Juha Ala-Rantala

MVVM-mallia noudattava WPF-sovellus lokitiedostojen tarkasteluun

Opinnäytetyö
Kevät 2019
SeAMK Tekniikka
Tietotekniikan tutkinto-ohjelma

SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Tutkinto-ohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Juha Ala-Rantala

Työn nimi: MVVM-mallin WPF-sovellus lokitiedostojen tarkasteluun

Ohjaaja: Petteri Mäkelä

Vuosi: 2019 Sivumäärä: 101

Opinnäytetyön tehtiin Seinäjoella toimivalle metalliteollisuuden yritykselle Finn-Power Oy:lle. Yritys valmistaa räätälöityjä ohutlevyjen työstöön suunniteltuja automatisoituja laite- ja järjestelmäratkaisuja sekä ohjelmistoja niiden ohjaamiseen. Yrityksen laitteet ja ohjelmat tuottavat paljon lokitiedostoja, jotka ovat oleellisia ongelmien ratkomisessa. Opinnäytetyön pääaiheena on suunnitella ja kehittää kevyt ja tehokas sovellus näiden lokitiedostojen tarkasteluun.

Opinnäytetyössä pääaiheina olivat MVVM-arkkitehtuurimalli ja WPF-käyttöliittymäkirjasto. Toisena pääaiheena oli ohjelman suunnittelu ja toteutus hyödyntäen näitä kahta tekniikkaa. Työssä perehdyttiin MVVM-mallin historiaan, rakenteeseen, etuihin ja seuraamuksiin. WPF-kirjaston ominaisuudet, komponentit ja MVVM-mallin tuki käytiin läpi. Työssä luotiin ohjelmalle vaatimusmäärittely, käytiin läpi ohjelman arkkitehtuuri, käytetyt työkalut ja itse sovelluksen toteutus.

Työn lopputuloksena saatiin aikaan yksinkertainen ja toimiva sovellus. Sovellus täyttää sille asetetut minimitavoitteet, mutta sovellusta voisi kehittää eteenpäin. Työn lopussa esitellään mahdollisia jatkokehitys ideoita.

Avainsanat: C#, MVVM, WPF, XAML

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Programming

Author: Juha Ala-Rantala

Title of thesis: WPF-application with an MVVM-pattern for Log file viewing

Supervisor(s): Petteri Mäkelä

Year: 2019 Number of pages: 101

This thesis was made for Finn-Power Oy in Seinäjoki. The company specializes in manufacturing custom laser and sheet metal machinery. The company also designs and develops software for controlling the machinery. Many of the machines and software produce log data, which is essential for solving customer's problems. The purpose of this thesis was to develop an application for viewing the log data.

The main topics of this thesis were the MVVM-pattern and the WPF-graphical subsystem. The other topics were designing and developing the actual application using the aforementioned technologies. The thesis studied the history, architecture, advantages and consequences of the MVVM-pattern. The features, components and the MVVM-pattern support of the WPF-technology were examined. In the thesis the software requirement specification was defined for the application. The thesis also studied the design, third party libraries and the development of the application.

The final result of the thesis was a simple and functional application. The application met the set goals, though the application can be improved in the future. At the end of the thesis a couple of ideas for further development were presented.

Keywords: C#, MVVM, WPF, XAML

SISÄLTÖ

Opinnäytetyön tiivistelmä....................1
Thesis abstract....................2
SISÄLTÖ....................3
Kuva- ja kuvioluettelo....................5
Käytetyt termit ja lyhenteet....................11
1 JOHDANTO....................13
1.1 Työn tausta....................13
1.2 Työn tavoite....................13
1.3 Työn rakenne....................14
1.4 Finn-Power Oy....................14
2 MVVM....................16
2.1 Historia....................16
2.2 Rakenne ja toimintaperiaate....................17
2.2.1 Näkymä....................19
2.2.2 Näkymämalli....................20
2.2.3 Malli....................20
2.3 Näkymien ja näkymämallien kokoonpanot....................20
2.4 MVVM-mallin mukainen sovellus....................22
2.5 MVVM-mallin edut ja seuraamukset....................22
2.5.1 Edut....................23
2.5.2 Seuraamukset....................23
3 WPF....................24
3.1 Arkkitehtuuri....................25
3.2 XAML....................29
3.3 Looginen ja visuaalinen puu....................32
3.4 Reititetyt tapahtumat....................33
3.5 Tiedon sidonta....................35
3.5.1 Tiedon muuntaminen....................38
3.6 Komennot....................40
3.7 Tyylit....................42

3.8 Laukaisimet 43
3.9 Ulkoasumallit 45
3.10 MVVM ja WPF 47
3.10.1 Näkymän yhdistäminen näkymämalliin 48
3.10.2 Näkymän ja näkymämallin kommunikointi 49
3.11 WPF ja UWP 52
4 VAATIMUSMÄÄRITTELY 54
5 SOVELLUKSEN SUUNNITTELU JA ARKKITEHTUURI 55
5.1 MVVM Light Toolkit 55
5.1.1 Ominaisuudet 55
5.1.2 Komennot 57
5.2 AdonisUI 59
5.3 Fody ja Costura 60
5.4 Sovelluksen arkkitehtuuri 61
6 TOTEUTUS 63
6.1 Malli 64
6.2 Pääikkuna 70
6.2.1 Näkymä 70
6.2.2 Näkymämalli 73
6.3 Aloitus 76
6.3.1 Näkymä 78
6.3.2 Näkymämalli 81
6.4 Latausnäkymä 83
6.4.1 Näkymä 83
6.4.2 Näkymämalli 85
6.5 Päänäkymä 86
6.5.1 Näkymä 88
6.5.2 Näkymämalli 93
7 YHTEENVETO 98
7.1 Jatkokehitys 98
7.2 Pohdinta 99
LÄHTEET 100

Kuva- ja kuvioluettelo

Kuva 1. Yksinkertainen XAML-tiedosto 29

Kuva 2. Näkymän luonti ohjelmallisesti 30

Kuva 3. Luotu näkymä 31

Kuva 4. MouseDown- ja PreviewMouseDown-tapahtumat 34

Kuva 5. Tiedon sitominen XAML-kielessä 36

Kuva 6. Tiedon sitomisen asetuksia 38

Kuva 7. Color-luokasta Brush-luokkaan muunnin 39

Kuva 8. Muuntajan nimen ja sijainnin määrittely 40

Kuva 9. TextBlock-kontrollin taustavärin määrittely muuntimella 40

Kuva 10. Komennon yhdistäminen ominaisuuksiin tiedon sidonnalla 42

Kuva 11. TextBox-kontrollille kohdistettu tyyli 42

Kuva 12. Yleinen tyyli 43

Kuva 13. Eri kontrolleille asetettuja tyylejä 43

Kuva 14. Yksinkertainen ominaisuuslaukaisin 44

Kuva 15. Moniominaisuuslaukaisin 45

Kuva 16. Lista olioista ilman tietomallia 46

Kuva 17. Listalle tietomallin määrittely 46

Kuva 18. Lista olioista mukautetulla tietomallilla 47

Kuva 19. Näkymälle näkymämallin määrittely XAML-kielessä 48

Kuva 20. Näkymämallin määrittely näkymälle taustakoodissa 48

Kuva 21. Näkymämalli määrittely tieto sidonnalla 49

Kuva 22. Tekstilohkon ja painikkeen sisältävä näkymä 49

Kuva 23. Näkymämallin käyttämä ilmoitusmetodi 49

Kuva 24. Status-ominaisuuden määrittely 50

Kuva 25. Komennon logiikka 50

Kuva 26. Komennon toteutus 51

Kuva 27. Komento ominaisuuden luonti 51

Kuva 28. Komennon alustaminen 51

Kuva 29. Luotu ohjelma 52

Kuva 30. Ominaisuus näkymämallissa ilman MVVM Light Toolkit käyttöä 56

Kuva 31. Ominaisuuden toteutus MVVM Light Toolkit avulla 56

Kuva 32. Komento ja komentoa hyödyntävä näkymämalli 58

Kuva 33. Komento RelayCommand-luokkaa hyödyntämällä 58

Kuva 34. EventToCommand-luokan määrittely 59

Kuva 35. Esimerkkilokitiedosto puolipisteillä 63

Kuva 36. Esimerkkilokitiedosto sarkainmerkeillä 63

Kuva 37. Tiedon varastointi olioluokka 64

Kuva 38. Mallin rajapintaluokka 64

Kuva 39. TotalFilesCount-metodi 65

Kuva 40. LogRowsFromSource-metodi 65

Kuva 41. Tiedoston rakenteen evaluointi metodi 66

Kuva 42. Log-tiedostosta luku 67

Kuva 43. Zip-tiedostosta luku 68

Kuva 44. Lokirivien käsittelymetodit 69

Kuva 45. Päivänmäärän jäsennysmetodi 69

Kuva 46. Pääikkunan attribuutit 70

Kuva 47. Ikkunan tyyli 70

Kuva 48. Pääikkunan näkymämallin määrittely 71

Kuva 49. Pääikkunan muuntimien määrittely 71

Kuva 50. Muuntimen toteutus 71

Kuva 51. Näkymät pääikkunassa 72

Kuva 52. Näkymien tyyli 73

Kuva 53. Pääikkunanäkymämallin ominaisuudet 74

Kuva 54. Päänäkymämallin rakentaja 74

Kuva 55. Lokien lukumetodi 75

Kuva 56. Latausnäkymän päivittäminen 75

Kuva 57. Mallin kanssa kommunikointi 75

Kuva 58. Loppukäsittely 76

Kuva 59. Aloitusnäkymä 77

Kuva 60. Aloitusnäkymä animaatio 77

Kuva 61. Vedä ja pudota animaatio 78

Kuva 62. Näkymän attribuuttien määrittely 78

Kuva 63. Vedä ja pudota määrittely ... 79

Kuva 64. Muuntimen määrittely ... 79

Kuva 65. Napautettavan alueen määrittely ... 80

Kuva 66. Vedä ja pudota näkymä ... 80

Kuva 67. Aloitusnäkymämallin rakentaja ... 81

Kuva 68. Aloitusnäkymämallin ominaisuudet ... 81

Kuva 69. Tiedostojen valintametodi ... 82

Kuva 70. Vedä ja pudota käsittelymetodi ... 82

Kuva 71. Latausnäkymä käyttöliittymä ... 83

Kuva 72. Latausnäkymän attribuutit ... 83

Kuva 73. Latausnäkymä määrittely ... 84

Kuva 74. Edistymispalkin animointiluokka ... 84

Kuva 75. Latausnäkymämallin ominaisuudet ... 85

Kuva 76. Latausnäkymämallin rakentaja ... 85

Kuva 77. Lokitiedostojen lukemisen keskeyttäminen ... 86

Kuva 78. Sovelluksen päänäkymä ... 86

Kuva 79. Suodattimet ... 87

Kuva 80. Suodatettu näkymä ... 87

Kuva 81. Valittu hakutulos ... 87

Kuva 82. Näkymän attribuuttien määrittely ... 88

Kuva 83. Taustavärimuunnin ... 89

Kuva 84. Näkyvyysmuunnin 89

Kuva 85. Suodatin 90

Kuva 86. Haku 90

Kuva 87. Listanäkymä 91

Kuva 88. SelectionChanged-tapahtumankäsittelijä 91

Kuva 89. Listanäkymän rivin tyyli 92

Kuva 90. Listanäkymän kolumnit 92

Kuva 91. Listanäkymän kolumnin tietomalli 93

Kuva 92. Päänäkymämallin ominaisuudet 93

Kuva 93. Päänäkymämallin rakentaja 94

Kuva 94. Suodatinominaisuudet 94

Kuva 95. Suodattimien luonti 94

Kuva 96. Komennot 95

Kuva 97. Suodattimien nollaus 95

Kuva 98. Hakumetodi 96

Kuva 99. Haun suorittamisehto 97

Kuvio 1. Application Model -mallin mukainen sovellus 17

Kuvio 2. MVVM-mallin rakenne 18

Kuvio 3. MVVM-mallia noudattava sovellus 18

Kuvio 4. Yksi näkymä ja näkymämalli 21

Kuvio 5. Näkymämallilla useampi näkymä ... 21

Kuvio 6. Näkymällä useampi näkymämalli ... 21

Kuvio 7. Esimerkki MVVM-mallin sovelluksesta ... 22

Kuvio 8. WPF-teknologian arkkitehtuuri ... 26

Kuvio 9. WPF-luokkahierarkian tärkeimmät luokat ... 27

Kuvio 10. Looginen puurakenne ... 32

Kuvio 11. Visuaalinen puurakenne ... 33

Kuvio 12. Tiedon sitomisen rakenne ... 35

Kuvio 13. Eri tiedon sitomisen muodot ... 36

Kuvio 14. Tiedon muuttumisen ilmoitustapa ... 37

Kuvio 15. Tiedon muuntaminen ... 38

Kuvio 16. Tiedon muuntaminen mukautetusti ... 39

Kuvio 17. Tapahtumien yhdistäminen käsittelijöihin ... 41

Kuvio 18. Tapahtumien yhdistäminen komentoon ... 41

Kuvio 19. Fodyn sijainti käännösprosessissa ... 60

Kuvio 20. Costura käytännössä ... 60

Kuvio 21. Luodun sovelluksen rakenne ... 61

Kuvio 22. Sovelluksen käytön kulku ... 61

Käytetyt termit ja lyhenteet

.NET Microsoftin kehittämä sovelluskehys, joka sisältää usealla eri ohjelmointikielellä toimivan luokkakirjaston.

C# Yleiskäyttöön kehitetty oliopohjainen ja vahvasti tyypitetty ohjelmointikieli.

CIL Common Intermediate Languange on kehitysympäristön tuottama käännetty ohjelmakoodi.

CLR Common Languange Runtime on CIL-kieltä suorittava ajoympäristö, joka kääntää CIL-kielen ohjelmakoodin järjestelmälle ymmärrettävään muotoon.

LINQ Language Integrated Query on .NET-komponentti, jonka avulla voi suorittaa kyselyjä olioihin ja tietokantoihin.

Silverlight Käyttöliittymäkirjasto web-pohjaisten sovelluksien kehittämiselle.

MVVM Model-View-ViewModel-arkkitehtuurimalli sovelluksien kehittämiselle.

UWP Universal Windows Platform on Microsoftin käyttöliittymäkirjasto, jonka tarkoitus on yhdistää Windows-pohjaista sovelluskehitystä.

WPF Windows Presentation Foundation -käyttöliittymäkirjasto Windows-työpöytäsovelluksien kehittämiselle.

Xamarin Alustasta riippumaton käyttöliittymäkirjasto sovelluksen kehittämiselle.

XAML Extensible Application Markup Language on XML-pohjainen merkintäkieli, jota käytetään määrittelemään käyttöliittymiä.

XML Extensible Markup Language on tarkkaa rakennetta noudattava merkintäkieli.

1 JOHDANTO

1.1 Työn tausta

Finn-Power Oy valmistaa erilaisia metallityöstökoneita, järjestelmiä ja ohjelmistoja niiden ajamiseen ja hallitsemiseen. Nämä järjestelmät ja ohjelmat tuottavat paljon dataa, jota yritys hyödyntää toiminnassaan. Kerättyyn dataan sisältyy muun muassa lokitiedostot, jotka ovat avainasemassa ongelmien ratkomisessa.

Lokitiedostojen tehokkaaseen tarkasteluun ei kuitenkaan ole ollut työkalua. Markkinoilla on tarjolla useita erilaisia ohjelmia tähän, mutta mikään ei täytä täysin kaikkia työkalulle asetettuja vaatimuksia. Yrityksen laaja kirjo erilaisia tuotteita tuottavat erilaisia lokitiedostoja erilaisissa muodoissa, mikä tarkoittaa tarvetta yrityksen sisällä suunnitellulle ohjelmalle. Näin ohjelman jokainen ominaisuus voidaan rakentaa tarpeiden mukaiseksi. Ohjelmaa voidaan myös muokata käyttäjien palautteen mukaisesti, korjata käytössä ilmenneet ongelmat nopeasti ja jatkokehittää tarvittaessa eteenpäin.

Opinnäytetyö perustuu toteutettuun projektiin ja sitä seuraamalla voi rakentaa vastaavan ohjelman.

1.2 Työn tavoite

Työn tavoitteena on suunnitella, kehittää ja toteuttaa sovellus, joka mahdollistaa lokitiedostojen tehokkaan tarkastelun. Sovelluksen tulee osata lukea ja näyttää erilaisten lokitiedostojen ja eri tiedostomuotojen sisältö. Luettuja lokitiedostoja pitää pystyä suodattamaan pois näkymästä ja takaisin niiden sisällön mukaan. Sovelluksessa tulee olla myös hakuominaisuus, joka korostaisi hakutermin osumat ja mahdollistaisi niiden välillä nopean hyppimisen. Ohjelman käyttöliittymän pitää olla moderni ja intuitiivinen käyttää. Lokitiedostojen lukunopeus täytyy optimoida mahdollisimman tehokkaaksi.

Työn toteuttamiseen annettiin täysin vapaat kädet. Ohjelman alustaksi valittiin Windows ja WPF-käyttöliittymäkirjasto. Alusta valittiin tekijän osaamisen sekä yrityksessä käytössä olevien käyttöjärjestelmien mukaan. Sovelluksen suunnittelussa ja toteutuksessa noudatettiin MVVM-arkkitehtuurimallia. MVVM-malli on hyvin ominainen WPF-sovelluksille ja se helpottaa niiden ylläpitoa sekä jatkokehitystä.

1.3 Työn rakenne

Luvussa 2 perehdytään MVVM-arkkitehtuurimalliin, jota käytettiin sovelluksen toteutuksessa.

Luvussa 3 tarkastellaan Microsoftin WPF-käyttöliittymäkirjastoa.

Luvussa 4 on sovellukselle luotu vaatimusmäärittely.

Luvussa 5 esitellään sovelluksen suunnittelu ja arkkitehtuuri.

Luvussa 6 käydään läpi sovelluksen toteuttaminen arkkitehtuurin mukaisesti, vaatimusmäärittelyä noudattaen ja läpikäytyjä tekniikoita hyödyntäen.

Luvussa 7 on opinnäytetyön tulokset, yhteenveto ja pohdinta.

1.4 Finn-Power Oy

Finn-Power Oy on metallintyöstökoneisiin erikoistunut teollisuusyritys. Yritys valmistaa räätälöityjä ohutmetallilevyjen työstöön suunniteltuja automatisoituja laite- ja järjestelmäratkaisuja sekä ohjelmistoja niiden ohjaamiseen. Finn-Power on osa italialaista Prima Industrie -konsernia ja yrityksen virallinen tunnus on Prima Power. Prima Power tunnetaan maailmalla johtavana metallintyöstökoneisiin erikoistuneena yrityksenä. (Työpaikat [Viitattu 25.1.2019].)

Finn-Power perustettiin vuonna 1969 ja yritys tuli alun perin tunnetuksi hydraulisilla letkuliitinpuristimilla. Vuonna 2008 yrityksestä tuli osa Prima Industrie -konsernia ja

vuonna 2011 tuotteiden yhteiseksi tunnukseksi otetaan käyttöön Prima Power. Yritys sijaitsi Kauhavalla suurimman osan historiastaan, mutta vuoden 2018 marraskuussa avattiin uusi päätoimipiste Seinäjoelle. (Historia [Viitattu 23.1.2019].)

Prima Powerin tuotantoyksiköitä löytyy Suomesta, Italiasta, Yhdysvalloista ja Kiinasta. Yrityksellä on asiakkaita yli 70 maasta, ja koneita sekä järjestelmiä on toimitettu yli 10000. Tuotevalikoimaan kuuluu laserleikkaus-, lävistys-, kulmaleikkuu- ja taivutustyöstökoneita. Prima Power on 3D- ja 2D-laserkoneiden johtaja ja yrityksen koneita käytetään monilla eri aloilla. Yritys pitää vahvuutenaan asiakaspalvelua ja tuotteiden automatisointia halutulle tasolle. (Yritys [Viitattu 26.1.2019].)

Prima Powerin avainperiaate on ”Next to you”, joka auttaa yritystä erottumaan kilpailevista yrityksistä. Yritys pyrkii vastaamaan ja ennakoimaan asiakkaittensa tarpeet sekä ymmärtämään asiakkaiden odotukset. Tuotteet räätälöidään vastaamaan asiakkaiden tarpeita ja tukemaan menestystä. Asiakkaiden kanssa pyritään pitkään yhteistyöhön koko tuotteen elinkaaren ajan. Asiakaspalvelua kehitetään investoimalla uusiin huoltokeskuksiin lähellä asiakkaita, ja asiakkaita voidaan myös palvella etänä. Yritys tekee yhteistyötä asiakkaiden kanssa ympäristöpäästöjen vähentämiseksi. (Next to you [Viitattu 24.1.2019].)

2 MVVM

Tässä luvussa perehdytään MVVM-malliin. Luvussa käydään läpi mallin historia, rakenne, toimintaperiaate, edut ja seuraamukset.

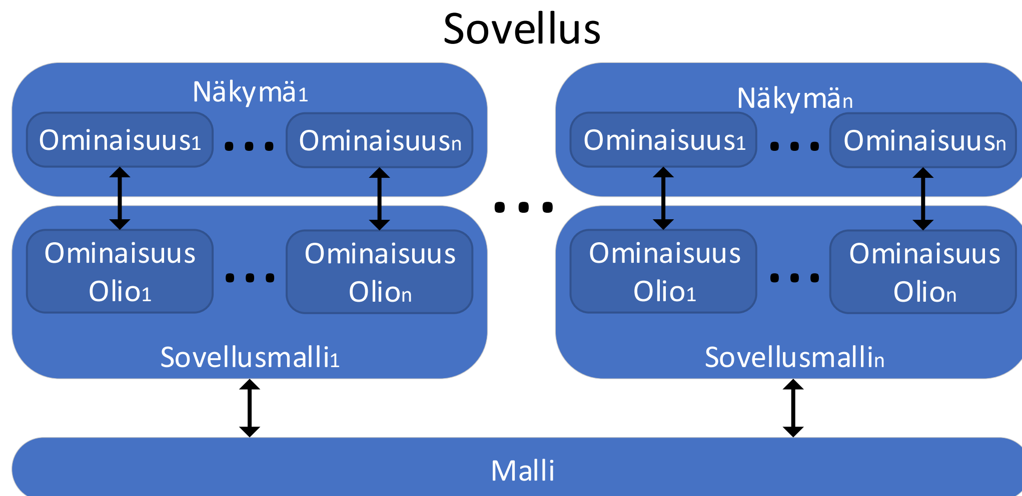
MVVM on Microsoftin kehittämä arkkitehtuurimalli ohjelmistojen rakentamiselle. Termi MVVM koostuu sanoista malli (*model*), näkymä (*view*) ja näkymämalli (*view model*). Mallissa käyttöliittymä sijaitsee näkymässä, logiikka näkymämallissa ja tieto mallissa. Malli kehitettiin ratkaisemaan MVC- ja MVP-mallien ongelmia. Microsoftilla työskentelevä John Grossman esitteli MVVM-mallin blogissaan vuonna -2005. (Vice & Siddiqi 2012, 77–81.)

2.1 Historia

Idea käyttöliittymän ja sen logiikan erottelusta ei ole uusi. MVVM-malli juontaa juurensa 1980-luvulle, jolloin samoja periaatteita noudattava Application Model julkaistiin. Kehityksen jatkuessa nimeksi vaihdettiin Presentation Model, joka vastaa käyttöliittymän logiikkaa ja tilaa hallitsevaa komponenttia. MVVM-mallissa tämä komponentti tunnetaan nimellä ViewModel. (Vice & Siddiqi 2012, 78–79.)

Application Model- ja Presentation Model -mallit kehitettiin ratkaisemaan MVC- ja MVP-mallien rajoitteita ja yhdistämään niiden vahvuuksia. MVC-malli sitoo käyttöliittymän ja sen logiikan yhteen, mikä vaikeuttaa niiden testaamista ja uudelleenkäyttöä. MVP-malli ratkaisi tämän luomalla näkymälle rajapinnan, jonka kautta kommunikointi näkymän kanssa tapahtui. Tämä kuitenkin vaati näkymän jatkuvaa manuaalista päivittämistä muutoksien yhteydessä. (Vice & Siddiqi 2012, 78–79.)

Application Model- (kuvio 1) ja Presentation Model -mallit erottivat näkymän ja logiikan toisistaan käyttämällä ulkoisia tilaobjekteja (Property Object), joihin tallennettiin näkymän tila. Tilaobjektien tilaa pystyttiin seuraamaan ja ne ilmoittivat muutoksista. Tämä mahdollisti näkymän ja ulkoisen luokan kuten Presentation Model tilojen synkronisoinnin toisiinsa. Toteutus ratkaisee MVC- ja MVP-mallien ongelmat, mutta vaatii kehittäjältä paljon työtä sen käyttöönottossa. (Vice & Siddiqi 2012, 78–79.)

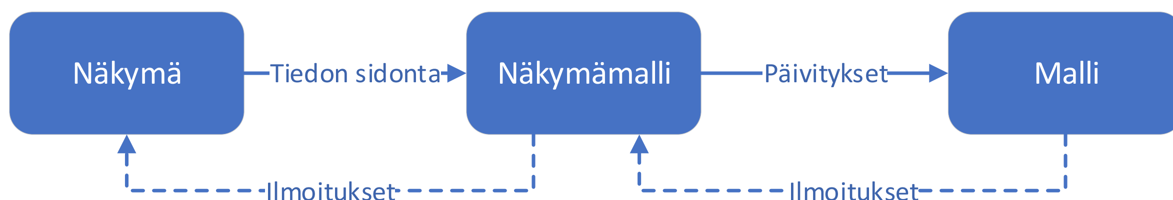


Kuvio 1. Application Model -mallin mukainen sovellus (Perustuu: Vice & Siddiqi 2012, 79)

Ensimmäiset Presentation Model -mallia heti tukevat käyttöliittymäkirjastot olivat Silverlight ja WPF, jotka Microsoft julkaisi .NET Framework 3.0- yhteydessä vuonna 2006. Käyttämällä näitä käyttöliittymäkirjastoja ei kehittäjän enää itse tarvinnut kirjoittaa omaa tilaobjektien toteutusta hyödyntääkseen Presentation Model -mallia. Microsoft oli pyrkinyt rakentamaan täyden tuen mallille kehittämällä datasidonnan (*databinding*) uusiin käyttöliittymäkirjastoihin. Microsoftin toteutus erosi kuitenkin Presentation Model -mallista ja Microsoftilla työskentelevä John Grossman nimesi mallin uudelleen MVVM-malliksi eli Model-View-ViewModel. (Vice & Siddiqi 2012, 78–79.)

