Miika Hämäläinen

MVVM-MALLIN MUKAISEN WPF-SOVELLUKSEN YKSIKKÖTESTAUS

Opinnäytetyö
IT-Tradenomi

2017

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Miika Hämäläinen	IT-Tradenomi (AMK)	Toukokuu 2017

Opinnäytetyön nimi

MVVM-mallin mukaisen WPF-sovelluksen yksikkötestaus

43 sivua

Toimeksiantaja

Savcor Oy

Ohjaajat

Jukka Selin, Vili Leino

Tiivistelmä

Ketterän ohjelmistokehityksen yleistyttyä ohjelmistotestausta pidetään nykyisin koko ajan yhä tärkeämpänä osana ohjelmistotuotantoa, kehitysympäristöstä riippumatta. Opinnäytetyössä tutkittiin erityisesti yksikkötestausta ja miten MVVM-ohjelmistoarkkitehtuurimallin mukaiselle WPF-sovellukselle on mahdollista toteuttaa yksikkötestit ja toimiva testiympäristö.

Työssä esiteltiin testauksen rooli ohjelmistotuotannossa yleisesti, sekä keskityttiin tarkemmin yksikkötestaukseen ja sen periaatteisiin. Opinnäytetyön teoriaosuudessa esiteltiin kaikki työn tekoon käytetyt menetelmät, jotka oli tärkeä ymmärtää lopputuloksen ja johtopäätöksien kannalta.

Käytännönosiossa tavoitteina oli luoda yksinkertaiselle prototyyppisovellukselle asianmukaiset yksikkötestit. Testien teko- ja suunnitteluvaiheessa prototyyppisovelluksen ohjelmakoodia jouduttiin hieman muokkaamaan, sekä joistakin näkymämallin riippuvaisuuksista jouduttiin luomaan rajapintoja ja mock-olioita, jotta sille oli edes mahdollista luoda yksikkötestit. Testien teko- ja suunnitteluprosessi dokumentoitiin ja sen pohjalta analysoitiin johtopäätöksiä ja tuloksia.

Lopputuloksena prototyyppisovellukselle onnistuttiin luomaan toimiva testiympäristö ja yksikkötestit. Lopputuloksesta voitiin tehdä johtopäätös, että yksikkötestien luominen MVVM-mallin mukaiselle WPF-sovellukselle on mahdollista, mutta ainakin kokonaiselle näkymämallille yksikkötestien tekeminen voi olla hankalaa ja monimutkaista useiden ulkoisten riippuvaisuuksien takia.

Asiasanat

ohjelmistotestaus, yksikkötestaus, WPF-sovellus, MVVM-sovellusarkkitehtuurimalli

XAMK
University of Applied Sciences

Author (authors)	Degree	Time
Miika Hämäläinen	Bachelor of Business Administration	May 2017

Thesis Title

Unit testing the WPF application by implementing a MVVM pattern

43 pages

Commissioned by

Savcor Oy

Supervisors

Jukka Selin, Vili Leino

Abstract

Because agile software development has become more common, software testing is currently considered an increasingly more important part of software production regardless of the development environment. The objective of the thesis was to study particularly unit testing and to solve the problem: How is it possible to create unit tests and a test environment for WPF application that implements a MVVM pattern.

The thesis presented the role of testing in software engineering in general and described more details of unit testing and its principles. The theoretical part of the thesis presented all the methods used for the practical part of the thesis. These methods were important to understand for the outcome and conclusions of the work. The objective of the practical part was to create appropriate unit tests for a simple prototype application. Because of this, the prototype application program code had to be edited. Also some of the view model dependencies needed interfaces and data mocking to make it possible to create unit tests for it. This entire process was documented and analyzed for the results and conclusions.

The result of the thesis was a fully functional unit testing environment, accompanied with various test cases for the prototype application. This demonstrated how unit testing were done in practice for the WPF- application that implements a MVVM pattern. The outcome was that creating unit tests for this kind of application was possible, but unit testing a complete view model can be difficult and complex due to a number of external dependencies.

Keywords

software testing, unit testing, WPF application, MVVM pattern

SISÄLLYS

1 JOHDANTO

Tutkin opinnäytetyössäni perinteisen Windows-alustaisen MVVM-mallin mukaan toteutetun WPF-sovelluksen yksikkötestausta. Opinnäytetyön toimeksiantajana toimii Savcor Oy, joka toimittaa mm. toiminnanohjausjärjestelmiä metsäalalle. Ohjelmistotestausta pidetään nykyisin kokoajan yhä tärkeämpänä osana ohjelmistotuotantoa ketterän ohjelmistokehityksen yleistyttyä. Keskityn opinnäytetyössä erityisesti yksikkötestaukseen, jossa keskitytään ohjelman pienimpien komponenttien testaamiseen.

Opinnäytetyön tavoitteena oli tutkia, miten MVVM-mallin mukaiselle WPF-sovellukselle on mahdollista toteuttaa yksikkötestejä ja toimiva testiympäristö. Käytännön osiossa luotiin prototyyppisovellukselle asianmukaiset yksikkötestit ja toimiva testiympäristö. Prototyyppisovellus toteutettiin toimeksiantajan käyttämässä kehitysympäristössä sekä samoja tekniikoita käyttäen.

Opinnäytetyön toisessa luvussa on käsitelty ohjelmistotestauksen teoriaa yleisellä tasolla, sekä tarkemmin yksikkötestausta. Luvussa on myös tutkittu testauksen integroitumista ohjelmistotuotantoon. Kolmannessa luvussa on esitelty käytännön työhön käytetyt tekniikat. Luvussa on kerrottu yleisesti kaikista käytetyistä tekniikoista, mutta WPF-käyttöliittymäkirjasto ja MVVM-sovellusarkkitehtuurimalli ovat käsitelty tarkemmin, koska opinnäytetyön kannalta on tärkeää ymmärtää MVVM-mallin toimintaperiaate sekä WPF-käyttöliittymäkirjaston tarjoamat hyödyt. Neljännessä luvussa on esitelty prototyyppisovellus ja dokumentoitu käytännön osioon kuuluva työ eli prototyyppisovellukselle tehdyt yksikkötestit sekä testiympäristö. Opinnäytetyön viimeisessä luvussa eli päätelmässä on pohdittu lopputulosta, ongelmakohtia, sekä esitetty mahdollisia jatkokehitysideoita työlle.

2 OHJELMISTOTESTAUS

Ohjelmistotestaus on keino varmistaa ohjelman laadukkuus ja toiminnollisuus, koska sen avulla ohjelmasta etsitään järjestelmällisesti vikoja ja puutteita. Ohjelmistotestausta voidaan tehdä monella eri tasolla, automatisoidusti tai manuaalisesti. Tässä luvussa käsittelen ohjelmistotestausta yleisenä osana ohjelmistotuotantoa, sekä erityisesti yksikkötestausta.

2.1 Testaamisen eri tasot

Testaus on prosessi, jossa suoritetaan ohjelmaa niin, että siitä on aikomuksena löytää virheitä. Testien tehtävänä ei ole osoittaa, että ohjelma toimii, vaan sen tehtävänä on löytää mahdollisimman paljon virheitä. (Myers 2012, 6.) Suurin ohjelmiston testaukseen liittyvä harhaluulo tai käsitys on, että testien tehtävänä on vain ja ainoastaan osoittaa, että ohjelma toimii. Tällöin niissä ei keskitytä löytämään mahdollisia virhetilanteita. Patton (2001, 19) kiteyttääkin asian seuraavasti: ”Jos testaat vain asioita joiden pitäisi toimia ja asettelet testit niin, että ne läpäistään, jää tällöin huomaamatta kaikki asiat jotka eivät toimi.”

Testaamisen tulee olla aina systemaattista ja harkittua, jotta sen tuoma lisäarvo ja hyöty saadaan parhaiten irti. Testiprosessi tulee jakaa pienempiin testitapauksiin ja ohjelman vääränlainen käyttö täytyy myös ottaa huomioon. Yleensä testitapaukset keskittyvät ohjelman pieniin osa-alueisiin tai yksittäisiin komponentteihin. Näin testien avulla voidaan paikantaa mahdolliset virhetapaukset mahdollisimman vaivattomasti ja aikaisessa vaiheessa, sekä kaikki testit muodostavat yhdessä kattavan kokonaisuuden.

Mustalaatikkotestaus (engl. black-box testing) on testausta, jossa ei välitetä ohjelman sisäisistä mekanismeista tai komponenteista, vaan se keskittyy yksinomaan siihen, miten ohjelman reagoi suoritettaviin syötteisiin ja suoritusehtoihin (Williams 2006). Mustalaatikkotestausta voidaan kutsua myös toiminnalliseksi testaukseksi. Nimi ”mustalaatikkotestaus” tulee siitä, että testattava ohjelma ajatellaan mustana laatikkona, jonka sisään testaaja ei näe.

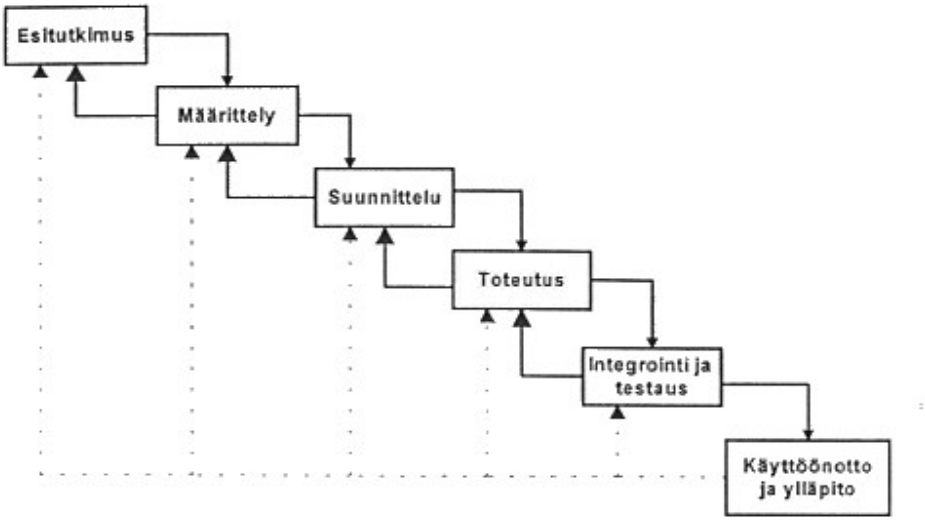
Lasilaatikkotestaus (engl. white-box testing) on testausta, joka keskittyy lähdekooditasolla testaamaan ohjelman mekanismeja tai komponentteja. Näin kehittäjät voivat tutkia toimivatko heidän koodinsa odotetulla tavalla (Williams 2006). Eli toisin kuin mustalaatikkotestauksessa, lasilaatikkotestauksessa ajatellaan testattava ohjelma lasisena laatikkona, jonka sisään testaaja näkee ja päälimäisenä tarkoituksena on testata, miten ohjelma toimii lähdekoodi tasolla. Lasilaatikkotestausta suorittaa yleensä kehittäjä itse.

Kolmas keskeisimmistä testimenetelmistä on harmaalaatikkotestaus (engl. gray-box testing). Harmaalaatikkotestauksessa yhdistellään osittain mustalaatikko- ja lasilaatikkotestausta molempia. Ohjelmaa testaan mustalaatikkona, mutta testausta laajennetaan osittaisella tuntemuksella ohjelmaa suorittavasta lähdekoodista (Patton 2009, 207).

2.2 Testauksen rooli ohjelmistotuotannossa

Ohjelmiston testaaminen kuuluu poikkeuksetta yhtenä osana ohjelmistotuotantoon, käytetystä menetelmästä riippumatta. Käytetystä menetelmästä riippuu se, missä vaiheessa tuotantoa, miten ja miksi testausta suoritetaan sekä kuinka iso rooli testauksella on tuotannossa. Esittelen tässä luvussa kaksi yleisimmin käytettyä ohjelmistotuotannon menetelmää ja miten testaus integroituu niihin.

Vesiputousmalli on yksi perinteisimmistä ohjelmistotuotannon vaihejakomalleista. Mallin perusajatuksena on, että sovellus suunnitellaan vaiheittain lineaarisesti. (Tietojenkäsittelytieteen laitos 2009.) Termi vesiputous viittaakin siihen, että suunnittelu- ja toteutusprosessi etenee vaihe vaiheelta alaspäin kuin vesiputous. Yleensä edelliseen vaiheeseen ei palata, kun se on hyväksytysti suoritettu. Kuvassa 1 on havainnollistettu vesiputousmallin vaihejakomalli sekä miten testaus integroituu siihen.



