CCFEA

University of Essex

MSc Financial Software Engineering

Dissertation

An extensible Open Source framework for backtesting algorithmic trading strategies within the Marketcetera platform

ABSTRACT

Numerous proprietary software applications provide backtesting functionality, but they often have expensive license fees. I developed a basic framework for *Marketcetera*, an open source automated trading platform, to historically backtest algorithmic trading strategies. The framework offers an exchange, that simulates the market, and a module for *Marketcetera*, which receives market data from the exchange and parses it to internal events. Strategies running within *Marketcetera* can place orders on the exchange, where they are matched against the current order book based on historical data. Market data from any data source can be utilised, through custom data loaders. Simple backtest reports are provided, but the generic reporting functionality allows the user to add custom reports. The framework can potentially integrate with any FIX enabled application, and is released as open source thus is free for everyone to use and modify. This paper exemplifies the use of the framework by analysing a simple trading strategy, and suggests future work for the framework including new features and validation methods.

Author:
Daniel Schiermer

Supervisor:
Dr. Steve Phelps

Sourceforge project page:
`http://sourceforge.net/projects/catsbf/`

SVN check-out:
`http://svn.code.sf.net/p/catsbf/code/trunk`

30-08-2011

Contents

1 Introduction

Backtesting of algorithmic trading strategies provides a way of evaluating how a strategy performs under a variety of market circumstances and trading situations before it is deployed [Bat07]. Based on the backtest results, one might be able to tweak the strategy parameters, to make it perform better, or completely discard the strategy. But, there are common backtesting pitfalls that can produce erroneous results. Some of these pitfalls are briefly discussed in section 2.1.

A number of proprietary software applications provide backtesting functionality (e.g. [KO03], [Mat09] and [PV08]), but they require a license. This license is often very expensive, and still only leaves the user with the right to use the software; it does not give access to the source code, or permission to modify the software to user-specific needs. In order to use the application it is generally necessary to write strategies using application specific APIs[1], or even a custom scripting language. Unless the same software is used for both backtesting and trading, one is forced to rewrite the strategies using the trading application's specific API. This is a big overhead, and introduces extra cost in terms of working hours, and maybe multiple license fees.

Most firms are always looking to cut cost, especially following the financial crisis starting in 2007 and continued in the years following [Org09].

Using *open source* software instead of buying expensive licenses, or developing applications from scratch in-house, provides companies with a way of cutting cost, and can reduce the time to go-live.

The objective of this dissertation is to develop an *open source* backtesting framework for algorithmic trading strategies within the *open source* automated trading platform *Marketcetera*. The full name of the framework is *CATSBF CCFEA Algorithmic Trading Strategy Backtesting Framework*, but will from this point be referred to as *CATSBF*. The source code is released under the *GNU General Public License* [GNU11], and is available

[1] An API - Application Programming Interface is a code "vocabulary" specifying how to communicate with and run methods within the application [Wik11a].

from *Sourceforge* at `https://sourceforge.net/p/catsbf/code/`.

Open source is a methodology where the source code is published and made available to the public. Anyone is allowed to copy, modify and redistribute the source code, without restrictions. This freedom allows the software to evolve through community cooperation. It is the author's hope that releasing *CATSBF* as *open source* results in users improving upon the product causing it to continuously evolve, and in time become a rigorous backtesting framework for algorithmic trading strategies. Hopefully it will reach a large enough maintenance community, so it can be submitted to the *Marketcetera Open Labs*, a collection of user maintained contributions to the *Marketcetera* platform[2].

CATSBF performs backtesting against historical high-frequency data, and it comes with an adapter to load market data from the *London Stock Exchange (SETS)*[3]. It is implemented with extensibility in mind, to ensure that users easily can implement custom market data loaders, to load data from their own sources. Users can also implement custom order execution strategies, if they are not satisfied with the one provided.

Reporting is another extensible feature. Custom backtesting reports can easily be added, to provide any information needed about the backtest. *CATSBF* contains two common reports, one that logs all actions made by the client (bids, offers and trades), and one that reports summarised data, such as total number of bought/sold shares, average buy/sell price and total profit/loss.

Marketcetera is already an advanced trading platform which offers a lot of trading functionality, but it lacks integrated backtesting, and such an extension is on their list of suggested user contributions [Mar11g]. This dissertation aims to fill this gap in *Marketcetera*'s features.

