CHALMERS

A stock trading system in WPF

Master of Science Thesis

JENNY PETTERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, May 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

A stock trading system in WPF

Jenny Pettersson

© Jenny Pettersson, May 2009.

Examiner: Jan Skansholm

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2009

CHALMERS TEKNISKA HÖGSKOLA

Ett börsanalyssystem i WPF

Ett examensarbete i datalogi utfört vid Institutionen för
Datavetenskap vid Chalmers Tekniska Högskola

Utfört av:
Jenny Pettersson

Examinator:
Jan Skansholm

Institutionen för Datavetenskap
Chalmers Tekniska Högskola och Göteborgs Universitet
Göteborg 2008

Sammanfattning

Denna rapport beskriver arbetet med att designa och utveckla en plattform för simulering och utvärdering av olika tradingsystem.

Det viktigaste kravet som sattes upp i examenarbetets förstudie var att åstakomma en flexibel och utbyggbar grunddesign för att man vid senare tillfälle ska kunna uttöka systemet vartefter behov av mer funktionalitet uppstår i en snabbt föränderlig finansiell värld. Vidare skulle Microsofts allra senaste teknik användas.

Resultatet blev systemet Trader som är en modulbaserad applikation med tydliga riktlinjer för hur moduler ska laddas, skapas, initieras samt hur de kommunicerar med varandra utan överflödiga beroenden. Vägen dit gick via inlärning av diverse designmönster och designprinciper. Omfattande användning av händelser och kommandon för kommunikation mellan olika moduler samt programmerandet mot interface istället för konkreta objekt i alla lägen gav få beroenden och en flexibel och utbyggbar grunddesign.

Trader kan koppla upp sig mot börsen för att hämta kursdata. Denna data kan sedan användas för att utvärdera hur olika tradingsystem skulle ha presterat i olika tidsintervall. Datorer har inte samma slutledningsförmåga som människor. De behöver därför klara regler att verka efter i form av ett tradingsystem. Ett tradingsystem brukar bestå av entry, exit, filter samt en metod för moneymanagement. Det finns i en första officiell version av plattformen stöd för simulering av två olika tradingsystem, Trendsystem och Momentumsystem. Dessa valdes för att illustrera två viktiga begrepp inom den tekniska analysen, trend och momentum.

Systemet är utvecklat i Visual Studio 2008 och SQL Server 2008 Express samt utnyttjar Microsofts senaste teknik såsom exempelvis WPF och LINQ.

Abstract

This report describes the work with designing and developing a platform for simulating and evaluating different trading systems.

The most important requirement which was stated in the pre-study was to build a flexible and extendable base design so that it is possible to extend the system later on in an ever changing financial world. Furthermore Microsoft's latest technology should be used.

The result is Trader which is a module based application with distinct directions for how modules are loaded, created, initialized and how they communicate with each other without any redundant dependencies. This was accomplished by learning different design patterns and design principles. Extensive using of events and commands for communication in addition to programming against interfaces instead of concrete objects at all times gave very few dependencies and a flexible and extendible base design.

Trader can connect to the stock exchange to get historical stock data. This data can be used to evaluate how different trading systems would perform in different time intervals. A computer has not the same powers of deduction as a human being. It needs clear directions in the shape of a trading system to make trading decisions. A trading system consists of entry, exit, filter and a method for money management. In the first official version of Trader there are two trading system implemented, Trendsystem and Momentumsystem. Those were chosen to illustrate two important concepts within the technical analysis field, trend and momentum.

The system is developed in Visual Studio 2008 and SQL Server 2008 Express and uses Microsoft's latest technology, for example WPF and LINQ.

Förord

Denna rapport beskriver ett examensarbete i datalogi som har utförts under åren 2007-2008 vid Institutionen för Datavenskap på Chalmers Tekniska Högskola och Göteborgs Universitet.

Jag skulle vilja tacka min handledare och examinator Jan Skansholm för konstruktiv kritik och stöd under arbetets gång. Jag vill även tacka Avanza Bank för möjligheten och tillåtelsen att använda deras klient-API AvanzaFix.

1. Innehållsförteckning

2. INLEDNING - 7 -

2.1. BAKGRUND - 7 -
2.2. SYFTE - 7 -
2.2.1. PROJEKTETS SYFTE - 7 -
2.2.2. RAPPORTENS SYFTE - 8 -
2.3. PROJEKTBESKRIVNING - 8 -
2.4. BEGRÄNSNINGAR - 8 -
2.5. RAPPORTENS STRUKTUR - 8 -

3. FÖRSTUDIE - 9 -

3.1. AUTOMATISERAD BÖRSHANDEL - 9 -
3.2. TEKNISK ANALYS - 10 -
3.2.1. DIAGRAM - 10 -
3.2.2. STÖD OCH MOTSTÅND - 11 -
3.2.3. TRENDEN - 11 -
3.2.4. MOMENTUM - 13 -
3.3. AVANZA - 13 -
3.3.1. AVANZAFIX - 13 -
3.3.2. KLIENTEN - 14 -
3.4. KRAV - 15 -
3.5. SYSTEMFÖRSLAG - 15 -

4. TEORETISKT RAMVERK - 18 -

4.1. UTVECKLINGSMILJÖ - 18 -
4.1.1. MICROSOFT .NET - 18 -
4.1.2. VISUAL STUDIO - 19 -
4.1.3. C# - 20 -
4.1.4. LINQ - 21 -
4.1.5. WPF - 22 -
4.2. MICROSOFT ENTERPRISE LIBRARY - 26 -
4.2.1. LOGGING APPLICATION BLOCK - 27 -
4.2.2. EXCEPTION HANDLING APPLICATION BLOCK - 28 -
4.2.3. POLICY INJECTION APPLICATION BLOCK - 28 -
4.3. COMPOSITE APPLICATION GUIDANCE FOR WPF - 29 -
4.3.1. BOOTSTRAPPER - 29 -
4.3.2. CONTAINER - 30 -
4.3.3. SERVICES - 30 -
4.3.4. MODULE - 30 -
4.3.5. SHELL AND VIEW - 31 -
4.3.6. EVENT AGGREGATOR - 31 -
4.3.7. COMMANDS - 31 -
4.4. DESIGNMÖNSTER - 32 -
4.4.1. ADAPTER - 32 -
4.4.2. COMMAND - 33 -

4.4.3. SINGLETON - 33 -
4.4.4. STRATEGY - 34 -
4.4.5. TEMPLATE METHOD - 34 -
4.4.6. STATE - 35 -
4.4.7. DATAMODEL-VIEW-VIEWMODEL - 36 -
4.5. DESIGNPRINCIPER - 37 -

5. IMPLEMENTATION - 39 -

5.1. ÖVERGRIPANDE DESIGN - 39 -
5.2. TRADER - 41 -
5.2.1. SHELL - 42 -
5.3. TRADER.CUSTOMCONTROLS - 44 -
5.3.1. WINDOWCONTROLS - 44 -
5.4. TRADER.COMMON - 44 -
5.4.1. LOGGNING - 45 -
5.4.2. FELHANTERING - 46 -
5.5. TRADER.MODULES.DATAACCESS - 47 -
5.6. TRADER.MODULES.CONNECTION - 47 -
5.7. TRADER.MODULES.INSTRUMENT - 49 -
5.8. TRADER.MODULES.DATAMANIPULATION - 50 -
5.9. TRADER.MODULES.SIMULATION - 53 -
5.9.1. TRENDSIMULATION - 53 -
5.9.2. MOMENTUMSIMULATION - 55 -

6. RESULTAT - 57 -

6.1. DISKUSSION - 57 -
6.2. SLUTSATSER - 58 -
6.3. REKOMMENDATION FÖR FORTSATT ARBETE - 58 -

7. KÄLLOR - 60 -

2. Inledning

I inledningen av rapporten beskrivs bakgrund till projektet, syfte, uppdragsgivare, begränsningar samt rapportens struktur.

2.1. Bakgrund

Under de senaste åren har förutsättningarna för handel av finansiella instrument förändrats fundamentalt. Den globala avregleringen av de finansiella marknaderna och Internets framfart världen över har bidragit till att alla aktörer, såväl professionella som amatörer, nu har tillgång till samma information till mycket låga kostnader[1]. Borta är den tid då endast några få utvalda kunde följa aktiemarknaden.

Till grund för handel med finansiella instrument ligger ofta olika typer av analys. Aktievärdering handlar om att hitta aktier som betalas lägre på börsen än vad de är värda enligt den värderingsmodell man använder[2]. En aktör kan använda sig av fundamental analys där man gör en bedömning av hur mycket en aktie borde kosta givet vissa förutsättningar som exempelvis företagets vinst och vinsttillväxt. En annan analysmetod som genom åren varit mål för en hel del skeptism och misstänksamhet men som på senare år vunnit större acceptans är teknisk analys. Teknisk analys är en analysmetod som främst analyserar tillgång och efterfrågan på värdepapper baserat på historisk kursdata och handelsvolym. Metoden går i korta ordalag ut på att man i diagramform eller med hjälp av olika tekniska indikatorer studerar prishistoriken för den underliggande marknaden för att på så sätt försöka prognostisera en framtida kursutveckling.

Ytterligare en trend som kan skönjas är att börshandeln blir alltmer datoriserad. Med datorers hjälp kan man idag kontrollera hundratals aktier och derivat i realtid och ett automatiskt handelssystem kan utföra en mängd handelsstrategier simultant. Fördelarna man vinner gentemot manuell aktiehandel ligger i precision, snabbhet, frånvaron av psykologiska spärrar, outröttlighet och obegränsad kapacitet. I Europa uppskattas idag 20–30 procent av all handel ske med hjälp av datorer och smarta programmerade orderverktyg, och i USA är andelen så hög som bortåt 50 procent[3]. Automatisk börshandel kan definieras som handel med finansiella instrument som görs automatiskt av en dator efter ett antal regler som är formulerade utifrån en specifik tradingstrategi. Dessa regler har ofta sin grund i teknisk analys men kan även basera sig på andra mer funtamentala parametrar. Ett tradingsystem brukar bestå av fyra generella delar, entry, exit, filter och moneymanagement. Automatisk börshandel har hittills i praktiken endast varit möjlig för stora institutioner, direktmedlemmar på börserna samt aktörer med väldigt stora resurser. På senare tid har dock denna möjlighet öppnats även för mindre aktörer som genom vissa börsmedlemmar kan koppla upp sig mot börsen.

2.2. Syfte

Syftet kan delas upp i två delar, projektets syfte samt rapportens syfte.

2.2.1. Projektets syfte

Främsta syftet med detta examensarbete är att utveckla en plattform för simulering och utvärdering av olika tradingsystem. Systemet ska ge stöd för simulering av egna automatiska handelsstrategier baserat på historisk kursdata. Programmet bör även vara utbyggbart för att kunna koppla på automatisk orderläggning mot riktiga börsen om någon simulerad handelsstrategi skulle visa sig vara lovande.

[1] http://www.aktiespararna.se/lar-dig-mer/Fordjupningar/Finansyrken/Lar-dig-rollerna-pa-finansmarknaden/
[2] http://has.aktiespararna.se/alltomaktier/snabbkurs.asp?id=7&path=vrdering_av_aktier
[3] http://www.aktiespararna.se/artiklar/Reportage/Matematiska-formler-tar-over-borshandeln/

Mitt egenintresse är först och främst att lära mig mer om den tekniska gången vid utvecklandet av ett lite större program, dels utifrån givna krav, dels mot begränsningar i teknik och omfattning. Vidare känns det nyttigt att skaffa sig erfarenhet av programmering av Windows-applikationer i .NET-miljö då många företag idag använder sig av denna teknik.

2.2.2. Rapportens syfte

Syftet med denna rapport är att beskriva hur jag har lagt upp och genomfört projektet, från idéstadiet till färdig produkt. Jag belyser här även tekniska begränsningar samt problem som har uppkommit under arbetets gång.

2.3. Projektbeskrivning

Examensarbetets huvuddelar kan delas in enligt följande delmoment:

- Förberedelsearbete.
- Förstudie och specificering av programmet.
- Inlärning av nödvändiga tekniker som C#, Microsoft SQL Server, designprinciper, designmönster, LINQ, WPF med mera.
- Implementation.
- Dokumentation av projektet i en rapport.
- Presentation av examensarbetet.

2.4. Begränsningar

Som alltid när det handlar om programmeringsarbete kan det vara svårt att bedöma hur lång tid implementationen av olika programfunktioner kommer ta. Det är lätt att ta i för mycket när man ska starta upp ett nytt projekt och därför kan antalet planerade funktioner för systemet komma att behöva skäras ned under arbetets gång då tiden för examensarbetet är begränsad. Dessutom har detta projekt begränsade ekonomiska resurser vilket kan leda till att planerad funktionalitet inte kan implementeras om det kräver investering i alltför kostsam programvara.

2.5. Rapportens struktur

Rapporten börjar med denna inledning och fortsätter sedan med en förstudie som beskriver teknisk analys, nätmäklarfirman Avanza och automatisk aktiehandel samt en genomgång av krav och systemförslag. Därefter följer ett teoretiskt ramverk som behandlar de olika teorier och tekniker som använts för att skapa programmet. Efterföljande avsnitt handlar om implementeringen av systemets olika delar. Slutligen förs en diskussion runt projektets resultat och rekommendationer för fortsatt utveckling av programmet ges. Sist i rapporten återfinns källor.

3. Förstudie

Syftet med detta examensarbete är att utveckla ett program som ska kunna användas för att utvärdera olika tradingsystem. Förstudien inleds därför med en kort genomgång av vad automatiserad börshandel och tradingsystem innebär. Därefter beskrivs kort teknisk analys och nätmäklarfirman Avanza vars FixAPI som kommer användas för uppkoppling mot börsen. Sist i kapitlet specificeras krav och systemförslag.

3.1. Automatiserad börshandel

Automatisk börshandel även kallad programhandel eller algoritmisk handel kan definieras som handel med finansiella instrument som görs automatiskt av en dator efter ett antal regler som är formulerade utifrån ett specifikt tradingsystem. I Europa uppskattas idag 20–30 procent av all handel ske med hjälp av datorer och smarta programmerade orderverktyg, och i USA är andelen så hög som bortåt 50 procent[4].

Fördelarna man vinner gentemot manuell aktiehandel ligger i precision, snabbhet, frånvaron av psykologiska spärrar, outröttlighet och obegränsad kapacitet. Datorn kan dessutom följa och handla på ett stort antal marknader på samma gång. Nackdelen med automatisk handel är att människan har något som en maskin förmodligen aldrig kan tillägna sig i samma skala, nämligen känsla och intuition.

Automatisk börshandel har hittills i praktiken endast varit möjlig för stora institutioner, direktmedlemmar på börserna samt aktörer med väldigt stora resurser. På senare tid har dock denna möjlighet öppnats även för mindre aktörer som genom vissa börsmedlemmar kan koppla upp sig mot börsen.

Datorer har inte samma slutledningsförmåga som människor. De behöver klara regler att verka efter i form av ett tradingsystem. Ett tradingsystem brukar bestå av ett antal generella delar:

- *Entry* - Det moment som avgör när vi går in i marknaden. Ofta handlar detta om att vissa villkor ska vara uppfyllda, att en köp- eller säljsignal har genererats.

- *Exit* - När vi väl gått in i marknaden, måste vi någon gång också gå ur den. Antingen sker detta för att ta hem en vinst, stoppa en förlust eller helt enkelt för att inte låsa kapital om det inte längre finns någon potential i marknaden.

- *Filter* - Ett filter eller sållning innebär lösare och mer övergripande kritier innan en regeln för köp- eller säljsignalen kommer in i bilden. Vi kan se det i ett större perspektiv såsom val av marknader eller att vi bara tillåter handel i papper med en viss omsättning, volatilitet eller annat kriterie. Alternativt kan ett filter vara mer specifikt för den valda marknaden/aktien, som ett direkt försteg till köpsignalen, såsom att det ska vara en trendande marknad eller att priset rör sig inom ett visst inervall.

- *Moneymanagement* - För att inte riskera att vi förlorar allt vårt kapital när vi går in i en sämre period behöver vi en sund moneymanagement-metod. Oftast består denna av en formel som avgör hur många aktier vi bör köpa till en fördefinierad risk. Vanligt är att man riskerar en viss procent av sitt kapital och utifrån detta beräknar hur många aktier man ska köpa. Moneymanagement sker när villkoren för köpsignalen är uppfyllt, precis innan ordern placeras i marknaden.

[4] http://www.aktiespararna.se/artiklar/Reportage/Matematiska-formler-tar-over-borshandeln/

Framtagandet av ett tradingsystem börjar alltså med en idé som kan komma från att man uppmärksammat olika mönster på marknaden. Det kan handla om allt från att en aktie alltid tycks ha sin lågpunkt vid ett visst klockslag, att två aktier alltid tycks följa varandra till att man vill testa hur väl olika TA-modeller fungerar på olika instrument. I detta projekt kommer just detta sista scenario användas som exempel på tradingidé då teknisk analys-modeller är välkända och lätt att definiera i matematiska formler datorn kan förstå. En kort genomgång av teknisk analys återfinns i nästa avsnitt.

Vidare behövs tillgång till historisk kursinformation för olika värdepapper för att kunna utvärdera hur bra vårt tradingsystem hade presterat och bestämma lämplig nivå för entry, exit och filter. För en fungerande moneymanagement-funktion behövs dessutom depåinformation. Hur mycket pengar finns på kontot, hur stor del av dessa vill vi riskera i en ny entry osv. Både kursdata och depåinformation kommer hämtas från Avanza som beskrivs närmare senare i detta kapitel.

3.2. Teknisk analys

Som nämndes i inledningen av denna rapport ligger olika typer av analys ofta till grund för aktiehandel. Den traditionellt mest använda metoden kallas för fundamental analys där man gör en bedömning av hur mycket en aktie borde kosta, givet vissa förutsättningar som exempelvis företagets vinst och vinsttillväxt. Grunden för detta projekt ligger dock inte i den fundamentala analysen utan i teknisk analys, en annan analysmetod som fått större acceptans på senare tid.

Teknisk analys är en analysmetod där man endast tar hänsyn till aktiens tidigare kursrörelser och handelsmönster (Haskel & Wilke, 2001). Den tekniska analysen är också närbesläktad med området aktiemarknadspsykologi. Lyckas man förutsäga ett överdrivet flockbeteende på marknaden har man en hel del att vinna.

Studier av diagram och kursmönster är en central del av den tekniska analysen. Genom att identifiera kurstrender kan man försöka förutsäga åt vilket håll kursen är påväg. Man brukar tala om trendlinjer, stöd och motstånd. En aktie befinner sig i en typisk uppåtgående trend när varje topp och botten ligger högre än den föregående. En nedåtgående trend är motsatsen när varje ny botten hamnar lägre än den förra. En viktig uppgift inom den tekniska analysen blir alltså att identifiera vändpunkterna när trenden vänder. Dessa kan man förutspå genom att identifiera olika kursmönster.

3.2.1. Diagram

En så kallad bar chart är det vanligaste diagrammet i teknisk analys. Det är baserat på aktiekursens lägsta-, högsta- och slutnotering under aktuell period. Ibland räknas även öppningkursen med. Eventuell öppningskurs samt stängningskurs visas som en liten pigg till vänster respektive höger om stapeln vars lägsta/högsta punkt motsvarar kursens lägsta/högsta notering.

Figur 1: Exempel på Bar Chart diagram

3.2.2. Stöd och motstånd

I teknisk analys talar man ofta om test av stöd och motstånd. Ett motstånd är en viktig nivå där en aktie ofta vänder nedåt. Den har helt enkelt svårt att passera motståndsnivån där det ofta är hög omsättning då många säljare lagt sig på denna nivå. Ju fler test ett motstånd utsätts för desto större är chansen att motståndet slutligen bryts. Ett stöd är på motsvarande sätt en nivå där aktien ofta vänder uppåt.

Figur 2: Diagram som illustrerar stöd och motstånd vid 125 respektive 145

3.2.3. Trenden

En uppåtgående trend kan identifieras genom att man med hjälp av en linje sammanbinder en akties bottnar i en uppgång. Linjen fungerar som ett golv och så länge detta är intakt är trenden fortsatt uppåtriktad. Om aktien svänger kraftigt inom trenden kan du även sammanbinda topparna. Då kan du se topparna inom trendkanalen lättare och slipper följa med ner till golvet varje gång aktien vänder nedåt.

En nedåtgående trend finns när du istället kan binda samman topparna till en nedåtlutande linje. Linjen fungerar som ett tak, och så länge taket inte bryts är trenden fortsatt nedåtriktad.

Figur 3: Exempel på en nedåtgående trendkanal

Hur kan man då mer konkret bestämma om en aktie trendar eller inte. Ett exempel på en teknisk indikator för att identifiera trenden är glidande medelvärde som kan beräknas för olika perioder. Nedanstående enkla formel används vid beräkningen av glidande medelvärde:

$$Enkelt\ glidande\ medelvärde\ (SMA) = \frac{Summa\ aktiekursnoteringar\ under\ x\ antal\ dagar}{x\ antal\ dagar}$$

Ett glidande medelvärde som har beräknats på många dagar är alltså mindre känsligt för varje ny dags aktiekursförändring jämfört med ett annat glidande medelvärde baserat på färre antal dagar. Glidande medelvärde beräknas varje period och då läggs denna dagens kursnotering till medan den äldsta kursnoteringen utgår från måttet.

Ibland kan hopp eller hack i kurvan uppstå vid beräkningen av ett enkelt glidande medelvärde. Många förespråkar därför användning av ett exponentionellt medelvärde framför enkelt glidande medelvärde. Formeln för ett 13-perioder exponentionellt glidande medelvärde är:

$$13\ EMA = senaste\ periodens\ pris\ x\ \frac{2}{13+1} +\ föregående\ periods\ EMA\ x\ (1 - \frac{2}{13+1})$$

Den tekniska analysen delar in trender i olika längder (Lagerholm & Nilsson, 2006):

- *Primär trend* mäter trenden i ett längre tidsperspektiv, från flera månader till flera år. Till exempel så används 200-dagars glidande medelvärde för att identifiera den primära trenden.

- *Sekundära trenden* har ett perspektiv från några veckor till några månader. Ofta används 50-dagars glidande medelvärde här.

