FAKULTÄT FÜR !NFORMATIK

Rekonstruktion & Visualisierung von XETRA Orderbuch Daten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Intelligente Systeme

eingereicht von

Mario Annau
Matrikelnummer 0125736

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Univ.Prof.Dr. Kurt Hornik
Mitwirkung: Univ.-Ass. Dr. Michael Hütl

Wien, 03.12.2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT FÜR !NFORMATIK

Reconstruction and Visualization of XETRA Orderbook Data

Master's Thesis

for the degree of

Master of Science

in the Field

Intelligent Systems

submitted by

Mario Annau
Matrikelnummer 0125736

at the
Faculty of Informatics at the Technical University Vienna

Mentoring:
Supervisor: Univ.Prof.Dr. Kurt Hornik
Assistance: Univ.-Ass. Dr. Michael Hütl

Vienna, 03.12.2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 03.12.2008 _____

(Unterschrift Verfasser/in)

Kurzzusammenfassung

Das verstärkte Angebot von historischen Orderbuch Datensätzen großer elektronischer Börsenplätze macht es sowohl Marktteilnehmern als auch Wissenschaftlern möglich, die Markt-Mikrostruktur in einem bisher noch nie bekannten detailgrad zu analysieren. In dieser Arbeit wird eine Reihe von Softwarepaketen vorgestellt, um XETRA Orderbuchdaten zu rekonstruieren und sie anschließend in intuitiver Art und Weise darstellen zu können. Die Visualisierungs-Benutzerschnittstelle ermöglicht die schnelle Exploration von großen Orderbuchdatensätzen durch stufenlose Zoom- und Filterfunktionen. Dadurch wird dem Benutzer ein völlig neuer Eindruck über die Entwicklung von Orderbuchzuständen und Liquiditätsdynamik gegeben

Abstract

More historical orderbook datasets have been made available by big electronic stock exchanges to enable market participants and academics to study market microstructure at a tremendous degree of detail. This thesis presents various software packages to reconstruct and intuitively visualize XETRA orderbook data. The visualization user interface provides fast data exploration of big orderbook datasets with step-less zoom and filter functions. Thereby the user is given a new impression about the orderbook state- and liquidity dynamics.

Contents

1 Introduction

This chapter motivates the use of the implemented orderbook reconstruction and visualization packages at the beginning (Section 1.1). The rest of this chapter is organized as follows: Section 1.2 describes most recent changes in the stock exchange market as well as the competitive environment in the equity market. In Section 1.3 an overview of electronic orderbook trading is given. Finally Section 1.5 stresses important aspects of the rising algorithmic trading community.

1.1 Overview

Advances in computer technology as well as changes in the competitive environment had a large impact on stock exchanges around the globe and led to major changes — especially in the last decade. The entire trading environment has changed due to the emergence of fast and cheap electronic orderbook trading and reduced the importance of classical floor traders. Transparency also improved as real-time orderbooks became widely available among most major stock exchanges. Even historical orderbook data is sold by stock exchanges, mainly for the rising number of algorithmic traders who want to backtest their high-frequency trading strategies. But also academic institutions require detailed historical orderbook datasets to study liquidity supply and demand of the market microstructure.
Historical orderbook datasets from stock exchanges are usually made available in a raw database format and capture every trading event in the system. In order to run an in-depth analysis various preliminary pre-processing steps have to be taken to ensure consistency of the entire database. Further basic statistical analysis tools do not give an intuitive view of orderbook data and orderbook states. As is generally known — a picture is worth a thousand words. A good visualization tool of orderbook data can save a lot of time doing explorative data analysis and gives a much deeper impression of orderbook states, or equivalently, liquidity demand and supply dynamics over time. Despite the existence of various chart analysis tools implemented in professional trading software packages no visualization of orderbook data itself has been presented in literature.
This thesis addresses exactly these issues and presents a software package to reconstruct and visualize high-frequency orderbook datasets. Thanks to Deutsche Börse Group a high-frequency dataset from the XETRA trading system was made available to show the functionality of the reconstruction and visualization engine. Therefore

the software library has been optimized and tested for XETRA orderbook data — but also other orderbook datasets could be mapped and used by the implemented software package.
After the initial order reconstruction process is completed the orderbook visualization engine is able to present the orderbook dataset in an easy-to-use user interface. The interface supports a zoom function to display orderbook states at any time and price interval as well as filter and export features. The intuitive way of displaying complex orderbook datasets facilitates explorative analysis of high-frequency orderbook data and therefore represents a viable tool to study market microstructure at any detail.

1.2 Changes in the Stock Exchange Market

The severe competition for liquidity among exchanges worldwide has led to major changes of the exchange market structure and the trading platforms. Potential cost savings and gains from economies of scale are the most common arguments for a consolidation in the stock exchange market.
From a technical viewpoint the development of a new trading platform which handles a sufficient order flow is usually very complex and expensive. As the development costs are fixed big economies of scale can be gained by adding more shares or other financial products to a trading platform. A scalable IT infrastructure facilitates the integration of various trading platforms accompanied with overall cost reductions (maintenance, staff, etc.). This is an important aspect for the computer intense electronic orderbook market (see Section 1.3) in particular which gained significant importance over the last years and accounts for the lion share of trading turnover on most major stock exchanges today. Also increased order flow and speed requirements of algorithmic trading clients (see Section 1.5) make cost intensive improvements of the IT infrastructure necessary and generally favors big exchanges. Last but not least diversification is another key point for further mergers between exchanges as they hope to attract more liquidity by offering a broadened range of tradeable products in more regions around the globe.
Although competition between the big stock exchanges is intense new competitors arose in form of electronic markets, 'crossing networks' and 'dark pools'. Electronic markets like the American BATS and Direct Edge or the European Chi-X and Turquoise aspire to become full exchanges and have already grabbed significant market shares. Those newly established platforms are typically backed by consortia of banks in hope to cut trading fees. In contrast crossing networks or dark pools have been established to trade large blocks away from the eye of the exchanges. By hiding the buyer's identity and the price of each trade, market movements against the traders' position can be avoided.
Altogether fierce competition among the major stock exchanges and its rivals has led

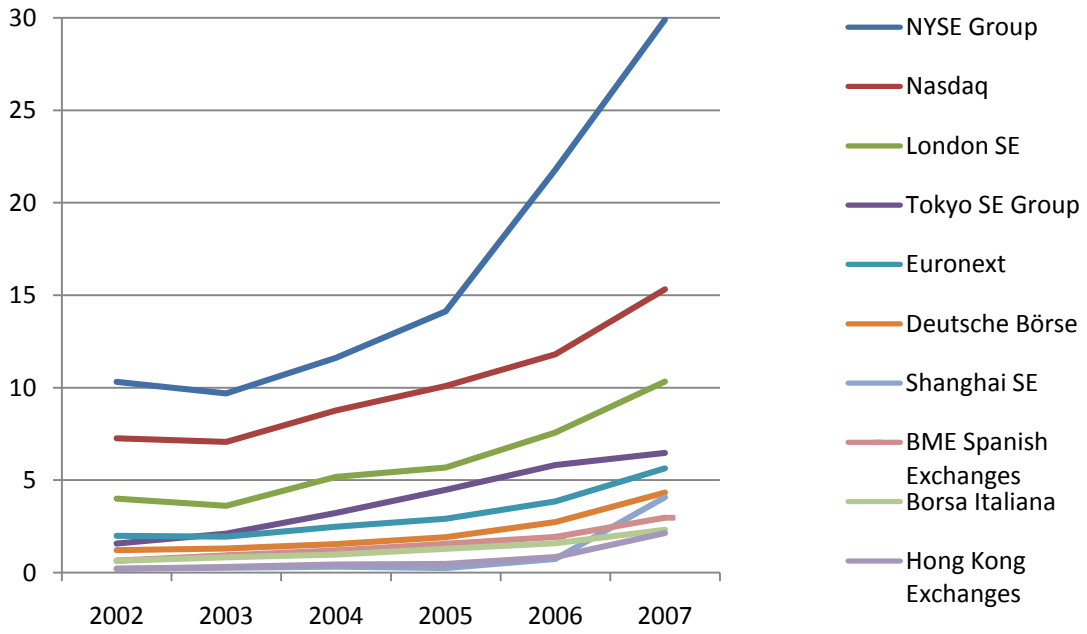

Figure 1.1: Historical increase in Value of Share Trading among major stock exchanges in trillion USD as reported by the World Federation of Exchanges in [WFE08a]
Top ten Stock exchanges have been selected by the their USD value of share trading turnover in 2007.

to reduced trading fees especially for heavy users like algorithmic traders. While trade margins shrank the value of share trading (VST) significantly increased in the past decade. Figure 1.1 shows the VST of the top ten stock exchanges (in terms of VST) as listed in [WFE08a]. As comparable data could only be obtained for the years 2002–2007 from [WFE08a] this figure does not reflect the actual market concentration after recent mergers. It can be seen that the NYSE is the dominant stock exchange with an estimated VST of 30T(rillion) USD , followed by NASDAQ (15T USD) and the LSE (10T USD). Compared to the top 3 equity markets the Tokyo SE, Euronext, Deutsche Börse and the Shanghai SE just play minor roles with a VST of around 5T USD. Taking into account the mergers between NYSE–Euronext and LSE–Borsa Italia the market concentration of the top three becomes even more evident.

1.3 Electronic Orderbook Trading

1.3.1 Overview

Electronic Orderbook trading is a method to trade various financial securities (such as stocks, bonds), currencies and derivatives electronically. Virtual market places bring potential buyers and sellers together who can access the exchange through electronic communication networks. Orderbooks form the core of electronic markets where buy and sell orders are matched against each other. The matching engine follows the rules defined by the exchanges' market model and is responsible for a transparent and continuous price determination process of the traded security.
Advances in computer technology and communication networks have led to an incredible increase in electronic trading on most stock exchanges around the globe. It can be seen that traditional floor trading and brokers are being reduced as market participants can directly trade financial assets over electronic trading platforms. Cost reductions for institutional investors are not the only advantages of electronic trading as only a fraction of traders and brokers have to be employed in the exchanges' back offices. Improvements also come from a faster and more transparent transaction process which involves order routing and matching.
Major stock exchanges have invested heavily in their IT infrastructure to reduce order roundtrip times to less than ten milliseconds. Market participants can connect their systems directly to the backbone of the stock exchanges' computer systems to further reduce latency times. The order matching engines are now capable to handle even higher order flows and more financial securities at a time granularity of 10ms.
Market transparency is ensured by (partially) open orderbooks which give traders an in-depth view of the liquidity supply and demand structure. Further modern trading software allows to monitor the entire trading process combining various additional information sources from data feeds or statistical indicators. Thus it is possible to get a very detailed picture of any market. It also allows market participants to react very quickly to surprising market events.
Finally these advantages of electronic trading typically lead to more efficiency in trading and to an increased liquidity of the markets. More liquidity typically reduces bid/ask spreads and therefore results in cheaper transaction costs for market participants.
Today's stock exchanges rely heavily on electronic trading platforms. Examples like the XETRA trading system (developed by Deutsche Börse) and the NASDAQ stock exchange have initially been implemented as fully electronic orderbook markets. Even the traditionally floor based NYSE has most recently introduced the Hybrid®platform where almost all listed stocks can be traded either immediately on the electronic platform or further routed on the trading floor.
Figure 1.2 shows the increasing market share of electronic trading done in the XETRA

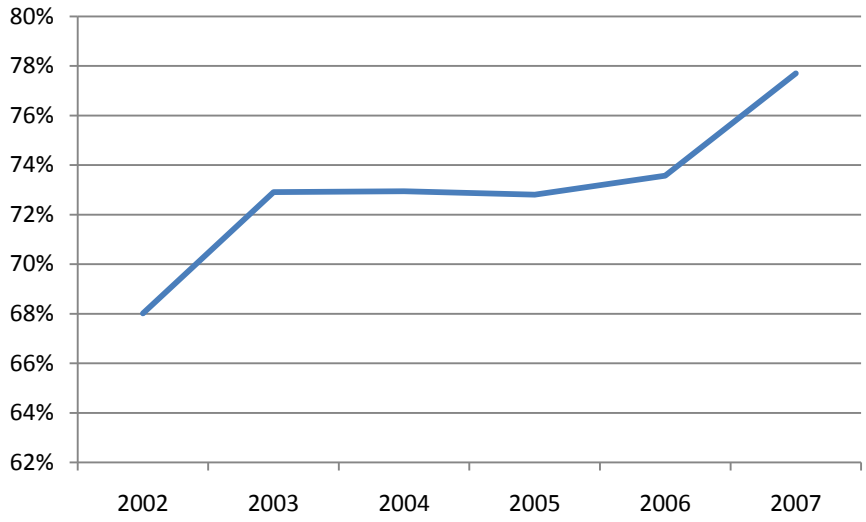

Figure 1.2: Share of electronic equity trading at Deutsche Börse AG in 2007 reported by World Federation of Exchanges ([WFE08b])

system by total value of share trading of the Deutsche Börse Group as reported by the World Federation of Exchanges (WFE) in [WFE08b]. It can be seen that the share has climbed from 68% to 78% in the five year time period between 2002 and 2007. Taking into account the value of share trading numbers reported in Figure 1.1 electronic orderbook trading was responsible for 3.37T USD of trading turnover in 2007.

1.3.2 Orderbook Data Availability

Data from the stock exchanges' internal orderbook typically allow the reconstruction of orderbook states at any time. As the in-depth view into the orderbook gives a detailed picture about the current liquidity supply- and demand situation of a tradeable product increasing interest arose from market participants to incorporate this additional information source in their trading strategies. Increasing demand for orderbook data mainly comes from algorithmic traders who are implementing high-frequency trading strategies (see Section 1.5). But also academic institutions require these datasets to study the empirical liquidity behavior in the market microstructure (Section 1.4). Offering historical data is a lucrative business and most stock exchanges have already reacted to make real-time data feeds and historical datasets available to paying clients. Real-time data feeds from the top three stock exchanges (NYSE, NASDAQ, LSE) lets

Stock Exchange	Product	Market Depth	Reference
NYSE	TotalView	full	[NYS08a]
NASDAQ	OpenBook	full	[NAS08b]
LSE	Level 2	full	[LSE08a]
Deutsche Börse	Level 2	10	[DBG08]

Table 1.1: Availability of real-time orderbook data from major stock exchanges

Stock Exchange	From	To	Price	Reference
NYSE	2002-1-24	present	24,500 USD	[NYS08b]
NASDAQ	2005-8-1	present	2,000 USD/month	[NAS08a]
LSE	1997	present	12,360 GBP/year	[LSE08b]
Deutsche Börse	-	-	-	

Table 1.2: Availability of historical orderbook data from major stock exchanges.

subscribers view the full bid/ask market depth (aggregated volumes of any limit) of any listed security. By contrast the Deutsche Börse stock exchange currently offers Level 2 realtime data with a market depth of only 10[1]. Table 1.1 summarizes the offered products of the above mentioned stock exchanges.

Most recently even historical orderbook data has been made available by stock exchanges. These datasets are interesting for backtesting of trading strategies as well as academic purposes. Only the LSE has made its complete internal orderbook already publicly available. It includes each order event of the book and is therefore comparable to the dataset investigated within this thesis. The datasets offered by the NYSE and NASDAQ only contain aggregated volumes for the full market depth. At least the number of orders aggregated is reported by the NYSE. The NASDAQ data product is further restricted to a minimum time interval of one minute.
The Deutsche Börse currently offers no publicly available historical orderbook datasets. Interested clients therefore have to search for data providers offering historical level 2 data which does not have the complexity of the stock exchange's internal rebuilt orderbook. Historical order data products are listed in Table 1.2.

[1] only the 10 best buy/sell orders are displayed

1.4 Literature on Open Orderbook Markets

As already mentioned in Section 1.2 the availability of orderbook data allows academics to study empirical market microstructure at any detail. This section gives a brief overview of literature related to the microstructure of open orderbook markets with a clear focus on studies using reconstructed (XETRA) orderbook data.
While Harris' book [Har03] is a broad treatment of economic theory and trading institutions O'Hara's [O'H95] is the standard reference for the theory of market microstructure. Most recently Hasbrouck [Has07] surveyed various econometric models for the empirical analysis of market microstructure.
Important articles about modeling limit orderbook markets come from Glosten [Glo94], Parlour [Par98] and Foucault-Kadan-Kandel [FKK01]. According to Glosten's model Limit order traders post orders with characteristics (price and volume) that depend on the underlying asset value. Parlour and Foucault-Kadan-Kandel suggest that the mix of patient and impatient trader is another key factor to model orderbook dynamics.
The emergence of orderbook data for academic purposes has led to various studies of market microstructure on almost all major stock exchanges around the world. By reconstruction of the limit orderbook from high-frequency raw data various aspects like liquidity supply/demand dynamics and commonalities have been analyzed. For example, De Winne [dWdH03] and Ekinci [Eki05] studied orderbook dynamics of the fully reconstructed orderbook from Euronext and the Istanbul stock exchange, respectively. Wagner [Wag] measured aggregate market liquidity of equity markets in the United States, the United Kingdom and Japan. Ranaldo [Ran] studied liquidity dynamics around public events on the Paris Bourse. Subrahmanyam [SCR] et al. analyzed commonalities in liquidity on the NYSE. Please refer to [LG] for the most recent collection of articles on stock market liquidity and market microstructure.
Considering the scope of this thesis the analysis of XETRA orderbook data is of particular interest. Papers by Grammig et al. ([GHR04],[FG05]) and Gomber et al. [GST04] focused on the liquidity supply and demand dynamics in the XETRA orderbook. Also the commonalities in the XETRA orderbook have been analyzed by Grammig in [BLGG05].
Loistl et al. have also published numerous papers to investigate the market microstructure using the dataset investigated within this theses. Ranging from modeling 'cadlag' market event time series [LPH06] over best execution issues [LPH07] a wide variety of aspects have been analyzed. Even algorithmic trading patterns have been found in [PLH07]. Most recently also liquidity commonalities in the orderbook have been studied using principal component analysis [PLHK08].

1.5 Algorithmic Trading

Algorithmic trading refers to computer programs that automatically generate and modify orders based on a predefined algorithm. Based on numerous features such as news feeds and technical indicators the program decides the timing, limit and size of orders and enters them into an electronic trading system. Speed and scalability to various instruments are clearly the main advantages of algorithmic trading programs. They react to changed market conditions faster than human traders and monitor an incredible amount of financial instruments. The rising popularity of algorithmic trading has led to numerous changes of the stock exchanges' IT infrastructure as most institutional investors are using automated trading programs. A study by AITE Group, a consultancy, reckons that the market share of algorithmic trading will reach over 50% by 2010 [AIT06]. New innovations in this new sector of the financial industry are also discussed in various trading magazines like Automated Trader [Aut08].
Before deployment the return of algorithmic trading strategies is typically backtested on historical data material. By monitoring the trading statistics of an algorithm its behavior and risk can be analyzed under various historical market conditions. After the algorithm's parameters have been calibrated it is deployed to the actual trading system. Further successful algorithms can be easily scaled to other financial instruments as they handle huge amounts of data feeds simultaneously. Last but not least algorithms have not the problem of emotional reactions and trading decisions of human traders.
Many institutional investors, especially hedge funds, are already running algorithmic trading desks on major stock exchanges. Algorithms are typically designed for one of the following trading strategies:

Transaction cost reduction: The basic idea behind this trading strategy is the splitting of large orders to optimize benchmarks like the time weighted average price (TWAP) or, most commonly, the volume weighted average price (VWAP). Kissel et al. [KG03] gives a good overview of such trading strategies. Entering large orders at once into the trading system typically leads to price movements in the adverse direction — also known as market impact. Most stock exchanges like the XETRA trading system also allow special order types (typically known as Iceberg orders, see Section 2.3.3) which do not reveal their entire volume in the orderbook. Unfortunately these quite simple order mechanisms also can be detected or reverse engineered by specialized algorithms. Transaction cost reduction algorithms are able to split these orders according to various optimization methods and even schedule the timing of order entry.

Arbitrage: Algorithmic arbitrage strategies are designed to generate risk-free profits by exploiting mispriced financial instruments. Speed is the essence for those algorithms to make continuous profits as most arbitrage opportunities only last for a few seconds. Most recently even algorithms that incorporate a certain

level of uncertainty have been implemented, also known as statistical arbitrage [Pol07].

Market Making: Market making involves placing a Limit order to sell (or offer) above the current market price or a buy Limit order (or bid) below the current price in order to benefit from the bid-ask spread. Most market makers make use of algorithms to determine the bid/ask spreads accordingly.

Model based strategies: Model based strategies include complex computer models calibrated and backtested with historical data. These models often contain artificial intelligence concepts like neural networks and evolutionary programming. Even news trading algorithms which handle news feeds from Thomson Reuters or Bloomberg in only fractions of a second are part of this category.

In order to detect short term trading opportunities orderbook data has become extremely useful to algorithmic traders. As high-frequency strategies often suffer from slippage costs[2], order limits and trading opportunities can be better calibrated by using high-frequency orderbook data.
In [Har08] Hartle compares price based and orderbook based high-frequency trading strategies on the S&P Mini future market. His results show that orderbook based strategies generate better trading signals and returns. In academic literature the prominent Penn Lehman Automated Trading Project (PLAT) and the project run by the CCFEA institute at Essex University were two major initiatives to test a broad range of trading strategies on high-frequency orderbook data.

1.5.1 Penn-Lehman Automated Trading Project

The PLAT project directed by Michael Kearns was initiated in 2002 as a trading competition were participants could implement their own trading agents using a C++ Application Programming Interface (API) [KO03]. The implemented trading strategies have been tested for a single stock as the framework interface only provided access to the Microsoft (MSFT) stock listed on the NASDAQ stock exchange. The orderbook data was obtained from the Island electronic crossing networks (ECN) which is one of the biggest electronic markets for NASDAQ stocks. For trading agents the orderbook was visible to a market depth of 15.
After a preliminary backtesting phase on historical data competitors could trade on the real-time orderbook. Orders from the island orderbook as well as those entered by virtual trading agents have been merged to a parallel market established by the project authorities. The implemented trading strategies reported showed some interesting results. Especially relatively simple trading algorithms such as the Static

[2] Price difference between order entering by client and order execution in the trading system

Orderbook Imbalance (SOBI) or the Electronic VWAP (E-VWAP) strategy showed remarkable returns [KKMO04]. More complex strategies also included price indicators [SR05], stock news mining [Har04], boosting algorithms [CF06] and evolutionary programming [SSSK06].
Unfortunately the PLAT project was discontinued in 2006.

1.5.2 CCFEA SETS E-PLATFORM

The SETS E-PLATFORM created by the CCFEA institute from Essex University also provides a framework to implement and backtest trading strategies using a JAVA API. Unlike the earlier mentioned PLAT project its dataset comes from the rebuild LSE orderbook. The matching procedure can be studied with the CCFEA Limit Orderbook application which is available online under [Mal07]. Figure 1.3 shows a screen shot of the application user interface.

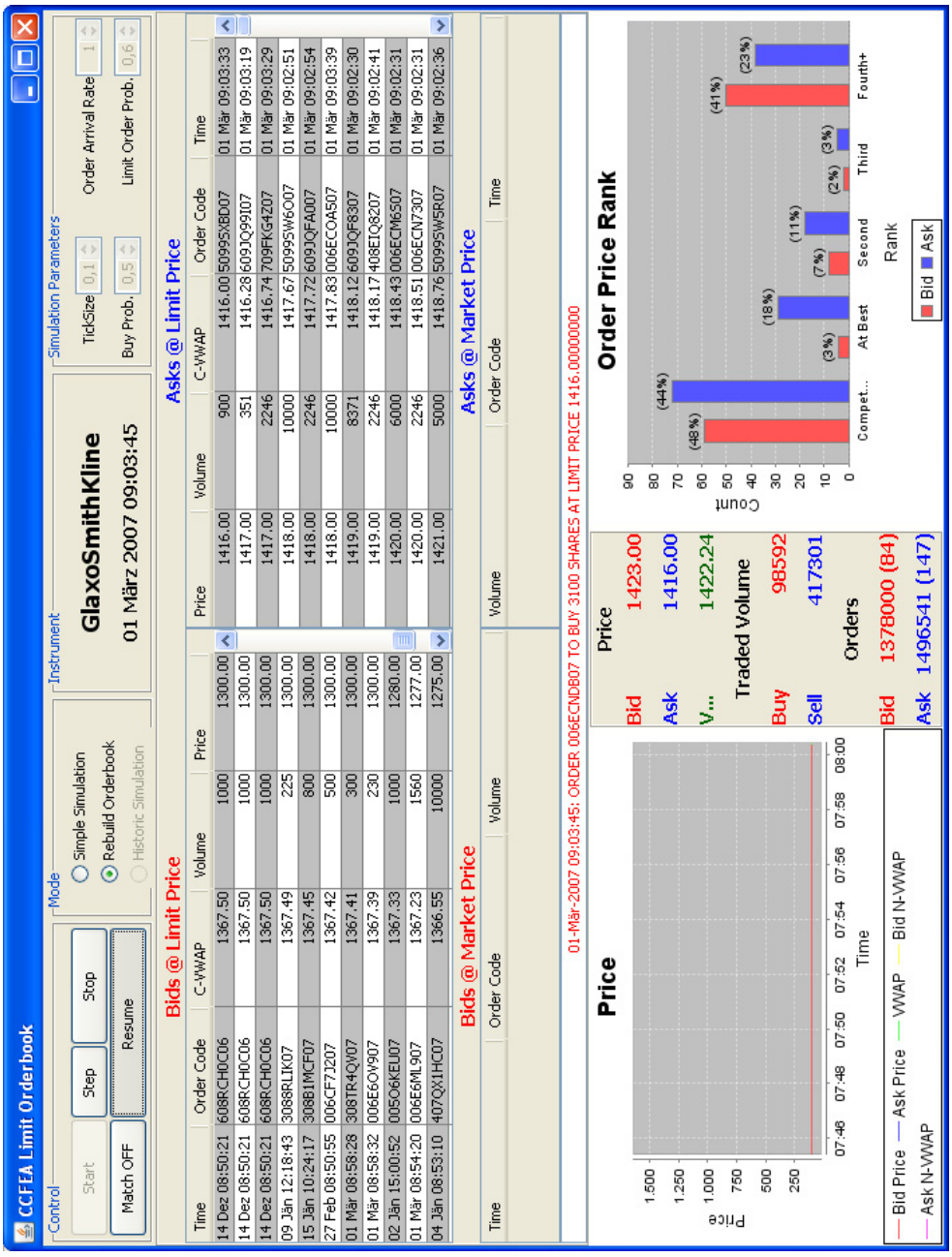

Figure 1.3: CCFEA Limit Orderbook Application

Release	Date	Content
1.0	10.06.1997	Introduction Xetra front end
2.0	28.11.1997	Introduction Xetra back end
3.0	12.10.1998	Xetra Grundstufe
4.0	02.05.2000	Xetra Warrant Trading
5.0	02.10.2000	European Alliance Release
6.0	30.04.2001	CCP-Readiness Release
7.0	19.08.2002	Xetra BEST Introduction
7.1	06.12.2004	Subscription Right Trading
8.0	23.04.2007	Performance & Speed Release
8.1	22.10.2007	MiFID Readiness Release
9.0	28.04.2008	Introduction to Certificates
9.1	Dec. 2008 (planned)	Introduction to ETF Trading

Table 2.1: XETRA Release History (1.0-9.1)

2 XETRA Market Model

This chapter gives a basic overview of the underlying market model used by the XETRA trading system. Starting with a brief introduction of the XETRA (Release) History (Section 2.1) orderbooks (Section 2.2), order types (Section 2.3), additional flags (Section 2.4) and trading phases (Section 2.5) will be further described for the XETRA Release 7.1, which defines all relevant rules for the dataset (Chapter 3) under investigation.