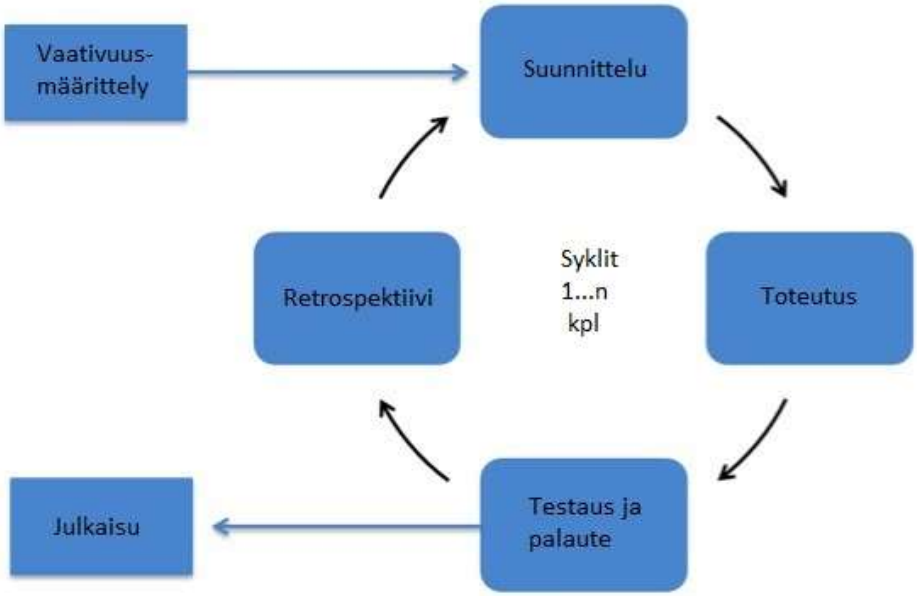
Kuva 1. Tyypillinen vesiputousmalli (Haikala & Märijärvi, 2004)

Kuten kuvasta 1 voidaan päätellä, testauksen rooli vesiputousmallissa on pieni ja sillä ei pyritä parantamaan ohjelmaa konseptitasolla, koska vaativuusmäärittelyä ei ole tarkoitus muuttaa alussa tehtyjen päätöksien jälkeen. Vesiputousmallissa testaus tulee toteutuksen jälkeen, joten sen tarkoitus on vain varmistaa, että sovellus toimii suunnitellusti. Vesiputousmalli soveltuu monesti vain hyvin rajoitettuun määrään projekteja. Nykyisin se ei sovellu suurimpaan osaan projekteista, koska on hyvin yleistä, ettei projektien alkuvaiheessa vielä tiedetä tarkalleen, minkälainen lopputuote tulee olemaan. (Haikala & Märijärvi 2004.)

Ketterällä ohjelmistokehityksellä tarkoitetaan ohjelmistotuotantoprojekteissa käytettyjä useita samantyylisiä menetelmiä, joissa perusidea on samanlainen, mutta erilaisia variaatioita on monenlaisia. Ketterässä kehityksessä lähdetään yleensä siitä, että ei ole vain yhtä oikeaa tapaa saavuttaa tavoiteltu lopputulos. Tästä johtuen harvojen ketterien menetelmien ohjeistossa määritellään se, miten missäkin tilanteessa pitää toimia. Pakollisten käytäntöjen sijaan kaikkien ketterien menetelmien taustalla on kokoelma periaatteita, joiden avulla uskotaan päästävän hyvään lopputulokseen. (Tolvanen 2013.)

Fowlerin ja Highsmithin (2001) manifestin mukaan ensisijaiset tavoitteet yleensä ovat toimiva ohjelmisto, suora viestintä ja nopea muutoksiin reagointi sekä pyrki-

mys minimoida riskejä jakamalla ohjelmistokehitys lyhyisiin sykleihin. Tällöin kehitys tapahtuu iteratiivisesti lyhyissä sykleissä ja kehitystä on mahdollista ohjata havaittujen tarpeiden mukaan. Syklit pidetään tyypillisesti 1 - 4 viikon mittaisena ”projekteina”, jonka aikana kehitys tapahtuu yhteistä päämäärää tavoitellen ja se on yleensä julkaistavissa oleva tuote tai ominaisuus. Kuvassa 2 olen havainnollistanut ketterän ohjelmistokehityksen vaiheet ja siinä näkyy myös testauksen integroituminen sykleihin.



Kuva 2. Ketterän ohjelmistokehityksen vaiheet

Ketterät ohjelmistokehitys menetelmät poikkeavat huomattavasti perinteisestä ja kaavamaisesta vesiputousmallista. Nykyään ylivoimaisesti suurin osa ohjelmistotuotannon projekteista toteutetaan ketterien ohjelmistokehitys menetelmien mukaan. Kuten kuvasta 2 voidaan päätellä, testauksen rooli verrattuna vesiputousmalliin kasvaa huomattavasti, koska kehityksen vaiheet toistuvat jatkuvasti jokaisessa syklissä. Näin ollen toisin kuin vesiputousmallissa, testausta ei suoriteta pelkästään ohjelman toimivuuden varmistamiseksi, vaan sillä on vielä mahdollista vaikuttaa myös konkreettisesti tuotteeseen sekä siinä käytettyjen ratkaisujen toimivuuteen. Toinen merkittävä havainto tässä on, että mahdolliset ongelmat ja puutteet pystytään havaitsemaan aikaisessa vaiheessa ja niihin kyetään reagoimaan nopeasti jo sen hetkisessä tai seuraavissa syklissä.

2.3 Yksikkötestaus

Keskityn tässä opinnäytetyössä erityisesti vain yksikkötestaukseen, joka on tyypillisintä lasilaatikkotestausta. Yksikkötestauksella tarkoitetaan testaamista, jossa testataan ohjelman lähdekoodin yksittäisiä osia. Käytännössä siis testattava yksikkö ”eristetään” lähdekoodista itsenäiseksi testattavaksi yksiköksi. Termillä yksikkö viitataankin ohjelman pienimpiin mahdollisiin toiminnollisuuksiin, kuten esimerkiksi olion tarjoamiin metodeihin. Sen päätavoitteena on pilkkoa ohjelma pieniin testattaviin yksiköihin ja varmistaa niiden oikeellisuus.

Yksikkötesti on automatisoitu ohjelmakoodi, joka kutsuu testattavaa yksikköä. Yksikkötesti on lähes aina kirjoitettu käyttämällä yksikkötestaukseen tarkoitettua ohjelmistokehystä. Se voidaan kirjoittaa helposti ja se suoriutuu nopeasti. Se on luotettava ja ylläpidettävä. Sen tulokset ovat johdonmukaisia ja ristiriidattomia, niin kauan kuin testattavaa yksikköä eli koodia ei muuteta. (Osherove 2014.)

Usein testattavassa yksikössä saattaa olla riippuvuuksia toisiin olioihin, jotka voivat olla monimutkaisiakin. Tällöin testattavaa koodia ei voida eristää suoraan yksiköksi, koska se on riippuvainen muista olioista. (Chaffee & Pietri 2002.) Jos testit tehtäisiin tästä huolimatta, ei testi enää olisi yksikkötesti vaan integraatiotesti, koska siinä testattaisiin useamman yksikön toimintaa. Jotta em. kaltaiselle tapaukselle voidaan luoda yksikkötesti, josta on eristetty yksikön riippuvuudet, täytyy apuna käyttää datan ”mockausta” (engl. data mocking).

Mock-oliot (engl. mock-object) imitoivat oikeaa oliota hallitusti ja tämä mahdollistaa sen, että testissä ei tarvitse käyttää alkuperäistä oliota toisesta yksiköstä. Tällöin testi voidaan luokitella yksikkötestiksi, koska se on itsenäinen testattava yksikkö. (Chaffee & Pietri 2002.) Yksikkötesti tulisi olla aina mahdollista suorittaa hyväksytysti ilman yksikön ulkoisia riippuvaisuuksia. Yksikkötestin tehtävä ei ole myöskään ottaa kantaa datan oikeellisuuteen, joten olioiden ”imitointi” on täysin hyväksyttävää. Tarvittavien Mock-olioiden toteuttaminen on yleensä ainakin

isommissa sovelluksissa todella työlästä ja monimutkaista, varsinkin jos sovelluksen kehitysvaiheessa ei ole ajateltu testattavuutta lainkaan. Mock-olioiden luomisen helpottamiseksi käytetään yleensä jotakin ohjelmistokehystä.

2.4 Yksikkötestaus ja testivetoinen kehitys

Testivetoinen kehitys (engl. test-driven development, TDD) on ohjelmointia tukeva tekniikka ja yleensä sitä käytetään, jotta saataisiin testauksesta irti kaikki sen tarjoamat hyödyt. Testivetoisen kehityksen ideana on, että luodaan ensin testitapaus ja kirjoitetaan sille testi ja vasta tämän jälkeen aletaan kirjoittamaan varsinaista tuotantokoodia niin, että se läpäisee testin. (Beck 2002.) Tällä pyritään lähtökohtaisesti parempaan suunnitteluun ja varmistumaan ohjelmiston oikeanlaisesta toiminnasta.

Lisäksi testivetoinen kehitys kannustaa yleensä kehittäjää tekemään siistimpää ja testattavampaa ohjelmakoodia, jonka toiminnan hän ymmärtää paremmin. Mikäli yksikkötestit kirjoitetaan tuotantokoodin jälkeen, jää usein huomaamatta niiden tarjoamat hyödyt. Tämä voi johtaa jopa siihen, että ne jäävät kirjoittamatta kokonaan, koska ne koetaan turhaksi ja niihin ei haluta kuluttaa aikaa.

3 KÄYTETTÄVÄT TEKNIIKAT

Tässä luvussa esittelen työtä ja prototyyppisovellusta koskettavat, keskeisimmät tekniikat. Niihin kuuluvat ohjelmankehitysympäristö, ohjelmointikielet, WPF-käyttöliittymäkirjasto ja MVVM-ohjelmistoarkkitehtuuri malli sekä yksikkötesteissä käytetyt ohjelmistokehykset.

3.1 Ohjelmankehitysympäristö, ohjelmointikielet ja ohjelmistokehykset

Visual Studio on Microsoftin kehittämä ja ylläpitämä ohjelmankehitysympäristö. Se tukee useita eri ohjelmointikieliä ja sen tarjoamat keskeisimmät työkalut ovat koodieditori, debuggeri sekä visuaalinen editori käyttöliittymän suunnitteluun. WPF-sovellukset tehdään pääasiassa Visual Studiolla, joten se toimii tämänkin opinnäytetyön kehitysympäristönä. Useimmat nykyiset Visual Studio -versiot si-

sältävät valmiiksi myös Microsoftin oman yksikkötestaukseen tarkoitetun ohjelmistokehyksen MS Unitin, jolla tulen toteuttamaan tämän opinnäytetyön prototyyppisovelluksen yksikkötestit ja niiden suorittamisen.

XAML (Extensible Application Markup Language) on Microsoftin kehittämä XML-pohjainen kuvauskieli. XAML on kieli, jolla rakennetaan sovelluksen visuaalinen ilme, kuten HTML kielellä rakennetaan nettisivujen visuaalinen ilme. XAML on osa Microsoft Windows Presentation Foundationia (*lyhenne WPF*). WPF käyttää sitä nimenomaan käyttöliittymien kuvauskielenä ohjelmointikielien sijasta. (What is XAML 2017.)

C# on yksinkertainen, moderni, olio-pohjainen ja tyyppiturvallinen ohjelmointikieli. C# syntaksi on helposti omaksuttavaa kaikille kenellä on tuntemusta C, C++ tai Java ohjelmointikielistä. C# yksinkertaistaa monia C++ ohjelmointikielen monimutkaisuuksia, mutta tarjoaa sen tehokkaat ominaisuudet, joita ei Java-ohjelmointikielestä löydy. (Introduction to the C# Language and the .NET Framework 2015.) WPF-sovellusten toimintalogiikka kirjoitetaan yleensä C# tai VB.NET ohjelmointikielellä.

Moq-ohjelmistokehys on yksi suosituimmista ja helppokäyttöisimmistä datan mockaukseen käytetyistä ohjelmistokirjastoista, jota käytetään .NET-ohjelmistokomponenttikirjaston kanssa. Moq-ohjelmistokehys helpottaa ja nopeuttaa huomattavasti olioiden mockausta. Se on täysin ilmainen ja avoimeen lähdekoodiin perustuva.

3.2 WPF-käyttöliittymäkirjasto

Windows Presentation Foundation (WPF) on Microsoftin kehittämä käyttöliittymäkirjasto käyttöliittymien renderöimiseen Windows-pohjaisissa sovelluksissa. Se julkaistiin vuonna 2006 osana silloista .NET ohjelmistokehystä ja se oli esiasennettuna Windows Vista-käyttöjärjestelmässä. Sen tarkoituksena oli ja on edelleen korvata perinteiset Win32-pohjaiset Windows Forms-käyttöliittymäkirjastot. (Yosifovich 2012, 1.)

3.2.1 Tärkeimmät ominaisuudet

Tämä luku perustuu MacDonaldin (2014) kirjassa esiteltyihin WPF:n tuomiin suurimpiin muutoksiin ja ominaisuuksiin Windows-ohjelmoinnin maailmaan:

Web-tyylinen asettelumalli
WPF korostaa joustavaa ulkoasun asettelua, joka järjestää mieluimmin kontrollit niiden sisällön mukaan, eikä ankkuroidusti koordinaattien mukaan. Tämän ansiosta käyttöliittymät ovat erittäin dynaamisia ja responsiivisia.