- *Tertiära trenden* är ännu kortare, dagar till veckor. 20-dagars medelvärde brukar användas.

- *Subtertiära trenden* sträcker sig endast några timmar till dagar och är mest intressant för daytraders som handlar värdepapper med väldigt kort tidsperspektiv.

En annan indikator som hamnar under kategorin trendindikatorer är ADX-indikatorn som är en del i systemet ”The directional system” framtaget av W.Wilder. ADX-indikatorn används för att beräkna trendstyrkan och kräver en par definitioner:

Range = avståndet mellan högsta och lägsta kursen för en period

True range = det största av range, avståndet mellan periodens högsta och föregående periods stängningskurs eller avståndet mellan periodens lägsta och föregånde periods stängningskurs.

Directional movement = den del av en periods range som ligger utanför föregånde periods range.

Beräkningen sker enligt följande algoritm:

1. *Identifiera directional movement, DM+ och DM-.*
2. *Identifiera true range vilket alltid är ett positivt tal.*
3. *Beräkna +DI och –DI som utrycker DM i relation till true range.*
4. *Utjämna -DI och +DI med glidande medelvärden.*
5. *Beräkna ADX-indikatorn vilken visar skillnaden mellan utjämnat -DI och +DI. Först beräknas Directional Indicator (DI) och sedan definieras ADX som 13-dagars exponentionellt glidande medelvärde på detta värde.*

När ADX har ett högt värde så innebär det att den underliggande trenden är stark. Detta är logiskt då ett stigande ADX innebär att skillnaden mellan utjämnat -DI och utjämnat +DI ökar. Ett lågt ADX-värde identifierar konsolideringar. Det bör påpekas att ett högt eller stigande ADX innebär att trenden är stark, inte att den är uppåtriktad.

3.2.4. Momentum

Olika momentumindikatorer kan användas för att mäta hastigheten i en akties prisutveckling. Ordet momentum används ofta synonymt med oscillator som är något som mer eller mindre oregelbundet pendlar kring ett jämviktsvärde. På aktiemarknader kan man ofta skåda ett överreaktionsfenomen i samband med att olika händelser inträffar eller nyheter presenteras. En förklaring till överskattningsfenomenet kan vara om aktören vid ett givet tillfälle saknar en säker uppfattning om vad ny information innebär för en aktie är det för denna aktör rationellt att sälja innehavet. Att behålla instrumentet kan innebära en alltför stor risk man inte är villig att ta. Överreaktioner i aktiers kursutveckling kan vi försöka utnyttja med hjälp av momentumindikatorer vars primära syfte är att avgöra hur långt ifrån en jämviktsnivå en aktie befinner sig. Momentumindikatoren kan då signalera om en aktie är överköpt (ligger långt över jämviktsläget) eller översålд (ligger långt under jämviktsläget).

Momentumindikatorerna är en stor grupp indikatorer. En medlem av denna grupp är Relative Strength Index (RSI) som sägs vara den flitigast använda tekniska indikatorn i mäklarrummen (Torssell & Nilsson, 2000). Formeln för denna indikator lyder:

$$X - perioders\ RSI = 100 - \frac{100}{(1 + \left(\frac{U}{N}\right))}$$

U = medelvärdet baserat på senaste betalkursen under perioder som aktien stängt upp de senaste x perioderna.

N = medelvärdet baserat på senaste betalkursen under perioder som aktien stängt ned de senaste x perioderna. Perioder där aktien stänger oförändrat lämnas därhän.

RSI är alltså skalad mellan 0 och 100. Ett vanligt mått för perioden är 14 dagar. Om ett 14-dagars RSI har ett värde på 100 innebär det att aktien har stängt upp 14 dagar i följd vilket tolkas som extremt överköpt. Ett värde på 0 innebär tvärtom att aktien stängt ned 14 dagar i följd, följdaktligen extremt översålд. Även denna indikator kan användas för att generera köp- respektive säljsignaler. Ofta sätts nivåerna 30 respektive 70 för att indikera översålda respektive överköpta nivåer.

3.3. Avanza

De finansiella marknaderna har under de senaste åren genomgått stora förändringar. Framförallt Internets framfart har bidragit till helt nya möjligheter även för småsparare och privatpersoner att följa och handla på börserna världen över. Det har gått drygt tio år sedan de första specialiserade fristående nätmäklarna såg dagens ljus[5]. De drog in som en uppfriskande stormvind på privatplacerarnas aktiemarknad och snart kunde varje svenskt hems kök i princip bli ett modernt handlarrum för aktier. Ett tag var nätmäklarna uppe i runt 13 stycken. Sedan kom år 2000 och den följande börsbaksmällan som ledde till att en efter en föll bort.

Avanza Bank klarade sig dock väl och är idag Sveriges största nätbank med mer än 200000 sparare och flest avslut på Stockholmsbörsen. Avanzas affärsidé och viktigaste uppgift är att hjälpa svenska privatsparare att göra bättre affärer. Avanza erbjuder ett heltäckande utbud av spartjänster med aktie-, fond- och pensionssparande. Sedan några år tillbaka finns även möjligheten för mindre aktörer att koppla upp sig mot börsen genom att använda AvanzaFix som beskrivs nedan.

3.3.1. AvanzaFix

AvanzaFix är Avanzas egna implementation av fixprotokollet. Fix är ett effektivt, systemoberoende protokoll som är avsett för att skicka finansiell information. Det används av finansiära aktörer som

[5] http://www.aktiespararna.se/artiklar/Reportage/Med-natet-som-arena/

banker och börser samt står för Financial Information eXchange[6]. Det är helt textbaserat och innehåller inga speciella datatyper. Avanza har strävat efter att följa fixspecifikationen till 100% dock med tillägg för egna meddelanden.

För att underlätta kommunikationen över Fix så skapar AvanzaFix objekt som representerar de olika meddelandena. Dessa objekt översätts sedan till de objekt som används i Avanzas övriga system. Detta leder till att Fix abstraheras bort från den kodning man gör, oavsett om man befinner sig på klient- eller serversidan. För att underlätta för externa parter att kommunicera med Avanza via AvanzaFix så tillhandahåller Avanza en klient som har den funktionalitet och de klasser som krävs för att göra detta. Fixklienten är skriven i Java. Genom att använda J# i Microsoft .NET finns det även stöd för denna plattform sånär som på några stödklasser från C#. Klienten använder samma implementation av AvanzaFix som den server som används på Avanza vilket borgar för säker kompabilitet.

AvanzaFix kommunicerar över TCP/IP med rena textmeddelanden vilket gör det lätt att läsa av meddelanden och komma över lösenord. AvanzaFix använder sig av både krypterade och okrypterade kanaler för informationsflödet. Den krypterade kanalen är avsedd för att användas till känslig information såsom exempelvis lösenord och depåinformation medan den okrypterade kanalen är avsedd för instrument, kursinformation och annan icke känslig data. Vilka portar som används för respektive kanal är automatiskt konfigurerat av klienten.

3.3.2. Klienten

Klientens uppbyggnad är sådan att man har abstraherat bort AvanzaFix, trots att det är grunden i klienten. AvanzaFix används för att skicka och ta emot fixmeddelanden men då dessa konverteras till rena objekt så är det inget man använder i klienten som sådan.

Grundpaketet för en klient är `se.avanza.atci`. I detta paket hittar man bl.a. `se.avanza.atci.AbstractClient`, `se.avanza.atci.Client` samt `se.avanza.atci.SecureClient`. AbstractClient är basklassen för både Client och SecureClient där Client är avsedd för en okrypterad uppkoppling och SecureClient för en krypterad. Vid skapandet av en klient anger man den maskin som man ska gå mot.

```
SecureClient client = new SecureClient("serveradress");
```

I detta läge har man bara skapat objektet, för att koppla upp sig mot servern måste man logga in. Inloggning kan ske på olika sätt men de två vanligaste sätten är antingen genom att ange en depå eller genom att använda ett token.

```
client.login("account","password","D");
```

Den sista parametern i login styr vilken tjänst man vill använda för att logga in. I detta fall avser man att logga in med Depå som tjänst. Med klienten har man nu skapat en uppkoppling mot fixservern.

När man vill ha data som innehåller information som inte är så känslig, exempelvis instrument, kurser med mera, så är det tänkt att man skall använda en osäker uppkoppling. Detta görs genom att använda en Client istället för en SecureClient.

```
Client client = new Client("serveradress");
client.login("account","password","D");
```

[6] http://www.fixprotocol.org/

Detta är dock inte tillrådigt eftersom lösenordet då skickas i klartext. För att lösa detta problem finns det token inloggning. Detta innebär att man kan generera en token på serversidan som används för en engångsinloggning. Utöver engångsinloggning finns det även möjligheter att erhålla långtidstokens, dvs. tokens som har flera dagars gilighetstid.

För att be om information från systemet använder man sig av något som kallas Peer. En peer är en accesspunkt mot fixservern hos Avanza. Genom att använda sig av peers minskar man komplexiteten och lägger ut funktionaliteten till vad man skulle kunna kalla för plugins. En peer ansluts till en klienten genom att anropa addPeer på klienten.

```
SecureClient client = new SecureClient("serveradress");
se.avanza.atci.trading.AccountPeer accountPeer = new
se.avanza.atci.trading.AccountPeer();
client.addPeer(accountPeer);
client.login("account","password","D");
```

Den adderade peeren kan sedan nås genom klienten med metoden getPeer. Till en peer kan man sedan addera lyssnare som kommer att notifieras då servern skickar uppdateringar. I exemplet med AccountPeer så kan det röra sig om uppdatering av en depå och dess värden. Lyssnaren som läggs till är en klass som implementerar motsvarande interface, i fallet AccountPeer lägger man exempelvis till en klass som implementerar interfacet AccountListener. Exempel på andra peers som fungerar på motsvarande sätt är MarketPeer, MarketHistoryPeer, NewsPeer med flera.

3.4. Krav

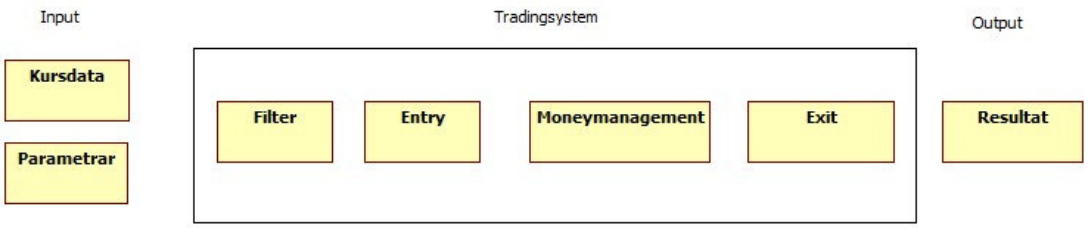
Kraven på detta system har ställts upp utifrån detta examensarbetes två primära syften.

Det första syftet bestod i att utveckla en plattform för simulering och utvärdering av olika tradingsystem. Systemet ska ge stöd för körning av egna automatiska handelsstrategier baserat på historisk kursdata. Dessvärre är tiden för exjobbet begränsad vilket innebär att långt ifrån alla önskade funktioner kommer att hinnas med att implementeras. Det viktigaste kravet blir alltså utbyggbarhet för att man vid senare tillfälle ska kunna uttöka systemet vartefter behov av mer funktionalitet uppstår. De finansiella marknaderna ändrar sig snabbt och nya funktioner kommer krävas hela tiden. Då dessa nya funktioner ska läggas till vill man helst att det ska innebära så lite ändring som möjligt av befintlig källkod. Ändras koden mycket ökar risken för fel i programmet och mer testning krävs. A och o vid utvecklandet är en bra design från början. Enligt (Gamma, Helm, Johnasson, & Vlissides, 1994) så måste man fundera på hur systemet kan komma att utvecklas under sin livstid för att få ett system som faktiskt kan förändras. En design som inte tar hänsyn till förändringar riskerar att få designas om vid stora förändringar vilket kostar mycket. Genom att följa beprövade designmönster och designprinciper kan man undvika detta eftersom dessa gör det möjligt för system att ändras. För att bli en god systemdesigner krävs erfarenhet, mycket erfarenhet. Att lära sig och försöka följa designmönster och designprinciper är förmodligen det mest effektiva sättet att stärka sina designkunskaper. Ett av kraven blir följdakligen att lära sig ett antal designprinciper och designmönster, dokumentera dessa i teoridelen och sedan tillämpa dem vid implementationen.

Det andra syftet var att lära sig mer om den tekniska gången vid utvecklandet av ett lite större program samt skaffa sig erfarenhet av programmering av Windows-applikationer i .NET-miljö. Ett krav som ställs med detta åtanke är att den allra senaste tekniken från Microsoft ska studeras och användas.

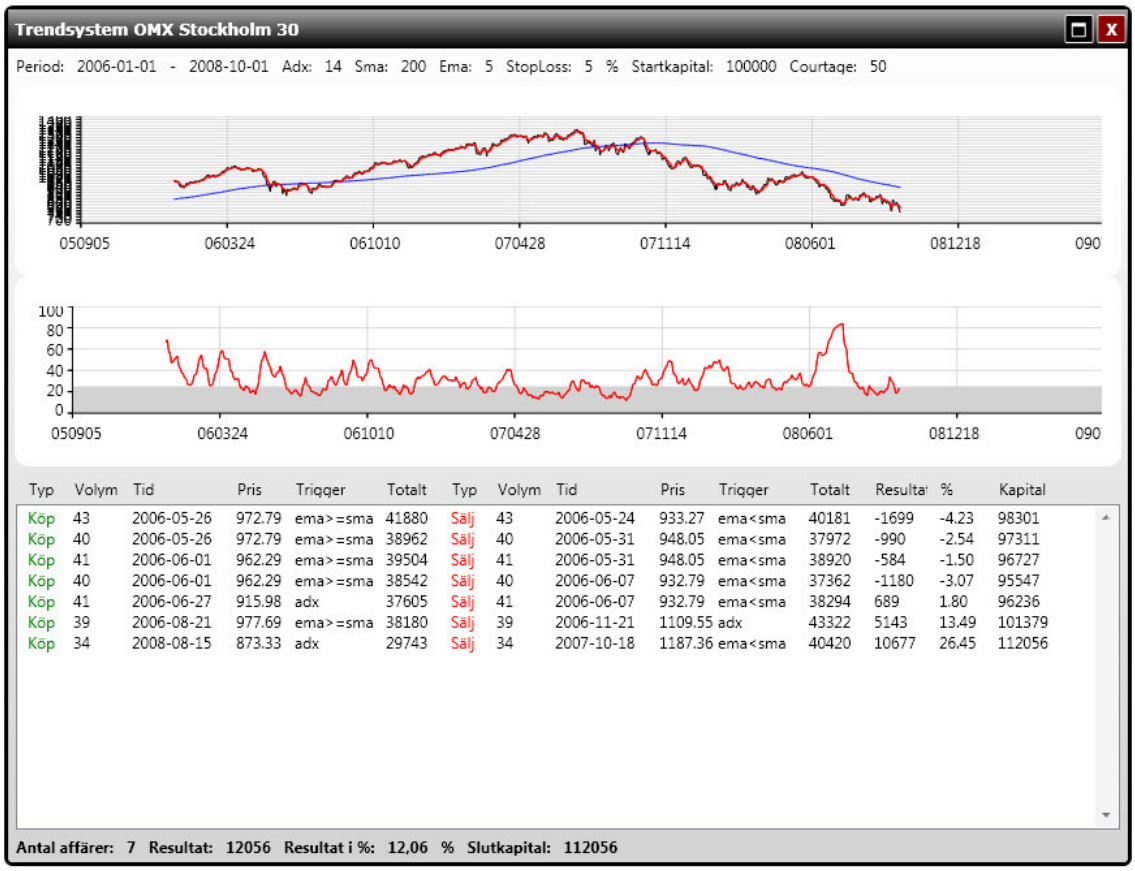
3.5. Systemförslag

Med förstudiens innehåll som grund har följande förslag på system tagits fram. Förväntningarna på vad det färdiga systemet ska kunna göra kan sammanfattas i nedanstående figur.



Figur 4: Schematisk illustration av färdigt system

Programmet ska kunna koppla upp sig mot Avanza via deras FixAPI för att hämta kursdata. Hämtad data ska sparas i en lokal databas för att göra senare simulering/testning av olika tradingsystem effektivare. Dessutom är det praktiskt att ha möjlighet att jobba med programmet i offlineläge vid tillfällen då man inte har tillgång till Internet. Minst två olika tradingsystem kommer implementeras och utvärderas inom ramen för detta examensarbete. Dessa tradingsystems entry kommer bygga på modeller som härstammar från den tekniska analysen. Målet är i detta skede inte att finna ”det” fantastiska tradingsystemet som alltid genererar vinst utan att bygga en plattform för utvärdering av tradingsystem. Detta mål verifieras tydligast med hjälp av några få, enkla modeller som är lätt att förstå. Entry i kombination med eventuellt filter genererar köp- och säljsignaler, moneymanagement bestämmer positionstorlek och exit när positionen ska avslutas. Resultatet presenteras sedan på ett överskådligt sätt med diagram och lista på vilka affärer som gjorts samt resultat. För att ge läsaren en bild att ta med sig genom resten av rapporten av vad som ska uppnås visas ett resultatfönster för en simulering av trendsystemet nedan.



Figur 5: Resultatruta i färdigt system

Programmet ska utvecklas i Visual studio 2008 med C# 3.0 som programmeringsspråk. Kursdata, instrument och annat sparas i en lokal SQL Server databas. Programmet kommer basera sig på ramverket .NET 3.5 och använda de senaste tekniknologierna som som introducerats där. Exempelvis kommer användargränsnitt byggas i WPF och dataåtkomst skötas med hjälp av LINQ. Som grund för programmmet kommer Microsofts helt nya ramverk Composite Application Guidance for WPF användas.

En utförligare beskrivning av systemet och vilka designval som tagits återfinns i kapitel 4 som handlar om implementeringen av programmets olika delar.

4. Teoretiskt ramverk

4.1. Utvecklingsmiljö

Utvecklingen av detta program har skett helt i Microsoftmiljö. Nedan introduceras först Microsofts ramverk .NET i stort innan utvecklingsverktyg samt det valda programmeringsspråket C# kort beskrivs. Kapitlet avslutas med en genomgång av LINQ respektive WPF. Detta är ämnen man kan skriva obegränsade mängder fakta om. Jag har i denna rapport begränsat mig till korta avsnitt om just de delar som kommer användas flitigt under projektets implementationsfas.

4.1.1. Microsoft .NET

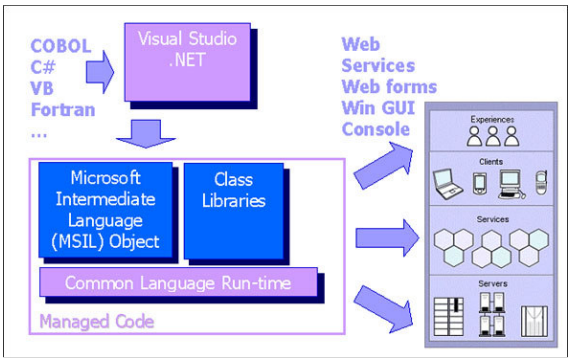
Microsoft har under de senaste åren utvecklat .NET som är en plattform med målet att förenkla kommunikation mellan olika program oavsett enhet, programmeringsspråk och operativsystem. Med hjälp av .NET-plattformen kan man skapa och köra en mängd olika applikationer som exempelvis konsolapplikationer, Windowsapplikationer med grafiska gränssnitt, serverbaserade webbapplikationer och webbtjänser, applikationer för mobila enheter med mera.

.NET Framework utgör kärnan i Microsofts .NET-plattform och finns att ladda ned gratis från Microsofts hemsida. .NET Framework är den miljö där .NET-baserade applikationer byggs, installeras och körs. Miljön tillhandahåller all nödvändig kompilering och exekvering av maskinkod vid körning av en .NET-applikation. Ramverket består av tre huvudsakliga delar:

- **Common Language Runtime (CLR):** Är den centrala delen av .NET som kompilerar ett program till plattformspecifik processorkod och exekverar det. CLR innehåller en mängd funktionalitet som ser till att applikationerna körs på ett säkert och korrekt sätt. CLR sköter exempelvis minneshantering, tråd- och kodexekvering, säkerhetsverifiering av kod, kompilering och en del andra systemsysslor. I .NET Framework ingår en språkspecifik kompilator till alla .NET-språk och kod som tas fram av denna kallas managed. Det är endast managed kod som kan dra nytta av de fördelar CLR innebär och för att kunna göra detta lagrar kompilatorn metadata tillsammans med de kompilerade koden som benämns MSIL (Microsoft Intermediate Language). Metadata utgörs av information som beskriver datatyper, klassattribut, referenser och används av runtimemiljön för att lokalisera och ladda objekt från minnet, sköta funktionsanrop och så vidare. Tack vare metadatan kan man integrera objekt skrivna i olika språk utan att typkonflikter uppstår och den erbjuder dessutom automatisk referensräkning och sophantering. CLR består av tre delar:
    1. **Common Type System** som är ett typsystem som stödjer de flesta typer och operationer som finns i moderna programmeringsspråk.
    2. **Metadatasystem** som binder metadata till typer vid kompileringstillfället. Exekveringsmaskinen utnyttjar sedan dessa vid exekvering.
    3. **Runtime-miljön** som exempelvis består av en exekveringsmaskin som exekverar .NET-baserade program genom att utnyttja information som finns lagrad i metadatasystemet.

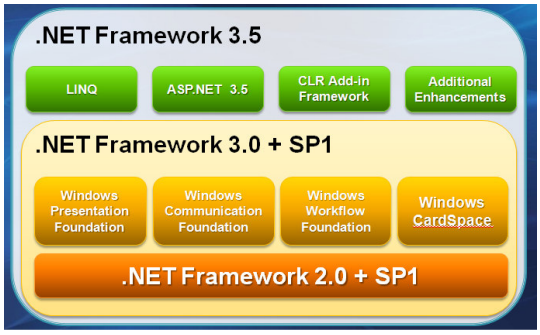
- **.NET Framework Class library** är ett klassbibliotek med tusentals återanvändbara klasser, gränssnitt och typer för att underlätta byggandet av .NET-applikationer. Alla objekt i biblioteket är skapade utifrån Common Language Specification för att de ska kunna användas av alla språk som följer dessa riktlinjer. All funktionalitet i .NET och klassbibliotekets alla objekt är skrivna i C#. Exempel på innehåll i klassbiblioteket är klasser för språkspecifika typer, nätverksoperationer, åtkomstoperationer för data/filer, operativsystemsysslor med mera. Klassbiblioteket innehåller även ett lager med programmerbara modeller.