2.1 XETRA History

XETRA®[1] is a fully electronic trading platform of Deutsche Börse AG. Based on the Deutsche Börse EUREX system XETRA was implemented in 1997 by Deutsche Börse Systems and Anderson Consulting (today Accenture) and has been further developed through various releases [ISE07]. Table 2.1 gives a short overview of the XETRA release history. The significant changes within the exchange industry over

[1]The name XETRA can be interpreted as an abbreviation of E**X**change **E**lectronic **TRA**ding

the last years also show up in the most recent releases of the trading platform. Especially performance improvements have been a big issue since release 8.0 to meet speed requirements of algorithmic traders who already account for over 40% of all executed trades. With a minimum roundtrip time of 4ms, an average roundtrip time of 13ms[Gro08a] and an availability of 99.97%[Gro08b] XETRA is one of the fastest and most reliable electronic trading platforms worldwide. Release 8.1 was aimed to meet MiFID requirements in terms of transparency (Best Execution). Later releases increased the number of tradeable instruments further, including certificates and exchange traded funds, to over 300,000. XETRA also offers remote access for over 260 participants in 19 countries and more than 4700 authorized traders [Gro08a].
Still in an early development cycle XETRA was created as a scalable platform under an independent brand name to be easily sold to other exchanges. Today a reasonable number of exchanges have implemented the XETRA platform like the Frankfurt Stock Exchange (1997), the Vienna Stock Exchange (1999), the Irish Stock Exchange (2000), the European Energy Exchange (2005) and the Shanghai Stock Exchange (upcoming).

2.2 Orderbook

An orderbook forms the basis of the market model's price determination process and ensures a transparent and continuous price development. Two different books are kept for (unexecuted) buy and sell orders, respectively. Incoming orders are sorted by the so-called Price-Time priority rule. Table 2.2 illustrates the sorting procedure in more detail where **T** describes the time[2] when the Order was entered into the book, **L** the limit[3] and **V** the volume of the particular order.
Buy(Sell) Orders with a higher(lower) limit get a higher priority and are therefore placed higher in the orderbook . If the limits of two orders are equal the one which has been entered earlier gets a higher priority (compare e.g. buy orders in row 3 and 4). Market orders (M) always have the highest priority in terms of price and are only sorted by its timestamp of order entry if other Market orders are available in the book (compare buy orders in row 1 and 2).
The bid/ask spread is defined as the difference between the limits of the best (highest priority) buy and sell order. In case of a crossed orderbook[4] the matching procedure starts which will be described in Section 2.3 for all different order types.

[2] For simplicity reasons time is an integer number instead of a date/time value
[3] For better readability limit has just one decimal place
[4] price overlapping of bid/ask orders or existence of Market orders which results in a less or equal to zero bid/ask spread

BUY			SELL		
T	L	V	T	L	V
2	M	100	4	M	100
3	M	300	3	98.2	200
1	98.0	100	2	98.3	100
5	98.0	1000	1	98.5	1100
4	97.9	500	5	98.6	300

Table 2.2: Orderbook Example
Example shows the Price-Time priority rule applied by the order sorting procedure to buy orders (left) and sell orders (right).

2.3 Order Types

The following sections give an overview of all different order types defined by the XETRA market model. Main order types like Limit orders (Section 2.3.1) and Market orders (Section 2.3.2) as well as Market-to-Limit orders (Section 2.3.3) also include examples to get a better insight of the matching procedure.

2.3.1 Limit Order

Limit orders represent the most common order type in the XETRA orderbook and are entered at a fixed limit price. If the limit price of a buy(sell) order overlaps with the limit of a sell(buy) the order is executed against all possible orders on the other side of the book with respect to the Price-Time priority rule.
Table 2.3 shows an example of Limit order matching. The incoming sell order at time T=5 has a lower or equal limit than the buy orders in row 1 and 2 and generates a crossed orderbook with a bid/ask spread of -0.1. Thus the sell order will be partially executed with a volume of 100 with the first buy order at 98.3. The rest is fully executed with the second buy order. The price received by the seller is therefore $98.3 * 100 + 98.2 * 100 = 19650$. The right side shows the resulting orderbook at T=6 with a bid/ask spread of 0.2.

2.3.2 Market Order

Market orders are unlimited buy/sell orders. They are to be executed immediately at the next best price determined. If the volume of the buy(sell) Market order is bigger than the volume of the best sell(buy) order several trades are generated. Therefore big Market orders have the characteristic of 'walking-up-the-book' which is shown in

BUY			SELL			BUY			SELL		
T	L	V	T	L	V	T	L	V	T	L	V
2	**98.3**	**100**	**5**	**98.2**	**200**	3	98.2	200	2	98.4	300
3	**98.2**	**300**	2	98.4	300	1	98.0	100	4	98.5	1000
1	98.0	100	4	98.5	1000	5	98.0	1000	4	98.7	400
5	98.0	1000	4	98.7	400	4	97.9	500	1	98.8	500
4	97.9	500	1	98.8	500						
T = 5						**T = 6**					

Table 2.3: Limit Order Matching Example
Example shows the matching of an incoming sell Limit order at T=5 with two buy orders. Result at T=6 is presented on the right side.

BUY			SELL			BUY			SELL		
T	L	V	T	L	V	T	L	V	T	L	V
2	**98.3**	**100**	**5**	**M**	**1000**	5	98.0	500	2	98.4	300
3	**98.2**	**300**	2	98.4	300	4	97.9	500	4	98.5	1000
1	**98.0**	**100**	4	98.5	1000				4	98.7	400
5	**98.0**	**1000**	4	98.7	400				1	98.8	500
4	97.9	500	1	98.8	500						
T = 5						**T = 6**					

Table 2.4: Market Order Example
Example shows the matching of a sell Market order at T=5 with 4 buy orders. Resulting orderbook at T=6 is shown on the right side.

Table 2.4.
The incoming sell order at T=5 with a volume of 1000 is partially executed against the best buy order at 98.3 (V=100), the second buy order at 98.2 (V=300) and the third buy order at 98.0 (V=100). Finally the rest volume of 500 is executed at 98.0 with the fourth buy order. The result at T=6 is shown on the right side.

BUY			SELL			BUY			SELL		
T	L	V	T	L	V	T	L	V	T	L	V
2	**98.3**	**100**	**5**	**M**	**1000**	3	98.2	300	5	98.3	900
3	98.2	300	2	98.4	300	1	98.0	100	2	98.4	300
1	98.0	100	4	98.5	1000	5	98.0	1000	4	98.5	1000
5	98.0	1000	4	98.7	400	4	97.9	500	4	98.7	400
4	97.9	500	1	98.8	500				1	98.8	500
T = 5						**T = 6**					

Table 2.5: Market-to-Limit Order Example
Example shows the matching of a Market-to-Limit sell order with one buy order at T=5. Resulting orderbook including changed Market-to-Limit order type at T=6 is shown on the right side.

2.3.3 Additional Order Types

Market-to-Limit Order

Market-to-Limit orders are unlimited buy/sell orders, which are to be executed at the auction price or (in continuous trading) at the best limit in the orderbook, if this limit is represented by at least one Limit order and if there is no Market order on the other side of the book. Any unexecuted part of a Market-to-Limit order is entered into the orderbook with a limit equal to the price of the executed part. This features prevents the order to walk-up-the-book.
Table 2.5 shows how a Market-to-Limit order works. In contrast to the Market Order example given in Table 2.4 the incoming Market-to-Limit order at T=5 is only executed against the best buy order at 98.3 (V=100) and immediately changes to a Limit order with L=98.3.

Iceberg Orders

Due to the price influencing effect of large Limit orders Iceberg orders, mainly used by institutional investors, are entered to hide the actual volume of an order. Main properties of Iceberg orders are the initial peak volume, the whole (hidden) volume and a specified limit price. When an Iceberg order is entered into the system only the peak volume is visible in the orderbook. Once the peak volume has been fully executed another peak (Limit order) with the same volume and limit is entered into the orderbook. Finally, if the peak volume is greater than the residual volume of an Iceberg order, the residual volume is entered into the orderbook. Because of the simple structure of Iceberg orders they can be identified using computerized methods.

Stop Orders

Stop orders contain an additional stop-limit attribute and are typically not immediately visible in the orderbook. With a stop order a trader can either 'stop into the market' in case of a buy-stop order (order is entered if price is equal or higher than limit) or limit losses with a sell-stop order (order is entered if price is equal or lower than limit).

2.4 Additional Flags

Additional flags added by the trader can further specify the matching behavior of an order. By using Immediate-or-Cancel and Fill-or-Kill flags traders have a better control over the execution price of an order. These flags can reduce slippage costs as well as market impact by restricting orders to 'walk-up-the-book'. By contrast the Triggered-Stop-Order flag identifies orders which have been entered because a specified stop-limit has been reached.

Immediate-or-Cancel: The Immediate-or-Cancel flag forces the trading system to check which parts of the incoming order can be matched immediately at arrival time. All unexecuted parts of the order are to be canceled immediately. In case of a Market order or an aggressive Limit order this restriction prevents a 'walking-up-the-book' of the order.

Fill-or-Kill: In contrast to the Immediate-or-Cancel flag the Fill-or-Kill restriction lets the system check if the full order can be executed immediately. If not the entire order is deleted.

Triggered-Stop-Order: The Triggered-Stop-Order flag indicates that the order has been entered into the system because a specified stop-limit has been reached (see also Stop orders in Section 2.3.3).

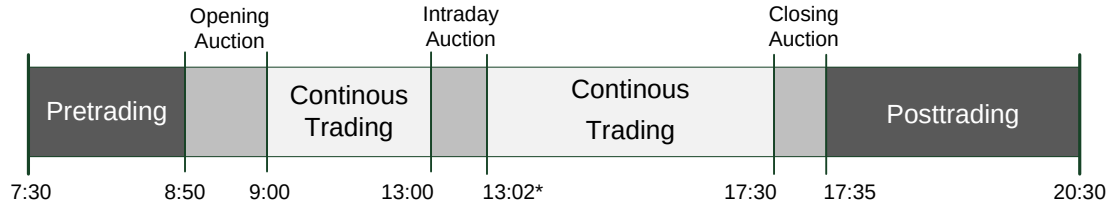

Figure 2.1: DAX Trading Phases of DAX stocks in the XETRA trading system [Gro03]

2.5 Trading Phases

This section gives a brief overview of the trading phases for DAX traded stocks initially defined by XETRA Release 7.0[Gro03]. Figure 2.1 shows the basic trade flow including the trading hours at the bottom. Opening hours of the XETRA trading system are from 7:30 until 20:30 while traders can enter, modify or delete their orders. The trading day starts with the Pre-Trading phase at 7:30 and ends with the Post-Trading phase at 20:30 (Section 2.5.1). Actual trading starts at 8:50 at the beginning of the Opening Auction (OA) (Section 2.5.2). The OA ends at 9:00 and the Continuous Trading (CT) phase is initiated until 17:30 (Section 2.5.3). The CT phase is only interrupted during the Intraday Auction (Section 2.5.2) at 13:00 and by possible volatility interruptions which can occur during the day (Section 2.5.4). The Closing Auction ends the trading session at 17:30 (Section 2.5.2).

2.5.1 Pre/Post-Trading

The Pre-Trading phase and the Post-Trading phase are the same for all equities whereas the course of the trading phase may vary from equity to equity. According to their segmentation, individual equities are traded in different trading models and at different trading hours. The Pre-Trading phase initiates the trading session. Market participants can enter orders and quotes for preparing the actual trading day and modify or delete their existing orders and quotes. The exchange confirms the member's order entry and maintenance by order confirmation. Market participants do not receive an overview of the market's orderbook situation as the orderbook is closed during this phase. The last price fixed or the best bid/best ask limits after the last auction of the previous day are displayed.

2.5.2 Auctions

In auctions, all order sizes (round lot and odd lot orders) are tradable. By considering all existing orders (Market orders, Limit orders, Market-to-Limit orders and Iceberg

orders) in one equity, a concentration of liquidity is ensured. Iceberg orders participate with their full volume in auctions. Concerning the price determination in auctions, Market-to-Limit orders are handled in the same way as Market orders. If there is no auction price Market-to-Limit orders, which were entered during the call phase of the auction, are deleted. If there is an auction price, remaining parts of Market-to-Limit orders, which are partly executed, and Market-to-Limit orders, which are not executed, are entered into the orderbook with a limit equal to the price of the auction. Price determination in auctions is effected according to the principle of most executable volume. At the same time Price-Time priority is valid so that the maximum of one order, which is limited to the auction price or unlimited, can be partially executed. The orderbook remains partially closed during the auction's call phase. As information about the market situation participants obtain the indicative price or the best bid/ask limit which may be augmented by market imbalance information. Market participants are informed via an auction plan about the time the individual equity is called.

2.5.3 Continuous Trading

Depending on the trading model and trading segment, orders of any size or round lots can be traded in the trading phase. The trading phase varies according to the respective trading segments. Depending on trading segment, equities will be traded in one of the trading models. Considering the analyzed dataset from the XETRA trading system all order types and matching rules apply as defined by the market model (see Section 2.2 and Section 2.3).

2.5.4 Volatility Interruptions

XETRA contains safeguards to improve price continuity and to increase the probability to execute Market orders. The main safeguards are volatility interruptions in auctions and continuous trading as well as Market order interruptions in auctions (not in auctions initiated by volatility interruptions). As far as designated sponsors exist for an equity, they will enter quotes during volatility interruptions. Volatility interruptions can be initiated in two ways: If the price of an equity lies either outside the dynamic or outside the static price range. Price ranges are typically adjusted by the historical volatility of respective equities.
The dynamic price range is determined by the last reference price (last traded price). By contrast, the static price range is wider and is determined by the reference price of the last auction.

3 Data Set

This chapter introduces the dataset of all DAX30 stocks from the XETRA trading system in the time period between 2005-1-5 and 2005-1-12. After a general introduction (Section 3.1) all attributes describing order modification records in the database will be described in Section 3.2. Finally event code sequences that describe the entire lifetime cycle of an order will be introduced in Section 3.3.

3.1 Introduction

The Dataset under investigation contains the fully rebuilt orderbook of all DAX30 stocks listed on the Frankfurt Stock Exchange and traded on the XETRA platform in the week between January 5th and 12th 2005. The Frankfurt Stock Exchange (FWB®) where the XETRA system has been initially implemented is by far the largest of Germany's seven stock exchanges While most of the DAX30 stocks are also listed on other exchanges over 97% of trading was done via the XETRA trading platform in 2005 [Gro05].
Each order modification (like Insertion, Execution or Deletion; see Section 3.2.12) done on the XETRA trading system is recorded as a single database entry. More than 5 million (5,014,200) database entries can be identified in the raw Microsoft Access database with a size of 1.7GB. The dataset represents the vast majority of trading done on DAX30 stocks as only relative small volumes are cross-listed on other stock exchanges across the globe (mostly the NYSE). The most important index for the German stock market, the XETRA DAX30 index, is also calculated from XETRA system.
As already mentioned in Section 1.2 rebuilt XETRA orderbook data is not publicly available. Thanks to Deutsche Börse AG access to the limited subset has been granted for academic purposes. The immense detail of the dataset allows the analysis of every single trading activity and the reconstruction of every order and orderbook state in the respective time interval.
To get a more detailed picture of the entire trading week under investigation Figure 3.1 shows the movement of the DAX index by a candlestick chart[1] aggregated to a time

[1] Candlesticks aggregate open, high, low and close prices in a respective time interval. Bars show the difference between open and close prices - white bars indicate positive price movements, black negative ones. Lines on tops and bottoms of the bars show the entire price range between highs and lows.

frame of one hour. The money volume traded was calculated from the dataset and is shown by blue bars. It can be seen that the index moved in a range between the high of 4325 on 2005-1-7. and the low of 4200 on 2005-1-12. Also the typical u-shape of the traded money volume catches the eye as most trades are done at the beginning and the end of each trading session. In the morning sessions until 11:00 and from 16:00 on, when the American stock exchanges open, more liquidity flows into the market.

Table 3.1 further summarizes the movement of each stock included in the DAX30 dataset in the analyzed trading week. By average daily turnover the Deutsche Telekom stock ranks at first place, followed by SAP and Deutsche Bank.

3.2 Attributes

Each modification to an order is represented by a database entry in the trading system. This section introduces the attributes describing each order modification entry found in the rebuilt XETRA orderbook database. 14 different attributes specify each order modification and are described in the following subsections. Table 3.2 lists all attributes of the orderbook database.
In order to improve readability database attribute names are emphasized in *italic shape*, database values which store enumeration-like data types in `rectangles` and all other database values in `teletypefont` characters. Table 3.2 also includes value examples for each database attribute.

3.2.1 ModificationTimestamp

The *ModificationTimestamp* describes the date/time value when the order modification was entered into the database. It is set by the XETRA trading system with a precision of up to 10ms. Figure 3.2 gives a short overview of trading activity in the trading week based on the number of order modifications and the *ModificationTimestamp*. Like the money traded volume histogram in Figure 3.1, Figure 3.2 shows the typical u-shaped histogram for most of the trading days with a strong activity at the beginning and at the end of each trading session. In contrast to the trading volume diagram the *ModificationTimestamp* histogram shows the number of incoming order modifications; thus it shows how stressed the connections to the trading system are. The number of order modifications per day deviates around the mean of 835,700 with a minimum of 723,700 on the January 10th and a maximum of 955,200 modifications on January 11th. The histogram also shows a discontinuous order modification flow — especially on 2005-1-11 during the Intraday Auction.

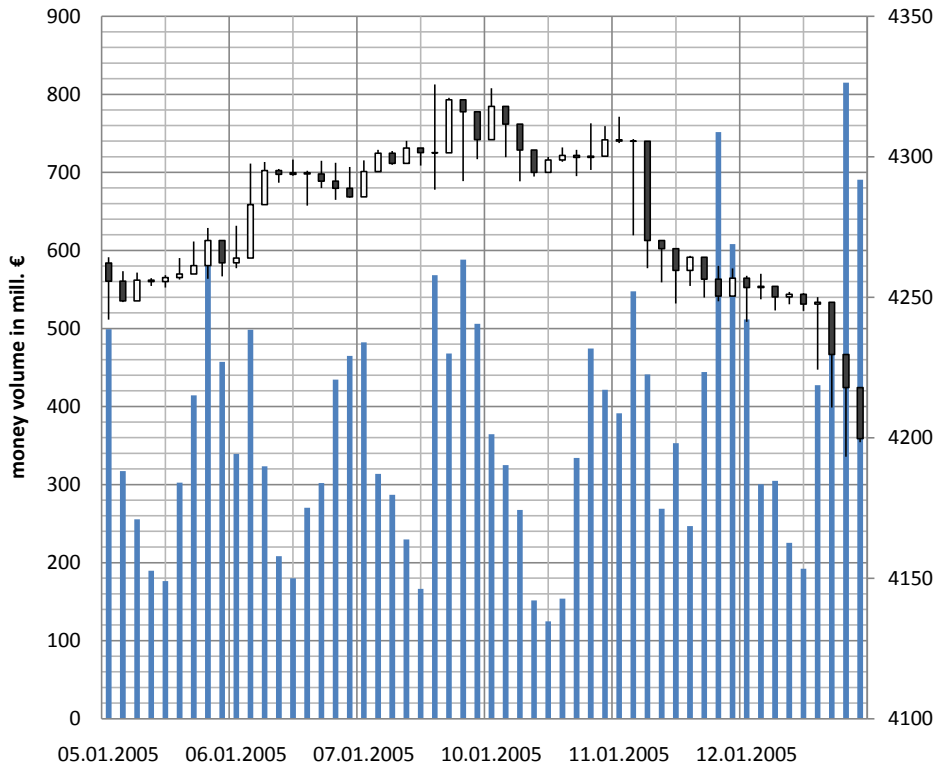

Figure 3.1: Movement of the DAX index and trading volume in the trading week between 2005-1-5 and 2005-1-12 aggregated to one hour
Movement of the DAX index indicated by candlestick chart;index levels are shown by the right vertical axis. 5 min. DAX intraday index data was obtained from [Md08] and aggregated.
Blue bars indicate Money volume traded (in million EUR); volume levels are shown by the left vertical axis. Volumes have been calculated and aggregated using the provided high-frequency dataset from Deutsche Börse.

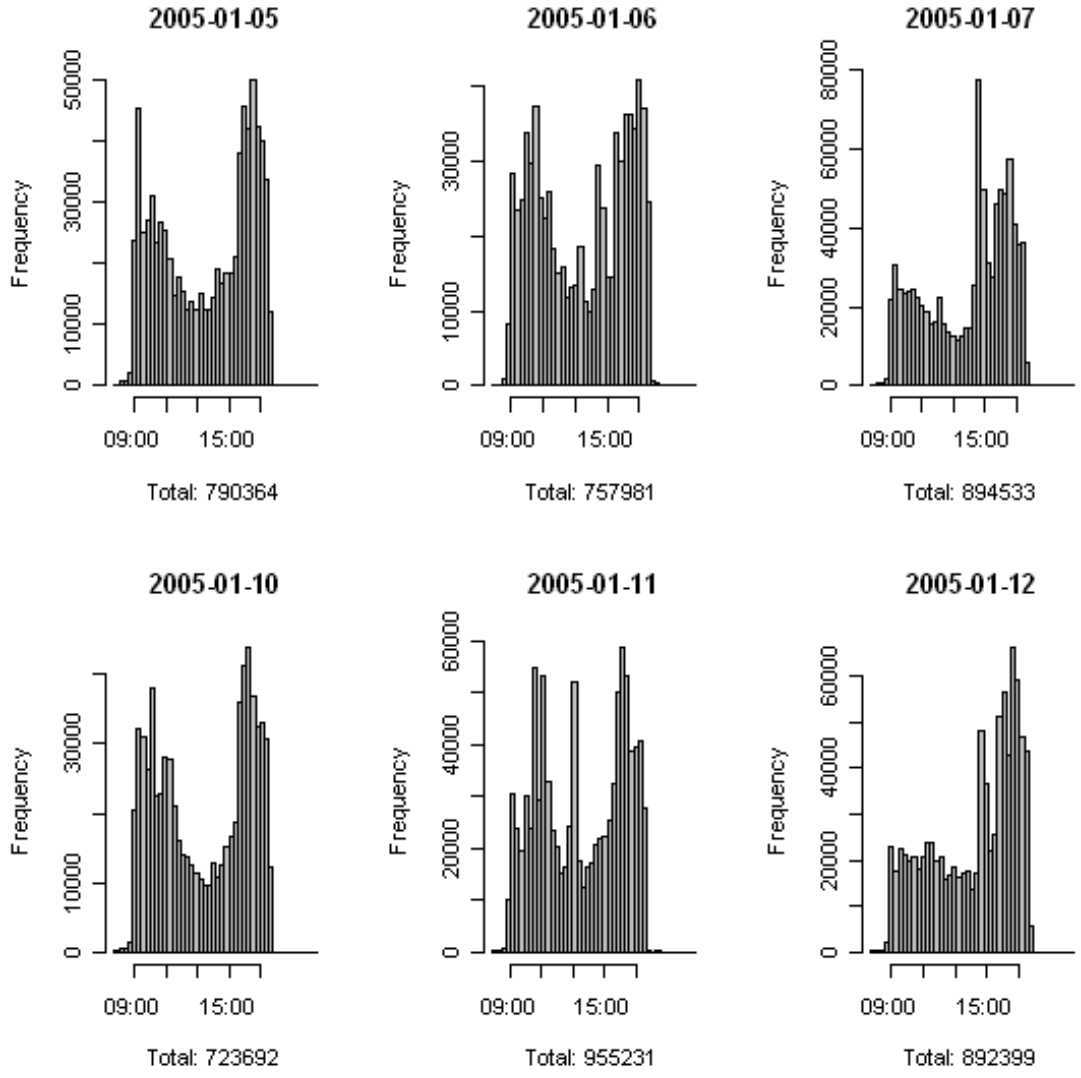

Figure 3.2: Histogram of order modifications by *ModificationTimestamp* attribute
Histograms were created in R with a variable bin size using the provided dataset. For each trading day between 2005-1-5 and 2005-1-12 a separate histogram has been generated indicating the frequencies of incoming order modifications to the XETRA trading system over time.

Stock Name	High	Low	VWAP	ADT	ADT %
ADIDAS-SALOMON AG O.N.	120.35	114.84	117.95	37.26	1.09%
ALLIANZ AG VNA O.N.	97.75	91	96.23	201.74	5.91%
ALTANA AG O.N.	46.67	44.17	45.39	27.27	0.8%
BASF AG O.N.	53.16	51.12	52.26	129.92	3.81%
BAY.HYPO-VEREINSBK.O.N.	17.71	16.22	17.02	120	3.52%
BAY.MOTOREN WERKE AG ST	34.91	33.55	34.29	84.78	2.48%
BAYER AG O.N.	24.59	23.47	24.02	124.97	3.66%
COMMERZBANK AG O.N.	16.34	15.32	16	79.78	2.34%
CONTINENTAL AG O.N.	49.2	47.65	48.43	42.14	1.23%
DAIMLERCHRYSLER AG NA O.N	36.35	34.86	35.7	157.26	4.61%
DEUTSCHE BANK AG NA O.N.	67.63	64.42	66.22	285.65	8.37%
DEUTSCHE BÖRSE NA O.N.	45.58	44.22	44.87	35.22	1.03%
DEUTSCHE POST AG NA O.N.	17.26	16.48	16.93	40.16	1.18%
DT.TELEKOM AG NA	16.88	15.96	16.46	411.95	12.07%
E.ON AG O.N.	67.97	65.92	67.05	168.64	4.94%
FRESEN.MED.CARE AG O.N.	59.09	57.1	58.12	14.04	0.41%
HENKEL KGAA VZO O.N.	67.32	64.12	66.22	25.61	0.75%
INFINEON TECH.AG NA O.N.	8	7.45	7.79	87.6	2.57%
LINDE AG O.N.	48.78	47.47	48.18	25.31	0.74%
LUFTHANSA AG VNA O.N.	10.8	10.44	10.65	33.13	0.97%
MAN AG ST O.N.	30.1	28.63	29.19	45.22	1.32%
METRO AG ST O.N.	42.23	39.15	40.98	80.09	2.35%
MUENCH.RUECKVERS.VNA O.N.	94.21	90.55	92.46	143.97	4.22%
RWE AG ST O.N.	43.7	40.73	42.8	171.65	5.03%
SAP AG ST O.N.	133.9	124.45	129.8	299.23	8.77%
SCHERING AG O.N.	56.76	53.28	54.51	97.49	2.86%
SIEMENS AG NA	62.88	61.04	62.1	248.6	7.28%
THYSSENKRUPP AG O.N.	16.68	16.14	16.35	49.95	1.46%
TUI AG O.N.	18.46	17.71	18.07	27.12	0.79%
VOLKSWAGEN AG ST O.N.	36.35	34.72	35.72	117.04	3.43%

Table 3.1: Price range and traded volume of DAX30 stocks traded on the XETRA system in the week between 2005-1-5 and 2005-1-12
All indicators have been calculated from the dataset of the analyzed trading week.
VWAP ... Volume Weighted Average Price
ADT ... Average Daily Turnover in million EUR
ADT % ... Share of trading done in respective trading week

Attribute	Section	Example Value
ModificationTimestamp	3.2.1	2005-1-5 15:30:21.110
OrderEntryTimestamp	3.2.2	2005-1-7 08:25:59.230
OrderExpiryDate	3.2.3	2005-1-10
ISIN	3.2.4	DE0007664005
Ordernumber	3.2.5	50120002058993926
AuctionTradeFlag	3.2.6	C
Ordertype	3.2.7	L
BuySell	3.2.8	B
Size	3.2.9	200
Price	3.2.10	98.3
Limit	3.2.11	95.4
ModReasonCode	3.2.12	1
Orderrestriction	3.2.13	F
Traderestriction	3.2.14	AU

Table 3.2: All order modification attributes defined in the orderbook database

3.2.2 OrderEntryTimestamp

The *OrderEntryTimestamp* attribute saves the date/time value when the order was entered into the database and is set by the trading system. This timestamp is the same for all modifications belonging to an order. When a new order is entered the *ModificationTimestamp* and *OrderEntryTimestamp* attribute value of the Insert order modification are the same. Further modifications that refer to an already entered order contain the same *OrderEntryTimestamp* as the first insert order. This is important as the *OrderEntryTimestamp* is also used to apply the Price-Time priority rule (see Section 2.2) and is therefore necessary for the sorting procedure in the orderbook if two orders have the same limit.