2.2 Rakenne ja toimintaperiaate

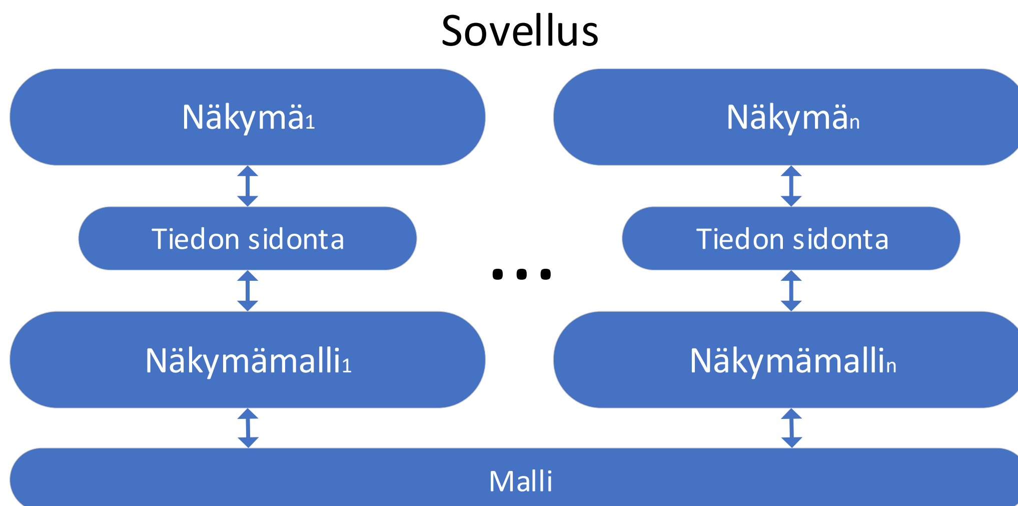
MVVM-mallin rakenne (kuvio 2) koostuu kolmesta ydinkomponentista, jotka ovat malli, näkymä ja näkymämalli. Mallista kaiken hyödyn irti saamiseksi on tärkeää ymmärtää jokaisen komponentin tehtävät. Tärkeätä on myös ymmärtää miten komponentit ovat vuorovaikutuksessa keskenään. (Microsoft 2017d.)



Kuvio 2. MVVM-mallin rakenne (Perustuu: Microsoft 2017d)

MVVM-mallissa näkymä on tietoinen näkymämallista ja näkymämalli on tietoinen mallista. Malli ei ole tietoinen näkymämallista, eikä näkymämalli ole tietoinen näkymästä. Näin näkymämalli eristää näkymän ja mallin toisistaan, mikä mahdollistaa näkymän ja mallin itsenäisen kehityksen. (Microsoft 2017d.)

MVVM-mallin rakenne on MVP-mallin kaltainen. Kommunikointi ei tapahdu enää rajapintojen kautta vaan näkymämalli ja näkymä kommunikoivat datasidonnan avulla. Näkymä yhdistyy näkymämalliin datasidonnan kautta ja näkymämalli malliin rajapintojen kautta. Tätä kutsutaan virheettömäksi MVVM-mallia noudattavaksi sovellukseksi (kuvio 3). Muutamia poikkeuksia on, kuten mahdollisuus yhdistää näkymä suoraan malliin. (Vice & Siddiqi 2012, 80.)



Kuvio 3. MVVM-mallia noudattava sovellus (Perustuu: Vice & Siddiqi 2012, 80)

Näkymämallien ja näkymien alustamiseen ja niiden yhdistämiseen on useita erilaisia tapoja. Nämä tavat voidaan jakaa kahteen kategoriaan, jotka ovat ”näkymä ensin”- ja ”näkymämalli ensin”-asetelmat. Alustamiseen ja yhdistämiseen käytettävä

tapa riippuu kehittäjän mieltymyksestä sekä sovelluksen rakenteesta. Tapojen toteutus eroaa toisistaan, mutta lopputulos on aina sama eli näkymämalli ja näkymä yhdistyvät datasidonnan kautta. (Microsoft 2017d.)

Näkymä ensin -asetelmassa sovellus koostuu näkymistä, jotka yhdistyvät niille kuuluviin näkymämalleihin. Näkymämallit eivät ole riippuvaisia näkymistä, mikä helpottaa yksikkötestattavan sovelluksen rakentamista. Sovelluksen rakennetta on helppo seurata ja ymmärtää, kun ei tarvitse tutkia, miten näkymät luodaan ja yhdistetään. (Microsoft 2017d.)

Näkymämalli ensin -asetelmassa sovellus koostuu näkymämalleista. Sovelluksessa on palvelu, jonka tehtävänä on paikantaa näkymä näkymämallille. Asetelma on luonnollisempi tapa rakentaa sovellus, sillä kehittäjä voi unohtaa näkymien luonnin ja keskittyä sovelluksen loogiseen puoleen. Asetelmassa näkymämallit voivat luoda toisia näkymämalleja. Asetelman ongelmapuolet alkavat esiintyä sovelluksen laajetessa. Sovelluksen rakenne monimutkaistuu, mikä vaikeuttaa sen yksikkötestausta ja jatkokehitystä. (Microsoft 2017d.)

2.2.1 Näkymä

Näkymä on vastuussa käyttöliittymän rakenteesta, sommittelusta ja ulkoasusta. Tavoitteena on, että näkymät määritellään täysin XAML-merkintäkielessä. Näkymien ei tule sisältää mitään liiketoimintalogiikkaa (*business logic*). Tietyissä tapauksissa näkymä voi sisältää rajallisesti käyttöliittymälogiikkaa, kuten animointia, jos sen toteuttaminen XAML-kiellä on mahdotonta. (Microsoft 2017d.)

Näkymän tehtävä on näyttää tiedot, ottaa vastaan käyttäjän antama syöte sekä välittää syöte näkymämallille. Kommunikointi näkymämallin kanssa tapahtuu datasidonnan kautta. Joissain tapauksissa näkymä voi olla vastuussa näkymien yhdistämisestä näkymämalleihin ja uusien näkymien luonnista. (Vice & Siddiqi 2012, 81.)

2.2.2 Näkymämalli

Näkymämalli hallitsee näkymän tilaa, sisältää näkymän logiikan ja muuttaa näkymän tilaa tarvittaessa. Kommunikointi näkymän kanssa tapahtuu datasidonnan avulla. Näkymämalli toimii välittäjänä näkymän ja mallin välissä. Tieto siirtyy näkymämallin kautta näkymälle ja käyttäjän antama syöte näkymältä mallille. Näkymämalli muuntaa mallilta tulevan tiedon näkymälle sopivaksi. (Vice & Siddiqi 2012, 81.)

Näkymämalli sisältää ominaisuuksia ja komentoja, joita näkymä voi käyttää datasidonnan avulla. Ominaisuudet ja komennot määrittelevät näkymän toiminnallisuuden ja näkymä määrittelee, miten toiminnallisuus näytetään. Näkymä kommunikoi näkymämallille käyttäjän eleet komentojen avulla ja näkymämalli suorittaa komentoa vastaavan logiikan. Ominaisuudet kattavat kaiken näkymässä näkyvän tiedon. Ominaisuus voi olla mallilta tullut tieto tai käyttäjän antama. (Microsoft 2017d.)

Näkymämallin ei tarvitse aina muuntaa mallilta tulevaa tietoa, jos malli on hyvin yksinkertainen. Malli voidaan suunnitella tukemaan datasidontaa, jolloin näkymä voi suoraa kommunikoida mallin kanssa. (Microsoft 2017d.)

2.2.3 Malli

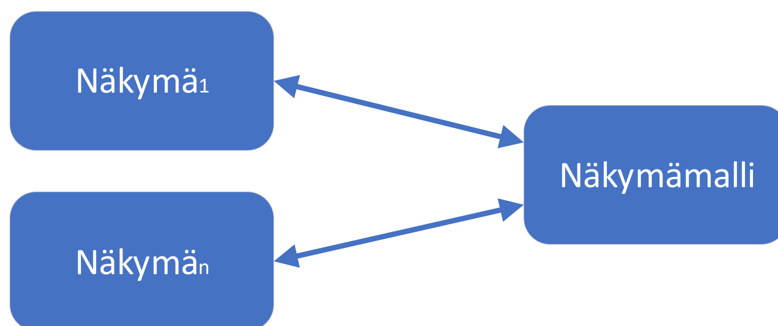
Mallin tehtävä on varastoida tietoa, toimia tietolähteenä ja kuvata tiedon rakenne. Tietolähteenä voi olla tiedosto, tietokanta tai palvelu. Malli sisältää ohjelman liiketoiminta- ja tiedonkäsittelylogiikat. Näkymämalli kommunikoi mallin kanssa rajapintojen avulla. (Microsoft 2017d.)

2.3 Näkymien ja näkymämallien kokoonpanot

Näkymällä ja näkymämallilla on useampi mahdollinen kokoonpano. Yleisimmässä kokoonpanossa jokaisella näkymällä on yksi näkymämalli. Näkymämallilla voi olla useampi näkymä ja näkymällä useampi näkymämalli. (Anderson 2010, 378–380.)

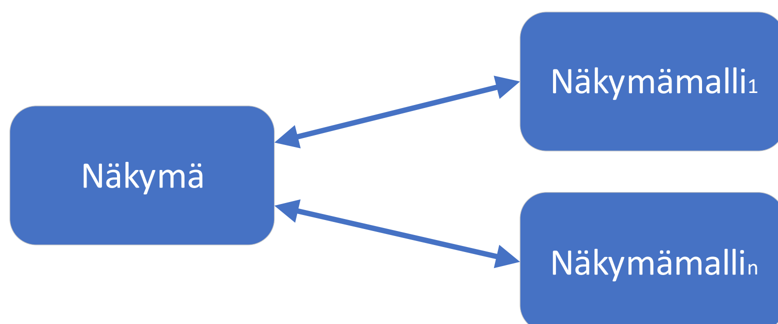
Kuvio 4. Yksi näkymä ja näkymämalli

Yleisimmässä kokoonpanossa jokaiselle näkymälle on yksi näkymämalli (kuvio 4). Kokoonpano tarjoaa sovellukselle selvän rakenteen ja on helpoin tapa oppia hyödyntämään MVVM-mallia. Kokoonpanoa voidaan käyttää sekä näkymä ensin- että näkymämalli ensin -asetelmissa. (Anderson 2010, 378–380.)



Kuvio 5. Näkymämallilla useampi näkymä

Yksi näkymämalli voi olla vastuussa useamman näkymän tilan hallinnasta (kuvio 5). Kokoonpanoa käytetään esittämään näkymän tila erilaisissa näkymissä. Kokoonpano toteutetaan usein näkymämalli ensin -asetelmana. Näkymämalli on vastuussa näkymien alustamisesta ja sisältää viitteet näkymiin. (Anderson 2010, 378–380.)



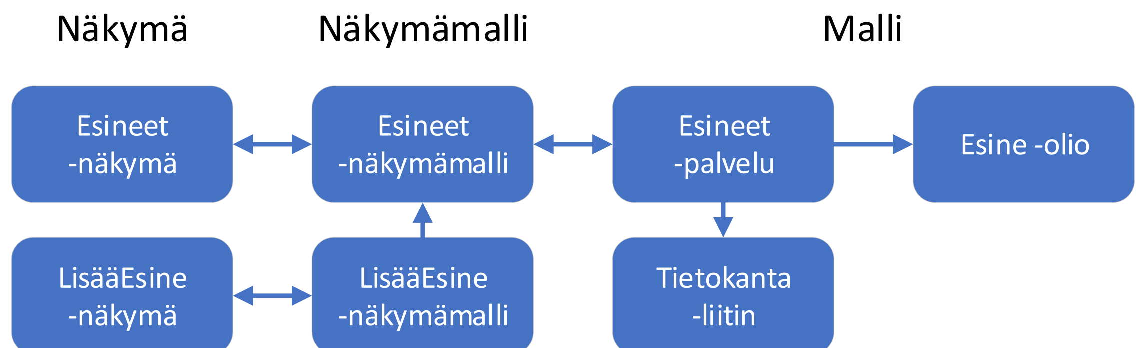
Kuvio 6. Näkymällä useampi näkymämalli

Yhdellä näkymällä voi olla useampi näkymämalli (kuvio 6). Kokoonpano toteutetaan usein näkymä ensin -asetelmana. Näkymä voi koostua useasta osasta, joiden toiminnallisuus on jaettu useaan näkymämalliin. Näkymässä voi olla myös elementti,

joka esittää usean näkymämallin tiedot. Kokoonpanossa ongelmana on usein toistuvan koodin määrä, mikä vaikeuttaa sovelluksen jatkokehitystä ja ylläpitoa. (Anderson 2010, 378–380.)

2.4 MVVM-mallin mukainen sovellus

Kuviossa 7 on esimerkki näkymien, näkymämallien ja mallin suhteista:



Kuvio 7. Esimerkki MVVM-mallin sovelluksesta

Esineet-näkymä on sovelluksen päänäkymä, joka esittää tiedot ja sisältää niiden käsittelyn. Sovelluksen toinen näkymä on LisääEsine-näkymä, jonka kautta voidaan lisätä uusia esineitä. Esineet-näkymän esityslogiikka sijaitsee Esineet-näkymämallissa ja LisääEsine-näkymän LisääEsine-näkymämallissa. LisääEsine-näkymämalli on yhdistetty Esineet-näkymämalliin ja kommunikoi sille LisääEsine-näkymässä lisätyt tiedot. Esineet-näkymämalli toteuttaa mallin Esineet-palvelun toiminnallisuuden. Esineet-palvelu hakee ja päivittää tiedot ulkoiseen tietokantaan Tietokanta-liittimen avulla. Sovelluksen hakema ja muokkaama tieto varastoidaan Esine-luokkaan. (Microsoft 2009.)

2.5 MVVM-mallin edut ja seuraamukset

Jokaisella arkkitehtuurimallilla on etuja ja seuraamuksia. MVVM-mallin vahvuudet tulevat komponenttien vahvasta erottelusta ja heikosta yhdistämisestä. Mallin täydestä hyödyntämisestä on sovelluskehityksessä paljon etuja. Malli ei kuitenkaan ole

vastaus kaikkiin sovelluskehityksen ongelmiin, ja myös MVVM-mallin käytöllä on seuraamuksia. (Microsoft 2012.)

2.5.1 Edut

MVVM-mallissa näkymälogiikka sijaitsee näkymämallissa, mikä helpottaa sen yksikkötestaamista. Kehittäjät voivat luoda yksikkötestejä näkymämallille ja mallille. Näkymää ei tarvitse testata erikseen, sillä näkymämallin yksikkötestit vastaavat näkymän toiminnallisuutta. (Microsoft 2017d.)

Suunnittelijat ja kehittäjät voivat työstää itsenäisesti ja samanaikaisesti komponenttejansa. Suunnittelijat voivat keskittyä näkymään näkymänlogiikasta huolehtimatta. Kehittäjät voivat työstää näkymämallia ja mallia näkymästä huolehtimatta. (Microsoft 2017d.)

Jatkokehitys ja ylläpito helpottuvat huomattavasti. Sovelluksen rakennetta on helppo seurata ja ymmärtää. Näkymiä voidaan suunnitella täysin uusiksi taustalla olevaa logiikkaa ja mallia muuttamatta. Mallia ei tarvitse muuttaa näkymälle sopivaksi, kun näkymämalli voi muuntaa mallilta tulevan tiedon näkymälle sopivaksi. (Microsoft 2017d.)

2.5.2 Seuraamukset

MVVM-mallin toteuttamiseen tarvitaan paljon toistuvaa koodia. Näkymämallissa sijaitsevat ominaisuudet ja komennot tarvitsevat erityisen paljon toistuvaa koodia. Kolmannen osapuolen kirjastot ja teknologiat pyrkivät ratkaisemaan tätä ongelmaa. Jos kolmannen osapuolen kirjaston käyttö ei onnistu, oman ratkaisun toteuttaminen vie paljon aikaa. (Vice & Siddiqi 2012, 128–129.)

MVVM-mallin opiskeleminen vie paljon aikaa. Mallissa on paljon ymmärrettävää, jotta mallista saa kaiken hyödyn irti. MVVM-malli toimii ainoastaan XAML-pohjaisissa sovelluksissa, mikä ei motivoi mallin opiskelua. Malli eroaa paljon muista malleista ja vaatii erilaisen ajattelutavan, mikä saattaa vaikeuttaa opiskelua. (Vice & Siddiqi 2012, 128–129.)

3 WPF

WPF on teknologiana hyvin laaja. Tässä luvussa keskitytään sen arkkitehtuurin ja ominaisuuksiin, jotka ovat tärkeimpiä MVVM-mallin hyödyntämiselle.

WPF eli Windows Presentation Foundation on Microsoftin kehittämä käyttöliittymäkirjasto. Kirjaston päätehtävä on helpottaa suunnittelijoita ja kehittäjiä luomaan vaikuttavia käyttöliittymiä. WPF on suunniteltu mahdollistamaan nykyaikaisten käyttöliittymien luonnin Windows-työpöytäsovelluksille. Kirjasto julkaistiin .NET Framework 3.0:n ja Windows Vistan yhteydessä vuonna 2006. (Microsoft 2010.)

WPF on Windows Forms -käyttöliittymäkirjaston jatkaja. Ennen WPF-kirjaston julkaisua Forms oli suosituin tapa luoda Windows-työpöytäsovelluksia. Sovelluksien ja niiden käyttöliittymien rakentaminen Formsilla vaati useiden eri teknologioiden käyttämistä. Monimutkaisten käyttöliittymien rakentaminen Formsilla oli vaikeaa ja aikaa vievää. WPF suunniteltiin luomaan yhtenäinen alusta käyttöliittymien rakentamiselle ja ratkaisemaan Formsin ongelmia. (Microsoft 2010.)

Windows Forms -käyttöliittymä määriteltiin täysin koodilla. Suunnittelijat tekivät kuvankäsittelyohjelmilla mallikuvia, joiden perusteella kehittäjät rakensivat käyttöliittymän. WPF-käyttöliittymät määritellään XAML-merkintäkielellä. XAML-kielen avulla kehittäjä ja suunnittelija voivat työstää yhteistyössä sovellusta. Suunnittelija voi rakentaa sovelluksen käyttöliittymän Microsoftin Blend-ohjelmaa käyttäen ja kehittäjä voi keskittyä sovelluksen logiikan kirjoittamiseen. (Microsoft 2010.)

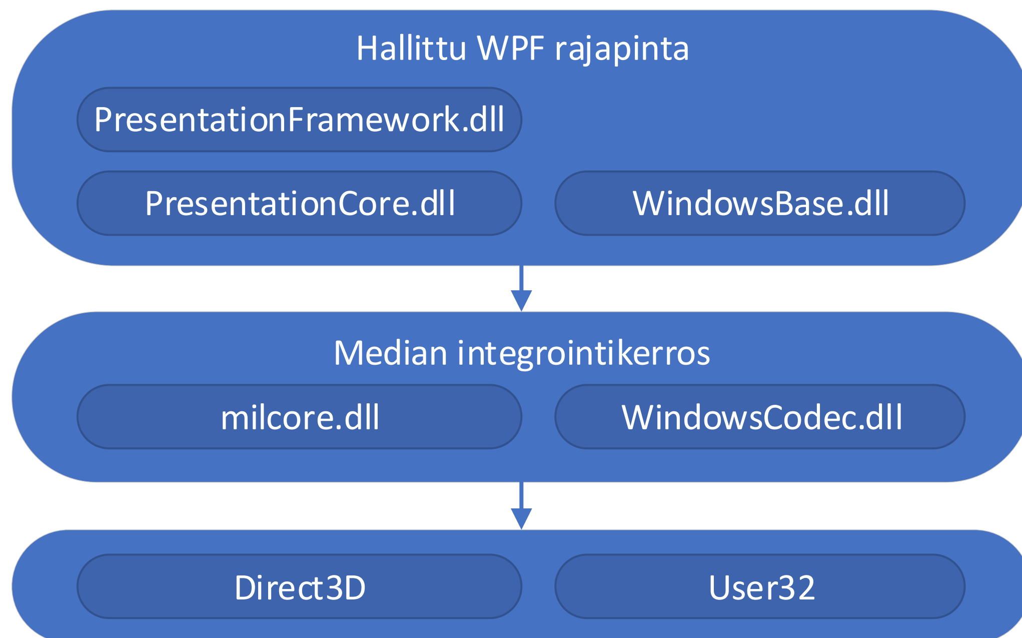
WPF-kirjaston tärkeimpiin ominaisuuksiin kuuluu laitteistokiihdytys (*hardware acceleration*), riippumattomuus tarkkuudesta (*resolution independence*), täysi kontrollien ulkoasun hallinta (*no fixed control appearance*), deklaratiivinen käyttöliittymä (*declarative user interface*) ja vektoripohjainen piirtäminen (*vector-based drawing*):

1. Laitteistokiihdytys: Kaikki WPF-kirjastolla tehdyt käyttöliittymät piirretään DirectX-teknologian kautta. DirectX käyttää prosessointiin tietokoneen prosessorin sijaan näytönohjainta, mikä parantaa sovelluksien suorituskykyä huomattavasti.

2. Riippumattomuus tarkkuudesta: Käyttöliittymä skaalautuu automaattisesti näytön tarkkuuden mukaan.
3. Täysi kontrollien ulkoasun hallinta: Kontrollien ulkoasu on täysin muokattavissa. Vanhemmissa Windows-käyttöliittymäkirjastoissa käyttöjärjestelmä määräsi tiettyjen kontrollien ulkoasun. WPF määrittelee kontrollien toiminnallisuuden, mutta ei niiden ulkoasua.
4. Deklaratiivinen käyttöliittymä: Käyttöliittymä määritellään XAML-merkintäkielessä. Näkymät ja käyttöliittymä voidaan rakentaa täysin ilman koodin kirjoittamista. Tiedon sidonta mahdollistaa käyttäjän antamaan syötteeseen reagoivan ja muuttuvan käyttöliittymän luonnin. Käyttöliittymän elementteihin voidaan yhdistää ehtoja, jotka täyttyessään muuttavat käyttöliittymää.
5. Vektoripohjainen piirtäminen: Kirjaston ytimenä toimii vektoripohjainen grafiikkamoottori. Kontrollit piirretään vektoreiden avulla, mikä tekee käyttöliittymästä täysin riippumattoman tarkkuudesta. WPF on täysin vastuussa käyttöliittymän piirtämisestä ja optimoinnista. (MacDonald 2010, 3–19.)

3.1 Arkkitehtuuri

WPF hyödyntää monikerroksista arkkitehtuuria. WPF-arkkitehtuuri rakentuu kolmesta kerroksesta. Tärkeimmät komponentit ovat PresentationFramework, PresentationCore ja milcore. Ylin kerros koostuu WPF-olioista, tyyleistä, käyttöliittymän perustyypeistä ja WPF-perustyypeistä. Keskimmäisen kerroksen komponenttien tehtävä on kuvien ja käyttöliittymän muuntaminen Direct3D-rajapinnalle prosessoitavaksi. Alimmalla tasolla on käyttöliittymän piirtäminen ja suorituskyvyn optimointi. (MacDonald 2010, 11–12.) Kuviossa 8 esitetään WPF-teknologian arkkitehtuuri.

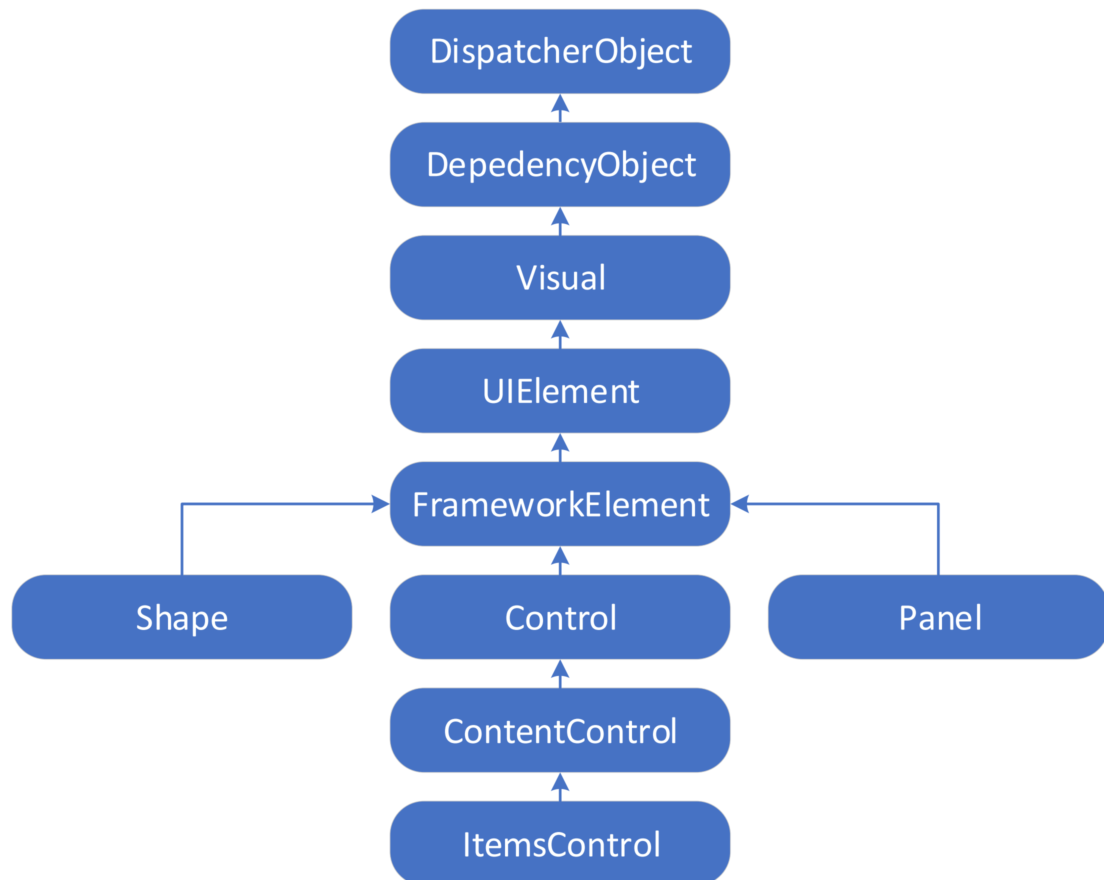


Kuvio 8. WPF-teknologian arkkitehtuuri (Perustuu: MacDonald 2010, 11)

WPF-teknologian arkkitehtuurin (kuvio 8) avainkomponenttien sisältö on seuraava:

- PresentationFramework.dll sisältää korkeimman tason WPF-oliot, -tyypit ja käyttöliittymän tyylit. Olioihin ja tyyppeihin kuuluu ikkunat, paneelit ja laaja osa kontrolleista.
- PresentationCore.dll on PresentationFrameworkin alempi taso. PresentationCore pitää sisällään WPF-perustyypit, joista kaikki muut kontrollit periytyvät.
- WindowsBase.dll sisältää yksinkertaisimmat oliot, joista tärkeimmät ovat DispatcherObject and DependencyObject. DispatcherObject hallitsee säikeitä ja DependencyObject on pohjana käyttöliittymän elementeille.
- milcore.dll on WPF-käyttöliittymän piirtämisen ydin. Milcore-komponentin tehtävä on muuntaa käyttöliittymä Direct3D-rajapinnalle prosessoitavaan muotoon. Windows Vista ja Windows 7 käyttävät milcore-komponenttia työpöydän piirtämiseen.
- WindowsCodec.dll prosessoi, skaalaa ja näyttää kuvatiedostot. Direct3D-rajapinnan tehtävänä on piirtää WPF-käyttöliittymä ja grafiikat. User32 on osa Windows-käyttöjärjestelmää ja se on vastuussa käyttöjärjestelmän ja sovellusten suorituskyvyn optimoinnista. (MacDonald 2010, 11–12.)