A csv data adapter, to replay historical data, already exists for *Marketcetera* [Mar11h],

[2] *Marketcetera Open Labs* is an open experimental area, where users can submit extensions, modules and new components to *Marketcetera*. The code has to be reviewed and approved by *Marketcetera* before it is added to the *Open Labs* homepage [Mar11i]

[3] *SETS - Stock-exchange Electronic Trading Service* [Exc11] is the *London Stock Exchange*'s fully electronic trading platform. *SETS* deliveres raw transaction data in CSV files. See [RP11] for an overview of how the exchange is implemented.

but it has strict rules about the data format, and does not provide backtesting functionality. Any such functionality would have to be implemented in the trading strategy, whereas nothing "test" specific is needed in the strategy with *CATSBF*. Additionally does *CATSBF* not have any data format, or data source, requirements; custom market data adapters can be implemented and applied. One could e.g. implement an adapter that fetches data from a database.

CATSBF is implemented as an exchange, to which *Marketcetera* connects and trade in exactly the same way it would connect and trade on any other exchange. The idea behind this solution is to completely separate the backtesting logic from the strategy, so nothing backtesting-specific has to be declared in the strategy. This results in a more realistic test bed, and allows successful trading strategies to be deployed very easily; it is only a matter of a few configuration changes.

Marketcetera is implemented in Java [Jav11] and uses the FIX Protocol[4] to integrate with other applications. *CATSBF* is implemented using the same technologies as *Marketcetera* to get the best integration.

The *open source* property of *Marketcetera* and *CATSBF*, and the use of open standards, FIX, does not only contribute to firms saving money on license fees and development. It also allows researchers to develop advanced trading strategies in an existing IDE [Wik11d] and backtest them in a realistic trading environment, where they can easily deploy the successful strategies.

Due to the use of the FIX Protocol, and the fact that *CATSBF* is a stand alone application, it is possible for other applications to utilise the framework as well. Any application that supports the FIX Protocol can potentially integrate with *CATSBF*. This means that trading strategies implemented in any FIX enabled application could be backtested with this framework.

CATSBF provides a basic backtesting framework, and it is left up to the user to modify

[4]The FIX protocol, *Financial Information eXchange Protocol*, is an open standard financial message specification, that is widely used in financial applications, such as broker and trading applications [Org11].

it to meet his needs.

The scope of this dissertation is to implement *CATSBF* and give a description of the implementation, along with a brief description on how to backtest, and discuss related work. Furthermore it includes an overview and evaluation of *Marketcetera*.

The remainder of this paper is organised as following. The next section gives a brief discussion on backtesting tips and pitfalls and reviews two related projects; *The Penn-Lehman Automated Trading Project* [KO03] and a paper on *Evaluation of automated-trading strategies using an artificial market* [Mat09]. Section 3 gives an overview of *Marketcetera* and an evaluation of *Marketcetera* as an automated trading platform. Section 4 describes the implementation of the framework and explains any simplifications and assumptions made. Section 5 suggests future work. Section 6 analyses a trading strategy based on a *CATSBF* backtest and section 7 concludes.

2 Related Work

Many proprietary trading applications provide functionality to backtest algorithmic strategies, [Sys11], [MSC11], [Mul11] and [Alp11], but to the author's knowledge, a historical, open source, backtesting framework, that can integrate with multiple automated trading applications does not exist.

This is not to say that similar projects do no not exist. Studies on testing trading strategies have been conducted on several occasions, [KO03], [Mat09] and [PV08]. These studies often take the approach of trying to find the best algorithmic trading strategy and, for that purpose, provide a test bed to evaluate the strategies' performance and compare the results.

This section describes one such project, the *Penn-Lehman Automated Trading Project* [KO03], and a comparison study of backtesting methods, by K. Izumi, F. Toriumi, H. Matsui [Mat09], which compares historically backtested trading strategies from the Kaburobo contest (more on Kaburobo in section 2.3), and the results from a backtest on their sim-

ulated exchange. First a brief discussion of how to backtest, and what pitfalls to avoid.

2.1 Backtesting Tips and Pitfalls

Markets are continuously changing, thus making it important to backtest trading strategies under various market circumstances. Thorough testing involves testing the strategy in both *bull* and *bear* [Gen01, pp. 77] markets, and under normal and extreme trading conditions [PV08].