3.2.3 OrderExpiryDate

The *OrderExpiryDate* must be set by the trader at order insertion and indicates the maximum lifetime of an order with a precision of one day. It is the same for all modifications referring to an already inserted order but can also be changed by the trader during the trading session. Approximately 98.5% of all orders in the investigated dataset have the *OrderExpiryDate* set to the same day (also referred as 'good-for-day' orders) which means that the order will be deleted automatically at the end of the trading session.

Name	*ISIN*	Frequency
ALLIANZ AG VNA O.N.	DE0008404005	302,414
SAP AG ST O.N.	DE0007164600	286,185
MUENCH.RUECKVERS.VNA O.N.	DE0008430026	271,141
DEUTSCHE BANK AG NA O.N.	DE0005140008	268,666
SIEMENS AG NA	DE0007236101	257,374
DAIMLERCHRYSLER AG NA O.N	DE0007100000	234,130
DT.TELEKOM AG NA	DE0005557508	233,364
E.ON AG O.N.	DE0007614406	221,488
BASF AG O.N.	DE0005151005	219,028
RWE AG ST O.N.	DE0007037129	206,408

Table 3.3: ISIN codes of top 10 DAX Stocks in terms of order modification frequencies

3.2.4 ISIN

The *ISIN*[2] number is a 12-character alpha-numerical code to identify tradeable financial instruments on stock exchanges around the word. It typically starts with a country code with a length of 2 followed by a numerical code. In the investigated database the *ISIN* code is also a foreign key to the Instruments table which contains more detailed information about the traded instrument like e.g. the stock name (see more in Section 4.4.2). Most importantly the dataset has to be separated (or grouped) by the *ISIN* to analyze the market of a single stock. Table 3.3 shows the *ISIN* numbers of the ten most actively traded DAX stocks based on the number of order modifications on the XETRA system. Compared with the average daily turnover ranking in Table 3.1 which placed Deutsche Telekom in front of SAP and Deutsche Bank, the top 3 stocks in terms of entered order modifications are Allianz followed by SAP and Münchner Rückversicherung.

3.2.5 Ordernumber

The *Ordernumber* identifies each order entered into the trading system over its lifetime cycle. It is set by the trading system as a numeric code with a length of 18. A total of 2,347,387 distinct *Ordernumber* values, or equivalently orders, can be identified in the database. Compared with the total order modification number of 5,014,200 the average number of order modifications per order is approximately 2.14. Although *Ordernumber* values cannot be changed, modifications to an order which violate the Price-Time priority rule lead to a deletion followed by a re-insertion of an order and

[2] abbreviation for 'International Securities Identification Number'

therefore to a new *Ordernumber* value.

3.2.6 AuctionTradeFlag

The *AuctionTradeFlag* identifies the trading phase (Section 2.5) in which the order modification entered the system. As trading phases like Volatility Interruptions can occur at any time during the trading session this attribute can be used to identify the respective trading phase in the orderbook for reconstruction purposes. The *AuctionTradeFlag* attribute can have one of the following values:

`A` **Intraday Auction:** The order modification was entered during Intraday Auction (Section 2.5.2) between 13:00 and 13:02.

`C` **Continuous Trading:** The order modification was entered during the continuous trading phase (Section 2.5.3) between 9:00 and 13:00 or 13:02 and 17:30.

`F` **Closing Auction:** The order modification was entered during the Closing Auction (Section 2.5.2) between 17:30 and 17:35.

`O` **Opening Auction:** The order modification was entered during the Opening Auction (Section 2.5.2] between 8:50 and 9:00.

`V` **Volatility Interruption:** The order modification was entered while a Volatility Interruption (Section 2.5.4) occurred. As already mentioned in Section 2.5.4 no fixed start/end time is defined.

`P` **Pre-Trading:** The order modification was entered during the Pre-Trading phase (Section 2.5.1) between 7:30 and 8:50.

`R` **Post-Trading:** The order modification was entered during the Post-Trading phase (Section 2.5.1) between 17:35 and 20:30.

Table 3.4 shows the Frequencies of the different *AuctionTradeFlag* values. It can be seen that most order modifications enter the system during the continuous trading phase, followed by a Closing Auction and an Opening Auction. In terms of order modification inflow Pre/Post-Trading phases and Volatility Interruptions just play a minor role. Also the start- and end-time ranges of trading phases are shown, calculated using the orderbook dataset. It can be seen that the time of trade phase changes is not as exact as described in [Gro03] (see also Section 2.5). Only one volatility interruption was detected at Allianz stock on 2005-1-11 which lasted for approximately 4 minutes.

AuctionTradeFlag	*Frequency*	startMin	startMax	endMin	endMax
C Continuous	4,885,137	09:00:02	09:00:07	17:30:03	17:40:07
F Closing Auction	64,728	17:30:00	17:30:01	17:35:32	17:35:35
O Opening Auction	31,883	08:50:01	08:50:08	09:00:39	09:03:52
P Pre-Trading	11,437	08:00:02	08:00:06	08:49:59	08:50:04
R Post-Trading	10,032	17:35:03	17:35:07	20:10:15	20:29:58
A Intraday-Auction	9,767	13:00:00	13:00:01	13:02:32	13:02:34
V Volatility Interruption	1,216	13:05:44*	13:05:44*	13:09:10*	13:09:10*

Table 3.4: Frequencies of order modifications by *AuctionTradeFlag* attribute including trading phase start- and end-times
Start- and end-times have been determined by *AuctionTradeFlag* and *ModificationTimestamp* attributes. min/max start/end times have been filtered over all DAX30 stocks for investigated period between 2005-1-5 and 2005-1-12
startMin . . . minimum trading phase start-time
startMax . . . maximum trading phase start-time
endMin . . . minimum trading phase end-time
endMax . . . maximum trading phase end-time
* . . . only one volatility interruption occurred in investigated time period on 2005-1-11 at the Allianz stock. Respective start- and end-times are given.

3.2.7 Ordertype

This Attribute describes the specific order type and is fixed (except for Market-to-Limit orders) during order lifetime. All different order types have been previously described in Section 2.3. The following *Ordertype* flags can be identified in the database:

L **Limit Order:** Orders with a fixed limit price as described in Section 2.3.1

M **Market Order:** Orders which are executed at the best price, see Section 2.3.2

I **Iceberg Order:** Limit orders with a visible peak size and a hidden volume, see Section 2.3.3. Although Iceberg orders are only visible as limit orders to market participants they are identified in the database with the I attribute value.

T **Market-to-Limit Order:** Order is inserted as a Market-to-Limit Order T and at first matched like a Market Order (against the best limit). Afterwards it even in the database changes to a Limit order L.

Table 3.5 gives an overview of the order modification frequencies by *Ordertype*. It can be clearly seen that Limit orders are the most frequent orders in the XETRA orderbook, followed by Market orders. Although Iceberg order modifications just represent a small share they are typically inserted with a high (hidden) volume.

Ordertype		Frequency
L	Limit	4,768,834
M	Market	132,825
I	Iceberg	105,409
T	Market-to-Limit	7,132

Table 3.5: Frequencies of order modifications by *Ordertype*

3.2.8 BuySell

The *BuySell* attribute distinguishes buy and sell orders and cannot be changed over order lifetime. It only holds one single character for buy B and sell S orders, respectively. In the dataset under investigation slightly more buy (2,564,342) than sell (2,449,858) order modifications could be found; they represent approximately 51% of all order modifications.

3.2.9 Size

The *Size* attribute stores integer values set by the trader and has different meanings depending on the *ModReasonCode* value (see Section 3.2.12).

Insert Modification: In case of an order insertion (*ModReasonCode* 1 or 101) the size attribute shows the volume to be inserted into the book.

(Partial-)Execution: If an order is fully executed or partially executed (*ModReasonCode* 1 or 5) the size attribute indicates the executed volume.

Modification: The inserted order volume can also be reduced by an order Modification (*ModReasonCode* 2). In case of an order Modification the *Size* indicates the open order volume to be reduced. Volume increases by order Modifications while letting the *OrderEntryTimestamp* unchanged could lead to a violation of the Price-Time priority rule and are not allowed. Therefore a later increase of the volume has to be done by an Insertion (*ModReasonCode* 1) of a new order.

Deletion: Finally, in case of deletions (*ModReasonCode* 3,103) the *Size* indicates the rest open order volume to be deleted from the orderbook.

3.2.10 Price

The *Price* shows the actual execution price of a specific orders (usually greater than zero) and is determined by the trading system. Like the *Limit* (Section 3.2.11) the

Price is a floating point number with two decimals. It is relevant for order modifications with the *ModReasonCode* 4 or 5, see also Section 3.2.12. Considering the ordertype, the execution price has to be best for Market orders (best bid/ask) or at least the limit price (Section 3.2.11) for limit orders (see also Section 2.3).

3.2.11 Limit

The *Limit* attribute specifies the limit price of an order in EUR. It is a floating point number with two decimals. Market orders always have a *Limit* of zero, Limit orders and Iceberg orders must have a *Limit* greater than zero. In case of Market-to-Limit orders the *Limit* attribute indicates the price of the Limit order which is generated after the previous Market-to-Limit order has been partially executed (see Section 2.3.3).

3.2.12 ModReasonCode

The *ModReasonCode* describes the reason for insertion of each database entry and is therefore one of the most important attributes. Generally, each modification to an order can be described by the *ModReasonCode*. It is a small integer number with a range from 1 to 103[3].
The following *ModReasonCode* values can be identified in the dataset:

1 **Insert:** An Insert entry marks the beginning of the order lifetime cycle. Each order must have exactly one Insert order modification. In case of a Limit-, Market or Market-to-Limit - order the *Size* shows the entire volume of the specific order. Iceberg orders only show up its peak volume at insertion. *ModificationTimestamp* and *OrderEntryTimestamp* are equal for each Insert database entry.

2 **Modification:** A Modification can occur at any time during the order lifetime cycle. If the Modification leads to changes according to the Price-Time priority rule (e.g. change of order limit) or has a negative impact on the priority of other orders in the orderbook (e.g. volume increase) a new order (with a new *Ordernumber* value) has to be entered into the book. Most commonly a volume decrease is changed under the same *Ordernumber* where the *Size* attribute shows the volume to be decreased.

3 **Deletion-by-User:** Deletion-By-User is a very common modification reason at the end of the order lifetime cycle and indicates a user initiated deletion of an order. The *Size* attribute (Section 3.2.9) shows the rest volume of an order to be deleted.

[3]The later introduced modification reasons System Insert 201 System Delete 203 and System Execution 204 extend the range to 204

4 **Execution:** The Execution entry marks the end of the order lifetime cycle and shows that the whole volume of an order has been executed; the *Size* attribute (Section 3.2.9) gives the volume of the last execution.

5 **Partial Execution:** A Partial Execution can occur at any time during order lifetime. It shows that only parts or the order volume have been executed while the order is still remaining in the book with an unexecuted rest volume.

6 **Deletion-by-System:** The Deletion-by-System modification ends the order lifetime cycle. Most commonly it is entered when a specific order restriction has been triggered like, for example, by the *OrderExpiryDate* (Section 3.2.3).

101 **Technical Insert (due to modified Trade Restriction):** The *TradeRestriction* attribute (Section 3.2.14) of an order can generally be changed and actually requires the XETRA trading system to delete and re-insert the order. In order to recycle all attribute values the order is preliminary deleted by a Technical Delete modification (*ModReasonCode* 103) and immediately re-inserted. The re-insertion is done by a Technical Insert modification 101 which is similar to a normal Insertion 1 and leaves all attributes, except the *TradeRestriction*, unchanged.

103 **Technical Delete (due to modified Trade Restriction):** As indicated above a Technical Delete has to be done if the *TradeRestriction* of an order has been changed. A Technical Delete modification is similar to a Delete modification 3 and temporarily removes an order from the orderbook.

Order modifications like Insertions 1, Modifications 2 and Deletions 3 are generally initiated by the trader. By contrast, (Partial)Executions (5,4), Deletions-by-System 6 and Technical Insert/Deletes (101,103) are generated by the trading system.
Table 3.6 lists the frequencies of *ModReasonCode* values in the orderbook. Insert modifications are obviously most common, followed by Deletions-by-User and Executions. By the extraction of *ModReasonCode* values from order modifications belonging to the same order the entire lifetime cycle of an order can be described. In Section 3.3 and Section 5.2.1 the *ModeReasonCode* induced lifetime cycle will be further specified.

3.2.13 Orderrestriction

The *Orderrestriction* attribute sets execution restrictions for a specific order. All different order restrictions have already been described in Section 2.4. The *Orderrestriction* is set by the trader and requires only one character in the database.
The *Orderrestriction* can have one of the following values:

\- **No Restriction:** No order restriction has been set.

Modification Reason	Frequency
1 Order Insert	2,284,628
2 Order Modification	36,165
3 Deletion By User	1,626,896
4 Order Execution	675,232
5 Order Partial Execution	373,760
6 System Deletion	16,597
101 Technical Insert	461
101 Technical Deletion	461
Total	**5,014,200**

Table 3.6: Order modification frequencies by modification reason codes (attribute *ModReasonCode*)

F **Fill-or-Kill:** The order has been entered by the trader with the Fill-or-Kill flag. If the entire volume cannot be executed it will be deleted.

I **Immediate-or-Cancel:** The Immediate-or-Cancel flag forces the trading system to test which parts of the order can be executed. In contrast to the Fill-or-Kill flag only the unexecuted volume will be deleted.

S **Triggered Stop Order:** The Triggered Stop Order indicates that a stop limit (entered by the trader) has been reached which triggered an order insertion by the trading system. Triggered Stop Orders can either be Market orders (Stop Market order) or limit orders (Stop Limit order).

Table 3.7 shows the frequencies of all *Orderrestriction* values in the dataset. It can be seen that order restrictions are not widely used in the dataset. Considering the subset of order modifications with the *Orderrestriction* attribute set, Immediate-or-Cancel restrictions are the most common, followed by Triggered Stop orders and Fill-or-Kill orders.

3.2.14 Traderestriction

Sets the Trading Phase (Section 2.5) in which the order is visible in the orderbook and ready for execution. The following *Traderestriction* values are possible:

\- **No Traderestriction:** The order can be executed in every trading phase.

AU **Auction only:** The order can only be executed in any auction (Opening-, Intraday- or Closing Auction), see Section 2.5.2.

Orderrestriction		Frequency
-	No Orderrestriction	4,887,209
I	Immediate-or-Cancel	116,984
S	Triggered Stop Order	9,279
F	Fill-or-Kill	728

Table 3.7: Frequencies of order modifications by *Orderrestriction* Attribute

Traderestriction		Frequency
-	No Restriction	4,997,396
AU	Auction Only	9,637
CA	Closing Auction Only	3,764
MT	Main Trading Phase	2,613
OA	Opening Auction Only	790

Table 3.8: Frequencies of order modifications by *Traderestriction* attribute

CA **Closing Auction only:** The order can only be executed during the Closing Auction (Section 2.5.2).

OA **Opening Auction only:** The order can only be executed during the Opening Auction (Section 2.5.2).

MT **Main Trading Phase only:** The order can only be executed during the Main Trading Phase (see Section 2.5) from 9:00 to 17:30. This restriction has no effect since 2003-11-1, when trading hours have been shortened on the XETRA trading system from 20:00 to 17:30.

Table 3.8 shows the frequencies of entered trade restrictions obtained from the dataset. Only very few orders use this restriction type; if used, the Auction Only restriction is the most common, followed by the Closing Auction Only and the Main Trading Phase.

3.3 Event Code Sequences

As already mentioned in Section 3.2.12 the lifetime cycle of an order can be described through various database records or order modifications which also contain the important *ModReasonCode* attribute. The *ModReasonCode* values form characteristic

event code sequences over time. The sequence order is chronological as defined by the *ModificationTimestamp*.
A sequence generally has to start with an order Insertion 1 which shows the exact time and size of the order entering the book. After Insertions, Modifications 2 and Partial Executions 5 are possible while the order still remains in the book. The order lifetime ends either with a Deletion(3 or 6) or an Execution (4).
Figure 3.9 shows the 15 most frequent event code sequences from the raw XETRA database. Short event code sequences only containing two *ModReasonCode* values form the lion share of the order dataset under investigation. As the most frequent 1–4 sequence could be seen as a quite 'normal' order (Insertion followed by an Execution) the striking 1–3 sequence (Insertion followed by Deletion) on the second place shows a high share of deleted orders. Most of the other listed orders basically contain Partial Executions (–5–) in between and end with either Executions (–4) or Deletions (–3). Also a lot of fragmentary event code sequences consisting of only one *ModReasonCode* value can be identified. The strategies how to handle obviously broken sequences like a single 3 'sequence' (deletion without insertion), a single 1 (insertions without deletions or executions) or a single 4 (executions without insertions) will be discussed in Section 5.2 in more detail.
To get a better understanding of what is actually happening in the database during different orderbook events and which records are generated we will revisit the order matching examples from Section 2.2.

3.3.1 Limit Order Example revisited

This section refers to the Limit order matching example from Section 2.3.1. As already mentioned the sell Limit order which enters the orderbook at Time T=5 with a volume of 200 and a limit of 98.2 is matched against the best two buy orders.
Table 3.10 shows the attribute values and database records of the sell order. On the left side of the table one can find the attribute values which are fixed for all database records generated by the sell order. The right side shows the database entries with varying attribute values like the *ModificationTimestamp* (*MT*), *Size*, *ModReasonCode* (*MRC*) and the executed *Price*. For simplicity reasons the *ModificationTimestamp* and the *OrderEntryTimestamp*, which are actually date/time attributes with a precision up to 10 milliseconds, are displayed as integer values. As can be seen on the right side of the table, three database records have been generated for the sell order. Immediately after insertion the order is partially executed against the best buy order (*Price* 98.30) and fully executed against the second best buy order (*Price* 98.20). The *ModificationTimestamp* value is the same for all records with T=5. The event code sequence of this order results therefore as 1–5–4.

Event Code Sequence	Frequency
1 – 3	3,002,564
1 – 4	1,042,492
1 – 5 – 4	255,060
1 – 5 – 3	126,627
1 – 5 – 5 – 4	103,136
1 – 2 – 3	64,965
1 – 5 – 5 – 5 – 4	51,205
3	38,209
1 – 5 – 5 – 3	32,516
1 – 5 – 5 – 5 – 5 – 4	28,350
1	27,983
1 – 6	22,324
1 – 5 – 5 – 5 – 5 – 5 – 4	17,787
4	15,499
1 – 5 – 6	14,979

Table 3.9: The 15 most common event code sequences found in the dataset from 2005-1-5 to 2005-1-12
ECS . . . Event Code Sequence
Sequences have been obtained using the implemented *Orderbook Engine* library. *ModReasonCode*s have been grouped by *Ordernumber* and sorted chronologically by the *ModificationTimestamp*. If the *ModificationTimestamp* of order modifications belonging to the same order are equal, modification reason codes have been sorted with respect to their causal order (e.g. Insert before Modification).

3.3.2 Market Order Example revisited

In this Section the database entries generated by the Market order example from Section 2.3.2 will be discussed in more detail.
Table 3.11 shows the attribute values of the entries for the sell Market order analogous to Table 3.10. The event code sequence of this order results as 1 – 5 – 5 – 5 – 4.

Fixed Attributes	*MT*	*Size*	*MRC*	*Price*
ISIN: DE0007664005	5	200	1	0.00
Ordernumber: 50120002058993926	5	100	5	98.30
Ordertype: Limit L	5	100	4	98.20
BuySell: Sell S				
Limit: 98.2				
OrderEntryTimestamp: 5				
AuctionTradeFlag: Continuous C				
Orderrestriction: No Restriction -				
OrderExpiryDate: 2005-1-12				

Table 3.10: Limit order example revisited

Fixed Attributes	*MT*	*Size*	*MRC*	*Price*
ISIN: DE0007664005	5	1000	1	0.00
Ordernumber: 50120002058994944	5	100	5	98.30
Ordertype: Market M	5	300	5	98.20
BuySell: Sell S	5	100	5	98.00
Limit: 0.00	5	500	4	98.00
OrderEntryTimestamp: 5				
AuctionTradeFlag: Continuous C				
Orderrestriction: No Restriction -				
OrderExpiryDate: 2005-01-12				

Table 3.11: Market Order Example Revisited

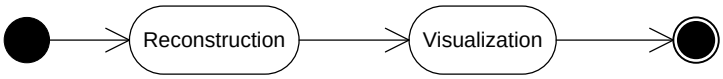

Figure 4.1: Basic Process Overview

4 Software Architecture

This chapter gives an overview of the architectural design of this software project. After a discussion of design goals in Section 4.1 soft- and hardware requirements are discussed (Section 4.2). Further a package overview is given in Section 4.3. While Section 4.4 sketches the database architecture each implemented software package like *Orderbook Engine* (Section 4.5), *Dataset Generator* (Section 4.5) and *Orderbook Visualization* (Section 4.7) will be discussed.
To improve readability some typographic definitions have been introduced for the rest of this thesis. Package names and namespaces are written in *San Serif italic*, class/object names and events in San Serif, function names and object values in `teletype font face` and Property/Attribute names in *italic*.

4.1 Design Goals

Generally the architectural design goal of this software project was the creation of a transparent and flexible object model to support a wide range of possible extensions and improvements. The basic functionality can be summarized as a 2-step procedure. The first step is the reconstruction process to integrate and clean the raw dataset. It is described in Chapter 5 in more detail. Separated from that the orderbook visualization (Chapter 6) is done afterwards in a second step. Although it would have also been possible to integrate the data cleaning procedure directly in the orderbook visualization process (and do the data cleaning procedure online) the separation between the reconstruction and the visualization process leads to a better performance of the orderbook visualization engine. The basic process overview of the implemented software is shown in Figure 4.1.
Performance has also been a big implementation issue to ensure a better user experience — particularly for the orderbook visualization part. Furthermore an intuitive user interface should make the interactive analysis of orderbook data as easy as possible.

Altogether the user interface design should allow the exploration of orderbook data and the visualization of the 'orderbook landscape' — just as simple as geo-browsing with applications like Google Earth.
To provide the above mentioned functional requirements it was decided that the C# programming language would be the 'language of choice'. Although (unmanaged) C++ can bring some improvements in terms of performance C# using the .NET framework supports a type-save runtime environment and automatic garbage collection — thus a more comfortable and faster development environment. As this software project is heavily database-dependent the feature rich and easy to use database connectivity libraries (ADO.NET) have been another strong argument for the .NET environment. Last but not least the recently introduced Windows Presentation Foundation (WPF) which is supported by .NET 3.5 enables the development of feature rich and scalable graphical representations (including accelerated 3D representations) and leaves enough room for further improvements of the visualization engine.

4.2 Requirements

4.2.1 Software

The complete software project has been implemented with Microsoft Visual Studio 2008 (VS2008) in the C# programming language. There is also a free VS2008 C# Express Edition [Mic08b] available to develop and compile the entire project. The type-safe .NET Environment 3.5 is necessary to execute the packages and is available under [Mic08a]. It is restricted to the Windows XP, Windows Vista and the Windows Server versions 2003 or 2008 operating systems. The Mono - Project [Mon08] by Novell also provides a free implementation of .NET - libraries under Linux but does not support .NET 3.5 yet.
The database has been integrated to the Microsoft SQL Server 2005 (see also Section 4.4.1) which is also available as a free Express Edition under [Mic05]. Due to the generic implementation of the database connectors the integration to other servers (like MySQL, PostgreSQL, Oracle) is also possible but has not been tested yet.

4.2.2 Hardware

The minimum hardware requirement is an Intel Pentium 3 (or equal) with 1GB RAM whereas a Pentium 4 with 2GB RAM is recommended. The orderbook visualization software needs approximately 8MB of Disk Space — the database of the analyzed DAX30 dataset has a size of 1.38GB.

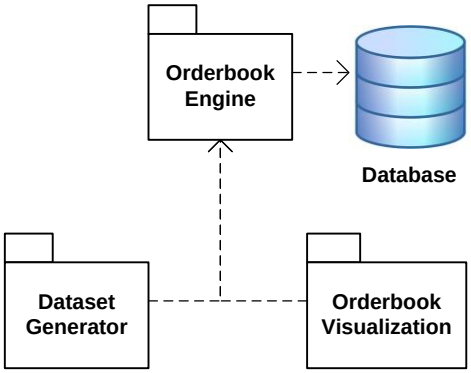

Figure 4.2: Overview of Software packages and database
The *Dataset Generator* and the *Orderbook Visualization* package both require the *OrderBookEngine* library. The *OrderBookEngine* also implements functions to directly access the SQL database.

4.3 Package Overview

The entire software project consists of three separated packages which provide the specified functionality already mentioned in Section 4.1. The separation has been done in a quite straightforward manner to cleanly divide the basic functionality from the user interface. Figure 4.2 provides a basic overview of the implemented packages. It shows the *Orderbook Engine* package which is required by the *Dataset Generator* and the *Orderbook Visualization*. The *Orderbook Engine* supports the core functionality of this project (especially for cleaning and reconstruction purposes) and encapsulates the order object model. Further it includes connectors to access the XETRA orderbook database. The *Dataset Generator* provides a user interface to clean selected parts of the dataset easily. The entire visualization of orderbook data is done by the *Orderbook Visualization* program.