- **Programmerbara modeller** underlättar byggandet av .NET-applikationer och består av exempelvis ASP.NET, ADO.NET, Web Forms, Windows Forms och XML Web Services.



Figur 6: .NET Framework[7]

En viktig aspekt med .NET är möjligheten att länka samman olika programmeringsspråk. Alla språk som följer Microsofts specifikation Common Language Specifikation är garanterat kompatibla med varandra och .NET. De vanligaste språken som brukar användas och ingår som standard i .NET-miljöer är C#, Visual Basic, C++ och J#. Alla språk i .NET är objektorienterade och de flesta av klasserna i .NET Frameworks klassbibliotek är samma oberoende av vilket språk man använder. Utöver detta finns det i klassbiblioteket språkspecifika klasser för Visual Basic, C#, C++ och J#, som bygger på de gemensamma klasserna.



Figur 7: .NET Framework 3.5[8]

.NET Framework har som ses i figur 2 släppts i olika versioner varav 3.0 och 3.5 är de två senaste. .NET Framework 3.5 innehåller sedan innovationer som C# 3.0, VB 9 och LINQ. Samtidigt med denna version släpptes även Visual Studio 2008 som beskrivs nedan.

4.1.2. Visual Studio

Visual Studio är Microsofts generation av utvecklingsverktyg som lanserades i en första version år 1997[9]. Från början bestod Visual Studio av flera olika utvecklingsmiljöer. Tanken på en gemensam plattform som kan användas oavsett vilket programmeringsspåk man föredrar realiserades i början av 2000-talet i och med Visual Studio .NET 2002 med hjälp av de första versionerna av .NET Framework. Plötsligt kunde utvecklare skriva applikationer i valfritt språk som ramverket hanterade och ändå dra full nytta av utvecklingsverktygets stöd för att till exempel kunna dra och släppa kontroller, Intellisense samt den nya gemensamma visuella designytan. När Visual Studio 2005 och Visual Studio Team System lanserades så innebar det ett jättekliv framåt vad gäller stöd för

[7] http://www.adtools.com/info/images/vis-studio.jpg
[8] http://guy.dotnet-expertise.com/content/binary/DotNetFramework35.png
[9] http://informator.se/nya_visual_studio_2008_och_.aspx

samarbete mellan projektmedlemmar och möjligheterna att strukturerat och effektivt höja kvaliteten och minska komplexiteten i utvecklingsprojekten.

Aktuell version är Visual Studio 2008 som lanserades i slutet av år 2007. Nyheter här är exempelvis LINQ, den nya programmeringsmodellen för dataåtkomst som integrerar ett nytt SQL-liknande frågespråk direkt i C# och VB.NET samt den första skarpa versionen av Visual Studio Designer for WPF. WPF-designstödet ger möjlighet att editera XAML-kod i Visual Studio och direkt se ändringarna i designvyn. Mer information om LINQ och WPF finns nedan.

4.1.3. C#

C# är ett programspråk som har växt mycket i popularitet under de senaste åren. Det är utvecklat av Microsoft som en del av .NET-plattformen. C# är ett objektorienterat språk som till stor del är baserat på C och C++ men även har en hel del gemensamt med java. Fördelar med C# är att det är relativt enkelt att lära sig och har en syntax som till stor del liknar språk som C, C++ och java. .NET innehåller dessutom ett stort kodbibliotek som förenklar utformningen av mer komplexa system. C# har flera användningsområden, det kan både användas som kompilerat språk på en lokal dator och som språk i ASP.NET. Detta gör det enkelt att länka samman program på en klientdator med serverdatorers program. C# erbjuder enkel integration med andra Microsoft-baserade programvaror och C#-kompilatorn kan användas utan licenser och speciella utvecklingsverktyg. Till nackdelarna brukar nämnas att C# liksom java körs i virtuell maskin vilket bland annat innebär att det kompileras just innan det körs vilket kan leda till fördröjd uppstartsfas.

I och med lanseringen av C# 3.0 under 2007 introducerades flera nyheter för att exempelvis göra LINQ möjligt. Några exempel ses nedan:

- *Objektinitialiserare*. Förenklar initialisering och konstruktion av objekt. Objektinitialiserare låter oss specificera värden för ett eller flera fält eller properties av objekt i ett och samma uttryck. Dom tillåter deklarativ initialisering för alla typer av objekt i stil med:

  ```
  var data = new ProcessData { Id = 23, Name = "MyProcess", Memory = 123456 };
  ```

- *Implicit typade lokala variabler*. En lokal variabels typ bestäms av uttrycken som används för att initialisera den. Vi tillåts alltså skapa en ny lokal variabel utan att ange typ enligt:

  ```
  var i = 5;
  ```

  Kompilatorn bestämmer typen automatiskt så trots att vi har använt en förenklad syntax så kan vi utnyttja fördelarna med starkt typade typer såsom IntelliSense och validering vid kompilering.

- *Anonyma typer*. Används frekvent inom LINQ och ser ut och nås på följande vis:

  ```
  var anon = new { FirstName = "Granville", LastName = "Barnett", Age = 20 };
  anon.Firstname = "John";
  anon.LastName = "Smith";
  anon.Age = 30;
  ```

  Detta är en anonym typ med 3 properties som alla är publika så man kan nå dem från andra delar av applikationen.

- *Lambda-uttryck*. I C# 2.0 introducerades anonyma metoder som tillåter körandet av kod där man tidigare hade förväntat sig en delegat. Lambda-uttryck är en nyhet i C# 3.0 som tar

ytterligare ett steg. Ett lambda-uttryck är en metod utan namn som returnerar värdet av ett uttryck. Ett lambda-uttryck skrivs på följande sätt:

(lista med parametrar) => funktion alternativt ett uttryck

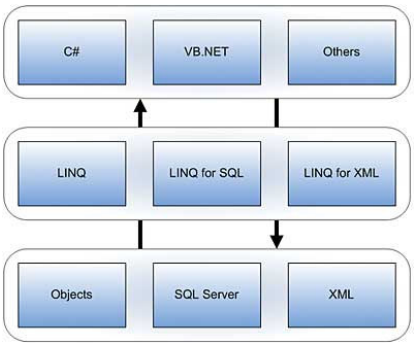
```
List<string> names = new List<string> { "John", "Jim", "Michelle" };
IEnumerable<string> filter = names.Where(p => p.StartsWith("J"));
```

- I C# 3.0 kan man bygga ut klasser utan att ärva ifrån dem genom *extension methods*. Dom kan dock inte användas för att ändra befintliga metoder då instansmetoden anropas med samma signatur. Man kan skapa en egen metod genom att använda nyckelordet this innan argumentet enligt:

```
namespace StringExtensions {

    public static class StringExtensionsClass {

        public static string RemoveNonAlphaNumeric(this string s) {
            MatchCollection col = Regex.Matches(s,"[A-Za-z0-9]");
            StringBuilder sb = new StringBuilder();
            foreach (Match m in col)
                sb.Append(m.Value);
            return sb.ToString();
        }
    }
}

using StringExtensions;
string phone = "123-123-1234";
string newPhone = phone.RemoveNonAlphaNumeric();
```

4.1.4. LINQ

LINQ står för Language INtegrated Query och är en utbyggnad av .NET-ramverket som gör det möjligt att använda SQL-liknande frågor direkt i vanlig kod för att hämta data från olika datakällor såsom disk, databas, XML, objekt av klasser som implementerar IEnumerable/IEnumerator som exempelvis alla Collections (Marguerie, Eichert, & Wooley, 2008). LINQ är inget nytt språk utan är integrerat i C# och VB.NET. LINQ enar dataåtkomst och ger möjlighet att blanda data från olika datakällor. Man slipper blanda in olika språk som SQL, XML eller Xpath i applikationer som är skrivna i generella språk som C#. Fördelar med LINQ är att utvecklare enkelt kan skapa kopplingar mellan olika objekt, debugga SQL-frågor samt använda intelliSense när man skapar SQL-frågor vilket innebär en bekvämare och mer robust utvecklingsmiljö.



Figur 8 - LINQ's struktur[10]

[10] http://www.vsj.co.uk/articles/display.asp?id=618

Varje mittenkomponent i ovanstående figur ger oss funktionaliteten för att interagera med denna komponents datakälla. I databasfallet bygger man exempelvis ett datakontextobjekt och mappar databastabeller till klasser med vanlig funktionalitet som finns i Visual Studio 2008. Information om koppling till databasen kan man exempelvis ha i en konfigurationsfil.

4.1.5. WPF

Windows Presentation Foundation (WPF) är efterföljaren till Windows Forms och nästa generations plattform för att skapa användargränssnitt med ramverket Microsoft .NET. Visionen med WPF är att ge utvecklare tillgång till en ny grafikmotor som hanterar både vektor samt pixelgrafik och samtidigt utnyttjar de befintliga resurserna som finns i de flesta datorer idag i form av avancerade grafikkort[11]. Vektorgrafik har fördelen att oavsett skärmtyp så kan applikationen anpassa sig till både storlek och upplösning på skärmen och erbjuda den bästa upplevelsen av applikationen. Genom att skapa en gemensam arkitektur och programmeringsmodell oavsett typ av applikation får man möjlighet att kombinera 2-dimensionell och 3-dimensionell grafik på kraftfulla sätt. Vidare kan man i WPF dela upp det grafiska gränssnittet skrivet i xaml från kod för logiken skrivet i något annat språk vilket ger en oerhörd styrka då olika roller av utvecklare kan jobba parallellt med lösningen.

Extensible Application Markup Language (XAML) är en XML-strukturerad syntax för att hierarkiskt beskriva objekt och deras förhållanden. När vi använder oss av XAML för att beskriva vårt användargränssnitt så lägger vi till en namnrymd i filen som kopplar våra XAML-element mot klasser i WPF's objektmodell samt attribut i elementen till egenskaper och händelser på klasserna. XAML kan också innehålla kod men måste då kompileras till en exekverbar applikation. Vi kan fortfarande använda C# eller något annat språk i Visual Studio för att skapa dessa gränssnitt, XAML är bara ytterligare ett sätt som ger möjligheten att generera dessa användargränssnitt i andra verktyg.

Alla WPF-applikationer startar genom att skapa ett så kallat Application-objekt. Detta objekt kontrollerar applikationens livslängd och ansvarar för att leverera händelser och meddelande till programmet. Dessutom sköter den synbara fönster, resurser och programmets globala tillstånd. Endast ett Application-objekt kan finnas och köras åt gången. Detta objekt kan nås från hela programmet via den statiska propertyn Application.Current (Box & Sells, 2007). En applikation består av diverse kontroller där varje kontroll är en abstraktion av någon sorts funktionalitet/beteende. Exempelvis innehåller en knapp logik för att ta emot användarinteraktion när användaren trycker på knappen. Kontroller använder sig av så kallade stilar "styles" och mallar "templates" för att definiera sin layout, interna data och innehåll samt vad som ska hända vid förutbestämda händelser. Med hjälp av stilar och mallar kan alltså utvecklaren separera själva beteendet av en kontroll från utseendet ungefär så som vi gör på webbsidor med CSS. Layout är ett formellt koncept i WPF och innebär att kontroller som används för att positionera kontroller på användargränssnittet implementerar den funktionaliteten vilket innebär att andra kontroller som egentligen inte bryr sig om var någonstans de är slipper den logiken. Dessa kontroller kallas paneler och är mycket viktiga i WPF. Precis som tidigare kan kontroller i WPF innehålla data från flera olika källor exempelvis resursfiler, databaser, XML-strukturer och andra externa källor. Kontroller kan också innehålla nästintill vad som helst till skillnad från tidigare där varje kontroll var skapad för att innehålla en specifik typ av data. Exempelvis kunde en knapp tidigare bara innehålla text vilket skapade en hel del begränsningar.

Det finns egentligen hur mycket som helst att skriva om WPF, nedan beskrivs kort några nya viktiga koncept som kommer utnyttjas vid implementationen av programmet.

[11] http://download.microsoft.com/download/c/7/b/c7bd0b18-0825-453d-9211-269388fed153/02_WPF.ppt

Dependency properties

WPF introducerar en ny typ av property som kallas dependency property. Dess värde beror i varje ögonblick av ett antal olika leverantörer. Dependency properties har skapats främst för att erbjuda rik funktionalitet direkt från deklarativ kod. Dependency properties erbjuder framförallt tre saker:

- *Change notification*. Alltid när en dependeny property ändras kan WPF automatiskt trigga ett antal händelser baserat på dess metadata. Man kan exempelvis rendera om visuella element, uppdatera nuvarande layout eller databindningar.

- *Property value inheritance*. Dependency properties värden kan ärvas ned genom elementträdet vilket betyder att exempelvis en Grid´s värde kan ärvas av en knapp som finns placerad i Griden.

- *Support for multiple providers*. WPF går som man kan se i figuren nedan igenom en 5-stegs-process för att bestämma en dependency properties slutgiltiga värde.
    1. Först bestäms basvärdet utifrån det lokala värdet, style triggers, template triggers, style setters, theme style triggers, theme style setters, property value inheritance eller defaultvärde med minskande precedens.
    2. Om värdet är ett uttryck valideras detta sedan.
    3. Om en eller flera animeringar finns har de möjlighet att ändra värdet.
    4. WPF skickar värdet till en CoerceValueCallback delegate om någon registrerades med propertien. Denna ansvarar för att returnera det nya värdet baserat på valfri logik. Exempelvis använder ProgressBar detta callback för att begränsa värdet mellan ett minimum och maximum värde.
    5. Till sist skickas värdet till en ValidateValueCallback delegate om någon sådan registrerades med propertien. Denna måste returnera true om värdet är giltigt och false annars. False orsakar att ett exception kastas som avbryter hela processen.

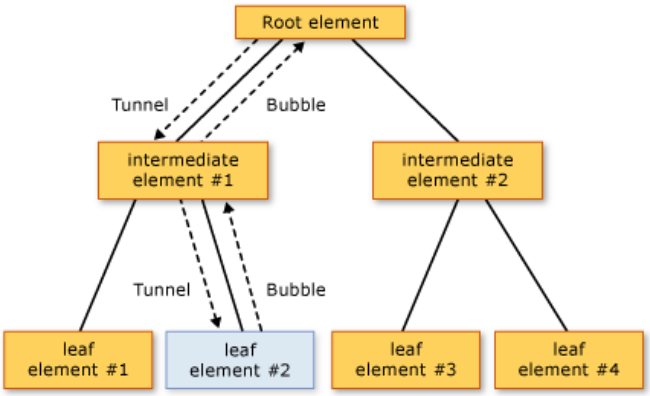
Figur 9: Processen för att beräkna en dependeny properties värde[12]

Det finns en speciell form av dependency properties som kallas attached properties. Denna kan bifogas med godtyckligt objekt. Exempelvis kan font sättas för en stackpanel trots att stackpanel inte har dessa properties genom att använda TextElement.FontSize och TextElement.FontStyle.

Routed events

Routed events är händelser som är designade för att fungera väl med elementträd. När en routed event uppstår kan den vandra upp eller ned genom ett träd av element. Routed events innebär att applikationer inte behöver känna till WPS:s elementsammansättning. Exempelvis när en användare trycker på en knapp så interagerar denne med något av knappens barnelement men tack vara att händelsen vandrar uppåt i det visuella trädet når det så småningom knappen som kan hantera den. Implementationen av detta beteende har mycket gemensamt med dependency properties.

[12] http://en.csharp-online.net/WPF_Concepts%E2%80%94Support_for_Multiple_Providers



Figur 10: Illustration av olika strategier för routed events[13]

När man registrerar en routed event väljer man en av tre tillgängliga strategier för hur händelsen ska färdas genom elementträdet:

1. *Tunneling:* Händelsen tas upp i rotelementet och färdas sedan nedåt i elementträdet till källan, det element där händelsen skedde, eller till den punkt där en hanterare markererar händelsen som hanterad.
2. *Bubbling:* Händelsen tas upp på källelementet och färdas sedan uppåt i trädet till rotelementet nås eller en hanterare markererar händelsen som hanterad.
3. *Direct:* Händelsen tas bara upp på det elementet där händelsen sker, källan. Detta är samma beteende som vanliga .NET-händelser har förutom att dessa händelser kan utnyttja mekanismer för routed events som exempelvis event triggers.

Commands

Genom använda commands kan man få kontroller som inte direkt känner till varandra att interagera på ett oberoende sätt. WPF har inbyggt stöd för commands som kan ses som en mer abstrakt och löskopplad version av events. Detta stöd kan kortfattat beskrivas i tre inslag:

1. Ett kommando är ett objekt som implementerar gränssnittet ICommand som har tre enkla medlemmar: Execute, CanExecute och CanExecuteChanged. Kontroller som exempelvis knappar har redan logik som ger möjlighet att interagera med kommandon genom XAML. Dessutom definierar WPF ett antal färdiga inbyggda kommandon så man slipper implementera ICommand själv. Dessa finns tillgängliga som statiska properties för fem olika klasser: ApplicationCommands, ComponentCommands, MediaCommands, NavigationCommands och EditingCommands.

2. Kommandon har automatiskt stöd för olika input såsom kortkommandon från tangentbordet. Exempelvis binds inbyggda kommandot ApplicationsCommands.Help till F1. Du kan även skapa dina egna bindningar genom att lägga till KeyBinding/MouseBinding objekt till relevanta objekts collection av InputBindings.

3. Vissa av WPF:s kontroller har inbyggt beteende bundna till olika kommandon. Exempelvis har TextBox sina egna inbyggna bindningar för Cut, Copy, Paste, Undo, Redo kommandon.

[13] http://msdn.microsoft.com/en-us/library/ms742806.aspx

Databindning

I WPF används ofta ordet data för att beskriva ett generellt .NET-objekt så databindning handlar alltså om att binda ihop .NET-objekt såsom exempelvis en collection, xml-fil, databastabell eller knapp. Ofta vill man visualisera exempelvis en databas, xml-fil eller liknande i ett WPF element som ListBox.

Nyckeln till databindningen är objektet Binding som binder samman två properties och håller en kommunikationskanal öppen mellan dem. Binding har source samt target och kan användas både i vanlig kod men även med XAML. Vanligast är att man binder dependency properties till varandra men även vanliga properties kan användas som source. Då dessa inte har någon automatisk uppdateringsfunktion måste man dock implementera interfacet INotifyPropertyChanged om man vill att bindningen ska reflektera uppdaterade värden. Det är ganska vanligt att flera element i ett användargränssnitt binder till olika properties på samma sourceobjekt. Istället för att markera varje Binding-objekt med source kan man sätta DataContext hos ett gemensamt FrameworkElement.

Databindning är lätt när source och target är av samma typ. Om så inte är fallet så erbjuder WPF två mekanismer för hur värdet från source tas emot och visas.

1. *Data Templates:* En data template är en bit UI som du kan applicera på ett generellt .NET-objekt när den renderas. Många WPF-kontroller har properties (av typen DataTemplate) för att bifoga denna mall och genom att sätta denna till en instans av DataTemplate kan man växla in ett helt nytt visuellt träd.

2. *Value Converters:* Medan data templates kan skräddarsy hur en target renderas kan value converters konvertera värdet från source till ett helt annat värde hos target. Exempelvis kan man ändra bakgrundsfärgen för ett element baserat på ett numeriskt värde från source. Detta kan ofta förekomma i en börsapplikation i exempelvis kurslistor då man vill förtydliga om värdet från source är positivt eller negativt i form av olika färger. Detta kan man åstakomma genom att peka Binding-objektets property Converter till en klass som implementerar gränssnittet IValueConverter. Denna klass har en metod Converter som anropas varje gång källans värde förändras.

WPF:s trådmodell

För att kunna bygga ett responsivt användargränssnitt måste man ha någon form av trådhanteringsstrategi. WPF använder typiskt 2 trådar[14]:

- Renderingstråden som verkar i bakgrunden och har hand om rendering.

- UI-tråden som tar emot input, hanterar händelser, målar skärmen och exekverar applikationskod.

UI-tråden köar olika arbetsuppgifter i ett Dispatcher-objekt. Dispatchern väljer arbetsuppgift enligt en prioriteringsbas och ser till att utföra den. Varje UI-tråd måste ha minst en Dispatcher och varje Dispatcher kan endast använda en tråd till att utföra arbetsuppgifter.

För att bygga ett användargränsnitt som ger snabb respons måste applikationen se till att Dispatcherns arbetsuppgifter är små för att minimera tiden Dispatchern lägger på var och en av dem. För att utföra resurskrävande operationer utan att blockera UI-tråden kan man använda en separat tråd som exekverar i bakgrunden och sedan rapporterar sitt resultat till UI-tråden. Detta är inte helt trivialt då Windows endast tillåter UI-element att nås av den tråd som elementet skapats av. Det

[14] http://msdn2.microsoft.com/en-us/library/ms741870.aspx

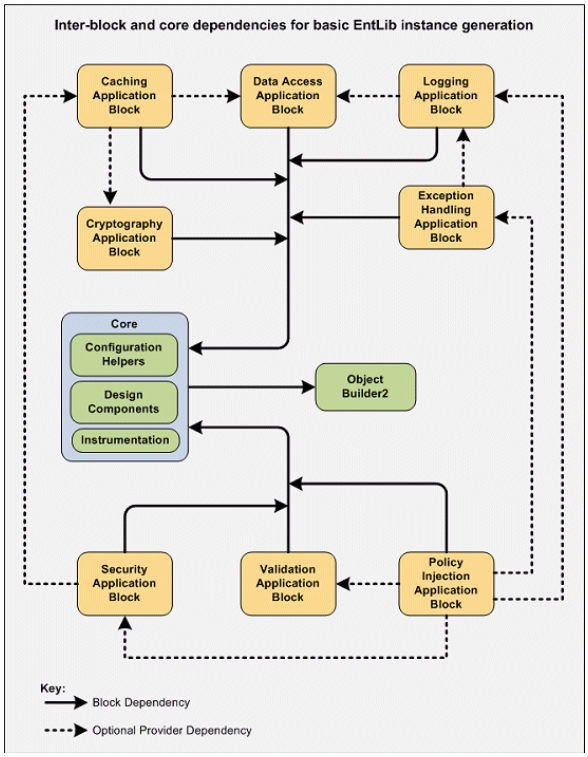
betyder att bakgrundstråden som vi använde för mer krävande arbeten inte kan nå och uppdatera vårt UI när den har exekverat färdigt.