WPF-luokkahierarkia (kuvio 9) hyödyntää vahvasti periyttämistä. Moni kirjaston luokkien, olioiden ja tyyppien ominaisuuksista on peritty toiselta luokalta. Kirjaston toiminnallisuuden ymmärtämisen kannalta on hyvä tietää luokkien perimissuhteet. (MacDonald 2010, 12–13.)



Kuvio 9. WPF-luokkahierarkian tärkeimmät luokat (Perustuu: MacDonald 2010, 13)

DispatcherObject on luokka, josta iso osa WPF-olioista periytyy. WPF-sovellukset noudattavat yhden säikeen mallia (*single-thread affinity model*), joka tarkoittaa sovelluksen käyttöliittymän suorittamista yhdellä säikeellä. DispatcherObject-luokan tehtävänä on valvoa tätä mallia ja varmistaa, että käyttöliittymäelementit suoritetaan oikeilla säikeillä. Eri säikeillä suoritettavien käyttöliittymäelementtien keskinäinen vuorovaikutus on epäluotettavaa ja vaarallista. (MacDonald 2010, 13.)

DependencyObject-luokasta perivät luokat saavat käyttöönsä riippuvuusominaisuudet (*depedency property*). Käyttöliittymän kontrollit ja elementit koostuvat ominai-

suuksista, joiden kautta käyttöliittymän kanssa ollaan vuorovaikutuksessa. Riippuvuusominaisuudet ilmoittavat tilan muutoksista, mikä mahdollistaa tilan seuraamisen. (MacDonald 2010, 14.)

Visual-luokka on pohja jokaiselle käyttöliittymän kontrollille ja elementille. Luokka sisältää piirtämiselle olennaisia ominaisuuksia, kuten läpinäkyvyys (*opacity*), muuntaminen (*transformation*) ja osumien tarkistaminen (*hit testing*). Visual-luokka toimii linkkinä korkeamman tason luokkien ja milcoren-komponentin välillä. (MacDonald 2010, 14.)

UIElement sisältää käyttöliittymälle tärkeät toiminnallisuudet, kuten asettelu (layout), syöte (*input*), kohdistaminen (*focus*) ja tapahtumat (*events*). UIElement-luokka on vastuussa käyttöliittymän koon määrittelystä ja syötteen muuntamisesta tapahtumiksi. (MacDonald 2010, 14.)

FrameworkElement on viimeinen osa käyttöliittymän kontrollien ja elementtien ytimestä. FrameworkElement sisältää paljon ulkoasuun liittyviä avainominaisuuksia kuten tyylit ja animaatiot. FrameworkElement-luokasta perivä kontrolli voi hyödyntää tiedon sidonta -teknologiaa. (MacDonald 2010, 14.)

Shape-luokasta periytyvät yksinkertaiset muodot kuten suorakulmio, monikulmio, viiva ja polku. Shape-luokasta periytyviä muotoja voidaan käyttää muiden kontrollien yhteydessä. (MacDonald 2010, 14.)

Control-luokasta periytyvät kaikki kontrollit, joiden kanssa käyttäjä voi olla vuorovaikutuksessa. Luokka sisältää ominaisuuksia kuten fontti, korostus- (foreground color) ja taustaväri (*background color*). Luokan tärkein ominaisuus on tuki mukautetuille ulkoasumalleille. Luokasta periville kontrollille voidaan luoda täysin muokattu ulkoasu, joka säilyttää kontrollin toiminnallisuuden. (MacDonald 2010, 14.)

ContentControl toimii pohjana kaikille kontrolleille, jotka sisältävät Content-ominaisuuden. Content voi koostua monimutkaisista muoto- ja kontrollikokonaisuuksista tai yksinkertaisesta tekstielementistä. (MacDonald 2010, 15.)

ItemsControl on pohjaluokka kaikille kokoelmia esittäville kontrolleille. Luokasta perivät kontrollit ovat hyvin joustavia. Kontrollien esittämän kokoelman ulkoasua voi muokata täysin tarpeiden mukaiseksi. (MacDonald 2010, 15.)

Panel-luokka on pohja asettelua hallitseville luokille, jotka voivat sisältää kontrolleja ja elementtejä. Luokan tehtävänä on asetella ja järjestää sen sisältö asetettujen sääntöjen mukaan. Luokasta periytyvien elementtien käyttö on avainosassa sovelluksen käyttöliittymää rakentaessa. (MacDonald 2010, 15.)

3.2 XAML

XAML eli Extensible Application Markup Language on Microsoftin kehittämä merkintäkieli, joka pohjautuu XML-merkintäkieleen ja seuraa sen syntaksia. XAML on teknologiana hyvin laaja ja sillä on useita käyttötarkoituksia. WPF-kirjaston yhteydessä XAML-kieltä käytetään määrittelemään käyttöliittymän rakenne. Käyttöliittymän rakenne koostuu .NET-olioista muodostetusta puurakenteesta. Käyttöliittymien luomiseen voidaan käyttää työkalua kuten Microsoft Blend -suunnitteluohjelmaa tai Microsoft Visual Studio -ohjelmointiympäristöä. XAML-koodia voi myös kirjoittaa käsin ja määritellä käyttöliittymän manuaalisesti. (MacDonald 2010, 21–25.)

XAML-kieli on rakenteeltaan hyvin yksinkertainen ymmärtää. XAML-tiedoston jokainen elementti vastaa samannimistä .NET-oliota. XML-kielen tavoin elementit voivat sisältää toisia elementtejä, mikä muodostaa puurakenteen. Elementtien ominaisuudet määritellään XML-kielen tapaan attribuuteilla. (MacDonald 2010, 21–25.)

```
<Window x:Class="WpfExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Main Window" Width="400" Height="200" >
    <Grid>
        <TextBlock Text="Example" FontSize="14" Foreground="Blue" FontWeight="Bold"
                   VerticalAlignment="Center" HorizontalAlignment="Center"/>
    </Grid>
</Window>
```

Kuva 1. Yksinkertainen XAML-tiedosto

Kuvan 1 yksinkertainen XAML-tiedosto sisältää elementit Window, Grid ja TextBlock. Window-elementti on tiedoston juurielementti, joka sisältää XAML-kielen tarvitsemat xmlns- ja xlmns:x-nimiavaruudet attribuutteina. Xmlns-attribuutti määrittelee WPF-luokkien sijainnin ja xlmns:x-attribuutti sisältää toimintoja, joilla voidaan

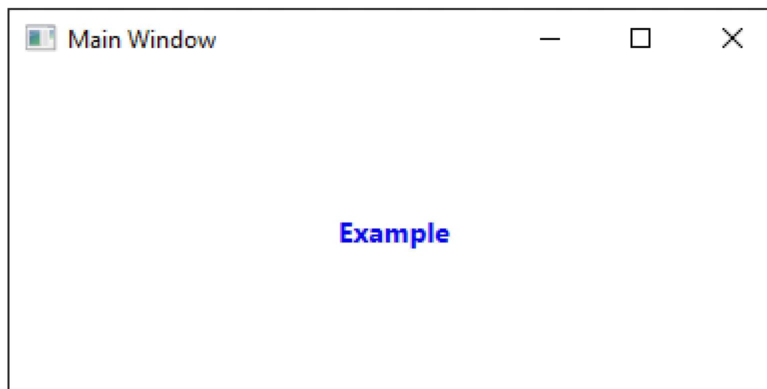
vaikuttaa kuvausmäärittelyjen tulkintaa. Juurielementin x:Class-attribuutti määrittelee MainWindow-luokan sijainnin XAML-nimiavaruudessa. Window-elementille määritellään koko Width- ja Height-attribuuteilla sekä otsikko Title-attribuutilla. Window-elementti sisältää Grid-elementin, jolle ei ole määritelty attribuutteja. Grid-elementti sisältää TextBlock-elementin, jolle on määritelty attribuutit Text, FontSize, Foreground, FontWeight, VerticalAlignment ja HorizontalAlignment. Window-elementin attribuutit Title, Width, Height sekä TextBlock-elementin attribuutit vastaavat elementtejä vastaavien luokkien ominaisuuksia. Attribuutit määritellään merkkijonoina, mutta luokissa ominaisuudet ovat erityyppisiä. TextBlock-elementin attribuuteista Text on merkkijono, FontSize on kokonaisluku, Background on Brush-olio, FontWeight on FontWeight-olio ja VerticalAlignment sekä HorizontalAlignment ovat enum-tietotyyppiä. XAML-muuntaa attribuutit merkkijonoista niitä vastaaviksi tyypeiksi .NET-kirjaston tyyppimuuntajien (*type converter*) avulla. Käyttöliittymä voidaan myös luoda ohjelmallisesti C#-ohjelmointikielellä (kuva 2). (MacDonald 2010, 24–26.)

```
Window window = new Window
{
    Title = "Main Window",
    Width = 400,
    Height = 200
};

Grid grid = new Grid();

TextBlock textBlock = new TextBlock
{
    Text = "Example",
    FontSize = 14,
    Foreground = Brushes.Blue,
    FontWeight = FontWeights.Bold,
    VerticalAlignment = VerticalAlignment.Center,
    HorizontalAlignment = HorizontalAlignment.Center
};

grid.Children.Add(textBlock);
window.Content = grid;
window.Show();
```

Kuva 2. Näkymän luonti ohjelmallisesti

Kaikki WPF-käyttöliittymän elementit ovat .NET-olioita, mikä mahdollistaa näkymän luonnin ohjelmallisesti. Olioiden ominaisuudet ovat samat kuin XAML-kielessä attri-

buutit. Käyttöliittymän luonti ohjelmallisesti on sekavampaa ja vaikeammin ylläpidettävää kuin XAML-kielessä. Käyttöliittymän suunnittelu ei myöskään voida käyttää mitään työkalua, mikä hidastaa työtä. (MacDonald 2010, 32-33.)



Kuva 3. Luotu näkymä

Lopputuloksen kannalta ei ole väliä määritteleekö käyttöliittymän XAML-kielellä vai ohjelmallisesti C#-ohjelmointikielellä. Luotu näkymä (kuva 3) näyttää samalta ulospäin, vaikka toteutustapa on eri. (MacDonald 2010, 33.)

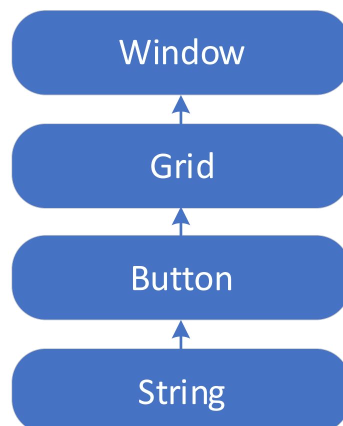
Jokaisen XAML-tiedoston juurielementti on joko Window, NavigationWindow, Page tai UserControl (James & Lalonde 2015, 127). Window-elementti eli ikkuna on säiliö (*container*) sovelluksen näkymille. Sovellus voi koostua yhdestä tai useammasta ikkunasta. Yleensä sovelluksella on yksi pääikkuna ja lisäikkunat ovat dialogi-ikkunoita. NavigationWindow-elementti on Window-elementin kaltainen, mutta pitää sisällään verkkoselaimien tapaiset navigaatiopainikkeet. Page-elementti on näkymä, joka sisältää ryhmän muita elementtejä. Page on suunniteltu käytettäväksi NavigationWindow-elementin sisällä. UserControl-elementti on Page-elementin kaltainen uudelleen käytettävä näkymä. UserControl-elementti voidaan sijoittaa lähes mihin tahansa elementtiin. (James & Lalonde 2015, 130-131.)

XAML-teknologia pohjautuu XML-kieleen sen joustavuuden ja siirrettävyyden vuoksi. XML-tiedostojen rakenne suunniteltiin loogiseksi, luettavaksi ja yksinkertaiseksi, mutta ei tiiviiksi. XAML-tiedostojen prosessoinnin tulisi olla nopeaa, mutta XML ei tätä vaatimusta täytä. Siksi Microsoft on kehittänyt tavan esittää XAML-tiedostot suppeammassa muodossa. BAML eli Binary Application Markup Language on XAML-kielen binäärinen esitysmuoto. Kehittäjien ei tarvitse huolehtia XAML-tiedostojen muuntamisesta BAML-tiedostoiksi. Visual Studion kääntäjä (*compiler*)

muuntaa tiedostot sovellusta kääntäessään (*compile*) ja liittää ne osaksi sovellusta resursseina. XAML-tiedostoja voidaan myös käyttää ilman niiden muuntamista BAML-muotoon. XAML-muodossa olevien tiedostojen prosessointi on kuitenkin hitaampaa kuin vastaavan BAML-tiedoston prosessointi. (MacDonald 2010, 43–50.)

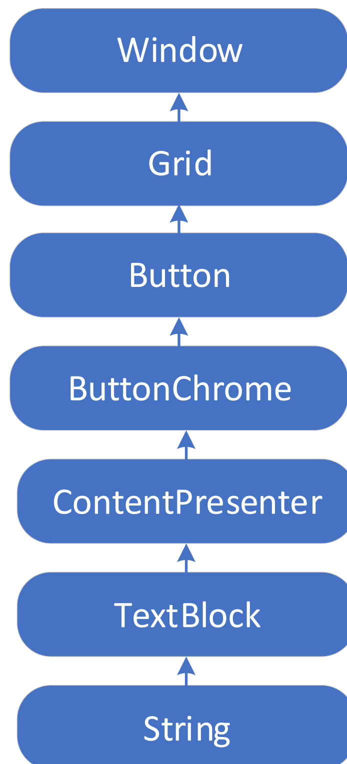
3.3 Looginen ja visuaalinen puu

WPF-käyttöliittymien yhteydessä puhutaan usein loogisesta ja visuaalisesta puusta. Jokaisella WPF-kirjastolla luodulla käyttöliittymällä on looginen puu, joka on identtinen toteutustavasta riippumatta. XAML on luonnollinen tapa määritellä käyttöliittymät sen hierarkkisen rakenteen vuoksi. Looginen puu on hyvin oleellinen osa WPF-teknologiaa. Ominaisuuksia peritään puun ylemmiltä komponenteilta, tapahtumat voivat matkustaa puuta ylös tai alas, ja tyylit toimivat loogisen puun kautta. (Nathan 2014, 56–59.)



Kuvio 10. Looginen puurakenne (Perustuu: Nathan 2014, 57)

Looginen puu (kuvio 10) on yksinkertaistettu kokonaisuus visuaalisesta puusta. Visuaalinen puu on kuin laajennettu looginen puu. Loogisen puun elementit puretaan niiden ydinkomponenteiksi visuaalisessa puussa. Visuaalinen puu paljastaa elementtien todellisen rakenteen ja tarkemmat ominaisuudet. Moni kontrolli on loogisesti yksi elementti, mutta todellisuudessa koostuu useasta eri kontrollista. (Nathan 2014, 56–59.)



Kuvio 11. Visuaalinen puurakenne (Perustuu: Nathan 2014, 58)

Visuaalinen puu (kuvio 11) paljastaa yksinkertaisemmat visuaaliset elementit, joista kontrollit koostuvat. Tämä mahdollistaa käyttöliittymän tarkemman muokkaamisen. Elementtien ulkoasua voidaan muuttaa asettamalla niille tyylejä. Elementeille voidaan asettaa ehtoja, jotka muuttavat niiden ulkoasua ehtojen täyttyessä. Kontrolleille voidaan luoda ulkoasumalleja, joiden avulla visuaalisen puun rakenteeseen voidaan vaikuttaa. (Nathan 2014, 56–59.)

3.4 Reititetyt tapahtumat

Reititetyt tapahtumat (*routed events*) ovat tapahtumia, jotka pystyvät matkustamaan loogista puuta ylös tai alas. Jokaisella tapahtuman läpikäyvällä elementillä on mahdollisuus käsitellä tapahtuma. Tapahtuman voi käsitellä eri tyypin elementti kuin mistä se on peräisin. Jotta tapahtuman käsittely on mahdollista, tarvitsee tapahtuma sitä vastaavan tapahtuman käsittelijän (*event handler*). Reititetyt tapahtumat voidaan jakaa kolmeen eri tyyppiin:

1. Suorat tapahtumat (*direct events*) ovat yksinkertaisimpia tapahtumia. Tapahtuma pysyy alkuperäisessä elementissä, jossa se sai alkunsa. Ainoastaan elementillä, josta tapahtuma on lähtöisin, on mahdollisuus käsitellä tapahtuma.
2. Nousevat tapahtumat (*bubbling events*) matkustavat loogista puuta ylöspäin. Tapahtuman saatua alkunsa se käy läpi jokaisen matkalla olevan elementin, kunnes se saapuu juurielementtiin. Nouseva tapahtuma on yleisin tapahtumatyyppi. Nouseviin tapahtumiin kuuluu käyttäjän antama syöte ja käyttöliittymän tilamuutokset.
3. Laskevat tapahtumat (*tunneling events*) matkustavat loogista puuta alaspäin. Tapahtuma lähtee liikkeelle loogisen puun ylimmästä elementistä, matkustaa määränpää elementille ja käy läpi jokaisen matkalla olevan elementin. Monelle nousevalle tapahtumalle on vastaava laskeva tapahtuma. Laskevat tapahtumat antavat mahdollisuuden käsitellä tapahtuman ennen nousevia tapahtumia. (MacDonald 2010, 109–110.)

Reititetty tapahtuma pitää sisällään tiedot tapahtuman lähteestä (*source*) ja alkuperäisestä lähteestä (*original source*), itse reititetyn tapahtuman tiedot *(routed event object*) sekä käsitelty-muuttujan (*handled*). Tapahtuman lähde on elementti tai kontrolli, jossa tapahtuma on luotu. Alkuperäinen lähde on lähes aina sama kuin lähde, mutta tietyissä tapauksissa alkuperäinen lähde voi olla lähteen sisällä oleva elementti. Reititetyn tapahtuman tietoja voidaan käyttää käsiteltäessä useampaa eri tapahtumaa samalla käsittelijällä. Tapahtuma voidaan asettaa käsitellyksi, mikä keskeyttää tapahtuman matkustamisen loogisessa puussa. (MacDonald 2010, 110–111.)

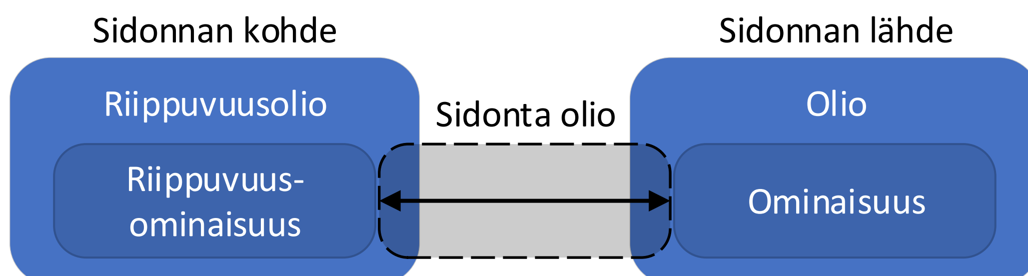
```
<UserControl x:Class="WpfExample.MainUserControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Width="800" Height="450">
    <Grid>
        <StackPanel MouseDown="StackPanel_MouseDown"
                    PreviewMouseDown="StackPanel_PreviewMouseDown"/>
    </Grid>
</UserControl>
```

Kuva 4. MouseDown- ja PreviewMouseDown-tapahtumat

Kuvassa 4 StackPanel-elementillä on tapahtuman käsittelijät reititetyille MouseDown- ja PreviewMouseDown-tapahtumille. MouseDown on nouseva tapahtuma ja PreviewMouseDown on laskeva tapahtuma. StackPanel-elementtiä hiirellä painaessa lähtee ensin PreviewMouseDown-tapahtuma laskeutumaan StackPanel-elementtiin UserControl- ja Grid-elementit läpi käyden. PreviewMouseDown-tapahtuman saavuttua StackPanel-elementtiin MouseDown-tapahtuma lähtee nousemaan Grid- ja UserControl-elementit läpi käyden. Jokaisella tapahtuman läpikäyvällä elementillä on mahdollisuus käsitellä ja pysäyttää tapahtuman kulku. (MacDonald 2010, 109–114.)

3.5 Tiedon sidonta

Tiedon sitominen (*data binding*) on yksi WPF-teknologian tärkeimmistä ominaisuuksista. Tiedon sidonnan toteuttaminen on monimutkaista ja paljon työtä vaativaa, mutta WPF-kirjastossa siitä on tehty yksinkertaista ja nopeaa. Tiedon sidonta tarkoittaa olion sisältämän tiedon esittämistä käyttöliittymässä. Lähes mikä tahansa käyttöliittymän elementti voidaan yhdistää lähes mihin tahansa olion ominaisuuteen. Tietoa voidaan myös työntää käyttöliittymästä takaisin. WPF sisältää paljon erilaisia kontrolleja, joiden avulla erilaista tietoa voidaan esittää erilaisissa muodoissa. Käyttöliittymän elementtejä voidaan myös yhdistää keskenään tiedon sidonnalla. (MacDonald 2010, 557.)



Kuvio 12. Tiedon sitomisen rakenne (Perustuu: Microsoft 2017c)

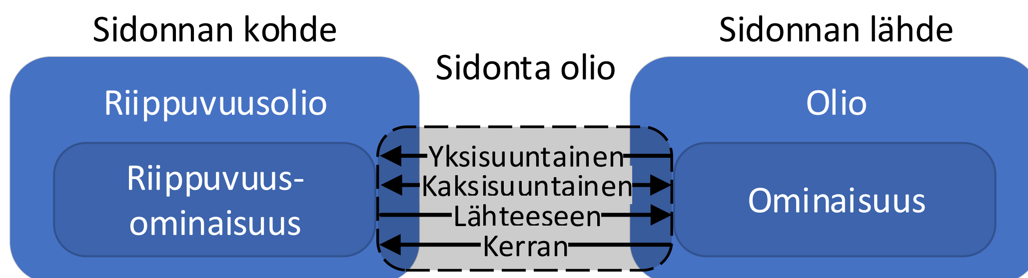
Tiedon sidonnassa (kuvio 12) kohteen ja lähteen välille luodaan sidontaolio, joka yhdistää kohteen ja lähteen. Sidontaolio sisältää tiedon kohteesta, kohdeominaisuudesta, lähteestä ja polusta ominaisuuteen lähteessä. Kohdeominaisuuden täytyy

olla riippuvuusominaisuus (*depedency property*), jotta sidontaolio saa tiedon muutoksista. Lähteenä voi toimia olio, käyttöliittymäelementti tai XML-tiedosto. Tiedon sidonnassa on tärkeä ymmärtää, että kohde yhdistetään lähteeseen. (Microsoft 2017c.)

```
<TextBox Text="{Binding Name}" Name="TextBoxControl" />
<Button Content="{Binding Text, ElementName=TextBoxControl}" />
```

Kuva 5. Tiedon sitominen XAML-kielessä

Kuvassa 5 TextBox-kontrollin Text-ominaisuus on sidottu Name-ominaisuuteen. TextBox-kontrollille on annettu nimi TextBoxControl, jonka avulla muut elementit voivat siihen viitata. Button-controllin Content-ominaisuus on sidottu TextBox-kontrollin Text-ominaisuuteen. (Microsoft 2017c.)



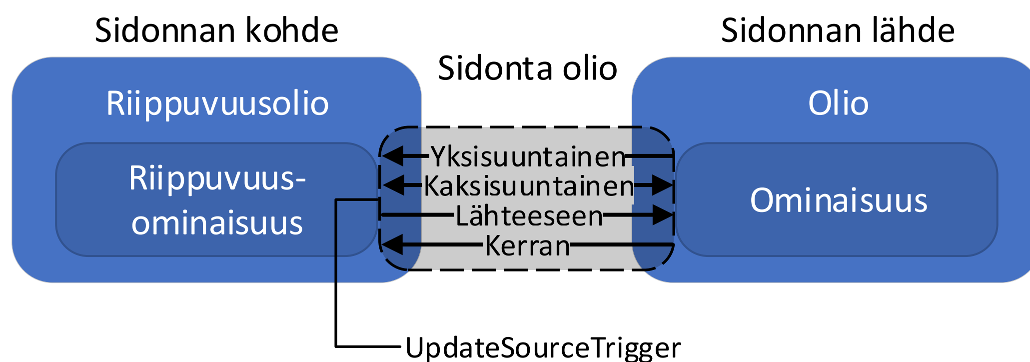
Kuvio 13. Eri tiedon sitomisen muodot (Perustuu: Microsoft 2017c)

Tiedon sitomisen suunta (kuvio 13) voidaan määritellä tarvittaessa. Tieto voi liikkua kohteesta lähteeseen, lähteestä kohteeseen tai molempiin suuntiin. Sidonnan kohde määrittelee sidonnan oletusmuodon, mutta muoto voidaan tarvittaessa määritellä käsin. Yksisuuntainen (*OneWay*) sidonta on yleisin oletusmuoto. Eri tiedon sitomisen suunnat on seuraavassa eritelty tarkemmin:

- Yksisuuntaisessa (*OneWay*) sidonnassa lähdetiedon muutokset tulevat kohteelle, mutta kohteella tapahtuvat muutokset eivät siirry takaisin lähteeseen. Muotoa käytetään kohteille, joiden ei tarvitse muokata tietoa. Kohde on usein tietoa esittävä kontrolli, jonka kanssa käyttäjä ei voi olla vuorovaikutuksessa.
- Yksisuuntainen lähteeseen (*OneWayToSource*) on yksisuuntaisen muodon (*OneWay*) vastakohta eli kohteen muutokset siirtyvät lähteelle, mutta

lähteessä tapahtuvat muutokset eivät siirry kohteelle. Muotoa voidaan käyttää, kun kohteen tekemiä muutoksia ei muokata lähteessä.

