AND InvoicingPeriodId={_invoicingPeriodId}";
                    var oldRowsSql = $@"SELECT {string.Join(",", Columns)} FROM {TempTableName} WHERE CON-
CAT(ContractId,'_',MeteringPointId,'_',Payer_Id,'_',InvoicingPeriodId,'_',InvoicingCycleLength) = @tes";
                    //newRows.Add(_repo.GetContext().Database.SqlQuery<InvoiceTestObject>(sql, tes, _invoic-
ingPeriodId).SingleOrDefault());
                    newRows.Add(_repo.GetContext().Database.SqlQuery<InvoiceTestObject>(newRowsSql, new SqlPa-
rameter("@tes",tes)).SingleOrDefault());
                    oldRows.Add(_repo.GetContext().Database.SqlQuery<InvoiceTestObject>(oldRowsSql, new SqlPa-
rameter("@tes",tes)).SingleOrDefault());
                }
                LoadResultsToTestCaseDB(newRows, oldRows);
            }
            else
            {
                _testRun.TestResult = TestResultEnum.Passed;
            }
        }
```

Ohjelmakoodiote 13. *CompareResults* metodi

Toisessa metodissa *CompareResults,* muodostetaan kaksi listaa *oldRows* ja *newRows*. Aiemman vertailun sisältäessä poikkeavia rivejä, haetaan ne inWorksin laskutaulusta sekä alkuperäisten laskujen taulusta erillisiin listoihin, jotka sitten siirtyvät kantaluokan vertailtavaksi. Eriävien tapauksien puuttuessa, testi merkataan onnistuneeksi tietokantaan ja testiajo lopetetaan.

7 KÄYTTÖLIITTYMÄ

Ohjelmistokehykseen on toteutettu myös ASP.NET käyttöliittymä, jotta testejä olisi helpompi selata ja korjata. Käyttöliittymä mahdollistaa testiajojen selaamisen, virheellisten ajojen rivi- ja solukohtaisten arvojen vertailun.

7.1 Testiajot

Käyttöliittymän avautuessa näkyviin tulevat kaikki testiajot. Ajot ovat listattuna taulukkoon, joka on muodostettu käyttämällä jQueryn kolmannen osapuolen kirjastoa JQuery DataTablesia. Rivikohtaisesti ovat listattuna testiajojen perustiedot. Riviä klikkaamalla päästään tarkastelemaan testiajon rivikohtaisia virheitä ja korjattuja bugeja.

Test results

TestRuns

Show 10 entries   Search:

Id	Execution Time	BatchId	TestRun Name	Result
1042	2017-11-26T19:24:03.617	7a9d4a25-fa88-446c-842a-9da8e8daf9a7	IdentifierCheckTest	Passed
1041	2017-11-26T19:11:12.687	918e84d9-4aa9-4806-9e27-f0d5c1f5b6c6	IdentifierCheckTest	Failed
1040	2017-11-26T18:49:38.01	770b73dc-02ae-4398-ba1c-3a5ec8331522	IdentifierCheckTest	Failed
1039	2017-11-26T18:11:07.753	be19530c-de81-4d48-95a7-868bf6daf904	IdentifierCheckTest	Failed
1038	2017-11-08T15:41:52.917	a88d4688-1be4-4b18-84fa-57dfd1ad9cdf	IdentifierCheckTest	Failed
1037	2017-11-08T15:03:58.553	81a2d7b1-c3d5-4c2e-9660-9408d3084dd9	IdentifierCheckTest	Failed
1036	2017-11-08T14:33:56.207	674573ee-7888-4810-980b-2e421bc6c84b	IdentifierCheckTest	Failed
36	2017-11-05T17:34:17.677	f58b08bc-6eea-4b6c-aef5-0ef288164cfe	NoLoopForIP	Failed
35	2017-11-04T19:38:47.863	f3b1fe69-ffb0-4709-9306-e95eb9516267	IdentifierCheckTest	Failed
34	2017-11-04T19:06:38.867	2992fa25-c5a9-434b-a1a8-61cab59e32d6	IdentifierCheckTest	Failed

Showing 1 to 10 of 43 entries   Previous 1 2 3 4 5 Next

© 2018 - Inworks testing framework

KUVA 11. Käyttöliittymän etusivu.