WPF använder följande design för att koordinera UI-tråden med andra trådar. De flesta klasser i WPF ärver från DispatcherObject. Under konstruktion, lagrar ett Dispatcher-objekt en referens till den Dispatcher som länkas till aktuell tråd. På så sätt så skapas en association till tråden som skapade Dispatcher-objektet. Vid inledningen av varje metod i Dispatcher-objektet kallas VerifyAccess som jämför aktuell tråds Dispatcher med den Dispatcher som lagrades under objektets konstruktion – om dom inte matchar kastas ett exception.

Hur kan då en bakgrundtråd interagera med användaren om det bara är skaparen av ett dispatcherobjekt som kan nå den? Bakgrundstråden kan inte nå UI-elementet direkt men den kan be UI-tråden att utföra en uppgift genom att registrera en arbetsuppgift till UI-trådens Dispatcher och sedan använda dess Invoke eller BeginInvoke metod.

4.2. Microsoft Enterprise Library

För att man inte ska behöva återuppfinna hjulet gång på gång tillhandahåller Microsoft ett öppet kodbibliotek för att adressera några av branschens vanligaste kodningsutmaningar. Syftet är att både koncentrera problemställningarna och att föreslå lösningar enligt ”best practices” och diverse designmönster. Enterprise Library innehåller ett antal olika Application Blocks och den senaste versionen släpptes i maj år 2008. Dessa olika block bygger på en gemensam och flexibel grund vilken möjliggör anpassningar till specifika projekt.



Figur 11: De olika blocken i Enterprise Library och deras beroenden[15]

Fördelen med Enterprise Library är att du själv kan välja vilka block du vill använda och att både källkod och enhetstester finns tillgängliga vilket innebär att du kan modifiera koden till att passa

[15] http://msdn.microsoft.com/en-us/library/cc511878.aspx

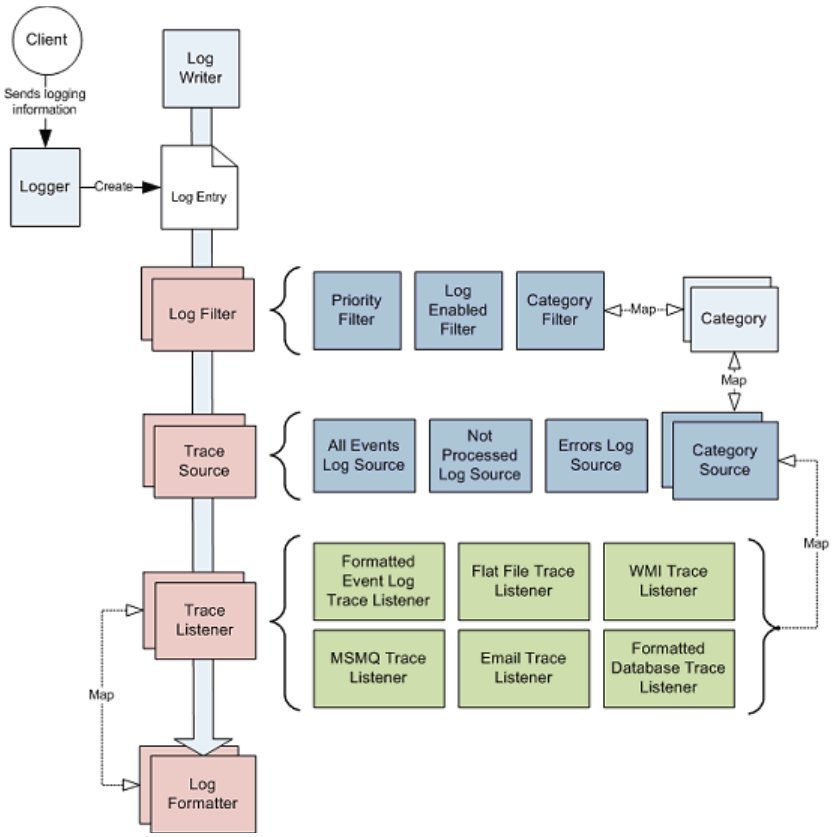
specifika behov. Med Microsoft Enterprise Library följer verktyget Enterprise Library Configuration som låter dig konfigurera din konfigurationsfil app.config fil på ett intuitivt och enkelt sätt.

I detta projekt används Enterprise Library för loggning och felhantering ofta med hjälp av policy injection. Dessa block beskrivs lite närmare nedan.

4.2.1. Logging Application Block

Ett block som används i detta projekt är Logging Application Block som ger stöd för loggning. Många applikationer, speciellt större programsystem kan ofta dra fördel av en konsistent metod att sköta loggningen. Detta block tillåter konfiguration av vad som ska loggas, när det ska loggas samt var och hur loggen ska sparas. Passar ingen av de inbyggda loggningsmetoderna som exempelvis loggning till databas, fil, email, xml med mera går det alldeles utmärkt att bygga sin egen anpassade loggningsmetod.

Applikationen kan generera information som blocket ska logga genom att använda en Logger-klass som automatiskt skapar ett loggningsobjekt och populerar detta objekt med information att logga.



Figur 12: Schematisk figur över hur Logging Application Block fungerar[16]

Loggningsobjektet skickas sedan vidare genom de filter som definierats i konfigurationen. Dessa filter kan blockera detta objekt baserat på kategori, prioritet eller när loggning ej är aktiverad. Om inget filter blockerar tar loggskrivaren emot lämplig källa. Man kan konfigurera en eller flera lyssnare för varje källa. Loggskrivaren kommer skicka loggningsobjektet till de matchande lyssnare som specificerats. Lyssnarna använder sedan en formatterare för att transformera informationen i loggningsobjektet till lämpligt format.

[16] http://msdn.microsoft.com/en-us/library/cc511933.aspx

4.2.2. Exception Handling Application Block

Hjälper utvecklare och beslutsfattare att skapa en konsistent strategi för felhantering i en applikation. Detta block har designats för att erbjuda lösningar till alla de vanligaste problem en utvecklare ställs inför när det gäller felhantering.

Idén är att man ska definiera policies och i konfigurationen för dessa tala om hur exceptions som följer en viss policy ska behandlas. Varje policy kan hantera olika typer av fel på olika sätt. För varje policy definierar man ett antal hanterare som man anropar då ett visst exception inträffar. Blocket har fyra fördefinierade hanterare men det är enkelt att lägga till egna om det skulle finnas behov av det. De som finns är:

- *Wrap Handler*. Täcker in ett exception i ett annat exception.
- *Replace Handler*. Ersätter ett exception med ett annat exception.
- *Logging Handler*. Formatterar information i ett exception och vidarebefodrar denna information till Logging Application Block.
- *Fault Contract Exception Handler*. Designad för att användas med WCF.

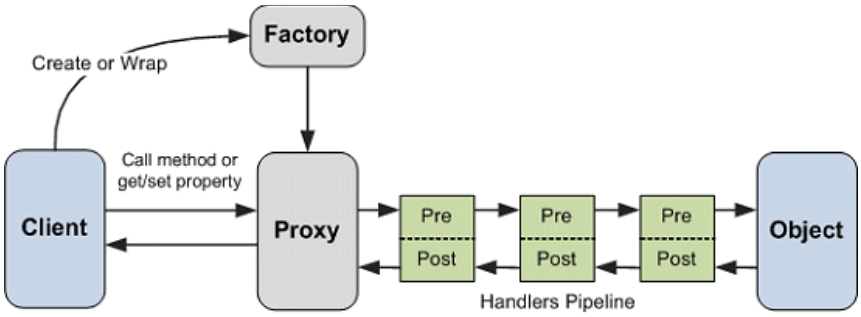
När man konfigurerar policies har man även friheten att ange vad som ska hända efter hanteraren har gjort sitt. Man kan välja mellan None, NotifyRethrow, ThrowNewException.

Exception Handling Application Block används med fördel i situationer där man vill ha ett enhetligt och flexibelt sätt att hantera exceptions. Exempelvis kanske man vill behandla exceptions från alla komponenter i ett visst lager på samma sätt. Krav förändras ofta och då kan man genom att använda detta block bara ändra policies i konfigurationen utan att behöva ändra någon källkod.

4.2.3. Policy Injection Application Block

Policy Injection Application Block (PIAB) kan användas för att separera logikkod från annan kod som sköter exempelvis felhantering, loggning, validering med mera. På så sätt blir koden lättare att läsa och underhålla.

PIAB kräver att man antingen skapar upp objektet med PIAB (PolicyInjection.Create) eller att man wrappar upp ett redan skapat objekt med PIAB (PolicyInjection.Wrap). Dessa metoder returnerar en Proxy till objektet, där Proxyn har ett antal hanterare som när man anropar en metod hanterar de attribut som metoden har knuta till sig. Om till exempel valideringshanteraren fallerar på valideringen av parametrarna kommer den att avbryta kedjan av hanterare med ett exception.



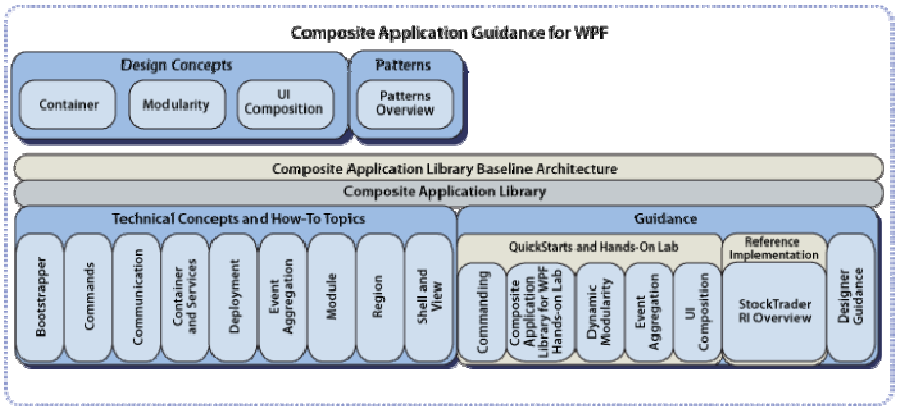
Figur 13: Design av PIAB[17]

[17] http://msdn.microsoft.com/en-us/library/cc511729.aspx

Policies är definitioner av vilka hanterare som ska exekvera för vilka objekt och metoder. Det finns två olika typer av policies, konfigurationbaserade samt attributbaserade policies.

4.3. Composite Application Guidance for WPF

I juni 2008 släppte Microsoft Composite Application Guidance for WPF (CAG) som är ett paket med guidning kring hur komposita applikationer baserade på WPF bör utvecklas. CAG är ett ramverk som erbjuder hjälp för att utveckla klientapplikationer i WPF enligt konstens alla regler och designmönster. CAG består av en referensimplementation, källkod, dokumentation, QuickStart-handledningar och labbar. Genom att använda CAG får man en bra infrastruktur för att ladda, skapa, initiera och organisera presentation samt kommunikation mellan olika moduler och komponenter. Nedan ses en figur som visar hur dokumentationen är uppbyggd.

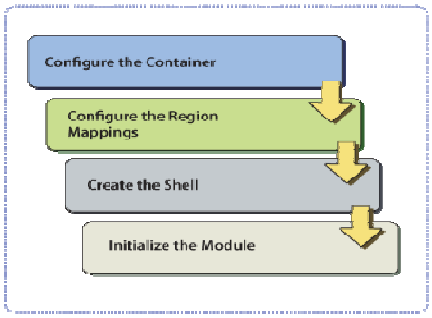


Figur 14: Composite Application Guidance for WPF dokumentation[18]

Nedan beskrivs kortfattat de tekniska koncept av CAG som denna applikation använder. Detta för att man ska få en känsla för vad ramverket erbjuder för funktionalitet samt lättare ska kunna ta till sig nästa kapitel som handlar om implementeringen av applikationen som kommer ha CAG som grund.

4.3.1. Bootstrapper

Ansvarar för initialiseringen av en applikation som använder CAG. Bootstrappern ger bättre kontroll över hur komponenterna i CAG binds till applikationen. I en vanlig WPF-applikation specificeras en URI i App.xaml som kör igång huvudfönstret. I en CAG-applikation är detta bootstrappern:s ansvar då huvudfönstret beror av vissa tjänster som behöver registreras innan detta kan visas. CAG innehåller en default abstrakt klass UnityBootstrapper som hanterar denna initialisering och använder Unity containern.



Figur 15: Bootstrappingprocess[19]

[18] http://msdn.microsoft.com/en-us/library/cc707844.aspx
[19] http://msdn.microsoft.com/en-us/library/cc707847.aspx

4.3.2. Container

Applikationer som baserar sig på CAG är ofta sammansatta av flera löst kopplade moduler. Dessa moduler behöver interagera med huvudfönstret och med varandra. För att binda samman de olika modulerna så är CAG-applikationer beroende av en dependency injection container. Denna container innehåller ett antal tjänster som erbjuder funktionalitet till andra moduler på ett löst kopplat sätt. Fördelar med att använda en container består i:

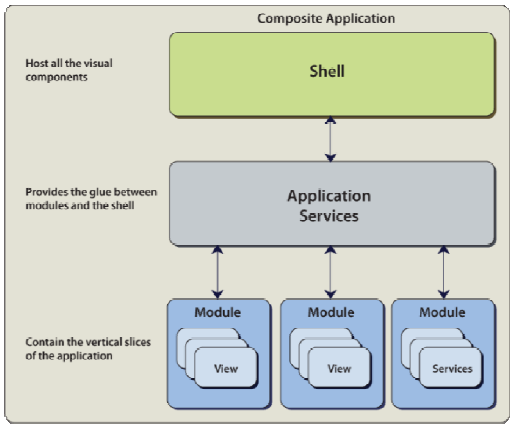
- En komponent behöver inte lokalisera sina beroenden eller hålla koll på dess livstid.
- Man kan ändra implementationen av beroenden utan att komponenten påverkas.
- Främjar testning då beroenden kan mockas.
- Lättare att underhålla då nya tjänster lätt kan initieras.

4.3.3. Services

En applikation som baserar sig på CAG är sammansatt av ett antal tjänster som applikationen konsumerar. Dessa tjänster injiceras genom containern. Förutom bastjänsterna (IModuleEnumerator, IModuleLoader, IRegionManager, IEventAggregator, ILoggerFacade) ramverket erbjuder kan man lätt lägga till sina egna applikationsspecifika tjänster om man är i behov av detta.

4.3.4. Module

En modulär design innebär att man delar upp ett system i ett antal olika funktionella enheter, moduler, som kan sättas samman till en större applikation. En modul kan inkludera komponenter som vyer och affärslogik samt infrastruktur som loggning. Moduler är oberoende av varandra men kan kommunicera med varandra på ett löst kopplat sätt.



Figur 16: Modulsammansättning[20]

Moduler i CAG definieras på ett sådant sätt att de kan upptäckas och laddas av applikationen under exekvering. Liksom en applikation kan en modul vara uppdelad i olika lager. En modul kan bestå av presentationslager, affärslager samt resursaccesslager enligt nedanstående figur. Men det går också dela upp en applikation där man väljer att ha en modul för alla vyer, en för alla applikationsspecifika tjänster och så vidare.

En modulklass i CAG implementerar interfacet IModule som innehåller en enkel Initialize-metod som anropas under modulens initieringsprocess. Modulerna kan behöva access till beroenden, dessa konstrueras av containern och kan som nämndes ovan injiceras i modulens konstruktor. När modulen

[20] http://msdn.microsoft.com/en-us/library/cc707850.aspx

initieras kan den exempelvis registrera vyer och tjänster. Beroende på vilken container som används kan registreringen även göras genom konfiguration. Fördelen med att göra den i kod är att registreringen endast sker om modulen faktiskt laddas.

CAG erbjuder olika typer av laddning. Vid statisk laddning har huvudfönstret en direkt referens till modulerna vilket innebär enklare användning och debugging. Nackdelen är att man måste lägga till nya referenser i huvudfönster-klassen och en rad kod i bootstrappern för varje modul man vill ladda. Vid dynamisk laddning upptäcks modulerna vid körning. Fördelen ligger i att moduler kan läggas till en applikation utan att den behöver modifieras eller få några nya referenser. En annan fördel är att man vinner flexibilitet då man kan välja hur modulerna ska upptäckas. CAG erbjuder support för att ladda moduler genom att scanna en katalog från konfiguration eller på begäran.

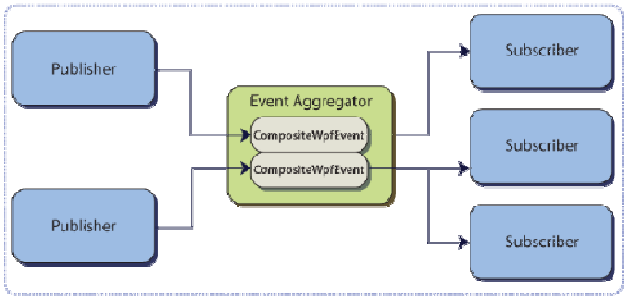
4.3.5. Shell and View

Shell är applikationens huvudfönster där det primära användargränsnittet visas. Det kan innehålla namngivna regioner där modulerna kan lägga till vyer. Det kan också definiera speciella toppnivåelement som menyer och verktygsfält. Shell definierar även applikationens övergripande utseende och känsla i exempelvis stilar och mallar.

En vy är en samling av UI-komponenter som definierar en del av hela användargränsnittet.
Även om det inte är ett krav från CAG bör man använda något av flera möjliga designmönster när manimplementerar en vy för att skilja vyns layout från dess logik. Detta är viktigt både för att man ska kunna testa och underhålla vyn.

4.3.6. Event Aggregator

EventAggregator erbjuds som en tjänst i containern och kan hämtas via interfacet IEventAggregator. Denna är ansvarig för att lokalisera och bygga händelser samt hålla en collection av händelser i systemet.



Figur 17: EventAggregator[21]

Klassen CompositeWpfEvent är en implementation av EventBase som underhåller en lista av prenumeranter och hanterar spridandet av händelser till dessa. Man kan sedan låta sina egna applikationsspecifika händelser ärva av CompositeWpfEvent. Då händelser ofta delas mellan olika moduler kan det vara lämpligt att definiera dem på ett generellt ställe.

4.3.7. Commands

Kommandon är ett sätt att hantera UI-händelser. Med hjälp av kommandon kan man på ett oberoende sätt binda UI till logiken som utför uppgiften. CAG introducerar två nya kommandon som innebär att kommandona kan bli vidarebefodrade utanför det logiska trädets gränser och inte behöver hantering i bakomliggande kodfil:

- *DelegateCommand* använder en delgat som strategi för att anropa en hanteringsmetod.

[21] http://msdn.microsoft.com/en-us/library/cc707867.aspx

- *CompositeCommand* är ett kommando som har multipla barnkommandon.

4.4. Designmönster

Designmönster har under senare år blivit ett alltmer vanligt förekommande begrepp inom objektorienterad systemutveckling. Designmönster är ett praktiskt sätt att beskriva återkommande mönster i designlösningar som har visat sig fungera. Det finns oräkneligt många designmönster, men ett antal av dessa har skrivits ner, namngivits och publicerats. Tack vare detta kan alla utvecklare världen över, som har kunskap om dessa mönster, återanvända designlösningarna i sina egna program. Dessutom får utvecklarna genom designmönstren en gemensam terminologi som gör det möjligt att tänka, beskriva och förstå varandra på en högre nivå.

Att lära sig designmönster är förmodligen det mest effektiva sättet att stärka sina designkunskaper. Designmönster har dessutom visat sig kunna höja abstraktionsnivån vid designbeslut, vilket dels leder till en effektivisering av denna process, men även minskar risken att fel begås då de döljer viss komplexitet.

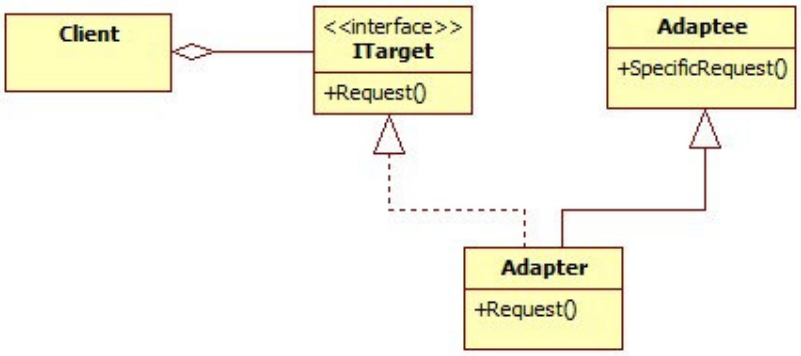
Gang Of Four (GoF) kallas den första och mest kända samlingen av designmönster. Där definieras 23 designmönster som delas in i tre olika kategorier.

- *Creational Patterns* beskriver hur objekt skapas, sätts samman samt hur dom representeras. Designmönstren abstract factory, builder, factory method, prototype samt singleton hör till denna kategori.
- *Structural Patterns* tittar på hur klasser och objekt är kopplade till varandra för att ingå i större strukturer. I denna kategori hittar man mönstrena adapter, bridge, composite, decorator, facade, flyweight samt proxy.
- *Behavioral Patterns* berör algoritmer och fördelningen av ansvar mellan objekt och klasser. Designmönster som beskriver kommunikation mellan objekt finns här. Chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method och visitor är namnen på mönstren i denna kategor.

Nedan beskrivs kort teorin bakom de designmönster som senare kommer användas under implementationen av systemet.

4.4.1. Adapter

Adapter är ett mångsidigt mönster som förenar olika typer som från början inte designats för att fungera ihop. Detta kan vara mycket användbart när man använder tredjepartsbibliotek där man inte har tillgång till koden.