Rikas piirtomalli
Pikseleiden piirtämisen sijaan WPF:ssä toimitaan mieluiten primitiivien kanssa (perusmuodot, tekstikappale ja muut graafiset elementit). Lisäksi siinä on myös uusia ominaisuuksia, kuten läpinäkyvyyden kontrollointi, ominaisuus kasata monta eri kerrosta (engl. layer) eri läpinäkyvyyksillä ja natiivi 3D-tuki.

Rikas tekstimallli
WPF mahdollistaa Windows-sovelluksessa näytettävän rikkaita, tyylillisiä tekstejä, missä tahansa kohtaa käyttöliittymää. Tekstejä voi jopa yhdistellä listojen, ”kelluvien” kuvioiden (engl. floating figure) sekä muiden käyttöliittymäelementtien kanssa. Lisäksi pitkien tekstien näyttämisen apuna voidaan käyttää edistyneitä tekstityökaluja luettavuuden parantamiseksi.

Animaatiot ohjelmoitavana konseptina
WPF:ssä animaatiot ovat luontaisena osana ohjelmistokehystä. Niiden piirtoon ei tarvita ajastimia tai muuta kikkailua, vaan WPF osaa suorittaa ne automaattisesti, kun ne on määritelty deklaratiivisten tagien avulla.

Tuki audio- ja videomedialle
Toisin kuin aiemmissa käyttöliittymätyökaluissa, kuten Windows Formsissa - WPF:ssä on huomattavasti parempi ja laajempi tuki audio- ja videomedialle. Se osaa toistaa kaikkia Windows Media Playerin tukemia formaatteja ja tukee jopa

usean median toistoa samanaikaisesti. WPF mahdollistaa myös näiden täydellisen ”upottamisen” osaksi käyttöliittymää, kuten vaikkapa esimerkiksi videoikkunan upottaminen pyörivään 3D-kuutioon on täysin mahdollista.

Tyylit ja ”sapluunat” (engl. template)
WPF:ssä pystyy luomaan tyylejä, joita on mahdollista uudelleenkäyttää koko ohjelman laajuisesti. Kaikkien elementtien (jopa peruselementtien) renderöintityyliä on mahdollista muokata. Näiden ansiosta erilaisten sekä koko ohjelman laajuisten teemojen luominen on helppoa kehittäjälle.

Komennot (engl. Commands)
WPF:ssä on mahdollista ”linkittää” yksi ja sama, koodissa määritelty komento moneen eri kontrolliin. Esimerkiksi käyttöliittymässä oleva nappi ja työkalupalkin valikosta löytyvä painike voivat käynnistää täsmälleen saman suoritettavan komennon.

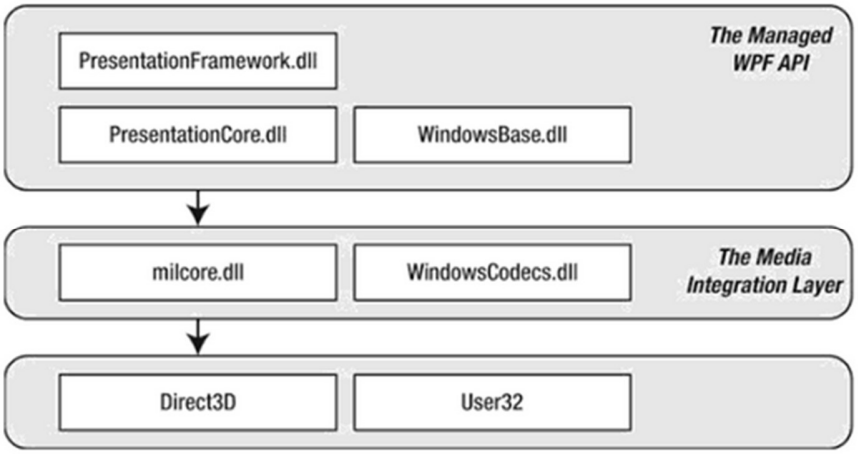
Deklaratiivinen käyttöliittymä
WPF ja eritoten Visual Studio mahdollistavat sovelluksen ikkunan sisällön rakentamisen XAML:la. Visual Studion visuaalinen käyttöliittymäeditori generoi tarvittaessa automaattisesti tarvittavan XAML:in käyttöliittymään tehdyistä muutoksista. Tämä mahdollistaa sen, että käyttöliittymän voi suunnitella tai tehdä täysin erillään ohjelman toimintalogiikasta.

Sivu-perustaiset ohjelmat
WPF:llä on mahdollista rakentaa selaintyylisiä sovelluksia, jossa on mahdollista navigoida ”sivujen” välillä aivan kuten esimerkiksi nettisivuilla.

Resoluutio riippumattomuus
WPF hyödyntää vektorigrafiikkaa, jonka ansiosta sillä tehdyt sovellukset ovat resoluutioriippumattomia. Tämä tarkoittaa sitä, että suuremmalla resoluutiolla tai Dpi:llä (dots per inch) kaikki ei pienene kuten perinteisissä Windows Forms -sovelluksissa, vaan grafiikka ja tekstit vain tarkentuvat.

3.2.2 WPF:n arkkitehtuuri

WPF käyttää monikerroksista arkkitehtuuria. Arkkitehtuuri koostuu kuvan mukaisesti kolmesta eri kerroksesta. Pääpiirteittäin ylin taso sisältää WPF -oliot ja tyypit. Keskimmäinen taso vastaa .NET -olioiden kääntämisestä oikeaan muotoon Direct3D-renderöijälle, joka puolestaan muodostaa alimman tason.



Kuva 3. WPF arkkitehtuurin rakenne (MacDonald 2012, 11)

PresentationFramework.dll (Kuva 3) sisältää ylimmäntason WPF-tyypit, kuten ikkunat, paneelit ja muun tyyppiset kontrollit. *PresentationCore.dll* sisältää perustyypit, josta esimerkiksi kaikki kontrollit ja UI-elementit perivät. *WindowsBase.dll* sisältää tärkeitä olioita, kuten DispatcherObject ja DependencyObject, jotka vastaavat mm. näppäimistön syötteistä, hiiren liikkeistä ja elementtien ominaisuuksista.

Keskimmäisen kerroksen *milcore.dll (Kuva 3)* kääntää visuaaliset elementit Direct3D:lle sopivaan muotoon. *milcore.dll* ei ole pelkästään osa WPF:ää, vaan se on myös olennainen järjestelmäkomponentti kaikissa Windows -käyttöjärjestelmissä Vistasta lähtien, koska DWM (Desktop Window Manager) käyttää sitä työpöydän renderöimiseen. *WindowsCodecs.dll* on rajapinta, joka tarjoaa tuen digitaalisten kuvaformaattien käsittelyyn.

Alimmalla tasolla *Direct3D* on rajapinta (Kuva 3), jota WPF-sovellus käyttää kaiken grafiikan renderöimiseen. User32 on rajapinta, jota kaikki ohjelmat käyttävät, mutta WPF-sovelluksessa sen osana ei ole osallistua renderöimiseen.

3.3 MVVM-arkkitehtuurimalli

Tässä luvussa esittelen MVVM-sovellusarkkitehtuurimallin, joka on nykyisin suosituin sovelluksissa käytetty arkkitehtuurimalli, kun puhutaan WPF-sovelluskehityksestä. MVVM lyhenne tulee sen komponenteista (Model-View-ViewModel). MVVM-mallin päätarkoituksena on erottaa sovelluksen toimintalogiikka sen käyttöliittymästä. Pieniin yksinkertaisiin sovelluksiin MVVM-mallin tuominen voi olla hieman ”yliampuvaa” ja se vain monimutkaistaa kehitystä turhaan, koska toimintalogiikka voidaan kirjoittaa suoraan käyttöliittymän ”taakse” (engl. code-behind). Suuremmissa sovelluksissa, joissa noudatetaan MVVM-mallia oikein, se helpottaa sovelluksen testausta, ylläpitoa ja kehitystä. Lisäksi se mahdollistaa käyttöliittymän suunnittelun erillään toimintalogiikan ohjelmoinnista.

3.3.1 MVVM-mallin historiaa

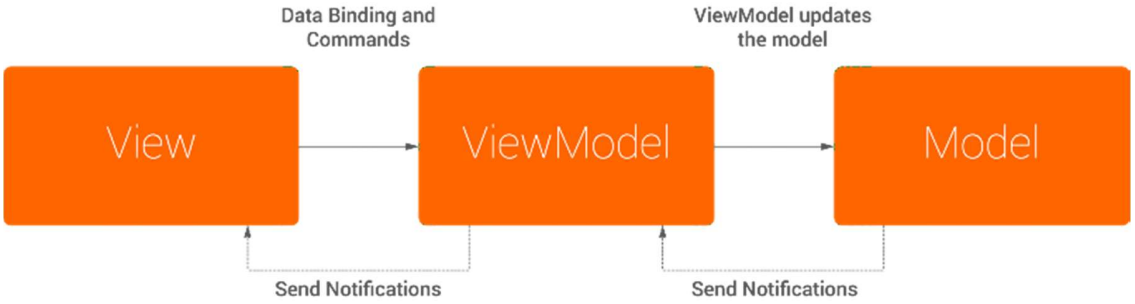
MVVM-mallin alkuperäistarkoituksena on päästä eroon joistakin MVC- ja MVP-mallien rajoitteista ja yhdistellä osia näiden vahvuuksista. Ihan ensimmäinen ”versio” mallista nähtiin osana Smaltalk-ohjelmointikielen ohjelmistokehystä, jo 1980-luvulla Application Model -nimellä. Myöhemmin sitä paranneltiin ja nimi päivitettiin Presentation Modeliksi. (Vice & Siddiqi 2012, 77.)

Presentation Modelin (myöhemmin tekstissä PM-malli) esitteli Martin Fowler vuonna 2004. Ensimmäistä kertaa PM-malliin pohjautuva MVVM-malli esiteltiin Microsoftin ohjelmistoarkkitehti John Gossmanin blogissa vuonna 2005. (Garofalo 2011, 1.) Fowler esitteli PM-mallin tarkoituksena luoda alustariippumaton käyttöliittymäabstraktio sovelluksen näkymästä, kun taas Gossman esitteli MVVM-mallin standardoituna tapana yksinkertaistaa WPF:n käyttöliittymäohjel-

mointia ja hyödyntää sen tärkeimpiä ominaisuuksia. MVVM- ja PM-malli ovat käytännössä identtisiä, mutta MVVM-malli on vain räätälöity WPF- ja Silverlight-alustoille. (Smith 2009.)

3.3.2 Perusrakenne

MVVM-mallin päätarkoitus on tarjota tapa, jolla voidaan erottaa käyttöliittymän kontrollit ja niiden toimintalogiikka toisistaan. MVVM-mallissa on kolme eri komponenttia: malli (engl. model), näkymä (engl. view) ja näkymämalli (engl. viewmodel). Jokainen näistä palvelevat eri roolissa. Jotta ymmärtää jokaisen komponentin roolit, on myös tärkeää, että ymmärtää miten komponentit kommunikoivat toistensa kanssa. (The MVVM Pattern 2012.) Kuvassa 4 on havainnollistettu näiden kolmen komponentin välisiä yhteyksiä.



Kuva 4. MVVM-mallin komponentit ja niiden väliset yhteydet. Käytännössä näkymä tuntee näkymämallin ja näkymämalli tuntee mallin, mutta malli ei tunne näkymämallia ja näkymämalli ei tunne näkymää.

View eli näkymä

Näkymä on vastuussa mallin datan graafisesta visualisoinnista näytölle ja tarvittaessa se sisältää kaikki kontrollit. Eli toisin sanoen, se on eräänlainen käyttöliittymä mallin datalle, mutta se ei sisällä yhtään sovelluksen bisneslogiikkaa. Näkymä voi olla WPF-ikkuna, Silverlight-sivu tai pelkästään XAML-data sapluunan kontrolli (engl. data template control) (Garofalo 2011, 1).

ViewModel eli näkymämalli

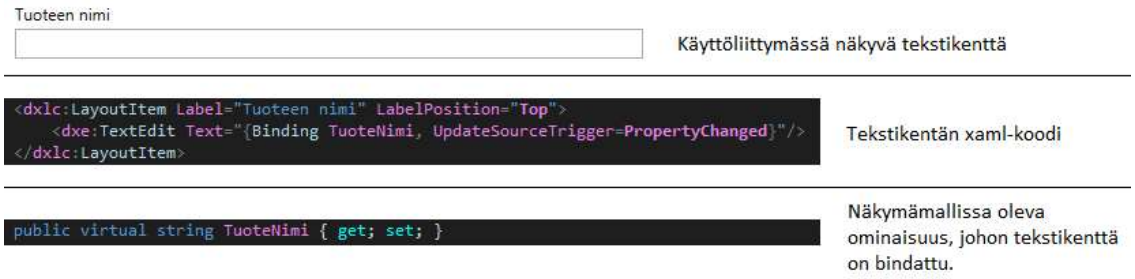
Näkymämalli toimii ”välittäjänä” näkymän ja mallin välillä, sekä se on vastuussa näkymä logiikasta. Yleensä näkymämalli kommunikoi mallin kanssa käyttämällä mallin omia metodeja. Tämän jälkeen näkymämalli välittää mallin datan näkymälle ja tarvittaessa muuntaa sen näkymälle sopivampaan muotoon. Näkymämalli myös tarjoaa myös toiminnollisuuden näkymän komennoille. (The MVVM Pattern 2012.)