7.2 Virheelliset rivit ja solut

Virheelliset objektit ovat listattuna toisella käyttöliittymän sivulla, joka avautuu kun testiajon riviä on klikattu. *DataTable* elementti on tällä sivulla jaettu kahtia bootstrapin *tab* välilehtiä käyttämällä. Ensimmäisellä välilehdellä Test Objects on listattuna testiajon virheelliset objektit ja Fixed Objects välilehdellä korjatuiksi bugeiksi merkatut objektit. Näytettävinä tietoina ovat muunmuassa rivin yksilöivä tunniste ja puuttuvat rivikohtaiset arvot.

Testiobjektin eroavat solukohtaiset arvot ovat listattuna samalla näytöllä. Eroavat kenttäarvot saa näkyviin klikkaamalla objektirivin plus nappia. Alitaulussa näytettävinä tietoina on alkuperäiset ja uudet eroavat solukohtaiset arvot.

TestRuns Failed rows & Fixed object

Test Objects Fixed Objects

Show 10 entries Search:

		Id	TestRun Id	Identifier	New Value Null?	Old Value Null?	Reason For Null
	+	2215	1040	77_77_2930_25_1	false	false	
	+	2216	1040	1150_1180_4499_25_1	false	false	

Field	Old Value	New Value	Info
Total	192.9800	192.3600	
TotalWithoutVAT	155.6100	155.1100	
Vat	37.3700	37.2500	

		Id	TestRun Id	Identifier	New Value Null?	Old Value Null?	Reason For Null
	+	2217	1040	1154_1184_3918_25_1	false	false	
	+	2218	1040	1159_1189_4503_25_1	false	false	

KUVA 12. Testiajon virheelliset ja korjatut rivit.

7.3 Käyttöliittymän rakenne

Käyttöliittymä on totetutettu käyttämällä MVC suunnittelumallia. Käyttöliittymä rakentuu kahdesta erillisestä näkymästä ja niiden omista controllereista, Typescriptillä toteutetusta käyttöliittymä logiikasta, sekä tietokannan dataa lisäävästä sekä hakevasta ApiContollerista. Toteuksessa on mukailtu inWorksin vastaavaa pienemmässä mittakaavassa.

7.3.1 Näkymät

Käyttöliittymän näkymät ovat toteuteutukseltaan hyvin yksinkertaisia ja suurin osa käyttöliittymällä näkyvästä datasta hoidetaan TypeScript koodilla. Näkymät sisältävät sivun otsikot, sekä *table* attributilla varustetun paikan *DataTables* taululle. Näkymällä on määritelty ladattava JavaScript tiedosto, joka on käännös saman nimisestä TypeScript toteutuksesta. Kaikki käyttöliittymän näkymät jakavat myös yhden pohjanäkymän, jonka body elementtiin, kukin näkymä ladataan. Pohjanäkymä sisältää sivun header- ja footer osiot, sekä myös molempien näkymien jakavat javascriptin lataukset.

```
@{
    ViewBag.Title = "TestRuns";
}
<script>
    var testObjectUrl = "@Url.Action("TestObject", "TestObject", new { Area = "" })";
</script>
<div class="row">
    <div class="col-md-6"><h2>TestRuns</h2></div>
</div>
<div class="row">
    <div class="col-xs-12">
        <table id="testRunsTable" class="table table-condensed table-hover"></table>
    </div>
</div>

@section Footer {
    @Scripts.Render("~/Scripts/App/index.js")
}
```

Ohjelmakoodiote 14. Testiajonäkymän cshtml koodi.

7.3.2 Typescript ja DataTables

```
$(function () {
        var def = getAllTestRuns();
        def.done((data) => {
            var rows = [];
            for (var i = 0; i < data.length; i++) {
                rows.push(generateTestRunRow(data[i]));
            }
            var columns = tableHelper.getColumns(Models.pageTypeEnum.TestRun);
            var helper = new Models.dataTableHelper();
            helper.columns = columns;
            helper.data = rows;
            helper.tableName = tableName;
            tableHelper.setDataTable(helper, [[1,"desc"]]);
            $('#' + helper.tableName + ' tbody').on('click', 'tr', function () {
                var data = helper.dataTable.row(this).data();
                var test = testObjectUrl;
                window.location.href = testObjectUrl + "?testRunId=" + data[0];
            });
        });
    });
```

Ohjelmakoodiote 15. Testiajonäkymän latauksessa ajettava TypeScript koodi.