When a strategy is backtested, it sometimes happen that it performs well in the test, but ceases to perform when deployed. This "over-performance" during the backtest, might be due to the backtest having different market conditions than the actual market, but often it is due to the following biases being present in the evaluation process [KK95].

- **Pre-test bias:** If the trading strategy is tested against the same data from which it was derived, it is likely to perform well.
- **Trading cost bias:** The failure of not taking implicit and explicit trading costs into account. Explicit costs being; commission and bid/ask spread. Implicit costs being the trading strategy's impact on the price, large orders will often in real markets result in unfavourable price movements.
- **Look-ahead bias:** Allowing the strategy to trade based on "future" information, such as comparing the current price with the daily closing price. This gives unrealistic results, because it would not be possible in real markets.
- **Survivor-ship bias:** Restricting the backtest of a portfolio strategy to only include stocks that survived until the end of the test period, thus leaving out delisted stocks and stocks that went bankrupt, tends to yield above average returns because the strategy is tested in an unrealistic scenario.

Kan and Kirikos [KK95] describes theses biases in more detail and discuss their impact on the test results.

K. Izumi, F. Toriumi, H. Matsui [Mat09] are able test the market impact of trading strategies, thus if the strategy under test is suspected to have high market impact, *trading cost bias*, it might be better to test it using an artificial market.

2.2 Penn-Lehman Automated Trading Project

Penn-Lehman Automated Trading Project (PLAT) [KO03] is a study of finding the best automated trading strategies, through contests. PLAT was conducted as a partnership between the University of Pennsylvania and Lehman Brothers' Proprietary Trading Group in New York City.

For the PLAT project the authors investigated a broad range of automated trading strategies in financial markets by carrying out a number of competitions between automated clients. To compare the strategies they used a realistic test bed for algorithmic trading strategies. The strategies were developed by a number of students from the University of Pennsylvania, and other universities, and all types of strategies were welcome, such as agent-based and statistical models.

The centrepiece of the project, the *Penn Exchange Simulator (PXS)*, was used to evaluate the trading strategies. *PXS* relates very much to *CATSBF*, as it is a market simulator that merges automated client orders with real-world data. *PXS* backtest strategies based on market data for the *Microsoft (MSFT)* stock, as traded on NASDAQ. It automatically calculates profits, losses and other quantities of interest. *CATSBF* is similar in the way that it only handles trades on one stock, but this can be any stock, from any data source (more on this in section 4).

PXS runs in one of two modes, *Live* or *Historical* mode. In *Live* mode market data is pushed in real time from NASDAQ to the clients, in *Historical* mode archived market data is replayed.

PXS has its own client API that contains a set of functions and data structures that permit placement and withdrawal of orders, and the computation of market information,

including both information that is generally interesting for all clients, and more client specific information such as the clients cash and share holdings.

The PLAT project is no longer active, and the Lehman Brothers went bankrupt in 2008.

2.3 Evaluation of Automated-Trading Strategies using an Artificial Market

K. Izumi, F. Toriumi, H. Matsui [Mat09] studied evaluation of automated trading strategies using different backtesting methods. In their study they evaluated automated trading strategies using an artificial market and compared the test results against results from a conventional historical backtest.

Based on a standardised electronic trading protocol, the FIX protocol, they constructed a new platform for artificial market simulation, and used it to evaluate automated trading strategies.

They reused automated trading strategies that earlier participated in an automated trading competition called *Kaburobo*, held by the Trade Science Corporation [Kab11c].

Kaburobo is a contest that bears a lot of resemblance to the *PLAT* project [KO03]. Users submit an automated trading strategy, developed using the Java based *KaburoboSDK* [Kab11b] or developed using a graphical user interface (GUI) that is provided by the organisers. When submitted the strategy is evaluated against historical data from the Tokyo Stock Exchange. Initially each agent is given $50m$ Yen to invest among all stocks in the Nikkei-225 index and 300 other stocks with larger trading volumes. The agents can react to the market events during the backtest, and autonomously change their strategy, but no manual intervention is allowed.[5]

At the end of the contest total profits and losses are calculated subtracting expenses, such as commissions.

[5]For more info on *Kaburobo* see [Kab11a] or the explanation in [Mat09].

The main focus of the study is the development of the artificial market to evaluate automated trading strategies, and to compare the two backtesting methods.

