YRKESHÖGSKOLAN
NOVIA

Utveckling av program för aktieanalys

En implementering av mönstret Passive View

Jonas Slotte

Examensarbete för YH-examen i teknik

Utbildningsprogrammet för informationsteknik

Vasa 2017

EXAMENSARBETE

Författare: Jonas Slotte

Utbildning och ort: Informationsteknik, Vasa

Profilering: Informationsteknik

Handledare: Kaj Wikman

Titel: Utveckling av program för aktieanalys – En implementering av mönstret Passive View

Datum 17.10.2017 Sidantal 43 Bilagor 3

Abstrakt

Detta examensarbete behandlar utvecklingen av applikationen Trading Assistant enligt mönstret Passive View. Trading Assistant är en Windows Forms applikationsprototyp för analys av aktiers historiska data, och har utvecklats av Black Label Bytes AB.

Ursprunget för examensarbetet var behovet av en omstrukturering av koden i användargränssnittet. Kod för presentation och affärslogik var sammankopplade på ett ostrukturerat sätt som försvårade utvecklingsarbetet. Det huvudsakliga målet med examensarbetet blev att modifiera den existerande koden på ett sätt som underlättar framtida utveckling av applikationen.

Examensarbetet behandlar två distinkta delar i utvecklingen av Trading Assistant. Den första delen i examensarbetet beskriver projektet för omstrukturering av koden i användargränssnittet till arkitekturmönstret Passive View.

Den andra delen av examensarbetet beskriver utvecklingen av nya funktioner enligt den nya strukturen. Under denna utveckling uppstod vissa svårigheter med att implementera funktionalitet med Passive View p.g.a. hur Windows Forms fungerar. Hur problemen löses eller kringgås, förklaras och motiveras.

Språk: svenska Nyckelord: Windows Forms, Passive View, MVP

OPINNÄYTETYÖ

Tekijä: Jonas Slotte

Koulutus ja paikkakunta: Tietotekniikka, Vaasa

Profilointi: Tietotekniikka

Ohjaaja: Kaj Wikman

Nimike: Osakeanalyysisovelluksen kehitys – Passive View -mallin toteutus

Päivämäärä 17.10.2017 Sivumäärä 43 Liitteet 3

Tiivistelmä

Tämä opinnäytetyö käsittelee Trading Assistant -sovelluksen kehitystä Passive View -mallia käyttäen. Trading Assistant on Windows Forms -sovellusprototyyppi, jota käytetään osakkeiden historiallisten tietojen analysointiin.

Opinnäytetyö sai alkunsa, kun huomattiin, että graafisessa käyttöliittymässä oleva koodi piti järjestää uudelleen. Esittelykoodi ja liiketoimintakoodi oli liitetty yhteen epäjärjestelmällisellä tavalla, mikä hankaloitti koodin muokkaamista.
Tavoite oli muokata olemassa olevaa koodia niin, että se helpottaisi sovelluksen kehittämistä tulevaisuudessa.

Opinnäytetyö on jaettu kehitysprosessin mukaan kahteen osaan. Ensimmäinen osa koskee alussa tehtyä projektia, jossa koodia järjesteltiin uudelleen. Sen tavoitteena oli muokata graafisessa käyttöliittymässä olevaa koodia Passive View -mallin mukaan.

Toinen osa käsittelee uusien ominaisuuksien kehitystä, mikä tehtiin uuden rakenteen mukaan. Tämän kehitysprosessin aikana esiintyi ongelmia, missä oli vaikea saada Windows Formsia toimimaan Passive View -mallin mukaan. Selitetään ja perustellaan, kuinka ongelmat ratkaistaan tai kierretään.

Kieli: ruotsi Avainsanat: Windows Forms, Passive View, MVP

BACHELOR'S THESIS

Author: Jonas Slotte
Degree Programme: Information Technology, Vaasa
Profile: Information Technology
Supervisor: Kaj Wikman

Title: Development of Stock Analysis Application – An Implementation Using the Passive View Pattern

Date October 17, 2017 Number of pages 43 Appendices 3

Abstract

This thesis is based on the development of the Trading Assistant application using the Passive View pattern. Trading Assistant is an application prototype for stock history analysis made using Windows Forms, and has been developed by Black Label Bytes AB.

The origin of the thesis was the need for a restructuring of the code in the graphical user interface. Presentation logic and business logic were combined in an unstructured manner, which complicated the development work. The primary goal was to restructure the code to increase future maintainability.

The thesis is a combination of two separate parts of the development process. The first one is the initial restructuring project, aiming to modify the existing code to match the guidelines of the Passive View pattern, where applicable.

The second part concerns the development of new features, which was done using the implemented structure. During this process, there were some difficulties regarding the inherent functionality of Windows Forms, when combined with the principles of the Passive View pattern. How the problems are solved or circumvented, is explained and motivated.

Language: swedish Key words: Windows Forms, Passive View, MVP

Innehållsförteckning

1 Inledning ... 1
1.1 Uppdragsgivare ... 1
1.2 Problembeskrivning ... 1
1.3 Syfte ... 1
2 Teori ... 2
2.1 Mönster ... 2
2.2 C# ... 2
2.2.1 Händelser ... 2
2.2.2 Egenskaper ... 2
2.3 .NET Framework ... 3
2.4 Windows Forms ... 3
2.4.1 Control och UserControl ... 4
2.4.2 DataGridView ... 4
2.4.3 Data Binding ... 5
2.5 Dialogruta ... 5
2.6 Refaktorisering ... 6
2.6.1 Extract Class ... 6
2.6.2 Extract Method ... 7
2.7 Gränssnitt ... 8
2.8 Model-View-Presenter ... 9
2.9 Passive View ... 9
2.9.1 Presenter ... 10
2.9.2 View ... 10
2.9.3 Model ... 12
2.10 Enhetstest ... 12
2.11 Dependency injection ... 12
2.12 Mocking ... 13
3 Programmet ... 14
3.1 Användningsområde ... 14
3.2 Befintliga funktioner ... 15
3.3 Tekniska detaljer ... 15
4 Verktyg ... 15
4.1 Utvecklingsmiljö ... 15
4.2 Versionshantering ... 16
4.3 Nätbaserade tjänster ... 16
5 Befintlig kod ... 16

5.1 Kodexempel från MainWindow 16
5.2 Kodexempel från MetricTestWindow 18
5.3 Kodexempel från StockMgmtWindow 19
5.4 Kodexempel från DataModel 20
5.5 Sammanfattning av kodexempel 20
6 Planering och målsättning 21
7 Genomförande av omstrukturering 22
7.1 Utflyttning av kod från användargränssnittet 22
7.2 Konstruktion av vy och dess gränssnitt 23
7.3 Konstruktion av Presenter 24
7.4 Konstruktion av Model 25
7.5 Konstruktion av TradingAssistantContext 27
8 Designval 28
8.1 Treeview 28
8.2 Indata från mus och tangentbord 29
8.3 Grafritning 29
8.4 Data Binding 30
8.5 Flera trådar 30
9 Efterarbete 31
9.1 Passive View och dialogrutor 31
9.2 Undervyer 32
9.3 Dynamiskt innehåll 34
9.3.1 Gruppering av undervyer i triader 35
9.3.2 Dynamiska utbyten av triader 35
9.4 Övriga erfarenheter 36
10 Resultat 36
10.1 Presentationskod 37
10.2 Affärskod 37
10.3 Dokumentation för kodbeskrivning 38
10.4 Sammanfattning av resultat 39
11 Diskussion 39
Källförteckning 41

1 Inledning

Detta examensarbete behandlar ett område i utvecklingen av programmet Trading Assistant, som är en produkt av Black Label Bytes AB. Trading Assistant är ett program för hantering och analys av aktiers historiska data. Arbetet berör arkitekturmönster i kod för grafiska användargränssnitt.

Examensarbetets genomförande är delat i två delar. Den första delen sammanfattar omstruktureringen av den befintliga koden, som gjordes som ett enhetligt projekt. Den andra delen behandlar utvecklandet av nya funktioner, som gjordes enligt den nya strukturen.

1.1 Uppdragsgivare

Examensarbetet gjordes på uppdrag av företaget Black Label Bytes AB. Företaget utvecklar mjukvara för datorer och mobila lösningar, samt erbjuder konsulteringstjänster.

1.2 Problembeskrivning

Detta examensarbete fick sitt ursprung i planeringen av nya funktioner för Trading Assistant. Arbetet på vissa komponenter hade redan inletts, och det började synas tecken på behov av en omstrukturering. Behovet motiveras med kodexempel i kapitel fem.

Den existerande kodbasen i programmet var strukturerad på ett sätt som skulle försvåra implementeringen av nya funktioner i framtiden. Därför beslöt uppdragsgivaren att starta ett projekt för att åtgärda detta problem först.

1.3 Syfte

Syftet med examensarbetet var att först strukturera om all kod i användargränssnittet på ett sådant sätt som underlättar framtida utveckling. Detta innebar också att koden skulle dokumenteras och stödas av automatiska enhetstest.

Utöver omstruktureringen skulle fortsatta funktioner implementeras enligt samma principer och mönster.

2 Teori

Examensarbetet fokuserar på mönster i grafiska användargränssnitt. I detta kapitel presenteras begrepp och teknologier som används i examensarbetet.

2.1 Mönster

Ordet mönster är direkt översatt från det engelska ordet pattern.

Ett mönster beskriver ett problem som är kontinuerligt återkommande i en miljö, och beskriver lösningen på detta problem så att den kan upprepas otaliga gånger utan göra den på samma sätt två gånger. (Alexander, et.al. 1977, Förord).

I sammanhanget mjukvaruutveckling har användningen av mönster den nyttan att de erbjuder ett gemensamt språk för programmerare, och en gemensam grund för att förstå vad som är viktigt i programmering. (Gabriel, 1996, 51–52).

Det finns mönster som kan tillämpas på alla olika nivåer av kod i ett program. I detta examensarbete ligger fokus på arkitekturmönster i grafiska användargränssnitt.

2.2 C#

Programmeringsspråket som användes i examensarbetet är C# (uttalas ”C sharp”). Språket är designat för att bygga en mängd olika applikationer som kör på .NET Framework. C# är ett enkelt, kraftfullt, typsäkert och objektorienterat språk. (Microsoft, C#).

2.2.1 Händelser

Språket C# har stöd för händelser (events). En händelse i C# är ett sätt för en klass att uppmärksamma sina klienter när något intressant händer med ett objekt. Det vanligaste stället att använda händelser i är det grafiska användargränssnittet. Händelser behöver dock inte användas endast för grafiska användargränssnitt. (Microsoft, Events Tutorial).

2.2.2 Egenskaper

En egenskap (property) är en medlem som erbjuder en flexibel mekanism som skriver, läser, eller räknar ut värdet på ett privat fält. Egenskaper kan användas som om de vore publika

datamedlemmar, men de är egentligen speciella metoder som kallas accessors. (Microsoft, 2017).

2.3 .NET Framework

Applikationen använder sig av Microsofts ramverk .NET Framework.

Ramverket som utgör grunden för programmet är .NET Framework (uttalas ”dot net”). Denna teknologi understöder konstruktion och exekvering av nästa generationens applikationer och XML webbtjänster. Ramverket är designat för att uppnå vissa mål så som:

- Erbjuda en konsekvent, objektorienterad programmeringsmiljö oavsett var kod lagras och oavsett var koden exekveras.
- Erbjuda en kodexekveringsmiljö som minimerar konflikter i mjukvaruspridning och versionshantering.

Ramverket kan användas för att utveckla bl.a. Konsolapplikationer, Windows GUI applikationer (Windows Forms), Windows Presentation Foundation applikationer (WPF) och ASP.NET webbapplikationer. (Microsoft, OverView of the .NET Framework).