Model eli malli
Malli on entiteetti (itsenäinen kokonaisuus), joka edustaa ohjelman bisneslogiikkaa ja tietokantakerrosta. Se voi olla esimerkiksi yksinkertainen asiakas entiteetti tai vaikkapa monimutkainen varastotilanne entiteetti. (Garofalo 2011, 1.)

3.3.3 Toimintaperiaate

Käytännössä näkymämalli eristää näkymän ja mallin, jotta malli voi ns. ”elää” tai kehittyä itsenäisesti ilman näkymää. Koska malli ei tunne näkymämallia se lähettää sille vain ilmoituksia muutoksista (kuva 4). Vastaavasti näkymämalli ei tunne näkymää, se voi lähettää sille vain ilmoituksia, kuten muutos-, varoitus- ja virheilmoituksia (kuva 4). (The MVVM Pattern 2012.)

Näkymä sisältää käyttöliittymän, jonka kentät voidaan sitoa suoraan näkymämallin ominaisuuksiin datan sidonnan avulla (engl. data binding) (kuva 4). Datan sidonta voi olla yksisuuntaista, kumpaan suuntaan tahansa tai samanaikaisesti molempiin suuntiin. Näin ollen esimerkiksi kun käyttäjä kirjoittaa vaikkapa käyttöliittymän tekstikenttään se voi muuttaa näkymallin ominaisuutta, joka on sidottu ko. tekstikenttään. Kuvassa 5 on havainnollistettu, miten käyttöliittymässä näkyvä tekstikenttä voidaan bindata eli sitoa xaml-koodissa näkymämallin ominaisuuteen, mikäli näkymämalli on asetettu näkymän ”data-kerrokseksi” (engl. Data layer) DataContext-ominaisuuden avulla. Kuvassa 5 on käytetty seuraavassa luvussa esiteltyä DevExpressin ohjelmistokehystä, jolloin bindaus tapahtuu ohjelmoijan näkökulmasta hyvin yksinkertaisesti ilman ns. boilerplate-koodia.



Kuva 5. Tekstikentän sidonta näkymämallin ominaisuuteen eli bindaus.

Näkymä kommunikoi näkymämallin kanssa myös käyttäen komentoja, jotka voivat laukaista näkymämallissa esimerkiksi metodin. Komennot voidaan puolestaan sitoa esimerkiksi käyttöliittymän painikkeisiin, kuten kuvassa 6 on havainnollistettu. Tällöin kun käyttäjä painaa käyttöliittymän painiketta, voi se laukaista näkymämallissa esimerkiksi metodin, joka tallentaa muutokset malliin.

Kuva 6. Painikkeen sidonta näkymämallissa laukaistavaan metodiin eli bindaus.

Myös kuvassa 6 on käytetty seuraavassa luvussa esiteltyä DevExpressin-ohjelmistokehystä, jolloin tässäkään tapauksessa ei tarvita ns. boilerplate-koodia metodin laukaisemiseksi, vaan se onnistuu lisäämällä xaml-koodissa kutsuttavan metodin nimen perään pelkästään Command-teksti.

3.3.4 DevExpressin MVVM-ohjelmistokehys

WPF-sovelluksen toteuttaminen MVVM-mallin mukaisesti voi olla haastavaa ja monimutkaista, sekä joissain tapauksissa WPF-alusta sellaisenaan ei tarjoa

täyttä tukea MVVM-mallin mukaiselle kehitykselle. Näiden ongelmien ratkaisemiseksi on kehitetty monia eri kolmannen osapuolen tarjoamia ohjelmistokehyksiä. DevExpressin MVVM-ohjelmistokehys on yksi niistä. (Getting Started with DevExpress MVVM Framework 2013.)

Ohjelmistokehys tarjoaa paljon hyviä yleiskäyttöisiä ominaisuuksia ongelmien ratkomiseksi. Sen pääkomponentteja ovat näkymämalleihin liittyvät ratkaisut, kuten pääluokat BindableBase, ViewModelBase ja POCO-luokka, näkymallien väliseen kanssakäymiseen tarkoitetut rajapinnat ja Messenger-luokka sekä komennot, käytökset, palvelut, konvertterit ja datan annotaatioattribuutit (nimikoidut attribuutit). Ilman ohjelmistokehystä ohjelman lähdekoodeihin täytyisi kirjoittaa paljon ns. ”boilerplate”-koodia, jotta ohjelman voisi toteuttaa MVVM-mallin mukaisesti. DevExpress MVVM-ohjelmistokehyksen ansiosta tätä ei tarvitse tehdä, koska se generoi tarvittavan koodin ohjelman ajon aikana ns. ”pinnan alla”.

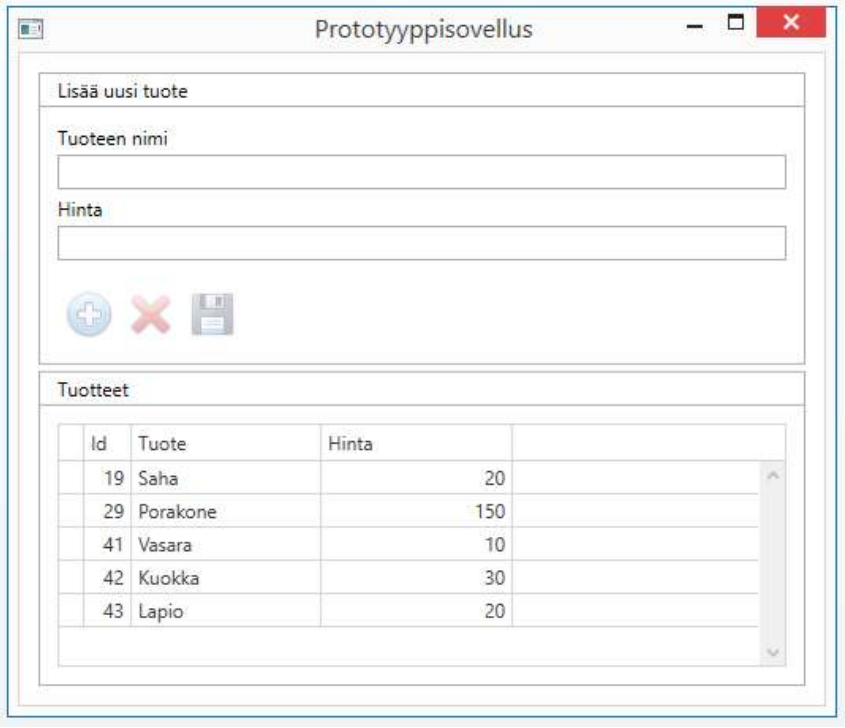
4 CASE: PROTOTYYPPISOVELLUS

Tässä luvussa esittelen toteuttamani prototyyppisovelluksen, joka on tehty pääpiirteittäin samoja tekniikoita käyttäen kuin Savcorin tekemät ohjelmistot. Sovellus on yksinkertainen CRUD-sovellus, jolla on mahdollista ylläpitää esimerkiksi kaupan tuotelistaa. Tuotteet tallennetaan MySQL-tietokantaan. Lopuksi tein prototyyppisovellukselle asianmukaiset yksikkötestit.

4.1 Prototyyppisovelluksen määrittely ja käyttötapaukset

Sovelluksen avulla tuotteita on mahdollista lisätä, poistaa ja muokata. Kaikki muutokset tallennetaan tietokantaan vasta, kun käyttäjä painaa tallennus-painiketta. Sovellusta ei käynnistetä, mikäli yhteyttä tietokantaan ei ole. Tästä ilmoitetaan käyttäjälle huomautus-ikkunalla ja tämän jälkeen sovelluksen suoritus suljetaan. Tässä huomionarvoista on se, että vaikka sovellusta ei voida käynnistää, mikäli yhteyttä tietokantaan ei ole, täytyy sovellukselle tehdyt yksikkötestit olla kuitenkin mahdollista suorittaa hyväksytysti siitä huolimatta.

Kaikki tuotteet ja niiden ominaisuudet näkyvät käyttöliittymässä muokattavana tuotelistana. Kuvassa 7 on kuvakaappaus ohjelman käyttöliittymästä eli näkymästä (PrototyyppiSovellusView).

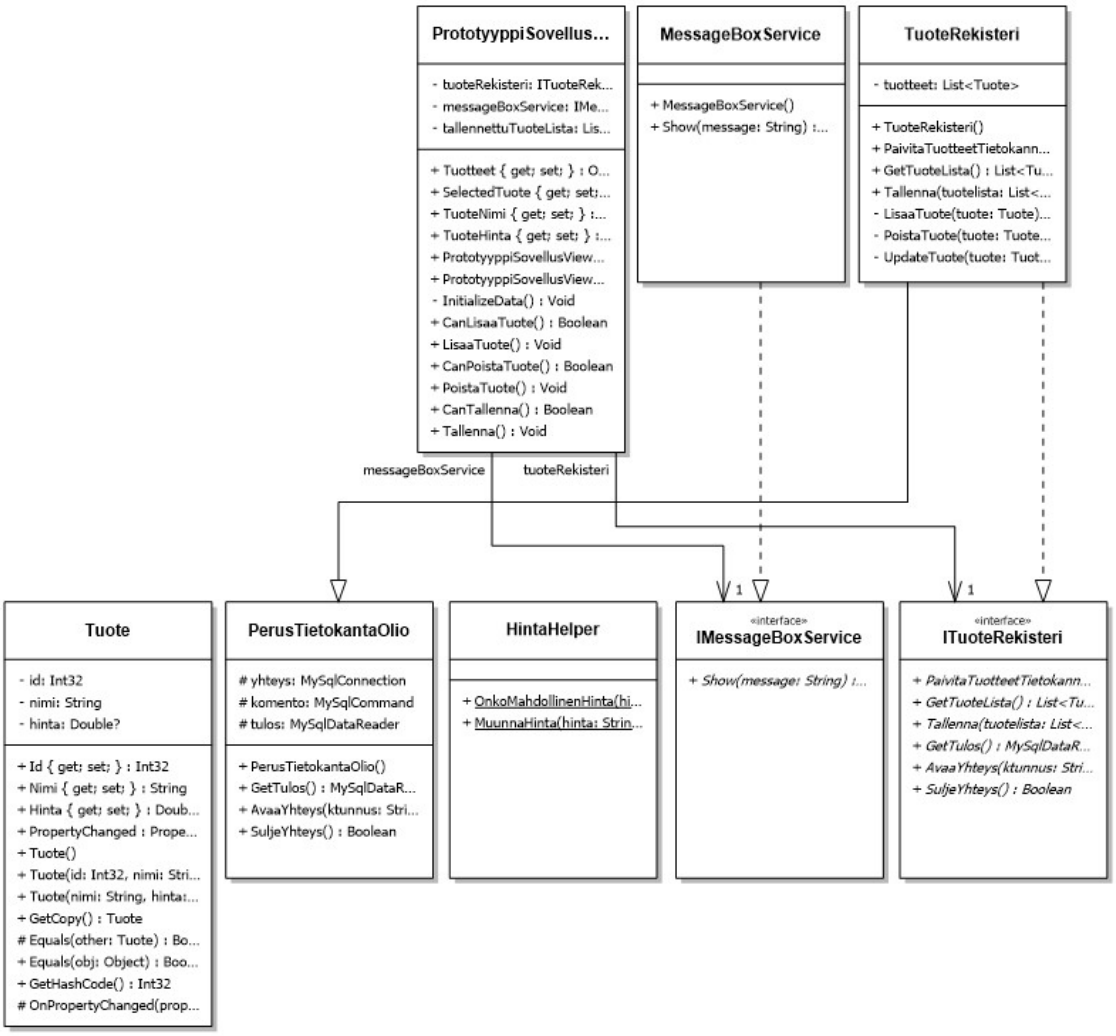


Kuva 7. Prototyyppisovelluksen käyttöliittymä eli näkymä.

Tallennus painike on aktiivisena vain kun tuotelistaan on tehty muutoksia. Tuotteiden lisäys painike on aktiivisena vain, kun lisättävälle tuotteelle on annettu nimi ja hinta. Poisto painike on aktiivinen vain kun tuotelistasta on valittu poistettava tuote. Sovelluksessa on myös validointi tuotteiden hinnalle. Tuotteen hinta voi sisältää vain lukuja ja hinnan täytyy olla positiivinen ja enemmän kuin nolla. Mikäli käyttäjä yrittää syöttää tuotteiden hinnaksi jotakin kelpaamatonta, huomautetaan siitä erillisellä huomautus-ikkunalla. Mikäli jostain syystä ohjelman käytön aikana yhteys tietokantaan katkeaa tai muutoksia ei muuten pystytä tallentamaan tietokantaan, ilmoitetaan siitäkin erillisellä huomautus-ikkunalla käyttäjälle.