Figur 18: Designmönstret Adapter

ITarget: Interfacet klienten vill använda.

Adaptee: En implementation som behöver adapteras.

Adapter: Klassen som implementerar ITarget.

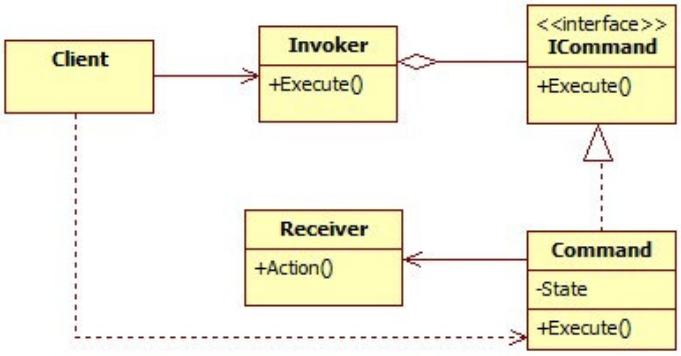
Request: En metod klienten vill använda.

SpecificRequest: Implementationen av Request's funktionalitet i Adaptee.

En av fördelarna med Adapter är att mönstret främjar programmering mot interface. Klienten arbetar mot ett domänspecifikt gränssnitt. Adaptteen erbjuder önskad funktionalitet men med ett annorlunda gränssnitt. Adapterobjektet implementerar ITarget och slussar vidare klientens anrop till Adaptee. Klienten är bara medveten om ITarget interfacet.

4.4.2. Command

Detta designmönster skapar distans mellan klienten som anropar en metod och objektet som kan utföra den. Mönstret används exempelvis vid menyhantering. Klienten har ett speciellt sätt att säga vad den vill ha gjort, den tänker i termer som öppna, kopiera, klistra in och så vidare. Objektet Receiver som det kan finnas flera olika av vet hur man utför dessa förfrågningar. Kommandoobjektet vet vidare vilken Receiver och Action som ska anropas. Invoker finns där för att skapa distans mellan klienten och Receiver. Interfacet ICommand ser till att alla kommandon håller en viss standard.



Figur 19: Designmönstret Command

Client: Skapar och kör kommandon.

ICommand: Interface som specificerar metoden Execute.

Invoker: Frågar Command om den kan utföra uppgiften i form av metoden Action.

Command: En klass som implementerar metoden Execute genom att anropa metoder på en Receiver.

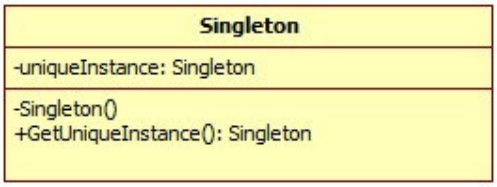
Receiver: En klass som kan utföra den efterfrågade metode Action.

Action: Metod som ska utföras.

4.4.3. Singleton

Detta designmönster är väldigt vanligt förekommande och används när man vill begränsa antalet instanser av en klass till en enda. Detta kan man åstakomma genom att man skapar en klass med en metod som skapar en instans om det inte redan finns en. Skulle det redan finnas en instans av

klassen returnerar metoden en referens till denna. För att förhindra skapandet av flera instanser görs konstruktorn privat.



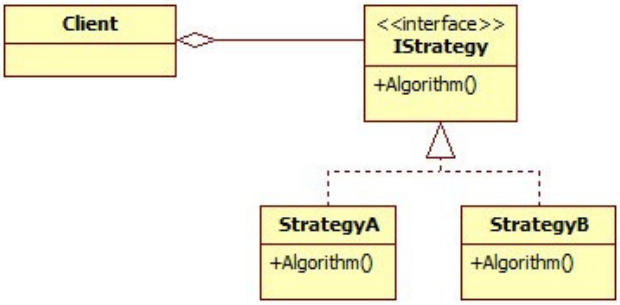
Figur 20: Designmönstret Singleton

Singleton: Klassen som innehåller mekanismen att kunna skapa en unik instans av sig själv.

Det finns några olika sätt att implementera singleton på. I detta program används CAG-ramverkets färdiga implementation.

4.4.4. Strategy

Ibland kan det finnas flera strategier för att lösa ett visst problem. Det kan leda till rörig kod med många villkorssatser. Hur designar man bäst relaterade men varierande algoritmer? Hur ska man designa för att tillåta dessa algoritmer att variera? Flytta ut algoritmerna från värdklassen och definiera varje algoritm eller strategi i en egen klass med ett gemensamt interface. Strategy låter klienterna välja vilken algoritm de vill använda och ger dem ett smidigt sätt att nå dem.



Figur 21: Designmönstret Strategy

Context: En klass som har hand om kontextinformation för IStrategy-objektets algoritm att jobba med.

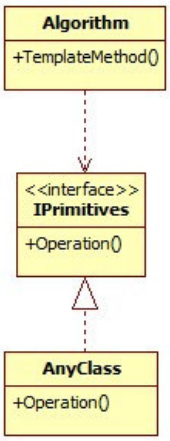
IStrategy: Definierar ett gemensamt interface för alla strategier.

StrategyA, StrategyB: Klasser som inkluderar algoritmer som implementerar IStrategy.

Genom att använda ett givet kontext kan klienten välja lämplig strategi från en familj av tillgängliga strategier. Kontext innehåller ofta en switch eller if-sats för att bestämma vilken strategi som ska användas. Kontext kan exempelvis vara en metod som tar hand om en knapptryckning.

4.4.5. Template Method

Låter algoritmer variera i vissa steg genom att flytta ut dessa specifika steg till subklasser. Strukturen av algoritmen förändras alltså inte men små väldefinierade delar av den behandlas någon annanstans.



Figur 22: Designmönstret Template Method

Algorithm: En klass som inkluderar en TemplateMethod.

TemplateMethod: En metod som överlåter vissa delar (operationer) på andra klasser.

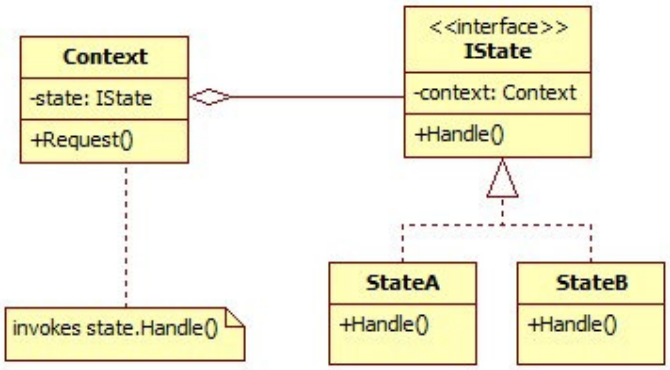
IPrimitives: Interface som definierar de delar (operationer) som TemplateMethod överlåter.

AnyClass: En klass som implementerar IPrimitives.

Operation: En av de metoder som TemplateMethod behöver för att slutföra sitt uppdrag.

4.4.6. State

Kan ses som en dynamisk version av mönstret Strategy som beskrevs ovan. Ett objekts beteende beror på vilket status det har. Detta kan åstakommas genom att låta en objektvariabel ändra sin subklass.



Figur 23: Designmönstret State

Context: En klass som håller en variabel av typen IState som pekar på ett viss objekt State som i sin tur definierar nuvarande kontext och det gränssnitt som kan vara av intresse för olika klienter.

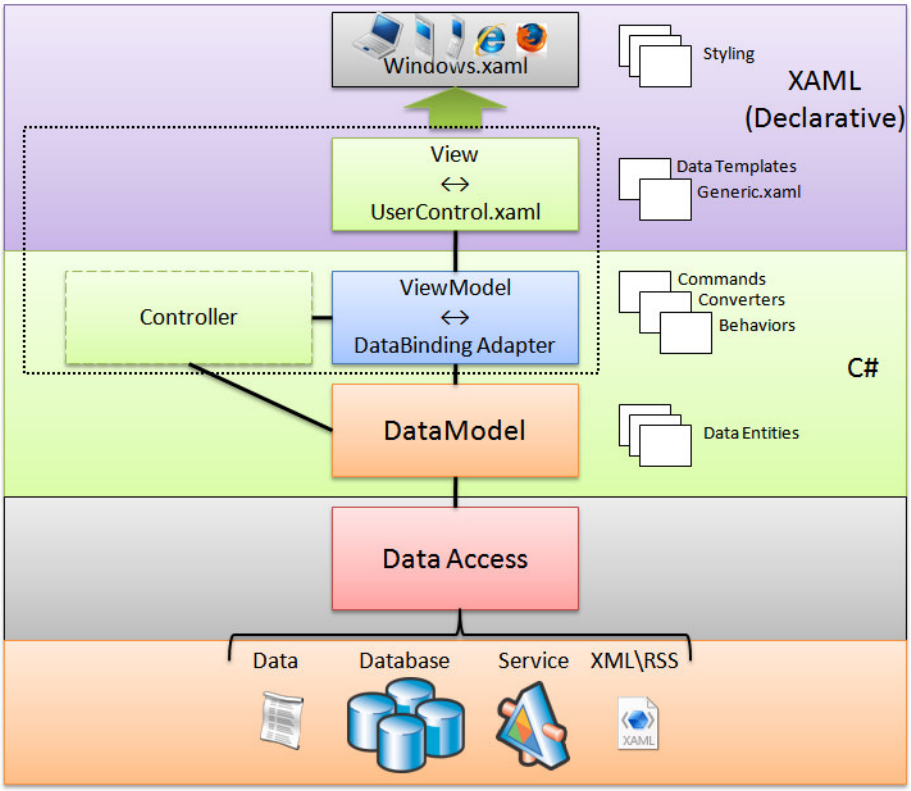
IState: Definierar ett interface för ett speciellt state hos Context.

StateA och StateB: Klasser som implementerar beteendet associerat med ett visst state hos Context.

Som ses i ovan diagram har aktuellt State-objekt alltid full accesss till den data objektet Context håller. Det innebär att antingen Context eller aktivt State när som helst kan besluta att det är dags att byta State. Detta sker genom att attributet state i Context pekas om till en annan implementation av IState vilket leder till att förfrågningar genast börjar skickas till nytt State-objekt som kan ha ett helt annorlunda beteende än föregående State. Olika States kan implementeras som egna klasser eller delegates om de inte innehåller så mycket data.

4.4.7. DataModel-View-ViewModel

WPF erbjuder stor möjlighet för designers och programmerare att tillsammans sida vid sida kunna utveckla riktigt bra applikationer. Denna kraftfullhet kommer dock inte utan komplexitet och WPF erbjuder tyvärr inget tydligt ramverk eller klara riktlinjer för hur man ska lösa många av de problem som uppstår vid utvecklandet av en applikation. Efter att ha läst diverse blogginlägg och artiklar kan man konstatera att många experter rekommenderar DataModel-View-ViewModel (DM-V-VM) vid utveckling av WPF-applikationer. Det är en variation av det välkända designmönstret Model-View-Controller (MVC) som är specialanpassat för WPF. Nedanstående diagram beskriver en typisk arkitektur för en applikation som är utvecklad enligt detta mönster.



Figur 24: Designmönstret DataModel-View-ViewModel[22]

DataModel ansvarar för att tillhandahålla data som är lätt för WPF att konsumera. Dess publika API får endast anropas från UI-tråden. Vidare måste DataModel implementera INotifyPropertyChanged eller INotifyCollectionChanged. DataModel är helt UI-oberoende och abstraherar krävande operationer som datahämtning utan att blockera UI-tråden. Den kan användas för att kombinera data från olika källor.

[22] http://www.ekampf.com/blog/2008/03/18/DevelopingARobustDataDrivenUIUsingWPFIntroduction.aspx

View består av visuella element och representerar det verkliga gränsnittet som användaren ser såsom fönster, knappar och grafik. Mönstret vi använder är att sätta DataContext för en View till dess ViewModel. Det gör det enkelt att komma åt ViewModel genom att använda WPF:s avancerade databindningsmekanism. Idealt implementeras View helt i XAML utan bakomliggande kodfil. I ett enkelt scenario kan vi binda View direkt till DataModel fast för det mesta kräver DataModel´s värden någon form av processande som konvertering eller beräkning innan de binds till ett UI-element. Det kan också handla om flera komplexa operationer som inte passar i en strikt endast deklarativ definition för en View men som är alltför applikationsspecifik för att ingå DataModel. Möjligen vill vi också spara en viss tillståndsdata för View.

För att bygga en bro över gapet mellan View och DataModel skapar vi ViewModel som utför ovan nämnda uppgifter. Den utgör en modell för en View i applikationen och abstraherar all beteendelogik för denna View vanligen med Commands. ViewModel kan även inkludera konverterare som kan transformera DataModel-typer till View-typer. ViewModel är ganska specifik för en viss View i applikationen men den ärver ingen WPF-klass och gör inga antaganden om det UI som kommer bindas till den.

4.5. Designprinciper

Det finns ett antal designprinciper vid objektorienterad programmering man bör försöka följa för att uppnå en god programdesign. Nedan återges kort ett antal designprinciper som finns beskrivna i boken Design Patterns (Freeman & Freeman, 2004).

1. *Föredra komposition framför arv*. Detta ger dig mer flexibla system som låter dig ändra ett beteende vid körning. Vidare innebär komposition mindre beroende mellan de olika klasserna än arv där bas- och subklass är tätt ihopkopplade. En nackdel med komposition kan vara att ett systems beteende kan vara svårare att förstå genom att endast kolla på koden. Komposition används i många designmönster exempelvis Strategy.

2. *Inkapsla de delar av din applikation som varierar*. Identifiera och separera de delar av ditt system som kommer variera i takt med nya krav från de delar som förblir oförändrade. Detta leder till färre ofrivilliga konsekvenser vid kodändringar samt mer flexibla system. Denna princip utgör grund för i stort sett alla designprinciper exempelvis Strategy.

3. *Programmera mot ett interface, inte en implementation*. Deklarerade variablers typ bör vara i form av en supertyp, vanligen ett interface eller en abstrakt klass. På detta sätt kan objekten som tilldelas variablerna vara av valfri konkret implementation av supertypen vilket leder till att klassen som deklarerar dom inte behöver veta något om den konkreta objekttypen.

4. *Skilj modell från design*. Låt inte ett icke UI-objekt ha en referens till ett UI-objekt. UI-objekten tillhör en speciell applikation medan icke UI-objekten idealt ska kunna återanvändas till flera användargränssnitt eller applikationer. Vidare bör inte UI-objekt ha någon applikationslogik i sig. Det enda de ska göra är att initiera UI-element, ta emot UI-händelser och delegera förfrågningar till icke UI-objekt.

5. *Sträva efter oberoende mellan objekt som interagerar*. En löst kopplad design låter oss bygga flexibla objektorienterade system som kan hantera förändring. Designmönstret Observer är ett utmärkt exempel på detta.

6. *Klasser bör vara öppna för uttökning men stängda för modifiering*. Vårt mål bör vara att klasser ska kunna uttökas genom att lägga till nytt beteende utan att behöva modifiera

existerande kod. Detta leder till flexibla applikationer som är tillräckligt flexibla för att lätt kunna förändras i takt med att nya krav.

7. *Principen om minsta möjliga vetskap – prata bara med dina direkta vänner.* Var vaksam på hur många klasser ett objekt är beroende av och också hur det ska interagera med dessa klasser. Detta hindrar oss från att skapa ett bräckligt system med många beroenden som är kostsamt att underhålla och svårt för andra att förstå.

8. *Anropa inte oss, vi anropar dig*. Högnivåkomponenter bestämmer när lågnivåkomponenter behövs och ska anropas.

9. *En klass bör endast ha en anledning att ändras*. Denna princip leder oss till att tilllåta varje klass att endast ha ett ansvar. Har en klass flera ansvarsområden har den även flera olika anledningar den kan behöva ändras för.

5. Implementation

Detta kapitel beskriver implementationen av det nya systemet och de designval som tagits vid utvecklingen av systemets olika delar.

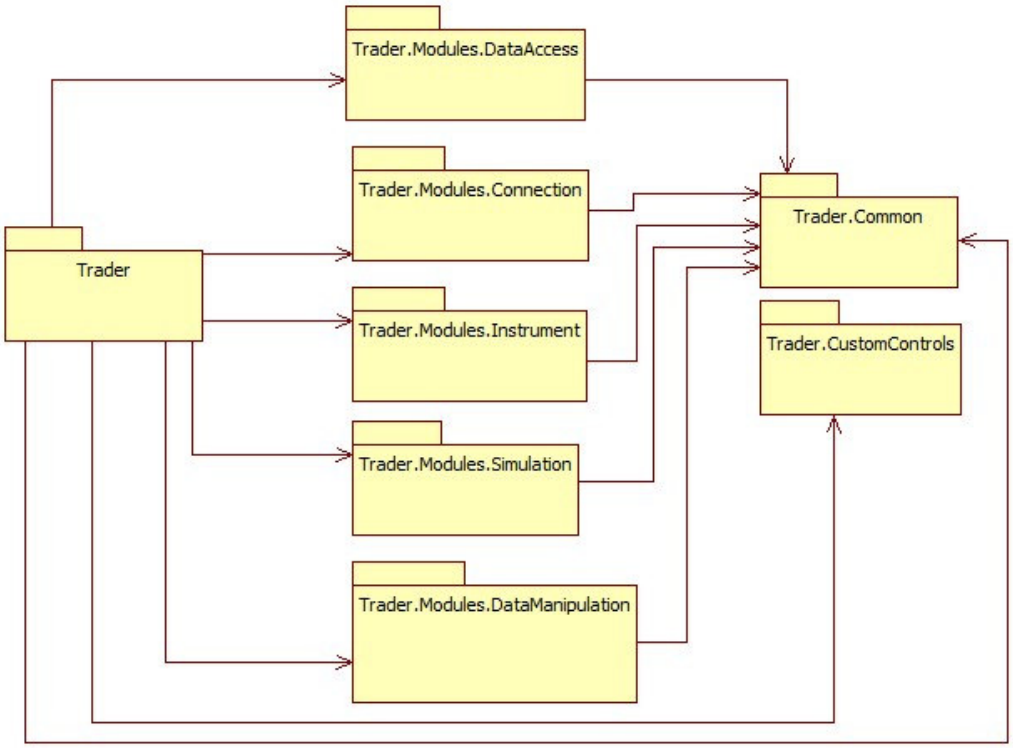
Börshandelssystemet kommer att byggas som en Windowsapplikation med hjälp utvecklingsverktyget Visual Studio 2008 som erbjuder god support och ger möjlighet till hög produktivitet vid utveckling i .NET. Systemet är uppbyggt kring diverse objekt och därför valdes C# 3.0, ett modernt programmeringsspråk som ger bra stöd för objektorientering. Vidare används SQL Server 2008 Express som lokal databas att lagra kursdata och annat i.

5.1. Övergripande design

Då huvudkravet på detta system fastställdes i förstudien till att programmet måste vara utbyggbart har det byggts upp ett ramverk av ett antal komponenter som var och en löser en specifik uppgift. En modulbaserad uppbyggnad främjar underhåll, utbyggbarhet och testning.

I ett första försök utvecklades det egna ramverket TradingFramework som skötte alla ramverkstypiska uppgifter som reflektion, modulladdning, loggning, felhantering med mera. Resultatet blev ett fungerande men oflexibelt ramverk med alltför många beroenden mellan de olika delarna. När Microsoft sedan släppte sitt Composite Application Guidance for WPF (CAG), ett ramverk som erbjuder infrastruktur för att ladda, skapa, initiera, organisera presentation och kommunikation mellan moduler och komponenter i början av sommaren krävdes en ny utvärdering. Varför lura sig själv och tro att man kan uppfinna hjulet bättre än ett stort antal experter på Microsoft. Då CAG är uppbyggt enligt diverse designmönster och ”best practises” samt har helt fritt tillgänglig källkod kändes valet att övergå till detta självklart.

Ett nytt ramverk Trader med CAG i botten utvecklades. Trader består av ett antal olika projekt vars beroenden ses i figuren nedan.



Figur 25: Trader översikt

Ett mål var att åstadkomma en lösning med så få beroenden som möjligt. Det första ramverket TradingFramework hade betydligt fler beroenden och då speciellt olika moduler emellan. I ett första skede finns ju endast sex stycken men på sikt kommer det förhoppningsvis finnas betydligt många fler moduler och då kan man lätt förstå vad som skulle hända med antalet beroenden om man inte löser detta på ett bra sätt. Genom att använda CAG så är modulerna oberoende av varandra men kan ändå kommunicera på ett löst kopplat sätt genom kommandon och händelser.

WPF erbjuder stor möjlighet för designers och programmerare att tillsammans sida vid sida kunna utveckla riktigt bra applikationer. Denna kraftfullhet kommer dock inte utan komplexitet och WPF erbjuder tyvärr inga klara riktlinjer för hur man ska lösa många av de problem som uppstår vid utvecklandet av en applikation. Efter att ha läst diverse blogginlägg och artiklar kan man konstatera att många experter rekommenderar designmönstret DataModel-View-ViewModel (DM-V-VM) vid utveckling av WPF-applikationer[23]. Det är en variation av det välkända designmönstret Model-View-Controller (MVC) som är specialanpassad för WPF. För att få en modulbaserad applikation som klart och tydligt separerar den visuella delen från beteendet och ordentligt kunna testa beteendet togs valet att utgå från designmönstret DataModel-View-ViewModel och anpassa detta till denna applikations specifika krav.

De moduler med användargränssnitt är uppbyggda enligt samma princip. Implementationen kan man säga består av fyra olika delar:

1. *View* består av visuella element och representerar det verkliga gränsnittet som användaren ser såsom fönster, knappar och grafik. Views är specificerade helt i xaml utan bakomliggande kodfil som DataTemplates och finns samlade i ViewResourceDictionary i huvudprojektet Trader. På så sätt kan olika Views på ett smidigt sätt dela olika stilar för att få ett enhetligt utseende. Varje DataTemplate's DataType sätts till respektive vys ViewModel för att möjliggöra databindning.