The system was developed using the FIX protocol for the communication between the server and the agents. Furthermore they implemented a wrapper that translates *KaburoboSDK* messages to and from FIX messages, which enables *Kaburobo* agents to trade on the artificial market.

They compared the test results obtained from the *Kaburobo* contest with the results from the artificial market. These results turned out to be very different, and they concluded that the artificial market provided better information than the conventional historical backtest, because, among other things, the artificial market could test the market impact of the trading strategies.

2.4 Relations to *CATSBF*

PLAT provides functionality to backtest automated trading strategies against historical data, like *CATSBF*. *PXS* is an extensive test bed for automated trading strategies, but is built with the purpose of comparing trading strategies in a competition. The competition aspect makes *PXS* unusable for hedge funds and investment banks to evaluate their strategies. First, it is a competition only held at specific dates, hence participation in the competition is required and strategies can only be tested at specific dates. Second, the internal workings of the participating strategies are revealed for academic purposes, hence the competitive advantage is lost. Third, a specific API is used for the *PXS*, thus if a strategy is successful it has to be rewritten using the API of a trading application. With *CATSBF* strategies can be tested at any time and kept secret to maintain the competitive advantage. It is, furthermore, possible to integrate *CATSBF* with the company's trading application, thus allowing the strategy to be written in a familiar API.

The [Mat09] study bears a lot of resemblance to *CATSBF*. They developed their backtesting engine using an open protocol, FIX, thus allowing easy integration via FIX for

FIX enabled applications, and via a wrapper for other applications. The separation of the client and server, allows for easy deployment of strategies tested with the artificial market. But, there are some key differences as well. *CATSBF* backtests against historical data, and is released as open source, for everyone to use. Whereas the artificial market of [Mat09] is not released to the public, at least not to the author's knowledge, and it can only backtest against an artificial agent-based market rather than empirical high-frequency data, as *CATSBF* allows.

3 Marketcetera

Marketcetera is an *Automated Open Source Trading Platform*, with focus on, but not limited to, automated strategy driven trading [Mar11j].

According to *Marketcetera*'s internal estimations [Mar11k], there is currently approximately 550 production installations of *Marketcetera*, as opposed to 450 in 2010. There has been approximately 13,000 downloads of the product, which suggests that a lot of people are experimenting with the product, or using it for research. From these numbers it seems that *Marketcetera* is widely used in production, and increasing each year.

The fact that *Marketcetera* has taken the *open source* approach means that the user is provided with a free trading platform that he can either use as is or customise to meet his business requirements.

It is broker and data provider independent. The standard FIX protocol is used for order routing, which means that it easily integrates with brokers, or exchanges, that are FIX enabled. Custom market data adapters and broker connections can be implemented to integrate with user specific providers. Multi-destination order routing is supported, hence it is possible for a strategy to trade with multiple brokers/exchanges simultaneously.

The platform consists of a number of integrated components, each with different responsibilities. Figure 1 gives a brief overview of the modules and how they interrelate. The

modules are described in more detail in the following [Mar11a].[6]