2.4 Windows Forms

Windows Forms är en plattform för utveckling av Microsoft Windows applikationer som baserar sig på .NET Framework. Den centrala komponenten i denna plattform är Form klassen, som kan översättas till formulär. En Form är en vanligen fyrkantig del av skärmytan som kan användas för att presentera information till, samt ta in data från användaren. (Microsoft, Introduction to Windows Forms).

I examensarbetet används fortsättningsvis benämningen formulär.

Ett formulär i Windows Forms består av två filer: En automatiskt skapad Designer fil och en bakomliggande fil som är ämnad för handskriven kod. Det grafiska användargränssnittet i ett formulär kan modifieras visuellt i designer-verktyget. Dessa ändringar påverkar innehållet i den automatiskt skapade Designer filen, och ger direkt visuell respons på hur användargränssnittet ser ut och fungerar.

2.4.1 Control och UserControl

Control är basklassen för komponenter med visuell representation i Windows Forms (Microsoft, Control Class).

UserControl är en klass som indirekt ärver från klassen Control, med några klasser emellan. Klassen UserControl ger utvecklaren möjligheten att konstruera visuella komponenter som kan användas på flera ställen i en applikation eller en organisation. (Microsoft, UserControl Class).

2.4.2 DataGridView

DataGridView är en Control som erbjuder ett kraftfullt och flexibelt sätt att visa data i ett tabellformat. DataGridView kan antingen användas till att visa små, endast läsliga mängder data, eller så kan den konfigureras till att visa modifierbara vyer av väldigt stora mängder data. (Microsoft, 2017).

I figur 1 visas ett formulär från Trading Assistant. Den innehåller bland annat en DataGridView Control, som fyllts med aktiedata. De två andra Control objekten är menyn högst upp, och en specialgjord filtreringsruta som är en UserControl.

Figur 1. Exempel på ett formulär innehållande en DataGridView Control, som fyllts med data.

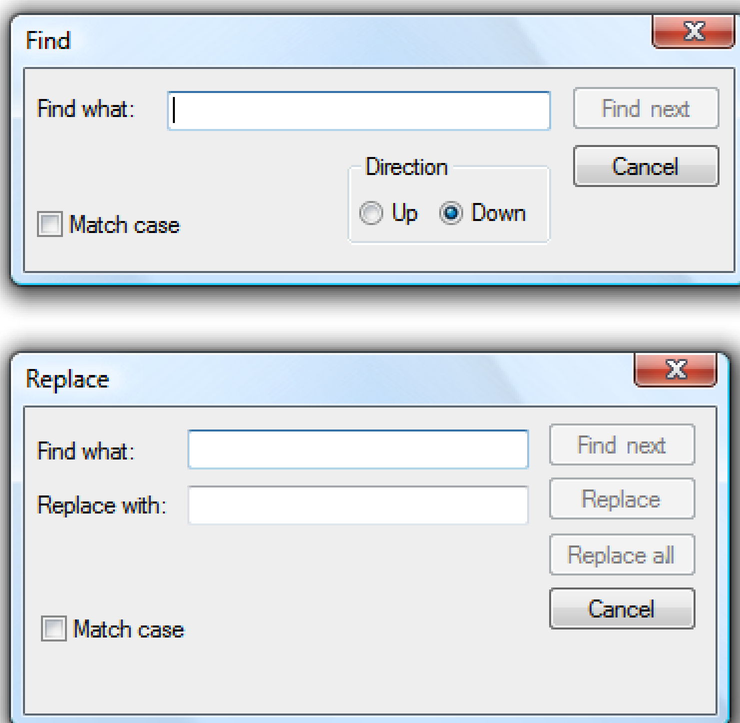
2.4.3 Data Binding

Data Binding i Windows forms erbjuder möjligheten att visa och ändra på information direkt i en datakälla, genom Control-objekt i ett formulär. Data Binding fungerar både med traditionella datakällor och nästan vilken struktur som helst som innehåller data. (Microsoft, 2017).

2.5 Dialogruta

En dialogruta är ett temporärt fönster som en applikation skapar för att ta emot indata från användaren. En dialogruta innehåller i Windows Forms vanligen en eller flera instanser av Control och UserControl genom vilka användaren kan mata in text, välja alternativ, eller bestämma vad som skall hända. Windows erbjuder fördefinierade dialogrutor för vanliga menyer så som Öppna och Skriv ut. (Microsoft, Dialog Boxes).

I figur 2 visas exempel på två vanliga dialogrutor som används för att hitta och ersätta text i Windows-miljö.



Figur 2. Exempel på dialogrutor (Microsoft, Common Dialogs).

2.6 Refaktorisering

Refaktorisering är en teknik för omstrukturering av existerande kod, som ändrar på den interna strukturen utan att ändra på dess funktion (Fowler, Refactoring). En ”code smell” är en ytlig indikation som ofta, men inte alltid, påvisar ett grundligare problem i ett system (Fowler, 2006).

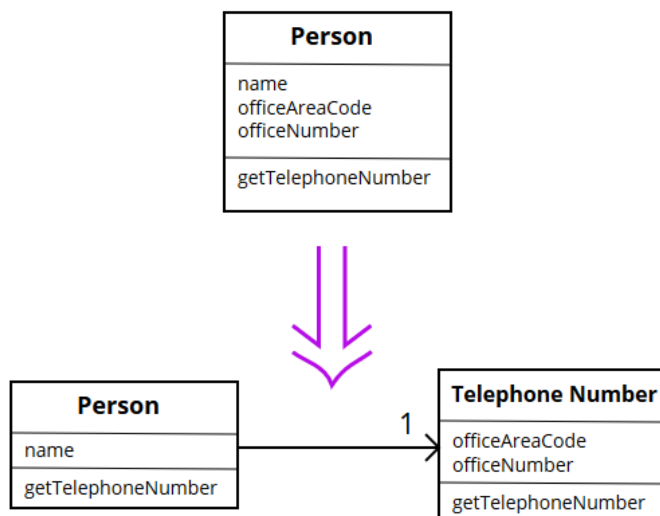
I Trading Assistants kod fanns identifierbara ”code smells” som obehandlade skulle försvåra fortsatt arbete. När någon av dessa upptäcktes under projektet, användes lämpliga refaktoriseringstekniker för att åtgärda dem. Ett exempel är den stora befintliga modellklassen DataModel med flera tusentals rader som hade hand om många olika funktioner i programmet. Denna typ av code smell åtgärdas genom uppdelning i mindre klasser (Sourcemaking, Large Class).

Här följer två av de refaktoriseringstekniker som användes mest under projektet.

2.6.1 Extract Class

Denna teknik tillämpas när man har en klass som utför uppgifter, som borde göras av två skilda klasser (Fowler, 1999).

I figur 3 visas Martin Fowlers exempel på Extract Class. Där har en klass Person till en början ansvaret för telefonnummer, vilket är lämpligt att bryta ut till en helt egen klass Telephone Number. Resultatet blir att kod som hanterar telefonnummer alltid kommer att finnas och modifieras i Telephone Number klassen, utan att påverka Person klassen.



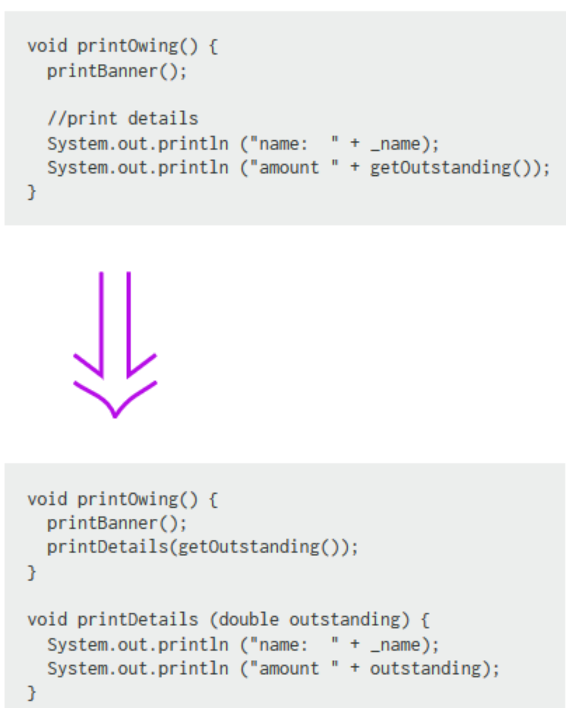
Figur 3. Exempel på Extract Class (Fowler, 1999).

Extract Class tillämpades exempelvis på klassen DataModel som var över 2000 rader lång. Den var viktig för samverkan mellan formulären men svår att förstå, och kunde uppdelas mera för att passa bättre in i den nya strukturen. Många av metoderna i klassen kunde brytas ut till självständiga klasser.

2.6.2 Extract Method

När det finns ett fragment av kod i en klass som kan grupperas på ett ställe, gör man fragmentet till en metod vars namn förklarar metodens syfte (Fowler, 1999).

I figur 4 visas Martin Fowlers exempel på Extract Method, där en metod printOwing i början innehåller ett anrop till metoden PrintBanner samt två anrop för utskrift. Kommentaren ovanför dessa utskriftsanrop används som grund för att bryta ut en ny metod printDetails. Resultatet är en noggrannare uppdelning av ansvar. Metoden printOwing har nu endast ansvaret för att kalla på två metoder i en viss ordning. Ändringar i hur detaljerna skrivs ut kommer nu endast att kräva ändringar i metoden printDetails.



Figur 4. Exempel på Extract Method (Fowler, 1999).

Extract Method användes mycket ofta under omstruktureringen för att bättre dela upp klassers uppgifter internt. När denna teknik tillämpades pragmatiskt fick man snabbt fler

metoder, som vidare kunde grupperas med tekniken Extract Class. På detta sätt kunde strukturen kontinuerligt finfördelas.

2.7 Gränssnitt

Enligt Svenska Akademiens Ordlista betyder ordet gränssnitt: ”förbindelselänk mellan en dator och dess kringutrustning; program som underlättar kontakten mellan dator och användare” (Svenska akademiens ordlista, 2015, 430).

Det finns flera olika typer av gränssnitt, två av dem behandlas ofta i detta arbete.

Det första är en referenstyp i C# som kallas interface. Ett interface innehåller signaturer för metoder, egenskaper, händelser eller indexerare. En klass eller struct-datatyp som implementerar gränssnittet måste implementera medlemmarna i gränssnittets specifikation. (Microsoft, interface (C# Reference)).

Kodexempel 1 visar ett gränssnitt IDisposable som är inbyggt i .NET. Detta gränssnitt implementeras av DisposableClass i kodexempel 2.

Kodexempel 1. Ett gränssnitt i C#.

```
public interface IDisposable
{
    //
    // Summary:
    //     Performs application-defined tasks associated with freeing, releasing, or resetting
    //     unmanaged resources.
    void Dispose();
}
```

Kodexempel 2. En klass som implementerar det tidigare gränssnittet IDisposable.

```
0 references
class DisposableClass : IDisposable
{
    53 references
    public void Dispose()
    {
        ////Disposal code
    }
}
```

Den andra typen av gränssnitt som behandlas ofta i detta arbete är det grafiska användargränssnittet. På engelska benämns detta Graphical User Interface, som förkortas GUI. Detta är den visuella delen av användargränssnittet som överför visuell information från applikationen till användaren.

2.8 Model-View-Presenter

I detta projekt implementerades en vidareutveckling av arkitekturmönstret Model-View-Presenter (MVP). Mönstret utvecklades av företaget Taligent Inc., och grundar sig på det klassiska Model-View-Controller (MVC) mönstret (Potel, 1996, 1).

Syftet med MVP är att maximera mängden kod som kan testas automatiskt. Kod i grafiska användargränssnitt är ofta svåra att testa. MVP strävar också till att separera affärslogik från logik i användargränssnittet, vilket gör koden lättare att förstå och underhålla. (Microsoft, The Model-View-Presenter (MVP) Pattern).