2. *ViewModel* utgör en modell för en View i applikationen och abstraherar all beteendelogik med kommandon. ViewModel kan ses som bryggan mellan View och resten av programmet. ViewModel är ganska specifik för en viss View i applikationen men den ärver ingen WPF-klass och gör inga antaganden om det UI som binds till den. Då dessa klasser är separerade från UI-elementen är de också relativt lätta att enhetstesta.

3. *DataModels* är enkla klasser som representerar data. Alla datamodeller implementerar INotifyPropertyChanged för att man ska kunna databinda Views till dem via en ViewModel.

4. *Serviceklasser* ansvarar för att tillhandahålla data till ViewModel. Det kan handla om data från börsen, lokal databas eller xml-filer. Service-klassernas publika API:n får endast anropas från UI-tråden och de är helt UI-oberoende. Krävande operationer abstraheras utan att blockera UI-tråden för att få ett användargränssnitt som svarar snabbt på användarens input.

Det kan verka lite magisk vid första anblicken men all kommunikation mellan View och ViewModel sker alltså med databindning och kommandon. För att illustrera hur View och ViewModel generellt pratar med varandra ges nedan ett exempel hämtat från en av applikationens moduler.

Knappar definieras i View på följande vis:

```
<Button Command="{Binding UpdateInstrumentCommand}">Uppdatera kurshistorik</Button>
```

[23] http://blogs.sqlxml.org/bryantlikes/archive/2006/09/27/WPF-Patterns.aspx

Ovanstående knapp används för att uppdatera kurshistorik och är alltså kopplad till kommandot UpdateInstrumentCommand i dess ViewModel:

```
public DelegateCommand<object> UpdateInstrumentCommand { get; set; }
UpdateInstrumentCommand = new DelegateCommand<object>(UpdateInstrument,
CanUpdateInstrument);
```

När man skapar kommandot anges två metoder, UpdateInstrument som anropas när knappen trycks in och CanUpdateInstrument som returnerar en bool som talar om huruvida knappen ska vara aktiv eller inte. Dessutom har man möjlighet att skicka med en parameter med kommandot. För att knappen ska vara aktiv vill vi i detta fall dels att ett instrument i instrumentlistan är valt dels att vi är uppkopplade mot marknaden.

```
private bool CanUpdateInstrument(object arg)
{
	if (SelectedInstrument != null &&
	container.Resolve<IConnection>().ConnectionState !=
	Enums.ConnectionStatus.NotConnected)
	{
		return true;
	}
	return false;
}
```

Tyvärr anropas inte CanUpdateInstrument automatiskt annat än när knappen skapas så detta måste vi göra själva genom att anropa UpdateInstrumentCommand.RaiseCanExecuteChanged() då något av de två villkoren förändras. Kopplingen till marknaden håller ViewModel koll på genom att prenumenera på ConnectionEvent:

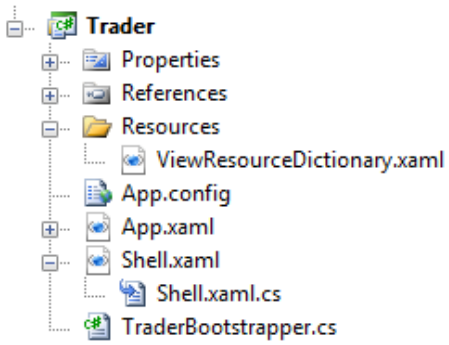
```
ConnectionEvent connectionEvent = eventAggregator.GetEvent<ConnectionEvent>();
SubscriptionToken connectionSubscriptionToken =
connectionEvent.Subscribe(ConnectionEventHandler, ThreadOption.UIThread, false);

private void ConnectionEventHandler(Enums.ConnectionEvent connectionEvent)
{
	UpdateInstrumentCommand.RaiseCanExecuteChanged();
}
```

Det andra villkoret är lite krångligare då det handlar om att ha koll på en annan kontroll på vår View. Lösningen på detta problem blev att ViewModel fick nedanstående Property som databinds till vyns ListBox's SelectedItem.

```
public IInstrument SelectedInstrument
{
	get { return selectedInstrument; }
	set
	{
		if (value != selectedInstrument)
		{
			selectedInstrument = value;
			SendPropertyChanged("SelectedInstrument");
			UpdateInstrumentCommand.RaiseCanExecuteChanged()
		}
	}
}
```

5.2. Trader

Trader är ramverkets startprojekt och dess enkla uppbyggnad ses nedan.

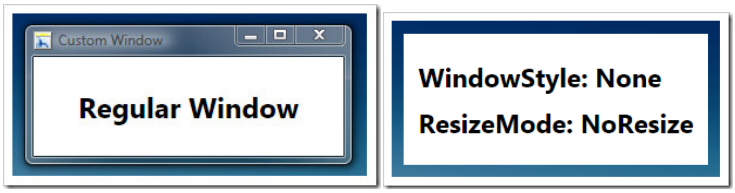


Alla WPF-applikationer startar genom att skapa ett Application-objekt. Application i detta fall är väldigt enkel, dess huvuduppgift är att skapa bootstrappern som i sin tur laddar och konfigurerar moduler, container och tjänster. Moduler laddas i två steg i CAG. Först upptäcker IModuleEnumerator alla moduler och skapar en collection av metadata om dessa. Sedan instanisierar IModuleLoader modulen och anropar dess Initialize-metod. Man kan välja mellan att ladda moduler statiskt eller dynamiskt. Vid statisk laddning har huvudprojektet en direkt referens till modulerna vilket innebär enklare användning och debugging. Vid dynamisk laddning upptäcks modulerna vid exekvering vilket innebär att moduler kan läggas till en applikation utan att den behöver modifieras eller få några nya referenser. Då det i nuläget inte finns något behov av fördelarna dynamisk laddning prioriteras enkelhet och statisk laddning av modulerna i Trader.

Slutligen skapar Application-objektet Shell som är programmets huvudfönster. Mer om detta nedan.

5.2.1. Shell

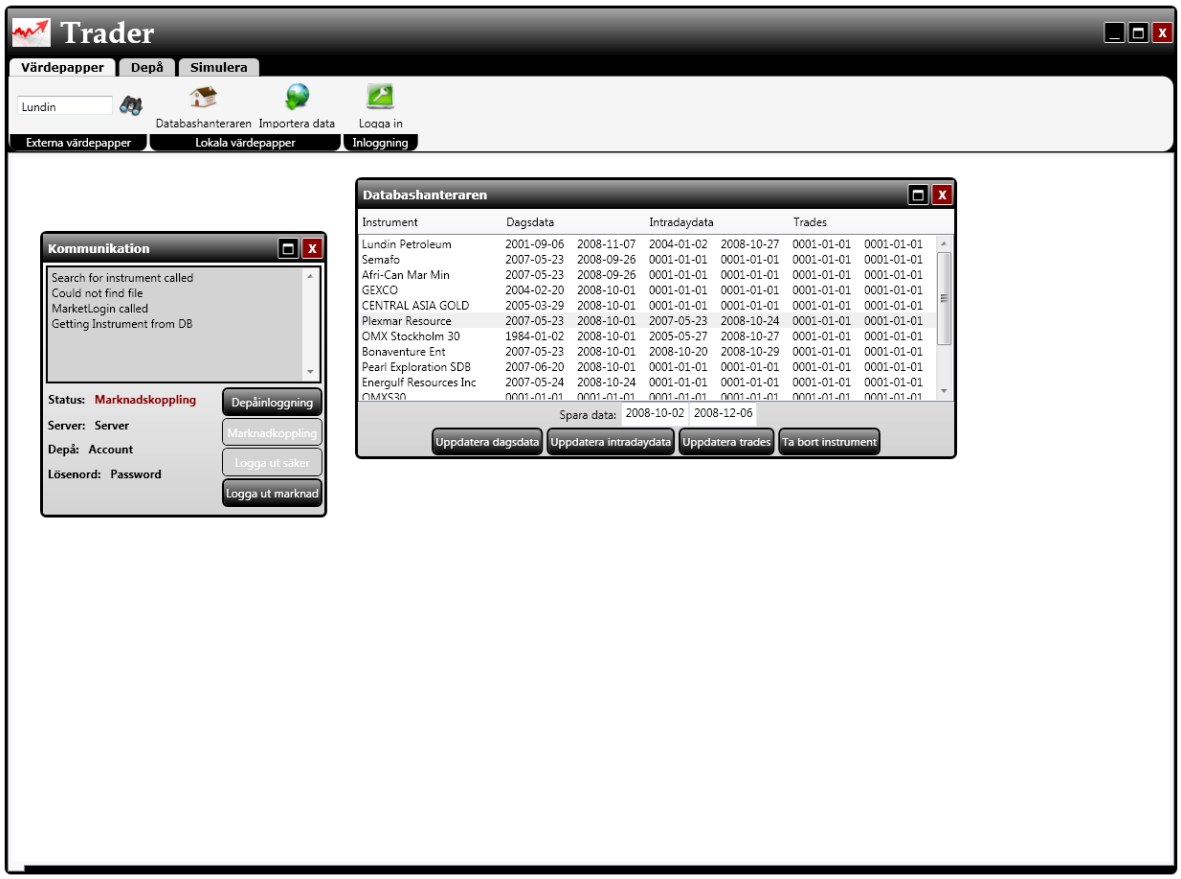
Det kanske mest intressanta i startprojektet är huvudprogramfönstret Shell. WPF stödjer tyvärr inte Multiple-document interface (MDI) på samma sätt som föregångaren Windows Forms gjorde. För en applikation av denna typ är support för ett toppfönster som kan innehålla multipla underfönster ett absolut krav. Detta innebär att stöd för MDI måste implementeras. Mer om hur detta är implementerat återfinns i avsnittet om Trader.CustomControls som är det projekt där dessa egenskapade specialkontroller hör hemma.



Figur 26: Fönster i WPF[24]

Ett huvudprogramfönster har alltså skapats. Genom att sätta WindowStyle = ”None” kan man som ses i figuren ovan ta bort standardutseendet för fönster i Windows. Istället har ett nytt unikt fönsterutseende definierats med hjälp av olika stilmallar. Genom att utnyttja WPF:s stöd för detta kan man ge de olika underfönsterna samma stil som huvudfönstret. Tyvärr tappas all fönsterfunktionalitet när man sätter WindowStyle = ”None” så funktioner som maximering, minimering, stängning, flyttning och storleksändring måste implementeras på egen hand. Det enda som inte är trivialt av denna funktionalitet är storleksändring. För att få detta att fungera så skapades en hjälpklass ResizingWindowAdorner i Trader.CustomControls.

[24] http://cloudstore.blogspot.com/2008/05/creating-custom-window-style.html



Figur 27: Applikationens huvudfönster

Huvudfönstret är uppbyggt av tre olika delar:

1. Ramen och headerområdet med logo, titel och knappar som ger fönstret de fönsterspecifika egenskaperna.

2. Ett övre menyområde i form av en instans av WPF:s klass TabControl som givits ett nytt applikationsanpassat utseende genom att byta ut kontrollens ursprungliga mall.

3. Själva huvudytan som har en ytterligare en upp och nedvänd TabControl i bakgrunden. Detta för att funktionalitet med olika arbetsytor senare ska kunna implementeras om behovet skulle uppstå i framtiden. I denna TabControl har den egenskapade kontrollen ChildWindowContainer placerats. Mer om denna i avsnittet om Trader.CustomControls.

Utan att gå in på djupet kan man säga att fönsterkontrollen hämtar att sitt utseende ”MainWindowStyle” från en stilmall i som finns tillsammans med massa andra fönsterrelaterade stilar i en så kallad ResourceDictionary ”WindowStyleDictionary” i projektet Trader.CustomControls. Stilmallens olika element har givits namn så man lätt i fönsterklassen kan hämta elementen och binda dem till ett beteende. Man eftersträvar alltså att koppla lös kontrollens utseende från dess beteende i allra möjligaste mån. På så sätt går samma utseende att använda för flera olika fönster.

Ett exempel är maximeringsknappen uppe i högra hörnet. Den finns definerad i stilmallen som:

```
<Button x:Name="PART_MaximizeButton" VerticalAlignment="Center"
Content="{StaticResource MaximizeButtonPath}" Grid.Column="3"
Style="{DynamicResource HeaderButton}" Padding="4" />
```

I fönsterklassen binds sedan knappen till sitt beteende genom:

```
Button maximizeButton = (Button)this.GetTemplateChild("PART_MaximizeButton");
if (maximizeButton != null)
{
     maximizeButton.Click += maximize_ButtonClick;
}
```

5.3. Trader.CustomControls

Detta projekt innehåller egenutvecklade kontroller.

5.3.1. WindowControls

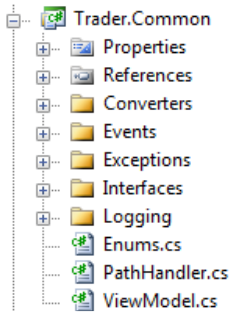
WPF stöder som sagt inte inte Multiple-document interface (MDI) på samma sätt som föregångaren Windows Forms gjorde. MDI innebär att ett huvudfönster kan innehålla flera olika underfönster. Denna uppbyggnad är ett absolut måste för en applikation som denna och därför blev detta ett problem som var tvunget att lösas. Efter att ha läst och fått inspiration från diverse bloggar på nätet så togs valet att implementera MDI-stöd i form av två egna kontroller.

Först och främst skapades kontrollen ChildWindowContainer. Denna kontroll kan innehålla multipla instanser av klassen ChildWindow som är den andra egenskapade kontrollen. ChildWindowContainer ärver av WPF:s typ UserControl och har en lista Children över alla ChildWindows. För att visa ChildWindows används en ItemsControl i form av klassen DragCanvas vilket tillåter dem att förflyttas inom föräldrafönstrets gränser. DragCanvas ärver av Canvas och hanterar musklick genom att kolla om det var ett UIElement som klickades på. Om det var det och detta UIElement är draggable söks det visuella trädet igenom tills ett ChildWindow hittas varpå användaren tillåts flytta detta fönster.

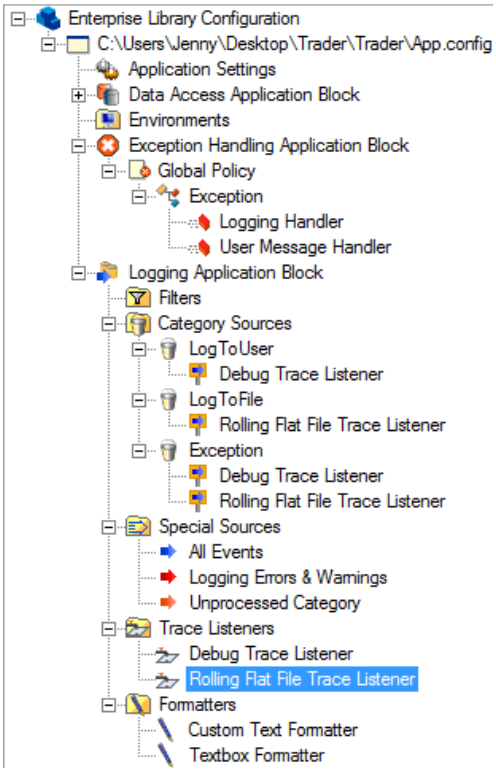
Även ChildWindow ärver av WPF-typen UserControl som har stylats för att se ut som ett fönster. Här vinner vi fördelen med att ha implementerat vår helt egna stil för huvudprogramfönstret då vi kan nu kan ge underfönstrena samma utseende. Ett ChildWindow kan maximeras, stängas, ändra storlek och flyttas runt inom en ChildWindowContainer. Att ändra storlek sker här på samma sätt som när huvudfönstret ändrar storlek med hjälp av klassen ResizingWindowAdorner. ResizingWindowAdorner ärver från WPF:s klass Adorner som ritar ut en osynlig ram runt objektet som ska kunna ändra storlek. Ramen har ett antal olika punkter som med hjälp av adornern känner av musrörelser och ändrar musmarkören beroende på vilken punkt det är frågan om. Om musen trycks ned över en punkt och sedan flyttas kommer adornern göra ett antal beräkningar för att bestämma hur objektet ska ändra storlek.

5.4. Trader.Common

Trader.Common är ett klassbibliotek som innehåller diverse hjälpklasser.



Då alla projekt i applikationen tillåts referera till Trader.Common är detta ett bra ställe att lägga generella klasser i. Exempelvis ligger alla interface för våra tjänster och datamodeller i mappen interfaces. På så sätt kan de nås från hela applikationen utan att aktuellt projekt behöver referera till exempelvis Trader.Modules.DataAccess-projektet där många service-klasser ligger.



Konfigurationen för loggning och felhantering syns i ovan figur. Hur dessa två viktiga ramverksområden implementerats diskuteras närmare nedan.

5.4.1. Loggning

Loggning är en livsnödvändig funktion för ett robust ramverk. Valet togs att använda Enterprise Library för både loggning och felhantering då det då det erbjuder stor valfrihet i var, hur, vad som ska loggas/felhanteras. Dessutom kan hanteringen vid senare skede ändras direkt i konfigurationen utan att någon kod behöver modifieras.

Som ses i konfigurationen loggas tre olika kategorier:

- *LogToUser* som lyssnas på av egenskapad Trace Listener ”Debug Trace Listener” som publicerar eventet DebugMessageEvent. Kommunikationsfönstret prenumererar sedan på dessa händelser och visar dem för användaren i en loggningsruta. Formatteraren är här satt till Textbox Formatter som endast skriver ut meddelande.

- *LogToFile* som lyssnas på av Entlibs Trace Listener ”Rolling Flat File Trace Listener” som skriver till en textfil som nyskapas i AppData-katalogen dagligen. Formatteraren är här satt till ”Custom Text Formatter” som har en mall som är satt till att skriva ut tid, kategori, tråd, metod, eventuella parametrar samt meddelande.

- *Exception* som loggar de exceptions som fångats av Global Policy och hanteras av Logging Handler. Dessa loggas både i loggningsrutan och till fil.

Loggningen sker i möjligaste mån i interfaceklasserna med attribut genom policy injection på följande form:

```
[LogCallHandler(Categories = new string[] { "LogToFile", “LogToUser” },
LogAfterCall = false, LogBeforeCall = true, BeforeMessage = "MarketLogin called",
IncludeParameters = true)]
void MarketLogin(string serverName);
```

Anropas denna metod skrivs följande post i logfilen:

Timestamp: 2008-09-28 05:00:01 Category: Debug Thread Id: 3960
Trader.Modules.DataAccess.Services.ConnectionService.MarketLogin(serverName - *servername*)
Message: MarketLogin called

Ibland vill man dock även ha möjlighet att logga inne i metoder. Vid dessa tillfällen används den egna klassen CustomLogger med metoden Log. Den är kopplad till Enterprise Librarys Logger-klass så varje loggningmeddelande kommer att behandlas på samma sätt som om det hade loggats genom Policy Injection som ovan.

5.4.2. Felhantering

Att bestämma en lämplig strategi för felhantering är ingen absolut vetenskap. Vad som kan anses vara en lämplig strategi är helt beroende av vilket typ av system det är frågan om och vilka som ska använda det. Då detta system ska användas av avancerade användare som kunskapsmässigt går att likställa med utvecklaren har följande riktlinjer satts upp vad gäller felhantering:

- Inga exceptions fångas i onödan. En metod bör endast fånga exceptions då den vill samla data för loggning, lägga till relevant information till exceptions, exekvera uppstädnings- eller återhämtningskod.

- Även om de flesta exceptions inte ska hanteras bör de fortfarande loggas. Många utvecklare fångar och loggar exceptions precis där de sker vilket innebär massa onödig, repetitiv kod. Det är bättre att låta exceptions bubbla upp genom applikationen och logga på ett centraliserat ställe vilket i denna applikation blir i Application-objektet.

- Även om exceptions inte kan hanteras bör man tänka på att städa upp. Målet är att använda try-finally eller ännu hellre using överallt där klasser som implementerar IDisposible används.

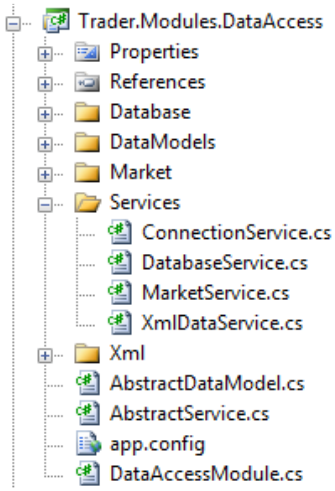
- Det råder delade meningar om när man bör kasta exceptions. Den ena skolan förordar endast när det sker något verkligt oväntat medan andra menar att exceptions ska kastas vid varje situation då programmets förväntade beteende inte kan exekveras, alltså då en metod inte kan göra det den ska göra. I denna applikation väljer vi ett mellanting av de två skolorna.

Vi använder oss av Microsofts Exception Handling Block. Exceptions kastas automatiskt vid verkligt oväntade händelser men även av oss själva då en metod av vikt inte lyckas utföra det den borde. Vid eget kastande används gärna applikationsspecifika exceptions då det innebär större valfrihet när de ska tas om hand. För att slippa massa onödig kod låter vi till en början de flesta exceptions bubbla upp ända till Application-objektet där dom fångas. Det finns definierat en exception policy med namnet ”Global Policy” som har två exceptions handlers. UserMessageExceptionHandler som med hjälp av UserMessageTextExceptionFormatter presenterar en dialogruta för användaren med

information om felet. Användaren får sedan välja huruvida man vill fortsätta köra programmet eller inte. Den andra exceptions handlern är en vanlig Logging Handler som loggar alla exceptions till loggfilen.

5.5. Trader.Modules.DataAccess

Denna modul saknar eget användargränssnitt men förser andra med data. Dessa andra moduler kommunicerar naturligt med de olika serviceklasserna som ses nedan.



Tjänsterna stoppas tillsammans med de interface de implementerar in i UnitityContainern och nås från övriga applikationen enligt nedanstående exempel:

```
container.RegisterType<IMarketService, MarketService>(new
ContainerControlledLifetimeManager());
container.Resolve<IMarketService>().SearchForInstrument(SearchString);
```

Alla interface som används på applikationsnivå finns I projektet Trader.Common som samtliga projekt refererar till. Det finns i nuläget egentligen tre ställen applikationen kan begära data från:

- DatabaseService som använder LINQ to Sql för att kommunicera med den lokala databasen TraderDB. Denna databas är uppbyggd i Sql Server 2008 Express som är en en gratis version Microsoft erbjuder tillsammans med Visual Studio.