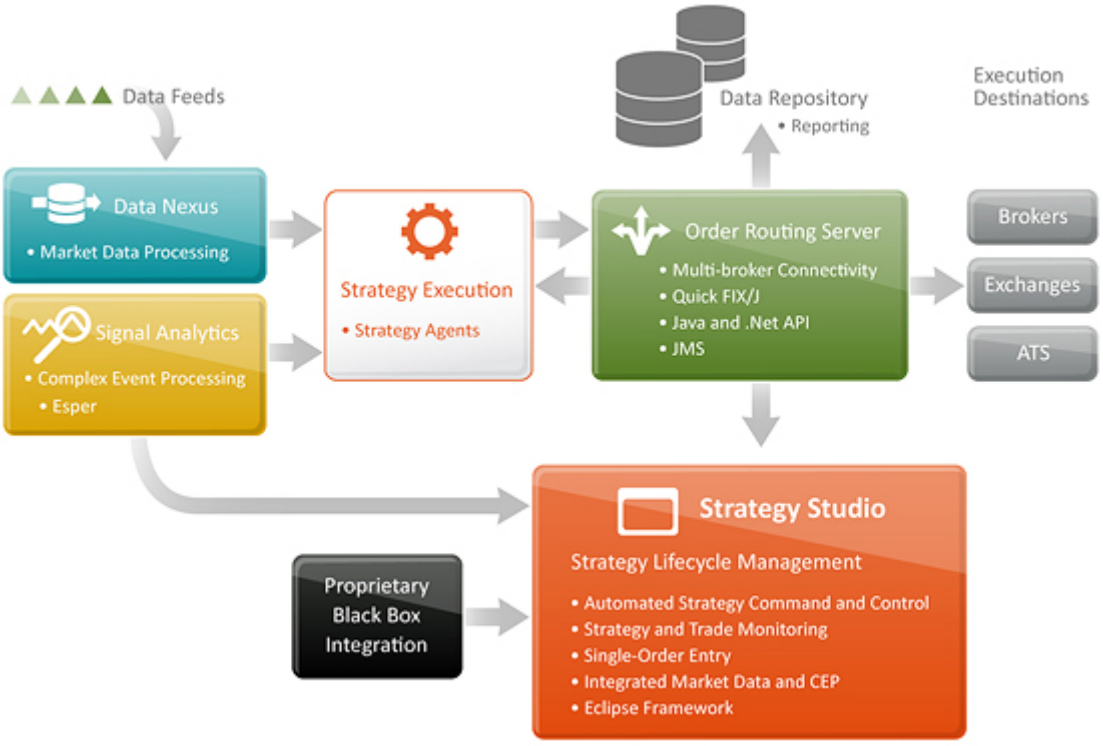

Figure 1: Overview of the Marketcetera modules and how they integrate. From the *Marketcetera* product page (used with permission): `http://www.marketcetera.com/site/products/marketcetera-platform`

3.1 Photon

Photon [Mar11d] is the platform's Graphical User Interface (GUI) based on the *Eclipse Rich Client Platform* [Ecl11], which makes it highly extensible.

Photon is a GUI application that supports manual order entry and order manipulation (cancel/replace), and comprehensive monitoring of; real-time market data quotes, FIX messages, output written to console and positions. Furthermore, it has an embedded web browser that can display additional security information.

[6] See [Mar11f] for a list of supported features.

3.2 Strategy Studio

Tools to manage the entire trading strategy life-cycle are featured in the *Strategy Studio* [Mar11e]. It is implemented as a view in *Photon*, and can be used to author, test and deploy strategies, and to monitor execution of both manual and automatic trades, in run-time.

A comprehensive set of APIs for authoring automated trading strategies is provided. Supported interfaces include support for event stream queries, hereunder complex event processing [Wik11b], e.g. for easy calculation of moving averages, and trading server connectivity to automatically place orders, monitor trades, positions and profit/loss.

Trading strategies can be developed in the popular programming languages Java [Jav11] or Ruby [Rub11] and for that *Photon* features an embedded Ruby scripting environment.

3.3 Strategy Execution Engine

The *Strategy Execution Engine* [Mar11e] is a complete standalone run-time environment responsible for efficient and secure automated execution of strategies. It supports multiple strategies running simultaneously, and allows the strategies to have interconnected data flows and to utilise other *Marketcetera* modules like the *Data Nexus* (see 3.4) and *Signal Analytics* (see 3.6). It runs as a service, and does not provide a GUI, thus consuming less resources than *Photon*.

3.4 Data Nexus

Enables connectivity to multiple market data streams, via market data adapters. The market data is transformed and disseminated to other platform modules with very low latency. Default data adapters are provided, but a consistent architecture allows users to easily integrate additional market data adapters.

3.5 Order Routing Server (ORS)

ORS [Mar11c] is responsible for routing orders, received from other modules, e.g. *Photon* or the *Strategy Execution Engine*, to the appropriate broker, or exchange, using the FIX protocol. When a reply is received it is forwarded to the module placing the order. It is furthermore the platforms gateway to the database, thus *Photon* obtains information about existing positions through the *ORS*.

3.6 Signal Analytics

Signal Analytics [Mar11b] provide tools for data analysis, such as *Complex Event Processing (CEP)* [Wik11b] of market data streams. CEP is implemented using the open source CEP library *Esper* [Esp11]. *Esper* processes events using a SQL query like language, and accepts any type of data. A strategy could, for example, implement an *Esper* query to calculate a moving average and react to it.