4.2 Prototyyppisovelluksen luokkakaavio ja testien suunnittelu

Kun prototyyppisovellus oli toteutettu aloin suunnittelemaan sille yksikkötestejä. Toinen mahdollinen lähestymistapa olisi ollut aiemmin esittelemäni testivetoisen-kehityksen tyylinen ratkaisu, missä olisin ensin kirjoittanut yksikkötestit ja vasta sen jälkeen varsinaisen ohjelmakoodin, mutta tällä kertaa päädyin suunnittelemaan yksikkötestit vasta jälkikäteen ja muokkaamaan sen mukaan ohjelmakoodia tarvittaessa. Kuvassa 8 on esitetty lopullisen prototyyppiohjelman luokkakaavio. Siinä näkyvät kaikkien luokkien muuttujat, ominaisuudet, tietotyypit ja metodit.



Kuva 8. Lopullisen esimerkkisovelluksen luokkakaavio.

Tavoitteenani oli tehdä ohjelman kaikille mahdollisille luokille omat yksikkötestit, pois lukien PerusTietokantaOlio- ja siitä perivä TuoteRekisteri-luokka, jotka sisältävät kaiken tietokannan kanssa kommunikointiin tarvittavan logiikan.

Prototyyppiohjelmassa on vain yksi näkymä (engl. view) ja sille yksi näkymämalli (engl. viewmodel). Ohjelman mallina (engl. model) toimii TuoteRekisteri-luokka, joka on käytännössä ohjelman tietokantakerros. Varsinaiset Tuote-oliot muodostetaan Tuote-luokasta ja niiden ominaisuutena ovat id, nimi ja hinta. Ohjelman käyttöliittymä eli näkymä (PrototyyppiSovellusView) ei sisällä yhtään erillistä ohjelman toimintalogiikkaan vaikuttavaa ohjelmakoodia (engl. code-behind), vaan kaikki toimintalogiikka on siirretty näkymämallille (PrototyyppiSovellusViewModel) datan sidonnan eli bindauksen avulla. Tämä mahdollistaakin käyttöliittymän ja ohjelman toimintalogiikan testaamisen erikseen. Lisäksi tämä mahdollistaa myös sen, että käyttöliittymää voidaan muuttaa helposti, eikä se vaikuta näkymämallin toimintalogiikkaan tai sille tehtyihin yksikkötesteihin käytännössä ollenkaan.

Jotta näkymämallin metodit säilyisivät suhteellisen yksinkertaisina ja pystyisin havainnollistamaan myös yksinkertaisten luokkien, eikä vain näkymämallin testausta, päätin ns. ”ulkoistaa” osan validointi- ja muunnoslogiikasta HintaHelper-luokkaan. Tällöin tälle ns. apuluokalle on mahdollista tehdä helpommin omat yksinkertaisemmat yksikkötestit. HintaHelper-luokka voisi olla myös ohjelman jatkokehityksen kannalta hyödyllinen yleiskäyttöinen luokka muuallakin ohjelmassa, mikäli ohjelmaan tehtäisiin lisää ominaisuuksia, joissa käsiteltäisiin tuotteiden hintaan liittyvää validointia. Tällöin hinnan validointiin tai muunnoksiin tarvittavaa ohjelmakoodia ei enää tarvitsisi kirjoittaa uudelleen muualla tai mikäli siihen tarvittaisiin jotain uutta logiikkaa, voitaisiin se kirjoittaa tähän apuluokkaan.

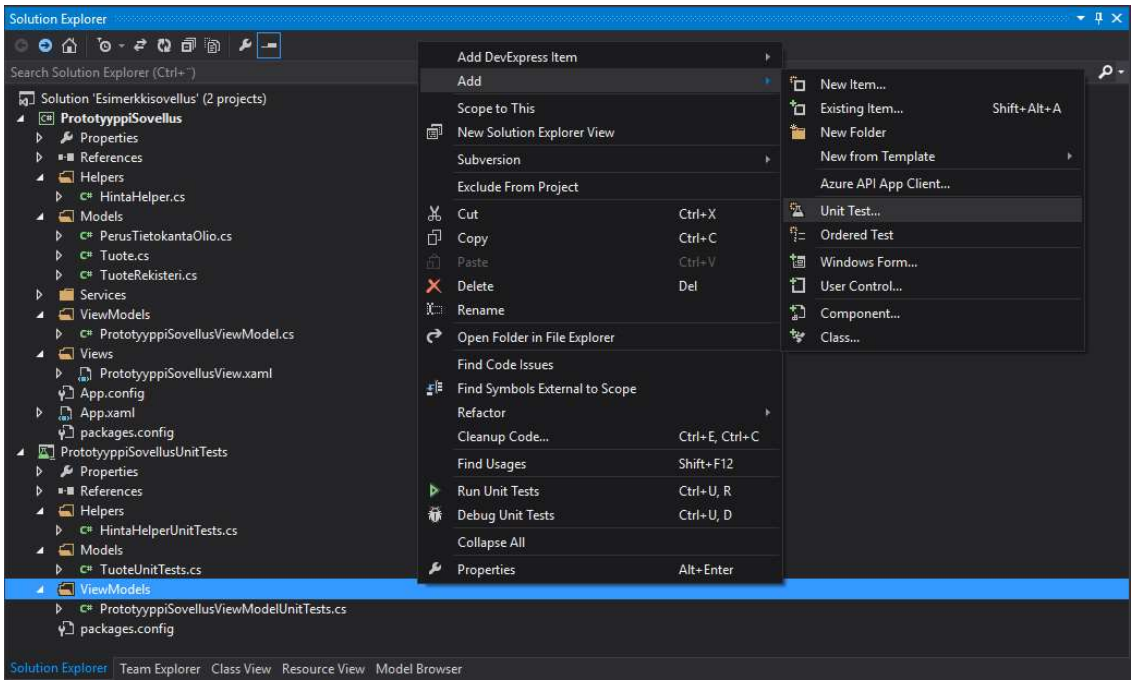
4.3 Testiympäristön luominen

Testiympäristön luominen Visual Studiossa ja MS Unit-ohjelmistokehyksellä tapahtuu luomalla uusi Unit Test Project- tyyppinen projekti. Projekti luodaan esimerkiksi, samaan Solutioniin eli Visual Studion projektiympäristöön, missä varsinainen testattava projekti on. Itse toimin juurikin näin. Tämän jälkeen luodun Unit Test-projektin referensseihin tulee lisätä testattava projekti eli tässä tapauksessa

PrototyyppiSovellus-projekti. Tässä vaiheessa täytyy testiprojektiin lisätä myös muut tarvittavat referenssit.

Minun tapauksessani muita tarpeellisia lisättäviä referenssejä olivat Moq- ja DevExpress MVVM-ohjelmistokehykset. Molemmat ohjelmistokehykset voidaan lisätä projektin referensseihin suoraan esimerkiksi Visual Studion NuGet-paketinhallintaohjelmalla tai lataamalla ja lisäämällä ne manuaalisesti GitHubista. Kun tarvittavat referenssit on lisätty, voidaan aloittaa toteuttamaan itse yksikkötestiluokkia.

Kuvassa 9 on kuvakaappaus Visual Studion Solution Explorerista, joka havainnollistaa projektien ja koko solutionin kansiorakenteen. Kuvassa 9 on lisäksi avattuna hiiren oikealla painikkeella valikko, jonka kautta yksikkötestin lisääminen haluttuun kansioon onnistuu helposti.



Kuva 9. Projektin ja testiprojektin kansiorakenne, sekä uuden yksikkötestin luominen.

Päätin toteuttaa testiprojektin kansioiden ja tiedostojen projektirakenteen samalla tavalla, kuin itse PrototyyppiSovellus-projektissa. Näin jokaista testattavaa luok-

kaa varten löytyy testiprojektista samanlaisesta kansiorakenteesta oma testiluokka, jonka ansiosta yksikkötestit on helppo löytää, jos niitä tarvitsee tulevaisuudessa esimerkiksi muokata tai kirjoittaa lisää.

4.4 Yksikkötestien kirjoittaminen yksinkertaisille luokille

Yksikkötestejä kirjottaessa tavoitteena on, että yksi testimetodi kuvaa yhtä tavoiteltua skenaariota. Usein testi metodeja joudutaan kirjoittamaan useita, jotta kaikki mahdolliset skenaariot tulevat testattua, jotka testattavan metodin tai ominaisuuden tulee läpäistä. Yksikkötestien metodien tulisi noudattaa johdonmukaista ja kaavaa, jotta toisten kehittäjien on helpompi ymmärtää niitä. Yleinen tapa/kaava, jota luotujen testien tulisi noudattaa, on seuraava: Arrange → Act → Assert. (Edwards 2013.) Kuvassa 10 on havainnollistettuna testimetodin perusrakenne. Hyvien tapojen mukaan testimetodit tulisi nimetä myös mahdollisimman kuvaavasti niin, että nimestä käy ilmi odotettu tulos ja se mitä testissä annetaan esimerkiksi syötteenä.

```
[TestMethod]
public void JokuTestiMetodi()
{
    // Arrange
    // *Kaikki tarvittava koodi testin valmistelemiseksi

    // Act
    //*Kaikki koodi jolla kutsutaan testattavaa metodia tai ominaisuutta

    // Assert
    //*Kaikki koodi, jolla todennetaan onko testi suoritettu odotetusti.
}
```

Kuva 10. Testimetodin perusrakenne.

Yksikkötestien kirjoittamisen päätin aloittaa ensimmäisenä luomalla yksikkötestit HintaHelper-luokalle, koska se on helposti testattavissa ja sen metodeissa ei ole mitään riippuvuuksia muihin luokkiin tai metodeihin, joten datan mockausta ei tarvita. HintaHelper-luokassa on kaksi julkista metodia, joille voidaan tehdä yksikkötestit. Ensimmäinen metodeista, joka tarkistaa annetun hinnan oikeellisuuden on toteutettu kuvan 11 mukaisesti.

```
public static bool OnkoMahdollinenHinta(string hinta)
{
    double result;

    if (Double.TryParse(hinta.Replace(".", ","), out result))
    {
        return !(result <= 0);
    }
    return false;
}
```

Kuva 11. Hinnan oikeellisuuden tarkistava OnkoMahdollinenHinta-metodi.

Kuten ohjelmakoodista (kuva 11) voimme päätellä, testattava metodi ottaa parametrina testattavan hinnan ja palauttaa sen mukaan boolean-tyyppisen totuusarvon, onko hinta mahdollinen vai ei. Toinen huomioitava seikka on se, että hinta annetaan metodille merkkijonona, eikä suoraan double-tyyppisenä muuttujana, koska käyttöliittymään sidotut tekstilaatikot palauttavat syötetyn hinnan tässä tapauksessa merkkijonona. Metodi tarkistaa myös, että onko hinta mahdollista muuntaa double-tyyppiseksi, koska tuotteiden hinnat ovat tietokannassa sen tyyppisenä. Tämän ansiosta metodi palauttaa epätoden totuusarvon, jos hinta sisältää muuta kuin numeroita. Desimaalierottimeksi hyväksytään pilkku tai piste. OnkoHintaMahdollinenHinta-metodille tein kuvan 12 mukaiset testimetodit.

```
[TestMethod]
public void OnkoMahdollinenHinta_TestEiKelvollinenHinta()
{
    const string negatiivinenHinta = "-5";
    const string nollaHinta = "0";
    const string tyhjaHinta = "";
    const string sisaltaTekstiaHinta = "5asd32";
    const string sisaltaaMerkkejaHinta = "4$*123";
    const string hintaKahdellaDesimaaliErottimella = "0,32,1";
    const string hintaSisaltaaOpraattoreita1 = "3+2-1*2/1";
    const string hintaSisaltaaSulkuja = "(23)";

    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(negatiivinenHinta));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(nollaHinta));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(tyhjaHinta));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(sisaltaTekstiaHinta));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(sisaltaaMerkkejaHinta));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(hintaKahdellaDesimaaliErottimella));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(hintaSisaltaaOpraattoreita1));
    Assert.IsFalse(HintaHelper.OnkoMahdollinenHinta(hintaSisaltaaSulkuja));
}

[TestMethod]
public void OnkoMahdollinenHinta_TestKelvollinenHinta()
{
    const string hinta = "123";
    const string hintaDesimaalilla1 = "0,13";
    const string hintaDesimaalilla2 = "0.13";

    Assert.IsTrue(HintaHelper.OnkoMahdollinenHinta(hinta));
    Assert.IsTrue(HintaHelper.OnkoMahdollinenHinta(hintaDesimaalilla1));
    Assert.IsTrue(HintaHelper.OnkoMahdollinenHinta(hintaDesimaalilla2));
}
```

Kuva 12. OnkoHintaMahdollinen-metodin testimetodit.

Ensimmäisessä testimetodissa testataan kelvottomia hintoja ja jälkimmäisessä testimetodissa hyväksyttäviä hintoja. Toinen HintaHelper-luokasta löytyvä ja kuvan 13 mukaisesti toteutettu metodi suorittaa hinta merkkijonon muuntamisen double-tyyppiseksi.

```
public static double MuunnaHinta(string hinta)
{
    var result = Double.Parse(hinta.Replace(".", ","));
    return result;
}
```

Kuva 13. MuunnaHinta-metodin ohjelmakoodi.