Käyttöliittymän tiedon hakemisen, selaimen logiikan sekä DataTables luokan alustukset on hoidettu TypeScriptillä. Näkymä toimii sivun pohjana, johon data ladataan typescriptillä ja se asetetaan luomalla uusi *DataTable* sille näkymällä määritettyyn paikkaan. Metodi *getAllTestRuns* tekee *HTTP-get* tyyppisen kutsun *ApiControllerille* ja palautuvasta datasta luodaan DataTables taulun rivejä *generateTestRunRow* metodissa. Metodissa *getColumns* alustetaan testiajonäkymän DataTables taulun solut, joita pitää olla sama määrä kuin tauluun ladattavan datan soluja. *dataTableHelper* on luokka, joka on toteutettu helpottamaan *DataTablesin* luontia, kullakin näkymällä. Molemmat käyttävät samaa *dataTableHelper* luokkaa omilla ominaisuuksillaan, jotta saadaan kullekkin näkymälle sopivat taulukot. Metodi *setDataTables* luo *helperille* annettujen ominaisuuksien mukaan uuden *DataTablen*.

Testiajonäkymän taulukolle sidotaan myös jQueryllä taulukon rivin klikkausta kuunteleva event, joka ohjaa käyttäjän rivikohtaisten arvojen näkymälle.

7.3.3 ApiController

Käyttöliittymän ApiController hoitaa näkymien datan hakemisen ja testiobjektien merkkaamisen korjatuksi/ei korjatuksi. ApiControlleria kutsutaan TypeScriptin *apiHelpers* luokan *Get, Post, Put* ja *Delete* metodeilla, jotka sitten *call* metodi muuttaa halutunlaiseksi JQuery AJAX-kutsuksi. Luokka *apiHelpers* on kopio inWorksin vastaavasta AJAX-kutsuja hoitavasta luokasta Parametrit ja palautuvat datat liikkuvat typescriptin ja C# ApiControllerin välillä JSON-objekteina. Kuvassa on testiajonäkymän testiajojen tiedot hakeva *GetAllTestRuns* metodi.

```
public class TestRunController : ApiController
    {
        //private TestFrameWorkDbContext context = new TestFrameWorkDbContext();
        private TestFrameWorkRepository _testRepository = new TestFrameWorkRepository();
        //private readonly ITestFrameWorkRepository _repository = IoC.Resolve<ITestFrameWorkRepository>();

        [AcceptVerbs("GET")]
        public List<TestRun> GetAllTestRuns()
        {
            var ret =_testRepository.GetAllQueryable<TestRun>().OrderByDescending(x => x.Execution-
TimeStamp).ToList();
            return ret;
        }
```

Ohjelmakoodiote 16. *ApiController* luokan perivä *TestRunController*.

8 POHDINTA

Työn tavoitteena oli toteuttaa ohjelmistokehys, jolla saadaan helposti selville koodiversioiden väliset generoitavassa datassa tapahtuneet muutokset. Lisäksi sovittiin, että tehdään toteutus inWorksin laskun muodostukseen. Lopputuloksena työstä saatiin toimiva ohjelma, joka kuitenkin odottaa vielä omaa testiympäristöä inWorksistä, jossa toteuttettua laskutuksen testiä voisi suorittaa.

Työn toteutus osio pysyi aikataulussa ja ensimmäinen versio valmistui keväällä 2017. Kesällä päätin vielä töissä saamani ohjelmointikokemuksen innoittamana kirjoittaa käyttöliittymä projektin noin 75 prosenttisesti uudestaan.

Tehdessäni toteutusta tuli opittua paljon energia-toimialasta ja sen monimuotoisuudesta. Kokemusta karttui lisää myös .NET ja ASP.NET ohjelmoinnista sekä erillaisista kolmannen osapuolen komponentteista kuten Nunit:sta ja JQuery DataTable:stä.

Työssä mielestäni onnistui hyvin ohjelmistokehyksen toteutus. Pohja kehykselle saatiin valmiiksi pikaisella aikataululla ja sen jälkeen oli helppo ohjelmoida yksityiskohtia sekä testata tuotoksia. Käyttöliittymä puolen ensimmäisen version logiikkaan en ollut tyytyväinen ja sen takia päätinkin ohjelmoida sen uudestaan. Olisin tutkinut enemmän energia-alaa ja sen laskutusta, jos aloittaisin työn tekemisen nyt ja näillä tiedoilla. Raportointiosion aikataulutus meni myös huonosti johtuen erinäisistä syistä ja sattumuksista.