4.4 Database Architecture

4.4.1 Data Integration

The integration of the original Microsoft Access Database which contains all DAX30 stocks in the time period between 2005-1-5 and 2005-1-12 has been done with the SQL Server Integration Services (SSIS) package which is included in the standard version of Microsoft SQL Server 2005. At first the Access data import wizard was used to create a basic import package. The raw Access database had a few attributes with wrongly assigned data types (mostly string types) which consumed too much

Attribute	AT(Length)	CT(Length)
ModificationTimestamp	Text(255)	DateTime
OrderEntryTimestamp	Text(255)	DateTime
OrderExpiryDate	DateTime	DateTime
ISIN	Text(255)	nvarchar(12)
Ordernumber	Text(255)	nvarchar(18)
AuctionTradeFlag	Text(255)	nvarchar(1)
Ordertype	Text(255)	nvarchar(1)
BuySell	Text(255)	nvarchar(1)
Size	Double	int
Price	Double	float
Limit	Double	float
ModReasonCode	Text(255)	tinyint
Orderrestriction	Text(255)	nvarchar(1)
Traderestriction	Text(255)	nvarchar(1)

Table 4.1: Datatype conversions from Access raw database to Microsoft SQL 2005 database
AT . . . Access data type from raw data base
CT . . . Converted SQL type which has been used in integrated SQL 2005 database

disk space. This issue has been corrected in the Integration process. Table 4.1 gives an overview of the data type conversions done in the SSIS integration package. While the second column contains the original Access data type the third column list the converted SQL 2005 data types. Under [Bro08] and [Mic07] you can find the exact data type specifications for Access and MS SQL 2005 data types, respectively. Most of the target SQL data types have been chosen to preserve all information from the Access database. Only the *Traderestriction* (*TR*) attribute (see Section 3.8) has been shortened to a character length of one. Just the first *TR* character has been preserved in the SQL database, e.g. `MT` (stands for Main Trading Phase) has been changed to `M`. This conversion still makes a distinction between all *TR* values possible.
While the original Access Database consumed 1.7GB of disk space its size could be reduced to 1.38GB through all mentioned data type conversions.

4.4.2 Database Structure

The integration package has been further modified to automatically create the target database and tables including correctly assigned primary keys and relationships.

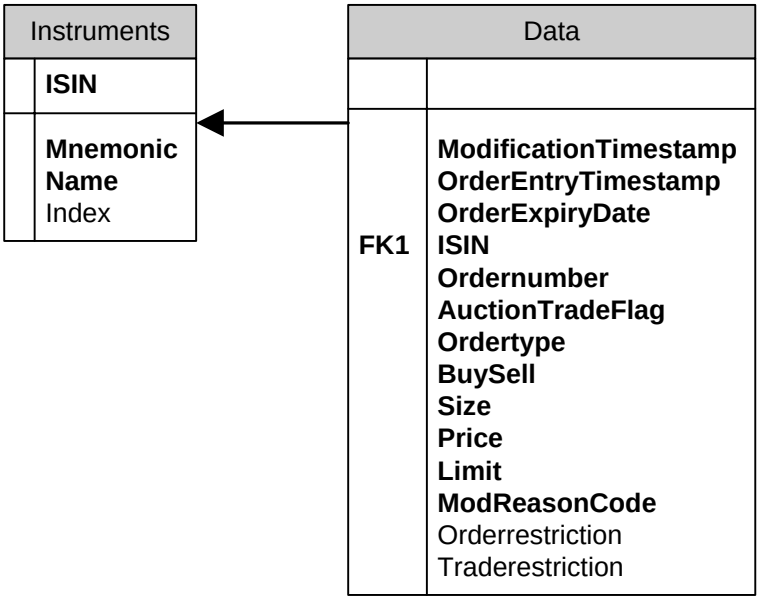

Figure 4.3: Diagram of integrated MS SQL 2005 database
Database consists of Instruments and Data table (stores order modifications). Data table is connected to Instruments through the *ISIN* attribute as foreign key, *ISIN* is also the primary key of the Instruments table. Bold attribute names indicate that they are not nullable.

The database diagram of the completely integrated dataset (drawn in Microsoft SQL Server Management Studio 2005) is shown in Figure 4.3.
As can be seen in Figure 4.3 the integrated database has a quite simple structure. The entire dataset is stored in only two tables that contain general descriptions about the instruments and the order modifications from the XETRA orderbook.
The Instruments table contains information regarding the underlying stock like the *ISIN* code, the *Long Name* and the *Mnemonic* of the stock.
The Data table contains all order modification attributes already mentioned in Section 3.2. It also stores the *ISIN* as a foreign key (FK1) from the Instruments table.

4.5 Orderbook Engine

The *Orderbook Engine* represents the core of this project. Built as a dynamic link library (dll) it encapsulates the complete order object model, the database access and all necessary data structures to reconstruct the orderbook. Figure 4.4 gives an

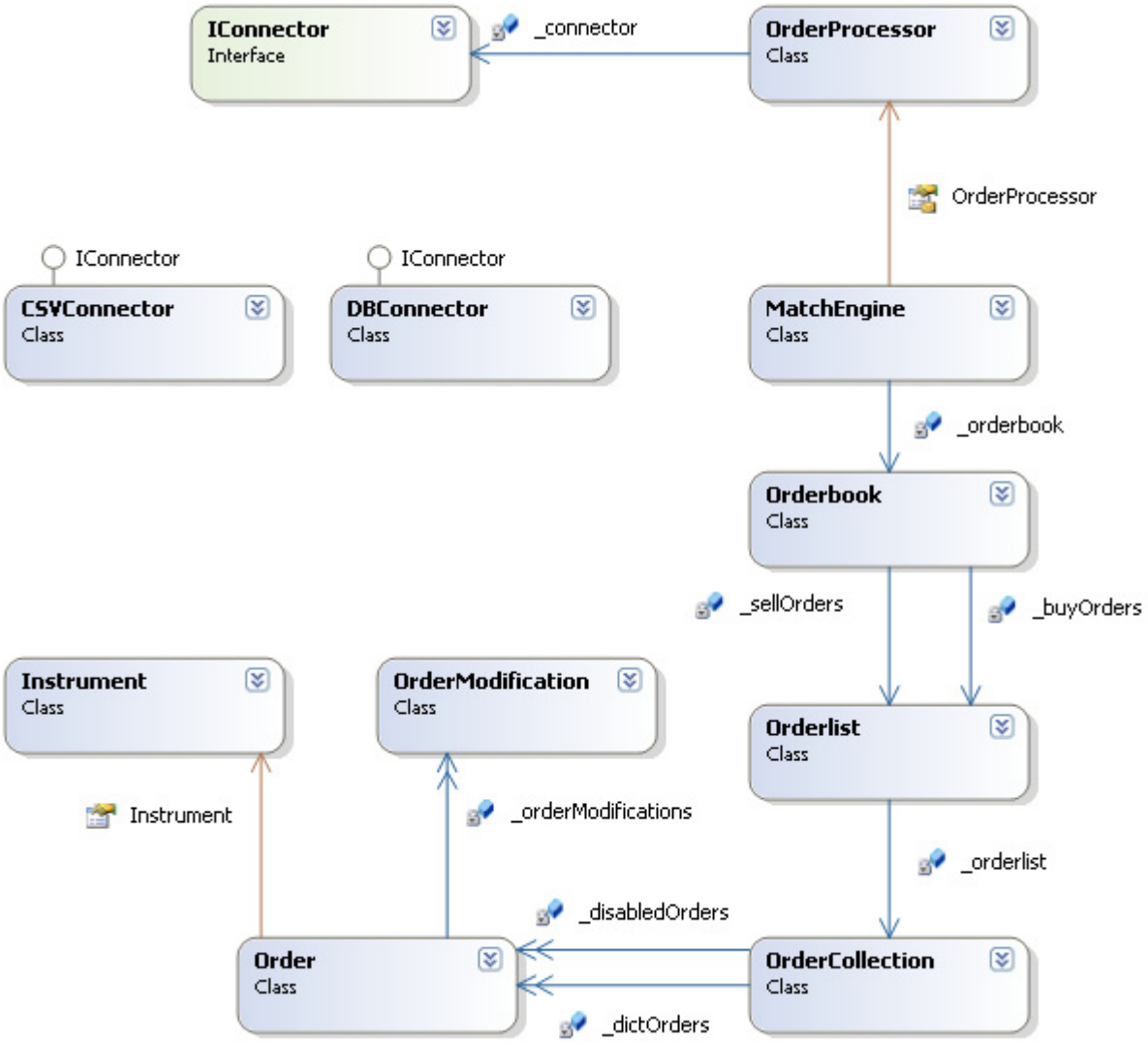

Figure 4.4: OrderBookEngine Class Diagram

overview of the basic *Orderbook Engine* Class Diagram.

4.5.1 Order Object Model

The order object model is represented by the classes Order, Instrument and Ordermodification. An order includes a single instance of the Instument class and stores multiple Ordermodification objects in a type-safe List Collection. The Instrument class contains properties like the *Name* and the *ISIN* of the traded stock. An Ordermodification object basically stores all attributes already described in Section 3.2.

4.5.2 Data Access

The complete data connection logic is implemented in the DBConnector and the CSVConnector class. While the DBConnector enables generic database access the CSVConnector imports data from comma separated value (CSV) files. Both classes can be accessed through the implemented IConnector interface which supports the extension to other data import classes. All connectors must implement the a `NewOrdersIterator` function. It provides the generation of Ordermodification objects through an iterator and can be accessed using a foreach-loop.

4.5.3 Orderbook Reconstruction

The classes involved in the orderbook reconstruction process are the MatchEngine, Orderprocessor, Orderbook, Orderlist and the Ordercollection classes.
The MatchEngine manages the whole reconstruction process including the static and dynamic orderbook reconstruction. It is mainly accessed by the *Dataset Generator* user interface. It includes instances of the Orderprocessor and the Orderbook classes.
The Orderprocessor reads data through a class which has implemented the IConnector interface. As a first reconstruction step it creates Order objects in 'raw mode'. In this context raw mode implies that Ordermodifications are just added to the Orders but generated Orders are not checked for consistency. The second step is the static order reconstruction where Orders are checked for static validity considering their event code sequences. The `Repair` function is called for each Order instance which has no valid event code sequence.
The Orderbook class is part of the dynamic order(book) reconstruction process. It contains all necessary data structures to store and match incoming Order(modification)s. Thereby it is possible to reconstruct the entire state of the original XETRA matching engine. Through replaying entire trading sessions it is possible to detect all kinds of inconsistencies in the dataset. The Orderbook class stores two Orderlist objects to manage buy and sell orders, respectively. The entire order reconstruction process will be described in Chapter 5 in more detail.

4.6 Dataset Generator

The *Dataset Generator* is an executable software package and provides a user interface to interactively control the entire reconstruction process. As the reconstruction of the complete dataset takes quite a lot of time the generator also allows the reconstruction of a selected subset. It was also designed to extract specific orderbook features from the dataset.
The *Dataset Generator* basically consists of one main form — its complete functionality is therefore event driven through delegates[1]. The *Orderbook Engine* library which actually contains the complete reconstruction logic is required to execute the *Dataset Generator*. Simply put, the generator instantiates all necessary database connection objects and passes them as arguments to the MatchEngine which is responsible for the actual reconstruction process as has been mentioned in Section 5.3.1. The MatchEngine reads the data from the specified source table, reconstructs it and writes the clean database table to the selected target table.

Using the generator user interface (see Figure 4.5) the reconstruction process can be described in a simple 3-step procedure: Firstly the source table has to be selected from the combo box. The connection strings to access a specified table are stored in the XML configuration file of the application. After the source has been selected all different stocks are listed in the result table below. Secondly all stocks which should be reconstructed are selected by the checkboxes on the left side. Finally the reconstruction process is started by pushing the 'Start' button.

4.7 Orderbook Visualization

The *Orderbook Visualization* program provides an interface to visualize the XETRA orderbook dataset and to do some basic exploratory data analysis. It implements a free zoom function as well as a filter function to visually emphasize specific orders. A basic class diagram of the *Orderbook Visualization* package is shown in Figure 4.6.
As already mentioned in Section 4.3 the *Orderbook Visualization* package requires the *Orderbook Engine* library to read order objects from the database. Mainly the IConnector interface and the Order class are required to read the reconstructed orderbook dataset (see top of Figure 4.6). The core functionality of the *Orderbook Visualization* package is represented by the Window1 class. It is responsible for the entire visualization logic and the user interface interaction. Various objects are stored in the Window1 class to encapsulate different parts of the visualization functionality: the ZoomState, Filter and TradingSession classes are the most important ones.
After reconstructed orderbook data has been read, Order objects are stored and aggregated in a special Hashtable data structure to support fast visualization. The Hashtable

[1] Delegates are quite similar to function pointers in C++ but have more features.

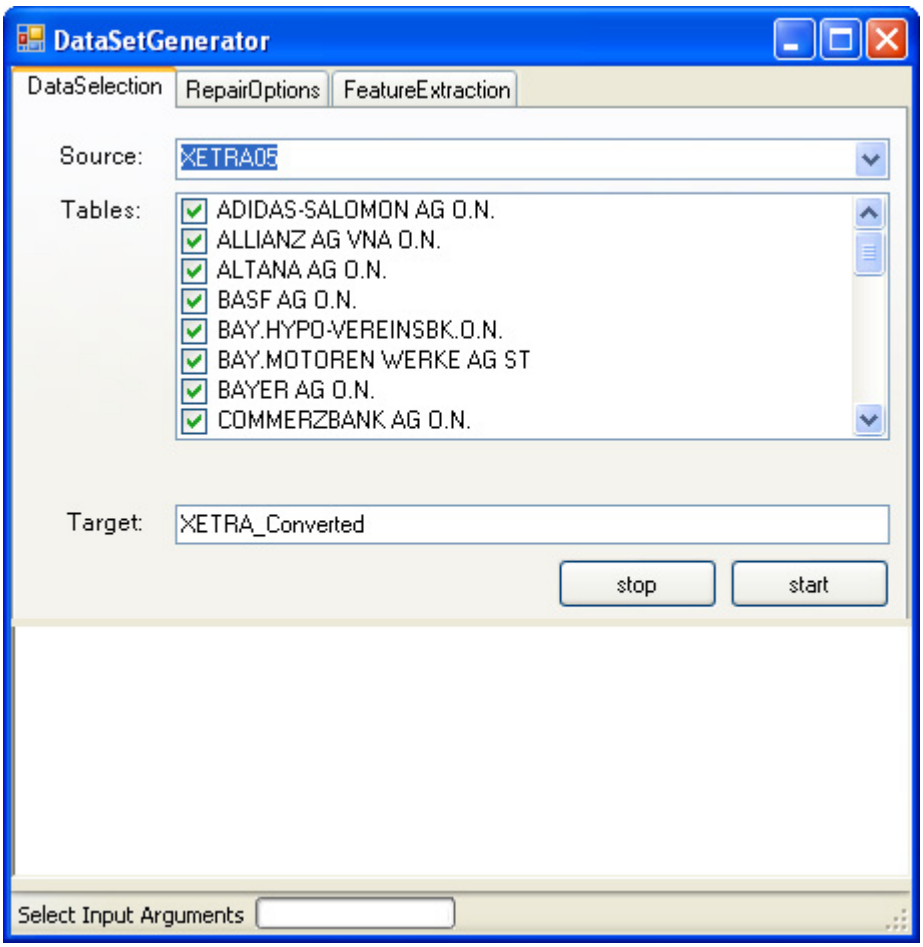

Figure 4.5: Dataset Generator User Interface

contains OrderRectangle objects as well as coordinates and is stored in the ZoomState object to allow fast navigation through the dataset. The data structure therefore stores a pixel representation of order objects.
Also a Filter class has been implemented to restrict the visualization of orders or OrderRectangles. It can be applied to various order attributes as well as aggregated OrderRectangle Attributes.
The TradingSession object allows the visualization engine to 'glue' orders from different trading sessions together. This aspect is especially important to display many trading sessions at once as big white spaces between different trading sessions are avoided.
Finally some helper classes have been implemented like the Statistics or the FileExport

class. The Statistics class is mainly used to calculate quantiles of distributions and is used to filter numerical attributes. The FileExport class encapsulates the entire logic which is needed to export orderbook visualizations to an image file. Please refer to Chapter 6 for more details about the orderbook visualization process.

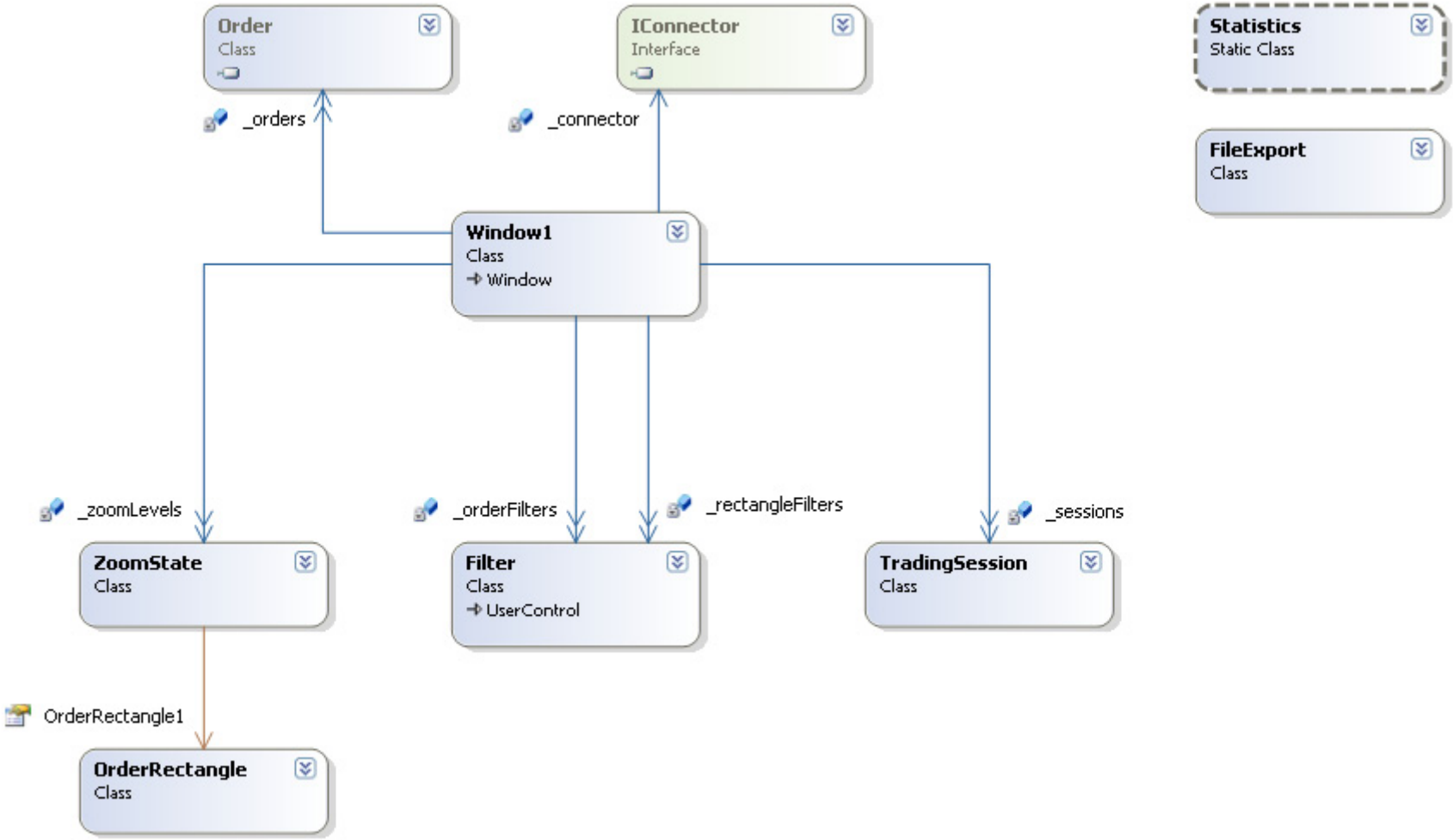

Figure 4.6: *Orderbook Visualization* class diagram

5 Reconstruction Process

This chapter describes the implemented reconstruction process for the analyzed XETRA dataset in more detail. Starting with an overview of the process (Section 5.1) the following sections further discuss the static order reconstruction (Section 5.2) and the dynamic order reconstruction process (Section 5.3) in more detail. Also the handling of hidden volumes (Section 5.2.3) is presented. Finally the results are given in Section 5.2.4.

5.1 Overview

As already mentioned in Chapter 3 the rebuilt orderbook dataset for all DAX30 - stocks from the XETRA trading system has a very high quality standard and allows an in-depth view of trading activity on the XETRA trading system. However, various inconsistencies in the dataset have been encountered and need to be corrected in order to provide a good (visual) representation of trading activity. Further a consistent dataset is necessary to reconstruct all states of the XETRA orderbook and to run various data analysis and simulation tools.
The reconstruction process has been implemented as a 2-step procedure as shown in Figure 5.1. First the static order reconstruction process has to correct errors within event code sequences. Already introduced in Section 3.3, the entire order lifetime cycle can be described through order modifications and, more importantly, sequences of modification reason codes. Even Table 3.9, where the most frequent sequences were shown, lists various incomplete sequences (missing Insertions, missing Executions, etc.). The static order reconstruction process implements strategies to repair these defect sequences.
The second step is represented by the dynamic reconstruction of orders. In contrast to the static order reconstruction which only corrects event code sequences on the order level, the dynamic order reconstruction monitors and corrects numerous dynamic errors on the order- as well as on the orderbook level. Therefore the order and orderbook states are reconstructed using order modifications from the static order reconstruction step while considering XETRA trading rules for order sorting and order execution. These rules have already been introduced in Chapter 2. Most important features monitored and corrected during the dynamic order reconstruction step are order size mismatches, bid/ask spreads and the trading turnover.

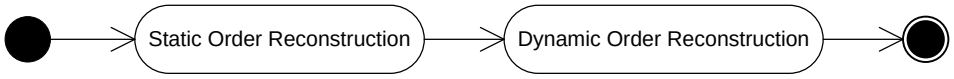

Figure 5.1: Basic Reconstruction Process Overview

5.2 Static Order Reconstruction

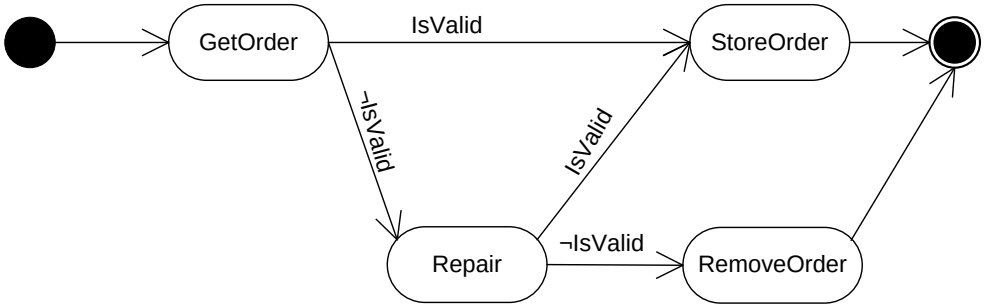

Figure 5.2: Static Order Reconstruction Process Overview

The goal of the Static Order Reconstruction Process handles the reparation of fragmentary order event code sequences. As already mentioned in Section 3.3 defect sequences occur quite often in the dataset and have to be repaired to ensure a proper functioning of the dynamic orderbook reconstruction.
The basic process overview of the static order reconstruction process can basically be described as an order validity check followed by an order repair function. Figure 5.2 shows a graphical process overview. After Data retrieval from the connector (function `GetOrder`) all orders are checked for validity (see Section 5.2.1). If the order is valid the loop continues with the next order and stores the valid order in the resulting order collection (implemented as a Hashtable). Else the `Repair` function of the invalid order is called (see Section 5.2.2). If not even the `Repair` function can reconstruct a valid event code sequence for the order it is deleted or equivalently not stored in the returned order collection.
In the following sections the order validity check (Section 5.2.1) as well as the order repair strategy (Section 5.2.2) will be discussed. Also a special treatment of Iceberg orders' hidden size is part of the static order reconstruction as it is necessary for the dynamic reconstruction process. It will therefore be discussed in Section 5.2.3. Finally

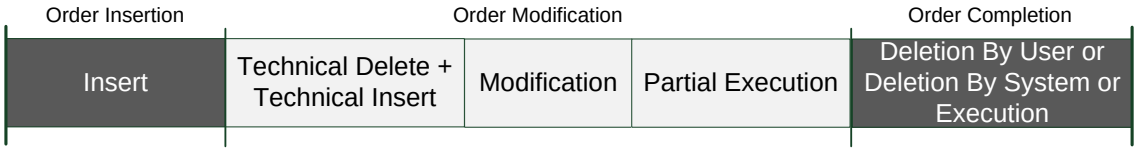

Figure 5.3: Basic order lifetime cycle described by order modification reasons
The entire order lifetime cycle is defined by 3 main categories of order modification reasons:
Order Insertion . . . Insert
Order Modification . . . Technical Delete + Technical Insert, Modification, Partial Execution
Order Completion . . . Deletion-by-User/System, Execution

the resulting dataset is presented in Section 5.2.4.

5.2.1 Order Validity Check

For the static order reconstruction process the validity check of order event code sequences represents a very important part. It is assumed that the event code sequence of any order follows a predefined structure. A definition of the structure used for the validity check is given in section *Order Structure*. Section *Invalid Order Statistic* gives a statistical overview of invalid orders.

Order Structure

This section defines the order event code structure used for the static order validity check. The basic structure is also sketched in Figure 5.3.
Any order has to start with an Insert order modification `1` (see Figure 5.3–Order Insertion). The Insert modification 'officially publishes' the order and all important attributes like the size, limit, etc. to the trading system and is therefore essential to represent the order properly.
The Insert modification can be followed by the combination of a Technical Deletion `103` and a Technical Insertion `101`, a Modification `2` or a Partial Execution `5` (see Figure 5.3–Order Modification). After any of those modifications the order still remains actively in the orderbook (or the trading system's memory).
In the investigated dataset Technical Deletions and Technical Insertions only occur in combination — a Technical Deletion is always followed by a Technical Insertion. They appear when the trader changes the *TradeRestriction* property of his already inserted order. This change leads to a Technical Deletion of the order; in effect the order is deleted from the orderbook but still held in memory of the trading system.

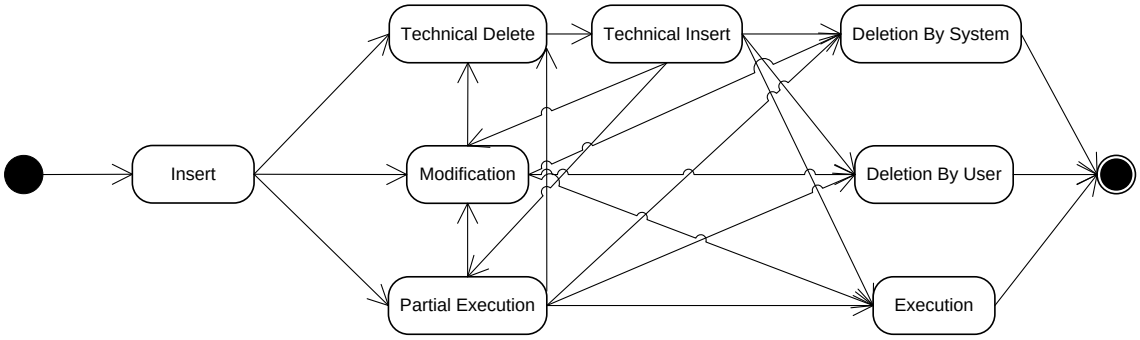

Figure 5.4: Order Modification State Diagram
Description of order modification reason sequences as state diagram. Following the arrows from start to end state all valid event code sequences can be generated.