Metodille annetaan parametrina merkkijono ja se palauttaa muunnoksen double-tyyppisenä. Huomioitavaa on se, että tälle metodille on myös mahdollista antaa vääränlaisia merkkijonoja, joita ei voida muuntaa double-tyyppiseksi. Tällöin se aiheuttaa suoritettavaan ohjelmaan poikkeuksen. Tein metodin tarkoituksella niin, että se voi aiheuttaa poikkeuksen, mikäli sille annetaan vääränlainen merkkijono. Tämän avulla haluan näyttää, että yksikkötestien tuloksena on myös mahdollista odottaa poikkeuksia. Varsinaisessa tuotantosovellus-projektissa ei kannattaisi käyttää tämänlaista metodia, koska se saattaa aiheuttaa väärin käytettynä helposti poikkeuksen, johon ohjelma kaatuu. Tuotantosovelluksessa olisi helpointa ja viisainta käyttää suoraan Double.TryParse-metodia konversiossa, mutta asian havainnollistamiseksi esimerkkisovelluksessa tämä kelpaa hyvin. MuunnaHinta-metodille tein kuvan 14 mukaiset testimetodit, joissa yritetään tehdä epäonnistunutta konversiota ja tuloksena sen odotetaan aiheuttavan poikkeus.

```
[TestMethod]
[ExpectedException (typeof(NullReferenceException))]
public void MuunnaHinta_EiKelvollinenMerkkijono_TestTyhjäMerkkijono()
{
    HintaHelper.MuunnaHinta(null);
}

[TestMethod]
[ExpectedException(typeof(FormatException))]
public void MuunnaHinta_EiKelvollinenMerkkijono_TestEiMahdollistaMuuttaa()
{
    HintaHelper.MuunnaHinta("tekstiä");
}

[TestMethod]
[ExpectedException(typeof(OverflowException))]
public void MuunnaHinta_EiKelvollinenMerkkijono_TestYliVuoto()
{
    HintaHelper.MuunnaHinta((Double.MaxValue).ToString());
}
```

Kuva 14. MuunnaHinta-metodin testimetodit, jotka odottavat konversiosta aiheutunutta poikkeusta.

Odotettu poikkeus määritellään testin alussa. Odotetulle poikkeukselle on hyvä antaa odotetun poikkeuksen tyyppi, jotta muut mahdolliset testin aikana syntyvät poikkeukset eivät jää huomioimatta. Esimerkiksi `[ExpectedException (typeof(NullReferenceException))]` odottaa testin aiheuttavan NullReferenceException-tyyppisen poikkeuksen ja itse testin tuleekin aiheuttaa juuri odotetun tyyppinen poikkeus, tai muuten testi epäonnistuu.

Seuraavana tein MuunnaHinta-metodille kuvan 15 mukaiset testit, joissa odotetaan onnistunutta hinnan konvertointia. Tässä huomion arvoista on se, että tällä kertaa testin lopputulosta vertaillessa kannattaa mieluiten käyttää AreEqual-metodia, koska vertaillaan kahta double-tyyppistä muuttujaa.

```
[TestMethod]
public void MuunnaHinta_TestKelvollinenHinta()
{
    const string hintaInt = "234";
    const string hintaDesimaalilla1 = "2,3";
    const string hintaDesimaalilla2 = "2.4";

    Assert.AreEqual(234, HintaHelper.MuunnaHinta(hintaInt));
    Assert.AreEqual(2.3, HintaHelper.MuunnaHinta(hintaDesimaalilla1));
    Assert.AreEqual(2.4, HintaHelper.MuunnaHinta(hintaDesimaalilla2));
}
```

Kuva 15. MuunnaHinta-metodin testimetodit, jotka odottavat onnistunutta muunnosta.

Mikäli testi epäonnistuu, se kertoo tarkasti, mitä lukua odotettiin ja mikä oli tuloksena saatu luku, toisin kuin aiemmin käytetyt IsTrue- tai IsFalse-metodit. Jos tuloksen vertailuun käytettäisiin esimerkiksi aiemmissa testeissä käytettyä IsTrue-metodia, se kyllä toimisi, mutta vikatilanteessa se ei kertoisi muuta kuin, että odotettu IsTrue-ehto ei toteutunut, jonka takia testi on epäonnistunut.

Seuraavana päätin tehdä Tuote-luokalle mahdolliset yksikkötestit. Tuote-luokalla on vain yksi julkinen ja hyvin yksinkertainen kuvan 16 mukainen GetCopy-metodi, joka palauttaa kopion ko. Tuote-oliosta. Tästä huolimatta sille voi hyvin tehdä oman yksikkötestin.

```
public Tuote GetCopy()
{
    return new Tuote(Id ,Nimi, Hinta);
}
```

Kuva 16. Tuote-luokan metodi, joka palauttaa kopion Tuote-oliosta.

GetCopy-metodille tein kuvassa 17 näkyvän testimetodin, jossa ensin luodaan ja kopioidaan Tuote-olio ja sen jälkeen vertaillaan, onko kopioidun Tuote-olion ominaisuudet samat, kuin alkuperäisen.

```
[TestMethod]
public void GetCopy_TestOnnistunutKopiointi()
{
    var tuote = new Tuote
    {
        Id = 1,
        Nimi = "Testituote",
        Hinta = 19.90
    };

    var tuoteKopio = tuote.GetCopy();
    Assert.AreEqual(tuote.Id, tuoteKopio.Id);
    Assert.AreEqual(tuote.Nimi, tuoteKopio.Nimi);
    Assert.AreEqual(tuote.Hinta, tuoteKopio.Hinta);
}
```

Kuva 17. GetCopy-metodin ainoa testimetodi.

Testissä olisi myös madollista vertailla suoraan tuote- ja tuoteKopio-olioita toisiinsa, mutta kun vertaillaan kaikkia olioiden ominaisuuksia toisiinsa erikseen, saadaan epäonnistuneen testin sattuessa tarkempi kuvaus epäonnistumisen syystä.

4.5 Yksikkötestien kirjoittaminen näkymämallille

Seuraavaksi aloitin tekemään yksikkötestejä prototyyppisovelluksen näkymämallille eli PrototyyppiSovellusViewModel-luokalle. Näkymämallille on ehdottomasti monimutkaisempaa ja työläämpää tehdä yksikkötestejä, koska näkymämalli sisältää yleensä kaiken näkymän ja mallin välisen toimintalogiikan. Tästä johtuen sen metodeissa, joita olisi tarkoitus testata, on paljon riippuvaisuuksia toisiin luokkiin ja niiden metodeihin. Tämä onkin suurin ongelma yksikkötestien tekemisessä kokonaiselle näkymämallille, koska kuten aiemmissa luvuissa oli mainittu, yksikkötestin tulisi aina olla eristettynä yhdeksi testattavaksi yksiköksi.

Prototyyppisovelluksen tapauksessa suurin näkymämallin metodeissa käytetty riippuvaisuus on TuoteRekisteri-luokka, joka sisältää tietokantalogiikan. Testeissä täytyy pystyä korvaamaan tuo luokka Mock-oliolla, jotta oikeaa luokkaa ei tarvita käyttää ja sen näkymämallissa tarvittavat metodit voidaan yli kirjoittaa palauttamaan testitilanteessa haluttu paluuarvo. Tätä varten minun täytyi tehdä muutoksia alkuperäisen näkymämalliin ja TuoteRekisteri-luokkaan. Jotta TuoteRekisteri-luokasta voidaan luoda Mock-olio, täytyi sitä varten tehdä kuvassa 18 esitelty ITuoteRekisteri-rajapinta, jonka itse TuoteRekisteri-luokka toteuttaa.

```
public interface ITuoteRekisteri
{
    bool PaivitaTuotteetTietokannasta();
    List<Tuote> GetTuoteLista();
    bool Tallenna(List<Tuote> tuotelista);
    MySqlDataReader GetTulos();
    bool AvaaYhteys(string ktunnus, string sala);
    bool SuljeYhteys();
}

public class TuoteRekisteri: PerusTietokantaOlio, ITuoteRekisteri
{
```

Kuva 18.ITuoteRekisteri-rajapinta, jonka TuoteRekisteri-luokka toteuttaa.

Kun rajapinta oli luotu, TuoteRekisteri-luokka voitiin pistää toteuttamaan ITuoteRekisteri-rajapinta lisäämällä perittävän PerustietokantaOlion perään pilkulla ITuoteRekisteri (kuva 18). Muita muutoksia TuoteRekisteri-luokkaan ei tarvinnut tehdä eli varsinaiseen toimintakoodia ei ollut tarvetta muutella.

Toinen riippuvuus esimerkkisovelluksen näkymallissa käytetyissä metodeissa oli System.Windows-nimiavaruudesta löytyvä MessageBox-luokka. Sen Show-metodin avulla huomautetaan käyttäjälle esimerkiksi väärin syötetystä hinnasta, avaamalla erillinen huomautuksen ja OK-painikkeen sisältävä ikkuna. Jo sinälläänsäkin tämä alkuperäinen toteutus rikkoo MVVM-mallin sääntöjä, koska jos näkymämallissa avataan suoraan tällä tavoin toinen ikkuna, se sitoo näkymämallin käyttöliittymäarkkitehtuuriin. Myös yksikkötestauksen näkökulmasta tämä aiheuttaa vakavan ongelman, koska testiä ajettaessa ko. huomautus-ikkuna avautuu ja testi pysähtyy, kunnes joku painaa OK-nappulaa.

Jotta kyseinen toteutus noudattaisi MVVM-mallia ja se olisi mahdollista korvata testitilanteessa, täytyi prototyyppisovelluksen tapauksessa MessageBox-ikkunan avaamiseen tehdä MessageBoxService-luokka, jota voidaan tarvittaessa kutsua avaamaan MessageBox-ikkuna. Jotta MessageBoxService-luokka voidaan korvata testitilanteessa, täytyy sitä varten tehdä myös IMessageBoxService-rajapinta, jonka MessageBoxService-luokka toteuttaa. Kuvassa 19 on esitelty IMessageBoxService-rajapinta ja sen toteuttava MessageBoxService-luokka, joka osaa avata MessageBox-ikkunan.

```
public interface IMessageBoxService
{
    void Show(string message);
}

public class MessageBoxService : IMessageBoxService
{
    public void Show(string message)
    {
        MessageBox.Show(message);
    }
}
```

Kuva 19. IMessageBoxService-rajapinta ja sen toteuttava MessageBoxService-luokka.

Kun MessageBoxService-luokka oli tehty pystyin muuttamaan näkymämallin käyttämään MessageBox-ikkunan avaamiseen MessageBoxService-luokasta muodostettua oliota. Seuraavana ongelmana oli, että prototyyppisovelluksen näkymämallin eli PrototyyppiSovellusViewModel-luokan konstruktori käytti näkymämallin luomiseen YritysRekisteri-luokkaa sekä MessageBoxService-luokkaa.

Tämä on yksikkötestauksen näkökulmasta ongelma, koska kun näkymämalli-luokasta luodaan testissä ilmentymä, käyttää konstruktori sen luomiseen ko. luokkia, jotka aiheuttavat riippuvuuden niihin. Niinpä näkymämalliin eli PrototyyppiSovellusViewModel-luokkaan täytyi tehdä toinen konstruktori, jota on mahdollista käyttää testeissä niin, että nuo riippuvuudet voidaan korvata.

Konstruktori ottaa parametreina ITuoterekisteri-rajapinnan toteuttavan luokan sekä IMessageBoxService-rajapinnan toteuttavan luokan. Sovelluksen oletus konstruktori on kuvassa 20 ylempänä näkyvä konstruktori, jota käytetään kun sovellusta ajetaan normaalissa käyttötilanteessa. Oletus konstruktori käyttää myös uutta, kuvassa 20 alempana näkyvää konstruktoria this-osoittimen avulla ja antaa sille parametreina TuoteRekisteri- ja MessageBoxService-luokat.

```
public PrototyyppiSovellusViewModel():this(new TuoteRekisteri(), new MessageBoxService())
{
}

public PrototyyppiSovellusViewModel(ITuoteRekisteri tuoteRekisteri, IMessageBoxService messageBoxService)
{
    this.tuoteRekisteri = tuoteRekisteri;
    this.messageBoxService = messageBoxService;
    InitializeData();
}
```

Kuva 20. PrototyyppiSovellusViewModel-luokan eli näkymämallin konstruktorit.

Nyt näkymämalli-olion voi testitilanteessa muodostaa käyttämällä ITuoteRekisteri-rajapinnan ja IMessageBoxService-rajapinnan toteuttavia mock-olioita, joiden metodit voidaan ylikirjoittaa tarvittaessa halutulla tavalla.