- Kaksisuuntaisessa sidonnassa (*TwoWay*) yhdistyvät yksisuuntainen- ja yksisuuntainen lähteeseen -sidontamuodot. Lähteessä tapahtuvat muutokset siirtyvät kohteeseen ja kohteessa tapahtuvat muutokset siirtyvät lähteeseen. Muotoa käytetään käyttöliittymän kontrolleissa, joiden kanssa käyttäjä voi olla vuorovaikutuksessa.
- Yhden kerran (*OneTime*) -sidonnassa kohde alustetaan lähdetiedolla, mutta myöhemmät muutokset eivät siirry kohteelle. Muotoa käytetään, jos lähdetieto ei muutu tai tarvitaan vain yksi tilannekatsaus lähdetiedosta. Muotoa voidaan käyttää myös alustamaan kohde väliaikaisella tiedolla. (Microsoft 2017c.)



Kuvio 14. Tiedon muuttumisen ilmoitustapa (Perustuu: Microsoft 2017c)

Lähdetiedon muuttumisen ilmoittaminen kohteelle on lähteen vastuulla. Kaksisuuntainen ja yksisuuntainen lähteeseen -sidontamuodoissa kohde ilmoittaa muutoksista lähteelle. Kohteen muutoksen ilmoittamisen hetki voidaan määritellä UpdateSourceTrigger-ominaisuudella (kuvio 14). Kuten sidontamuodolla UpdateSourceTrigger-ominaisuudella on myös oletusarvo kohteesta riippuen. Yleisin oletusarvo on PropertyChanged, jossa tiedon muuttumisesta ilmoitetaan heti sen muuttuessa. Toiseksi yleisin on LostFocus, jossa tiedon muuttumisesta ilmoitetaan kohteen menetettyä kohdistamisen. Kolmas mahdollinen asetus on Explicit, jossa sovelluksen täytyy manuaalisesti kutsua UpdateSource-metodia. (Microsoft 2017c.)

```
<TextBox Text="{Binding Name}" Name="TextBoxControl" />
<Button Content="{Binding Text, ElementName=TextBoxControl, Mode=OneWay,
    UpdateSourceTrigger=PropertyChanged}" />
```

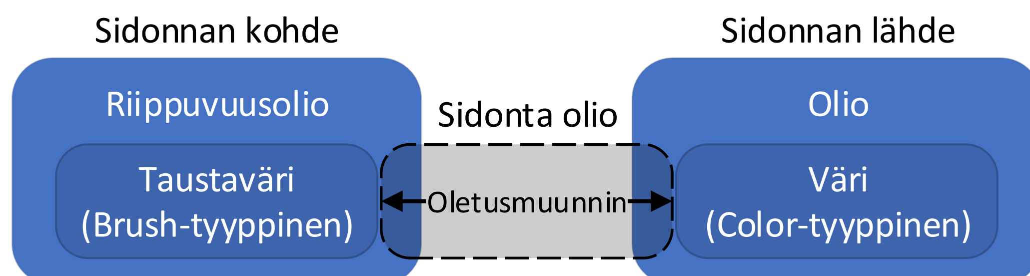
Kuva 6. Tiedon sitomisen asetuksia

Kuvassa 6 Button-kontrollille on määritetty tiedon sidonnan yhteydessä Mode- ja UpdateSourceTrigger-ominaisuudet. Kontrollin Content-ominaisuus päivittyy aina TextBox-kontrollin Text-ominaisuuden muuttuessa. (Microsoft 2017c.)

3.5.1 Tiedon muuntaminen

Tiedon sidonnan yhteydessä voidaan käyttää tiedon muuntamista (data conversion). Kohteelle voidaan määritellä lähteen ja UpdateSourceTrigger-ominaisuuden lisäksi myös muunnin (*converter*). Muuntimia käytetään muuntamaan lähdetieto kohteelle sopivaksi tai esittämään lähdetieto eri muodossa lähdetietoa muuttamatta. WPF-kirjasto sisältää muutaman yksinkertaisen muuntimen ja mukautettuja muuntimia voidaan luoda lisää. (Microsoft 2017c.)

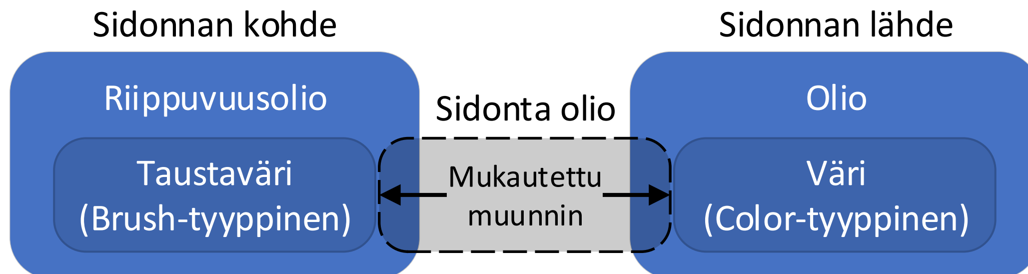
Muuntimia voidaan käyttää esittämään tiedot eri tavoin eri kulttuureille. Valuutta, päivänmäärä tai muu tieto voidaan muuntaa kulttuurille ominaiseen yksikköön. Muuntimilla voidaan esittää sama lähdetieto usealla eri tavalla eri kohteissa samanaikaisesti. Kohteelle voidaan asettaa useampi lähde ja käyttää monimuunninta (MultiValueConverter) tuottamaan kohteelle arvo. (Microsoft 2017c.)



Kuvio 15. Tiedon muuntaminen (Microsoft 2017c)

Ilman määritettyä muunninta tiedon sidonta pyrkii käyttämään oletusmuunninta (kuvio 15). Kuvion 16 tekstimuodossa olevan värin nimen muuntaminen Brush-olioksi

onnistuu oletusmuuntimen avulla, jos värin nimeä vastaava väri on olemassa. Oletusmuuntimen puuttuessa tai muuntamisen epäonnistuessa sovellus kaatuu. (Microsoft 2017c.)



Kuvio 16. Tiedon muuntaminen mukautetusti (Microsoft 2017c)

Luomalla ja määrittelemällä mukautetun muuntimen (kuvio 16) varmistutaan, että tiedon muunnos onnistuu haluttuun muotoon. Mukautetuilla tiedon muuntimilla voidaan muuntaa lähdetieto haluttuun muotoon. (Microsoft 2017c.)

```
public class ColorBrushConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        return Binding.DoNothing;
    }
}
```

Kuva 7. Color-luokasta Brush-luokkaan muunnin

Mukautettu muunnin luodaan uutena luokkana (kuva 7). Muunnin periytyy IValueConverter-rajapinnasta, mikä vaatii muunninta toteuttamaan Convert- ja ConvertBack-metodit. Convert-metodi saa parametreina lähdetiedon, kohdetyypin, muunnin-parametrin ja tiedon kulttuurista. Yksinkertaisessa Color-luokasta Brush-luokkaan muuntimessa käytetään vain lähdetietoa. ConvertBack-metodi muuntaa kohteelta tiedon takaisin lähdetiedoksi. Vaikkei ConvertBack-metodia tarvitsisi, IValueConverter-rajapinta vaatii sen toteuttamisen. (Microsoft 2017c.)

```
<Window.Resources>
    <local:ColorBrushConverter x:Key="ColorToBrush" />
</Window.Resources>
```

Kuva 8. Muuntajan nimen ja sijainnin määrittely

Muuntimen sijainti täytyy määrittää, alustaa ja asettaa nimi x:Key-attribuutilla (kuva 8). Muuntimeen viitataan sen nimellä. (Microsoft 2017c.)

```
<TextBlock Background="{Binding ColorName, Converter={StaticResource ColorToBrush}}" />
```

Kuva 9. TextBlock-kontrollin taustavärin määrittely muuntimella

Muunnin asetetaan määrittelemällä Converter-ominaisuus tiedon sidonnan yhteydessä (kuva 9). Muuntimet ovat staattisia resursseja (*static resource*), minkä vuoksi ne vaativat StaticResource-määritelmän. (Microsoft 2017c.)

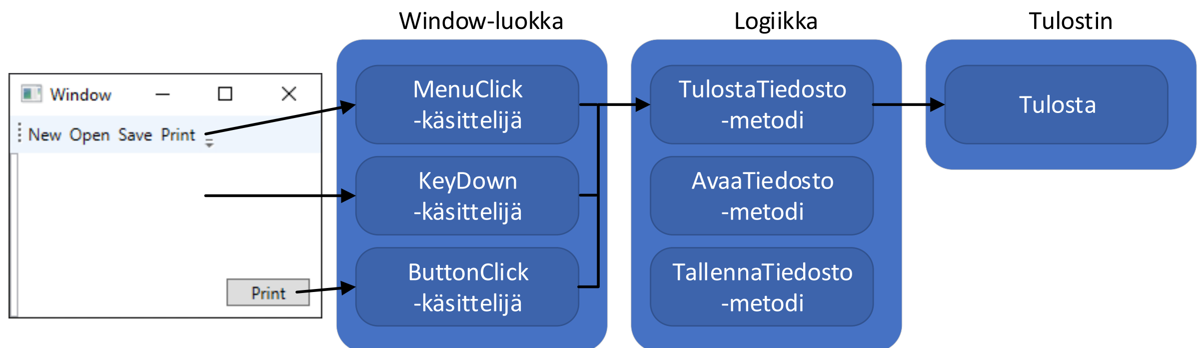
3.6 Komennot

Komennot (*commands*) ovat tapa käsitellä käyttäjän syötettä WPF-sovelluksissa ja niillä on useita eri käyttötarkoituksia ja -tapoja. Komennoissa on eroteltu komennon herättävä elementti ja komennon suorittama logiikka toisistaan. Erottelu mahdollistaa saman komennon logiikan suorittamisen eri lähteistä. Logiikkaa voidaan muokata tarvittaessa eri kohteille sopivaksi. Sovelluksen käyttöliittymän valikoiden, työkalupalkkien ja näppäinyhdistelmien toiminnallisuus voidaan toteuttaa komennoilla. Komennoilla voidaan ilmaista, onko tietty toimenpide suoritettavissa. Usein sovelluksissa painikkeet poistetaan käytöstä, kun komento ei ole suoritettavissa. Komennolle voidaan asettaa suoritusehdot, joiden täyttyessä painike otetaan takaisin käyttöön. WPF-kirjaston komennot koostuvat neljästä avainominaisuudesta:

1. Komento (*command*) esittää suoritettavaa komentoa. Komento ei sisällä tietoa suoritettavasta logiikasta.
2. Komennon lähde (*command source*) on komennon suorittava elementti. Lähde voi olla näppäinyhdistelmä, painike tai muu käyttöliittymän elementti.
3. Komennon kohde (*command target*) on elementti, johon komento suoritetaan. Komennosta riippuen kohdetta ei aina tarvita.

4. Komennon sidonta (*command binding*) yhdistää komennon suoritettavaan logiikkaan. Sidonnan avulla samaa komentoa voidaan kutsua käyttöliittymän eri elementeistä. (Microsoft 2017a.)

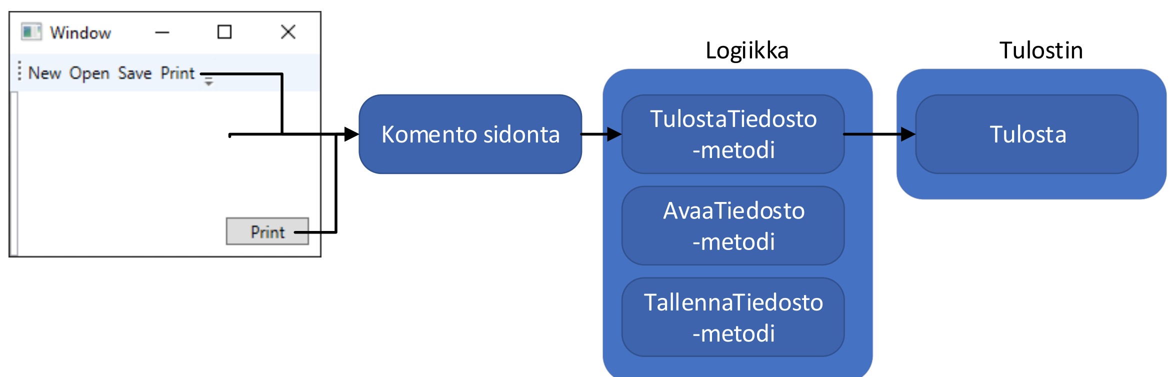
Seuraavasta kuviossa 17 ilmenee tapahtumien yhdistäminen käsittelijöihin käytännössä.



Kuvio 17. Tapahtumien yhdistäminen käsittelijöihin (Perustuu: MacDonald 2010, 243)

Ilman komentoja elementtien tapahtumille luodaan tapahtumankäsittelijä, joka päättelee tapahtuman lähteestä suoritettavan logiikan. Toteutus on toimiva, mutta sovelluksen käyttöliittymän tilan hallinta vaikeutuu. Sovelluksen rakenne monimutkaistuu ja toistuvaa koodia tulee paljon. (MacDonald 2010, 243–258.)

Kuviossa 18 esitetään komentojen käyttö tapahtumankäsittelijöiden sijasta.



Kuvio 18. Tapahtumien yhdistäminen komentoon (Perustuu: MacDonald 2010, 244)

Komentojen avulla eri elementtien tapahtumille voidaan asettaa sama toimenpide. Jokainen komento yhdistetään samaan käsittelijään, joka pitää myös huolen käyttöliittymän tilasta. Komentojen avulla sovelluksen rakenne yksinkertaistuu ja ylläpito helpottuu. (MacDonald 2010, 243–258.)

```
<Window.InputBindings>
    <KeyBinding Key="P" Modifiers="Ctrl" Command="{Binding PrintCommand}" />
</Window.InputBindings>
<Grid>
    <Menu>
        <MenuItem Command="{Binding PrintCommand}" />
    </Menu>
    <Button Command="{Binding PrintCommand}" />
</Grid>
```

Kuva 10. Komennon yhdistäminen ominaisuuksiin tiedon sidonnalla

Komennot tukevat tiedon sidontaa. Kuvassa 10 elementtien komennot on yhdistetty komento-ominaisuuksiin tiedon sidonnalla. Komento-ominaisuudelle määritetään suoritettava logiikka sekä vapaaehtoisesti ehdot suorittamiselle. (MacDonald 2010, 243–258.)

3.7 Tyylit

Käyttöliittymän kontrolleille ja elementeille voidaan määritellä tyylejä (*styles*). Tyylit mahdollistavat kontrollien ja elementtien ulkoasun muokkaamisen toiminnallisuuteen vaikuttamatta. Sovelluksen käyttöliittymälle voidaan luoda yhtenäinen ulkoasu käyttämällä tyylejä. Tyylien kanssa voidaan käyttää laukaisimia, jotka ovat MVVM-mallin kannalta tärkeitä. (Vice & Siddiqi 2012, 89.)

```
<Style TargetType="{x:Type TextBox}" x:Key="CustomTextBox">
    <Style.Setters>
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="14" />
        <Setter Property="FontWeight" Value="Bold" />
    </Style.Setters>
</Style>
```

Kuva 11. TextBox-kontrollille kohdistettu tyyli

Tyyleissä määritellään usein kohde, jolle tyyli on kohdistettu (kuva 11). Kohteen määrittelemällä tyyli tunnistaa kohteen sisältämät ominaisuudet, joiden arvoja voidaan muokata. Kohde määritetään TargetType-attribuutilla. Tyylille määritetään avain, jolla tyyliin voidaan viitata. Tyylin asettamat arvot määritetään Setter-ominaisuuksina. Setter sisältää tiedon ominaisuudesta sekä siihen asetettavasta arvosta. (MacDonald 2010, 283–289.)

```
<Style x:Key="GeneralStyle">
    <Style.Setters>
        <Setter Property="Control.FontFamily" Value="Times New Roman" />
        <Setter Property="Control.FontSize" Value="14" />
        <Setter Property="Control.FontWeight" Value="Bold" />
    </Style.Setters>
</Style>
```

Kuva 12. Yleinen tyyli

Kohdistettujen tyylien lisäksi voidaan määritellä yleinen tyyli (kuva 12), joka voidaan asettaa eri luokan elementeille. Yleisen tyylin toiminnan edellytyksenä on, että jokaisesta elementistä täytyy löytyä tyylin asettamat ominaisuudet. (MacDonald 2010, 283–289.)

```
<TextBox Style="{StaticResource CustomTextBox}" />
<TextBox Style="{StaticResource GeneralStyle}" />
<TextBlock Style="{StaticResource GeneralStyle}" />
```

Kuva 13. Eri kontrolleille asetettuja tyylejä

Tyyli asetetaan elementille määrittelemällä tyylin nimi Style-ominaisuuteen (kuva 13). Erikseen määritetyt tyylit ovat staattisia resursseja, minkä vuoksi ne vaativat StaticResource-määritelmän. TextBox-kontrollille on asetettu sille kohdistettu tyyli. Yleinen tyyli toimii TextBox- ja TextBlock-kontrolleissa, sillä molemmista löytyy tyylin määrittämät ominaisuudet. (MacDonald 2010, 283–289.)

3.8 Laukaisimet

Laukaisimien (*triggers*) avulla voidaan määritellä muuttuva arvo käyttöliittymän ominaisuuksille. Laukaisimelle asetetaan yksi tai useampi ehto, joka voi olla ominaisuus tai tapahtuma. Ehdon täyttyessä laukaisin asettaa ominaisuudelle määritetyn arvon.

Laukaisimia voidaan käyttää tiedon sidonnan kanssa, mikä mahdollistaa näkymänmallin hallita laukaisimia. Ulkoasun mallit ja tyylit voivat sisältää laukaisimia. Laukaisimet voidaan jakaa kolmeen kategoriaan:

1. Ominaisuuslaukaisin (*property trigger*) ja moniominaisuuslaukaisin (*multi property trigger*) seuraavat yhden tai useamman ominaisuuden tilaa. Määritetty arvo asetetaan, kun jokainen ehto täyttyy.
2. Tietolaukaisin (*data trigger*) ja monitietolaukaisin (*multi data trigger*) ovat kuin ominaisuuslaukaisin, mutta tukevat tiedon sidontaa. Yhden tai useamman sidotun olion ominaisuuden tilaa seurataan ja arvo asetetaan määritettyyn jokaisen ehdon täyttyessä.
3. Tapahtumalaukaisin (*event trigger*) seuraa reititettyä tapahtumaa ja asettaa määritetyn arvon ehdon täytyttyä. (MacDonald 2010, 294–297; Vice & Siddiqi 2012, 88–89.)

Yksinkertaisimman ominaisuuslaukaisimen ehdoksi voi asettaa minkä tahansa ominaisuuden tai riippuvuusominaisuuden. Laukaisin tunnistaa seuratun ominaisuuden ja asetetun ehdon. (MacDonald 2010, 294–297.)

```
<Style TargetType="{x:Type TextBox}" x:Key="CustomTextBox">
    <Style.Setters>
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="14" />
    </Style.Setters>
    <Style.Triggers>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="Background" Value="Blue" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Kuva 14. Yksinkertainen ominaisuuslaukaisin

Kuvan 14 ominaisuuslaukaisin seuraa TextBox-kontrollin kohdistamisen tilaa IsFocused-ominaisuudella. Laukaisin tunnistaa ominaisuuden boolean-muuttujaksi ja True-ehdon täyttyessä asettaa kontrollin taustavärin siniseksi. (MacDonald 2010, 295.)

```
<Style TargetType="{x:Type TextBox}" x:Key="CustomTextBox">
    <Style.Setters>
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="14" />
    </Style.Setters>
    <Style.Triggers>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property="IsFocused" Value="True"/>
                <Condition Property="IsMouseOver" Value="True"/>
            </MultiTrigger.Conditions>
            <MultiTrigger.Setters>
                <Setter Property="Background" Value="Blue" />
            </MultiTrigger.Setters>
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

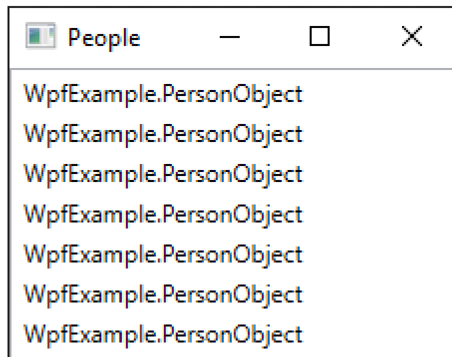
Kuva 15. Moniominaisuuslaukaisin

Kuvassa 15 moniominaisuuslaukaisin seuraa TextBox-kontrollin IsFocused- sekä IsMouseOver-ominaisuuksia. Laukaisimen ehdon täyttyminen vaatii molempien arvojen olevan True-tilassa. Ehtojen täyttyessä laukaisin asettaa kontrollin taustavärin siniseksi. (MacDonald 2010, 295.)

3.9 Ulkoasumallit

Ulkoasumalleilla (*control template*) voidaan muuttaa käyttöliittymän elementtien ulkonäköä. Mallit koostuvat elementin ulkoasun rakenteesta ja käyttäytymisestä. Tyylit rajoittuvat muokattavan elementin olemassa oleviin attribuutteihin, mutta malleilla voidaan muokata ulkonäköä täysin. Tyylien tapaan myös mallit säilyttävät elementin toiminnallisuuden ja tukevat laukaisimia. (Microsoft 2017b.)

Ulkoasumalleilla saadaan tieto esitettyä halutulla tavalla. Tiedon esittämiseen käytettäviä malleja kutsutaan tietomalleiksi (*data template*). Tietomallissa määritellään elementille uusi rakenne ja tiedon sidonnalla määritetään polut ominaisuuksiin lähteessä. Tietomallien käyttö ei ole pakollista, mutta usein elementti ei osaa itsestään esittää tietoa oikein. (Microsoft 2017d.)



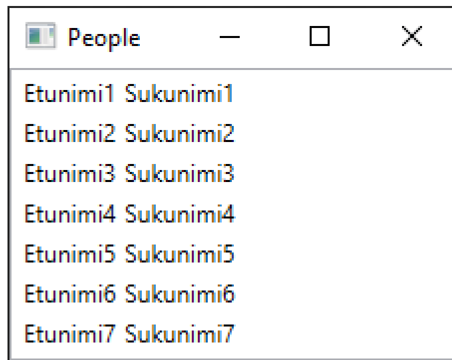
Kuva 16. Lista olioista ilman tietomallia

Kuvassa 16 ListBox-elementtiin on yhdistetty People-lista PersonObject-olioista tiedon sidonnalla. Oliot sisältävät tiedot henkilön etunimestä FirstName-ominaisuudessa ja sukunimestä LastName-ominaisuudessa. ListBox pyrkii esittämään oliot kutsumalla niiden ToString-metodia, joka oletuksena palauttaa olion nimen. Olion ToString-metodi voitaisiin ylikirjoittaa, mutta se ei ole aina mahdollista ja on hyvin joustamatonta. (Microsoft 2017d.)

```
<Grid>
    <ListBox ItemsSource="{Binding People}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding FirstName}" />
                    <TextBlock Text="{Binding LastName}" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
```

Kuva 17. Tietomallin määrittely listalle

Kuvassa 17 ListBox-elementille luodaan tietomalli, jonka avulla ListBox osaa esittää tiedon halutulla tavalla. ListBox-elementin sisältämät elementit korvataan tietomallilla, joka sisältää StackPanel-elementin ja kaksi TextBlock-elementtiä. StackPanel asettelee TextBlock-elementit vierekkäin vaakatasossa. TextBlock-elementtien sisältö yhdistetään tiedon sidonnalla haluttuihin ominaisuuksiin. (Microsoft 2017d.)



Kuva 18. Lista olioista mukautetulla tietomallilla

Kuvassa 18 ListBox osaa nyt esittää tiedon halutulla tavalla. Tietomalli asetetaan jokaiseen ListBox-elementin sisältämään elementtiin. Tietomalli on vastuussa vain tiedon esittämisestä ja tiedon ulkoasusta. ListBox-elementti määrittelee sen sisältämien elementtien asettelun ja ulkoasun. (Microsoft 2017d.)

3.10 MVVM ja WPF

MVVM-malli on Microsoftin suosittelema arkkitehtuurimalli WPF-sovellusten kehittämiseen. Microsoft on rakentanut WPF-alustan ottamaan kaiken irti MVVM-mallista. Vaikka mallia hyödyntämättömien sovellusten luonti on mahdollista, saadaan MVVM-mallin avulla WPF-sovelluksista yksinkertaisempia ja ylläpidettävämpiä. Malli on hyvin suosittu XAML-pohjaisten sovellusten kehittäjien keskuudessa, mihin myös WPF lukeutuu. (Vice & Siddiqi 2012, 78–79.)

WPF-kirjaston tärkeimmät ominaisuudet MVVM-mallin kannalta ovat riippuvuusominaisuudet, tiedon sidonta -teknologia, komentojen ja tapahtumien infrastruktuuri, ulkoasun mallit ja tyylit. Nämä ominaisuudet mahdollistavat näkymien tehokkaan erottamisen niiden esityslogiikasta. WPF oli ensimmäinen käyttöliittymäkirjasto, joka mahdollisti Presentation Model -mallin hyödyntämisen ilman erillistä kirjastoa. (Vice & Siddiqi 2012, 82–93.)