3.7 Evaluation of Marketcetera as an Automated Trading Platform

Marketcetera has no license fees, because it is *open source*. Companies often need to modify or extend their trading application to meet specific requirements. Sometimes they even develop entire systems in-house, to keep trading knowledge as secret as possible. Using *Marketcetera* as the foundation for a trading platform not only results in saving money on license fees, as opposed to buying a proprietary solution and extending it, but it also saves a lot of time on development, because *Marketcetera* provides a robust and thoroughly tested foundation that can be customised to meet specific user needs. Furthermore, it is implemented in Java, a well known open source programming language, suggesting that many firms already have the required skills to extend the platform, or at least would not have problems hiring people with such skills.

Marketcetera trading strategies are authored in existing well known programming languages, Java and Ruby, while other proprietary vendors use their own scripting languages. This means that one can risk spending a lot of time learning the vendor specific languages and APIs, and when a lot of time and money is invested in learning these APIs, one will loose this investment if the vendor is changed, thus people tend to stick with the same vendor.

Another crucial aspect to Marketcetera as an automated trading platform is it's scalability. The *Strategy Execution Engine* code module is called the *Strategy Agent*, and is a standalone environment for running automated strategies. Multiple instances of the *Strategy Agent* can run simultaneously and each agent can run multiple strategies at once [Mar11e]. This architecture causes it to scale well, hence it can easily handle a strategy (or a number of strategies) that control an entire portfolio (or several portfolios) of securities; simply add more agents as needed.

Even though the module scales well for multiple strategies and large portfolios, there is, at least, one drawback. Java uses automated garbage collection[7], causing the application not to guarantee real-time execution. Not being real-time will probably not affect the average user, but for high-frequency trading a real-time application is required, otherwise firms can potentially loose a lot of money.

In order to be a complete automated trading platform, Marketcetera needs a way for users to test their trading strategies thoroughly before deploying them, which is hopefully resolved, in time, with *CATSBF*.

4 Framework

CATSBF is implemented as part of this dissertation as a software project. It is intended as a basic framework for automated trading strategy historical backtesting. It is open

[7]Garbage collection is a form of automated memory management, that once in a while tries to free occupied memory that is no longer in use. It is impossible to tell when this is done and takes up the computer resources, which is a problem [Wik11c].

source and thus allows users to use it as is, or extend it to their specific needs. Because it is only a basic framework, a number of simplifications are made during the development (see section 4.3.10).

In order to make the backtest resemble a realistic scenario, *CATSBF* must simulate an exchange and provide a market data adapter for *Marketcetera*. Therefore, two separate modules are implemented to constitute *CATSBF*; the *CCFEA Market Data Adapter*, a *Marketcetera* market data adapter, and a simulated exchange, *CCFEA Exchange*. These two module are not directly connected, rather they communicate as client and server. The communication happens via FIX messages, just like the *Marketcetera* platform would do in a real trading scenario.

4.1 CCFEA Marketcetera Market Data Adapter

Marketcetera integrates with market data providers via vendor specific adapters. The *CCFEA Marketcetera Market Data Adapter* is implemented as such an adapter to send FIX market data requests to the *CCFEA Exchange*, and parse market data messages received from the exchange. Currently the only supported market data format is FIX `MarketDataIncrementalRefresh`[8].

4.2 CCFEA exchange

To manage the market simulation and the backtest itself the *CCFEA Exchange* was developed. The exchange is responsible for everything that has to do with simulating the market, including loading market data from external sources and replaying it in sequential order to reconstruct the order book, accepting market data requests and orders via FIX, sending market data updates, sending order execution reports and performing the actual backtest. This module is basically the heart of *CATSBF*.

It is split into several sub components. The first component is the *CCFEA Market Data*

[8]Words in `teletypefont` refers to code classes.

Loader which has the job of loading market data from external sources, such as CSV files or external databases, and insert it into the exchange's database using the database adapter. An SQLite [SQL11] database adapter is provided with *CATSBF*, *CCFEA SQLite Adapter*, which handles all integration with the integrated SQLite database, hereunder insertions and fetches of market data.

Two subcomponents are responsible for the communication with *Marketcetera*. These are two FIX servers, one that accepts market data requests and pushes market data to *Marketcetera*, *CCFEA Market Data Server*, and one that accepts orders from the strategy running in *Marketcetera*, *CCFEA Order Server*, and checks them for execution. The two modules communicate with *Marketcetera* via FIX messages. The servers are implemented as two separate components because they are logically separate entities due to their different responsibilities. They each communicate with different *Marketcetera* modules. The *CCFEA Order Server* communicates with the *Marketcetera Order Routing Server* and the *CCFEA Market Data Server* communicates with the *CCFEA Marketcetera Market Data Adapter*. Figure 2 gives an overview of the modules and how they relate to each other.