A fulfillment of the changed *TradeRestriction* value triggers the Technical Insertion (or re-insertion) of the order with the same attributes. The entire *TradeRestriction* change process leaves the unique *Ordernumber* value unchanged for all modifications done.
An order Modification 2 done by the trader usually reduces the size of an already inserted order. A size decrease is allowed by the market model because the priority of the order remains unchanged and does not affect the priority of other orders in terms of the Price-Time priority rule.
Partial Executions 5 can occur during the order lifetime and just reduce the size/volume of the (still active) order.
The order lifetime cycle is completed after either a Deletion-by-System 6, a Deletion-by-User 3 or an Execution 4 modification (see Figure 5.3–Order Completion). After any of those modifications the order is deleted from the orderbook and not further involved in the matching/trading process.
A Deletion-by-System modification is an automatic deletion of an order by the trading system. It most often occurs due to the actual time exceeds the order expiry date (attribute *OrderExpiryDate*). Therefore most of the Deletion-by-System modifications are executed at the end of each trading session. In contrast to the automatic deletions a Deletion-by-User modification indicates a trader initiated deletion of an order.
The Execution modification executes the rest volume of an order against an order on the other side of the book.
To make the description of the order event code sequence structure more precise Figure 5.4 shows the structure in a state diagram.

I-ECS	Frequency
3	38,209
1	27,983
4	15,499
2 3	5,881
2 2 3	1,006
5 4	895
5 3	344
5 5 4	249
1 5	237
2 2 2 3	196

Table 5.1: Frequency of ten most common invalid order event code sequences in the dataset
I-ECS . . . Invalid Event Code Sequence

Invalid Order Statistic

Based on the basic order structure already described in section *Order Structure* the validity check identifies broken event code sequences. 91,406 orders have been identified to be erroneous by the validity check function. Considering all 2,347,387 Orders (or equivalently *Ordernumbers*) the share of invalid orders is 3.89%. Table 5.1 shows the frequencies of the 10 most common invalid event code sequences.

5.2.2 Order Repair Strategy

As could be seen in Section 5.2.1 about 4% of all orders contain fragmentary event code sequences. On the one hand invalid order sequences have to be avoided because they lead to an improper orderbook reconstruction. The orderbook reconstruction is necessary for the dynamic order reconstruction process and will be described in Section 5.3. On the other hand also invalid orders have to be preserved to ensure that the reconstructed orderbook is as close to reality as possible and no important order information is lost. Generally the order repair strategy is aimed to preserve orders which contain Execution 4 or Partial Execution 5 modifications in their sequences. Executed orders have a proven effect on the trading system and do not only affect the turnover statistic of the exchange but are also matched against orders on the other side of the orderbook. Therefore the reconstruction priority of executed orders is considered to be higher.
As you could already see from the most common invalid event code sequences in Table 5.1 the sequence errors can be separated in 3 categories: *Missing Insert* mod-

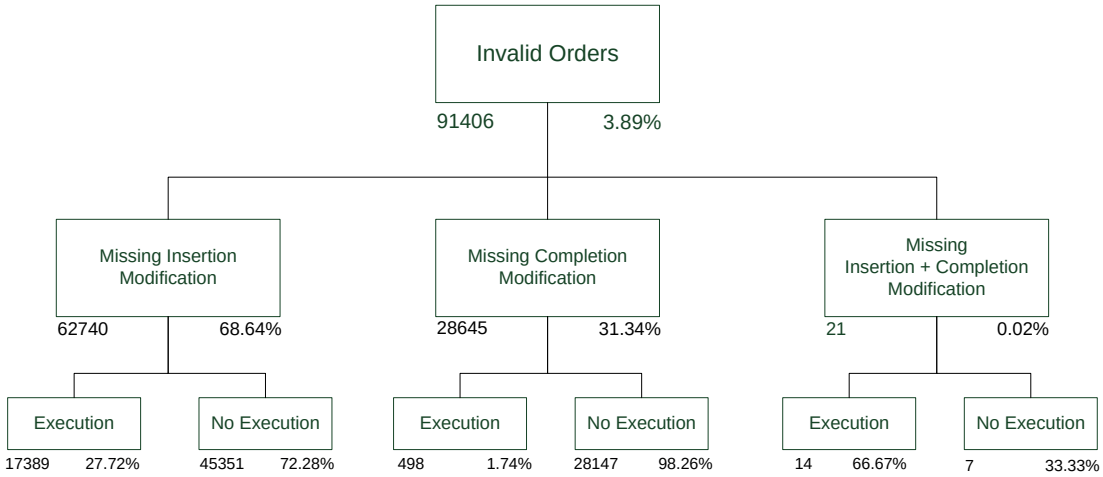

Figure 5.5: Segmentation of orders having defect event code sequences
Segmentation is first done by the 3 main error types and further separated depending on (Partial-)Execution modifications inside the order sequence.

ification, *Missing Completion* modification and *Missing Insert and Completion* modification. Considering the general goal of the order repair strategy those categories can be further subdivided in sequences which contain (Partial-)Executions and in sequences which do not. For example, the most common invalid sequence [3], consisting of only one Deletion-by-User modification, can be categorized as *Missing Insertion* modification *without execution*. The segmentation of invalid order sequences is also summarized in Figure 5.5 as a tree diagram.
The repair process identifies and corrects sequences with a missing Insert modification first and deals with missing completion modification sequences afterwards. The repair strategies to deal with all 3 different invalid event code sequence categories are described in the following subsections.

Missing Insert Modification

Missing Insert order modifications are the most common reason for defect event code sequences and represent more than two thirds of all invalid sequences. Table 5.2 shows the frequencies of all fragmentary sequences in question. To shorten the table sequences with similar patterns have been summarized, e.g. the sequence [5+]–[4] stands for one or more consecutive partial executions [5+] followed by an execution [4]. As can be seen from Table 5.2 single deletions and single executions represent the lion share of Missing Insert modification sequences.

ECS	Frequency	Execution
3	38,209	No
4	15,499	Yes
2+ → 3	7,128	No
5+ → 4	1,381	Yes
5+ → 3	441	Yes
2+ → 4	44	Yes
6	14	No
2 → 5 → 3	9	Yes
5 → 2 → 3	5	Yes
5 → 2 → 4	4	Yes
2 → 2 → 5 → 3	3	Yes
2 → 5 → 5 → 3	1	Yes
2 → 5 → 5 → 4	1	Yes
5 → 5 → 2 → 3	1	Yes
Total	62,740	

Table 5.2: Missing Insertion Sequences
ECS . . . Event Code Sequence
Execution . . . indicates if sequence contains Executions 4 or Partial Executions 5

In order to complete fragmentary sequences the `Repair` function has to insert the missing Insert modifications at the beginning of each sequence. Attributes common to each order modification in the sequence like *Limit*, *ordertype*, etc. (see Section 3.2) are typically taken from the remaining order modification in the sequence. Further the *ModificationTimestamp* of the insert modification is set to the same value as the *OrderEntryTimestamp*. Only the inserted volume has to be calculated using all available order modifications.
The calculation of the inserted volume is done chronologically by a loop through all existing modifications. In case of an Execution 4 or a Partial Execution 5 the *Size* attribute of each modification is added to the inserted volume (initially set to zero). A Modification 2 typically leads to a volume reduction of the inserted order (indicated by the size attribute). Therefore the *Size* value of a Modification 2 is also added to the inserted volume. The *Size* of a Deletion-by-User 3 and Deletion-by-System modification 6 shows the rest volume of the order which has been deleted. If an order contains one of these modifications at the end of its sequence the respective *Size* value of the deletion modification is added to the inserted volume.

Finally all necessary attributes are integrated in the reconstructed Insert order modification. The *ModReasonCode* of the modification is set to [201] (instead of [1]) to identify reconstructed Insert modifications.
The repair strategy stated above ensures the completion of all defect event code sequences listed in Table 5.2 and equivalently reduces the number of invalid orders by approximately 69%.

Missing Completion Modification

Event Code Sequence	Frequency	Execution
[1]	27,983	No
[1]–[5+]	489	Yes
[1]–[2+]	162	No
[1]–[5+]–[2]	6	Yes
[1]–[2]–[5+]	3	Yes
[1]–[1]–[3]–[6]	2	No
Total	28,645	

Table 5.3: Missing Completion Event Code Sequences

In contrast to the reconstruction of missing Insert modifications the reparation of missing completion modifications is more problematic. Completion modifications can either be Executions [4], Deletions-by-User [3] or Deletions-by-System [6] as already defined in Section 5.2.1–*Order Structure*. Table 5.3 lists all uncompleted event code sequences. Orders consisting of only one single insert modification clearly represent the lion share of the Missing Completion modification sequences. The sequence [1]–[3]–[6] represents a rare exception with only two occurrences. Here the order is completed by a User Delete modification [3] followed by a [6] System Delete Modification. Generally it is almost impossible to 'correctly' reconstruct an uncompleted order due to the range of possibilities of modification combinations at the end of the order event code sequence. Although it would have been possible to reconstruct e.g. executions if there is a turnaround mismatch in the orderbook (see Section 5.3) these reconstruction strategies have been found to be too complicated with just a little chance to be correct. Therefore all event code sequence have been completed with the newly introduced System Delete [203] *ModReasonCode* value.
The System Delete modifications which are appended at the end of each modification sequence simply contain all necessary attributes which are common to all modifications in an order (like *Limit*, *Ordertype*, etc.) and the rest volume of the order (expressed by the *Size*). The *ModificationTimestamp* (*MT*) has been set to the *MT* of the last order

plus 1ms. From the trading system's viewpoint all orders are therefore immediately deleted from the system. Of course this approach oversimplifies the way to deal with uncompleted orders and may leave room for discussion. But the immediate deletion of those orders from the book by a System Delete [203] reduces problems that could occur during orderbook reconstruction and therefore makes the dynamic order reconstruction process much easier.
Finally all 31% of uncompleted orders are kept in the system although the System Delete modification appended at the end ensures an immediate deletion from the orderbook after the last recorded order modification. As most of the orders consist of just one single Insert modification they actually play no role in either the dynamic order reconstruction process or the order visualization process.

Missing Insert and Completion Modification

Event Code Sequence	Frequency	Execution
[5+]	13	Yes
[2]	6	No
[2]–[2]	1	No
[2]–[5]	1	Yes
Total	21	

Table 5.4: Missing Insertion/Completion Sequences

Orders consisting of sequences with missing Insert and completion modifications only represent the tiny fraction of 0.02% of all invalid orders (see Table 5.4 for all sequences). Due to the repair process structure already described in Section 5.2.2 the event code sequence of those orders is first corrected by an Insert modification at the beginning (see Section *Missing Insert modification*) and afterwards by a System Delete [203] modification at the end. For example, an invalid [5]–[5]–[5] event code sequence results in a [201]–[5]–[5]–[5]–[203] sequence. A quite strange order results from the reconstruction of [2+] event code sequences. Due to the calculation of the insert order size which is the sum of all modification sizes a resulting [201]–[2+]–[203] order means that the inserted order volume is reduced to zero over its order lifetime and deleted afterwards. Only 7 such orders could be identified in the dataset, therefore this behavior can be neglected.

5.2.3 Hidden Size

Iceberg orders, already mentioned in Section 2.3.3, are a special order type and can be described through a peak size which is visible in the orderbook and the entire (hidden) volume. While the peak volume is published by the *Size* attribute of the insert order the entire hidden volume equals the sum of all (Partial-)Execution *Size* values. In order to provide a proper *Size* consistency check also the hidden volume has to be published to the dynamic order reconstruction process when the order enters the system. Therefore a new *HiddenSize* attribute is introduced during the static order reconstruction process for all orders in the reconstruction database. For non-Iceberg orders the *HiddenSize* value simply equals the *Size* of the Insert order modification; for Iceberg orders it is its actual hidden size.

5.2.4 Result

The dataset which results from the static order reconstruction process still contains all as 'invalid' classified orders with consistent order event code sequences. The newly introduced System Insert 201 and System Delete 203 *ModReasonCode* values make it possible to identify all order modifications which have been inserted during the static order reconstruction process. Further the *HiddenSize* attribute publishes the hidden volume of Iceberg orders already in the Insert order modification and makes a size consistency check for all order types during the dynamic order reconstruction process possible.
Table 5.5 looks quite similar to Table 5.1 and shows the most frequent repaired event code sequences resulting from the static order reconstruction process.

Event Code Sequence	Frequency
201 – 3	38,209
1 – 203	27,983
201 – 4	15,499
201 – 2 – 3	5,881
201 – 2 – 2 – 3	1,006
201 – 5 – 4	895
201 – 5 – 3	344
201 – 5 – 5 – 4	249
1 – 5 – 203	237
201 – 2 – 2 – 2 – 3	196

Table 5.5: Frequency of repaired order event code sequences

5.3 Dynamic Order Reconstruction

In Section 5.2 the reconstruction of defect order event code sequences has been described. Although fragmentary event code sequences are the main source of inconsistencies in the dataset only a complete reconstruction of the orderbook can show up all kinds of inconsistencies. Detectable errors can be separated in two main groups: Errors in single orders (or order level errors) and errors in the complete orderbook (orderbook level errors). Order level errors typically appear as size mismatches in orders. This error type can be detected quite easily and occurs if the inserted order volume is not exactly executed or deleted by the system. By contrast orderbook level errors can have various forms and result in an inconsistent orderbook.
While Section 5.3.1 shows the implementation of the orderbook reconstruction Section 5.3.2 and Section 5.3.3 describe order– and orderbook level errors including the repair strategies in more detail.

5.3.1 Orderbook Reconstruction

The reconstruction of the orderbook plays a crucial part for the dynamic order reconstruction process. In addition to the general introduction of orderbooks in Section 2.2 this section will give a brief technical overview of the implemented orderbook classes and their interactions. The classes of interest during the dynamic order(book) reconstruction process are the MatchEngine, Orderbook, Orderlist, Ordercollection and the Order class. Please review the class diagram from Figure 4.4 to get a better understanding of connections between the classes in question.

Addition of Order Modifications

The MatchEngine controls the whole reconstruction process and collects all Ordermodifications that result from the static order reconstruction process. The MatchEngine also instantiates the Orderbook object and adds Ordermodifications chronologically to the Orderbook. The chronological addition of Ordermodifications (by the *ModificationTimestamp*) to the Orderbook leads to a 'fast replay/simulation' of all orderbook states. The Orderbook object itself manages two Orderlists for buy and sell orders respectively. After the addition of new Ordermodifications the Orderbook separates them by the *BuySell* attribute. Buy(Sell) orders are added to the Buy(Sell)-Orderlist accordingly. The Orderlist implements an easy-to-use interface to the encapsulated Ordercollection class and has exclusive rights to modify its Ordercollection. Insert modifications added to the Orderlist lead to the instantiation of new Order objects. During the generation of Order objects the Orderlist also registers for various order events; thus the Orderlist can react to all kinds of (order-)events like order deletion (to be

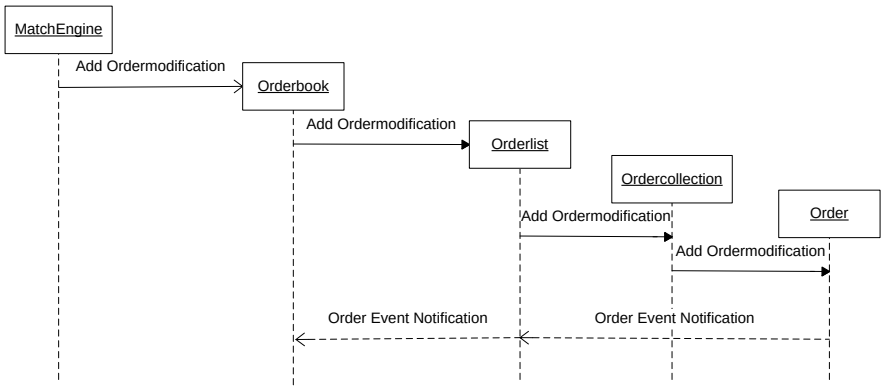

Figure 5.6: Sequence diagram of all classes involved when order modification is added to the orderbook

discussed later). Finally the newly generated Order is added to the Ordercollection object. All other modification types (*ModReasonCode*s) are directly added to the Ordercollection object accordingly. The Ordercollection is the actual data container and stores all Order objects. It is also responsible for the correct sorting of Orders and will be discussed in the next section.
Figure 5.6 visualizes the interactions of all involved classes during additions of Ordermodification objects in a sequence diagram.

Ordercollection

The Ordercollection class provides data structures for the storage and sorting of order objects. Order objects are stored in two separated Hashtables for enabled and disabled orders, respectively. These tables contain the *Ordernumber* as Key (string) and the Order object as value. The correct implementation of active trade restrictions makes this separation necessary and will be discussed in the next section in more detail. For performance reasons the order sort mechanism is implemented by the SortedDictionary class (first introduced in the .NET 2.0 Framework). The sorting procedure not only has to take the Price-Time priority rule (already discussed in Section 2.2) into account, but also the order type (Market orders generally have a limit of zero but a higher (bid-)priority). To simplify the sorting data structure the SortedDictionary contains a special sort string as key and a string list of *Ordernumber*s as values.
The sort key starts with one character indicating the order type. Possible values are Market `M`, Limit `L`, Iceberg `I` and Market-to-Limit `T`. The next characters identify the limit which is rounded to two decimals and multiplied by 100 afterwards (which in

Figure 5.7: Order sort key

Figure 5.8: Order sort key example

effect erodes the comma). The last 17 characters contain the *OrderEntryTimestamp* with a precision of 1ms. Figure 5.7 shows the structure of the order sort key. To get a better understanding of the generated order sort key we consider the example of a 98.13 EUR Limit order with the *OrderEntryTimestamp* `2005-01-05 09:15:12.110`. The order key generated is shown in Figure 5.8. . The introduced sort string is handled by the OrderComparer which implements the search logic of our SortedDictionary. The comparer splits the generated string, converts its items to their underlying data types and applies a 3-level sorting procedure. The sorting procedure has the following hierarchical levels (ordered by priority): order type, limit and time of order entry (by *OrderEntryTimestamp*. Occasionally it can happen that entered orders have the same priority or equally the same order type, limit and *OrderEntryTimestamp*. Therefore their *Ordernumber*s are stored in the same string list as values with the common order sort key as value of the SortedDictionary.

Handling of Trade Restrictions

Trade restrictions, already introduced in Section 3.8, restrict the visibility and execution of an order to defined trading phases (see Section 2.5). The trade restriction can be set to Auctions, Closing Auctions, Opening Auctions and Main Trading Phases. If, for example, a trader enters a Limit order into the system during the Continuous Trading Phase at 16:00 with a Closing Auction Only trade restriction, the order will only be visible and executed during the Closing Auction at 17:30. The order is therefore disabled and not visible in the orderbook from 16:00–17:30 and is enabled (and enters the book) at 17:30 during the Closing Auction. Another example would be a Limit order with an Auction Only restriction which enters the orderbook at 13:00

during the Intraday Auction. If the order cannot be executed during Intraday Auction it will be disabled during the Continuous Trading Phase until Closing Auction.
The examples above have shown an important and required feature of the orderbook — the ability to enable and disable orders based on the *TradeRestriction* attribute. This feature requires that the Orderbook also sets the actual trading phase correctly. Basically the Orderbook sets its trade phase based on the *AuctionTradeFlag* of incoming *Ordermodification*s. It would generally be possible to set the trading phase based only on the simulation time of the trading system (see Section 2.5) but by the usage of the *AuctionTradeFlag* also unexpected trading phases can be managed like e.g. Volatility Interruptions. After the Orderbook has recognized a trade phase change it notifies the Orderlist by sending a message in form of a TradePhaseChangedEvent. Based on the new trading phase the Orderlist enables and disables Orders in the Ordercollection by their trade restrictions.

Order Events

In order to notify the system about changes made to an order a number of events are implemented in the Order class. Events are defined for executions and deletions and are sent to the Orderlist object. The event generation can be described as follows:
First an Ordermodification is added by the Orderlist object to the Ordercollection. The Ordercollection further adds the Ordermodification to the according Order (see also Section *Addition of OrderModifications*). In case of a Partial Execution, an Execution or a Deletion By User/System the Order fires an OrderExecution event or an OrderDeleted event, respectively. OrderExecution events are generally passed to the Orderbook which manages the executed order volumes. Full-Executions and Deletions-By-User/System (notified by the OrderDeleted event) inform the Orderlist to delete the specified orders from the Ordercollection. The function of order notification events has already been sketched in Figure 5.6. Market-to-Limit orders are handled in a special way through the OrderModifiedInsert and the OrderModifiedDelete events. As already described in Section 2.3.3 a Market-to-Limit order is first inserted as a Market order and changes to a Limit order after the first partial execution. The order is therefore deleted from Ordercollection (initiated by the OrderModifiedDelete event) and re-inserted with its changed attributes by the OrderModifiedInsert event.

5.3.2 Order Level Errors

Order level errors represent the simplest error type detected by the dynamic orderbook reconstruction process. Unlike the static order reconstruction process, which only repaired event code sequences based on the *ModReasonCode* attribute those errors occur due to size mismatches inside the order. Generally the inserted size/volume of an order which entered the orderbook has to equal the summed sizes/volumes

of all (Partial-)Executions, Modifications and Deletions-by-User/System. The order therefore must have a volume of zero when it is booked out of the orderbook. A volume not equal to zero at completion indicates a size mismatch and has to be investigated further.
The definition of an order size mismatch leads to two simple cases: volumes greater and volumes less than zero at completion. Volumes greater than zero should normally appear if order modifications (which lower the volume) are missing. By contrast, the reason for negative values could be a wrong size in either the Insert or the completion modification.
The introduction of the *HiddenSize* attribute (see Section 5.2.3) enables the system to check even Iceberg orders for size mismatch errors.
Although no size mismatches could be detected in the investigated dataset the check for order level errors was a nice utility to debug various orderbook reconstruction errors.

5.3.3 Orderbook Level Errors

The following section describes the detection of dynamic errors resulting from the reconstructed orderbook in Section 5.3.1. To capture all facettes and errors of the trading process a re-implementation of the XETRA matching engine combined with a consistency check of matched orders and modifications from the dataset would probably be the most exact way to go. Early implementations of the XETRA matching engine showed a great increase of the dynamic orderbook reconstruction runtime and have therefore been abandoned. Instead it was discovered that a 'matching' directly done by order modifications would be sufficient for reconstruction purposes. Only a consistency check for important parameters of the orderbook class has been implemented and leads to a dramatic decrease in reconstruction runtime. However, in some cases you cannot make exact statements about which orders have exactly been matched — but that is actually of no interest during for reconstruction process. An implementation of a re-engineered XETRA matching engine makes more sense in the context of trading strategy testing or live simulation and is left for further developments of this software package.
The dynamic orderbook parameters which have been analyzed for reconstruction purposes are the bid/ask spread and the executed volumes in the orderbook. The test is done after each addition of Ordermodifications to the Orderbook. Errors have been written to a predefined log in the CSV file format for statistical reporting issues.

Bid/Ask Spread

The bid/ask spread of an orderbook is defined as the difference between the best ask (sell) and the best bid (buy) limit (see also Section 2.2). Crossed orderbooks or, equivalently, negative bid/ask spreads should lead to the immediate execution of respective orders. Therefore the spread is checked to be positive after the addition of OrderModifications to the book. In some cases, however, the spread can also become zero or negative.
Orders are only executed at the end of auctions which can lead to a negative spread while an auction is hold. For that reason orderbooks with negative spreads during auctions are still considered to be valid.
The insertion of Market orders can also lead to a spread of zero during Continuous Trading Phases. Although those orders should be executed immediately when inserted (*OrderEntryTimestamp* equals *ModificationTimestamp* of Execution) the XETRA matching engine sometimes executes those orders a few milliseconds afterwards. If a Market order exists in the buy-orderbook, the best bid(ask) is set to the best ask(bid) which results in a spread of zero.
By the definitions for valid bid/ask spreads mentioned above no spread errors could be encountered in the investigated dataset.

Executed Volumes

After the addition of new Ordermodifications to the Orderbook the volume of executed orders is reported to the Orderbook class through the OrderExecution event. Typically orders contain (Partial-)Execution modifications with the same size and *ModificationTimestamp* of their matched counterpart. The executed buy and sell volumes reported to the Orderbook after each modification addition iteration (or equivalently after each simulation time step) therefore need to be equal. Unequal volumes indicate that executed orders are missing.

The error output statistic in Table 5.6 shows the aggregated numbers of volume mismatches encountered during the dynamic orderbook reconstruction process. The raw error data showed that mismatches always happen due to executions on only one side of the orderbook within the investigated time period. Further the mismatches appeared rather uniform over the simulation period.
Therefore we define Buy(Sell) Order Mismatches as executed buy(sell) orders without executions in the sell(buy) orderbook. Although the summed volumes of Buy- and Sell Order Mismatches appear rather big at first sight the missing volumes only represent 0.0021% of all shares traded in the sample period. A by-stock analysis shows that shares of Fresenius Medical Care have the highest relative error rate of 0.02% while the lowest is given by Metro AG with only 0.0003%.
In order to correct missing volumes a new Market order has been inserted at each

time when an executed volume mismatch was detected. An executed buy volume less than the executed sell volume triggers the insertion of a new buy order and vice versa. The insertion size is generally given by

$$|(executedBuyVolume) - (executedSellVolume)|$$

From the trading system's point of view the inserted order is immediately executed with a newly introduced System Execution `204` *ModReasonCode* which leads to a complete order event code sequence of `201`–`204`. The newly introduced Market order therefore consists of two order modifications with *ModificationTimestamp*s set to the respective time of order mismatch and corrects all volume mismatches that occurred in the investigated time period.

Stock Name	BOM	SOM	BS	Exec.Shares*
ADIDAS-SALOMON AG O.N.	97	0	97	3,797,385
ALLIANZ AG VNA O.N.	537	634	1171	25,176,559
ALTANA AG O.N.	760	8	768	7,217,032
BASF AG O.N.	591	285	876	29,849,186
BAY.HYPO-VEREINSBK.O.N.	962	479	1441	84,639,113
BAY.MOTOREN WERKE AG ST	440	990	1430	29,678,956
BAYER AG O.N.	346	574	920	62,443,602
COMMERZBANK AG O.N.	1391	1741	3132	59,850,440
CONTINENTAL AG O.N.	100	176	276	10,449,262
DAIMLERCHRYSLER AG NA O.N	532	442	974	52,869,814
DEUTSCHE BANK AG NA O.N.	38	229	267	51,786,795
DEUTSCHE BRSE NA O.N.	195	0	195	9,425,059
DEUTSCHE POST AG NA O.N.	101	0	101	28,479,967
DT.TELEKOM AG NA	200	653	853	300,265,473
E.ON AG O.N.	488	300	788	30,200,060
FRESEN.MED.CARE AG O.N.	64	564	628	2,903,284
HENKEL KGAA VZO O.N.	0	417	417	4,645,115
INFINEON TECH.AG NA O.N.	2351	200	2551	134,910,873
LINDE AG O.N.	295	0	295	6,309,161
LUFTHANSA AG VNA O.N.	187	640	827	37,344,071
MAN AG ST O.N.	400	339	739	18,592,895
METRO AG ST O.N.	63	0	63	23,462,145
MUENCH.RUECKVERS.VNA O.N.	992	373	1365	18,699,341
RWE AG ST O.N.	546	124	670	48,139,296
SAP AG ST O.N.	438	868	1306	27,688,252
SCHERING AG O.N.	901	312	1213	21,473,705
SIEMENS AG NA	163	100	263	48,063,271
THYSSENKRUPP AG O.N.	698	559	1257	36,674,671
TUI AG O.N.	255	268	523	18,013,437
VOLKSWAGEN AG ST O.N.	343	560	903	39,332,589
Total	14,474	11,835	26,309	1,272,380,809

Table 5.6: Executed Size Order Mismatch
BOM . . . Buy Order Mismatch Volume
SOM . . . Sell Order Mismatch Volume
BSV . . . Sum of Buy- and Sell Order Mismatch Volumes
* Number of Executed Shares have been double-counted (executed buy and sell volume)

6 Orderbook Visualization

This chapter highlights the implementation of the orderbook visualization process; the orderbook visualization user interface will be discussed separately in Chapter 7. Starting with an introduction (Section 6.1) which also describes the main features of the *Orderbook Visualization* package the visualization process (Section 6.2) is outlined including Data Retrieval (Section 6.3), Order Binning (Section 6.4), Visualization (Section 6.5), Zoom (Section 6.6) and Filtering (Section 6.7).