3.10.1 Näkymän yhdistäminen näkymämalliin

Näkymä ja näkymämalli voidaan yhdistää usealla eri tavalla WPF-sovelluksessa. Jokainen tapa tuottaa saman lopputuloksen, mutta eri tavoilla on käyttötarkoituksensa. Käytettävä tapa valitaan käyttötarkoituksen ja sovelluksen rakenteen perusteella. (Chaudhary 2016.)

```
<UserControl x:Class="LogFileViewer.Example.ExampleViewUserControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:viewModels="clr-namespace:LogFileViewer.Example">
    <UserControl.DataContext>
        <viewModels:ExampleViewModel />
    </UserControl.DataContext>
</UserControl>
```

Kuva 19. Näkymälle näkymämallin määrittely XAML-kielessä

Ensimmäisellä XAML-tavalla näkymä alustaa ja yhdistää näkymämalliin näkymän luonnin yhteydessä. UserControl-juurielementissä on määritelty näkymämallin sijainti. Näkymämalli määritellään elementtinä UserControl-elementin DataContext-ominaisuuden alle (kuva 19). Näkymä kutsuu käynnistyessään näkymämallin rakentajaa (*constructor*). (Chaudhary 2016.) Sama tapa voidaan myös toteuttaa näkymän taustakoodissa (*code-behind*).

```
public partial class ExampleViewUserControl : UserControl
{
    public ExampleViewUserControl()
    {
        InitializeComponent();
        DataContext = new ExampleViewModel();
    }
}
```

Kuva 20. Näkymämallin määrittely näkymälle taustakoodissa

Näkymän rakentajassa voidaan määritellä näkymälle näkymämalli (kuva 20). Näkymän DataContext-ominaisuudelle alustetaan uusi näkymämalli. Näkymämallin määrittely näillä tavoilla luokitellaan ”näkymä ensin” -asetelmaksi. (Chaudhary 2016.)

```
<Grid>
    <views:ExampleViewUserControl DataContext="{Binding ExampleViewModel}" />
</Grid>
```

Kuva 21. Näkymämalli määrittely tiedon sidonnalla

Näkymämalli voidaan myös määritellä tiedonsidonnalla (kuva 21). Olemassa oleva näkymämalli yhdistetään tiedon sidonnalla näkymän DataContext-ominaisuuteen. Näkymämalli on luotu ennen näkymää, mikä tekee tavasta ”näkymämalli ensin” -asetelman. (Chaudhary 2016.)

3.10.2 Näkymän ja näkymämallin kommunikointi

Näkymä ja näkymämalli kommunikoivat keskenään tapahtumilla. Jotta näkymä saa ilmoituksia tiedon muutoksista ja näkymämallin osatakseen käsitellä näkymän tapahtumat tarvitsee näkymämalli hieman työtä. (Lapthorn 2012.)

```
<StackPanel>
    <TextBlock Text="{Binding Status}" />
    <Button Content="Click me!" Command="{Binding UpdateStatusCommand}" />
</StackPanel>
```

Kuva 22. Tekstilohkon ja painikkeen sisältävä näkymä

Kuvassa 22 on määritelty tekstilohko ja painike. Tekstilohkon tekstiksi Text-attribuuttiin on tieto sidottu Status-ominaisuus. Painikkeelle on määritelty sisältö ja komennoksi Command-attribuuttiin on tieto sidottu UpdateStatusCommand-ominaisuus. Painiketta painaessa tahdotaan tekstilohkon tekstin muuttuvan.

```
public class ExampleViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChange(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Kuva 23. Näkymämallin käyttämä ilmoitusmetodi

Jotta näkymä saa ilmoituksia muutoksista näkymämallilta, täytyy näkymämallin toteuttaa INotifyPropertyChanged-rajapinta. Rajapintaa varten luodaan PropertyChangedEventHandler-tapahtuma ja OnPropertyChange-metodi (kuva 23). Metodia kutsumalla luodaan uusi tapahtuma, joka ilmoittaa näkymälle annetun ominaisuuden muuttumisesta. (Lapthorn 2012.)

```
private string status = "Command not called.";
public string Status
{
    get { return status; }
    set
    {
        status = value;
        OnPropertyChange("Status");
    }
}
```

Kuva 24. Status-ominaisuuden määrittely

Näkymämallin ominaisuuden arvon asettamisen yhteydessä kutsutaan OnPropertyChange-metodia (kuva 24). Metodille annetaan muuttuneen ominaisuuden nimi. (Lapthorn 2012.)

```
public void UpdateStatus()
{
    Status = "Command called!";
}
```

Kuva 25. Komennon logiikka

Komentojen ja tapahtumien suoritettava logiikka määritellään erillisenä metodina (kuva 25). Tarvittaessa metodi voi myös ottaa vastaan parametrejä. (Prajapati 2015.)

```
public class UpdateStatusCommand : ICommand
{
    private ExampleViewModel exampleViewModel;
    public UpdateStatusCommand(ExampleViewModel viewModel)
    {
        exampleViewModel = viewModel;
    }
    public bool CanExecute(object parameter)
    {
        return true;
    }
    public void Execute(object parameter)
    {
        exampleViewModel.UpdateStatus();
    }
    public event EventHandler CanExecuteChanged;
}
```

Kuva 26. Komennon toteutus

Komennolle luodaan uusi luokka (kuva 26), joka toteuttaa ICommand-rajapinnan. Rajapinta vaatii Execute-metodin, CanExecute-metodin ja CanExecuteChanged-tapahtumakäsittelijän määrittelyn. Execute-metodi sisältää logiikan, jonka komento suorittaa. CanExecute-metodilla voidaan asettaa ehto komennon suorittamiselle. CanExecuteChanged-tapahtumankäsittelijä suorittaa tilamuutoksien tapahtuessa CanExecute-metodin. (Prajapati 2015.)

```
private UpdateStatusCommand updateStatusCommand;
public ICommand UpdateStatusCommand
{
    get
    {
        return updateStatusCommand;
    }
}
```

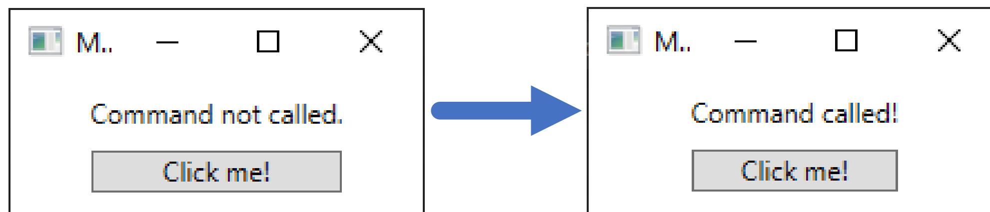
Kuva 27. Komento-ominaisuuden luonti

Näkymämallissa määritellään komennolle ominaisuus (kuva 27), johon näkymä voi tieto sitoa. Komento on luotu UpdateStatusCommand-luokan oliolla, mutta julkisesti komento on ICommand-rajapinnan toteuttava luokka. (Prajapati 2015.)

```
public ExampleViewModel()
{
    updateStatusCommand = new UpdateStatusCommand(this);
}
```

Kuva 28. Komennon alustaminen

Näkymämallin alustuksen yhteydessä alustetaan myös komento-olio (kuva 28). Komennolle annetaan viittaus sen alustavaan näkymämalliin, jonka sisältämää metodia se kutsuu. (Prajapati 2015.)



Kuva 29. Luotu ohjelma

Luodussa ohjelmassa (kuva 29) käyttäjän painaessa painiketta teksti vaihtuu ilmaisemaan tilan muutosta. Näkymä lähettää näkymämallille ilmoituksen painikkeen painamisesta ja näkymämalli suorittaa tapahtumaa vastaavan logiikan, joka muuttaa tekstiominaisuutta. Tekstiominaisuuden muutoksesta lähtee ilmoitus näkymälle ja teksti päivittyy näkymässä.

3.11 WPF ja UWP

UWP eli Universal Windows Platform on Microsoftin uusin alusta sovelluksien kehittämiselle. UWP on samanaikaisesti WPF-alustan jatkaja sekä sen vaihtoehtoinen tapa luoda työpöytäsovelluksia. UWP on uusi teknologia, joka tuo mukanaan paljon uudistuksia ja parannuksia verrattuna vanhempaan WPF-teknologiaan. WPF on teknologiana ehtinyt kypsyä ja saanut laajan kolmannen osapuolen kirjastojen tuen. UWP on uutena teknologiana jatkuvasti kehityksen alla, eikä ole vielä ehtinyt saamaan suurta suosiota. (Microsoft 2018a.)

WPF ja UWP ovat osittain samankaltaisia ja osittain erilaisia. WPF-kirjaston tapaan käyttöliittymän voi rakentaa UWP-sovellukseen XAML-merkintäkielen avulla. (Microsoft 2018a.) UWP tarjoaa myös muita mahdollisuuksia käyttöliittymän luomiseen kuten DirectX ja HTML (Microsoft 2018b). WPF-sovellukset toimivat Windows-käyttöjärjestelmällä Windows 7 -versiosta ylöspäin, kun taas UWP-sovellukset toimivat vain Windows 10 -versiossa (Microsoft 2018a). UWP-sovellukset toimivat myös Windows Phone -älypuhelimissa, Xbox-pelikonsolissa, Internet of Things -pilvessä

ja Surface Hub -laitteissa (Microsoft 2018b). WPF-sovellukset voidaan rakentaa tukemaan kosketusnäyttöjä (Microsoft 2010), mutta UWP-alustan sovellukset on suunniteltu alusta asti toimimaan kosketusnäyttölaitteilla (Microsoft 2018b). UWP tarjoaa paremman suorituskyvyn ja turvallisuuden suorittamalla sovellukset hiekkalaatikossa *(sandbox)* (Microsoft 2018b).

UWP-alustan suurimpana haittana voidaan pitää sille Microsoftin asettamia rajoituksia. UWP ei tue vanhempia Windows-versioita (Microsoft 2018b), mikä antaa WPF-kirjastolle edun. UWP-käyttöliittymän kontrollien muokkaaminen on WPF-kontrolleja rajallisempaa. Microsoft vaatii UWP-sovelluksille sertifioinnin ja rajoittaa virallisen julkaisemisen Windows Store -sovelluskauppaan. (Microsoft 2018b.) WPF-sovelluksia voi julkaista vapaasti ilman sovelluskauppaa tai sertifiointia. WPF-sovellus perii sovellusta suorittavan käyttäjätilin oikeudet resurssien käsittelyyn. (Microsoft 2010.) UWP-sovelluksien suorittaminen hiekkalaatikossa rajoittaa niiden resursseihin pääsyä. UWP-sovellukset eivät pysty käynnistämään uusia prosesseja tai palveluja, kommunikoimaan toisten prosessien kanssa tai käsittelemään laitteen tiedostoja. UWP on turvallisempi alustana, mutta tietynlaisten sovelluksien luonti on mahdotonta. (Microsoft 2018b.)

4 VAATIMUSMÄÄRITTELY

Tarve työkalusta lokitiedostojen läpikäymisen helpottamiseksi oli ollut yrityksessä jo hetken tiedossa. Yrityksen järjestelmät ja ohjelmistot tuottavat paljon lokitiedostoja, mutta mitään erityistä työkalua niiden tarkasteluun ei ollut. Erilaisia sovelluksia lokien käsittelyyn on paljon, mutta mikään ei ole täyttänyt jokaista asetettua vaatimusta. Työkalun pitäisi vastata yrityksen tarpeita täysin ja sen tulisi olla luotettava. Lokeja käytetään paljon seuraamaan järjestelmien tiloja eri ajankohtina ja ratkaisemaan asiakkaiden ongelmia. Lokitiedostoja kerätään paljon, mutta niiden täysi hyödyntäminen on vaikeaa ilman eritystä työkalua.

Projektin alkaessa pidettiin palaveri työkalun tarpeesta ja sille asetetuista vaatimuksista. Työkalun täytyy olla mahdollisimman monen käytettävissä, sen tulee olla kevyt ja tehokas. Lokitiedostoja on erilaisissa tiedostomuodoissa ja lokitiedostojen rakenne vaihtelee. Työkalun tulisi osata tunnistaa ja lukea lokien tiedostomuoto ja rakenne. Luettuja lokitiedostoja pitäisi pystyä suodattamaan ja järjestämään. Tärkeänä ominaisuutena pidettiin hakuominaisuutta, jonka avulla hakutermiä vastaavien lokirivien välillä voisi siirtyä nopeasti.

Työkalun toteuttamiseen annettiin täysin vapaat kädet. Vastaavan työkalun voisi tuottaa usealla eri teknologialla ja tekniikalla. Työpöytäsovelluksessa suorituskyky on sidottu tietokoneen tehoon. Ylimääräisiltä kustannuksilta myös vältytään, mitkä esimerkiksi palvelintoteutuksessa tulisivat vastaan. Yrityksessä on käytössä Microsoftin teknologiaa. Yrityksestä ehdotettiin työkalun toteuttamista WPF-sovelluksena, joka on myös tekijälle jo ennestään tuttua. MVVM-arkkitehtuurimallin opettelua myös suositeltiin, koska se on yhteensopiva WPF-käyttöliittymäkirjaston kanssa.

Työkalun edetessä järjestettiin useita palavereja, joissa tarkistettiin työkalun tilanne ja tarvittaessa asetettiin uusia tavoitteita. Sovelluksen rakentaminen yrityksen sisällä antaa mahdollisuudet sen ylläpidolle ja jatkokehitykselle. Käyttäjien palautteeseen voidaan reagoida nopeampaa ja korjata ilmeneviä ongelmia.

5 SOVELLUKSEN SUUNNITTELU JA ARKKITEHTUURI

Sovelluksen kehittämisen nopeuttamiseksi käytettiin muutamaa kolmannen osapuolen kirjastoa. Tärkein hyödynnetty kirjasto on Laurent Bugnionin MVVM Light Toolkit -komponenttikirjasto. MVVM-mallin mukaisten sovelluksien kehittämiseen on useita eri kirjastoja ja MVVM Light Toolkit on yksi suosituimmista. Työkalun käyttöliittymän suunnittelun nopeuttamiseksi käytettiin Benjamin Rühlin kehittämää AdonisUI-käyttöliittymäkirjastoa. Sovelluksen jakelun helpottamiseksi käytettiin Fody-työkalua ja sen Costura.Fody-lisäosaa.

Kaikki sovelluksessa käytetyt kirjastot perustuvat avoimeen lähdekoodiin. Jokainen kirjasto on lisensoitu MIT-lisenssillä, joka antaa täydet oikeudet kirjastojen kaupalliseen käyttöön, muunteluun, jakeluun ja yksityiseen käyttöön.

5.1 MVVM Light Toolkit

MVVM Light Toolkit on Laurent Bugnionin kehittämä komponenttikirjasto. Kirjasto on suunniteltu helpottamaan ja nopeuttamaan MVVM-mallin mukaisten sovelluksien kehittämistä ja ylläpitoa. Uusimman version lähdekoodi löytyy Bugnionin GitHub-sivulta ja valmiiksi käännetyn version voi ladata Microsoftin NuGet-palvelusta. Kirjasto tukee kaikkia XAML-pohjaisia sovelluskehyksiä kuten UWP, WPF, Silverlight ja Xamarin. (Bugnion 2014.)

Kirjasto koostuu useasta eri komponentista ja apuluokasta. Kirjaston tärkeimpinä komponentteina voidaan pitää ViewModelBase-, ObservableObject- ja RelayCommand-luokkia. Kirjasto sisältää myös luokat Messenger ja DispatcherHelper. Komponenttikirjasto tuo myös mukanaan useamman sovelluspohjan, luokkapohjan ja koodinpätkän nopeuttamaan sovellusten kehitystä. (Bugnion 2014.)

5.1.1 Ominaisuudet

ViewModelBase- ja ObservableObject-luokat on suunniteltu vähentämään toistuvan koodin määrää MVVM-mallia noudattavissa sovelluksissa. Luokat sisältävät paljon

näkymämalleissa toistuvaa koodia näkymän ja näkymämallin väliseen kommunikointiin liittyen. ViewModelBase-luokasta perimällä näkymämallien ei tarvitse itse erikseen toteuttaa INotifyPropertyChanged-rajapintaa ilmoittaakseen näkymälle tiedon muutoksista. Näkymämallin ei myöskään tarvitse huolehtia olioiden tietojen paljastamisesta näkymälle, kun oliot peritään ObservableObject-luokasta. (Bugnion 2014.)

```
public class ExampleViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChange(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
    private string status;
    public string Status
    {
        get { return status; }
        set
        {
            status = value;
            OnPropertyChange("Status");
        }
    }
}
```

Kuva 30. Ominaisuus näkymämallissa ilman MVVM Light Toolkit -kirjaston käyttöä

Kuvassa 30 on määritelty Status-ominaisuus näkymämallissa MVVM-mallille tutulla tavalla. Näkymämallin täytyy periä ja toteuttaa INotifyPropertyChanged-rajapinta sekä kutsua OnPropertyChanged-metodia tiedon muuttuessa.

```
public class MVVMLightViewModel : ViewModelBase
{
    private string status;
    public string Status
    {
        get { return status; }
        set { Set(ref status, value); }
    }
}
```

Kuva 31. Ominaisuuden toteutus MVVM Light Toolkit -kirjaston avulla

Kuvassa 31 on toteutettu kuvan 30 koodi MVVM Light Toolkit -kirjaston avulla. Näkymämalli peritään ViewModelBase-luokasta, joka pitää sisällään Set-metodin. Set-metodi kutsuu tiedon asettamisen yhteydessä myös OnPropertyChanged-metodia,

jonka ViewModelBase-luokka toteuttaa. Tiedon muuttumisen ilmoittamiseen ei tarvitse enää käsin kirjoittaa muuttuvan arvon nimeä, sillä Set-metodi huolehtii myös siitä. (Bugnion 2014.)

ObservableObject-luokka toimii ViewModelBase-luokan tavoin. Tietoa varastoiva olioluokka voidaan periyttää ObservableObject-luokasta, jolloin vältytään toistuvan koodin kirjoittamiselta. ViewModelBase- ja ObservableObject-luokat ajavat osittain saman asian, mutta ne ovat kuitenkin tarkoitettu eri käyttötarkoituksiin. ViewModelBase-luokka sisältää paljon muita näkymämallille olennaisia ominaisuuksia ja metodeja. ObservableObject-luokka on hyvin kevyt ja pitää sisällään lähes ainoastaan tiedon muuttumisen ilmoittamisen logiikan. (Bugnion 2014.)

5.1.2 Komennot

Tapahtumien ja komentojen käsittelyn helpottamiseksi kirjasto sisältää RelayCommand-luokan. RelayCommand-luokka toteuttaa ICommand-rajapinnan ja yksinkertaistaa komentojen luontia näkymämallissa. RelayCommand-luokka huolehtii myös komennon suoritusehtojen tilan tarkastamisesta ja ilmoittaa näkymälle tilamuutokset. RelayCommand-luokkaa hyödyntämällä uusille komennoille ei tarvitse kirjoittaa kokonaisia luokkia, mikä selventää sovelluksen rakennetta ja vähentää toistuvaa koodia. (Bugnion 2014.)

```
public class ExampleViewModel
{
    private UpdateStatusCommand updateStatusCommand;
    public ICommand UpdateStatusCommand
    {
        get { return updateStatusCommand; }
    }
    public ExampleViewModel()
    {
        updateStatusCommand = new UpdateStatusCommand(this);
    }
    public void UpdateStatus()
    {
        Console.WriteLine("Command called!");
    }
}
public class UpdateStatusCommand : ICommand
{
    private ExampleViewModel exampleViewModel;
    public UpdateStatusCommand(ExampleViewModel viewModel)
    {
        exampleViewModel = viewModel;
    }
    public bool CanExecute(object parameter)
    {
        return true;
    }
    public void Execute(object parameter)
    {
        exampleViewModel.UpdateStatus();
    }
    public event EventHandler CanExecuteChanged;
}
```

Kuva 32. Komento ja komentoa hyödyntävä näkymämalli

Kuvassa 32 on komennon sisältävä näkymämalli ja ICommand-rajapinnan toteuttava komento. Yksinkertaisenkin komennon toteuttaminen vaatii paljon koodia ja jokaiselle uudelle komennolle on luotava oma luokka.

```
public class MVVMLightViewModel : ViewModelBase
{
    RelayCommand UpdateStatusCommand => new RelayCommand(UpdateStatus);
    public void UpdateStatus()
    {
        Console.WriteLine("Command called!");
    }
}
```

Kuva 33. Komento RelayCommand-luokkaa hyödyntämällä

MVVM Light Toolkit -kirjaston RelayCommand-luokan avulla komentojen toteuttamiseen tarvittava koodi on määrältään huomattavasti pienempi (kuva 33). RelayCommand-luokalle annetaan parametreina komennon suorittama logiikka sekä tarvittaessa ehto suorittamiselle. Komento ei tarvitse viitettä sen hyödyntämään näkymämalliin, mikä vähentää komennon alustamiseen tarvittavaa koodia. (Bugnion 2014.)

```
<UserControl xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
             xmlns:mvvm="http://www.galasoft.ch/mvvmlight"
             <...>
             >
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Event">
            <mvvm:EventToCommand Command="{Binding Command}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
<...>
```

Kuva 34. EventToCommand-luokan määrittely

Kirjasto sisältää tapahtumien käsittelemiseen EventToCommand-luokan. Luokka muuntaa näkymän tapahtumat komennoiksi, jotka voidaan tietositoa näkymämalliin ja käsitellä. EventToCommand-luokka hyödyntää Microsoftin Interaction-kirjastoa ja tarvitsee sen sijainnin määrittelyn näkymässä toimiakseen. Luokka hyödyntää Interaction-kirjastosta EventTrigger-luokkaa. Näkymässä EventToCommand-elementti määritellään Interaction-elementin EventTrigger-elementin sisään (kuva 34). (Bugnion 2014.)

5.2 AdonisUI

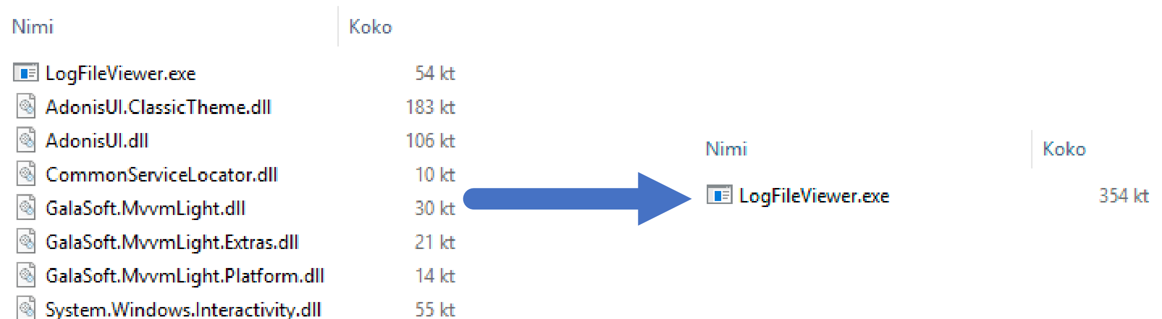
AdonisUI on kevyt käyttöliittymäkirjasto WPF-sovelluksille. Kirjaston päätehtävänä on tarjota WPF-käyttöliittymälle paranneltuja komponentteja ja elementtejä. Kirjaston tavoitteena on pysyä mahdollisimman lähellä WPF-kontrollien alkuperäistä ulkoasua. Lähes jokaiselle WPF-kontrollille on paranneltu ulkoasumalli ja tyyli. Kirjasto tarjoaa mahdollisuuden vaihtaa väriteemaa kesken sovelluksen suorittamisen. WPF-kontrolleille on luotu laajennuksia, jotka tuovat lisäominaisuuksia kontrollia muuttamatta. Kirjasto sisältää myös muutaman uuden kontrollin, joita WPF-käyttöliittymäkirjastosta ei löydy. (AdonisUI [Viitattu 19.3.2019].)

5.3 Fody ja Costura

Kuvio 19. Fodyn sijainti käännösprosessissa

Fody on työkalu .NET-käännösten (*assemblies*) käsittelyyn (kuvio 19). Microsoftin kääntäjä kääntää .NET-komponenttikirjaston tukemat kielet Common Intermediate Languange -kieleen. CIL-kieli on hyvin lähellä konekieltä, mikä tekee siitä vaikealukuista ihmisille. Fody pyrkii tekemään käännöksien manipuloinnista yksinkertaista lisäosien avulla. (Fody [Viitattu 19.3.2019].) Fody ja CIL-kieli ovat konseptina hyvin laajoja, niitä ei käsitellä laajemmin tässä opinnäytetyössä.

Costura on lisäosa Fody-työkaluun. Costura-lisäosan avulla voidaan yhdistää .NET-käännökset sovellukseen sulautettuina (embedded) resursseina. Costura sulauttaa paikalliset käännökset resursseina sovellukseen. Sovelluksesta luotua käännöstä muokataan, että se löytää käännökset resursseista. (Costura [Viitattu 19.3.2019].) Costura-lisäosaa käytetään tässä työssä helpottamaan sovelluksen jakelua. Sovelluksen kääntämisen yhteydessä syntyvät käännökset saadaan pakattua yhteen sovelluksen suorittavaan tiedostoon (kuvio 20).

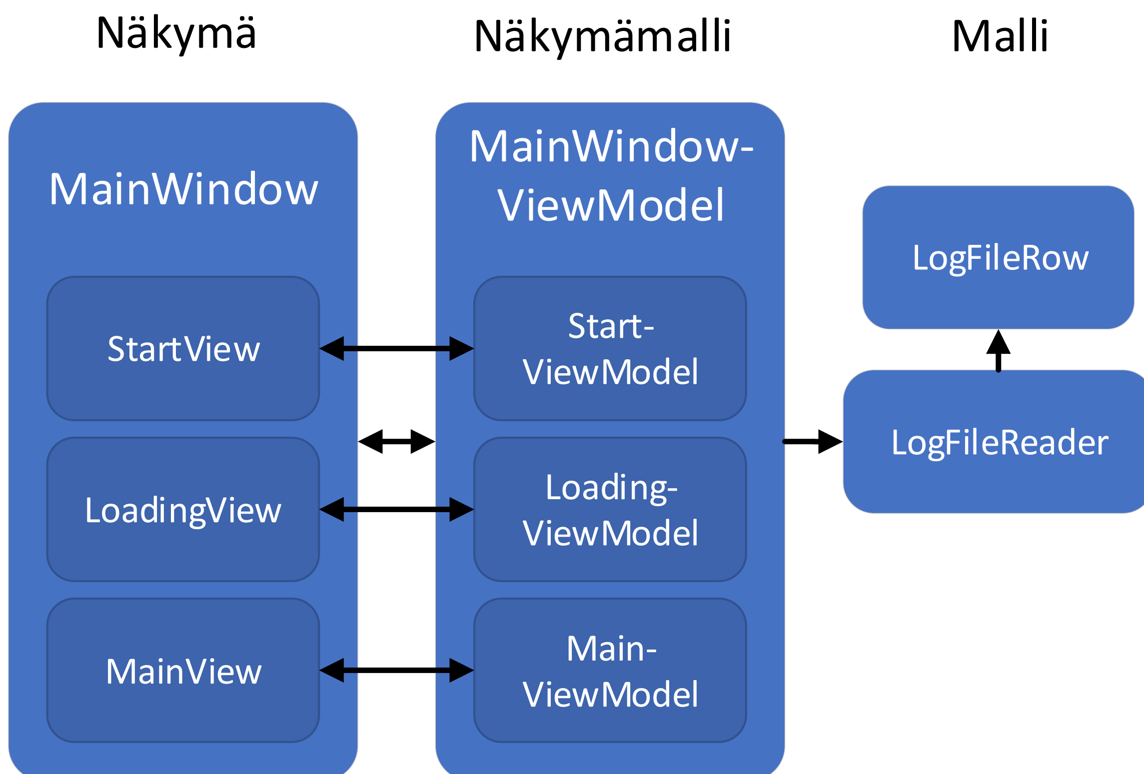


Kuvio 20. Costura käytännössä

Sovellus etsii käynnistyessään tarvitsemansa .dll-kirjastotiedostot. Sovellus ei toimi, jos yksikin sen tarvitsema tiedosto puuttuu. Sovellusta jakaessa ja siirtäessä täytyy varmistaa, että .dll-tiedostot siirtyvät mukana. Costura-lisäosan avulla sovellus voidaan jakaa yhtenä .exe-tiedostona.

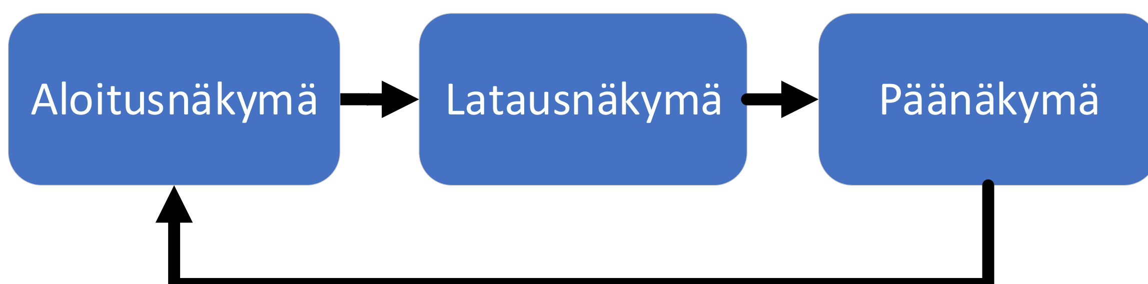
5.4 Sovelluksen arkkitehtuuri

Toteutettavalle sovellukselle laadittiin ennen toteuttamista arkkitehtuuri ja rakenne. Kuviossa 21 on esitetty sovelluksen rakenne:



Kuvio 21. Sovelluksen rakenne

Sovelluksen päänäkymä on MainWindow-ikkuna, joka sisältää näkymät StartView, LoadingView ja MainView. Päänäkymän näkymämalli on MainWindowViewModel, joka pitää sisällään näkymien näkymämallit StartViewModel, LoadingViewModel ja MainViewModel. Malli koostuu LogFileReader-apuluokasta ja LogFileRow-oliosta.



Kuvio 22. Sovelluksen käytön kulku

Sovelluksen käytön kulku toimii silmukassa (kuvio 22). Sovellus näyttää käynnistyessään StartView-näkymän, jossa käyttäjä antaa sovellukselle tarkasteltavat lokitiedostot. Isojen lokitiedostojen luku saattaa kestää, jolloin näytetään LoadingView-näkymä. Näkymä näyttää lokitiedostojen lukemisen tilanteen ja antaa mahdollisuuden keskeyttää niiden lukemisen. Mikäli lukeminen keskeytetään, näkymä vaihdetaan takaisin StartView-näkymään. Kun tiedostot saadaan luettua, siirrytään MainView-näkymään. Näkymä pitää sisällään luetut lokitiedostot sekä työkalut niiden tarkasteluun. Näkymästä voidaan palata takaisin StartView-näkymään.

MainWindowViewModel-näkymämalli on vastuussa näkymien vaihtelusta ja oikean näkymän näyttämisestä. StartViewModel ilmoittaa annetut lokitiedostot, LoadingViewModel ilmoittaa keskeytyksen ja MainViewModel ilmoittaa paluukäskystä. Malli ilmoittaa tiedostojen lukemisen tilanteen MainWindowViewModel-näkymämallille, joka päivittää tilanteen LoadingViewModel-näkymämallille. MainViewModel-näkymämalli sisältää esityslogiikan lokitiedostojen suodattamiseen ja etsimiseen.

Malli sisältää LogFileReader-apuluokan, joka käsittelee annetut lokitiedostot. Apuluokka pitää sisällään oikean logiikan lokitiedostojen prosessointiin. Lokitiedostot luetaan koneen paikalliselta tallennustilalta. Luettujen ja prosessoitujen lokitiedostojen tieto tallennetaan LogFileRow-olioihin.

6 TOTEUTUS

Sovelluksen toteuttaminen aloitettiin luomalla logiikka, joka lukee annetun lokitiedoston. MVVM-mallin mukaisessa sovelluksessa tämä logiikka sijaitsee mallissa. Logiikan rakentamiseksi perehdytään ensin luettaviin lokitiedostoihin. Lokitiedostoja on kahdessa eri muodossa: erillisiä log-tekstitiedostoja ja useammista log-tiedostoista koostuvia zip-pakettitiedostoja. Log-tyyppisten tiedostojen rakenne vaihtelee kahden eri rakenteen välillä.