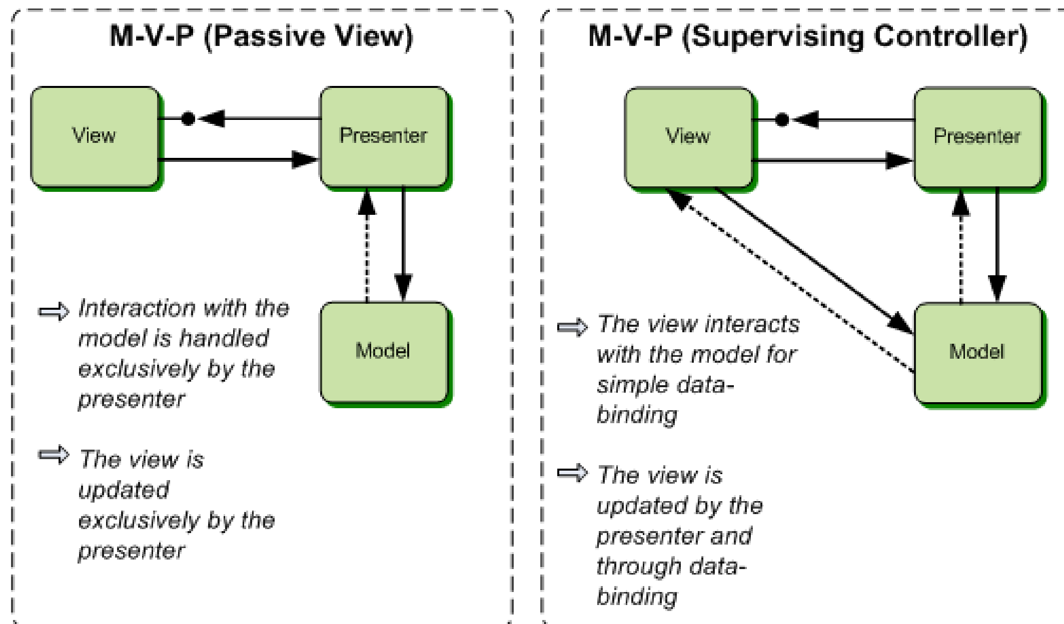
Detta mönster beskriver tre distinkta delar: Model, View och Presenter. Alla dessa tre bildar en MVP-triad (Bower, A. och McGlashan, B., 2000, 4–5). Delarna beskrivs närmare i kapitlet om Passive View, eftersom definitionerna inte är helt likadana som i det ursprungliga MVP mönstret.

2.9 Passive View

Passive View är en vidareutvecklad version av mönstret MVP och blev tidigt i planeringen vald som det mest lämpliga mönstret. Orsaken till att Passive View valdes är att det erbjuder en minimering av mängden uppgifter som koden i användargränssnittet behöver sköta (Fowler, 2006).

Målet med Passive View är också att maximera möjligheterna att automatisera enhetstest av presentationen, detta förutsätter att man flyttar ut så mycket svårtestad kod som möjligt ur View till Presenter (Miller, 2007).

I figur 5 visas en bild av hur Microsoft tolkar detta mönster, tillsammans med den nära besläktade varianten Supervising Controller. Notera skillnaden att Supervising Controller tillåter kopplingar mellan Model och View genom Data Binding.



Figur 5. Microsofts förklaring av Passive View och det liknande mönstret Supervising Controller (Microsoft, Model-View-Presenter).

2.9.1 Presenter

Presenter, också kallad Controller, hänvisar till den komponent som behandlar händelser som uppstått av användarens handlingar. Presenter har också hand om logiken för presentationen i en applikation. (Greer, 2007).

Eftersom Passive View innebär mera ansvar åt Presenter är det befogat att kalla Presenter för Controller, men orden syftar på samma sak i Passive View. För att vara konsekvent och hänvisa till det ursprungliga MVP, användes suffixet Presenter i namnen på klasserna med presentationslogiken. I detta examensarbete används fortsättningsvis ordet presentatör.

Bruket av Passive View medför att presentatören får en betydlig mängd kod, eftersom den ska innehålla all presentationslogik (Fowler, 1999).

2.9.2 View

View hänvisar till det grafiska användargränssnittet, vars uppgifter i Passive View har reducerats till att visa information och ta in data från användaren. Detta är ansvaret som kvarstår, när presentatören ansvarar för presentationslogik, och View inte ansvarar för att uppdatera sig från Model (Fowler, 1999).

I detta examensarbete används i fortsättningen ordet vy för att hänvisa till denna komponent.

Om det blir aktuellt att byta ut en vy som är gjord med annan teknologi, kommer Passive View att underlätta detta arbete. Då koden i användargränssnittet inte innehåller presentationslogiken, behövs det inte så mycket ny kod i den nya vyn.

I detta examensarbete har en vy konstruerats på ett visst sätt för att så långt som möjligt överlåta kontroll åt presentatören. Den konkreta vyn gjordes i Windows Forms. Denna konkreta vy publicerar egenskaper och metoder genom ett gränssnitt, som en presentatör använder för att läsa information i vyn och manipulera dess innehåll. Vyn kommer att kommunicera med händelser, som bara uppmärksammar om det skett något i användargränssnittet.

I kodexempel 3 presenteras ett gränssnitt som används av en vy i en dialog för skapande av nya aktiedelningar. De tre händelserna står för all nödvändig kommunikation från vyn till presentatören. Genom egenskaperna får presentatören tillgång till den data som vyn innehåller, som t.ex. Textfält. Genom metoderna kan presentatören manipulera vyn.

Kodexempel 3. Ett gränssnitt för en vy.

```
interface IAddNewSplitDialogView:IDialogView
{
    event NullEventHandler SaveClicked;
    event NullEventHandler CancelClicked;
    event NullEventHandler InputChanged;
    0 references
    DateTime PickedDate { get; }
    3 references
    int CountBefore { get;}
    3 references
    int CountAfter { get; }
    5 references
    int SelectedStockID { get; set; }
    2 references
    void AddStockRow(int id, string companyname);
    8 references
    void SetStatusLabel(string text, Color color);
    19 references
    void Close();
    3 references
    bool SaveEnabled { get; set; }
}
```

2.9.3 Model

I detta arbete hänvisar Model till den representation av data, som användargränssnittet använder. Detta examensarbete använder hädanefter ordet modell för att hänvisa till denna komponent.

Modellen innehåller data som skall visas i vyn av presentatören. Då denna data uppdateras eller något speciellt händer, så tillkännager modellen detta med händelser. Då kan presentatören använda sig, likt vyn, av egenskaper och metoder för att komma åt data i modellen. I denna applikation var det vanligt att modellerna hade ett ”Stock” objekt som representerade all information om en aktie.

2.10 Enhetstest

Enhetstest är en nivå av mjukvarutest där individuella enheter eller komponenter av en mjukvara testas. Syftet är att försäkra sig om att varje enhet av programmet fungerar som det är tänkt. En enhet är den minsta möjliga delen av ett program som kan testas. (Software Testing Fundamentals, Unit Testing).

I detta examensarbete användes automatiska enhetstest för att kontrollera att logiken i användargränssnittet fungerade efter omstruktureringen.

2.11 Dependency injection

För att göra det lättare att byta ut delar samt att testa komponenterna i mönstret tillämpades tekniken Dependency Injection.

Dependency Injection (DI) är en stil av objektkonfiguration, där ett objekts fält och innehåll manipuleras av ett yttre objekt. Dependency Injection är ett alternativ till att låta objektet konfigurera sig självt. (Jenkov, 2015).

I kodexempel 4 visas en presentatör som genom sin konstruktor injiceras med objekt som har implementerat gränssnitten IDataExplorerView och IDataExplorerModel. I konstruktorn blir referenserna till de yttre beroendena kopierade till privata fält, som sedan kan manipuleras direkt av DataExplorerPresenter.

Kodexempel 4. Dependency Injection i en presentatör.

```
public class DataExplorerPresenter
{
    /// <summary>
    /// The local View which is instanciated through constructor
    /// </summary>
    IDataExplorerView _view;
    IDataExplorerModel _model;
    /// <summary>
    /// Constructor which is injected with instance of View
    /// </summary>
    1 reference
    public DataExplorerPresenter(IDataExplorerView view, IDataExplorerModel model)
    {
        _model = model;
        _view = view;
        _view.NeedMetricsForStock += ViewDemandedMetricsForStock;
        _model.UpdatePresenter += UpdateTreeView;
    }
```

Med hjälp av Dependency Injection kan man göra det lättare att vid behov byta ut olika komponenter av samma typ. Detta utnyttjades i testning av presentatörer.

2.12 Mocking

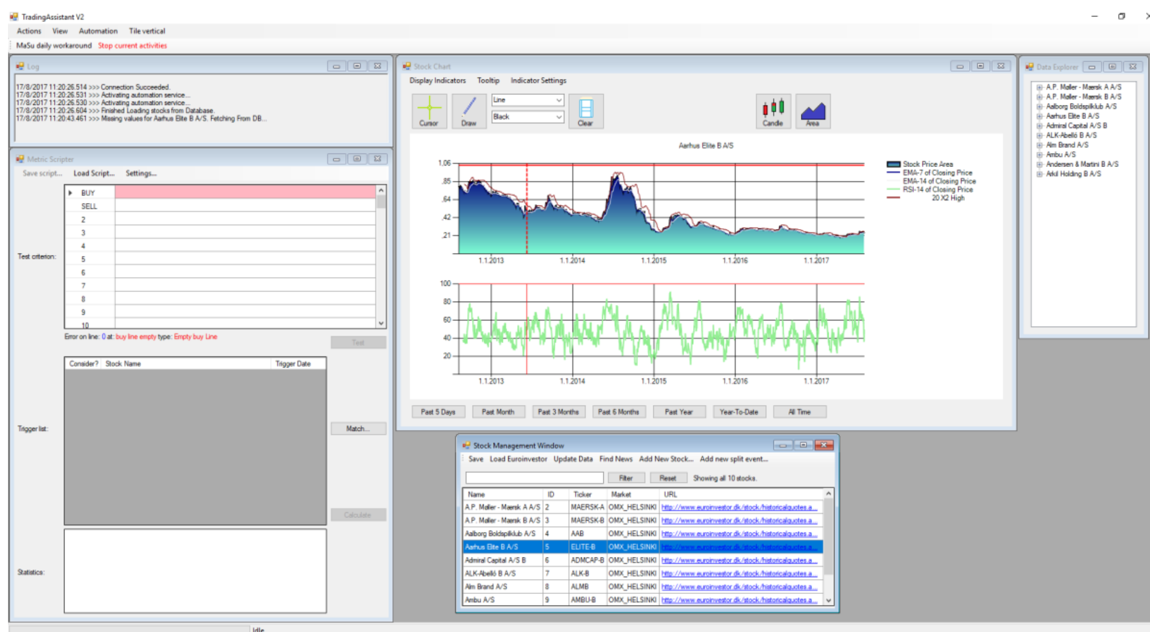
Mocking är en process som används inom enhetstest när enheten som testas har yttre beroenden. Syftet med mocking är att fokusera på koden som testas och inte på beteende eller tillstånd hos yttre beroenden. Mocking innebär att man ersätter dessa beroenden av strikt kontrollerade ersättande objekt, som simulerar beteendet hos de riktiga motsvarigheterna. En mock är ett ersättningsobjekt som kan verifiera att den använts. (Telerik, Unit Testing and Mocking Explained).

Denna process användes i synnerhet under automatiserade enhetstest av presentatörer, för att inte behöva använda de riktiga implementeringarna av vy eller modell. De ersattes i testen av egenhändigt gjorda mock-objekt som implementerade samma gränssnitt, men dessutom innehöll kod för att kontrollera hur en presentatör påverkat dem.

I examensarbetet översätts inte termerna mock och mocking, eftersom det saknas lämpliga motsvarigheter på svenska.

3 Programmet

Examensarbetet har gjorts på programmet Trading Assistant, som beskrivs närmare i detta kapitel. Programmet är ett verktyg för analys av aktiehandel. I figur 6 presenteras det huvudsakliga användargränssnittet i Trading Assistant.