Ensimmäisenä metodina näkymämallissa on konstruktorissakin (kuva 20) käytetty InitializeData-metodi, joka on esitelty kuvassa 21. Metodi alustaa nimensä mukaisesti näkymämallissa käytetyn datan, jos tietokantaan on yhteys. Jos tietokantaan ei ole yhteyttä, metodi ilmoittaa siitä MessageBox-ikkunalla ja sulkee ohjelman.

```
private void InitializeData()
{
    if (tuoteRekisteri.PaivitaTuotteetTietokannasta())
    {
        Tuotteet = new ObservableCollection<Tuote>(tuoteRekisteri.GetTuoteLista());
        savedTuoteLista = Tuotteet.Select(x => x.GetCopy()).ToList();
    }
    else
    {
        messageBoxService.Show("Ei yhteyttä tietokantaan. Ohjelma suljetaan.");
        if (Application.Current != null)
        {
            Application.Current.Shutdown();
        }
    }
}
```

Kuva 21. Näkymämallin InitializeData-metodi.

Tälle metodille ei voi tehdä yksikkötestejä, koska sen näkyvyys ei ole julkinen, mutta sitä on syytä tarkastella, koska sitä käytetään näkymämallin konstruktorissa. Siitä voidaan päätellä, että kun testeissä TuoteRekisteri-luokka mockataan, täytyy ainakin sen PaivitaTuotteetTietokannasta-metodi yli kirjoittaa palauttamaan haluttu tulos tai muuten näkymämallin metodeille ei voida kirjoittaa testejä, koska InitializeData-metodi suorittaa else-haaran ja testattavissa metodeissa tarvittavaa dataa ei ole alustettu.

Seuraavana päätin tehdä näkymämallin testiluokkaan näkymämallin alustamiseen käytettävän metodin, jota voidaan käyttää jokaisessa testimetodissa niin, että jokaisessa niissä ei tarvitse erikseen kirjoittaa samaa koodia vaan voidaan pelkästään kutsua kuvassa 22 näkyvää InitializeViewModel-metodia.

```
public PrototyyppiSovellusViewModel ViewModel;
public Mock<ITuoteRekisteri> TuoterekisteriMock = new Mock<ITuoteRekisteri>();
public Mock<IMessageBoxService> MessageBoxServiceMock = new Mock<IMessageBoxService>();

public void InitializeViewModel()
{
    TuoterekisteriMock.Setup(x => x.PaivitaTuotteetTietokannasta()).Returns(true);
    var tuoteLista = new List<Tuote>
    {
        new Tuote {Id = 1, Nimi = "Saha", Hinta = 19.90},
        new Tuote {Id = 2, Nimi = "Kirves", Hinta = 12.90}
    };
    TuoterekisteriMock.Setup(x => x.GetTuoteLista()).Returns(new List<Tuote>(tuoteLista));
    ViewModel = new PrototyyppiSovellusViewModel(TuoterekisteriMock.Object, MessageBoxServiceMock.Object);
}
```

Kuva 22. Testeissä käytetty InitializeViewModel-metodi, joka alustaa näkymämallin.

Testiluokassa luodaan ensin Moq-ohjelmistokehyksen avulla ITuoterekisteri- ja IMessageBoxService-rajapinnan toteuttavat mock-oliot ja tämän jälkeen InitializeViewModel-metodissa ylikirjoitetaan TuoteRekisteriMock-olion PaivitaTuotteetTietokannasta- ja GetTuoteLista-metodit. Käytettävien metodien yli kirjoitus tapahtuu käyttämällä mock-olion Setup- ja Returns-metodeja. Kuvassa 22 on PaivitaTuotteetTietokannasta()-metodi ylikirjoitettu palauttamaan tosi-totuusarvo sekä GetTuoteLista-metodi on ylikirjoitettu palauttamaan tuoteLista-listan, jossa on kaksi tuotetta.

Seuraavaksi olikin näkymämallin metodeille mahdollista kirjoittaa omat yksikkötestit. Ensimmäisenä tein yksikkötestit näkymämallin CanLisaaTuote- ja LisaaTuote-metodeille. CanLisaaTuote-metodi palauttaa boolean totuusarvon onko lisättävälle tuotteelle annettu nimi ja hinta. CanLisaaTuote-metodin avulla käyttöliittymän lisäyspainike poistetaan käytöstä, mikäli lisättävälle tuotteelle ei ole annettu nimeä ja hintaa.

LisaaTuote-metodi lisää tuotteen Tuotteet-kokoelmaan, mikäli sen hinta on kelvollinen. Jos lisättävän tuotteen hinta ei ole kelvollinen, ei sitä lisätä Tuote-kokoelmaan, vaan ilmoitetaan käyttäjälle, että tuotteen hinta ei ole kelvollinen. LisaaTuote-metodissa käytetään aiemmin esittelemiäni HintaHelper-luokan metodeja hinnan tarkistuksessa.

Periaatteessa tämä hieman rikkoo yksikkötestauksen sääntöjä, koska testattavassa yksikössä käytetään toisen luokan metodeja, mutta tässä tapauksessa päätin sallia sen, koska en halunnut alkaa luomaan uutta rajapintaa HintaHelper-luokasta ja mockamaan myös HintaHelper-luokkaa, koska sille oli jo olemassa omat yksikkötestit ja kyseessä on todella yksinkertainen luokka. CanLisaaTuote- ja LisaaTuote-metodeille tein kuvan 23 mukaiset testimetodit.

CanLisaaTuote-metodin testeissä (kuva 23) ensimmäisessä testataan tyhjällä tuotteella odotettua epätosi-arvoa ja toisessa testissä tosi-arvoa antamalla lisättävälle tuotteelle nimi ja hinta. LisaaTuote-metodin testeissä (kuva 23) koitetaan ensiksi lisätä kelvotonta tuotetta ja sen jälkeen lisätään kelvollinen tuote. Odotettu

tulos ja varsinainen tulos saadaan tarkistettua hyödyntämällä Tuotteet-kokoelman Count-metodia, joka palauttaa tuotteiden määrän kokoelmassa.

```
[TestMethod]
public void CanLisaaTuote_TestEpatosiEiNimeaEiHintaa()
{
    InitializeViewModel();
    ViewModel.TuoteNimi = "";
    ViewModel.TuoteHinta = "";
    var tulos = ViewModel.CanLisaaTuote();
    Assert.IsFalse(tulos);
}

[TestMethod]
public void CanLisaaTuote_TestTosiNimiJaHintaAnnettu()
{
    InitializeViewModel();
    ViewModel.TuoteNimi = "Vasara";
    ViewModel.TuoteHinta = "12,60";
    var tulos = ViewModel.CanLisaaTuote();
    Assert.IsTrue(tulos);
}

[TestMethod]
public void LisaaTuote_TestEiKelvollinenTuoteEiLisata()
{
    InitializeViewModel();
    ViewModel.TuoteNimi = "Sorkkarauta";
    ViewModel.TuoteHinta = "tässäpä onkin tekstiä";
    var odotettuTulos = ViewModel.Tuotteet.Count;
    ViewModel.LisaaTuote();
    var tulos = ViewModel.Tuotteet.Count;
    Assert.AreEqual(odotettuTulos, tulos);
}

[TestMethod]
public void LisaaTuote_TestKelvollinenTuoteLisataan()
{
    InitializeViewModel();
    ViewModel.TuoteNimi = "Sorkkarauta";
    ViewModel.TuoteHinta = "20.90";
    var odotettuTulos = ViewModel.Tuotteet.Count + 1;
    ViewModel.LisaaTuote();
    var tulos = ViewModel.Tuotteet.Count;
    Assert.AreEqual(odotettuTulos, tulos);
}
```

Kuva 23. Näkymämallin CanLisaaTuote- ja LisaaTuote-metodien testimetodit.

Seuraavana tein testit näkymämallin CanPoistaTuote- ja PoistaTuote-metodeille (kuva 24). Nämä metodit ovat yksinkertaisemmat, kuin tuotteen lisäykseen käytetyt metodit, koska niissä ei ole muuta validointilogiikkaa kuin, että poistettava tuote on valittuna.

```
public bool CanPoistaTuote()
{
    return SelectedTuote != null;
}

public void PoistaTuote()
{
    Tuotteet.Remove(SelectedTuote);
}
```

Kuva 24. Näkymämallin CanPoistaTuote- ja PoistaTuote-metodit.

CanPoistaTuote- ja PoistaTuote-metodeille tein kuvan 25 mukaiset testimetodit. CanPoistaTuote-metodin ensimmäisessä testissä ei ole valittu poistettavaa tuotetta, joten siinä odotetaan epätosi-arvoa ja toisessa testissä poistettava tuote on valittu, joten siinä odotetaan tosi-arvoa. PoistaTuote-metodin testeissä (Kuva 25) ensimmäisessä ajetaan metodi ilman, että poistettavaa tuotetta on valittu, joten odotetussa tuloksessa ei mitään tuotetta ole poistettu Tuotteet-kokoelmasta. Toisessa testimetodissa valituksi tuotteeksi asetetaan ensin Tuotteet-kokoelman ensimmäinen tuote ja poistetaan se. Odotetussa tuloksessa Tuotteet-kokoelmassa on siis yksi tuote vähemmän kuin alkuperäisessä kokoelmassa.

```
[TestMethod]
public void CanPoistaTuote_TestEiValittuaTuotettaEpatosi()
{
    InitializeViewModel();
    var tulos = ViewModel.CanPoistaTuote();
    Assert.IsFalse(tulos);
}

[TestMethod]
public void CanPoistaTuote_TestPoistettavaTuoteValittuTosi()
{
    InitializeViewModel();
    ViewModel.SelectedTuote = ViewModel.Tuotteet[0];
    var tulos = ViewModel.CanPoistaTuote();
    Assert.IsTrue(tulos);
}

[TestMethod]
public void PoistaTuote_TestEiValittuaTuotettaEiPoistoa()
{
    InitializeViewModel();
    var odotettuTulos = ViewModel.Tuotteet.Count;
    ViewModel.PoistaTuote();
    var tulos = ViewModel.Tuotteet.Count;
    Assert.AreEqual(odotettuTulos, tulos);
}

[TestMethod]
public void PoistaTuote_TestValittuTuotePoistetaan()
{
    InitializeViewModel();
    var odotettuTulos = ViewModel.Tuotteet.Count - 1;
    ViewModel.SelectedTuote = ViewModel.Tuotteet[0];
    ViewModel.PoistaTuote();
    var tulos = ViewModel.Tuotteet.Count;
    Assert.AreEqual(odotettuTulos, tulos);
}
```

Kuva 25. Näkymämallin CanPoistaTuote- ja PoistaTuote-metodien testimetodit.

Viimeisinä testattavina metodeina näkymämallissa eli PrototyyppiSovellus-ViewModel-luokassa oli kuvassa 26 esitetyt CanTallenna- ja Tallenna-metodit. CanTallenna metodi palauttaa boolean totuusarvon sen mukaan, onko Tuotteet-

kokoelma muuttunut tietokantaan tallennetusta kokoelmasta. Jos Tuotteet-kokoelmaan on tullut muutoksia metodi palauttaa tosi-arvon ja muutoin epätosi-arvon. Näkymämallin Tallenna-metodissa (kuva 26) on huomioitava, että siinä käytetään TuoteRekisteri-luokan Tallenna-metodia, joka palauttaa boolean totuusarvon sen mukaan, onko tallennus tietokantaan onnistunut.

```
public bool CanTallenna()
{
    var nykyinenTuoteLista = Tuotteet.ToList();

    return
        nykyinenTuoteLista.Except(tallennettuTuoteLista)
            .Union(tallennettuTuoteLista.Except(nykyinenTuoteLista))
            .Any();
}

public void Tallenna()
{
    if (tuoteRekisteri.Tallenna(Tuotteet.ToList()))
    {
        Tuotteet = new ObservableCollection<Tuote>(tuoteRekisteri.GetTuoteLista());
        tallennettuTuoteLista = tuoteRekisteri.GetTuoteLista().Select(x => x.GetCopy()).ToList();
        messageBoxService.Show("Tuotteet tallennettu tietokantaan.");
    }
    else
    {
        messageBoxService.Show("Tuotteiden tallennus tietokantaan ei onnistunut.");
    }
}
```

Kuva 26. Näkymämallin CanTallenna- ja Tallenna-metodit.

Tallenna-metodin testeissä on siis mockattava TuoteRekisteri-luokan Tallenna-metodi palauttamaan testitapauksessa haluttu arvo. TuoteRekisteri-luokan Tallenna-metodi ottaa parametrikseen Tuote-listan. Näkymämalli ei sisällä tietokantaan tallennus logiikkaa ollenkaan vaan TuoteRekisteri-luokan Tallenna-metodi hoitaa kaiken sen. Näkymämallin Tallenna-metodi vain ilmoittaa MessageBox-ikkunalla onnistuiko tietokantaan tallentaminen vai ei. Niimpä näkymämallin Tallenna-metodille ei tarvitse, eikä voikaan tehdä kovin monimutkaisia testejä, vaan siihen riittää kuvan 27 mukaiset testit, joissa testataan vain, että metodi suoritetaan loppuun, palauttipa TuoteRekisteri-luokan Tallenna-metodi kumman tahansa totuusarvon.

```
[TestMethod]
public void Tallenna_TestTallennusOnnistui()
{
    TuoterekisteriMock.Setup(x => x.Tallenna(new List<Tuote>())).Returns(true);
    InitializeViewModel();
    ViewModel.Tallenna();
    Assert.IsTrue(true);
}

[TestMethod]
public void Tallenna_TestTallennusEpaOnnistui()
{
    TuoterekisteriMock.Setup(x => x.Tallenna(new List<Tuote>())).Returns(false);
    InitializeViewModel();
    ViewModel.Tallenna();
    Assert.IsTrue(true);
}
```

Kuva 27. Näkymämallin Tallenna-metodin testimetodit.