4.3 Software Design

Several software design patterns [Eri, pp. 12-14] are used to make *CATSBF* extensible and, some parts, easily exchangeable with custom implementations. Furthermore, I used the *Spring* [Spr11] framework to configure all dependencies, thus enforcing *Inversion of Control* and *Dependency Injection* [Fow04].

Parts of *CATSBF* are meant to be customised to specific user needs, and the use of *Spring* and design patterns allows custom implementations to be injected as a matter of changing the configuration.

This section describes how design patterns are utilised along with important implementation aspects. Please consult figure 3 for a UML class diagram of the *CCFEA Exchange*.

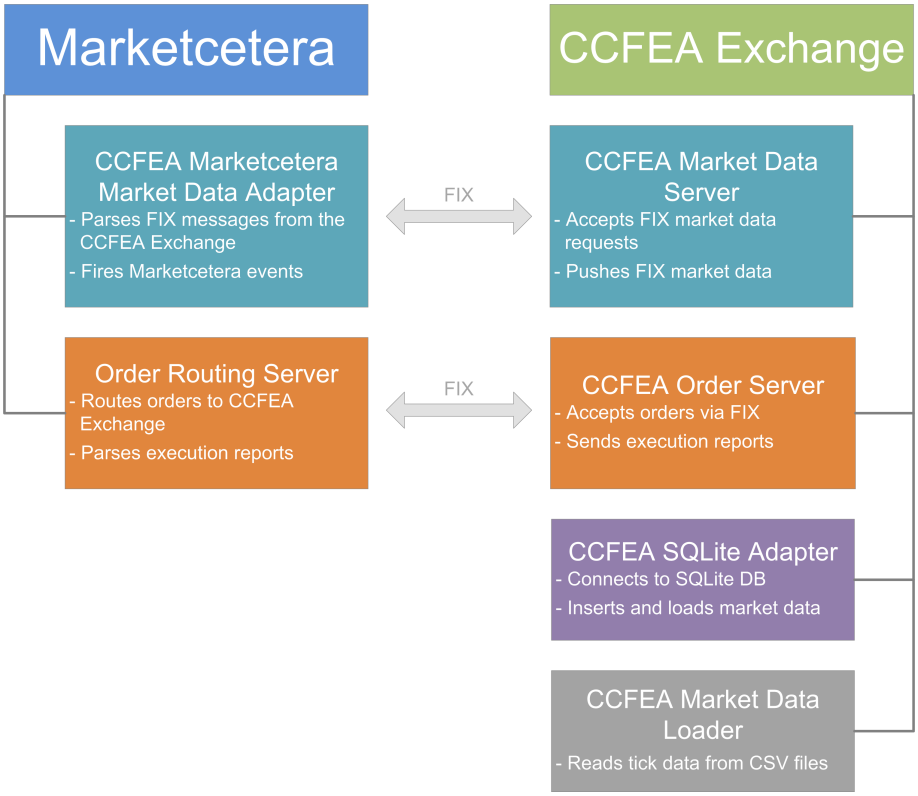

Figure 2: Overview of the *CCFEA Exchange* and how the different modules interrelate and communicate with *Marketcetera*.

4.3.1 Order Execution Strategy

How orders are executed, and what impact they have on the order book, depends on the *Order Execution Strategy*. Custom execution strategies can be implemented, due to the use of the *Strategy* [Eri, pp. 315-324] design pattern. They simply have to comply with the `IOrderExecutionStrategy` interface, which has two methods, one to check a *Bid* for execution and one to check an *Offer* for execution. *CATSBF* comes with a default strategy implementation, `CcfeaOrderExecutionStrategy` (explained in detail in section 4.3.11).