Figur 6. Användargränssnittet i Trading Assistant.

3.1 Användningsområde

Detta program används till att hantera en stor mängd data om aktier och börsvärden. Med denna data kan programmet rita upp grafer samt göra flexibel teknisk analys. Användaren av programmet kan själv skriva skript med ett unikt språk för att skapa köp- och säljpåminnelser, samt studera hur dessa påminnelser skulle ha fungerat på historiska data.

Programmet är för tillfället gjort endast för användare med hög kännedom av ekonomiska termer och teknisk analys, samt god tolkningsförmåga av aktiekurser. I framtiden kommer programmet att utvecklas till att vara mera tillgängligt för lekmän.

3.2 Befintliga funktioner

När detta projekt påbörjades hade programmet redan hunnit utvecklas under en längre tid. Programmet kunde ladda ner börsdata från internet, lagra det i en databas samt visa data på skärmen i grafer och tabeller. Det gick även att visa resultat av teknisk analys på aktier och köra enkla skript.

3.3 Tekniska detaljer

Programmet har skrivits i C# och är utvecklat på plattformen Windows Forms. Programmet använder flertrådsteknik.

Programmet är specifikt designat att köras mot en MySQL databas som är inställd för att tillåta de läsningar och skrivningar som programmet genomför. Historikvärden laddas ned till databasen från webbsidor som offentliggör historikdata dagligen.

4 Verktyg

I detta kapitel sammanfattas några av de verktyg och tjänster som användes under arbetet. Dessa verktyg valdes baserat på tidigare erfarenheter och kostnad. De flesta av verktygen användes redan av företaget.

4.1 Utvecklingsmiljö

Som utvecklingsmiljö valdes Microsoft Visual Studio Community 2015 för att den är Microsofts egen produkt och har använts för att utveckla detta program tidigare. Programmet klarar också av att integrera en klient av det använda versionshanteringssystemet.

Visual studio ger också möjligheten att konstruera och köra automatiserade enhetstest på producerad kod. Denna funktionalitet utnyttjades ofta under arbetet.

4.2 Versionshantering

Som versionshanteringssystem valdes Git, i stället för alternativet Apache Subversion. Git valdes för att se om det var ett fungerande alternativ, som också kunde användas i framtiden. En visuell Git klient användes som en integration i Visual Studio, men kommandotolk användes ofta för mer avancerade uppgifter.

4.3 Nätbaserade tjänster

Jira är namnet på det webbaserade projekthanteringssystemet från Atlassian som användes under projektet. Detta system valdes för att det redan användes inom företaget.

För att lagra allt versionshanterat material användes tjänsten Atlassian Bitbucket. Denna tjänst valdes för att den är gratis för upp till fem användare, stöder versionshanteringssystemet Git, och för att den integrerar sig automatiskt med Jira. Denna integration möjliggör bl.a. att en uppdatering av källkoden kan göras synlig i projekthanteringen.

5 Befintlig kod

I detta kapitel presenteras några exempel ur den ursprungliga koden, med syftet att påvisa behovet av omstruktureringen. Motiveringarna görs med avskalade kodexempel från den befintliga källkoden. För att fokusera uppmärksamheten till det som är relevant, har vissa triviala kodrader tagits bort.

5.1 Kodexempel från MainWindow

I kodexempel 5 ses ett utdrag av koden i början av programmets huvudsakliga formulär, MainWindow. Det första som kan noteras är att alla formulär i programmet också existerar här. Det betyder att MainWindow, som egentligen är ett grafiskt användargränssnitt, ansvarar för skapande och upprätthållande av andra grafiska användargränssnitt.

Kodexempel 5. Utdrag ur koden i det centrala formuläret, MainWindow.

```
12 references
public partial class MainWindow : Form
{
    private StockMgmtWindow sMgmtWindow = null;
    private StockChartWindow sChartWindow = null;
    private LogWindow sLogWindow = null;
    private DataExplorerWindow sExploreWindow = null;
    private MetricTestWindow sTestWindow = null;

    private DataModel sDataModel = new DataModel();
    private BackgroundWorker bwWorker = null;

    private MySqlConnection sDbConnection = null;


    private Thread TimeOfDayThread = null;
    private bool bRunThreads = true;

    private bool bLoadingStocks = false;
    private bool bUpdatingStocks = false;
    private bool bCheckingNews = false;
    private bool bAnalysingStocks = false;

    private BackgroundWorker oApplicationWorker = null;

    1 reference
    public MainWindow()
    {
```

Ur kodexempel 5 kan också noteras att den huvudsakliga instansen av DataModel existerar här. Detta är ett mycket större problem, eftersom nu ansvarar användargränssnittet för skapandet och hanteringen av hela modellen, som utgör grunden för applikationen.

Annat som kan noteras ur kodexempel 5 är instanserna av Thread och BackgroundWorker klasserna, som används för att låta applikationer köras på flera trådar. MainWindow har då ansvaret för trådhantering, som inte är en funktion relaterad till användargränssnittet.

Endast genom att se på några av de privata medlemmarna som fanns i MainWindow blev det möjligt att identifiera problem med ansvarsfördelningen i applikationen. Koden i MainWindow innehöll mycket som inte alls hade något med grafiska användargränssnitt att göra.

I kodexempel 6 presenteras en till metod från MainWindow. Denna metod ansvarar för att sända e-post om intressanta nyhetshändelser. Att sända e-post är en funktionalitet som i detta fall inte har något med grafiska användargränssnitt att göra.

Kodexempel 6. En metod i MainWindow som använder SmtpClient.

```
1 reference
private void SendEmailAboutNewsPieces(string subject, string body)
{
    using (MailMessage mail = new MailMessage())
    {
        mail.Subject = subject;
        mail.Body = body;
        mail.IsBodyHtml = true;
        mail.From = new MailAddress(emailFrom);
        mail.To.Add(emailTo);
        mail.CC.Add(emailFrom);

        //mail.Attachments.Add(new Attachment("C:\\SomeFile.txt"));
        //mail.Attachments.Add(new Attachment("C:\\SomeZip.zip"));

        using (SmtpClient smtp = new SmtpClient(smtpAddress, portNumber))
        {
            smtp.Credentials = new NetworkCredential(emailFrom, password);
            smtp.EnableSsl = enableSSL;
            smtp.Send(mail);
        }
    }
}
```

5.2 Kodexempel från MetricTestWindow

I kodexempel 7 visas en metod som hanterar ett klick på en knapp i MetricTestWindow. Denna kod är i det grafiska användargränssnittet, men har en del ansvar utanför området. Något som inte ses i kodexemplet är att triggerListView är en DataGridView, och lTriggerEvents är en lista med TriggerEvent objekt.

Det första som kan noteras ur metoden är att den går igenom Stock objekten som finns i modellen, samt gör avancerade analyser på dessa. Resultatet av analysen blir en lista med TriggerEvent objekt, som på samma gång läggs in i en lista på skärmen. Denna metod ansvarar ensam för att hantera knapptryckningen, göra analys på alla Stock objekt i modellen, hantera analysresultaten, och uppdatera användargränssnittet.

Det andra som kan noteras är att den sista raden, recalculateStatsButton.Enabled får värdet sant. Detta innebär att en annan knapp i användargränssnittet blir aktiverad. Detta är ett exempel på presentationslogik som kunde separeras från det grafiska användargränssnittet.

Kodexempel 7. En metod som hanterar ett klick på en knapp i MetricTestWindow, ett formulär för test av skript.

```
1 reference
private void testMetricButton_Click(object sender, EventArgs e)
{
    lTriggerEvents.Clear();
    triggerListView.Rows.Clear();

    // Run technical analysis to find trigger events
    for (int i = 0; i < sDataModel.Stocks.Count; i++)
    {
        Stock s = sDataModel.Stocks[i];
        List<TriggerEvent> lEvents = TechnicalAnalysis.AnalyseCombination(s, sParsedCombination);
        if (lEvents.Count > 0)
        {
            lTriggerEvents.AddRange(lEvents);
        }
    }

    for(int i = 0; i < lTriggerEvents.Count; i++)
    {
        TriggerEvent sEvent = lTriggerEvents[i];
        int iIndex = triggerListView.Rows.Add(true, sEvent.ConcernedStock.Name, sEvent.TriggerDate);
        triggerListView.Rows[iIndex].Tag = sEvent;
    }

    recalculateStatsButton.Enabled = true;
}
```

5.3 Kodexempel från StockMgmtWindow

StockMgmtWindow är ett formulär som skall hantera visning av tillgängliga aktier.

Kodexempel 8 visar en metod som hanterar ett knapptryck för att ladda ned aktiedata från internet. Denna metod finns i koden för StockMgmtWindow och är för lång för att visas i detta examensarbete, vilket kan observeras från de två radnumren. Denna metod innehåller all kod som behövs för att ladda ned aktiedata från en sida på internet, vilket inte alls har något med användargränssnittet att göra.

Kodexempel 8. Metod i StockMgmtWindow.

```
            1 reference
192  [+]    private void LoadEuroinvestorStocksButton_Click(object sender, EventArgs e)...
303
```

5.4 Kodexempel från DataModel

DataModel var i början den klass som innehöll största delen av all modellkod i applikationen. I kodexempel 9 sammanfattas alla dess publika metoder och egenskaper. Denna klass hade ansvar för en stor mängd olika uppgifter och var mycket svårläst.

Kodexempel 9. Ett sammandrag av publika metoder och egenskaper i klassen DataModel.

```
2 references
bool DataLoaded { get; }
1 reference
int NextInternalId { get; }
5 references
MainWindow ParentWindow { get; set; }
51 references
List<Stock> Stocks { get; set; }

2 references
void AnalyzeStocks(object senderBw, bool bCalculateNewValues, bool bSaveToExcelFiles);
1 reference
void ConvertXmlNewsURLs();
2 references
void GetNewsURLs(object senderBw);
2 references
Stock GetStockById(int id);
4 references
List<CalculatedMetric> GetTAMetricsForStockFromDB(Stock s);
5 references
List<DayRecord> GetTradingDaystatsForStockFromDB(Stock s);
2 references
void InspectWWWNews(ref List<StockNewsPieceComb> lStocksNews, object senderBw);
1 reference
void LoadedId(int id);
2 references
void LoadStocksInfoFromDB(BackgroundWorker sender);
3 references
int ReserveNextFreeIntenalId();
2 references
void SaveAllToDB();
3 references
void SaveStockToDB(Stock s, bool bSaveStockInfo, bool bSaveTradingStats, bool bSaveMetrics);
1 reference
void SaveToXML(BackgroundWorker bw);
2 references
void TestCombination();
2 references
void TestMetric(Stock sToTest, object sPrm);
2 references
void UpdateStocksHistoricalData(object senderBw);
```

Denna klass anropades ofta direkt från användargränssnittet, och kunde lämpligen struktureras om tillsammans med koden i presentationslagret.

5.5 Sammanfattning av kodexempel

Dessa kodexempel visar på att det fanns en del problem i koden för programmet. För det första hade de grafiska användargränssnitten fått ansvaret för en hel del uppgifter som skulle vara mera lämpliga om de implementerades på modellnivå.

Det andra som kan konstateras är att all denna kod var mycket svår att testa med automatiska enhetstest. Efter några ändringar i koden måste allt testas manuellt genom att köra programmet och debugga, vilket skulle ta en hel del tid. Detta är ett problem som också hade sitt ursprung i att de flesta klasserna inte programmerats mot gränssnitt.

6 Planering och målsättning

Efter att behovet av en omstrukturering klargjorts, påbörjades planeringen. Det första som gjordes var att forska i hur omstruktureringen skulle genomföras. Då syftet var att få ut logik från användargränssnittet i en Windows Forms applikation, blev det snabbt klart att mönstret Passive View skulle vara lämpligt för detta. Dessutom skulle presentationslogiken bli möjlig att testa, vilket var en viktig fördel.