```
20/3/2019;11:36:7;327;System 3;Error;Longer log file text example 1530
20/3/2019;11:36:7;327;System 7;Warning;Some other words 3617
20/3/2019;11:36:7;327;System 2;Error;Longer log file text example 1063
20/3/2019;11:36:7;327;System 6;Warning;Some other words 3167
20/3/2019;11:36:7;327;System 1;Error;Short log file text 588
20/3/2019;11:36:7;327;System 5;Info;Different description 2707
20/3/2019;11:36:7;327;System 0;Error;Short log file text 105
20/3/2019;11:36:7;327;System 4;Info;Sample log file text 2240
```

Kuva 35. Esimerkkilokitiedosto puolipisteillä

Kuvassa 35 lokitiedoston rivien tiedot on eroteltu puolipisteillä. Yksi rivi tekstitiedostossa esittää yhtä tapahtumaa. Jokainen lokitiedoston rivi seuraa samaa rakennetta. Rivin pitää sisällään tiedon päivänmäärästä, kellonajasta, millisekunneista, lokitiedon tuottaneesta järjestelmästä, lokitiedon tyypistä ja kuvauksesta.

```
19/3/2019	10:28:13	589	System 231	Error	Short log file text 231
19/3/2019	10:28:13	599	System 1688	Info	Different description 1688
19/3/2019	10:28:13	609	System 576	Error	Longer log file text example 576
19/3/2019	10:28:13	620	System 580	Error	Longer log file text example 580
19/3/2019	10:28:13	630	System 1221	Info	Sample log file text 1221
19/3/2019	10:28:13	651	System 943	Info	Sample log file text 943
19/3/2019	10:28:13	662	System 1599	Info	Different description 1599
19/3/2019	10:28:13	673	System 1312	Info	Sample log file text 1312
```

Kuva 36. Esimerkkilokitiedosto sarkainmerkeillä

Osassa lokitiedostoissa rivien tieto on eroteltu sarkainmerkillä (kuva 36). Sarkainmerkillä erotettujen lokitiedostojen rivit seuraavat muuten samaa rakennetta kuin puolipisteellä erotetut.

6.1 Malli

Mallin rakentaminen aloitetaan luomalla olioluokka (kuva 37), johon varastoidaan lokitiedostojen tiedot. Yksi olio sisältää samat tiedot kuin yksi rivi lokitiedostosta. Lokitiedostojen sisältämä päivänmäärä, aika ja millisekunnit voidaan käsitellä järkevämpään DateTime-luokkaan. Rivin muut tiedot pidetään merkkijonoina.

```
public class LogFileRow
{
    public DateTime LogDateTime { get; set; }
    public string LogSystem { get; set; }
    public string LogType { get; set; }
    public string LogText { get; set; }
    public LogFileRow() { }
}
```

Kuva 37. Tiedon varastointiolio luokka

Mallin luokille luodaan rajapintaluokka (*interface class*), joka pakottaa mallin logiikan sisältävän luokan toteuttavan rajapinnan määrittelemät metodit (kuva 38). Rajapintojen avulla voidaan varmistaa näkymämallin toiminnan jatkuminen, vaikka mallin logiikka muuttuisi täysin.

```
interface ILogFileReader
{
    int TotalFilesCount(List<string> paths);
    Task<IEnumerable<LogFileRow>> LogRowsFromSource(string source,
                                                   IProgress<int> progress,
                                                   CancellationToken token);
}
```

Kuva 38. Mallin rajapintaluokka

Sovelluksen mallin vaaditaan toteuttavan kaksi metodia. TotalFilesCount-metodille annetaan parametrina lista lokitiedostojen sijainneista ja se palauttaa luettavien tiedostojen lukumäärän. Metodia käytetään näyttämään käyttäjälle tiedostojen lukemisen edistyminen latausnäkymässä. Toinen on LogRowsFromSource-metodi, joka palauttaa asynkronisesti suoritettavan tehtävän (*Task*). Tehtävä palauttaa annetusta sijainnista luetusta tiedostosta listan, joka koostuu LogFileRow-olioista. Metodille annetaan sijainnin lisäksi parametreinä IProgress-rajapinta ja CancellationToken-luokka. IProgress-rajapinta ilmoittaa tiedostojen lukemisen etenemisestä ja CancellationToken-luokan avulla voidaan keskeyttää tiedostojen lukeminen.

```
public int TotalFilesCount(List<string> paths)
{
    int count = 0;
    foreach (string path in paths)
    {
        if (path.EndsWith(".log"))
        {
            count++;
        }
        else if (path.EndsWith(".zip"))
        {
            using (ZipArchive archive = ZipFile.OpenRead(path))
            {
                List<ZipArchiveEntry> entries = archive.Entries
                    .Where(o => o.FullName.EndsWith(".log")).ToList();
                count += entries.Count;
            }
        }
    }
    return count;
}
```

Kuva 39. TotalFilesCount-metodi

Kuvassa 39 toteutettu TotalFilesCount-metodi käy läpi listan jokaisen tiedostosijainnin. Jokainen log-tiedosto lasketaan yhdeksi tiedostoksi ja zip-tiedostojen tapauksessa lasketaan jokainen tiedoston sisältämä log-tiedosto. Sijainnit läpi käytyään metodi palauttaa tiedostojen yhteen lasketun lukumäärän.

```
public async Task<IEnumerable<LogFileRow>> LogRowsFromSource(
    string source,
    IProgress<int> progress,
    CancellationToken token)
{
    List<LogFileRow> rows = new List<LogFileRow>();
    int sourceType = EvaluateSource(source);
    if (source.EndsWith(".log"))
    {
        rows = await Task.Run(() => ReadAllLinesLogFile(
            source, sourceType,
            progress, token));
    }
    if (source.EndsWith(".zip"))
    {
        rows = await Task.Run(() => ReadAllLinesZip(
            source, sourceType,
            progress, token));
    }
    return rows;
}
```

Kuva 40. LogRowsFromSource-metodi

Kuvan 40 LogRowsFromSource-metodi evaluoi lähdetiedoston rakenteen EvaluateSource-metodilla. Tiedoston lukemiseen käytettävä metodi päätellään tiedostotyypistä.

```
private int EvaluateSource(string source)
{
    if (source.EndsWith(".log"))
    {
        string firstLine = File.ReadLines(source).First();
        if (firstLine.Split(';').Length == 6)
        {
            return 1;
        }
        if (firstLine.Split('\t').Length == 6)
        {
            return 2;
        }
    }
    else if (source.EndsWith(".zip"))
    {
        using (ZipArchive archive = ZipFile.OpenRead(source))
        {
            ZipArchiveEntry entry = archive.Entries
                .Where(o => o.FullName.EndsWith(".log")).First();
            using (StreamReader reader = new StreamReader(entry.Open()))
            {
                string currentLine = reader.ReadLine();
                if (currentLine.Split(';').Length == 6)
                {
                    return 1;
                }
                if (currentLine.Split('\t').Length == 6)
                {
                    return 2;
                }
            }
        }
    }
    return 0;
}
```

Kuva 41. Tiedoston rakenteen evaluointimetodi

EvaluateSource-metodi (kuva 41) päättelee lähdetiedoston rakenteen rivien jakomerkillä. Koska jokaisen lokitiedoston rivi sisältää saman määrän erotusmerkkejä, voidaan tiedoston rakenne tunnistaa erotusmerkkien lukumäärällä. Lokitiedostoista luetaan ensimmäinen rivi ja tarkistetaan, voidaanko se jakaa puolipisteellä vai sarkainmerkillä. Log-tyypin tiedostosta ensimmäinen rivi voidaan lukea suoraan, mutta zip-tiedosto täytyy ensin avata.

```
private List<LogFileRow> ReadAllLinesLogFile(string source,
    int type, IProgress<int> progress, CancellationToken token)
{
    List<LogFileRow> rows = new List<LogFileRow>();
    using (StreamReader reader = new StreamReader(new FileStream(
        source, FileMode.Open, FileAccess.Read, FileShare.Read,
        4096, FileOptions.Asynchronous | FileOptions.SequentialScan)))
    {
        string currentLine;
        while ((currentLine = reader.ReadLine()) != null)
        {
            if (token.IsCancellationRequested)
            {
                break;
            }
            if (type == 1)
            {
                rows.Add(LogRowFromLineTypeOne(currentLine));
            }
            if (type == 2)
            {
                rows.Add(LogRowFromLineTypeTwo(currentLine));
            }
        }
    }
    progress.Report(1);
    return rows;
}
```

Kuva 42. Log-tiedostosta luku

Log-tiedostojen lukumetodi on esitetty kuvassa 42. Tiedostot luetaan rivi kerrallaan ja rivin jäsentämisestä (*parse*) huolehtii erillinen metodi tiedoston rakenteesta riippuen. Tiedostoon avataan ”vain luku” -yhteys. Luetut rivit lisätään listaan, joka palautetaan metodin lopuksi. Keskeyttäminen tarkastetaan jokaisella kierroksilla ja tiedostonluku lopetetaan, jos keskeyttämistä on pyydetty. Eteneminen ilmoitetaan metodin lopussa ennen listan palauttamista.

```
private List<LogFileRow> ReadAllLinesZip(string source,
    int type, IProgress<int> progress, CancellationToken token)
{
    List<LogFileRow> rows = new List<LogFileRow>();
    using (ZipArchive archive = ZipFile.OpenRead(source))
    {
        List<ZipArchiveEntry> entries = archive.Entries
            .Where(o => o.FullName.EndsWith(".log")).ToList();
        foreach (ZipArchiveEntry entry in entries)
        {
            if (token.IsCancellationRequested)
            {
                break;
            }
            using (StreamReader reader = new StreamReader(entry.Open()))
            {
                string currentLine;
                while ((currentLine = reader.ReadLine()) != null)
                {
                    if (token.IsCancellationRequested)
                    {
                        break;
                    }
                    if (type == 1)
                    {
                        rows.Add(LogRowFromLineTypeOne(currentLine));
                    }
                    if (type == 2)
                    {
                        rows.Add(LogRowFromLineTypeTwo(currentLine));
                    }
                }
            }
            progress.Report(1);
        }
    }
    return rows;
}
```

Kuva 43. Zip-tiedostosta luku

Zip-tiedostojen käsittelymetodi on esitetty kuvassa 43. Log-tiedostojen lukeminen zip-tiedostosta vaatii zip-tiedoston avaamisen. Zip-tiedostosta käydään läpi jokainen log-tyyppinen tiedosto. Lukemisen keskeyttäminen tarkastetaan ennen jokaisen tiedoston lukemisen aloittamista ja jokaisen rivin kohdalla. Lukemisen eteneminen ilmoitetaan jokaisen tiedoston lukemisen jälkeen.

```
private LogFileRow LogRowFromLineTypeOne(string line)
{
    string[] splitLine = line.Split(';');
    return new LogFileRow
    {
        LogDateTime = SetLogDateTime(splitLine[0], splitLine[1], splitLine[2]),
        LogSystem = splitLine[3],
        LogType = splitLine[4],
        LogText = splitLine[5]
    };
}
private LogFileRow LogRowFromLineTypeTwo(string line)
{
    string[] splitLine = line.Split('\t');
    return new LogFileRow
    {
        LogDateTime = SetLogDateTime(splitLine[0], splitLine[1], splitLine[2]),
        LogSystem = splitLine[3],
        LogType = splitLine[4],
        LogText = splitLine[5]
    };
}
```

Kuva 44. Lokirivien käsittelymetodit

Lokirivien jäsentämiselle on kaksi erilaista metodia eri erottelumerkeille (kuva 44). Luettu lokirivi jaetaan erottelumerkin mukaan palasiksi, joista luodaan olio. Päivämäärän jäsentämiseen käytetään erillistä metodia, joka palauttaa DateTime-luokan olion.

```
private DateTime SetLogDateTime(string date, string time, string ms)
{
    string[] splitDate = date.Split('/');
    string[] splitTime = time.Split(':');
    return new DateTime(
        int.Parse(splitDate[2]), int.Parse(splitDate[1]), int.Parse(splitDate[0]),
        int.Parse(splitTime[0]), int.Parse(splitTime[1]), int.Parse(splitTime[2]),
        int.Parse(ms));
}
```

Kuva 45. Päivänmäärän jäsennysmetodi

Ajan jäsennysmetodille (kuva 45) annetaan parametreinä lokirivistä jäsennetyt päivämäärä, aika ja millisekunnit. Päivämäärä ja aika jaetaan DateTime-luokan rakentajalle sopiviksi. Metodi palauttaa luodun DateTime-olion.

6.2 Pääikkuna

Sovelluksessa on yksi Window-elementti, joka on sovelluksen pääikkuna. Pääikkuna sisältää sovelluksen jokaisen näkymän. Pääikkuna yhdistetään päänäkymämalliin, joka pitää sisällään logiikan näkymien vaihteluun. Näkymämalli sisältää logiikan lisäksi viittauksen pääikkunan sisältämien näkymien näkymämalleihin. Malli on yhdistetty päänäkymämalliin.

6.2.1 Näkymä

Pääikkunan näkymässä määritellään sovelluksen muut näkymät. Pääikkunalle määritelty tyyli periytyy alemmille näkymille.

```
<Window xmlns:viewModels="clr-namespace:LogFileViewer.ViewModel"
        xmlns:views="clr-namespace:LogFileViewer.View"
        xmlns:converters="clr-namespace:LogFileViewer.Utilities.Converters"
        Style="{StaticResource CustomWindowStyle}"
        <...>
        >

    <...>

</Window>
```

Kuva 46. Pääikkunan attribuutit

Pääikkunan Window-elementissä (kuva 46) määritellään näkymämallien, näkymien ja muuntimien sijainti. Ikkunalle määritellään myös mukautettu tyyli, jolla muutetaan ikkunan ulkoasua ja otetaan AdonisUI-käyttöliittymä käyttöön.

```
<Style x:Key="CustomWindowStyle" TargetType="{x:Type Window}"
       BasedOn="{StaticResource {x:Type Window}}">
    <...>
</Style>
```

Kuva 47. Ikkunan tyyli

Ikkunan tyylissä (kuva 47) asetetaan BasedOn-attribuutin arvoksi viittaus AdonisUI-kirjaston ikkunatyyliin. AdonisUI-kirjaston ikkunatyyli on yleinen kaikki ikkunat ylikirjoittava tyyli. Asettamalla tyylin BasedOn-attribuutti AdonisUI-kirjaston ikkunatyyliin voidaan luoda ikkunalle oma tyyli hyödyntämällä AdonisUI-kirjaston tyylejä.

```
<Window.DataContext>
    <viewModels:ParentViewModel />
</Window.DataContext>
```

Kuva 48. Pääikkunan näkymämallin määrittely

Ikkunalle määritellään näkymämalli (kuva 48) ”näkymä ensin” -asetelman tavoin. Näkymä löytää näkymämallin sijainnin Window-elementin attribuutti määrittelyjen avulla.

```
<Window.Resources>
    <converters:BoolToBoolConverter x:Key="InvertBool" True="False"
                                    False="True" Null="False" />
    <converters:BoolToBoolConverter x:Key="NullBool" True="False"
                                    False="False" Null="True" />
    <converters:BoolToBoolConverter x:Key="BaseBool" True="True"
                                    False="False" Null="False" />
</Window.Resources>
```

Kuva 49. Pääikkunan muuntimien määrittely

Näkymässä määritellään yhdestä muuntimesta kolme erilaista versiota (kuva 49). Eri versioiden ominaisuuksille annetaan eri arvot. Muuntimia käytetään tahdotun näkymän näyttämiseen.

```
public class BaseBooleanConverter<T> : IValueConverter
{
    public T True { get; set; }
    public T False { get; set; }
    public T Null { get; set; }

    public virtual object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        if (value == null)
            return Null;
        return (bool)value ? True : False;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture) => Binding.DoNothing;
}
public sealed class BoolToBoolConverter : BaseBooleanConverter<bool>
{
}
```

Kuva 50. Muuntimen toteutus

Pohjamuunnin BaseBooleanConverter-luokassa (kuva 50) paljastetaan kolme ominaisuutta, mikä mahdollistaa saman muuntimen käytön eri tarkoituksiin. Muuntajan ominaisuuksille ei ole määritelty mitään erityistä tyyppiä. Pohjamuunnin-luokasta voidaan periä uusi muunnin, joka asettaa ominaisuuksille tahtomansa tyypin.

```
<Grid>
    <views:StartViewUserControl DataContext="{Binding StartViewModel}"
                                Style="{DynamicResource FadingControl}"
                                IsHitTestVisible="{Binding
                                 DataContext.LogsProcessed,
                                 ElementName=ParentWindow,
                                 Converter={StaticResource InvertBool}}" />

    <views:LoadingViewUserControl DataContext="{Binding LoadingViewModel}"
                                  Style="{DynamicResource FadingControl}"
                                  IsHitTestVisible="{Binding
                                   DataContext.LogsProcessed,
                                   ElementName=ParentWindow,
                                   Converter={StaticResource NullBool}}" />

    <views:MainViewUserControl DataContext="{Binding MainViewModel}"
                               Style="{DynamicResource FadingControl}"
                               IsHitTestVisible="{Binding
                                DataContext.LogsProcessed,
                                ElementName=ParentWindow,
                                Converter={StaticResource BaseBool}}" />
</Grid>
```

Kuva 51. Näkymät pääikkunassa

Sovelluksen näkymät (kuva 51) ovat samassa tasossa Grid-elementissä. Jotta näkymät eivät sekoittaisi toisiaan, käytetään näkymien IsHitTestVisibile-ominaisuutta. Ominaisuus on boolean-muuttuja. Muuttujan arvon ollessa tosi näkymän kanssa voi käyttäjä olla vuorovaikutuksessa ja päinvastoin. Sidottaessa tieto IsHitTestVisible-ominaisuuteen arvolle määritellään käytettävän ominaisuuden sijainti ja muuntaja. Samaa ominaisuutta hyödyntämällä eri muuntimien avulla saadaan haluttu näkymä näkyville. Näkymille asetetaan myös mukautettu tyyli ja näkymämallit. Pääikkunan näkymämalli sisältää sovelluksen muiden näkymien näkymämallit, jotka tietosidotaan päänäkymässä näkymiin.

```
<Style x:Key="FadingControl">
    <Style.Triggers>
        <DataTrigger Binding="{Binding IsHitTestVisible,
                      RelativeSource={RelativeSource Self}}"
                     Value="False">
            <...>
        </DataTrigger>
    </Style.Triggers>
</Style>
```

Kuva 52. Näkymien tyyli

Yleisessä FadingControl-tyylissä (kuva 52) on laukaisin, joka suorittaa yksinkertaisen animaation. Laukaisin on sidottu tyyliä käyttävän elementin IsHitTestVisible-ominaisuuteen. Tyyliä voidaan käyttää animoimaan mikä tahansa elementti elementin IsHitTestVisible-ominaisuudella.

6.2.2 Näkymämalli

Päänäkymämalli sisältää sovelluksen muut näkymämallit. Näkymämallia käytetään hallitsemaan koko sovelluksen tilaa ja yhdistämään malliin.

```
private ILogFileReader _logFileReader;

private StartViewModel _startViewModel;
public StartViewModel StartViewModel
{
    get => _startViewModel;
    set => Set(ref _startViewModel, value);
}
private LoadingViewModel _loadingViewModel;
public LoadingViewModel LoadingViewModel
{
    get => _loadingViewModel;
    set => Set(ref _loadingViewModel, value);
}
private MainViewModel _mainViewModel;
public MainViewModel MainViewModel
{
    get => _mainViewModel;
    set => Set(ref _mainViewModel, value);
}
private bool? _logsProcessed = false;
public bool? LogsProcessed
{
    get { return _logsProcessed; }
    set { Set(ref _logsProcessed, value); }
}
```

Kuva 53. Pääikkunanäkymämallin ominaisuudet

Pääikkunanäkymämallissa (kuva 53) on ominaisuuksina muut näkymämallit, LogsProcessed-muuttujan ja viittauksen mallin rajapintaan. LogsProcessed-muuttujaa käytetään vaihtelemaan näkymien välillä.

```
public ParentViewModel()
{
    _logFileReader = new LogFileReader();

    StartViewModel = new StartViewModel(this);
    LoadingViewModel = new LoadingViewModel(0);
    MainViewModel = new MainViewModel(this);
}
```

Kuva 54. Päänäkymämallin rakentaja

Pääikkunannäkymämallin rakentajassa (kuva 54) alustetaan malli sekä muut näkymämallit. Aloitus- ja päänäkymämalleille annetaan viittaus pääikkunanäkymämalliin.

```
public async void ReadLogs(List<string> paths)
{
    if (paths.Count > 0)
    {
        <...>
    }
}
```

Kuva 55. Lokien lukumetodi

Pääikkunannäkymämallissa sijaitsee lokien lukemisesta vastaava metodi (kuva 55). Lokitiedostot luetaan asynkronisesti, mikä vaatii metodille async-määrittelyn. Metodille annetaan parametrinä lista luettavista tiedostosijainneista ja aluksi metodi tarkistaa niiden määrän.