6.1 Introduction

The Orderbook Visualization package is designed to provide an easy-to-use interface for the visual exploration of XETRA orderbook data. Although statistical software packages like R or Matlab support more sophisticated tools for data analysis in general ([R08], [Mat08]) the implementation of an interactive data visualization tool including a fast zoom function had to be done in an object oriented language because of flexibility and performance issues. As already mentioned in Chapter 4 the C# programming language was selected because of its powerful Windows Presentation Foundation (WPF) graphics library, fast SQL Server data access libraries and — last but not least — because of the highly productive Visual Studio 2008 development environment. However, the execution of the *Orderbook Visualization* tool requires the Microsoft Windows XP SP2 (or higher) operating system.
The graphical presentation of orderbook data also reveals interesting trading patterns. Typical patterns found in the XETRA orderbook can be described as order sequences whereas the orders have common attributes like e.g. the inserted volume. By using the filters implemented in the *Orderbook Visualization* program, orders having specific attributes can be emphasized which also highlights interesting trading patterns in the dataset.
Finally, the *Orderbook Visualization* program not only allows users to do preliminary explorative data analysis and study the states and mechanisms of the XETRA Orderbook; they can even explore trading patterns found on the XETRA orderbook. The detection of visual trading patterns in the XETRA dataset will be further described in Section 8.
The supported features of the *Orderbook Visualization* package are summarized below.

Data Retrieval: The implemented data connectors already support the import of reconstructed orderbook data from database or from CSV files, respectively. The connectors are implemented in a very generic way (see Chapter 4) and can be extended to various other databases or file formats.

Data Export: In order to store graphical representations of the orderbook export functions for various image file formats have been implemented (Section 7.4.3). The implemented framework also includes functions to export selected orderbook data to databases as well as text files (CSV) which can be analyzed with other statistical tools like R or Matlab.

Data Visualization: The visualization of orderbook data naturally represents the core functionality of this software package. Beside the intuitive graphical representation of orders also other features like visualization of different layers for grids, spreads, etc. are possible. Altogether it enables the user to get a detailed and intuitive overview of what actually happened in the orderbook at any time.

Data Navigation: The fast navigation through the visualized orderbook also represents one of the main features of this program. It is done by the `Zoom` function and provides the visualization of orderbook data at any level of detail. Order level information can also be retrieved by simply clicking any visualized order.

Filtering: By using implemented filters only a subset of orders can be displayed according to defined filter parameters for any order attribute. The application of filters can reveal interesting trading patterns in the orderbook. Even some algorithmic patterns can be detected and displayed, zoomed, exported, etc. in a very intuitive manner.

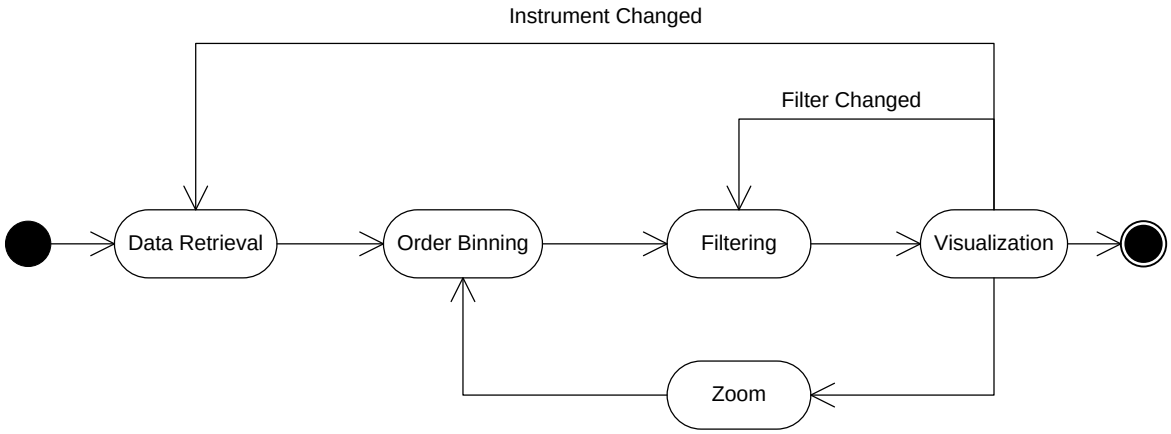

Figure 6.1: Overview of the Visualization Process

6.2 Process Overview

This section describes the visualization process of orderbook data also sketched in Figure 6.1.
As a first step orderbook data is imported by the data retrieval process. Implemented connectors are able to import data resulting from the reconstruction process (Section 5) form either SQL databases or CSV files.
After Ordermodifications and Orders have been generated they are stored in data structures designed to provide a fast access for two-dimensional graphical representation purposes. The storage location can be described as data'bins' whereas a bin represents a single pixel on a x-y pane. Attributes like *ModificationTimestamp* and *Limit* are scaled properly to determine the correct bin for data storage. This whole process is also referred as order binning and will be described in Section 6.4 in more detail.
The order filtering process provides a filter function either on the Order level or the aggregated OrderRectangle level. The Filter on the Order level supports parameters for any Order specific attribute (like e.g. *Ordertype*) and is executed after the binning process. By contrast, OrderRectangle filters apply to aggregated attributes calculated from order bins (like e.g. *Volume*). The visualization process is responsible for the representation of already binned and filtered orders. After orders have been rendered and displayed on screen the dataset can be navigated using the `Zoom` function. Technically a zoom leads to a cycle of order re-binning followed by a visualization of zoomed orders. A filter change simply leads to a re-application of filters to the binned orders followed by a Visualization. For the entire order binning–visualization–zoom cycle performance optimization was a big issue and led to a fast navigation through large

orderbook datasets. A change of the instrument leads to a re-execution of the entire visualization process, starting with data retrieval. All mentioned actions involved in the visualization process will be described in the following sections in more detail, including data retrieval (Section 6.3), order binning (Section 6.4), visualization (Section 6.5), zoom (Section 6.6) and filtering (Section 6.7).

6.3 Data Retrieval

The data retrieval process has been implemented using the connectors from the *Orderbook Engine* library and is therefore quite similar to the reconstruction process data import functions described in Chapter 5. Imported data results from the reconstruction process and contains 'synthetic' Ordermodifications as well with the newly introduced attribute *HiddenSize* and the *ModReasonCode*s System Insert 201, System Execution 204, and System Delete 203. As already outlined those Ordermodifications should ensure the static and dynamic consistency of the dataset.
Basically data retrieval is implemented by the `GetData` function in the main Window1 class which instantiates a data connector and reads data through its IConnector interface. Imported Ordermodification objects will further be stored in Order objects, as described in Section 6.4.

6.4 Order Binning

6.4.1 Overview

The order binning process describes the storage of orders generated by the data retrieval process (Section 6.3) in special data structures. The structures are designed to provide a fast data access for the proper visualization of orders in the visualization process.
The overall goal of the visualization process can be described as the graphical representation of orders, which are generally represented in an event-based way by order modifications, on a two dimensional x-y pane. The data structures are designed as a data grid to directly represent each pixel of the pane. The order binning process therefore plays a crucial part due to the actual pre-transformation of Orders by storing them in the grid. It is also one of the main sources when performance issues are considered.

6.4.2 Basic Binning Idea

As already mentioned in Section 5.2.1 the order lifetime cycle can be described through a sequence of various events. Beginning with an Insert modification and maybe some

modifications in between, the lifetime always ends with a completion modification. For visualization purposes two order properties are of special interest: limit and the order lifetime.
Except for Market(-to-Limit) orders the limit is always the same for all modifications of an order. The *Limit* attribute is a floating point number with a precision of two decimal points (see also Section 3.2.11). However, other order attributes like the *Volume* can change in case of Modifications (*ModReasonCode* 2) or Partial executions (*ModReasonCode* 5).
The order lifetime starts at the time defined by the *OrderEntryTimestamp* or the *ModificationTimestamp* of the insert order. Its end is marked by the *ModificationTimestamp* of the completion modification. The attribute *Orderlifetime* is therefore defined as the difference between the completion *ModificationTimestamp* and the insertion *ModificationTimestamp*.
For visualization purposes the attributes *Limit* and *Orderlifetime* have to be displayed on a 2-dimensional pane. The x-axis is defined as time and the *Limit* is displayed by the y-axis. The display area of the pane is limited by the four attributes min-/maxTime and min-/maxLimit. Internally the pane is created by a two-dimensional array of pixels whereas the colors are set by a 3-dimensional vector for red, green and blue values. In order to allow a fast visualization of imported orders it was decided to create a data structure with a 1:1 representation of each pixel of the pane. The basic data structure can therefore be simplified to a 2-dimensional array (or data grid); its actual implementation is left to Section 6.4.3. Due to the discrete character of pixels or array elements each element could also be seen as an order storage bin spanning a specified time and price interval. Therefore the resolution of the pane which is defined as the number of horizontal and vertical pixels also sets the possible size of the data structure and equivalently the number of bins. The number of bins is another characteristic attribute of the pane and becomes crucial when performance issues are considered. After these preliminary thoughts the basic binning idea can now be defined as a mapping problem of Orders (defined by OrderModifications) on a 2-dimensional data grid. The attributes which define *Limit* and *Orderlifetime* have to be properly scaled to put the generated orders in the data structure.
The implemented data structures are reviewed in Section 6.4.3 and the determination of order bins is be described in Section 6.4.4. In Section 6.4.5 the operational binning process is further analyzed.

6.4.3 Data Structures

The data structures implemented for order storage are designed to provide a fast data access of the visualization process. At first sight a simple two-dimensional array would be sufficient for this purpose. However, only a limited amount of space spanned by the time and limit attribute is filled with Order objects. Therefore an

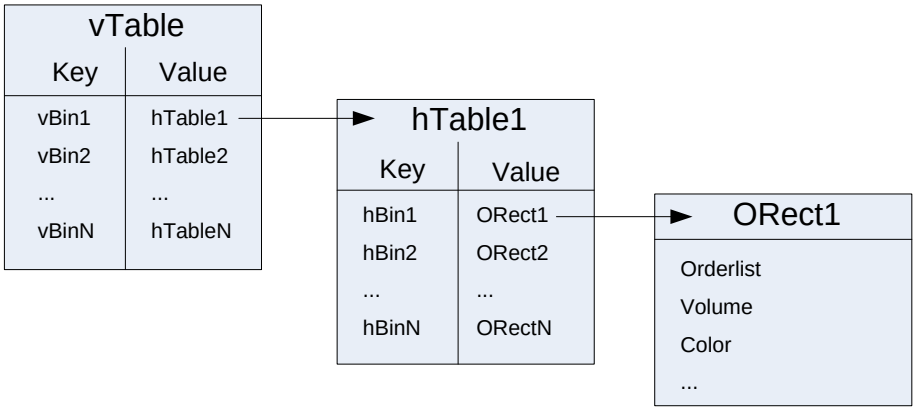

Figure 6.2: Design of the order visualization data structure
The order visualization data structure consists of 3 parts:
vTable (SortedDictionary) stores vertical coordinates as key and a SortedDictionary (hTables) as value
hTable stores horizontal coordinates as key and OrderRectangle objects as value
OrderRectangle actually represents a bin and contains an Orderlist of all respective orders, aggregated features (e.g. volume) and color information

implementation with a simple array would lead to a massive waste of memory. It was therefore decided to implement the data structure as a double-Hashtable. The basic structure design is shown in Figure 6.2.

The first Hashtable (Hashtable1) stores the y-coordinate (or vertical bin-number) as defined by the Order *Limit* as key and a second Hashtable as value. Hashtable2 contains the horizontal bin-number (defined by time) and an OrderRectangle object as value.
Compared to the simple two-dimensional array approach the Hashtables just use as much memory as necessary and provide a fast lookup method by the keys which are simply the according x-y coordinates of Orders. The visualization process (see Section 6.5) therefore implements a fast access when Orders have to be addressed by their coordinates. The actual order storage is done by the OrderRectangle object which contains a list of all Order objects added. Most importantly it also manages Order attributes which can change over time when further Ordermodifications are added. The best example is the *Volume* attribute that can change when e.g. a Partial Execution modification is added afterwards. The reason for the special handling of those order attributes, which will also be referred as OrderRectangle attributes in this chapter, is the recycling of Order objects in the memory's heap to reduce memory requirements and will be discussed in Section 6.4.5. Attributes set by the

OrderRectangle object like pixel color are also accessed by the visualization engine.

6.4.4 Order Placement

The bin sizes of the 2-dimensional structure can be defined by the display borders for the attributes time (minTime, maxTime), limit (minLimit, maxLimit) and by the number of horizontal (hDim) and vertical bins (vDim). The horizontal bin-size for the time attribute results as

$$hBinSize = \frac{maxTime - minTime}{hDim}$$

Analogous the vertical limit bin size is defined as

$$vBinSize = \frac{maxLimit - minLimit}{vDim}$$

Figure 6.3 shows an example grid with a resolution of 6x7 containing all relevant attributes.

After the actual shape of the grid has been determined by the grid parameters the placement of orders into the data structure will be described. As already mentioned in Section 6.4.2 the necessary order attributes are the *Limit* and the order start- and endtime defined by the *ModificationTimestamp* of the Insert- and completion order modification. Based on these attributes the corresponding bin coordinates have to be calculated as described in the following sections.

Limit Calculation

The calculation of the vertical coordinate (vBin) for an order with a limit of oLimit simply follows from the grid definition and is given by

$$vBin = vDim - \lfloor \frac{oLimit - minLimit}{vBinSize} \rfloor$$

whereas vDim, minLimit and vBinSize have already been defined in Section 6.4.4. If oLimit is outside the defined borders minLimit or maxLimit the order lies in the clipping area and cannot be displayed.

Time Calculation

In contrast to the calculation of the vertical bin coordinate in Section 6.4.4 the time calculation is a bit more complicated. For the visualization of consecutive trading days sessions actually have to be 'glued together'. In order to calculate the actual trading

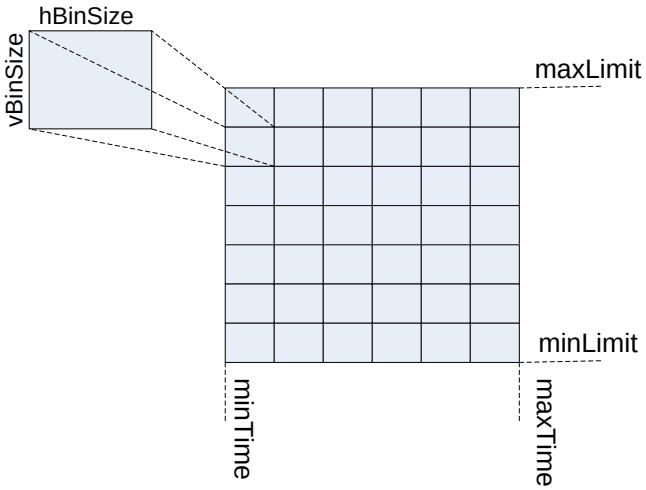

Figure 6.3: Example of a 6x7 Visualization grid containing all important parameters
vBinSize ... vertical bin size (limit interval)
hBinSize ... horizontal bin size (time interval)
min/maxLimit ... limit price borders
min/maxTime ... time borders

time a TradingSession object is generated for each trading day between the borders minTime and maxTime. A TradingSession object contains the attribute *startTime*, *endTime* and the resulting trading time which results by subtracting *startTime* from *endTime*.
If, for example, the trading time between two DateTime objects `2005-01-05 10:30:00.000` and `2005-01-12 16:15:00.000` have to be calculated and a general TradingSession is defined with a *startTime* of `09:00:00.000` and `17:30:00.000` the TradingSessions generated are listed in Table 6.1. It can be seen that the weekend from 2005-01-09 until 2005-01-10 has been skipped which results in a total trading time interval of 56:45:00.000. Using the time interval calculation method described above the calculation of the horizontal time bin is quite similar to the vertical limit bin calculation for any given order time oTime (either *startTime* or *endTime*) and is given by

$$hBin = 1 + \lfloor \frac{oTime - minTime}{hBinSize} \rfloor$$

The division of time intervals (or Timespan objects in .NET) is actually not defined. The Timespan object has therefore been converted to long variables which represent respective Timespan values as tick numbers (one tick equals a time interval of 100 nano

TS	startTime	endTime	trading Time
01	'05.01.2005 10:30:00.000'	'05.01.2005 17:30:00.000'	07:30:00.000
02	'06.01.2005 09:00:00.000'	'06.01.2005 17:30:00.000'	10:30:00.000
03	'07.01.2005 09:00:00.000'	'07.01.2005 17:30:00.000'	10:30:00.000
04	'10.01.2005 09:00:00.000'	'10.01.2005 17:30:00.000'	10:30:00.000
05	'11.01.2005 09:00:00.000'	'11.01.2005 17:30:00.000'	10:30:00.000
06	'12.01.2005 09:00:00.000'	'12.01.2005 16:15:00.000'	07:15:00.000
		Total	**56:45:00.000**

Table 6.1: Trading Session Time Calculation Example
startTime and endTime is given in the format 'DD.MM.YYYY HH:MM:SS.MS' trading time by HH:MM:SS.MS

Grid		**Order**	
Attribute	Value	Attribute	Value
minTime	'10:30:00'	Limit	97,97
maxTime	'11:05:00'	startTime	'10:37:30'
minLimit	97,90	endTime	'10:57:30'
maxLimit	98,00		
hDim	7		
vDim	5		
vBinSize	0.02	start-bin (hBin/vBin)	(2/2)
hBinSize	'00:05:00'	end-bin (hBin/vBin)	(6/2)

Table 6.2: Specifications of an Order Placement Example
Grid and order parameters are given on top, resulting binsize and bins are shown below.

seconds). Time intervals as well as the size of the horizontal time bin (hBinSize) are therefore internally represented as ticks of type long.
In case that oTime lies outside the defined trading session time intervals the time is 'rounded' to the next *startTime* or *endTime* of the according TradingSession. Considering our TradingSession example above a timestamp of `2005-01-05 18:00:00.000` would be rounded to `2005-01-05 18:00:00.000`, a timestamp of `2005-01-06 07:30:00.000` to `2005-01-06 09:30:00.000`. Therefore even orders which have been entered during Pre- and Post-Trading phases are displayed at the beginning or the end of the defined TradingSessions.

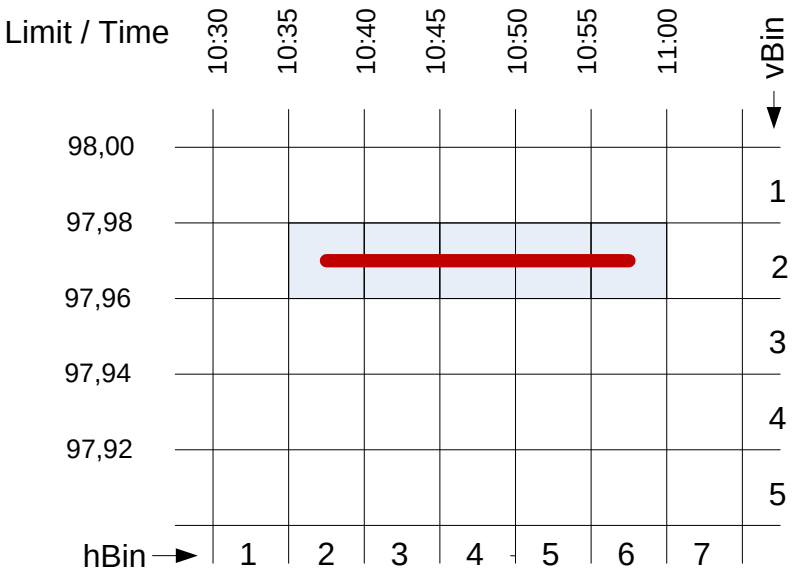

Figure 6.4: Order Placement Example
Specified order in 6.2 drawn on the visualization grid from (2/2) to (6/2)

Order Placement Example

To put all considerations about order placement together we consider a simple example. All relevant parameters for the placement calculation example are given in Table 6.2. The attributes below the horizontal line in Table 6.2 have been calculated using the formulas defined in previous sections and show the results for the bin sizes (hBinSize, vBinSize) and the resulting coordinates for the order insert- and completion modification. As can be seen from Figure 6.4 the finally placed order is not only inserted into the start- and end-bin (2/2 and 6/2). It is also added to the bins which lie in between like 3/3, 4/2 and 5/2. Bins to which the respective order object has been added are marked with a gray background.

6.4.5 Binning Process

The implemented binning process slightly deviates from the general order placement strategy from Section 6.4.4 and will be described in this section. The general process is shown in Figure 6.5 and can be described as a loop through all OrderModifications generated by the data retrieval process (Section 6.3). After the OrderModification has been retrieved (function `GetModification`) an according Order is generated and

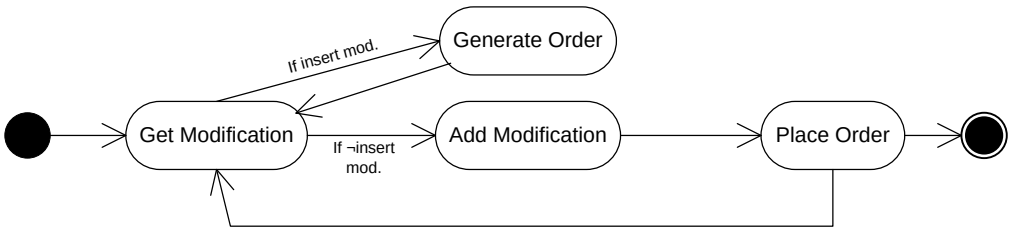

Figure 6.5: Binning Process Overview

placed in the data structures (Section 6.4.3).
As a first step the OrderModification is imported through an iterator and analyzed by its *ModReasonCode*. In case of an insert modification an Order object will be generated and is preliminary stored in a Hashtable container. Afterwards the OrderModification iterator continues with the next modification. All other modification types are simply added to the already generated Order in the Hashtable. In contrast to the order placement example described in Section 6.4.4 Orders are not actually placed only by their startTime and endTime. After the addition of an OrderModification the Order is already placed into the data grid. The order endTime is therefore defined as the modification added to the Order — its startTime as the modification before the added order. This approach is used for the correct calculation of OrderRectangle attributes (see Section 6.4.3). However, it could lead to multiple Order insertions at the OrderModification connection points. This issue has been corrected by a order query executed before insertion.
Finally when the binning process has iterated through all OrderModification objects the resulting data structure is filled with Order objects and ready for the graphical visualization process described in the next section.

6.5 Visualization

The visualization process has the goal to graphically represent Orders stored in the data structures (see Section 6.4.3). As can be seen from the process overview in Section 6.2 the visualization process is executed after the binning (Section 6.4) and filtering (Section 6.7) of orders has been finished. The entire information about binned and filtered orders is aggregated in an OrderRectangle object and stored in the data structures (Section 6.4.3). The OrderRectangle object therefore plays a crucial part for the visualization process as it not only contains information about the Order objects and aggregated attributes (like *Volume*). It also defines graphical parameters like the

color and coordinates (by its location in the structure) for any (filled) order-bin or pixel.
The color attribute is set when Orders are added to the OrderRectangle object and dependent on the *BuySell* attribute. For buy orders it is defined as blue and for sell orders as red, respectively. If an OrderRectangle contains buy and sell orders its color is set to green.
The task of the visualization process results as an iteration through all OrderRectangle objects stored in the data structures followed by the painting of respective pixels on a two-dimensional image. Although this procedure sounds quite simple early visualization implementations using WPF Rectangle objects for the representation of each OrderRectangle showed serious performance problems. Finally it has been decided to draw the complete graphical orderbook representation as a bitmap picture using the WriteableBitmap class included in the *System.Windows.Media.Imaging* namespace. The WriteableBitmap class is known to be one of the fastest ways to display images in the WPF framework and proved to be the best solution for our purposes. The fast visualization performance combined with incredible re-scaling performance when the image (included in a Viewbox) is resized have been one of the main arguments for using the WPF framework in .NET 3.5 as already mentioned in Chapter 4

6.6 Zoom

The `Zoom` function is one of the most important data navigation features and allows an unbounded scale-up of any visualized orderbook area. It enables the user to get an in-depth view of any orderbook state at any time or price interval. A zoom-out function has also been implemented to recover the previous visualization state of the orderbook.
Technically speaking only a subset of Orders already binned and visualized from the data structures (Section 6.4.3) are used for `Zoom`. The subset is defined by new visualization borders which we will define in this section as zoomMin-/MaxTime for the time- and zoomMin-/MaxLimit for limit dimension. Although zoom limits change the resolution, or equivalently the numbers of horizontal (hDim) and vertical bins (vDim), stay the same. This results in a change of hBinSize and vBinSize (see Section 6.4.4) and therefore a re-binning of relevant orders becomes necessary.
The entire zoom-process can be summarized as follows:
1. Get Relevant Orders
2. Extract OrderModifications from Orders
3. Re-Bin Orders and store them in data structures

The first step is the determination of relevant Orders based on the newly introduced zoom borders that define the limit and time interval. The borders are

converted to respective hBin/vBin values which define the extraction window for the current data structures. With those limits all relevant Orders can be retrieved from the data structures.
As a second step the OrderModifications of relevant Orders are extracted as the binning procedure already introduced in Section 6.4 needs single OrderModification objects.
Finally the binning procedure creates new data structures for the zoom state and stores the newly generated Orders.
It has to be mentioned that this approach is probably not the most efficient one and could be further optimized in various ways. The re-binning procedure, for example, could directly map orders(or modifications) from the old data structures to the zoom structures whereas less calculations have to be done. Further the re-instantiation of orders leads to a relatively high memory usage and should also be re-thought. These issues leave further room for improvements although the zoom function already shows a very good performance.
In order to provide a zoom-out function the pre-zoomed state has to be kept as not all orders can be recovered from the zoomed data structure state. For that reason zoom states, or equivalently data structures holding orders, are saved in a so-called zoom-state list. A zoom-out therefore leads to a simple replacement of the currently viewed data structure by the previous one in the list. Although this approach leads to a very fast zoom-out performance it is quite memory intense. Figure 6.6 shows a visualization example of the zoom procedure.