Som första mål sattes att den existerande koden skulle omstruktureras till Passive View. För att dra nytta av mönstret blev det andra målet att göra enhetstest kring presentationslogiken.

Det tredje målet blev att möjliggöra upprätthållandet av denna struktur efter att den implementerats. Förverkligandet av detta mål skulle bli en dokumentation som beskriver applikationens struktur. Denna skulle utformas för att vara till nytta för utvecklare som inte är bekanta med koden.

7 Genomförande av omstrukturering

Detta kapitel sammanfattar det arbete som gjordes i det inledande projektet, med målet att strukturera om den befintliga koden i användargränssnittet. Det är viktigt att understryka att projektet innebar en fullkomlig omstrukturering och inte bara en refaktorisering. Enligt Martin Fowler så är det inte refaktorisering om systemet sätts ur funktion i några dagar, vilket var vanligt under projektet (Fowler, 2004).

Innan projektet påbörjades, flyttade arbetsgivaren grundläggande klasser för aktier och teknisk analys ut till ett separat bibliotek. Detta eliminerade risken att dessa kritiska delar skulle påverkas av arbetet på användargränssnittskoden. Den kod som på något vis var beroende av Passive View implementeringen flyttades inte till biblioteket.

Den befintliga koden var uppdelad i fem formulär som tillhörde ett huvudformulär. Dessa formulär strukturerades om en åt gången. Arbetsmetoderna förbättrades för varje formulär som omstrukturerades, och bidrog till en ökad förståelse för mönstret Passive View. Vissa undantag gjordes p.g.a. begränsningar i Windows Forms och presenteras i kapitel åtta.

Koden som fanns i användargränssnittet var inte enhetstestad, och eftersom denna kod helt skulle plockas isär, var det inte gångbart att göra förberedande enhetstest. Därför gjordes arbetet så pragmatiskt som möjligt för att undvika problem.

7.1 Utflyttning av kod från användargränssnittet

Det första som gjordes på formulären var att flytta ut kod från den bakomliggande filen för användargränssnittet. Dessa filer innehöll, förutom användargränssnittskod, ofta kod för beräkningar och logik som inte användargränssnittet skall ansvara för.

I kodexempel 10 ses en metod som hanterar ändringar i ett textfält. I metoden TriggerString.ParseString görs en tolkning av en textsträng, som möjligen kan producera ett resultatobjekt. Detta är affärslogik som inte är relaterad till användargränssnittet, och är ett exempel på kod som flyttades ut.

Notera att det i ett if-else uttryck görs ett färgval baserat på om sParsedCombination är null eller ej. Detta är ett exempel på presentationslogik som flyttades ut från

användargränssnittet. Denna typ av logik kunde vara svår att uppmärksamma, eftersom den ofta var kombinerad med Windows Forms kod.

Kodexempel 10. Kod i användargränssnittet med presentationslogik.

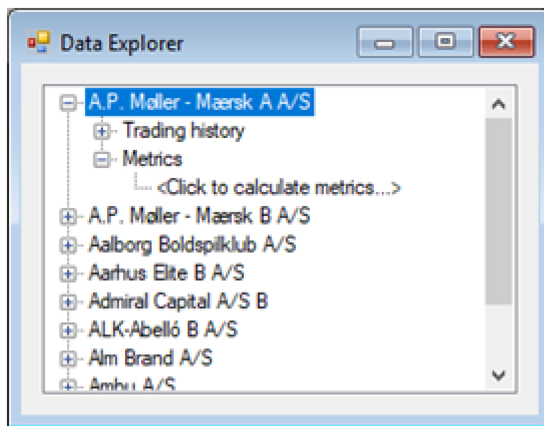
```
private void testCriterionTextBox_TextChanged(object sender, EventArgs e)
{
    string sCrit = testCriterionTextBox.Text;
    int iError = 0;
    sParsedCombination = TriggerString.ParseString(sCrit, out iError);

    if (sParsedCombination == null)
    {
        testCriterionTextBox.BackColor = Color.Bisque;
    }
    else
    {
        testCriterionTextBox.BackColor = SystemColors.Window;
    }
}
```

Koden som flyttades ut lades i en temporär fil, för att säkerställa att ingen kod försvunnit efter utflyttningen.

7.2 Konstruktion av vy och dess gränssnitt

Först identifierades vilka händelser, egenskaper och metoder som skulle behövas i en vy. Utifrån dessa gjordes sedan ett gränssnitt som var oberoende av Windows Forms kod, för att möjliggöra utbyte av användargränssnittets teknologi. Något som också togs i beaktande var hur mycket kontroll av den grafiska logiken som behövdes. Eftersom presentatören kontrollerar beteende hos vyn genom ett gränssnitt var det viktigt att fastställa vilka delar av gränssnittet som faktiskt behövdes, det fanns aldrig behov av kontroll över alla element i det grafiska användargränssnittet.

I figur 7 ses formuläret Data Explorer med motsvarande gränssnittskod i kodexempel 11. Fönstret visar data i en trädstruktur från aktier som laddats in i programmet, och användaren kan klicka för att genomföra teknisk analys vid behov. Notera hur gränssnittet IDataExplorerView är konstruerat med en enda metod för att ladda in alla noder på en gång. Den har en också en metod för att uppdatera en enda aktienod. Denna metod används då användaren begär analysresultat för en aktie.



Figur 7. Formuläret Data Explorer.

Kodexempel 11. Gränssnittet IDataExplorerView.

```
3 references
public interface IDataExplorerView
{
    /// <summary>
    /// Event fired when user clicks on empty stock metric marker, signal to calculate metrics in model
    /// </summary>
    event StringEventHandler NeedMetricsForStock;
    /// <summary>
    /// Send ALL available stock data to view (stock tags are populated with stockviewData objects)
    /// </summary>
    2 references
    void PopulateWithStockData(List<StockViewData> datalist);
    /// <summary>
    /// Send ONE specific stock data to view,usually when something has changed in one stock
    /// </summary>
    3 references
    void UpdateSingleStockNode(StockViewData data);
}
```

Det konstaterades vara lämpligt att göra formulär till vyer, i stället för att dela upp dem i UserControls. Motiveringen var att om ett formulär måste innehålla mycket kod, så kan det tyda på att det har för många uppgifter. Detta var inte ett problem med de existerande formulären.

7.3 Konstruktion av Presenter

Nästa steg blev att konstruera en presentatör per vy. Den fil som innehöll kod som flyttats ut från ett användargränssnitt, användes som grund för innehållet i en presentatör. Eftersom gränssnittet för modellen inte ännu blivit konstruerad, användes en direkt referens till den stora klassen DataModel, som fortfarande utgjorde grunden för applikationen. Detta gjordes som ett mellansteg för att lättare kontrollera fel som uppstått när kod flyttats ut från vyn.

Logiken för presentatörerna konstruerades och testades tills den motsvarade det tidigare beteendet. Mycket vikt lades på att inte skriva Windows Forms kod i presentatörerna.

Kodexempel 12 visar en bit ur en färdig presentatör som innehåller en konstruktor och en metod. I konstruktorn kopplar presentatören ihop modell och vy genom att prenumerera på händelser. Metoden hanterar ett klick i vyn, som signalerar modellen att uppdatera sig.

Kodexempel 12. Konstruktor och en metod från presentatören DataExplorerPresenter.

```
public DataExplorerPresenter(IDataExplorerView view, IDataExplorerModel model)
{
    _model = model;
    _view = view;
    _view.NeedMetricsForStock += ViewDemandedMetricsForStock;
    _model.UpdatePresenter += UpdateTreeView;
    _view.StockNodeClicked += _view_StockNodeClicked;
}

1 reference
private async void _view_StockNodeClicked(string stockname)
{
    Stock stock = (_model.Stocks.Find(x => x.CompanyName == stockname));
    bool result = await  _model.IsStockHealthy(stock);
    if(result)
    _view.UpdateSingleStockNode(new StockViewData(stock, _model.GetMetricsForStock(stock)));
}
```

7.4 Konstruktion av Model

Den existerande implementeringen av modellkoden var en enda stor klass DataModel med tusentals rader kod, massa metoder för helt olika syften och en stor mängd publika variabler. Det skulle ha varit möjligt att använda denna klass som modell direkt, men då skulle den fyllas med kod som flyttats ut från flera olika formulär. Därför gjordes valet att göra modeller för varje formulär, så att lokal funktionalitet kunde modifieras åtskilt från gemensam funktionalitet. Ifall det fanns kod i DataModel som endast behövdes i ett formulär, skulle denna kod flyttas till motsvarande formulärs modell. För att underlätta identifieringen av denna kod, var det nödvändigt att dela upp DataModel i mindre delar.

Ur DataModel flyttades en hel del klasser ut med refaktoriseringstekniken Extract Class (se figur 8). En del klasser gjordes runt tillhörande gränssnitt för att möjliggöra utvecklingen av liknande komponenter med andra funktionsprinciper. Dessa klasser kunde sedan individuellt justeras utan att påverka någon annan del av programmet.

Figur 8. Klasser som flyttades ut ur DataModel.

Efter att all funktionalitet i brutits ut till separata klasser, återstod endast ett ansvar för DataModel. Klassen skall i fortsättningen endast agera som en lättillgänglig behållare för gemensamma applikationskomponenter.

I kodexempel 13 visas en del av en modell vid namn StockManagementModel. I konstruktorn får modellen tillgång till komponenter i DataModel via DI, och kopplar ihop det som behövs. Observera prenumerationen som kopplas till UpdatePresenter; när den huvudsakliga aktielistan ändras, uppmärksammar den detta genom en händelse StockDataUpdated. Denna prenumererar modellen på, och hanterar den i metoden UpdatePresenter som i sin tur avfyrar en egen händelse ModelUpdated. Presentatören har då möjlighet att prenumerera på denna händelse, och få veta när modellen uppdaterats.

De övriga metoderna i kodexempel 13 hanterar lokal affärslogik, som är specifik för denna modell. T.ex. LoadEuroInvestorStocks kommer att anropas då användaren har begärt att aktier skall laddas ned från internet. Detta skall bara kunna göras i denna modell och triad.

Kodexempel 13. Konstruktor och metoder i StockManagementModel.

```
public StockManagementModel(ref DataModel model)
{
    _container = model.StockContainerInstance;
    _communicator = model.Communicator;
    _manager = model.StockIDManager;
    _container.StockDataUpdated += UpdatePresenter;
    _webservice = model.Webservice;
    _reposervice = model.DBService;
    _model = model;
}
1 reference
void UpdatePresenter()
{
    ModelUpdated?.Invoke();
}
2 references
public void ConvertXmlNewsURLs()...
2 references
public void LookForNews()...
2 references
public async void GetNewStockDataAndHistory()...
2 references
public void LoadEuroInvestorStocks()...
```

7.5 Konstruktion av TradingAssistantContext

Då en Windows Forms applikation skall startas, görs det vanligen genom att det huvudsakliga formulärets Show metod kallas. Detta går emot Passive View eftersom ett formulär, det grafiska användargränssnittet, har ansvaret att fungera som en startpunkt för applikationen.

Ett sätt att ge mer kontroll över startproceduren var att implementera en klass som ärver från ApplicationContext, som kan användas som startpunkt. Denna gavs då ansvaret att manipulera livstiden för applikationens grundläggande komponenter. Den fick t.ex. ansvaret för skapandet av alla MVP-triader och grundläggande tjänster. Sedan när programmet körs så är det kontexten som startas och gör nödvändiga förberedelser. Då allt är klart, visas alla formulär som behövs för de centrala funktionerna.

Eftersom kontextklassen innehåller alla instanser av modell, vy och presentatör som programmet huvudsakligen består av, var det lämpligt att placera hanteringen av formulärens synlighet i denna klass. I något skede kommer denna funktion att hanteras av en skild komponent.