```
<...>
LogsProcessed = null;
LoadingViewModel =
    new LoadingViewModel(_logFileReader.TotalFilesCount(paths));
IProgress<int> progressStatus = new Progress<int>(status =>
    { LoadingViewModel.CurrentProgress += status; });

<...>
```

Kuva 56. Latausnäkymän päivittäminen

Lokien lukumetodin alussa (kuva 56) asetetaan LogsProcessed-muuttaja arvoon null, mikä piilottaa aloitusnäkymän ja näyttää latausnäkymän. Latausnäkymämalli alustetaan luettavien tiedostojen lukumäärällä, jota käytetään latausnäkymän edistymispalkissa. IProgress-rajapinnasta luodaan uusi Progress-olio, jonka ilmoitukset lisätään latausnäkymän edistymispalkkiin.

```
<...>
List<Task> taskList = new List<Task>();
foreach (string path in paths)
{
    taskList.Add(Task.Run(() =>
    _logFileReader.LogRowsFromSource(path, progressStatus,
    LoadingViewModel.cancelToken)));
}
await Task.WhenAll(taskList);

<...>
```

Kuva 57. Mallin kanssa kommunikointi

Jokaisesta luettavasta tiedostosta luodaan uusi tehtävä (kuva 57). Tehtäville annetaan parametreina tiedostosijainti, Progress-olio ja latausnäkymän CancellationToken. Metodin suorittaminen jatkuu, kunnes jokainen tehtävä on joko valmis tai keskeytetty.

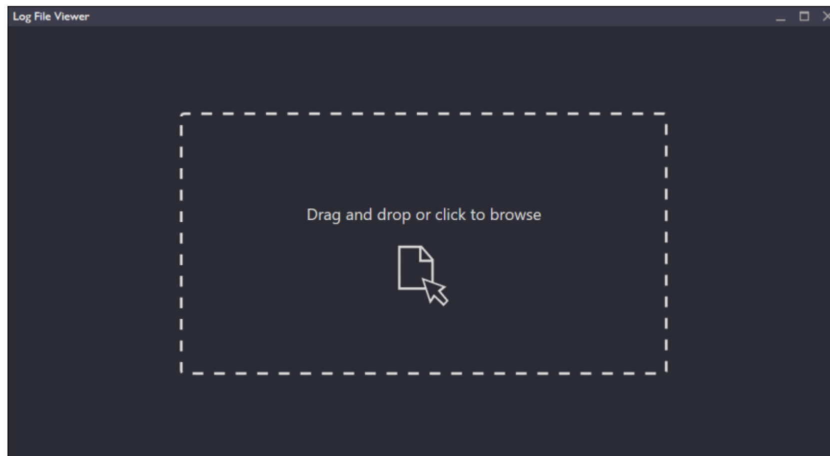
```
<...>
if (!LoadingViewModel.cancelToken.IsCancellationRequested)
{
    MainViewModel.LogRows.Clear();
    foreach (Task<IEnumerable<LogFileRow>> task in taskList)
    {
        if (task.IsCompleted)
        {
            foreach (LogFileRow row in task.Result)
            {
                MainViewModel.LogRows.Add(row);
            }
        }
    }
    MainViewModel.CreateFilters();
    LogsProcessed = true;
}
else
{
    LogsProcessed = false;
}
<...>
```

Kuva 58. Loppukäsittely

Tehtävien valmistuttua (kuva 58) tarkastetaan, onko keskeyttämistä pyydetty. Mikäli keskeytys on pyydetty, palataan takaisin aloitusnäkymään. Jos keskeytystä ei ole pyydetty, lisätään luettujen lokitiedostojen rivit päänäkymään, luodaan suodattimet ja näytetään päänäkymä.

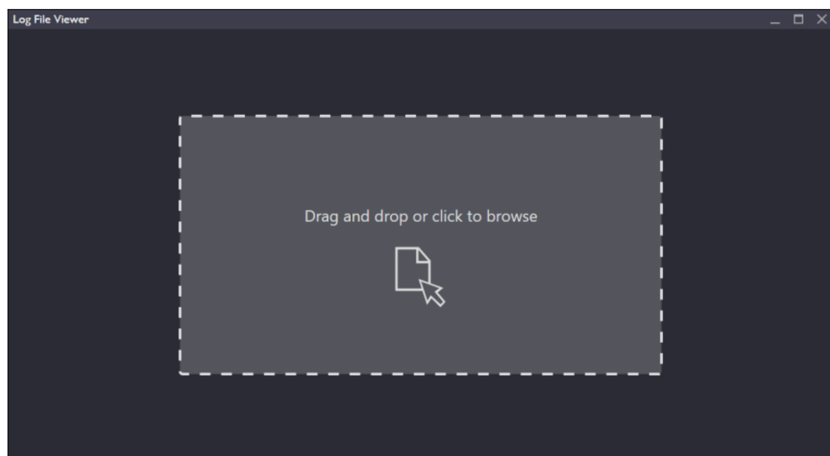
6.3 Aloitus

Aloitusnäkymässä annetaan luettavat lokitiedostot. Sovellus tukee myös ”vedä ja pudota” -tyylin tiedostojen antamista.



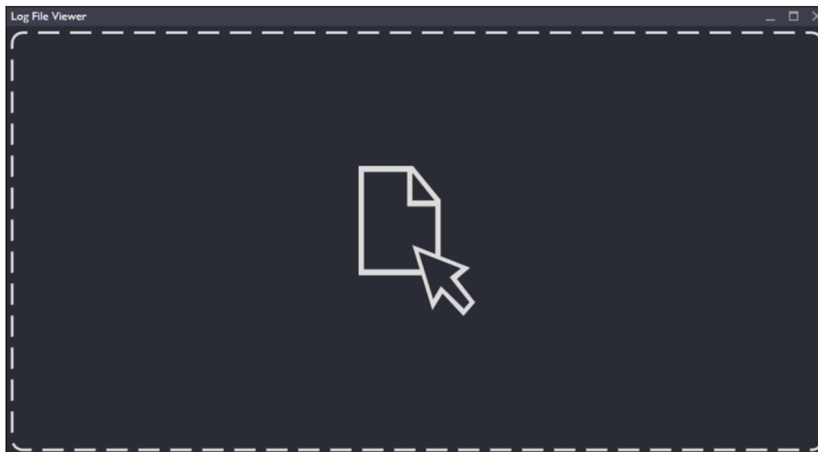
Kuva 59. Aloitusnäkymä

Sovelluksen aloitusnäkymä (kuva 59) sisältää yksinkertaisen ohjeistuksen tiedostojen antamiselle. Lokitiedostot voidaan antaa napauttamalla näkymän keskiosaa tai vetämällä ja pudottamalla.



Kuva 60. Aloitusnäkymä animaatio

Hiiren vieminen näkymän keskiosaan animoi napautettavan alueen (kuva 60). Keskiosasta napauttamalla aukeaa Windows-käyttöjärjestelmän tiedostovalintaikkuna.



Kuva 61. Vedä ja pudota -animaatio

Näkymä muuttuu (kuva 61), kun sovelluksen ikkunaan vedetään tiedosto. Tiedosto luetaan käyttäjän pudottaessa se ikkunaan.

6.3.1 Näkymä

Aloitusnäkymä ilmoittaa näkymämallille tiedostojenselaamisen sekä vedä ja pudota -tapahtumat. Näkymä näyttää vedä ja pudota -tapahtumien sattuessa toisen näkymän.

```
<UserControl xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
             xmlns:mvvm="http://www.galasoft.ch/mvvmlight"
             xmlns:converters="clr-namespace:LogFileViewer.Utilities.Converters"
            <...>
             >

    <...>

</UserControl>
```

Kuva 62. Näkymän attribuuttien määrittely

Näkymän UserControl-elementille (kuva 62) määritellään muuntimien sijainti sekä MVVM Light Toolkit -kirjaston EventToCommand-luokan vaatimat luokat.

```
<Grid AllowDrop="True" Background="Transparent">

    <i:Interaction.Triggers>
        <i:EventTrigger EventName="PreviewDragEnter">
            <mvvm:EventToCommand Command="{Binding DragDropCommand, Mode=OneWay}"
                                 PassEventArgsToCommand="True"/>
        </i:EventTrigger>
        <i:EventTrigger EventName="PreviewDragLeave">
            <mvvm:EventToCommand Command="{Binding DragDropCommand, Mode=OneWay}"
                                 PassEventArgsToCommand="True"/>
        </i:EventTrigger>
        <i:EventTrigger EventName="PreviewDrop">
            <mvvm:EventToCommand Command="{Binding DragDropCommand, Mode=OneWay}"
                                 PassEventArgsToCommand="True"/>
        </i:EventTrigger>
    </i:Interaction.Triggers>

    <...>

</Grid>
```

Kuva 63. Vedä ja pudota -määrittely

Kuvassa 63 on määritelty vedä ja pudota -ominaisuuden tapahtumat komennoiksi. Näkymän Grid-elementille määritetään AllowDrop-attribuutti, mikä mahdollistaa vedä ja pudota -tapahtumien käsittelyn. MVVM Light Toolkit -kirjaston EventToCommand-luokan avulla tapahtumat saadaan yhdistettyä komentoihin näkymämallissa.

```
<UserControl.Resources>
    <converters:BoolToBoolConverter x:Key="InvertBool"
                                    True="False" False="True" />
</UserControl.Resources>
```

Kuva 64. Muuntimen määrittely

Näkymässä käytetään yhtä muunninta (kuva 64), joka kääntää boolean-muuttajan arvon päinvastoin. Muuntajaa käytetään vaihtelemaan aloitusnäkymän sekä vedä ja pudota -näkymän välillä.

```
<Grid Margin="256 128" Style="{DynamicResource FadingControl}"
      IsHitTestVisible="{Binding DisplayDropZone,
       Converter={StaticResource InvertBool}}">

    <Rectangle Style="{DynamicResource AnimatedRectangleStyle}">
        <i:Interaction.Triggers>
            <i:EventTrigger EventName="MouseDown">
                <mvvm:EventToCommand Command="{Binding BrowseForFilesCommand,
                                     Mode=OneWay}"/>
            </i:EventTrigger>
        </i:Interaction.Triggers>
    </Rectangle>

    <StackPanel VerticalAlignment="Center" IsHitTestVisible="False">
        <TextBlock Margin="0, 32" FontSize="24" HorizontalAlignment="Center"
                   Foreground="{DynamicResource SemiWhiteBrush}"
                   Text="Drag and drop or click to browse" />
        <ContentControl Width="75" Height="90"
                        Content="{DynamicResource DragDropIcon}" />
    </StackPanel>

</Grid>
```

Kuva 65. Napautettavan alueen määrittely

Aloitusnäkymä koostuu Grid-elementistä, joka pitää sisällään napautettavan alueen (kuva 65) luovan Rectangle-elementin ja ohjetekstin sisältävän StackPanel-elementin. Rectangle-elementille on määritelty hiireen reagoiva tyyli ja hiiren napautuksen tapahtumasta komennoksi muuntava EventToCommand.

```
<Grid Style="{DynamicResource FadingControl}"
      IsHitTestVisible="{Binding DisplayDropZone}" >
    <ContentControl Content="{DynamicResource DragDropIcon}"
                    Width="175" Height="225"
                    VerticalAlignment="Center"
                    HorizontalAlignment="Center" />
    <Rectangle Style="{DynamicResource StaticRectangleStyle}" />
</Grid>
```

Kuva 66. Vedä ja pudota -näkymä

Vedä ja pudota -näkymä (kuva 66) koostuu Grid-elementistä, joka sisältää kuvan sisältävän ContentControl-elementin ja Rectangle-elementin. Rectangle-elementille on määritelty yksinkertainen tyyli.

6.3.2 Näkymämalli

Aloitusnäkymämalli sisältää logiikan vedä ja pudota -tapahtumien käsittelylle. Tapahtumien yhteydessä asetetaan muuttuja, joka kertoo aloitusnäkymälle vedä ja pudota -näkymän tilan.

```
private ParentViewModel _parent;

public StartViewModel(ParentViewModel parent)
{
    _parent = parent;
}
```

Kuva 67. Aloitusnäkymämallin rakentaja

Aloitusnäkymämallin rakentajalle (kuva 67) annetaan parametrina viite pääikkunan näkymämalliin. Viitettä käytetään ilmoittamaan pääikkunan näkymämallille näkymässä annetut lokitiedostot.

```
private bool _displayDropZone = false;
public bool DisplayDropZone
{
    get => _displayDropZone;
    set => Set(ref _displayDropZone, value);
}

public RelayCommand BrowseForFilesCommand =>
    new RelayCommand(BrowseForFilesAction);

public RelayCommand<DragEventArgs> DragDropCommand =>
    new RelayCommand<DragEventArgs>(DragDropAction);
```

Kuva 68. Aloitusnäkymämallin ominaisuudet

Näkymämalli (kuva 68) sisältää DisplayDropZone-muuttajan, BrowseForFiles-komennon ja DragDrop-komennon. DisplayDropZone-muuttujan avulla näytetään vedä ja pudota -näkymä. BrowseForFiles-komentoa käytetään avaamaan tiedostojenvalintaikkuna ja DragDrop-komento käsittelee vedä ja pudota -tapahtumat.

```
private void BrowseForFilesAction()
{
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Multiselect = true;
    ofd.Filter = "Zip/Log (*.zip, *.log)|*.zip;*.log";
    bool? result = ofd.ShowDialog();

    if (result == true && result.HasValue)
        _parent.ReadLogs(ofd.FileNames.ToList());
}
```

Kuva 69. Tiedostojen valintametodi

BrowseForFiles-metodi (kuva 69) avaa Windows-käyttöjärjestelmän tiedostojenvalintaikkunan, asettaa sille suodattimet ja ottaa vastaan ikkunan tuloksen. Jos ikkunassa valittiin tiedostoja, lähetetään tiedostot pääikkunan näkymämallille.

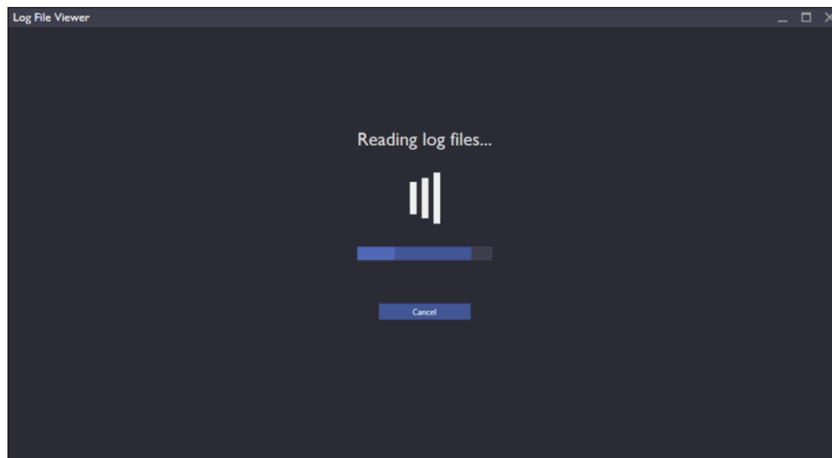
```
private void DragDropAction(DragEventArgs e)
{
    if (e.RoutedEvent.Name == "PreviewDragEnter")
        DisplayDropZone = true;
    if (e.RoutedEvent.Name == "PreviewDragLeave")
        DisplayDropZone = false;
    if (e.RoutedEvent.Name == "PreviewDrop")
    {
        if (e.Data.GetDataPresent(DataFormats.FileDrop))
        {
            string[] paths = (string[])e.Data.GetData(DataFormats.FileDrop);
            List<string> filterdPaths = paths.ToList().Where(x =>
            x.EndsWith(".zip") || x.EndsWith(".log")).ToList();
            _parent.ReadLogs(filterdPaths);
            DisplayDropZone = false;
        }
    }
    e.Handled = true;
}
```

Kuva 70. Vedä ja pudota -käsittelymetodi

DragDrop-metodi (kuva 70) näyttää tai piilottaa vedä ja pudota -näkymän ja ilmoittaa pääikkunan näkymämallille annetut tiedostot. Annetuista tiedostoista suodatetaan pois tiedostot, jotka eivät ole zip- tai log-tiedostoja.

6.4 Latausnäkymä

Latausnäkymä (kuva 71) näytetään aloitus- ja päänäkymän välissä. Näkymä sisältää tilasta ilmaisevan tekstin, animaation, edistymispalkin ja keskeyttämispainikkeen. Edistymispalkki ilmaisee lokitiedostojen lukemisen tilanteen. Keskeyttämispainikkeella voidaan keskeyttää tiedostojen luku.



Kuva 71. Latausnäkymä käyttöliittymä

6.4.1 Näkymä

Latausnäkymä ilmoittaa näkymämallille keskeyttämisestä. Näkymä saa tilanne- ja edistymisilmoituksia näkymämallilta.

```
<UserControl xmlns:utilities="clr-namespace:LogFileViewer.Utilities.Other"
             xmlns:adonisUi="clr-namespace:AdonisUI;assembly=AdonisUI"
             <...>
             >
      <...>

</UserControl>
```

Kuva 72. Latausnäkymän attribuutit

Latausnäkymän UserControl-elementille (kuva 72) määritellään apuluokkien ja AdonisUI-kirjaston sijainti. AdonisUI-kirjaston sijainti täytyy määritellä, että kirjaston tyylejä voidaan asettaa elementeille.

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
    <TextBlock FontSize="28" HorizontalAlignment="Center"
               Foreground="{DynamicResource SemiWhiteBrush}"
               Text="{Binding Status}" FontFamily="Gill Sans MT"/>
    <ContentControl Margin="0 32" Width="80" Height="80"
                    ContentTemplate="{DynamicResource
                    {x:Static adonisUi:Templates.LoadingBars}}"
                    Foreground="{DynamicResource
                    {x:Static adonisUi:Brushes.ForegroundBrush}}" />
    <ProgressBar Width="200" Height="20" Minimum="0" Maximum="{Binding MaxProgress}"
                 utilities:ProgressBarSmoother.SmoothValue="{Binding CurrentProgress}"/>
    <Button Margin="32 64" Content="Cancel" Command="{Binding CancelCommand}"
            Style="{DynamicResource {x:Static adonisUi:Styles.AccentButton}}" />
</StackPanel>
```

Kuva 73. Latausnäkymä määrittely

Latausnäkymä (kuva 73) koostuu StackPanel-elementistä, joka pitää sisällään TextBlock-, ContentControl-, ProgressBar- ja Button-elementit. ContentControl-elementin sisällöksi asetetaan AdonisUI-kirjaston sisältämä animoitu kuva. ProgressBar-elementin tilanne animoidaan ProgressBarSmoother-apuluokalla. Button-elementille asetetaan AdonisUI-kirjaston sisältämä AccentButton-tyyli.

```
public class ProgressBarSmoother
{
    public static double GetSmoothValue(DependencyObject obj)
        => (double)obj.GetValue(SmoothValueProperty);

    public static void SetSmoothValue(DependencyObject obj, double value)
        => obj.SetValue(SmoothValueProperty, value);

    public static readonly DependencyProperty SmoothValueProperty =
        DependencyProperty.RegisterAttached("SmoothValue", typeof(double),
            typeof(ProgressBarSmoother), new PropertyMetadata(0.0, Changing));

    private static void Changing(DependencyObject d,
        DependencyPropertyChangedEventArgs e)
    {
        var anim = new DoubleAnimation((double)e.OldValue,
            (double)e.NewValue, new TimeSpan(0, 0, 0, 0, 250));
        (d as ProgressBar).BeginAnimation(ProgressBar.ValueProperty,
            anim, HandoffBehavior.Compose);
    }
}
```

Kuva 74. Edistymispalkin animointiluokka

ProgressBarSmoother-apuluokka (kuva 74) animoi ProgressBar-elementin tilan vaihtelun sulavammaksi. Apuluokka vertaa edellistä arvoa uuteen arvoon ja animoi edistymispalkin täyttymisen.

6.4.2 Näkymämalli

Latausnäkymämalli hallitsee lokitiedostojen lukemisen ennenaikaista keskeyttämistä. Näkymämallissa sijaitsee myös ominaisuudet lokitiedostojen lukemisen tilan esittämiseen.

```
public CancellationTokenSource tokenSource;
public CancellationToken cancelToken;

private string _status;
public string Status
{
    get => _status;
    set => Set(ref _status, value);
}

private int _currentProgress;
public int CurrentProgress
{
    get => _currentProgress;
    set => Set(ref _currentProgress, value);
}

private int _maxProgress;
public int MaxProgress
{
    get => _maxProgress;
    set => Set(ref _maxProgress, value);
}

public RelayCommand CancelCommand
    => new RelayCommand(CancelAction, CanCancel);
```

Kuva 75. Latausnäkymämallin ominaisuudet

Latausnäkymämalli (kuva 75) sisältää keskeyttämiseen käytetyt CancellationTokenSource- ja CancellationToken-oliot. Näkymämallissa on myös sovelluksen tilan ilmaiseva Status sekä edistymistä ilmaisevat CurrentProgress- ja MaxProgress-muuttujat. CancelCommand-komennolla voidaan keskeyttää lokitiedostojen luku.

```
public LoadingViewModel(int maxProgress)
{
    CurrentProgress = 0;
    MaxProgress = maxProgress;
    Status = "Reading log files...";

    tokenSource = new CancellationTokenSource();
    cancelToken = tokenSource.Token;
}
```

Kuva 76. Latausnäkymämallin rakentaja

Rakentajametodille (kuva 76) annetaan parametrina luettavien tiedostojen lukumäärä. Näkymämallille asetetaan ominaisuuksille uudet arvot ja keskeyttämistä varten luodaan uudet keskeytysoliot.

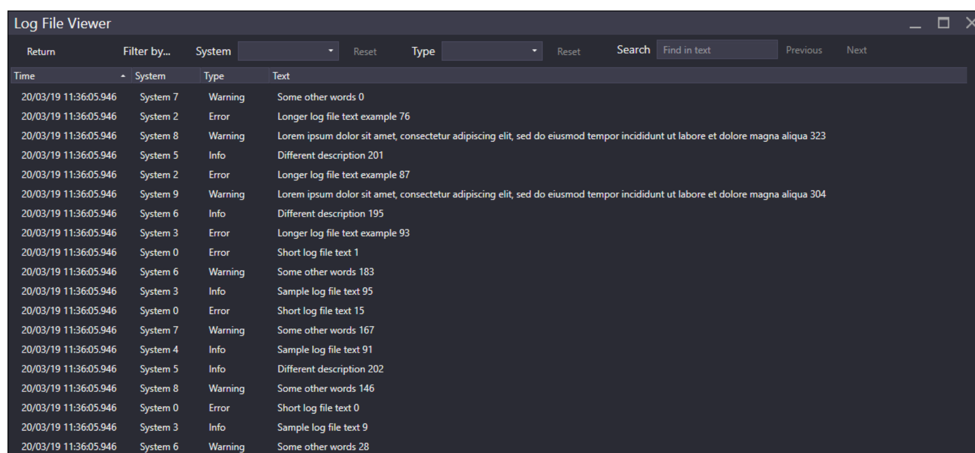
```
private void CancelAction()
{
    tokenSource.Cancel();
    Status = "Cancelling...";
}
private bool CanCancel() => !cancelToken.IsCancellationRequested;
```

Kuva 77. Lokitiedostojen lukemisen keskeyttäminen

Keskeyttämismetodi (kuva 77) kutsuu CancellationToken-olion Cancel-metodia ja asettaa tilatekstin. Asynkronisuuden vuoksi lokitiedostojen lukeminen ei aina keskeydy heti. Keskeytystä ei voida kutsua toista kertaa, joten komento poistetaan käytöstä.

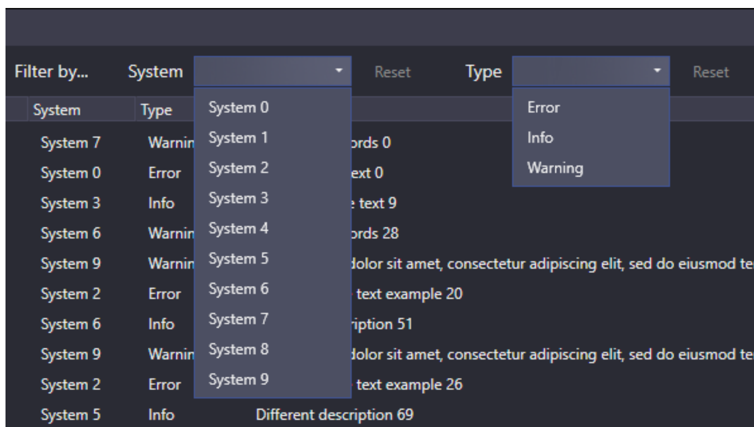
6.5 Päänäkymä

Sovelluksen tärkein näkymä on päänäkymä (kuva 78). Päänäkymässä näytetään luetut lokitiedostot. Lokitiedostoja voidaan selata, järjestää, suodattaa ja etsiä. Päänäkymästä voidaan palata takaisin aloitusnäkymään.



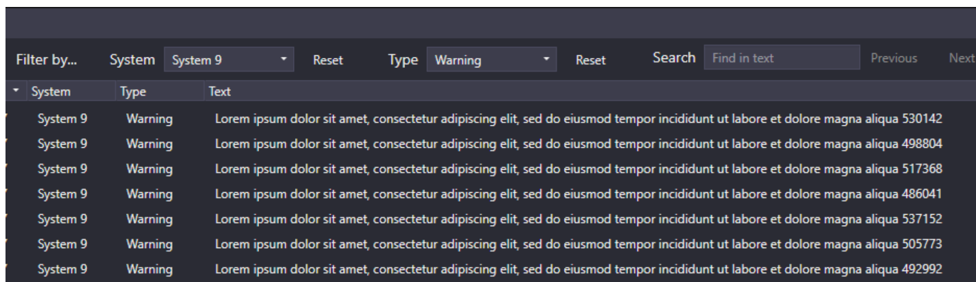
Kuva 78. Sovelluksen päänäkymä

Listanäkymään luettuja lokirivejä voi järjestää kolumnien ylätunnisteista painamalla. Rivejä voi suodattaa ja hakea yläpalkista löytyvillä ominaisuuksilla.