6.7 Filtering

The implemented filters extend the data exploration capabilities of the *Orderbook Visualization* package and restrict the visualization of Orders or OrderRectangles to a specified subset. Generally the filters can be divided in two different classes: OrderFilters and OrderRectangleFilters. OrderFilters are able to filter single orders by their attributes (see Section 3.2) and already take place during the binning process (Section 6.4). In contrast OrderRectangleFilters are designed to filter entire OrderRectangle objects by their aggregated OrderRectangle attributes which have been introduced in Section 6.4.
Order and OrderRectangle attributes are generally implemented as properties marked by custom attributes. Marked properties are identified by the visualization engine at runtime using reflection. This approach facilitates the implementation of new Order(Rectangle)Filters after a new Property has been added to the Order(Rectangle) class and marked by a custom attribute.
Generally multiple filters for any defined attribute can be applied and are used in conjunction. However, a distinction has to be made between OrderFilters and OrderRectangleFilters. The execution of OrderFilters typically lead to an enabling or dis-

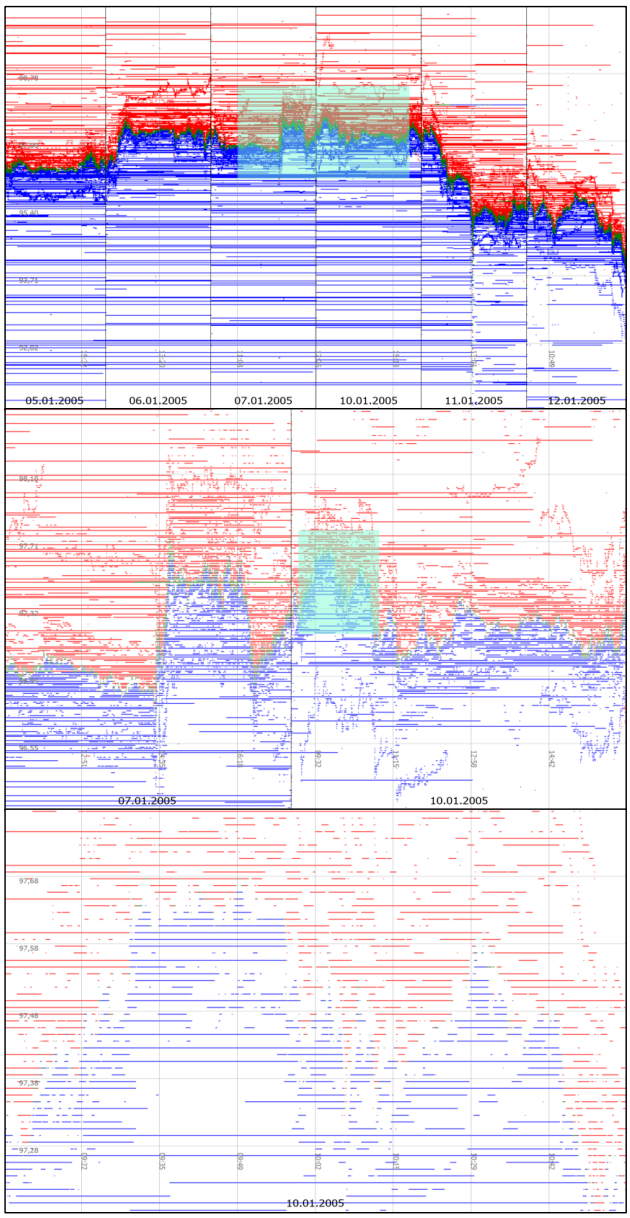

Figure 6.6: Example of the orderbook visualization zoom function
The orderbook visualization zoom capabilities are shown in a 3-step zoom navigation example. Zoom-in can be viewed by top-down reading the pictures. Blue rectangle shows the zoom borders of the next zoom state below.

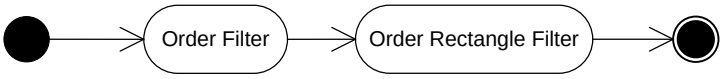

Figure 6.7: Basic Filtering Process

abling of orders in all OrderRectangles (see Section 6.7.1 for details). Due to the fact that OrderRectangle attributes are only calculated for enabled orders the application of OrderFilters affect OrderRectangle attributes and have to be executed before. Aggregated OrderRectangle attributes are only executed afterwards, just before the actual visualization process (Section 6.5). The general filtering process is also sketched in Figure 6.7.
In order to make the generic implementation of filters as simple as possible filter parameters can be generated automatically based on the property data type. For enumerations and numeric types extraction methods have already been implemented. The enumeration parameters simply result from all items listed in the Enum type. The filter parameters for numeric attributes result from a fixed number of quantiles defined by the distribution of the filter property and therefore require more calculation effort. Currently numerical filter parameters are set to quintiles whereas other values could be defined for each different filter.
The next sections describe the implementation of OrderFilters (Section 6.7.1) and OrderRectangleFilters (Section 6.7.2) in more detail.

6.7.1 Order Filters

OrderFilters are defined for marked Order properties and allow users to filter Orders by attributes like *Ordertype* or *Traderestriction*. Table 6.3 shows all already implemented OrderFilter attributes. Orders are generally stored and aggregated in OrderRectangle objects which are located in the data structures described in Section 6.4.3. An OrderFilter has to enable or disable Orders located in the Orderlist of the OrderRectangle objects based on various filters and parameters. If an applied filter restriction leads to a disabling of Orders all Orders in question have to be deleted from the Orderlist and added to a Hashtable for disabled orders located in the same OrderRectangle object. This Hashtable also stores a list of filters which have been applied to each Order. Contrary, when a filter restriction is released the disabled orders in each OrderRectangle are scanned while deleting the respective filter from each Order in the list. Only if no more filters are applied to disabled orders the Order is enabled, or to be more precise, deleted from the disabled-orders Hashtable and added to the Orderlist. Each time when an Order is enabled or disabled the OrderRectangle attributes have to be recalculated e.g. a disabled Order leads to a reduction of the aggregated *Volume* and vice versa.

Property Name	Datatype	Description
BuySell	Enumeration	Order BuySell Attribute
TradeRestriction	Enumeration	Order TradeRestriction Attribute
EnterAuctionTradeFlag	Enumeration	AuctionTradeFlag Attribute of Insert Modification
Ordertype	Enumeration	Ordertype Attribute
EnterSize	Integer	Size Attribute of Insert Modification
Orderlifetime	Integer	Orderlifetime (endTime - startTime) in seconds
Limit	Double	Order BuySell Attribute

Table 6.3: Order filter attributes
Attributes which can be used using the implemented filter functions.

To take applied order filters into account the visualization process now simply has to check if the Orderlist of the OrderRectangle is empty by calling the OrderRectangle.`IsEmpty` function. A positive return value signals that all Orders in the OrderRectangle have been disabled and results in a visualization of an white pixel.

6.7.2 Rectangle Filters

OrderRectangleFilters are defined for OrderRectangle attributes apply only after the execution of OrderFilters. While numerous OrderFilters have been defined (see Section 6.7.1) only the aggregated *Volume* property (data type integer) has been defined as an OrderRectangle attribute. In contrast to OrderFilters OrderRectangleFilters enable or disable entire OrderRectangle objects based on the filter attributes. The enabling/disabling of OrderRectangles is done in a loop through all elements and sets the *IsRestricted* property of OrderRectangles accordingly. The visualization engine only has to check the *IsRestricted* field to correctly draw the pixel.

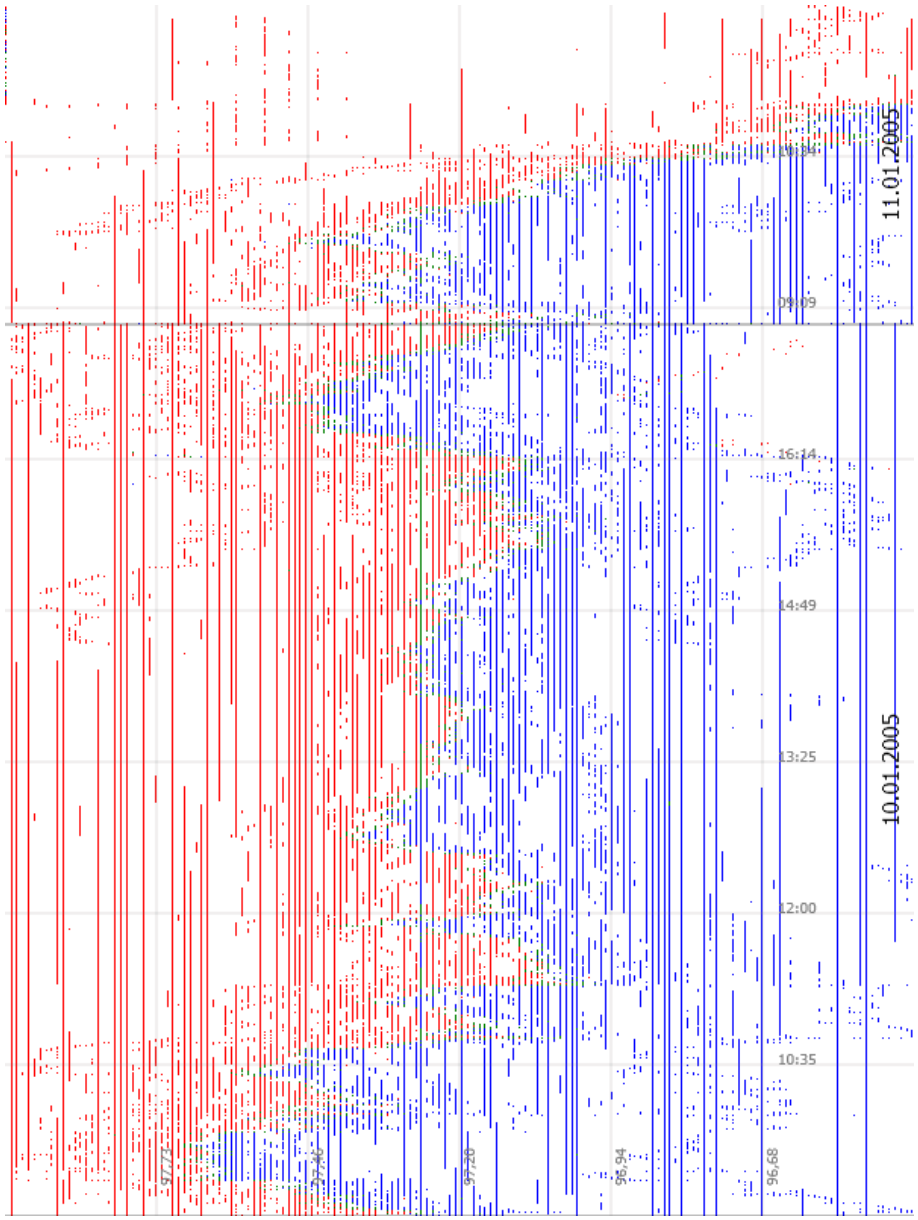

Figure 6.8: Example of the Orderbook visualization function
Example shows the zoomed orderbook states of the Allianz stock from 10.1.2005-11.1.2005.

7 Orderbook Visualization Interface

This chapter gives a brief description of the orderbook visualization user interface. After a short introduction (Section 7.1) and a basic user interface overview (Section 7.2) all functions that can be accessed by the interface are described like data navigation (Section 7.3), data import/export (Section 7.4), layers (Section 7.5) and filters (Section 7.6).

7.1 Introduction

The orderbook visualization interface is designed to provide fast and intuitive data analysis functions. Implemented using the Microsoft WPF library the visualization window can be easily scaled and viewed on various display devices. Further the WPF library gives the designer much freedom to extend the user interface in various ways. Interactive zoom and filter functions enable users to get an in-depth impression of the dynamics in an open limit orderbook. Implemented data connectors provide data import and export functions for many data sources/destinations.
The typical work-flow when using the orderbook visualization interface can be described as follows:

1. Open orderbook visualization program
2. Load orderbook data (from database or CSV-text file)
3. Zoom/Navigate to area of interest
4. Filter orders to highlight specific trading patterns
5. Export visualization to image or data files

Altogether the orderbook visualization interface represents a viable tool to do explorative data analysis of orderbook data. Providing an in-depth view of the market microstructure users can visualize and understand many aspects of micro price dynamics and liquidity commonalities which are of big interest in recent market microstructure literature.

7.2 User Interface Description

7.2.1 Overview

The orderbook visualization interface shown in Figure 7.1 can be divided in 6 regions:

1. **Orderbook Visualization Menu:** The Orderbook Visualization Menu provides all functions which are also available from the tool bar for data navigation, import/-export functions, layer and filter purposes.

2. **Tool Bar:** The Tool Bar itself provides those functionalities in a more intuitive way through buttons and will be described in Section 7.2.2 in more detail.

3. **Database Tree View:** The interface area 3 is dedicated to the Database Tree View to list all available databases and instruments. Using the Database Tree View it is easy to change the displayed instrument quickly.

4. **Orderbook Visualization Box:** The actual orderbook data is visualized in the Orderbook Visualization Box and also plays an important part for data navigation and exploration.

5. **Filter Area:** The Filter Area shows all activated filters including filter parameters represented by check-boxes. Parameters can therefore be easily changed by simply (un-)checking a box accordingly.

6. **Info Bar:** Finally, the Info Bar shows all information about the currently loaded stock, the current cursor position in terms of time and limit and a button to show/hide the Filter Area.

7.2.2 Tool Bar

The Tool Bar provides all functions of the orderbook visualization interface. The functions can be divided in 3 main categories: data navigation, import/export and layer/filtering. A detailed description of all buttons in the Tool Bar is given by Figure 7.2.
The data navigation facilities include a switch between order exploration mode and zoom-in mode (Figure 7.2–buttons 1,2). Further a zoom out is provided (Figure 7.2–button 3). Data navigation facilities will be described in Section 7.3 in more detail.
Data import functions allow the import of order modifications either from a CSV file or from a database (Figure 7.2–buttons 4,5). Visualized orderbooks shown in the Orderbook Visualization Box can be either exported as an image file or directly printed (Figure 7.2–buttons 6,7). Import/export facilities will be further described in Section 7.4.

Additional layers can also be shown for either the grid or the bid/ask spread (Figure 7.2–buttons 8,9) and are part of the third Tool Bar category (see Section 7.5). This category also includes the addition/removal of order filters (Section 7.6) and a specialized tool for order sequence detection (see Chapter 8).

7.3 Data Navigation

For explorative data navigation the user can switch between order exploration mode and the zoom-in mode.

7.3.1 Order Exploration Mode

While the order exploration mode is activated (default on startup) users can get a quick overview of Orders inside specific OrderRectangles. By simply moving the mouse over a filled OrderRectangle a yellow selection rectangle is drawn. The height of the rectangle is always set to one as only orders with the specified limit are selected. Its width is determined by a loop through all OrderRectangles before and after the filled mouse-over rectangle. The loop is stopped when a consecutive OrderRectangle no longer contains any orders of the mouse-over rectangle whereby the start-bin and end-bin for the selection rectangle is determined. The width of the selection rectangle is therefore typically determined by the order having the longest lifetime.
By a mouse click on the previously selected area a new window pops up showing all orders and their attributes. Thus the user can get a detailed view of orders actually added to the selected OrderRectangles.

7.3.2 Zoom-In Mode

The zoom-in mode enables the most important data navigation functions and supports a detailed visualization of any time/limit area. While the left mouse button is pressed the user can draw a rectangle in the Orderbook Visualization Box representing any time/limit area of interest. A release of the mouse button immediately initiates a zoom in the selected area of interest. Internally the zoom process described in Section 6.6 is triggered including re-binning and filtering of orders. Figure 6.6 already showed visualizations of consecutive zoom states where the upper 2 states still include the zoom rectangle drawn in the Orderbook Visualization Box.

7.3.3 Zoom Out

The zoom-out function recovers the previous zoom state (if available). While the zoom-in mode is activated a zoom out can also be done by a right mouse-click.

7.4 Data Import/Export

The following sections describe the file import (Section 7.4.1) and database import (Section 7.4.2) functions. Also the data export functions are summarized in Section 7.4.3.

7.4.1 File Import

The file import function is able to import order modifications from CSV files. This feature has been actually implemented for visualization demo purposes only as no database installation or complete dataset is necessary. After a CSV file has been chosen from the file dialog data is imported using the CSVConnector class (see also Section 6.3).

7.4.2 Database Import

A click on the database import button (Figure 7.2–button 5) opens the Database Tree View window on the left side (see also Figure 7.1–area 3). The tree view shows all retrieved database names as parent nodes (indicated by blue symbol) which could be retrieved using the connection string settings stored in the app.config XML configuration file. Only if the database contains a valid structure as defined in Section 4.4.2 the node can be expanded and all Instrument names are shown. After an instrument of interest has been selected a click on the 'Load' button below the tree view starts the entire order visualization procedure described in Chapter 6.

7.4.3 Data Export

The already implemented data export functions are able to either store visualized orderbooks as image files or to directly print them.
A click on the 'Save' button (Figure 7.2–button 6) opens a file dialog where the location and file type can be selected. Image filetypes available are listed in Table 7.1. The implemented print function (Figure 7.2–button 7) enables the user to directly print the visualized orderbook including all visualization parameters on a DIN-A4 page.

7.5 Layers

In order to visualize orderbook data in context to various parameters two different layers have been implemented for data grids and bid/ask spreads. The layers are simply added to the Orderbook Visualization Box and can be selectively turned on and off. Also a simultaneous display of multiple layers is possible. The layers support various basic WPF Shape objects (like Lines, TextBoxes, Rectangles, Curves, etc.) and

Extension	Filetype Description
.bmp	Windows Bitmap
.jpg	JPEG
.gif	Graphics Interchange Format
.png	Portable Network Graphics
.tiff	Tagged Image File Format
.wmp	Windows Media Photo
.xps	XML Paper Specification

Table 7.1: Supported export image filetypes

could also be extended for other features like technical indicators.
The two following sections describe the data grid and bid/ask spread layer in more detail.

7.5.1 Data Grid

The data grid layer shows horizontal and vertical lines for price and limit values, respectively. The distance between price and limit lines is previously set to a fixed interval by the parameters vGridDensity and hGridDensity. The actual price values of horizontal lines are shown in text boxes on the left side; time values are drawn vertically at the bottom of the Order Visualization Box.
Trading sessions are also separated by thick vertical lines while horizontal text boxes at the bottom indicate each trading day.

7.5.2 Bid/Ask Spread

The bid/ask spread first introduced in Section 2.2 and already calculated in Section 5.3 is an essential orderbook feature to set buy and sell orders in relation to each other. Most importantly the price distance of an order to the spread determines its aggressiveness. Aggressive incoming orders on only one side of the book typically lead to considerable price movements and are therefore a very interesting topic to be studied. The bid/ask spread layer makes this important orderbook feature visible to the user though some preliminary calculations have to be done. The current layer implementation takes all orders added to the data structures (see Section 6.4.3) to calculate the spread. However, this approach leads to problems if many Market orders have been added to the book. It is not as accurate as the spread calculation done during the dynamic order reconstruction process in Section 5.3. A better solution would be to store the calculated spread (or other orderbook features) in Section 5.3 to an additional table in the reconstructed database which avoids its re-calculation and leads to

accurate results.

7.6 Filters

Filters broaden the explorative data analysis capabilities of the orderbook visualization interface. Basically any Order(Rectangle)-attribute which is detected using reflection (Section 6.7) can be filtered. A click on the 'Filter' button (Figure 7.2–button 10) shows a checkbox table containing all available filters. A simple 'checking' of a respective filter adds it to the Filter Area including all available parameters values. Multiple filters can be added and modified by the parameter check-boxes in the Filter Area at the bottom. By changing a filter parameter a FilterChanged event is triggered which initiates a filtering–visualization cycle as described in Section 6.2. This event is also fired when a filter is removed form the list.

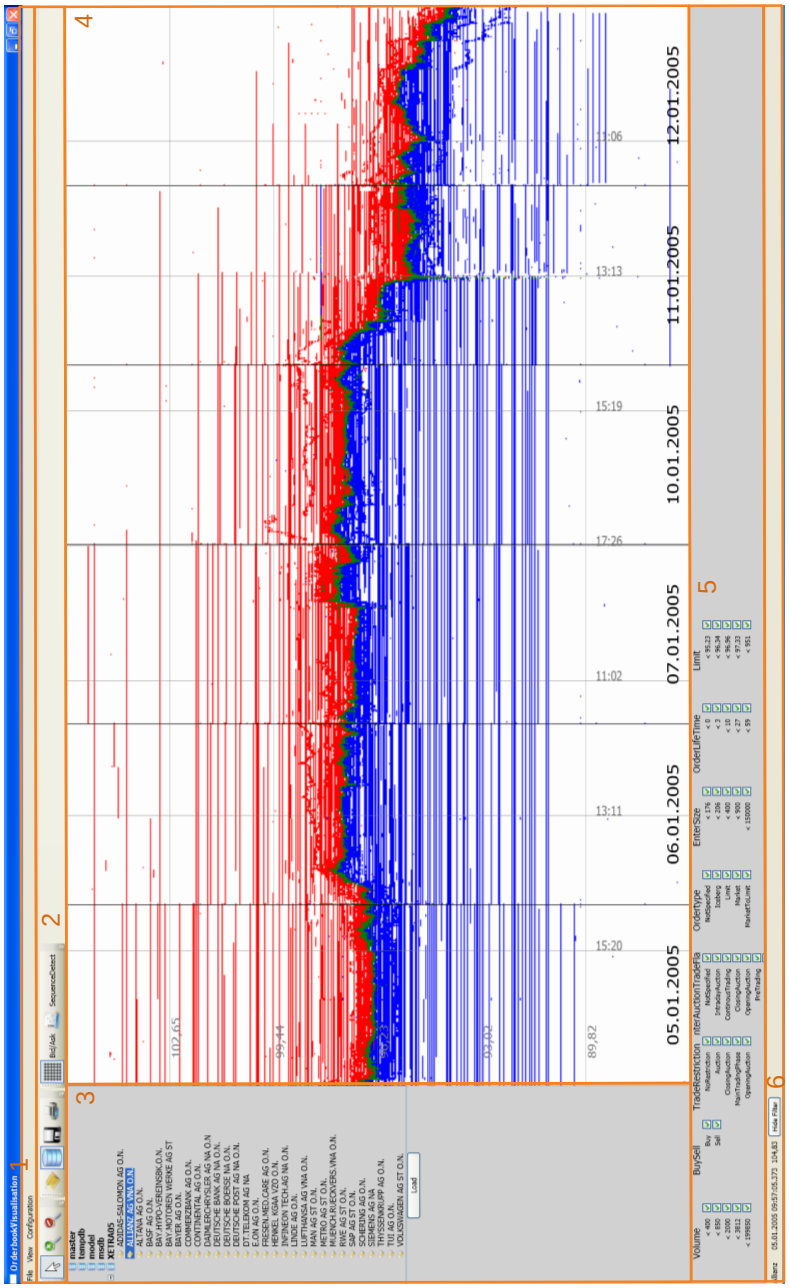

Figure 7.1: Orderbook Visualization User Interface

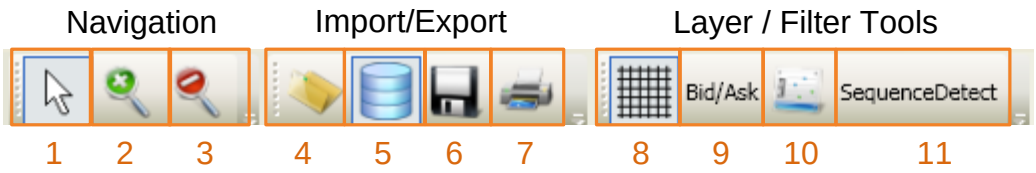

Figure 7.2: Tool Bar Figure

1. Order Exploration Mode
2. Zoom-In Mode
3. Zoom Out
4. Open File
5. Open Database
6. Save (Picture)
7. Print (Picture)
8. Show/Hide Visualization Grid
9. Show/Hide Bid/Ask Spread
10. Add/Remove Filters
11. Detect Order Sequences

8 Detection of Algorithmic Trading Patterns

An interesting application of the *Orderbook Visualization* package is the detection of algorithmic trading patterns in the orderbook. This chapter gives an overview of work related to this topic in Section 8.1 followed by a definition of detectable algorithmic fleeting orders (Section 8.2), detection strategy of algorithmic fleeting orders (Section 8.3) and finishes with some selected visualization of found structures (Section 8.4).

8.1 Relevant Work

Market microstructure literature already discussed in Section 1.4 presents numerous approaches to model orderbook dynamics. Most of those models assume that limit orders cannot be withdrawn or revised once submitted. Also argued by Fong and Liu [FL06], who have analyzed orderbook data from the Australian Stock Exchange, this assumption of previous theoretical models is simply not realistic as traders do monitor and change entered orders frequently. Empirical findings of high order cancelation rates on most major (electronic) stock exchanges like NYSE, LSE and Deutsche Börse AG's XETRA trading system give further evidence to this suggestion. Canceled orders even show a clustering of order lifetimes as shown by Prix [PLH07] et al. for orders on the XETRA trading system. Prix et al. in [PLH07] use the same DAX30 dataset which has been analyzed within this thesis (see Chapter 3) and found that even 68% of money volume has been canceled in the time period between 2005-1-5 and 2005-1-12. Order lifetimes are also reported to be clustered at multiples of one minute. Yeo [Yeo05] finds that more than 40% of NYSE orders have been canceled and that 95% of the survive less than five minutes.
These findings not only suggest that orders are frequently revised by cancelations and re-inserts but that this process is done in an automated manner by trading algorithms. Hasbrouck and Saar [HS07] first introduced the term fleeting orders appearing in the spread of Island ECN orders. Large [Lar04] suggests a theoretical model for fleeting orders.
In [PLH08] Prix et al. used a DAX30 dataset covering the complete trading activity on the XETRA trading system from 2006-12-6 to 2006-12-13 and from 2007-1-10 to 2007-1-17. They found that about 50% of all orders can be identified as fleeting orders

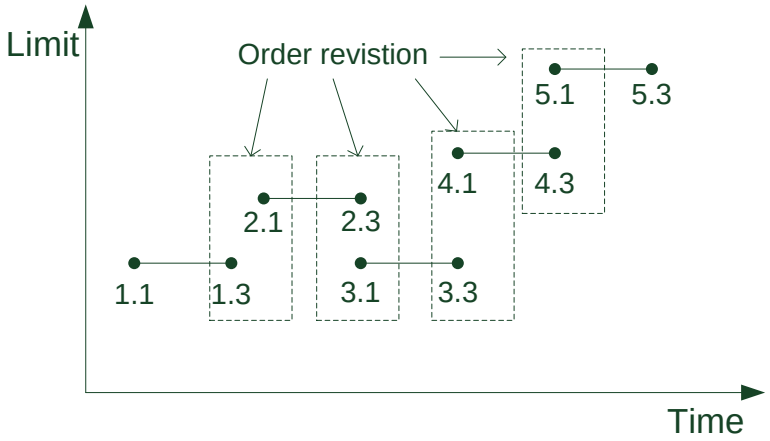

Figure 8.1: Example of a Fleeting Order
x.1 . . . (Re-)Insertion of Order x
x.3 . . . Deletion of Order x

and integrated in an order chain structure. Algorithmic fleeting orders are therefore a major source of (systematic) liquidity in the XETRA orderbook. By identifying those order structures properly a big share of the liquidity supply and demand behavior in the orderbook can be explained. Even liquidity forecasts can be done if fleeting orders are successfully detected and analyzed.

8.2 Definition of Algorithmic Fleeting orders

Algorithmic fleeting orders (or order chain structures) can be defined as frequently revised orders by cancelation of an already entered order and re-insertion of an order with changed attributes. As found by Prix et al. in [PLH08] re-inserted orders in chain structures typically modify only the limit of an order and leave other attributes (like volume/size) unchanged. The limit price of re-inserted orders mostly changes only by a small fraction. Because of the Price-Time priority rule defined in the XETRA market model (see Section 2.2) a revision of the limit price makes a cancelation of an order followed by a re-insertion of a new one necessary. Figure 8.1 shows an example of a fleeting order consisting of 5 canceled and re-inserted orders.