- XmlDataService som använder Linq to Xml för att läsa konfiguration och annat från xml-filer som finns lagrade i AppData-katalogen.

- ConnectionService och MarketService som kommunicerar med Avanza via AvanzaFix.

Tidskrävande tjänster som exempelvis hämtning av alla instrument från Avanzas databas abstraheras bort från UI-tråden till bakgrundstrådar för att få ett responsivt användargränsnitt.

5.6. Trader.Modules.Connection

Syftet med denna modul är att erbjuda inloggningsfunktionalitet samt synliggöra kommunikationen med marknaden och programmets aktivitet via ett loggningsfönster. Nedan ses kommunikationsfönstrets utseende.

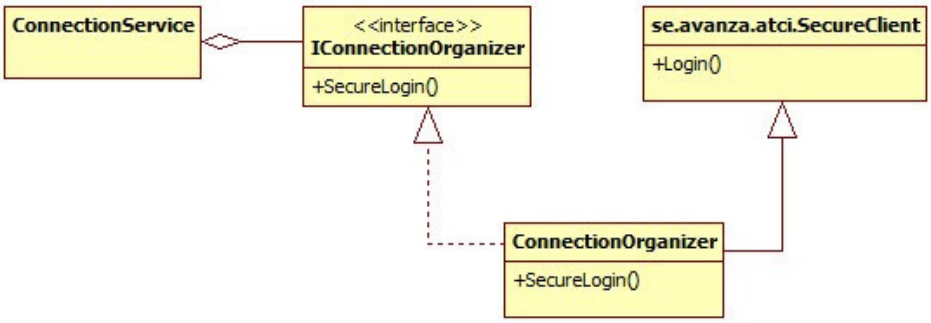
Figur 28: Applikationens inloggningsruta

ConnectionView pekar på ConnectionViewModel som i sin tur refererar till både tjänsten ConnectionService och DataModel-objektet Connection. Knapparna i connectionView är kopplade till sin logik via kommandon i ConnectionViewModel.

För att få tillgång till Avanzas klient-API som vi använder för att koppla upp oss mot marknaden behöver man först logga in hos Avanza. AvanzaFix använder sig som tidigare beskrivits av både krypterade och okrypterade kanaler för informationsflödet. Den krypterade kanalen är avsedd för att användas till känslig information såsom exempelvis lösenord och depåinformation medan den okrypterade kanalen är avsedd för instrument, kursinformation och annan okänslig information.

Vid inloggning kan man använda sig av ett antal olika förfaranden beroende på om man vill logga in säkert eller inte. För mer information om detta se avsnittet om AvanazaFix. Kortfattat kan sägas att man vid säker inloggning använder sig av objektet SecureClient medan man vid marknadskoppling använder UnsecureClient och någonting som kallas Token. Token lagras i databasen på serversidan och är giltiga i olika tidsperioder. Går något fel vid inloggningen kastas ett LoginException och i detta får man objektet RemoteInfo som parameter. RemoteInfo innehåller massa nyttig information såsom inloggningstid, orsak till felet med mera.

ConnectionViewModel använder sig av ConnectionService för att kommunicera med Avanza. ConnectionService använder i sin tur ConnectionOrganizer som innehåller de Avanzaspecifika Secure- och UnsecureClient. Genom detta upplägg och strävandet av att använda sig av interface överallt skulle man vid eventuellt byte av mäklare i framtiden lätt kunna byta ut kopplingen mot Avanza genom en ny implementation av IConnectionOrganizer. Detta är ett exempel på designmönstret adapter som beskrevs i teoridelen.



Figur 29: Designmönstret Adapter implementerat

Till ConnectionOrganizer binds en klass MarketListener som implementerar Avanzas interface ConnectionListener och tar emot uppkopplingsstatus från Avanza:

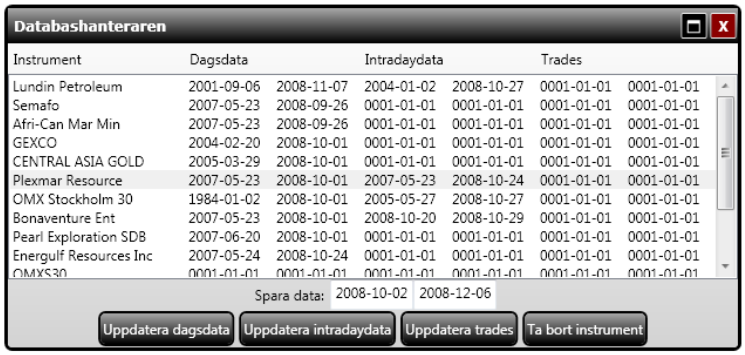
```
public void onConnectionClose(se.avanza.atci.IClient client, int code, string
reason)
{
eventAggregator.GetEvent<ConnectionEvent>().Publish(Enums.ConnectionEvent.SecureCon
nectionClosed);
}

public void onConnectionOpen(se.avanza.atci.IClient client)
{
eventAggregator.GetEvent<ConnectionEvent>().Publish(Enums.ConnectionEvent.SecureCon
nectionOpened);
}
```

MarketListener publicerar sedan denna status I form av en ConnectionEvent. ConnectionViewModel lyssnar på nyss nämnda ConnectionEvent men även på DebugMessageEvent som behövs för att populera loggningsrutan.

5.7. Trader.Modules.Instrument

Denna modul innehåller två olika Views och ViewModels som har med instrumenthantering att göra.

I huvudfönstret finns ett sökfält under fliken värdepapper. Då en användare fyller i fältet och trycker på sökknappen aktiveras SearchInstrumentViewModel som använder MarketService för att genomföra sökningen. MarketService hämtar alla instrument i Avanzas databas undersöker om någon matchar söksträngen. Om så är fallet publiceras en händelse FoundInstrumentEvent som SearchInstrumentViewModel lyssnar på och visar i sökresultatrutan SearchInstrumentView.

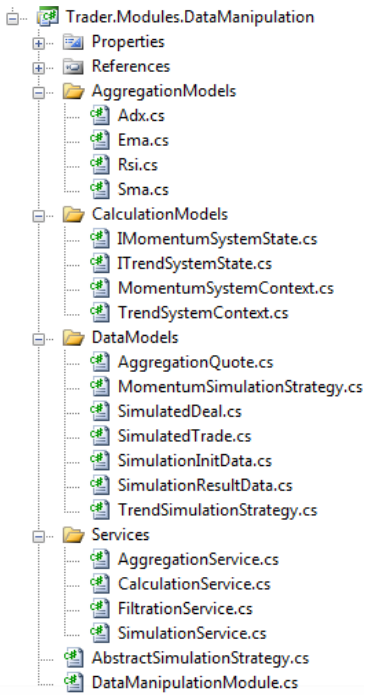


Figur 30: Databashanteraren

Databashanteraren ingår även i denna modul i form av InstrumentsViewModel och InstrumentsView som illustrerar den lokala databasens innehåll genom att interagera med DatabaseService. Genom databashanteraren kan en användare ta bort instrument samt uppdatera kursdata. Då någon av kursuppdateringsknapparna trycks in använder InstrumentViewModel MarketService för att fråga Avanza om kursdata för ett visst instrument mellan två olika datum. Erhållen kursinformation sparas sedan ned i den lokala databasen varpå ett DBInstrumentsChangedEvent skickas ut så InstrumentsView kan uppdatera sig och visa aktuellt databastillstånd.

5.8. Trader.Modules.DataManipulation

Denna modul saknar på samma sätt som Trader.Modules.DataAccess eget användargränssnitt men erbjuder övriga delar av applikationen olika tjänster för att manipulera data. Nästan all funktionalitet som används för att köra olika tradingssytem finns här.



Detta projekts tjänster används i nuläget endast för att simulera olika tradingsystem men skulle i framtiden även exempelvis kunna nyttjas för att filtrera och aggregera kursdata som ska sparas i databasen, för att visa teknisk analys i realtidsdiagram eller vid beräkning av köp-/säljsignaler för värdepappershandel i skarp miljö.

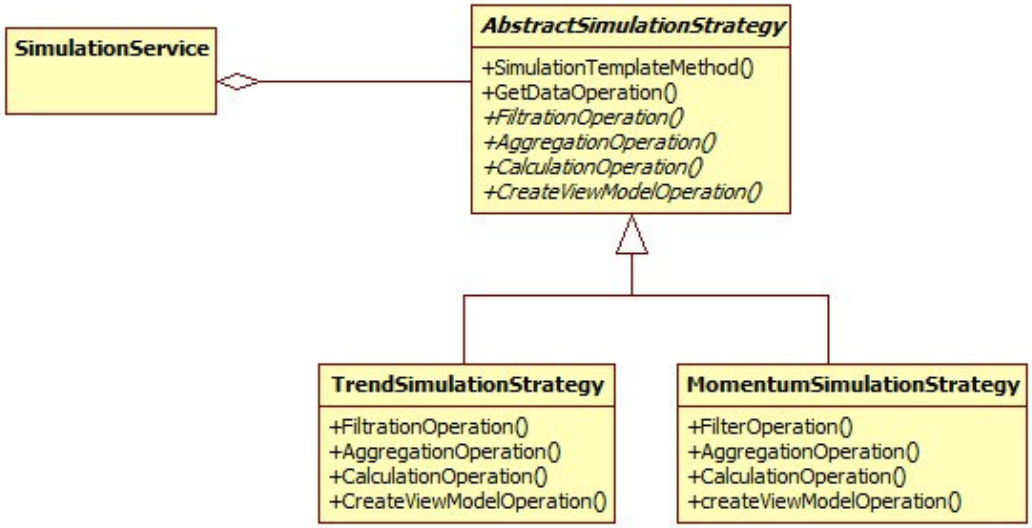
- *SimulationService* erbjuder ett publikt gränsnitt för att kunna simulera olika tradingsystem. Den skapar och initierar objekten TrendSimulationStrategy och MomentumSimulationStrategy som beskrivs närmare nedan.

- *FiltrationService* innehåller metoder för filtrering av data.

- *AggregationService* innehåller metoder för aggregering av olika dataserier. Den använder olika AggregationModels som här i ett första skede utgörs av fyra olika TA-indikatorer, rsi, adx, sma och ema som alla introducerades i avsnittet om teknisk analys.

- *CalculationService* innehåller metoder för att beräkna köp och säljsignaler för olika tradingsystem. Den skapar och använder sig av olika CalculationModels som beskrivs närmare nedan.

I mappen med DataModels är som sagt klasserna med namn som slutar på SimulationStrategy extra intressanta. Som namnen vittnar om motsvarar de simuleringar av olika tradingsystem som är uppbyggda enligt designmönstret Strategy. I ett första försök vid byggandet av simuleringfunktionaliteten återfanns den mesta koden direkt i SimulationService-klassen vilket snabbt ledde till en stor oflexibel klass med alltför många ansvar. Vid refactoreringen som följde implementerades båda designmönstren Strategy och Template Method.



Figur 31: Designmönstret Strategy implementerat

Genom att använda mönstret Strategy frikopplas objektet från dess beteende genom att algoritmerna flyttas ut från värdklassen och definieras i varsin egen klass med ett gemensamt interface. För att ha möjlighet att låta de olika simuleringsalgoritmerna dela visst beteende byttes interfacet ut mot en abstrakt klass AbstractSimulationStrategy. Som ses i figuren ovan har denna klass en huvudmetod som heter SimulationTemplateMethod. Denna metod är uppbyggd enligt designmönstret Template Method som låter algoritmen den illustrerar variera i vissa steg genom att flytta ut dessa specifika steg till subklasser. Strukturen av algoritmen förändras alltså inte men små väldefinierade delar av den, operationer, behandlas någon annanstans.

```
protected void SimulationTemplateMethod(object data)
{
      InitializeSimulationOperation(data as SimulationInitData);
      GetDataOperation();
      FiltrationOperation();
      AggregationOperation();
      CalculationOperation();

      Dispatcher.BeginInvoke(DispatcherPriority.ApplicationIdle, new
ThreadStart(delegate
      {
            CreateViewModelOperation();
      }));
}

protected void GetDataOperation(){}
```

```
protected abstract void InitializeSimulationOperation(SimulationInitData initData);
protected abstract void FiltrationOperation();
protected abstract void AggregationOperation();
protected abstract void CalculationOperation();
protected abstract void CreateViewModelOperation();
```

Ovanstående kod är tagen från klassen AbstractSimulationStrategy som ni ser får TrendSimulationStrategy samt MomentumSimulationStrategy välja huruvida de vill överlagra metoden GetDataOperation medan de måste implementera resterande abstrakta metoder. Om vi exempelvis kollar på AggregationOperation så använder TrendSimulationStrategy följande metoder hos AggregationService:

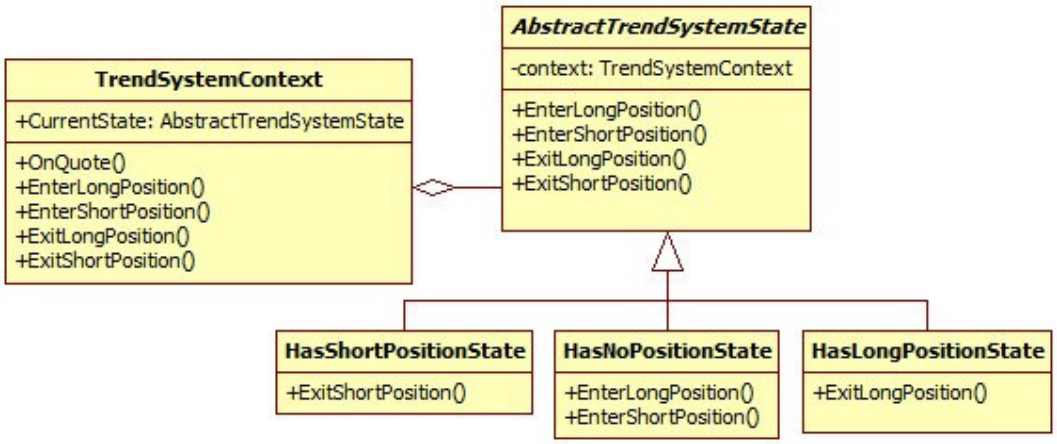
```
protected override void AggregationOperation()
        {
            AdxSerie =
container.Resolve<IAggregationService>().AggregateAdxSerie(quoteSerie,
AdxParameter);
            SmaSerie =
container.Resolve<IAggregationService>().AggregateSmaSerie(quoteSerie,
SmaParameter);
            EmaSerie =
container.Resolve<IAggregationService>().AggregateEmaSerie(quoteSerie,
EmaParameter);
        }
```

Medan MomentumSimulationStrategy använder aggregateMomentumSerie:

```
protected override void AggregationOperation()
{
            AggregateSerie =
container.Resolve<IAggregationService>().AggregateRsiSerie(QuoteSerie, parameter);
}
```

Denna uppbyggnad gör det lätt att lägga till simuleringar för fler tradingsystem utan att behöva modifiera någon existerande kod. Dessutom kan man lätt återanvända valfria servicemetoder vilket minimerar mängden duplicerad kod.

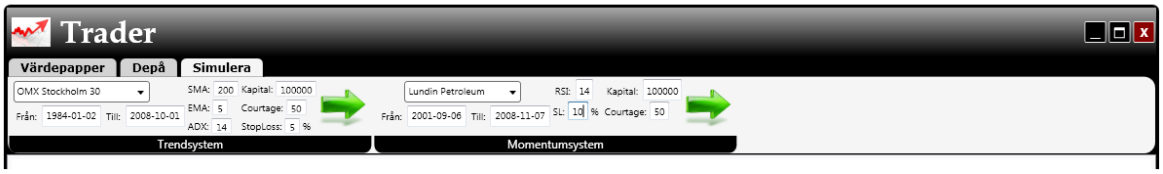
En annan serviceklass som refactorerades rejält efter ett första misslyckat försök är CalculationService. Även denna från början var en stor, oflexibel klass bestående av mängder av olika if-satser för olika kursscenarion. Problemet uppmärksammades snabbt då det var nästintill omöjligt att uttöka de olika tradingsystemen med ytterligare villkor för entries och exits. Designmönstret State användes här för att få en mer flexibel design. I fallet med det trendföljande systemet skapar CalculationService först objektet TrendSystemContext och skickar sedan ett kursobjekt åt gången till kontextobjektet som hanterar kursen och beräknar eventuella signaler beroende på i vilket state det befinner sig. Aktuellt State har tillgång till kontextobjektet och kommer om det är lämpligt skapa objekten trade respektive deal som läggs till simularingens resultatobjekt. Till sist sätter aktuellt State eventuellt kontextobjektets CurrentState till ett nytt State.



Figur 32: Designmönstret State implementerat

5.9. Trader.Modules.Simulation

Den sista deloperationen CreateViewModelOperation i AbstractSimulationStrategy:s Template Method går ut på att man vill illustrera simuleringensresultat för användaren som begärt den genom huvudfönstrets flik ”Simulera” som ses nedan. De Views respektive ViewModels som sköter detta återfinns i projektet Trader.Modules.Simulation.



Figur 33: Huvudfönstrets flik "Simulera"

Det finns i en första officiell version av plattformen endast stöd för simulering av två tradingsystem, Trendsystem och Momentumsystem. Dessa valdes för att illustrera två grundläggande begrepp inom den tekniska analysen som kortfattat beskrivs i förstudien, trend och momentum.

5.9.1. TrendSimulation

Det trendföljande tradingsystemet är baserat på två olika typer av glidande medelvärden och indikatorn adx. Systemet går kortfattat ut på att man vill försöka följa med marknaden i betydande trender. För att upptäcka en ny trend använder man ett kort exponentiellt glidande medelvärde 5 EMA och ett långt enkelt glidande medelvärde 200 SMA. Signaler genereras när dessa korsar varandra. För att försöka undvika falska signaler används 14-dagars ADX som ett slags filter.

Tradingsystemet vi använder är definierat enligt:

- Entry –Signal för att gå in i marknaden ges vid två tillfällen:

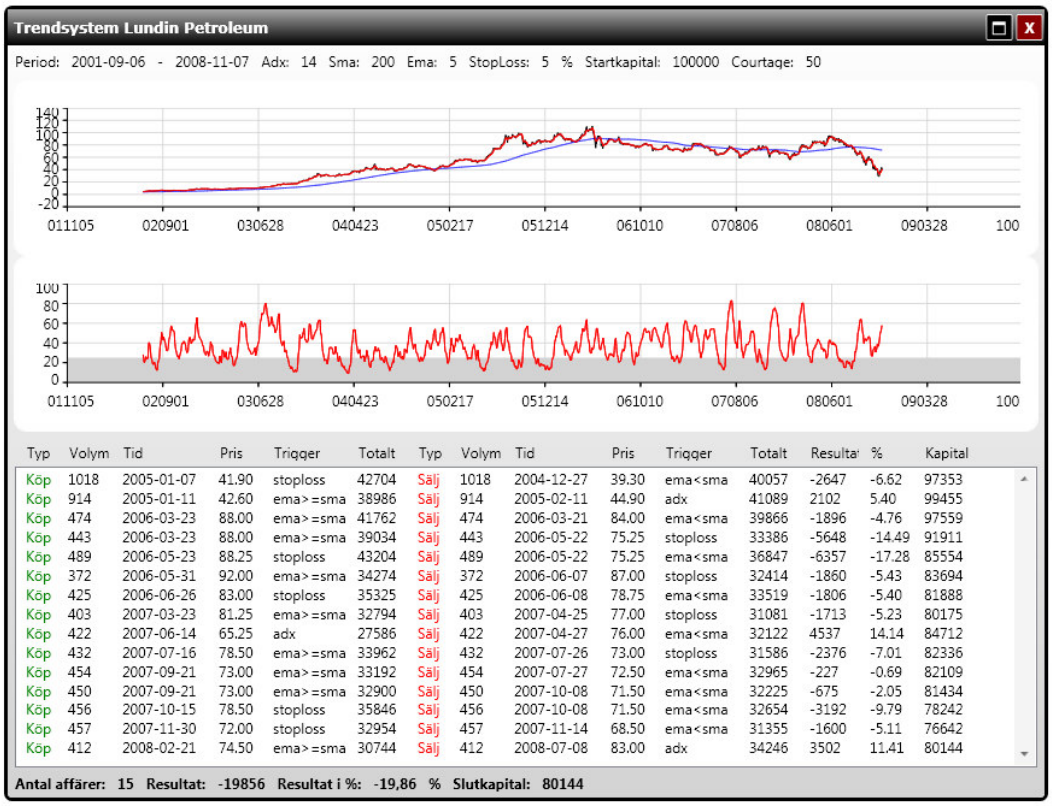
    1. Gå lång (köp) om adx är större eller lika med 25 och om 5 EMA korsar 200 SMA underifrån.
    2. Gå kort (blanka) om adx är större eller lika med 25 och 5 EMA korsar 200 SMA ovanifrån.

- Exit - När vi väl gått in i marknaden, måste vi någon gång också gå ur den. I detta tradingsystem avslutas aktell position när:

1. Om stoploss-nivån nås. Stop loss sätts som 5 % av kursen.
2. Om 5 EMA korsar 200 SMA i motsatt riktning från när position togs.
3. Om ADX faller under 20.

- Filter – Man kan här säga att ADX-indikatorn verkar som ett filter. Systemet används bara då kursen trendar tydligt.