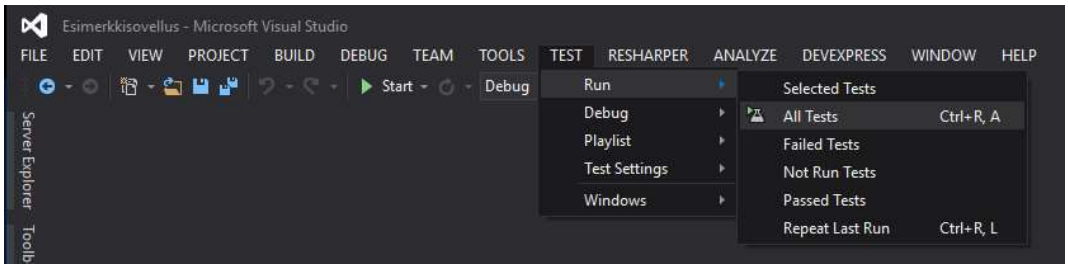
CanTallenna-metodille tein kuvan 28 mukaiset testimetodit. CanTallenna-metodin ensimmäisessä testissä ei Tuotteet-kokoelmaan tehdä muutoksia, joten siinä odotetaan tuloksena epätosi-arvoa. Toisessa, kolmannessa ja neljännessä testissä Tuote-kokoelmaan tehdään muutos ja odotetaan tuloksena tosi-arvoa.

```
[TestMethod]
public void CanTallenna_TestEiMuutoksiaEpatosi()
{
    InitializeViewModel();
    var tulos = ViewModel.CanTallenna();
    Assert.IsFalse(tulos);
}

[TestMethod]
public void CanTallenna_TestTuotteessaMuutoksiaTosi()
{
    InitializeViewModel();
    ViewModel.Tuotteet[0].Nimi = "Muutettu nimi";
    var tulos = ViewModel.CanTallenna();
    Assert.IsTrue(tulos);
}

[TestMethod]
public void CanTallenna_TestLisattyTuoteTosi()
{
    InitializeViewModel();
    ViewModel.Tuotteet.Add(new Tuote());
    var tulos = ViewModel.CanTallenna();
    Assert.IsTrue(tulos);
}

[TestMethod]
public void CanTallenna_TestPoistettuTuoteTosi()
{
    InitializeViewModel();
    ViewModel.Tuotteet.Remove(ViewModel.Tuotteet[0]);
    var tulos = ViewModel.CanTallenna();
    Assert.IsTrue(tulos);
}
```

Kuva 28. CanTallenna-metodin testimetodit.

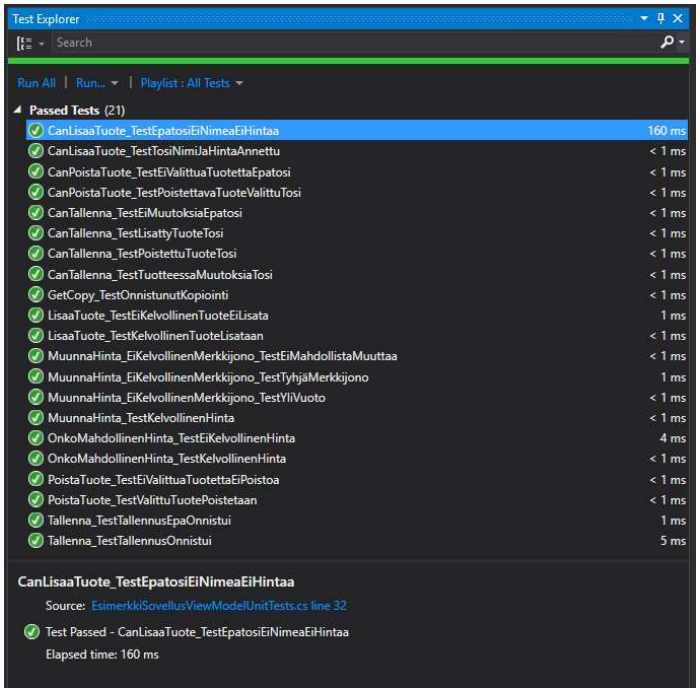
4.6 Testien ajaminen MS Unit:lla

Kun halutut yksikkötestit on tehty, voidaan ne ajaa Visual Studio-kehitysympäristössä kaikki kerralla tai yksitellen. Kaikkien testien ajaminen tapahtuu esimerkiksi kuvan 29 mukaisesti, valitsemalla Visual Studion ylävalikosta Test-kohta ja sen alivalikosta Run All Tests.



Kuva 29. Kaikkien testien ajaminen kerralla Visual Studio-kehitysympäristössä.

Testien ajaminen aukaisee Visual Studio:n Test Explorer-ikkunan (Kuva 30), josta testien suoritusta voi tarkastella. Kuvassa 30 on Test Explorer-ikkuna, jossa on suoritettuna kaikki prototyyppisovellukselle tekemäni yksikkötestit. Yksikkötestejä tuli kirjoitettua yhteensä, jopa 21 kappaletta, vaikka kyseessä onkin kohtuullisen yksinkertainen sovellus.



Kuva 30. Visual Studio:n Test Explorer-ikkuna.

5 PÄÄTÄNTÖ

Työn tavoitteena oli tutkia, miten MVVM-mallin mukaiselle WPF-sovellukselle on mahdollista toteuttaa yksikkötestejä ja toimiva testiympäristö. Käytännön työssä tarkoituksena oli luoda prototyyppisovellukselle asianmukaiset yksikkötestit. Mielestäni onnistuin saavuttamaan asetetut tavoitteet hyvin, sekä työstä on hyötyä toimeksiantajalle tulevaisuudessa, kun testauksen roolia projekteissa tullaan kehittämään.

Opinnäytetyön tekeminen osoitti, että yksikkötestien tekeminen MVVM-mallin mukaiselle WPF-sovellukselle, ainakin kokonaiselle näkymämallille voi olla työlästä ja haastavaa, koska sen testattavat yksiköt sisältävät usein paljon riippuvaisuuksia muihin yksiköihin. Vaikka riippuvaisuuksien korvaamiseksi eli datan mockaamisen helpottamiseksi löytyykin ohjelmistokehyksiä, kuten työssä käytetty Moq, ei se siltikään ole usein helppoa, koska korvattavasta riippuvuudesta joudutaan yleensä muodostamaan oma rajapinta. Tästä johtuen kannattaakin harkita, onko kokonaiselle näkymämallille kannattavaa tehdä yksikkötestejä, jos testien teko aiheuttaa merkittävästi päänvaivaa, sekä tekemiseen saattaa kulua hyvinkin paljon aikaa. Tässä voisikin olla yksi työn mahdollinen jatkokehitysmahdollisuus, jossa voitaisiin pyrkiä ratkaisemaan kyseinen ongelma. Pari muuta jatkokehitysmahdollisuutta voisi olla myös, miten ohjelman tietokantakerrosta ja käyttöliittymää voitaisiin testata omalla tavallaan ja minkälaisia ominaisuuksia eri ohjelmistokehykset tarjoavat näihin.

Sen sijaan työ osoitti, että pienemmille sovelluksessa käytettäville, yleiskäyttöisille luokille, yksikkötestien tekeminen on nopeaa ja suhteellisen vaivatonta, jos riippuvaisuuksia toisiin yksiköihin ei ole. Tällöin yksikkötestien tekeminen on varmasti erittäinkin hyödyllistä, koska testitapauksia ei tarvitse aina suorittaa manuaalisesti, sekä testattava luokka voi olla hyvinkin tärkeä sovelluksen oikeanlaisen toiminnan kannalta ja sitä saatetaan käyttää monessa eri paikassa.

Käytännön työtä tehdessä huomasin myös, että tehtiinpä yksikkötestejä kokonaiselle näkymämallille tai minkälaiselle luokalle tahansa, joka tapauksessa se parantaa ainakin ohjelmakoodin laadukkuutta varsinkin, jos testattavuutta ajatellaan

jo ohjelman tekovaiheessa. Tästä johtuen jatkokehitysmahdollisuutena voisi lähteä myös tutkimaan testivetoisen kehityksen mahdollisuuksia tässä ympäristössä.

Itse opin opinnäytetyöprosessissa paljon uutta, koska suurin osa käytettävistä tekniikoista ei ollut itselleni entuudestaan tuttuja. Yksikkötestauksesta minulla ei ollut minkäänlaista käytännöntason kokemusta ennen opinnäytetyötä. Aloittelevana ohjelmistokehittäjänä yksikkötestaukseen perehtymisestä oli minulle hyötyä myös tulevaisuudessa, laadukkaamman ohjelmakoodin kirjoittamista ajatellen.

Välillä työtä tehdessä mieleeni tuli, että prototyyppisovelluksena olisi voinut toimia jokin hieman monimutkaisempi ja laajempi sovellus, mutta kun ottaa huomioon lopputuloksen, se tuskin olisi tuonut mitään lisäarvoa ja se olisi vain monimutkaistanut asioita turhaan. Lisäksi tiukasta aikataulusta johtuen olisi ollut ikävää, jos monimutkaisemman prototyyppisovelluksen takia olisi jäänyt jotakin toteuttamatta.

Haasteellisintahan työssä olikin kohtuullisen tiukka aikataulu, jonka aikarajat tulivat välillä yllättävänkin nopeasti vastaan, varsinkin kun tein opinnäytetyötä harjoittelun ohessa ja tekemiseen tuli välillä pitkiäkin taukoja. Toisaalta harjoittelun aikana minulle annettiin tarvittaessa aikaa ja apua opinnäytetyön tekemiseen kiitettävästi, joten pystyin pitämään hyvin kiinni kaikista asetetuista aikatauluista ilman suurempaa stressiä. Opinnäytetyöstä jäi minulle kaikin puolin erittäin positiivinen kokemus, josta on varmasti hyötyä tulevaisuutta ajatellen.

LÄHTEET

Beck, K. 2002. Test driven development: By example. 1. painos. Boston: Pearson Education.

Chaffee, A & Pietri, W. 2002. Unit testing with mock objects. WWW-dokumentti. Saatavissa: https://www.ibm.com/developerworks/library/j-mocktest [viitattu 18.2.2017]

Edwards, B. 2013. MVVM - Writing a Testable Presentation Layer with MVVM. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/magazine/dn463790.aspx [viitattu 21.3.2017]

Fowler, M & Highsmith, J. 2001. The Agile Manifesto. WWW-dokumentti. Saatavissa: http://dimsboiv.uqac.ca/8INF851/web/part1/introduction/The_Agile_Manifesto.pdf [viitattu 16.2.2017]

Garofalo, R. 2011. Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern. 1. painos. California: O'Reilly Media.

Getting Started with DevExpress MVVM Framework. 2013. DevExpress. WWW-dokumentti. Saatavissa: https://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx [viitattu 17.2.2017]

Haikala, I. & Märijärvi, J. 2004. Ohjelmistotuotanto. 1. painos. Helsinki: Talentum

Introduction to the C# Language and the .NET Framework. 2015. Microsoft Developer Network. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx [viitattu 13.2.2017]

MacDonald, M. 2012. Pro WPF in C#. 4. painos. New York: Apress.

Myers, G. 2012. The art of software testing. 3. painos. New Jersey: John Wiley & Sons.

Osherove, R. 2013. The Art of Unit Testing. 2. painos. New York: Manning Publications.

Patton, R. 2001. Software testing. 2. painos. Indianapolis: Sams Publishing.

Smith, J. 2009. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx [viitattu 16.2.2017]

The MVVM Pattern. 2012. Microsoft Developer Network. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/hh848246.aspx [viitattu 16.2.2017]

Tietojenkäsittelytieteen laitos. 2009. Ohjelmistoprosessit ja ohjelmistojen laatu. WWW-dokumentti. Saatavissa: https://www.cs.helsinki.fi/u/taina/opol/k-2009/pdf/luku-6_2.pdf [viitattu 18.2.2017]

Tolvanen, P. 2013. Ketterän kehityksen yleiset periaatteet. WWW-dokumentti. Saatavissa: https://www.meteoriitti.com/2013/06/06/ketteryys-haltuun-ketteran-kehityksen-yleiset-periaatteet [viitattu 16.2.2017]

Vice, R & Siddiqi, M S. 2012. MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF. 1. painos. Birmingham: Packt Publishing.

What is XAML. 2017. Microsoft Developer Network. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/cc295302.aspx [viitattu 13.2.2017]

Williams, L. 2006. Testing Overview and Black-Box Testing Techniques. WWW-dokumentti. Saatavissa: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf [viitattu 3.2.2017]

Williams, L. 2006. White-Box Testing. WWW-dokumentti. Saatavissa: http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf [viitattu 3.2.2017]

Yosifovich, P. 2012. Windows Presentation Foundation 4.5 Cookbook. 1. painos. Birginham: Packt Publishing.