The fleeting order example shown in Figure 8.1 shows that cancelations and re-insertions of orders are typically not done at exactly the same timestamp. Revised orders can either be inserted shortly before or after the previous order in the chain. The

respective time window of cancelation and re-insertion is determined by the roundtrip time and indicated by dashed rectangles in Figure 8.1. The roundtrip time is defined as the time difference between the submission of an order modification and the confirmation by the exchange. Roundtrip times are currently reported to be 13ms on average. Considering the dataset under investigation we reckon that the roundtrip times in 2005 were higher by a factor of 3-4 which results as an average roundtrip time of 50ms.
The major drawback of those assumptions is that they cannot be tested directly as no information about the market participants who have entered the orders is available from the rebuilt orderbook dataset. Only the stock exchanges themselves have respective information and hold them under disclosure. However, very simple assumptions about fleeting orders' behavior and clear statistical and even visual evidence suggests that fleeting orders found in the dataset come from trading algorithms which continually revise entered orders.

8.3 Detection of Algorithmic Fleeting orders

Considering the strategy of algorithmic fleeting orders as defined in Section 8.2 a relatively simple algorithm can be written to detect them. For the detection of most obvious fleeting orders the implemented algorithm makes the following simple assumptions:

1. Fleeting orders consist of order chains where each order consists of an insertion modification 1 and is deleted by a deletion 3.
2. Insertion *ModificationTimestamp* of the successor order lies in a small time frame around the deletion *ModificationTimestamp* of previous chain element.
3. Re-inserted orders have the same *Volume* as previously deleted ones.
4. In case of an ambiguous order chain continuation (more orders have corresponding properties) select the successor with the smallest limit price difference to the predecessor.

While point 1 and 2 are considerable assumptions for the detection of fleeting orders, 3 and 4 represent simplifications and leave room for discussion.
The equal-volume assumption in 3 can miss fleeting orders which change volumes but should detect most chains. It is also supported by the findings in [PLH08] where almost all fleeting order chains left the order volume unchanged.
The assumption in point 4 handles the problem of ambiguous order chain continuation and selects successors according to its limit price. Various ambiguities can be found in

the orderbook using the proposed algorithm — especially when high trading volumes enter the book. Evidence from [PLH08] suggests that limits of re-inserted orders are only changed by a small amount whereas no exact solution can be found as traders are entering orders anonymously into the book.
The detection algorithm is designed to achieve a fast detection performance and takes the input parameters time-window and size-tolerance. According to point 2 and 3 time-window has been set to one second, size-tolerance to zero. Simply put, the algorithm extracts all deleted order objects from the order data structure and sorts them in 2 lists; list1 is sorted by the insertion *ModificationTimestamp*, list2 by the deletion *ModificationTimestamp*. While looping through all orders in list1 the algorithm tries to match all orders (by insert *ModificationTimestamp*) against orders in list 2 (by deletion *ModificationTimestamp*) within the defined time-window and by taking the size-tolerance parameter into account. If ambiguities occur it is proceeded according to point 4.
Although the presented algorithm is quite simple its reasonable performance make it a viable tool for the detection of fleeting orders in the orderbook.

8.4 Visualization of selected Chain Structures

Consequently in line with the filter framework (Section 6.7) the sequence detection algorithm (Section 8.3) is also implemented as a filter. It is therefore possible to interactively turn it on/off, even in combination with other Order- or OrderRectangle filters.
For the 'visual' detection of fleeting orders the combination of the *EnterOrderSize* Order filter (filters by size indicated in insert order modification) and/or the *Volume* OrderRectangle filter (filters aggregated order-bin volume) with the detection algorithm has proved to work best. The following sections describe various interesting fleeting order patterns detected with the orderbook visualization interface.

8.4.1 Buy-in-Market chains

Buy-in-Market chains can be described as limit buy orders which are continually revised by increasing its limit. As common to all fleeting orders defined in Section 8.2, the limit increase is done by deletion of the inserted order and insertion of a new order with the same volume and a higher limit price.
Traders who use this pattern obviously want to continually increase the probability of an order execution in a given time interval and therefore place the limit closer to the bid/ask spread. Figure 8.2 shows 3 different Buy-in-Market chains at different volumes found in the SAP orderbook. These are relatively small volume chains; chain 1 has a volume of 67 shares, chain 2 of 10- and chain 3 100 shares. For the selected

Figure 8.2: Buy-in-Market order example
3 parallel Buy-in-Market order chains in the orderbook of the SAP stock
Time-Period: 16:21:10–17:08:34; Limit-Interval: 129.87–132.56
Filter settings: EnterSize (quintiles 0<x<100,100<x<205), Sequence Detect

visualization window, the limit price is increased in chain 1 from 131.20 to 131.43, in chain 2 from 129.92 to 131.68 and in chain 3 from 131.52 to 131.92.

8.4.2 Sell-in-Market chains

Analogous to Buy-in-Market, Sell-in-Market chains continually decrease the limit of a sell limit order. The strategy behind can be simply described as 'sell volume X in defined time frame'. Figure 8.3 shows a very clear Sell-in-Market in the orderbook of the Deutsche Börse AG stock. The sell limit market chain has a volume of 15 shares and its limit is decreased in the selected time frame from 46.96 to 45.40.

8.4.3 CIC order chains

Constant-Initial-Cushion (CIC) order chains have already been discovered by Prix et al. [PLH07] and can easily be detected with the orderbook visualization interface. According to their definition, CIC order chains consist of buy and sell limit orders, i.e. of coincidental order placements on both sides of the market, where the bids and asks have the same order size. The name 'CIC orders' is motivated by the observed order limits as all bids (asks) of CIC order chains have the same constant cushion at insertion. The cushion is defined as

$$\text{cushion} = \begin{cases} \text{best bid} - \text{bid order limit}, & \text{for a bid} \\ \text{ask limit} - \text{best ask limit}, & \text{for an ask} \end{cases}$$

By the insertion of bids and asks the CIC strategy assumes that prices fluctuate around a mean price. If the stock price falls below the CIC order cushion for a bid, the stock is purchased. Otherwise, if the price increases above the CIC order cushion the stock is sold. Profits are therefore made by fishing profitable roundtrip times. Please refer to [PLH07] for a more precise (statistical) analysis of CIC orders.
Figure 8.4 shows a very obvious CIC order chain in the orderbook of the Allianz stock. It can be seen that the observed CIC order pattern can be found over almost the entire investigated trading week of the dataset.

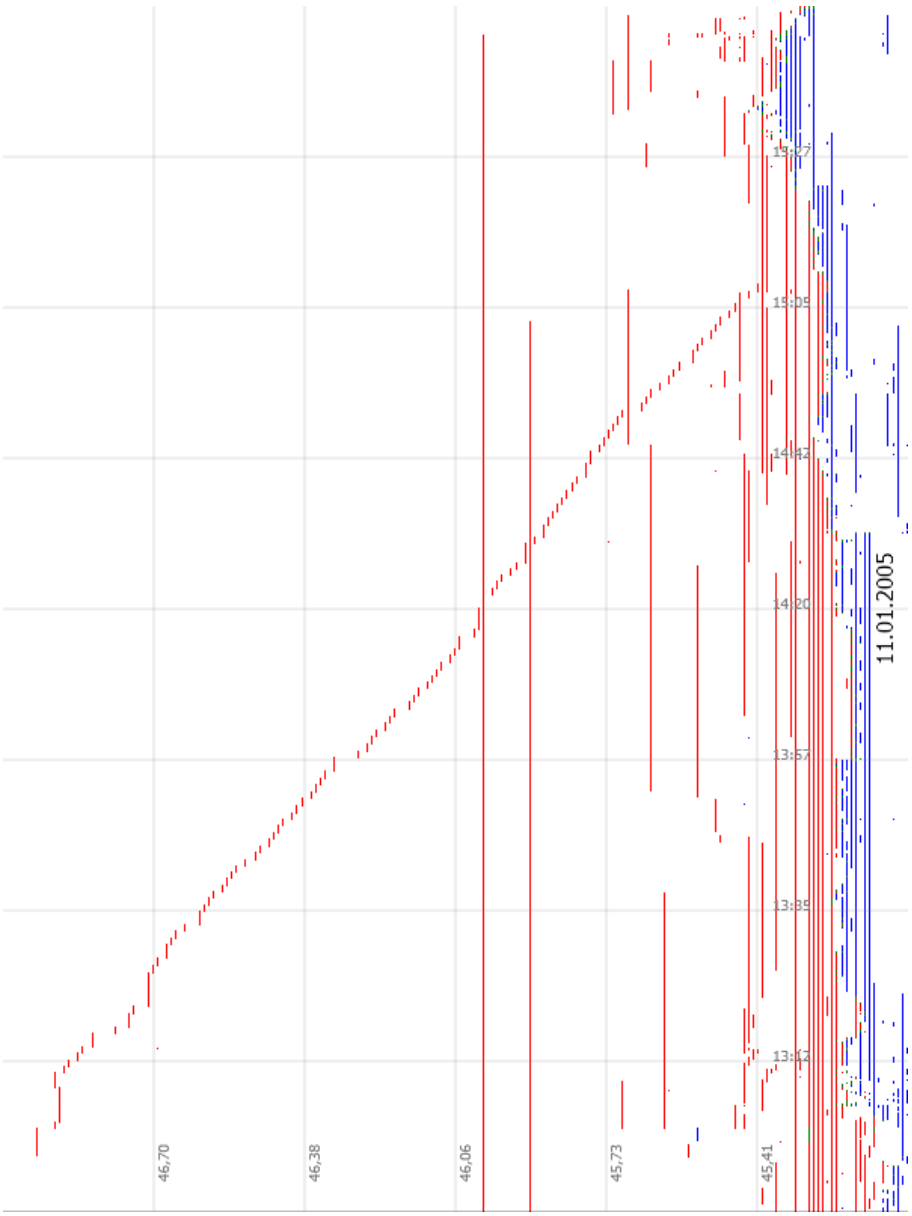

Figure 8.3: Sell-in-Market order example
Sell-in-Market order chain in the orderbook of Deutsche Börse AG stock
Time-Period: 12:50:04–15:49:24; Limit-Interval: 45.09–47.03
Filter settings: Sequence Detect

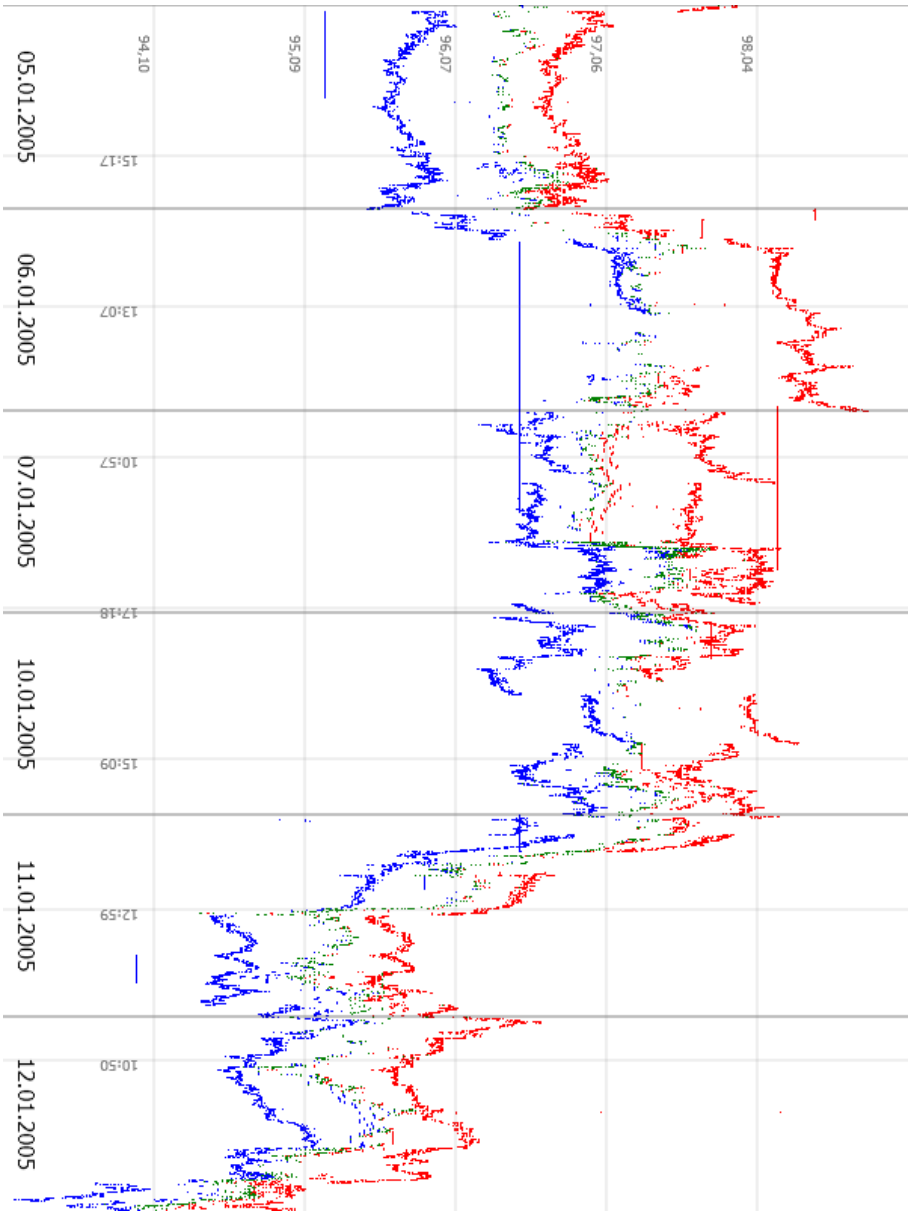

Figure 8.4: CIC order chain example
CIC order chain example in the orderbook of Allianz AG stock
Time-Period: 2005-1-6–2005-1-12; Limit-Interval: 93.08–99.22
Filter settings: EnterSize (quintile 176<x<206), Sequence Detect

9 Conclusion and Outlook

The increasing availability of real-time and historical high-frequency orderbook data allow an in-depth analysis of orderbook states and price dynamics in the market microstructure. Thanks to Deutsche Börse AG the presented software packages for reconstruction and visualization of orderbook data could be tested on the complete rebuilt orderbooks of all DAX30 stocks traded on the electronic XETRA system.
By using the implemented *Orderbook Engine* library data cleaning and reconstruction of orderbook states can be done at a very high performance. The resulting dataset is consistent and can be used for various data analysis purposes as well as market simulations.
The *Orderbook Visualization* package makes it possible to quickly navigate through very large orderbook datasets. Combined with numerous filters preliminary explorative data analysis can be done immediately. The intuitive user interface provides tools to highlight many aspects of the orderbook. Filters, Layers and Detection Algorithms can easily be turned on and off and used in combination. Therefore a tremendous freedom is given when visualizing rebuilt orderbook states. Further the step-less zoom function allows views of the orderbook in a specified market at the macro level as well as on the micro level on a tick-by-tick basis.
Even algorithmic trading patterns can visually be identified by combining various filters and sequence detection algorithms .
The flexible object model implemented in C# lets plenty of room for further improvements. An extension of already defined data connectors could be used to import/export orderbook datasets from/to various other sources/destinations. Data exploration capabilities could be extended by implementing further filters and detection algorithms. The WPF graphical framework used makes numerous other visualizations possible — even in 3D. Last but not least, the already implemented orderbook reconstruction functions could be used to extend the Orderbook Visualization Interface to a 'live' trading simulator to e.g. backtest trading strategies or detect systemic liquidity and hidden volumes in an environment which comes very close to reality.

Bibliography

[AIT06] AITE. Algorithmic trading 2006: More bells and whistles [online]. October 2006. Available from: http://www.aitegroup.com/reports/200610311.php [cited December 3, 2008].

[Aut08] AutomatedTrader. Automated trader [online]. October 2008. Available from: http://www.automatedtrader.net/ [cited December 3, 2008].

[BLGG05] Helena M. Beltran-Lopez, Pierre Giot, and Joachim Grammig. Commonalities in the Orderbook. *SSRN eLibrary*, 2005.

[Bro08] Allen Browne. Sql data types [access 2007 develpers reference] [online]. 2008. Available from: http://msdn.microsoft.com/en-us/library/bb208866.aspx [cited December 3, 2008].

[CF06] German G. Creamer and Yoav Freund. A boosting approach for automated trading. In *Proceedings of the Data Mining for Business Applications Workshop on International Conference on Knowledge Discovery and Data Mining (KDD)*, Philadelphia, 2006.

[DBG08] DBG. Deutsche börse group, level 2 data [online]. 2008. Available from: http://deutsche-boerse.com/dbag/dispatch/de/binary/gdb_content_pool/imported_files/public_files/10_downloads/50_informations_services/10_market_data_dissemination/11_information_products/10_spot_market/Spot_Market_Germany.PDF [cited December 3, 2008].

[dWdH03] R. de Winne and C. de Hondt. Rebuilding the limit orderbook on euronext or how to improve market liquidity assessment. *Unpublished working paper*, 2003.

[Eki05] Cumhur Ekinci. Limit Orderbook Reconstruction And Beyond: An Application To Istanbul Stock Exchange. *EconWPA Finance*, (0510025), 2005.

[FG05] Stefan Frey and Joachim Grammig. Liquidity Supply and Adverse Selection in a Pure Limit Orderbook Market. *SSRN eLibrary*, 2005.

[FKK01] Thierry Foucault, Ohad Kadan, and Eugene Kandel. Limit orderbook as market for liquidity [online]. 2001. Available from: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=269908.

[FL06] Kingsley Y. Fong and Wai-Man R. Liu. Limit Order Cancellation and Revision Activities. *SSRN eLibrary*, 2006.

[GHR04] Joachim Grammig, Andréas Heinen, and Erick W. Rengifo. Trading Activity and Liquidity Supply in a Pure Limit Orderbook Market. An Empirical Analysis Using a Multivariate Count Data Model. *SSRN eLibrary*, 2004.

[Glo94] Lawrence R. Glosten. Is the electronic open limit orderbook inevitable? *Journal of Finance*, (49):1127–61, 1994.

[Gro03] Deutsche Börse Group. Xetra auction plan [online]. November 2003. Available from: http://deutsche-boerse.com/dbag/dispatch/en/kir/gdb_navigation/trading_members/12_Xetra/40_Auction_Plan [cited December 3, 2008].

[Gro05] Deutsche Börse Group. Xetra: Leading international trading platform [online]. December 2005. Available from: http://www.google.at [cited December 3, 2008].

[Gro08a] Deutsche Börse Group. Xetra: Leading international trading platform [online]. April 2008. Facts and Functionalities. Available from: http://deutsche-boerse.com/dbag/dispatch/en/binary/gdb_content_pool/imported_files/public_files/10_downloads/31_trading_member/10_Products_and_Functionalities/20_Stocks/BR_Xetra.pdf [cited December 3, 2008].

[Gro08b] Deutsche Börse Group. Xetra trading platform [online]. 2008. Features. Available from: http://deutsche-boerse.com/dbag/dispatch/en/kir/gdb_navigation/technology/20_Applications/10_Trading/10_Xetra_Trading_Platform?horizontal=page3 [cited December 3, 2008].

[GST04] Peter Gomber, Uwe Schweickert, and Erik Theissen. Zooming in on Liquidity. *SSRN eLibrary*, 2004.

[Har03] Larry Harris. *Trading and Exchanges. Market Microstructure for Practitioners*. Oxford University Press Inc, 2003.

[Har04] Gurushyam Hariharan. News mining agent for automated stock trading. Master's thesis, University of Texas, Austin, 2004.

[Har08] Thom Hartle. Using orderbook data to improve automated model performance [online]. 2008. Available from: http://www.automatedtrader.net/automated-trader-strategies-737.xhtm [cited December 3, 2008].

[Has07] Joel Hasbrouck. *Empirical Market Microstructure*. Oxford University Press Inc, 2007.

[HS07] Joel Hasbrouck and Gideon Saar. Technology and Liquidity Provision: The Blurring of Traditional Definitions. *SSRN eLibrary*, 2007.

[ISE07] Deutsche Börse Group Irish Stock Exchange. Xetra release 9.0 functional description [online]. December 2007. Release 9.0. Available from: http://www.ise.ie/getFile.asp?FC_ID=1086&docID=501 [cited December 3, 2008].

[KG03] Robert Kissell and Morton Glantz. *Optimal Trading Strategies: Quantitative Approaches for Managing Market Impact and Trading Risk*. Oxford University Press Inc, 2003.

[KKMO04] Sham M. Kakade, Michael Kearns, Yishay Mansour, and Luis E. Ortiz. Competitive algorithms for vwap and limit order trading. In *EC '04: Proceedings of the 5th ACM conference on Electronic commerce*, pages 189–198, New York, NY, USA, 2004. ACM Press. doi:http://doi.acm.org/10.1145/988772.988801.

[KO03] Michael Kearns and Luis Ortiz. The penn-lehman automated trading project. *IEEE Intelligent Systems*, 18(6):22–31, 2003. doi:http://doi.ieeecomputersociety.org/10.1109/MIS.2003.1249166.

[Lar04] Jeremy H. Large. Cancellation and Uncertainty Aversion on Limit Order Books. *SSRN eLibrary*, 2004.

[LG] Francois-Serge Lhabitant and Greg N. Gregoriou, editors. *Stock Market Liquidity, Implications for Market Microstructure and Asset pricing*.

[LPH06] Otto Loistl, Johannes Prix, and Michael Hutl. Doubly stochastic markov process: A causal approach to modelling cadlag market event time series. In *9th Conference of the Swiss Society for Financial Market Research*, Zurich, Switzerland, April 2006.

[LPH07] Otto Loistl, Johannes Prix, and Michael Hutl. Best execution and the xetra orderbook. In *Campus For Finance – Research Conference*, Campus For Finance - Research Conference, Vallendar, Germany, January 2007.

[LSE08a] LSE. London stock exchange, level 2 data [online]. 2008. Available from: http://www.londonstockexchange.com/en-gb/products/marketdata/Information+Products/realtimedata.htm [cited December 3, 2008].

[LSE08b] LSE. London stock exchange, rebuild orderbook [online]. July 2008. Available from: http://www.londonstockexchange.com/en-gb/products/informationproducts/historic/rebuild.htm [cited December 3, 2008].

[Mal07] Azeem Malik. etradplat [online]. 2007. Facts and Functionalities. Available from: http://www.etradplat.com/ [cited December 3, 2008].

[Mat08] The MathWorks. The r project for statistical computing [online]. September 2008. Available from: http://www.mathworks.de/products/matlab/index.html.

[Md08] Markt-daten.de. Daten [online]. October 2008. Available from: http://www.markt-daten.de/daten/daten.htm [cited December 3, 2008].

[Mic05] Microsoft. Microsoft sql server 2005 express edition [online]. November 2005. Available from: http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=220549b5-0b07-4448-8848-dcc397514b41.

[Mic07] Microsoft. Msdn sql server development center - data types (transact-sql) [online]. September 2007. Available from: http://msdn.microsoft.com/en-us/library/ms187752(SQL.90).aspx [cited December 3, 2008].

[Mic08a] Microsoft. .net 3.5 [online]. 2008. Available from: http://www.microsoft.com/downloads/details.aspx?FamilyId=333325FD-AE52-4E35-B531-508D977D32A6&displaylang=en [cited December 3, 2008].

[Mic08b] Microsoft. Visual c# 2008 express edition [online]. 2008. Available from: http://www.microsoft.com/germany/Express/product/visualcsharpexpress.aspx [cited December 3, 2008].

[Mon08] Mono. The mono project [online]. 2008. Available from: http://www.mono-project.com/Main_Page [cited December 3, 2008].

[NAS08a] NASDAQ. Model view [online]. October 2008. Available from: http://www.nasdaqtrader.com/Trader.aspx?id=ModelView [cited December 3, 2008].

[NAS08b] NASDAQ. Nasdaq total view [online]. 2008. Available from: http://www.nasdaqtrader.com/Trader.aspx?id=TotalView [cited December 3, 2008].

[NYS08a] NYSE. New yord stock exchange, openbook [online]. 2008. Available from: http://www.nyxdata.com/openbook [cited December 3, 2008].

[NYS08b] NYSEEuronext. Openbook history [online]. July 2008. NYSE Orderbook Data. Available from: http://www.nyxdata.com/nysedata/default.aspx?tabID=743 [cited December 3, 2008].

[O'H95] Maureen O'Hara. *Market Microstructure Theory*. Blackwell, 1995.

[Par98] Christine A. Parlour. Price dynamics in limit order markets. *Review of Financial Studies*, (11):789–816, 1998.

[PLH07] Johannes Prix, Otto Loistl, and Michael Hutl. Algorithmic trading patterns in xetra orders. *European Journal of Finance*, 13(8):717–739, 2007.

[PLH08] Johannes Prix, Otto Loistl, and Michael Hutl. Chain-structures in xetra order data. campus for finance. In *Research Conference*, Vallendar, Germany, January 2008.

[PLHK08] Johannes Prix, Otto Loistl, Michael Hutl, and Emanuel Albin Kopp. Cross-sectional liquidity interactions from intraday perspectives. In Soares/Pina/Catalao-Lopes, editor, *New Developments in Financial Modelling*, pages 213–244, Cambridge, 2008. Cambridge Scholars Publishing.

[Pol07] Andrew Pole. *Statistical Arbitrage: Algorithmic Trading Insights and Techniques*. Wiley and Sons, 2007.

[R08] R. The r project for statistical computing [online]. September 2008. Available from: http://www.r-project.org/index.html.

[Ran] Angelo Ranaldo. Intraday market dynamics around public information arrivals. In Francois-Serge Lhabitant and Greg N. Gregoriou, editors, *Stock Market Liquidity, Implications for Market Microstructure and Asset pricing*.

[SCR] Avanidhar Subrahmanyam, Tarun Chordia, and Richard Roll. Commonality in liquidity. In Francois-Serge Lhabitant and Greg N. Gregoriou, editors, *Stock Market Liquidity, Implications for Market Microstructure and Asset pricing*.

[SR05] G.C. Silaghi and V. Robu. An agent strategy for automated stock market trading combining price and orderbook information. In *Proceedings of the IEEE Congress on Computational Intelligence Methods and Applications (CIMA 2005)*, pages 189–198, 2005. work in progress report.

[SSSK06] Harish Subramanian, Ramamoorthy Subramanian, Peter Stone, and Ben Kuipers. Designing safe, profitable automated stock trading agents using evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, July 2006.

[Wag] Niklas Wagner. On the dynamics of market illiquidity. In Francois-Serge Lhabitant and Greg N. Gregoriou, editors, *Stock Market Liquidity, Implications for Market Microstructure and Asset pricing.*

[WFE08a] WFE. World federation of exchanges, time series [online]. September 2008. Historical Time Series from 1997-2007. Available from: http://www.world-exchanges.org/WFE/home.asp?menu=421 [cited December 3, 2008].

[WFE08b] WFE. World federation of exchanges, value of share trading [online]. September 2008. Historical Time Series from 1997-2007. Available from: http://www.world-exchanges.org/WFE/home.asp?menu=27 [cited December 3, 2008].

[Yeo05] Wee Yong Yeo. Cancellations of Limit Orders. *SSRN eLibrary*, 2005.

List of Tables

List of Figures