9 JATKOKEHITYSIDEOITA

Ohjelmistokehyksen päälle tulisi kehittää paljon erilaisia dataa vertailevia testitapauksia. Testitapauksia voisi toteuttaa esimerkiksi kaikkeen generoitavaan tai tiedostoista tuotavaan dataan. Laskutuksen testitapausta voisi kehittää lisäämällä laskutuksen muodostukseen tarvittavien kohteiden vertailun sekä laskurivien tulostuskohtaiset vertailut. Ohjelmistokehyksen lokitusta voisi vielä parantaa ja ajossa kaatuneiden testien poikkeukset voisi myös tallentaa testikannan niille varattuun tauluun. Osa ohjelmistokehyksen ja käyttöliittymän ohjelmakoodista kaipaa myös hiukan selkeämpää muotoilua ja nimeämistä.

Käyttöliittymän ulkoasua voisi muotoilla inWorksin ulkoasua vastaavaksi ja siihen voisi toteuttaa oman näkymän erityyppisten testien ajamiseksi. Lisäksi kaatuneiden testiajojen poikkeukset voisi näyttää omalla näkymällään. Näytettävän datan määrää taulukoissa voisi myös lisätä kuten muun-muassa testin lopetusajan ja keston. Testiobjekteille voisi lisätä testattavan objektin tyypin.

LÄHTEET

BRAINVIRE INFOTECH INC. 2016. Few Good Reasons for Why you Need Web APIs for ASP.NET Application Development. [Verkkojulkaisu]. [Viitattu: 2018-4-11]. Saatavilla: https://www.brainvire.com/few-good-reasons-for-why-you-need-web-apis-for-asp-net-application-development/

DAITYARI, SHAUMIK. 2015. Jump Start Git. [Verkkojulkaisu]. Paddington: SitePoint. [Viitattu: 2018-4-11]. Saatavilla: https://books.google.fi/books?id=uRp5CgAAQBAJ

ENTITYFRAMEWORKTUTORIAL.NET. 2016. What is Code-First? [Verkkojulkaisu]. [Viitattu: 2018-4-11]. Saatavilla: http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx

GARCÍA, BONI. 2017. Mastering Software Testing with JUnit 5. [Verkkojulkaisu]. Birmingham: Packt Publishing. [Viitattu: 2018-4-8]. Saatavilla: http://proquest.safaribooksonline.com.ezproxy.savonia.fi/book/software-engineering-and-development/software-testing/9781787285736

GURU99. 2018. Static Testing Vs Dynamic Testing. [Verkkojulkaisu]. [Viitattu: 2018-4-8] Saatavilla: https://www.guru99.com/static-dynamic-testing.html

HARWANI, B. M. 2015. Learning Object-oriented Programming in C♯ 5.0. [Verkkojulkaisu]. Hampshire: Cengage Learning. [Viitattu: 2018-4-11]. Saatavilla: https://books.google.fi/books?isbn=1285854578

HYVÄRINEN, TAUNO. 2014. Kuopio : Solteq Oy

JOHANAN, JOSHUA;KHAN, TALHA JA ZEA, RICARDO. 2016. Web Developer's Reference Guide. [Verkkojulkaisu]. Birmingham: Packt Publishing. [Viitattu: 2018-4-11]. Saatavilla: http://proquest.safaribooksonline.com.ezproxy.savonia.fi/book/web-development/9781783552139

JORGENSEN, PAUL C. 2016. Software Testing, 4th Edition. [Verkkojulkaisu]. Boca Raton: Auerbach Publications. [Viitattu: 2018-2-3]. Saatavilla: http://proquest.safaribooksonline.com.ezproxy.savonia.fi/book/software-engineering-and-development/software-testing/9781466560680

KULMALA, KARRI. 2017. inWorks-laskutus. Seinäjoki : Solteq Oyj, 2017

LEWIS, WILLIAM E. 2017. Software Testing and Continuous Quality Improvement, Third Edition, 3rd Edition. [Verkkojulkaisu]. Boca Raton: Auerbach Publications. [Viitattu: 2018-2-3]. Saatavilla: http://proquest.safaribooksonline.com.ezproxy.savonia.fi/book/software-engineering-and-development/software-testing/9781351722209

MICROSOFT. 2017. An introduction to Azure Functions. [Verkkojulkaisu]. [Viitattu: 2018-4-19]. Saatavilla: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview

MICROSOFT. 2018. Introduction to object storage in Azure. [Verkkojulkaisu]. [Viitattu: 2018-4-19]. Saatavilla: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction

MICROSOFT. 2016. Entity Framework Working with DbContext. [Verkkojulkaisu]. [Viitattu: 2018-1-22]. Saatavilla: https://msdn.microsoft.com/en-us/library/jj729737(v=vs.113).aspx

MICROSOFT. 2017. Introduction to the C# Language and the .NET Framework. [Verkkojulkaisu]. [Viitattu: 2017-8-1]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

MICROSOFT. 2017. Entity Framework Overview. [Verkkojulkaisu]. [Viitattu: 2018-4-11]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

MICROSOFT. 2015. interface (C# Reference). [Verkkojulkaisu]. [Viitattu: 2018-1-21]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

MICROSOFT. 2016. Overview of ASP.NET Core MVC. [Verkkojulkaisu]. [Viitattu: 2017-11-5] .Saatavilla: https://docs.microsoft.com/fi-fi/aspnet/core/mvc/overview

MICROSOFT. 2017. ADO.NET Overview. [Verkkojulkaisu]. [Viitattu: 2018-5-11]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview

MICROSOFT.ASP.NET overview. [Verkkojulkaisu]. [Viitattu: 2017-11-5]. Saatavilla: https://docs.microsoft.com/fi-fi/aspnet/overview

MICROSOFT. 2015. Inheritance (C# Programming Guide). [Verkkojulkaisu]. [Viitattu: 2018-1-21]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance

MICROSOFT. 2017. SQL Server Overview. [Verkkojulkaisu]. [Viitattu: 2017-11-6]. Saatavilla: https://technet.microsoft.com/en-us/library/ms166352(v=sql.90).aspx

NLOG. Welcome to NLog! [Online] [Viitattu: 30. 1 2018.] http://nlog-project.org/.

POOLE, CHARLIE JA PROUSE, ROB. 2017. NUnit. [Verkkojulkaisu]. [Viitattu: 2017-11-6]. Saatavilla: http://nunit.org/

SHARMA, LAKSHAY. 2016. toolsqa. toolsqa. [Verkkojulkaisu]. [Viitattu: 2018-3-22]. Saatavilla: http://toolsqa.com/software-testing/difference-between-error-mistake-fault-bug-failure-defect/

SINHA, CHANDAN. 2015. c# & the .Net Framework. [Verkkojulkaisu]. [Viitattu: 2017-8-1]. Saatavilla: https://books.google.fi/books?id=TYVGCgAAQBAJ

SOLTEQ OYJ. 2017. inWorks asiakastiedon hallinta ja laskutus. Jyväskylä : Solteq Oy

SOLTEQ OYJ. 2017. Solteq ostaa inpulse works oyn laajentaa utilities toimialalle ja vahvistaa bi ja analytiikkaosaamistaan. [Verkkojulkaisu]. [Viitattu: 2017-11-19]. Saatavilla: https://www.solteq.com/fi/sijoittajat/tiedotteet-ja-julkaisut/tiedotteet/solteq-ostaa-inpulse-works-oyn-laajentaa-utilities-toimialalle-ja-vahvistaa-bi-ja-analytiikkaosaamistaan/

—. Solteq. Yritys. [Verkkojulkaisu]. [Viitattu: 2017-11-19]. https://www.solteq.com/fi/yritys/

THE JQUERY FOUNDATION. 2017. jQuery API. [Verkkojulkaisu]. [Viitattu: 2017-11-6] Saatavilla: https://api.jquery.com/

TUTORIALSPOINT. 2018. Software Testing Tutorial. [Verkkojulkaisu]. [Viitattu: 2018-2-20]. Saatavilla: https://www.tutorialspoint.com/software_testing/index.htm.

MICROSOFT. 2018. Microsoft Azure. [Verkkojulkaisu]. [Viitattu: 2018-5-18]. Saativilla: https://azure.microsoft.com/en-us/overview/what-is-azure/

MICROSOFT. 2018. Visual Studio IDE overview . [Verkkojulkaisu]. [Viitattu: 2018-5-18]. Saativilla: https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide

TECHOPEDIA. Transact-SQL (T-SQL). [Verkkojulkaisu]. [Viitattu: 2018-5-18]. Saatavilla: https://www.techopedia.com/definition/24476/transact-sql-t-sql

WOLFF, IVO GABE DE. 2016. TypeScript Blueprints. [Verkkojulkaisu]. Birmingham: Packt Publishing. [Viitattu: 2018-4-11]. Saatavilla: http://proquest.safaribooksonline.com.ezproxy.savonia.fi/book/web-design-and-development/9781785887017