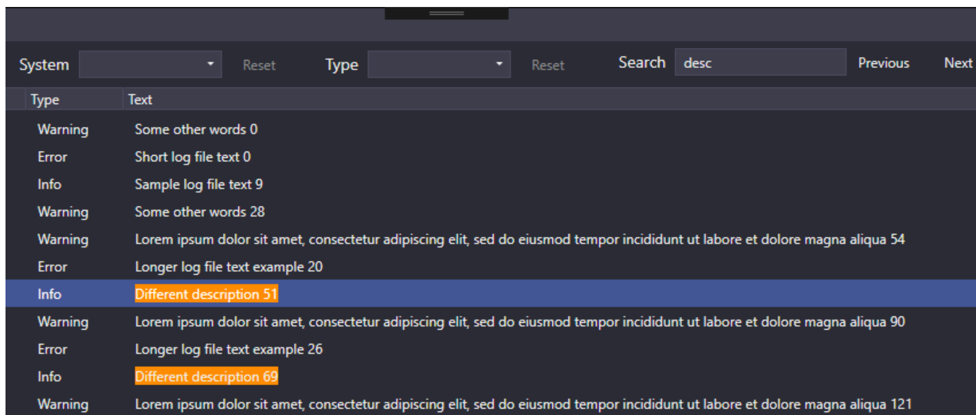
Kuva 79. Suodattimet

Suodattimet (kuva 79) luodaan automaattisesti luetuista lokiriveistä. Suodattimia voi valita useamman käyttöön samanaikaisesti.



Kuva 80. Suodatettu näkymä

Suodattimet näyttävät listanäkymässä vain suodattimia vastaavat rivit (kuva 80). Suodattimen voi nollata Reset-painikkeella.



Kuva 81. Valittu hakutulos

Hakuominaisuus korostaa hakukentän termiä vastaavat rivit (kuva 81). Hakutuloksia voi selata läpi Previous- ja Next-painikkeilla. Mikäli hakutuloksia ei ole, hakupainikkeet eivät toimi

6.5.1 Näkymä

Päänäkymä on sovelluksen monimutkaisin näkymä. Näkymä sisältää lokitiedostojen tarkasteluun käytettävät ominaisuudet.

```
<UserControl xmlns:converters="clr-namespace:LogFileViewer.Utilities.Converters"
             xmlns:adonisUi="clr-namespace:AdonisUI;assembly=AdonisUI"
             xmlns:adonisExtensions="clr-namespace:AdonisUI.Extensions;assembly=AdonisUI"
             <...>
             >
    <UserControl.Resources>
        <converters:TextMatchToColorConverter x:Key="TextToColor"
                                              True="DarkOrange" False="Transparent"/>
        <converters:FiltersToVisibilityConverter x:Key="FilterToVis"
                                                 True="Visible" False="Collapsed"/>
    </UserControl.Resources>

    <...>

</UserControl>
```

Kuva 82. Näkymän attribuuttien määrittely

Näkymälle (kuva 82) määritellään muuntimien, AdonisUI- ja AdonisExtensions-kirjastojen sijainnit. Näkymässä käytetään muuntimia hakutuloksien korostamiseen ja lokirivien suodattamiselle.

```
public Brush True { get; set; }
public Brush False { get; set; }

public object Convert(object[] values, Type targetType,
    object parameter, CultureInfo culture)
{
    string cellText = values[0] == null ? string.Empty : values[0].ToString();
    string searchText = values[1] as string;

    if (!string.IsNullOrEmpty(searchText) && !string.IsNullOrEmpty(cellText))
    {
        return cellText.ToLower().Contains(searchText.ToLower()) ? True : False;
    }
    return False;
}
```

Kuva 83. Taustavärimuunnin

Taustavärimuunnin (kuva 83) paljastaa ominaisuudet korostusvärin muokkaamiselle. Muuntimelle annetaan lokirivin kuvaus ja hakutermi. Muunnin tarkistaa löytyykö hakutermiä vastaava teksti kuvauksesta ja palauttaa tuloksen.

```
public Visibility True { get; set; }
public Visibility False { get; set; }

public object Convert(object[] values, Type targetType,
    object parameter, CultureInfo culture)
{
    LogFileRow row = values[0] as LogFileRow;
    string system = values[1] == null ? string.Empty : values[1] as string;
    string type = values[2] == null ? string.Empty : values[2] as string;

    return row.LogSystem.ToLower().Contains(system.ToLower())
        && row.LogType.ToLower().Contains(type.ToLower()) ? True : False;
}
```

Kuva 84. Näkyvyysmuunnin

Näkyvyysmuunnin (kuva 84) palauttaa lokirivin näkyvyyden suodattimien perusteella. Lokirivi piilotetaan, jos se ei täytä suodattimien ehtoja.

```
<StackPanel Orientation="Horizontal" Margin="32 0 0 0">
    <Label Content="System" Style="{DynamicResource ItemGroupLabelStyle}" />
    <ComboBox Name="SystemCombo" Text="" IsEditable="False"
              Width="125" ItemsSource="{Binding Systems}"
              Margin="8 0" SelectedItem="{Binding SelectedSystem}"/>
    <Button Content="Reset" Width="50"
            Style="{DynamicResource {x:Static adonisUi:Styles.ToolbarButton}}"
            Command="{Binding ResetCommand}" CommandParameter="System"/>
</StackPanel>
```

Kuva 85. Suodatin

Suodattimen (kuva 85) rakenne käyttöliittymässä koostuu Label-, ComboBox- ja Button-elementeistä. ComboBox-elementtiin tieto sidotaan lista suodattimen asetuksista ja valittu suodatin. Button-elementille on asetettu AdonisUI-kirjaston tyyli, nollauskomento ja komentoparametri, josta päätellään nollattava suodatin.

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Right"
            VerticalAlignment="Center" Margin="0 0 128 0">
    <Label Content="Search" Style="{DynamicResource ItemGroupLabelStyle}"
           Margin="0 0 8 0" />
    <TextBox Name="SearchTextBox" Width="150"
             adonisExtensions:WatermarkExtension.Watermark="Find in text"
             Text="{Binding SearchTerm, UpdateSourceTrigger=PropertyChanged}"/>
    <Button Content="Previous" Width="65"
            Style="{DynamicResource {x:Static adonisUi:Styles.ToolbarButton}}"
            Command="{Binding SearchCommand}" CommandParameter="Previous"/>
    <Button Content="Next" Width="65"
            Style="{DynamicResource {x:Static adonisUi:Styles.ToolbarButton}}"
            Command="{Binding SearchCommand}" CommandParameter="Next"/>
</StackPanel>
```

Kuva 86. Haku

Hakuominaisuus (kuva 86) koostuu yhdestä TextBox-elementistä ja kahdesta Button-elementistä. TextBox-elementtiin syötetään hakutermi, jota käytetään etsimään hakutuloksia listanäkymästä. Elementille on asetettu AdonisExtensions-kirjaston Watermark-ominaisuus, joka sisältää ohjetekstiä. Button-elementit on yhdistetty komentoihin ja komennoille on asetettu parametrit, joiden avulla päätellään suoritettava logiikka näkymämallissa.

```
<ListView Grid.Row="1" ItemsSource="{Binding LogRows}"
          SizeChanged="ListView_SizeChanged"
          ScrollViewer.HorizontalScrollBarVisibility="Hidden"
          SelectedItem="{Binding SelectedItem}"
          SelectionChanged="ListView_SelectionChanged">
    <ListView.ItemContainerStyle>
        <...>
    </ListView.ItemContainerStyle>
    <ListView.View>
        <...>
    </ListView.View>
</ListView>
```

Kuva 87. Listanäkymä

Listanäkymän (kuva 87) tietolähteeksi asetetaan lista luetuista lokiriveistä. Listan valittu rivi on tietosidottu näkymämallin ominaisuuteen, että se voidaan asettaa näkymämallista. Listan koon muutoksen ja valitun rivin muutoksen tapahtumille on luotu käsittelijät. Listan näkymälle ja näkymän riveille on määritelty muokatut ulkoasut, joita käytetään haku- ja suodatinominaisuuksien kanssa.

```
private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    ListView listView = sender as ListView;
    listView.ScrollIntoView(listView.SelectedItem);
}
```

Kuva 88. SelectionChanged-tapahtumankäsittelijä

SelectionChanged-metodia (kuva 88) käytetään hakuominaisuuden kanssa. Näkymämallin hakuominaisuus asettaa listalle valitun rivin, mutta näkymämalli ei voi siirtää listanäkymää. SelectionChanged-metodi siirtää listanäkymän valittuun riviin, kun rivi ei ole näkyvissä.

```
<Style TargetType="{x:Type ListViewItem}"
       BasedOn="{StaticResource {x:Type ListViewItem}}">
    <Setter Property="Visibility">
        <Setter.Value>
            <MultiBinding Converter="{StaticResource FilterToVis}">
                <Binding />
                <Binding Source="{x:Reference SystemCombo}"
                         Path="SelectedItem"
                         UpdateSourceTrigger="PropertyChanged"/>
                <Binding Source="{x:Reference TypeCombo}"
                         Path="SelectedItem"
                         UpdateSourceTrigger="PropertyChanged"/>
            </MultiBinding>
        </Setter.Value>
    </Setter>
</Style>
```

Kuva 89. Listanäkymän rivin tyyli

Listanäkymän riveille määritellään tyyli (kuva 89), jota käytetään toteuttamaan suodatusominaisuus. Rivien näkyvyys määritellään muuntimella, jolle annetaan lokirivin kuvaus ja suodattimissa valitut arvot.

```
<GridView>
    <GridView.Columns>
        <GridViewColumn Header="Time" DisplayMemberBinding="{Binding LogDateTime,
            StringFormat='{}{0: dd/MM/yy HH:mm:ss.fff}'}" Width="150"/>
        <GridViewColumn Header="System" Width="85"
                        DisplayMemberBinding="{Binding LogSystem}" />
        <GridViewColumn Header="Type" Width="85"
                        DisplayMemberBinding="{Binding LogType}" />
        <GridViewColumn Header="Text" >
            <GridViewColumn.CellTemplate>
             <...>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
    </GridView.Columns>
</GridView>
```

Kuva 90. Listanäkymän kolumnit

Listan näkymäksi asetetaan GridView-elementti (kuva 90), joka on sopiva lokirivien tiedon esittämiselle. Elementille määritellään kolumnit, joihin tietosidotaan lokiriviolion ominaisuudet. Lokiriviolion kuvausominaisuus voi olla hakutulos, joten sille asetetaan mukautettu tietomalli.

```
<DataTemplate>
    <TextBlock Text="{Binding LogText}">
        <TextBlock.Background>
            <MultiBinding Converter="{StaticResource TextToColor}">
                <Binding Path="LogText" />
                <Binding Source="{x:Reference SearchTextBox}"
                         Path="Text"
                         UpdateSourceTrigger="PropertyChanged"/>
            </MultiBinding>
        </TextBlock.Background>
    </TextBlock>
</DataTemplate>
```

Kuva 91. Listanäkymän kolumnin tietomalli

Tietomalli (kuva 91) pitää sisällään TextBlock-elementin, johon on tietosidottu lokiriviolion kuvausominaisuus. TextBlock-elementille määritellään taustaväri muuntajan avulla. Muuntajalle annetaan viite hakutermin sisältävään TextBox-elementtiin, ja korostettavat rivit tarkistetaan aina hakutermin muuttuessa.

6.5.2 Näkymämalli

Päänäkymämallissa on lista esitettävistä lokiriveistä. Näkymämalli myös hallitsee haku- ja suodatusominaisuuksia.

```
private ObservableCollection<LogFileRow> _logRows;
public ObservableCollection<LogFileRow> LogRows
{
    get => _logRows;
    set => Set(ref _logRows, value);
}
private string _searchTerm;
public string SearchTerm
{
    get => _searchTerm;
    set { Set(ref _searchTerm, value); SearchCommand.RaiseCanExecuteChanged(); }
}
private LogFileRow _selectedItem;
public LogFileRow SelectedItem
{
    get => _selectedItem;
    set => Set(ref _selectedItem, value);
}
```

Kuva 92. Päänäkymämallin ominaisuudet

Päänäkymämalli (kuva 92) sisältää listan lokiriviolioista, hakutermin ja listassa valitun rivin. Listan tyyppi on ObservableCollection, jotta näkymä voi seurata sen muutoksia. Hakutermin vaihtuessa tarkistetaan hakukomennon suoritusehdot.

```
private ParentViewModel _parent;

public MainViewModel(ParentViewModel parent)
{
    _parent = parent;
    LogRows = new ObservableCollection<LogFileRow>();
}
```

Kuva 93. Päänäkymämallin rakentaja

Päänäkymämallin rakentajalle (kuva 93) annetaan viite pääikkunan näkymämalliin, jonka avulla voidaan palata takaisin aloitusnäkymään. Päänäkymämallin rakentajassa alustetaan myös uusi lista lokiriviolioista.

```
private string _selectedSystem;
public string SelectedSystem
{
    get => _selectedSystem;
    set { Set(ref _selectedSystem, value);
        SearchCommand.RaiseCanExecuteChanged(); }
}
private List<string> _systems;
public List<string> Systems
{
    get => _systems;
    set => Set(ref _systems, value);
}
```

Kuva 94. Suodatinominaisuudet

Suodatin (kuva 94) on toteutettu näkymämallissa kahdella ominaisuudella. Valittu suodatin voidaan nollata tarvittaessa ja lista suodattimista luodaan, kun lokitiedostot on luettu. Valitun suodattimen muuttuessa tarkistetaan, voidaanko suodattimen nollauskomento suorittaa.

```
public void CreateFilters()
{
    Systems = LogRows.Select(x =>
    x.LogSystem).Distinct().OrderBy(s => s).ToList();
    <...>
}
```

Kuva 95. Suodattimien luonti

Suodattimet luodaan (kuva 95) valitsemalla luetuista lokiriveistä jokainen ominaisuus ja poistamalla toistuvat arvot. Jokainen suodatin luodaan samalla tavalla.

```
public RelayCommand BackCommand
    => new RelayCommand(BackAction);

public RelayCommand<string> ResetCommand
    => new RelayCommand<string>(ResetAction, CanReset);

public RelayCommand<string> SearchCommand
    => new RelayCommand<string>(SearchAction, CanSearch);
```

Kuva 96. Komennot

Päänäkymämalli sisältää komennot (kuva 96) aloitusnäkymään palaamiseen, suodattimien nollaukseen ja hakutulosten selaamiseen. Suodattimien nollaus- ja hakutulosten selauskomennoille asetetaan myös suoritusehto. Suodatinta ei voi nollata, jos mitään suodatinta ei ole valittu. Hakua ei voida suorittaa, jos hakutuloksia ei ole.

```
private void ResetAction(string filter)
{
    if (filter.Equals("System"))
        SelectedSystem = null;
    <...>
}
private bool CanReset(string filter)
{
    if (filter.Equals("System"))
        return SelectedSystem != null;

    <...>

    return false;
}
```

Kuva 97. Suodattimien nollaus

Kuvassa 97 on suodattimien nollausmetodi ja nollausmetodin suoritusehto. Suodattimet nollataan asettamalla valittu suodatin tyhjäksi. Nollattava suodatin tunnistetaan komennon antaman parametrin perusteella. Nollauskomento voidaan suorittaa, kun valittu suodatin ei ole tyhjä.

```
private void SearchAction(string direction)
{
    string system = SelectedSystem ?? string.Empty;
    string type = SelectedType ?? string.Empty;
    List<LogFileRow> matches = LogRows.Where(row =>
    row.LogText.ToLower().Contains(SearchTerm.ToLower())
    && row.LogSystem.ToLower().Contains(system.ToLower())
    && row.LogType.ToLower().Contains(type.ToLower())).ToList();

    if (direction.Equals("Previous"))
    {
        int index = SelectedItem != null ? matches.IndexOf(SelectedItem) : 0;
        matches.RemoveRange(index, matches.Count - index);
        matches.Reverse();

        if (matches.Count == 0)
        {
            matches = LogRows.Where(row =>
            row.LogText.ToLower().Contains(SearchTerm.ToLower())
            && row.LogSystem.ToLower().Contains(system.ToLower())
            && row.LogType.ToLower().Contains(type.ToLower())).ToList();
            matches.Reverse();
        }
        SelectedItem = matches.First();
    }
    if (direction.Equals("Next"))
    {
        int index = SelectedItem != null ? matches.IndexOf(SelectedItem) : -1;
        matches.RemoveRange(0, index + 1);

        if (matches.Count == 0)
        {
            matches = LogRows.Where(row =>
            row.LogText.ToLower().Contains(SearchTerm.ToLower())
            && row.LogSystem.ToLower().Contains(system.ToLower())
            && row.LogType.ToLower().Contains(type.ToLower())).ToList();
        }
        SelectedItem = matches.First();
    }
}
```

Kuva 98. Hakumetodi

Hakuominaisuuden metodi (kuva 98) hakee alussa kaikki osumat, joita käytetään päättelemään seuraava ja edellinen hakutulos. Haun etenemissuunta päätellään komennon antamasta parametrista. Seuraavan ja edellisen hakutuloksen päättelyyn käytetään listanäkymässä valittua riviä. Seuraava ja edellinen hakutulos on valitun rivin lähin rivi, joka täyttää haun ehdot. Jos hakutuloksia ei ensin löydy, laajennetaan hakua. Hakutuloksia on aina vähintään yksi, koska hakukomentoa ei anneta suorittaa nollalla hakutuloksella.

```
private bool CanSearch(string direction)
{
    if (!string.IsNullOrWhiteSpace(SearchTerm))
    {
        string system = SelectedSystem == null ? string.Empty : SelectedSystem;
        string type = SelectedType == null ? string.Empty : SelectedType;
        return LogRows.Where(row =>
                row.LogText.ToLower().Contains(SearchTerm.ToLower())
             && row.LogSystem.ToLower().Contains(system.ToLower())
             && row.LogType.ToLower().Contains(type.ToLower())).Count() > 0;
    }
    return false;
}
```

Kuva 99. Haun suorittamisehto

CanSearch-metodia (kuva 99) käytetään tarkistamaan hakutulosten lukumäärä. Metodi ottaa huomioon hakutuloksen lisäksi myös suodattimet. Jos hakutuloksia on vähintään yksi, hakukomento voidaan suorittaa.

7 YHTEENVETO

Opinnäytetyössä käytiin läpi MVVM-arkkitehtuurimallin historia, rakenne ja toimintaperiaate. Työssä myös perehdyttiin WPF-teknologian arkkitehtuurin, käyttöliittymään ja MVVM-mallin kannalta tärkeisiin ominaisuuksiin. Näitä kahta tekniikkaa käytettiin toteuttamaan sovellus lokitiedostojen tarkastelua varten. Sovellukselle tehtiin vaatimusmäärittely, käytiin läpi käytetyt tekniikat ja suunnittelu sekä sovelluksen toteutus.

Lopputuloksena saatiin aikaan yksinkertainen ja toimiva sovellus. Sovellus täyttää sille asetetut minimitavoitteet, joista tärkeimpinä pidettiin suodatus- ja hakuominaisuuksia sekä lokitiedostojen lukua eri tiedostomuodoista. Sovelluksen toteuttaminen suunnitteluvaiheesta kesti odotettua kauemman, sillä tekijällä ei ollut aiempaa kokemusta MVVM-mallista. Mallin opettelu ja ymmärtäminen vei oman aikansa, mutta sovelluksen ylläpito ja jatkokehitys on helpompaa.

Luotu työkalu on yrityksessä useamman henkilön käytössä. Työkalusta on saatu palautetta ja käyttäjien kohtaamia ongelmia on korjattu. Sovellus on nyt huoltotilassa, mutta sovellusta voidaan jatkokehittää tarvittaessa.

7.1 Jatkokehitys

Sovellus oli tekijän ensimmäinen askel MVVM-mallin mukaisen sovelluksen suunnitteluun ja toteuttamiseen. MVVM-mallia enemmän osaavana sovelluksen rakennetta voisi selventää ja keventää entisestään. Sovelluksen ominaisuuksien toteuttamisen uudelleensuunnittelu saattaa olla tarpeen seuraavan kahden jatkokehitysidean onnistumisen kannalta.

Sovellus on suunniteltu lukemaan ja käsittelemään vain tietynlaisia lokitiedostorakenteita. Mikäli tarve ilmenee käsitellä eri rakenteisia lokeja, täytyy sovellukseen logiikka luoda erikseen. Sovelluksen tulevaisuuden turvaamiseksi käyttäjällä voisi olla mahdollisuus määritellä luettavan lokitiedoston rakenne. Käyttäjän määrittele-

män rakenteen avulla luettaisiin lokitiedostot, luotaisiin näkymä ja suodattimet. Hakuominaisuus korostaa tällä hetkellä koko rivin, josta hakutermiä vastaava osuma löytyy. Hakuominaisuus voisi korostaa vain osumaa vastaavan tekstin.

Sovelluksen suorituskyky laskee erittäin suuria lokitiedostoja käsitellessä. Tiedostojen lukunopeus on hyvä, mutta suodatus- ja hakuominaisuudet tarvitsisivat hieman optimointia. Sovelluksen muistinkäyttö nousee jatkuvasti useita kertoja isoja lokitiedostoja lukiessa, mikä saattaa viitata ongelmaan muistinhallinnassa. Suurin osa työkalulla tarkasteltavista lokitiedostoista ovat sen verran pieniä, etteivät suorituskyvyn ongelmat ilmene.

7.2 Pohdinta

Sovellus oli tekijälle ensimmäinen isompi WPF-teknologialla toteutettu sovellus. Sovellusta aiemmin toteutetut sovellukset ovat olleet paljon pienempi kokoisia ja tavoitteisia. WPF-käyttöliittymäkirjastosta opittiin paljon itse sovellusta tehdessä sekä opinnäytetyön teoriaa kirjoittaessa. Opinnäytetyön myötä saatiin syvempi perehdytys WPF-sovelluksiin.

MVVM-malli oli tekijälle täysin uusi tapa suunnitella ja toteuttaa sovelluksia. Mallin opiskeluun ja siihen perehtymiseen käytettiin paljon aikaa ennen sovelluksen suunnittelun ja toteutuksen aloittamista. Sovellusta toteuttaessa mallin hyödyntämisestä opittiin paljon käytännön kautta. MVVM-mallin täysi hyödyntäminen vaatii kuitenkin vielä paljon opettelua.

Sovelluksen toteuttamisen aikana on tullut kehityttyä ohjelmistojen suunnittelijana ja kehittäjänä. Sovelluksesta käytiin useampi palaveri, joissa seurattiin sovelluksen kehitystä ja asetettiin uusia tavoitteita tarvittaessa. Käyttöliittymän suunnittelu on parantunut, mutta vielä on opittavaa käyttäjäkokemuksesta. Käyttäjien palautteen tärkeys on käynyt ilmi sovelluksen käyttöönoton jälkeen.

LÄHTEET

AdonisUI. Ei päiväystä. Github. [Verkkosivu]. [Viitattu 19.3.2019]. Saatavana: https://github.com/benruehl/adonis-ui

Anderson, C. 2010. Pro Business Applications with Silverlight 4. New York: Apress.

Bugnion, L. 2014. MVVM Light Toolkit Fundamentals. [Verkkosivu]. [Viitattu 3.2.2019]. Saatavana: https://www.pluralsight.com/courses/mvvm-light-toolkit-fundamentals. Vaatii käyttöoikeuden.

Chaudhary, V. 2016. Different Ways to Bind WPF View And View Model. [Verkkosivu]. [Viitattu 21.3.2019]. Saatavissa: https://www.dotnetforall.com/different-ways-bind-view-view-model/

Costura. Ei päiväystä. Github. [Verkkosivu]. [Viitattu 19.3.2019]. Saatavana: https://github.com/Fody/Costura

Fody. Ei päiväystä. Github. [Verkkosivu]. [Viitattu 19.3.2019]. Saatavana: https://github.com/Fody/Home

Historia. Ei päiväystä. Prima Power. [Verkkosivu]. [Viitattu 23.1.2019]. Saatavana: https://www.primapower.com/fi/historia/

James, B. & Lalonde, L. 2015. Pro XAML with C#. New York: Apress.

Lapthorn, B. 2012. WPF/MVVM Quick Start Tutorial. [Verkkosivu]. [Viitattu 21.3.2019]. Saatavissa: https://www.codeproject.com/Articles/165368/WPF-MVVM-Quick-Start-Tutorial

MacDonald, M. 2010. Pro WPF 4.5 in C#. Fourth edition. New York: Apress.

Microsoft. 2009. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. [Verkkosivu]. [Viitattu 16.3.2019]. Saatavana: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Microsoft. 2010. [Verkkosivu]. Introducing Windows Presentation Foundation. [Viitattu 27.2.2019]. Saatavana: https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/aa663364(v=msdn.10)

Microsoft. 2012. Model-View-ViewModel (MVVM) Applications: General Introduction. [Verkkosivu]. [Viitattu 9.2.2019]. Saatavana: https://blogs.msdn.microsoft.com/ivo_manolov/2012/03/17/model-view-viewmodel-mvvm-applications-general-introduction/

Microsoft. 2017a. [Verkkosivu]. Commanding Overview. [Viitattu 15.3.2019]. Saatavana: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/commanding-overview

Microsoft. 2017b. [Verkkosivu]. Customizing the Appearance of an Existing Control by Creating a ControlTemplate. [Viitattu 14.3.2019]. Saatavana: https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/customizing-the-appearance-of-an-existing-control

Microsoft. 2017c. Data Binding Overview. [Verkkosivu]. [Viitattu 7.3.2019]. Saatavana: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview

Microsoft. 2017d. The Model-View-ViewModel Pattern. [Verkkosivu]. [Viitattu 2.2.2019]. Saatavana: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Microsoft. 2018a. Choose Your Platform. [Verkkosivu]. [Viitattu 9.3.2019]. Saatavana: https://docs.microsoft.com/en-us/windows/desktop/choose-your-technology

Microsoft. 2018b. What's a Universal Windows Platform (UWP) app? [Verkkosivu]. [Viitattu 9.3.2019]. Saatavana: https://docs.microsoft.com/fi-fi/windows/uwp/get-started/universal-application-platform-guide

Nathan, A. 2014. WPF 4.5 Unleashed. New York: Sams Publishing.

Next to you. Ei päiväystä. Prima Power. [Verkkosivu]. [Viitattu 24.1.2019]. Saatavana: https://www.primapower.com/fi/next-to-you/

Prajapati, S. 2015. ICommand Interface in WPF. [Verkkosivu]. [Viitattu 21.3.2019]. Saatavissa: https://www.codeproject.com/Articles/1052346/ICommand-Interface-in-WPF

Työpaikat. Ei päiväystä. Prima Power. [Verkkosivu]. [Viitattu 25.1.2019]. Saatavana: https://www.primapower.com/fi/avoimet-tyopaikat/

Vice, R. & Siddiqi, M. S. 2012. MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF. [Verkkokirja]. Birmingham: Packt Publishing. [Viitattu 23.1.2019]. Saatavana: Ebsco eBook Collection. Vaatii käyttöoikeuden.

Yritys. Ei päiväystä. Prima Power. [Verkkosivu]. [Viitattu 26.1.2019]. Saatavana: https://www.primapower.com/fi/yritys/