I kodexempel 14 visas ett utdrag ur en metod i TradingAssistantContext där en triad kopplas ihop. Den konkreta vyn är en StockChartView som är ett formulär. Denna vy injiceras i

konstruktorn för StockChartPresenter, tillsammans med en instans av StockChartModel. Den grundläggande klassen DataModel som innehåller kod som delas av flera formulär, injiceras samtidigt in i konstruktorn för StockChartModel.

Kodexempel 14. Exempel på hur en triads instanser kopplas ihop i TradingAssistantContext.

```
StockChartViewInstance = new StockChartView();
StockChartPresenterInstance = new StockChartPresenter(StockChartViewInstance, new StockChartModel(ref sDataModel));
StockChartViewInstance.MdiParent = MainWindowViewInstance;
ActiveForms.Add(StockChartViewInstance);
```

8 Designval

I detta kapitel motiveras specifika val som gjordes under arbetet. Det är viktigt att understryka att ett arkitekturmönster används för att lösa problem, inte för att skapa nya. I vissa fall skulle det bli mycket mera komplicerad kod som är svår att upprätthålla, om Passive View följdes slaviskt.

8.1 Treeview

TreeView är en komponent i användargränssnittet som har en trädliknande struktur. Ett exempel på en sådan kan ses i figur 7. Baserat på det tidigare valet att endast ha Windows Forms kod i vyer, blev det problem när koden skulle göras till Passive View.

Det skulle bli mycket komplicerad kod i presentatören för att fullständigt kontrollera logiken för en TreeView. Ett sätt att åstadkomma detta kunde ha varit att ha en modell som innehåller hela den visuella trädstrukturen, med tillståndsvariabler som håller reda på vilka noder som är öppnade.

För att undvika att göra denna mycket komplexa implementering, gjordes valet att låta vyn hantera logiken för klick på noder. Detta innebär att den klickade noden visar data som lagrats i den. Presentatören har kvar ansvaret att konvertera och skicka den data som skall visas till vyn.

8.2 Indata från mus och tangentbord

Hanteringen av musrörelser och musklick, samt indata från tangentbordet hanterades i vyerna. Detta gjordes för att Windows Forms hanterar indata genom att skicka ut händelser vars parametrar är exklusiva för denna teknologi. Dessa händelser fångades upp i vyerna, och skickades vidare med egenhändigt utvecklade händelser vars parametrar inte var beroende av Windows Forms klasser.

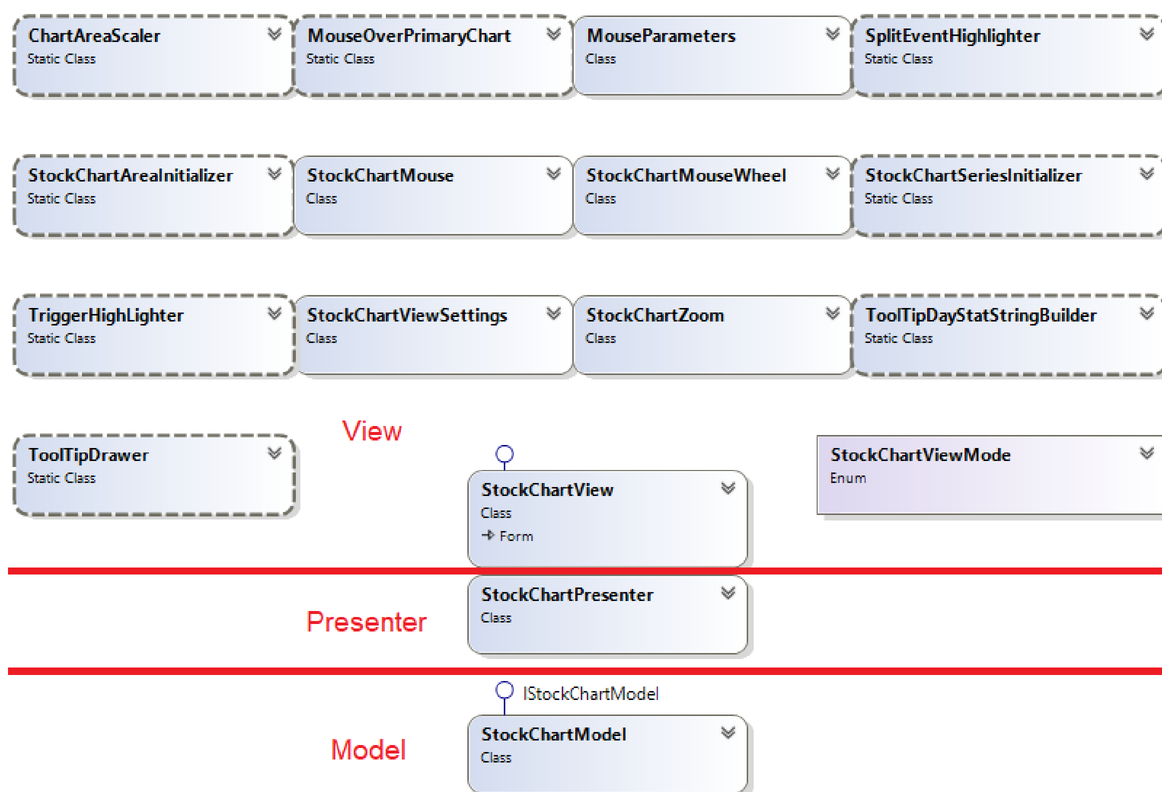
8.3 Grafritning

Den inbyggda Chart klassen i Windows Forms som används för att rita grafer, lagrar och hanterar data internt. Eftersom Chart är en komponent som är ett grafiskt användargränssnitt, men samtidigt också en modellrepresentation med tillhörande manipulering, så är den i konflikt med Passive View.

Den presentatör som skulle hantera formuläret med en Chart, fick därmed syftet att endast kontrollera förberedelserna för grafritning, samt att koppla ihop grafen med omkringliggande element i användargränssnittet.

Detta innebar att det samlades en stor del kod i vyn, eftersom all grafritningskod måste existera och manipuleras i det formulär där den finns. Det var dock möjligt att bryta ut en hel del klasser från detta formulär.

I figur 9 visas klasserna som är relaterade till grafritning av aktiedata och hur de sitter ihop i sammanhanget. Värt att notera är att alla dessa utbrutna klasser är gjorda runt teknologin som används i Windows Forms, och kommer att bytas ut om denna teknologi byts till t.ex. Windows Presentation Foundation (WPF). Detta är en stark motivering till varför Windows Forms-relaterad kod aldrig placerats i presentatörerna under detta projekt.



Figur 9. Klasserna för grafritning av aktiedata, uppdelade i Model, View och Presenter.

8.4 Data Binding

Eftersom Passive View strävar till största möjliga separation har Data Binding i allmänhet undvikits, eftersom det kopplar en modell mycket starkt till det grafiska användargränssnittet. Sättet som Data Binding fungerar på är specifikt för användargränssnittets teknologi.

Ett försök att implementera Data Binding gjordes genom att koppla en DataGridView till en lista med inställningar. Problemet är att Data Binding i Windows Forms kräver att modellen görs direkt tillgänglig i koden för vyn för att åstadkomma kopplingen som behövs. Detta var i direkt konflikt med Passive View. Om strukturen var gjord runt mönstret Supervising Controller, som tillåter direkt koppling mellan modell och vy, skulle Data Binding vara mera lämpligt.

8.5 Flera trådar

För att låta användargränssnittet fungera ostört då långa beräkningar görs, hade programmet från början konstruerats för att köras på flera trådar.

Applikationen använde i stor omfattning BackgroundWorker klassen för att köra bakgrundsarbete på flera trådar. BackgroundWorkerns design är föråldrad, den funkar till exempel inte så bra med asynkron kod. Det överlägsna alternativet är att använda Task.Run. (Cleary, S., 2013, Task.Run vs BackgroundWorker: Intro).

Alla existerande BackgroundWorker instanser byttes ut till asynkron kod som utnyttjar Task.Run.

9 Efterarbete

Efter omstruktureringen gjordes en resultatbedömning av arbetet. Eftersom de existerande formulären var omstrukturerade och applikationen fungerade, var det lämpligt att påbörja arbetet på nya funktioner.

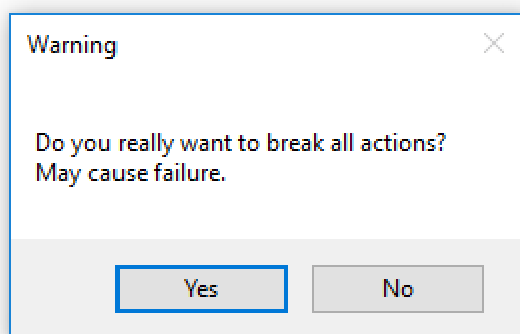
I detta kapitel behandlas arbetet på nya komponenter enligt den nya strukturen. Innehållet fokuserar på koncept, som det vid tidpunkten för utförandet endast fanns begränsad information om.

Trots att projektet för omstruktureringen var avslutat kunde det fortfarande göras ändringar i arkitekturen. Under den fortsatta utvecklingen av applikationen blev syftet med Passive View klarare och nödvändiga ändringar gjordes efter behov. Bland annat hade några fåtal rader med logik lämnats kvar i några formulär, som genast flyttades ut i motsvarande presentatörer när de upptäcktes.

9.1 Passive View och dialogrutor

En dialogruta ”pausar” exekveringen på den aktuella tråden för användargränssnitt, och väntar på att användaren ger tillåtelse att fortsätta.

I Windows Forms så är enkla dialogrutor tillgängliga med MessageBox objektet, men det erbjuder bara väldigt begränsade inställningar. Figur 10 är ett exempel på en enkel MessageBox som endast returnerar ja eller nej som resultat.



Figur 10. Exempel på en MessageBox.

Mer avancerade dialogrutor kan konstrueras med ett formulär som visas med en skild ShowDialog metod, men då måste hela dialogrutans struktur göras från grunden.

När det fanns behov av specialiserade dialogrutor gjordes de enligt det använda Passive View mönstret. Eftersom processen var bekant och strukturens syfte var klart, gick det snabbt att implementera dessa komponenter. Arbetet på dialogrutorna gav också insikt i hur tidigare implementeringar av Passive View kunde förbättras.

En del frågetecken uppstod i arbetet gällande var dialogrutorna skulle skapas och köras. Valet blev att den modala dialogen skapas och körs där den behövs, oavsett var i koden det gäller. Motiveringen för detta val grundar sig på att dialogrutan är som en liten applikation i sig. Dess syfte är att göra ett uppehåll för att kommunicera med användaren, och sedan försvinna. Vanligtvis skapades dialogrutor i en presentatörs kod, men också i t.ex. i procedurer som hämtar data från databasen.

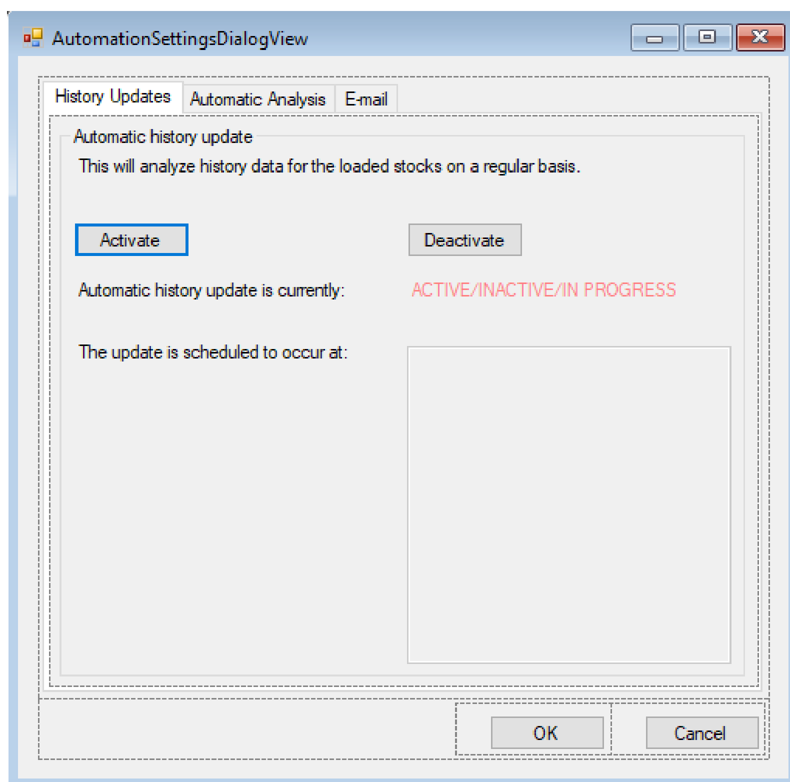
Enklare dialoger gjordes med MessageBox objektet. Ett exempel på detta är då applikationen startas. Då visas en dialog för att tillåta att användaren återskapar databasen om den skulle saknas, något som endast behöver resultatet ja eller nej.