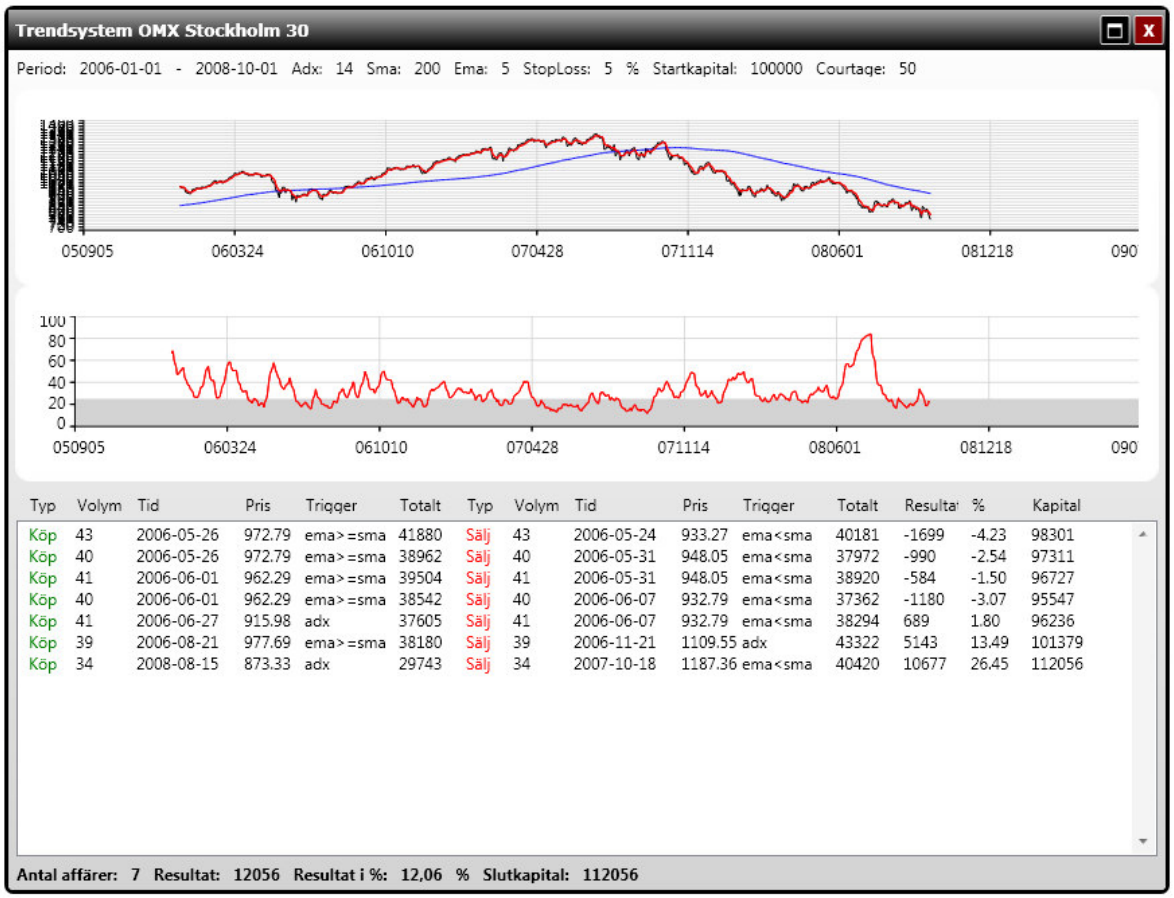
- Moneymanagement – Vi vill riskera högst 2% av vår tradingdepå i en affär och handlad volym sätts därefter.



Figur 34: Körning nr 1 av trendsystemet

Ovanstående bild visar en körning av trendsystemet baserat på dagsdata för oljebolaget Lundin Petroleum under perioden 2001-09-06 till 2008-11-07. Högst upp i resultatrutan ses vad användaren valde för indata till tradingsystemet. Startkapital har satts till 100 000 kr och stoploss till 5 % enligt systemets rekommendation. Översta diagrammet illustrerar kursen(svart linje), långt medelvärde SMA(blå linje) och EMA(röd linje). Undre diagrammet visar ADX-indikatorn med grått område under 25. Entrysignaler tas endast om ADX befinner sig övanför detta gråa område. Nedanför diagrammen redovisas alla affärer systemet gjort. I kommentaren under kolumnen trigger förklaras varför en entry eller exit gjorts. Moneymanagement-metoden går ju ut på att mer än 2% av portföljens totala värde aldrig ska riskeras i en affär vilket innebär att programmet tar hänsyn till signalpris samt stoploss-nivå för att beräkna lämplig volym. Som man kan se har detta system presterat riktigt dåligt. Av startkapitalet 100 000 kr finns vid slutet endast 80144 kr kvar. Här är ett typexempel på att ett trendsystem inte fungerar bra i en konsoliderande marknad utan tydliga trender. Resultat försämras ytterligare av att vi jobbar med dagsdata vilket innebär att vår stoploss inte löser ut i tid vid 5%

förlust. Förhoppningsvis blir man utstoppad tidigare om man sätter automatiska stoploss-nivåer som får verka i realtid.



Figur 35: Körning nr 2 av trendsystemet

Den andra testkörningen av trendsystemet för indexet OMXS30 under perioden 2006-01-01 till 2008-10-01 presterar bätte. Som man kan se i det översta kursdiagrammet trendar OMXS30 mycket tydligare under denna period än Lundin Petroleum gjorde ovan. Ett resultat på 12% på två år är kanske inget att jubla över men man brukar ofta jämföra tradingsystem mot att köpa ett värdepappaer och behålla det under hela perioden. Hade man köpt OMXS30 2006-01-02 för 963 och sedan sålt den 2008-10-01 för 769 hade man gjort en förlust på över 20%.

5.9.2. MomentumSimulation

Vårt momentumsystem är ett enkelt system som baserar sig på Relative Strength Index (RSI) som sägs vara den flitigast använda tekniska indikatorn i mäklarrummen. Systemet går ut på att man vill försöka utnyttja överreaktioner på marknaden, man vill köpa när ett värdepapper är översålt och sälja (blanka) då ett värdepapper är överköpt.

Tradingsystemet vi använder är definierat enligt:

- Entry –Signal för att gå in i marknaden ges vid två tillfällen:

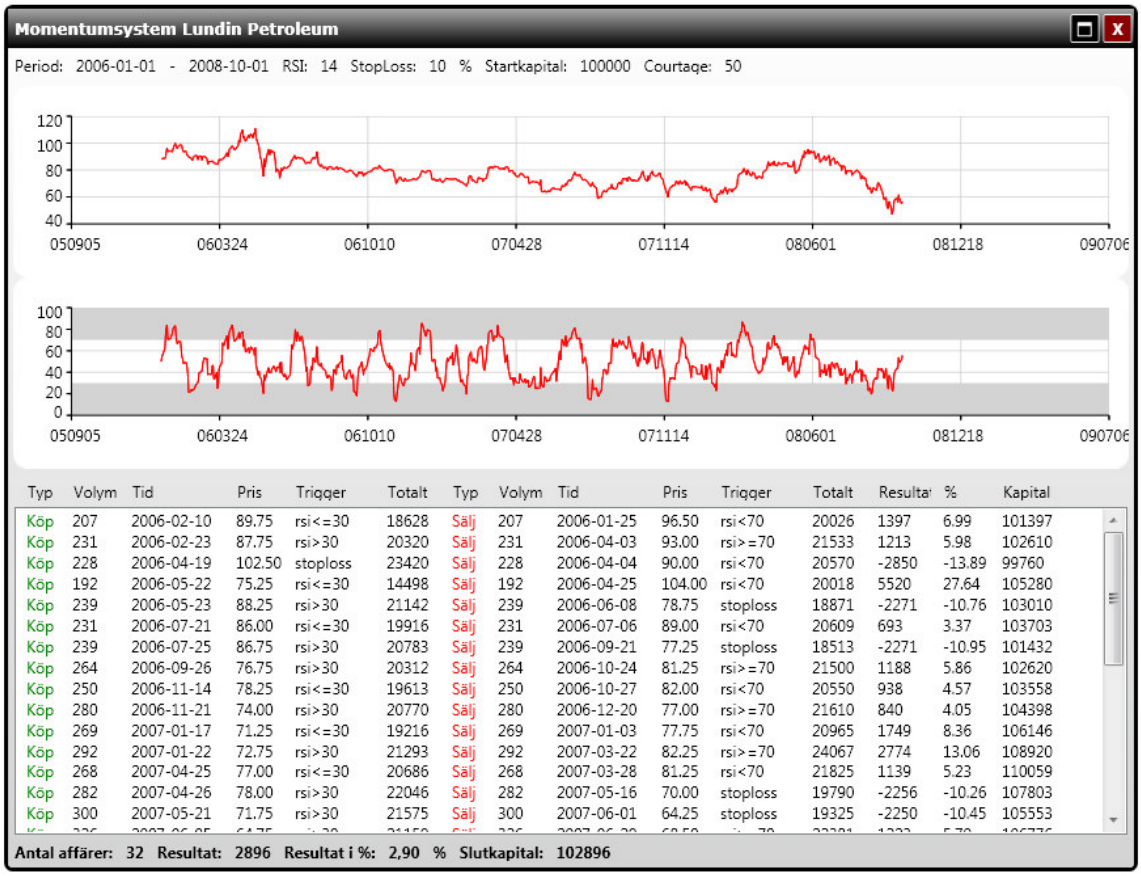
    1. Gå lång (köp) om RSI korsar 30 underifrån.
    2. Gå kort (blanka) om RSI korsar 70 ovanifrån.

- Exit - När vi väl gått in i marknaden, måste vi någon gång också gå ur den. I detta tradingsystem avslutas aktell position när:

    1. Om stoploss-nivån nås. Stop loss sätts som ett procenttal av kursen
    2. Vid lång position – om RSI når 70 underifrån.
    3. Vid kort position – om RSI når 30 ovanifrån.

- Filter – Vi använder inget filter här i ett första skede men exempelvis skulle vi kunna sätta ADX-inikatorn som filter även här om modellen skulle generera för många falska signaler.

- Moneymanagement – Även här vill vi riskera högst 2% av vår tradingdepå i en affär och handlad volym sätts därefter.



Typ	Volym	Tid	Pris	Triqqer	Totalt	Typ	Volym	Tid	Pris	Triqqer	Totalt	Resultat	%	Kapital
Köp	207	2006-02-10	89.75	rsi<=30	18628	Sälj	207	2006-01-25	96.50	rsi<70	20026	1397	6.99	101397
Köp	231	2006-02-23	87.75	rsi>30	20320	Sälj	231	2006-04-03	93.00	rsi>=70	21533	1213	5.98	102610
Köp	228	2006-04-19	102.50	stoploss	23420	Sälj	228	2006-04-04	90.00	rsi<70	20570	-2850	-13.89	99760
Köp	192	2006-05-22	75.25	rsi<=30	14498	Sälj	192	2006-04-25	104.00	rsi<70	20018	5520	27.64	105280
Köp	239	2006-05-23	88.25	rsi>30	21142	Sälj	239	2006-06-08	78.75	stoploss	18871	-2271	-10.76	103010
Köp	231	2006-07-21	86.00	rsi<=30	19916	Sälj	231	2006-07-06	89.00	rsi<70	20609	693	3.37	103703
Köp	239	2006-07-25	86.75	rsi>30	20783	Sälj	239	2006-09-21	77.25	stoploss	18513	-2271	-10.95	101432
Köp	264	2006-09-26	76.75	rsi>30	20312	Sälj	264	2006-10-24	81.25	rsi>=70	21500	1188	5.86	102620
Köp	250	2006-11-14	78.25	rsi<=30	19613	Sälj	250	2006-10-27	82.00	rsi<70	20550	938	4.57	103558
Köp	280	2006-11-21	74.00	rsi>30	20770	Sälj	280	2006-12-20	77.00	rsi>=70	21610	840	4.05	104398
Köp	269	2007-01-17	71.25	rsi<=30	19216	Sälj	269	2007-01-03	77.75	rsi<70	20965	1749	8.36	106146
Köp	292	2007-01-22	72.75	rsi>30	21293	Sälj	292	2007-03-22	82.25	rsi>=70	24067	2774	13.06	108920
Köp	268	2007-04-25	77.00	rsi<=30	20686	Sälj	268	2007-03-28	81.25	rsi<70	21825	1139	5.23	110059
Köp	282	2007-04-26	78.00	rsi>30	22046	Sälj	282	2007-05-16	70.00	stoploss	19790	-2256	-10.26	107803
Köp	300	2007-05-21	71.75	rsi>30	21575	Sälj	300	2007-06-01	64.25	stoploss	19325	-2250	-10.45	105553

Figur 36: Körning nr 1 av momentumsystemet

Ovanstående bild visar en körning av momentumsystemet. Värdepappret är återigen Lundin Petroleum och perioden är 2006-01-01 till 2008-10-01. Det översta diagrammet visar kursutvecklingen medan det nedre visar indikatorn RSI. Områdena under 30 respektive över 70 är gråa och illustrerar vart entries och exits triggas. Man kan se att denna modell hanterar det icke trendande marknadsläget betydligt bättre än trendsystemet. Resultatet plus 2,90 % kan ställas i relation till att köpa Lundin Petroleum 2006-01-02 för 85 och sälja 2008-10-01 för 57,50, en förlust på 32%.

6. Resultat

Systemet Trader består av ett antal olika moduler. Man kan koppla upp sig mot sin mäklare för att hämta kursdata. Denna data kan sedan användas till att simulera hur olika tradingsystem skulle ha presterat i olika tidsintervall.

6.1. Diskussion

I inledningen sattes examensarbetets syfte till att utveckla en plattform för simulering och utvärdering av olika tradingsystem. Vidare var syftet att lära sig mer om den tekniska gången vid utvecklandet av ett lite större program och att skaffa sig erfarenhet av programmering av Windows-applikationer i .NET-miljö då många företag idag använder sig av denna teknik.

Dessa syften ledde i förstudien till följande uppsatta krav. Det viktigaste kravet var utbyggbarhet för att man vid senare tillfälle ska kunna uttöka systemet vartefter behov av mer funktionalitet uppstår i en snabbt föränderlig finansiell värld. För att ha en chans att uppnå detta behövdes kunskap om ett antal designprinciper och designmönster inhämtas, dokumenteras i teoridelen och sedan tillämpas vid implementationen. Vidare ställdes kravet att den allra senaste tekniken från Microsoft ska studeras och användas. Nedan diskussion belyser hur väl syfte och krav uppfyllts.

Ett projekt av denna omfattning kräver god disciplin och bra planering. Båda dessa punkter har i viss mån saknats främst beroende på att examensarbetet utförts som bisyssla vid sidan av ordinarie heltidstjänst. De flesta moment har tagit mycket längre tid än som beräknades från början.

Först och främst innebar det en hög tröskel att ta fram den grundläggande designen. Det första kravet sattes till att bygga en flexibel plattform som är utbyggbar i framtiden. Då tidigare erfarenhet av att bygga ett stort, komplext system saknades så krävdes ett antal försök och flera misslyckande innan det blev tillräckligt bra för att uppfylla uppställt krav.

Resultatet blev en modulbaserad applikation med tydliga riktlinjer för hur moduler ska laddas, skapas, initieras samt hur de kommunicerar med varandra utan överflödiga beroenden. Beslutet att använda vissa delar av Microsofts nya ramverk Composite Application Guidance for WPF i grunden innebar här en god start. Detta ramverk är uppbyggt enligt konstens alla regler och designmönster samt har helt öppen källkod, man kan alltså se hur utvecklarna implementerat olika delar samt modifiera det precis enligt egna applikationspecifika önskemål. Kommunikationen har vidare lösts genom omfattande användning av händelser och kommandon. Ytterligare ett steg mot färre beroenden är att programmering mot interface eftersträvats överallt. Klassen som använder en annan klass ska alltså inte behöva veta något om den konkreta objekttypen och vidare inte behöva förändras vid modifieringar av denna objekttyp.

I det första kravet som ställdes ingick att lära sig ett antal designprinciper och designmönster, dokumentera dessa i en teoridel och sedan tillämpa dem vid implementationen. Detta gjordes och den färdiga applikationen innehåller konkreta implementeringar av alla de sju designmönstrena som beskrevs i teoridelen. Speciellt ett klurigt område var att separera den visuella delen från dess beteende på ett bra sätt och där kom det WPF-specifika designmönstret DataModel-View-ViewModel slutligen till undsättning. Resultatet blev att användargränssnittet View refererar till ViewModel som innehåller logik för att ta emot och delegera vidare UI-händelser. UI-objekt tillhör en speciell applikation medan icke UI-objekt idealt ska kunna återanvändas till flera användargränssnitt eller applikationer vilket uppnås genom att Viewmodel inte refererar till View. Genom att exempelvis använda designmöster som state och strategy uppnåddes vidare diverse andra designprinciper som att man bör föredra komposition framför arv, inkapsla objekt som varierar och att klasser bör vara öppna för uttökning men stängda för modifiering.

Det sista kravet som ställdes var att använda den allra senaste tekniken från Microsoft. På denna punkt måste erkännas att den nog var väl optimistisk och en stor källa till examenarbetets försening. Ny teknik kan var hur fräck som helst, problemet är dock att den sällan är fullständig i en första version. Exempelvis saknade WPF stöd för MDI (Multiple Document Interface), möjligheten att ett huvudfönster kan innehålla flera underfönster, som föregången Windows forms hade. Detta var ett absolut krav för en applikation av denna typ och tog en hel del tid i anspråk att implementera på egen hand. Vidare saknade Visual Studio från början heltäckande stöd för WPF och LINQ men det mesta föll på plats i och med att Visual Studio 2008 släpptes. Förutom att inledande versioner sällan är helt fullständiga är det ofta väldigt svårt att hitta god handledning och bra exempel att följa vid implementationen. Från början har ju ingen hunnit arbeta med tekniken än. Ingen har stött på problemen och utvecklat lösningar för att kringgå dem. Ett exempel på detta är WPF som erbjuder fantastiska nya möjligheter vid skapandet av moderna applikationer. Kraftfullheten kommer dock inte utan komplexitet och Microsoft erbjöd från början inga klara riktlinjer för hur man skulle lösa många av de problem som uppstår vid utvecklandet av en större applikation i WPF. Releasen av Composite Application Guidance for WPF innebar i somras en klar förbättring men fortfarande saknas tydliga riktlinjer för hur användargränssnittet ska kommunicera effektivt med resten av applikationen.

Med facit i hand kan man trots exjobbets försening ändå konstatera att det var bra att gå den hårda vägen när det gäller ny teknik. Den utgör ju framtiden och applikationen skulle exempelvis inte alls fått den rätta känslan och utssendet utan WPF.

6.2. Slutsatser

Trots diverse förseningar sitter jag nu alltså med ett färdigt system. Blev programmet som väntat? Var det värt allt arbete? Uppfylldes kraven som hade satts upp?

Det var det absolut väl värt allt arbete. Det finns några få programvaror tillgängliga för teknisk analys och automatisk handel men genom att betala dyra pengar för en sådan mjukvara kommer man inte ens i närheten av den flexibilitet som plattformen Trader erbjuder. Det är egentligen bara fantasin som sätter stopp för vilka olika entries, exits och tradingsystem man kan implementera och utvärdera med hjälp av detta system. Hade detta varit ett examensarbete med ekonomiskt fokus hade givetvis mer energi, arbete och efterforskning lagts ned på att justera demosystemen till att prestera bättre. Nu låg istället fokus på tekniksidan och de två enkla TA-systemen fyllde väl sin roll att verifiera att systemet Trader kan anses ha uppfyllt de krav och syften som angivits i examenarbetet. Framtida modifieringar av funktionalitet och utseende kommer att gå smidigt att implementera tack vare allt arbete som lades ned för att skapa en robust grunddesign och användandet av diverse designmönster och designprinciper.

Examensarbetet är nu avslutat men arbetet med Trader kommer fortlöpa. Det är nu det roliga arbetet börjar, jakten på högpresterande tradingsystem. Konstateras kan att detta utan konkurrens har varit det mest givande projektet under hela Chalmerstiden. Många hinder på vägen har givit en brant inlärningskurva och värdefull kunskap i att bygga ett större komplext system i ny Microsoftteknik.

6.3. Rekommendation för fortsatt arbete

Ett system som Trader går egentligen att vidareutveckla hur mycket som helst. Jag väljer ändå att här sätta punkt för mitt examensarbete då de mål som sattes upp i början har uppnåtts.

Vidare arbete består i att först och främst implementera fler tradingsystem att utvärdera. I detta arbete ingick ju endast två enklare modeller som baserar sig på teknisk analys för att verifiera att syftet med examensarbetet uppnåtts. Om några system sedan verkar lovande så blir nästa steg att

införa riktig orderläggning baserat på dessa tradingsystem. Tack vare att så mycket energi lades ner på att bygga ett flexibelt och utbyggbart system ska inte detta vara alltför tidskrävande.

Högt upp på önskelistan för fortsatt utveckling återfinns även bättre stöd och mer flexibla grafer. En bild säger mer än tusen ord och erfarenheten är att det absolut bästa sättet att åskådliggöra knepiga marknadslägen är via grafer. Beslutet att använda WPF har här en central betydelse då denna teknik innebär att många begränsningar som fanns med tidigare tekniker för att skapa användargränssnitt inte längre finns.

7. Källor

Bishop, J. (2008). *C# 3.0 Design Patterns.* Sebastopol: O'Really Media Inc.

Box, D., & Sells, C. (2007). *Essential Windows Presentation Foundation.* Crawfordsville: Addison-Wesley.

Freeman, E., & Freeman, E. (2004). *Design Patterns.* O´Reilly Media, Inc.

Gamma, E., Helm, R., Johnasson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software .* Addison-Wesley Professional.

Haskel, A., & Wilke, B. (2001). *Analyshandboken.* Stockholm: Aktiespararna Kunskap.

Hendry, B. a. (2004). *Modeling Financial Markets.* New York: McGraw-Hill.

Katz, J. a. (2000). *The encyclopedia of trading strategies.* New York: McGraw-Hill.

Lagerholm, M., & Nilsson, P. (2006). *Smallcap handboken.* Bromma: Aktiespararna Kunskap.

Marguerie, F., Eichert, S., & Wooley, J. (2008). *LINQ in Action.* Greenwich: Manning Publication Co.

Martin, R. C., & Martin, M. (2007). *Agile Principles, Patterns and Practices in C#.* Upper Saddle River: Pearson Education.

Sharp, J. (2008). *Visual C# 2008.* Redmond: Microsoft Press.

Siljeström, J. (2001). *Warranthandboken.* Kristianstad: Aktiespararna Kunskap.

Torssell, J., & Nilsson, P. (2000). *Boken om trading.* Stockholm: BörsInsikt AB.

Troelsen, A. (2007). *Pro C# with .NET 3.0.* New York: Apress.