9.2 Undervyer

De huvudsakliga formulären som programmet består av gjordes till vyer. Detta var främst möjligt p.g.a. att formulärens innehåll var relativt kompakta, och det fanns få tydliga behov av en större uppdelning. Komplexa dialogrutor måste däremot göras annorlunda.

Ett exempel där UserControl-vyer användes är en dialog för automationsinställningar (se figur 11), konstruerad med en s.k. TabControl som grund. Denna inbyggda Control

möjliggör snabba byten mellan olika sidor, s.k. Tab Pages. Med hjälp av denna komponent för användargränssnitt kunde de olika inställningarna delas upp i grupper.



Figur 11. Dialog för automationsinställningar, som gjordes med Composite View.

Det var stor risk att innehållet i de olika inställningsgrupperna skulle ändras, men innehållet behövde inte ändras dynamiskt under exekvering.

Ett problem var att det inte fanns mycket tillgänglig information om hur detta kunde genomföras i praktiken. Något som bidrog till lösningen var konceptet Composite View. Detta koncept innebär att en huvudsaklig vy gör varenda undervy tillgänglig, och motsvarande huvudsakliga presentatör åläggs hanteringen av dem (Wilson, B., 2008).

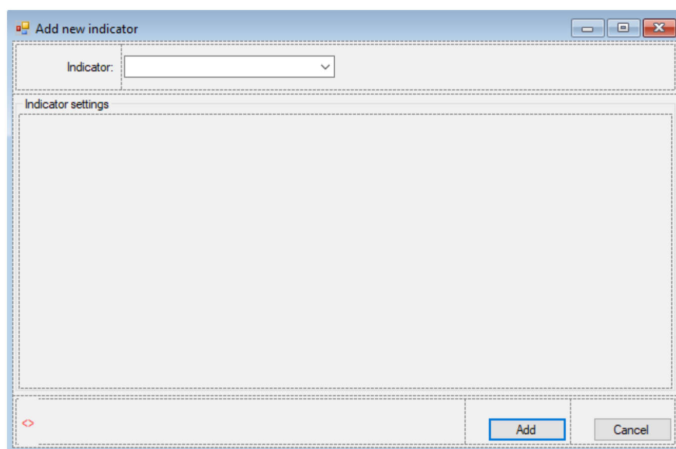
Därför gjordes skilda UserControl-vyer för varje sida, som grupperar liknande inställningar. Det blev en vy per UserControl, samt en vy för det formulär som var till grund för dialogrutan. Detta koncept utvecklades ytterligare genom att skapa en hel MVP-triad för varje UserControl som representerade en Tab Page. Lösningens klassdiagram, som finns i dokumentationen, presenteras i bilaga 1.

Då Windows Forms vanligtvis skapar de konkreta instanserna av UserControl-objekt i koden i den bakomliggande designer filen, var det en utmaning att koppla ihop UserControl-undervyerna till sina egna presentatörer och modeller. Dessa bedömdes vara mest lämpliga att skapa i den kontrollerande presentatören, där de kunde påverkas av gemensam logik. Motiveringen till detta var att denna specifika dialogruta inte hade dynamiskt innehåll, och kunde därför lämpligen ha starkare band mellan sina interna komponenter.

9.3 Dynamiskt innehåll

Något som länge förblev oklart var hur det skulle gå att åstadkomma dynamiska utbyten av undervyer under exekvering, samtidigt som Passive View tillämpades.

Detta var relevant i en dialog för att skapa nya indikatorer, som skulle byta ut innehållet i användargränssnittet baserat på användarens val (Se figur 12). Den kontroll som detta formulär erbjuder är valet av indikator, samt om indikatorn skall läggas till eller förkastas.



Figur 12. dialog för skapande av nya indikatorer. Det tomma området i mitten får dynamiska innehållet under exekvering.

Den dynamiska funktionaliteten kunde ha gjorts genom att skapa nödvändiga undervyer inuti det formulär som använder dem. Detta skulle innebära att varje ny undervy måste läggas in som en ändring av koden i huvudvyn.

Men genom att flytta skapandet av undervyerna utanför formuläret, kan nya alternativa undervyer läggas till senare utan att påverka formulärets implementering. Detta är mera fördelaktigt ur underhållssynvinkel, men ställer mer avancerade krav på implementeringen.

9.3.1 Gruppering av undervyer i triader

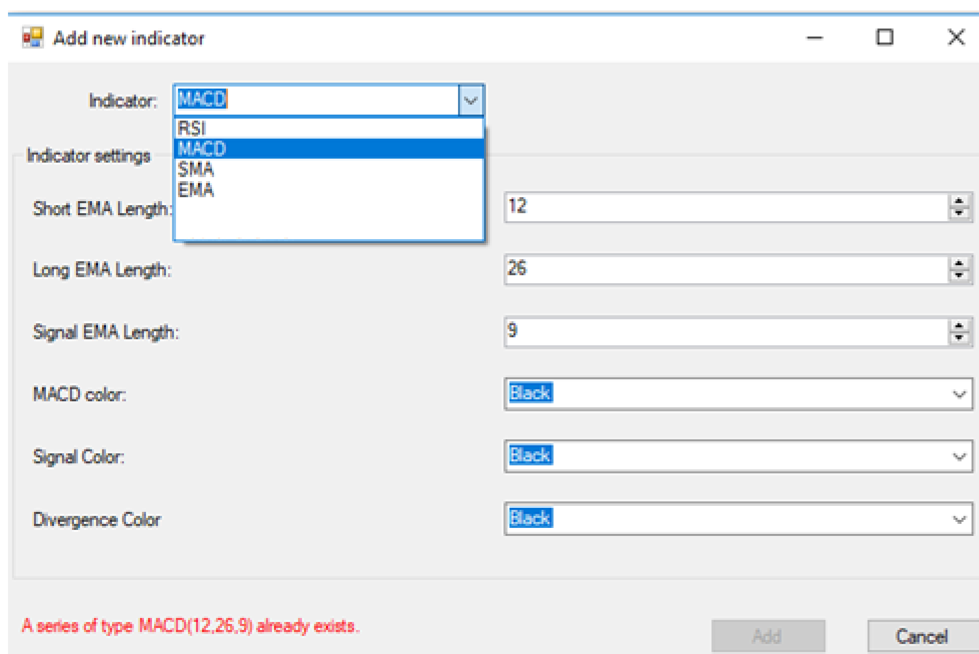
Alla undervyer hade i det nuvarande exemplet sin motsvarande presentatör och modell. För att slippa hålla manuell koll på vilken vy, presentatör och modell som hör ihop, så var det bättre att så snabbt som möjligt innesluta dem i triader.

Alla skapade triader kunde då injiceras i en lista till dialogens huvudsakliga presentatör. Den konkreta implementeringen av huvudvyn gavs ansvaret att konvertera alla undervyer, i detta fall UserControl-objekt, till samma teknologi.

9.3.2 Dynamiska utbyten av triader

I figur 13 visas hur användargränssnittet fylls när användaren väljer namnet på en indikator. Varje alternativ har sin egen MVP triad. Alternativen är enkla strängar som är specifika för triaden. När användaren gör ett val, kontrolleras denna sträng mot de triader som dialogen fått vid sin skapelse. Den korrekta undervyn visas på skärmen i det dynamiska området.

Den valda triadens presentatör är redan ihopkopplad med sin vy och modell, och fungerar fritt i bakgrunden. Alla underpresentatörer kunde på detta vis testas åtskilt från huvudpresentatören.



Figur 13. Val av indikator, exempel på dynamiskt innehåll.

9.4 Övriga erfarenheter

Passive View påverkade i synnerhet arbetsmängden när programmet senare utsattes för omfattande ändringar. Detta kom fram då en enkel textruta (TextBox) skulle bytas ut till en tabell med textrutor i en speciellt konfigurerad DataGridView, för att tillåta en mer radbaserad funktionalitet.

Detta innebar i huvudsak en hel del ändringar i vyn, och eftersom den nya tabellen kunde manipuleras i mycket större omfattning än en textruta, blev det också omfattande ändringar i vyns gränssnitt och den motsvarande presentatören.

Eftersom enhetstest hade gjorts runt presentatören så måste dessa också uppdateras. Detta innebar att mock-objekt gjorda på gränssnittet måste modifieras, vilket fördubblade arbetsmängden.

Utifrån detta kunde observeras, att radikala ändringar av funktionsprincipen hos användargränssnittet innebär en ökad mängd arbete om Passive View används. Detta kan användas som argument mot användningen av Passive View.

10 Resultat

Resultatet blev en ny struktur i hela applikationen, samt över 30 sidor av dokumentation som beskriver denna struktur. Dokumentationen kommer att underlätta arbetet för vem som än åläggs arbeta på detta program, oavsett om det är att arbeta på befintliga funktioner eller lägga till nya.

Den nya strukturen kan delas in två lager: Presentationslagret innehåller kod som är relaterad till användargränssnittet, medan affärslagret innehåller kod för beräkningar och hantering av data. De följande underkapitlen sammanfattar hur koden har påverkats av arbetet.

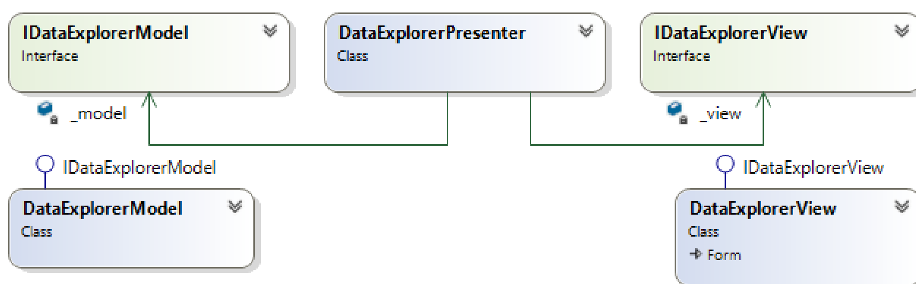
10.1 Presentationskod

Hela presentationslagret fick en struktur som baseras på Passive View mönstret, med undantag för designval orsakade av Windows Forms. Alla existerande formulär fick en egen MVP-triad. Under det fortsatta utvecklingsarbetet fick applikationen två nya formulär och nio avancerade dialoger som också konstruerades på samma sätt.

Alla presentatörer är ihopkopplade till motsvarande vy och modell med gränssnitt. Detta gör det möjligt att lägga till nya funktioner, testa presentationslogik, eller byta ut teknologin för användargränssnitt, med mycket mindre arbete än om allt skulle vara strukturerat som tidigare.

Runt presentatörerna har automatiserade enhetstest konstruerats för att kontrollera att presentationslogiken fungerar som förväntat. Då presentatörerna modifieras måste dessa handskrivna tester också modifieras, vilket kunde göras snabbare med ett testramverk.

I figur 14 visas ett exempel på den färdiga strukturen för DataExplorer funktionens triad. Notera att presentatören kommunicerar endast genom gränssnitt. DataExplorerView är den konkreta Windows Forms implementeringen av det grafiska användargränssnittet, som kan bytas ut till ett mock-objekt vid test av presentatören. Exempel på en mock-vy som används i test ses i bilaga 2.



Figur 14. Klassdiagram för MVP-triaden DataExplorer.

10.2 Affärskod

Till följd av omstruktureringen i presentationslagret blev det också ändringar i resten av applikationen. Ansvaret för start av applikationen flyttades över från det centrala formuläret,

MainWindowView, till den nya klassen TradingAssistantContext. På detta sätt blev MainWindowView mera passiv i enlighet med Passive View.

DataModel klassen blev en behållare för alla klasser som måste nås från flera olika formulär.

Det producerades en stor mängd nya klasser av mindre och lätthanterligare storlek, som brutits ut från existerande klasser. Allt detta gör det lättare att söka efter komponenter och att förstå deras funktion och syfte.

10.3 Dokumentation för kodbeskrivning

Förutom själva omstruktureringen producerades en kortfattad dokumentation som beskriver kodens struktur och komponenter. Denna dokumentation skrevs på engelska för att göra det läsbart för en större mängd personer, samt för att förhindra att programmeringstermer inte tappar sin betydelse i en översättning.

Dokumentationen innehåller bl.a. en beskrivning av Passive View mönstret så att läsaren får en inblick i vad som tillämpats utan att behöva söka information på internet. Den innehåller också anmärkningar på var detta program har avvikit från mönstret, och varför. På denna punkt är dokumentationens innehåll mycket lik detta examensarbete.

I dokumentationen beskrivs hur och var MVP-komponenternas instanser är skapade, samt vilka delar som används för att koppla ihop dem.

Beskrivningen av avancerade dialoger är en viktig del av dokumentationen. Att implementera sådana i Windows Forms med strukturen Passive View är en relativt komplicerad process. Det fanns begränsat med information på internet om avancerade dialoger med mönstret Passive View, så därför beskrevs denna process extra tydligt.

Sekvensdiagram och klassdiagram har ritats för att bättre visa hur alla delar av arkitekturen samspelar med varandra. Ett av klassdiagrammen ur dokumentationen presenteras i bilaga 1. Detta diagram visar hur dialogen för automationsinställningar är konstruerad. Där kan noteras att det finns två undertriader som hanterar inställningar på skilda inställningsobjekt.

Ett exempel på sekvensdiagram i dokumentationen presenteras i bilaga 3. Detta diagram används som exempel på hur applikationen skall hantera samspel mellan två olika och helt åtskilda formulär. Diagrammet visar hur ett klick på en aktie i formuläret för hantering av

aktier, genom händelser och funktionsanrop, åstadkommer en visning av den klickade aktiens historikdata i formuläret för grafritning.

10.4 Sammanfattning av resultat

Det kvarstår med säkerhet kodsegment som kunde förbättras, men källkoden som helhet är nu mycket mer objektorienterad, och drivs av händelser. Tack vare bruket av i huvudsak två refaktoriseringstekniker förbättrades kodens läsbarhet. En direkt märkbar skillnad är att alla komponenter i källkoden nu är organiserade i kategorier, vilket gör det mycket lättare att hitta en specifik komponent.

11 Diskussion

En omstrukturering av mjukvara är inte omedelbart vinstgivande. För en kund är det sällan viktigt hur källkoden ser ut, bara mjukvaran fungerar och levereras i tid. En omstrukturering på fungerande kod är därför svår att motivera, då de direkta kostnaderna är omfattande.

Fördelarna med att ha en lättförståelig struktur märks främst hos de som skriver koden. Efter omstruktureringen märktes att mindre tid behövdes för att hitta specifika bitar kod. Sådant bidrar direkt till minskade underhållskostnader.

När man har en del av en funktionalitet som man skall lägga till sitt system finns det två sätt att göra det på. Ett sätt är snabbt, men stökigt och kommer säkerligen att göra vidare ändringar svårare i framtiden. Det andra sättet resulterar i en renare design, men kommer att ta längre att implementera. Detta koncept kallar Martin Fowler ”Technical Debt” (Fowler, 2003). Tack vare detta arbete så har nu denna skuld minskat avsevärt och kommer i framtiden att betala sig tillbaka många gånger om i både tid och minskad arbetsbörda. Dessutom var det fördelaktigt att omstruktureringen gjordes då programmet ännu var i ett prototypstadie, och källkoden ännu var begriplig.

Gällande Passive View, har mönstret konstaterats passa endast för simpla implementeringar i Windows Forms. Om man enbart strävar efter maximal möjlighet till testning, eller lätta utbyten av komponenter i användargränssnittet, är Passive View ett bra alternativ.

Nackdelarna med Passive View är den ökade mängden kod som behövs för att koppla ihop komponenterna. Vissa funktioner i Windows Forms så som Data Binding kan inte utnyttjas inom ramarna för mönstret. Avancerade TreeView, DataGridView och Chart objekt blir svårare att implementera enligt Passive View utan att göra kompromisser. De flesta exempel som kan hittas på internet om Passive View och Windows Forms är mycket simpla, och påvisade aldrig dessa problem.

I detta examensarbete tillämpades mönstret Passive View då det i vissa fall skulle ha varit effektivare att använda mönstret Supervising Controller, som tillåter Data Binding. Detta gjordes med avsikten att hålla strukturen konsekvent. Resultatet blev då att mönstrets enda syfte, att lösa ett specifikt problem, blev ignorerat.

För övrigt kodades mock-objekt manuellt, vilket är en långsam process. Det skulle ha varit snabbare att använda något testramverk som kan konstruera mock-objekt från gränssnitt.

Källförteckning

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S., 1977. *A pattern language.* [Online] http://library.uniteddiversity.coop/Ecological_Building/A_Pattern_Language.pdf [hämtat: 28.5.2017]

Bower, A. & McGlashan, B., 2000. *Twisting the triad.* [Online] http://www.object-arts.com/downloads/papers/TwistingTheTriad.PDF [hämtat: 2.7.2017]

Cleary, S., 2013. *Task.Run vs BackgroundWorker: Intro.* [Online] https://blog.stephencleary.com/2013/05/taskrun-vs-backgroundworker-intro.html [hämtat: 2.7.2017]

Fowler, M., 2006. *Passive View.* [Online] https://www.martinfowler.com/eaaDev/PassiveScreen.html [hämtat: 28.5.2017]

Fowler, M., 2003. *Technical Debt.* [Online] https://martinfowler.com/bliki/TechnicalDebt.html [hämtat: 12.6.2017]

Fowler, M. *Refactoring* (u.å.). [Online] https://refactoring.com/ [hämtat: 25.6.2017]

Fowler, M., 2006. *CodeSmell.* [Online] https://martinfowler.com/bliki/CodeSmell.html [hämtat: 25.6.2017]

Fowler, M., 1999. *Extract Class.* [Online] https://refactoring.com/catalog/extractClass.html [hämtat: 25.6.2017]

Fowler, M., 2004. *RefactoringMalapropism* [Online] https://martinfowler.com/bliki/RefactoringMalapropism.html [hämtat: 6.10.2017]

Gabriel, R. P., 1996. *Patterns of software.* [Online] https://web.archive.org/web/20030801111358/http://dreamsongs.com/NewFiles/PatternsOfSoftware.pdf [hämtat: 28.5.2017]

Greer, D., 2007. *Interactive Application Architecture Patterns.* [Online] http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/ [hämtat: 24.7.2017]

Jenkov, J., 2015. *Dependency Injection.* [Online] http://tutorials.jenkov.com/dependency-injection/index.html [hämtat: 24.6.2017]

Microsoft. *Introduction to Windows Forms* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/aa983655(v=vs.71).aspx [hämtat: 31.5.2017]

Microsoft., 2017. *C#.* [Online] https://docs.microsoft.com/en-us/dotnet/csharp/csharp [hämtat: 18.6.2017]

Microsoft. *Events Tutorial* (u.å). [Online] https://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx [hämtat: 24.6.2017]

Microsoft. *Dialog Boxes* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/windows/desktop/ms632588(v=vs.85).aspx [hämtat: 17.8.2017]

Microsoft, 2017. *Properties* [Online] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties [hämtat: 17.8.2017]

Microsoft., *Model-View-Presenter* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/ff709839.aspx [hämtat: 17.8.2017]

Microsoft, 2017. *Overview of the .Net Framework.* [Online]https://docs.microsoft.com/en-us/dotnet/framework/get-started/overView [hämtat:18.6.2017]

Microsoft. *The Model-View-Presenter (MVP) Pattern* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/ff649571.aspx [hämtat: 14.8.2017]

Microsoft, 2017. *Interface*. [Online] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface [hämtat: 14.8.2017]

Microsoft. *Control Class* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/system.windows.forms.control(v=vs.110).aspx [hämtat: 15.7.2017]

Microsoft, 2017. *DataGridView Control* (Windows Forms). [Online] https://docs.microsoft.com/en-us/dotnet/framework/winforms/controls/datagridview-control-windows-forms [hämtat: 15.7.2017]

Microsoft. *UserControl Class* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/system.windows.forms.usercontrol(v=vs.110).aspx [hämtat: 24.7.2017]

Microsoft, 2017. *Windows Forms Data Binding*. [Online] https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-data-binding [hämtat: 8.10.2017]

Microsoft. *Common Dialogs* (u.å.). [Online] https://msdn.microsoft.com/en-us/library/windows/desktop/dn742498(v=vs.85).aspx [hämtat: 8.10.2017]

Miller, J., 2007. *Build your own CAB Part #4 – The Passive View*. [Online] http://codebetter.com/jeremymiller/2007/05/31/build-your-own-cab-part-4-the-passive-View/ [hämtat: 12.6.2017]

Potel, M., 1996. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java.* [Online] http://www.wildcrest.com/Potel/Portfolio/mvp.pdf [hämtat: 28.5.2017]

Software Testing Fundamentals. *Unit Testing* (u.å.). [Online] http://softwaretestingfundamentals.com/unit-testing/ [hämtat: 24.6.2017]

SourceMaking. *Large Class* (u.å.). [Online] https://sourcemaking.com/refactoring/smells/large-class [hämtat: 1.7.2017]

Svenska Akademins Ordlista, 2015. [Online] http://spraakdata.gu.se/saolhist/bildfiler/SAOL14referens.html [hämtat: 14.8.2017]

Telerik. *Unit Testing and Mocking Explained* (u.å.). [Online] http://www.telerik.com/products/mocking/unit-testing.aspx [hämtat: 25.6.2017]

Wilson, B., 2008. *Composite Views in Model-View-Presenter*. [Online] http://bradwilson.typepad.com/blog/2008/06/composite-Views.html [hämtat: 8.7.2017]

Exempel på dokumentationsinnehåll: Klassdiagram för dialog med en MVP-triad per TabPage.

Exempel på en mock-vy som ersätter en Windows Forms vy vid testning.

```
public class MockDataExplorerView: IDataExplorerView
{
    #region TestResult containers
    public StockViewData dataOfAddedSingleStock;
    public List<StockViewData> dataAddedToTreeview;

    public bool PopulateTreeviewWithDataCalled;
    public bool UpdateSingleStockNodeCalled;
    #endregion

    public event StringEventHandler NeedMetricsForStock;
    public event StringEventHandler StockNodeClicked;

    3 references
    public void PopulateWithStockData(List<StockViewData> datalist)
    {
        PopulateTreeviewWithDataCalled = true;
        dataAddedToTreeview = datalist;
    }

    5 references
    public void UpdateSingleStockNode(StockViewData data)
    {
        UpdateSingleStockNodeCalled = true;
        dataOfAddedSingleStock = data;
    }

    1 reference
    public void FireNeedMetricsForStock(string StockName)
    {
        NeedMetricsForStock?.Invoke(StockName);
    }
}
```

Sekvensdiagram från dokumentationen, som exempel på kommunikation mellan två separata formulär.