4.3.2 Market Data Loader

Most users will probably need custom *Market Data Loaders*. The *Adapter* [Eri, pp. 139-150] pattern provides a way of creating a common interface for market data loaders, `IMarketDataLoader`, and thus making it possible to implement custom data loaders. A

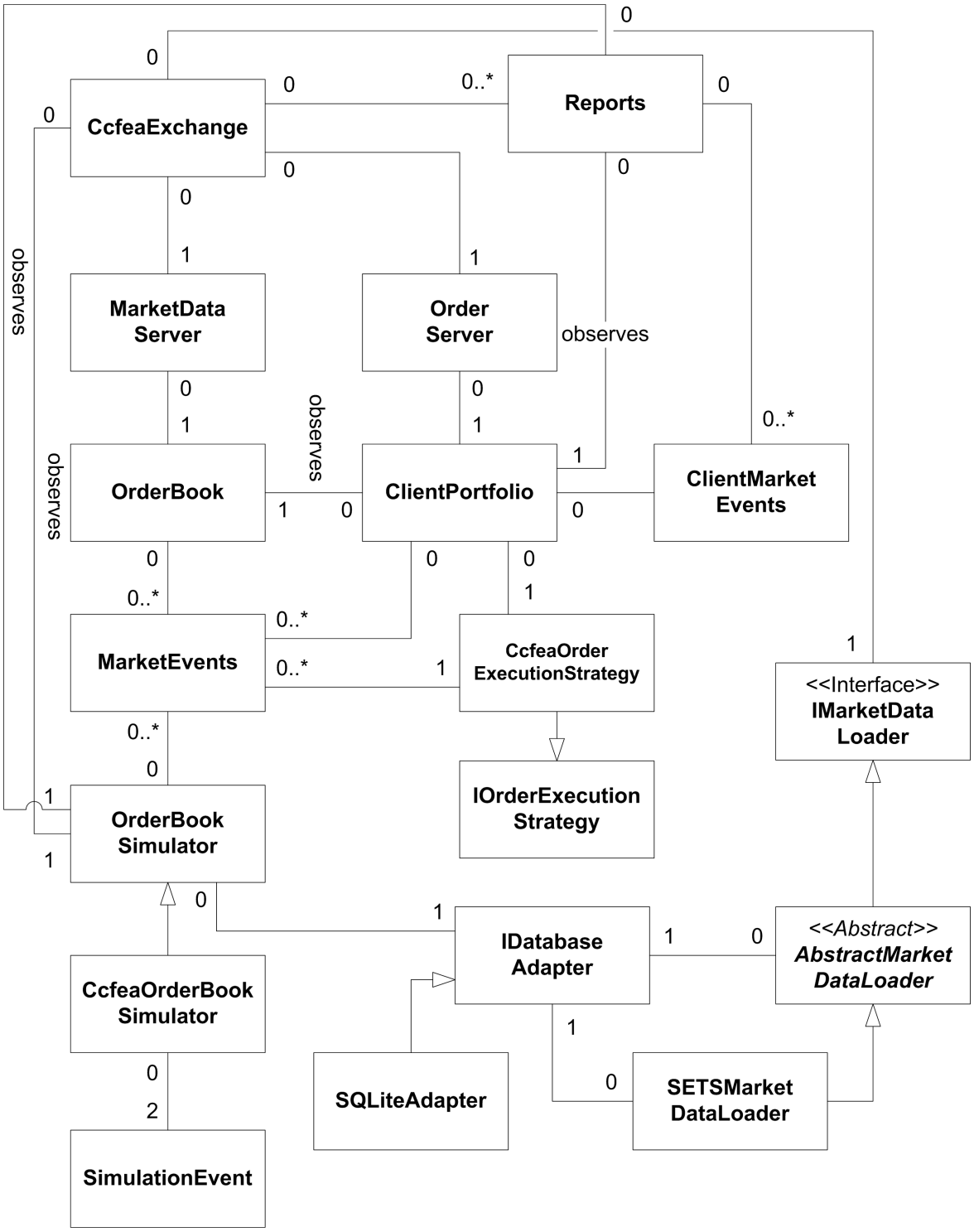

Figure 3: Simplified class diagram of the *CCFEA Exchange* including the most important classes and their relations.

custom market data loader must implement the interface, or extend the `AbstractMarket-DataLoader`. The `AbstractMarketDataLoader` pre-implements the database integration used to create the necessary database tables and functionality to insert data, hence the

custom data loader can focus on reading the actual data. If more control is needed, the `IMarketDataLoader` interface should be implemented. *CATSBF* comes with a market data loader implementation that loads CSV market data from the London Stock Exchange (SETS), `SETSMarketDataLoader`.

4.3.3 Database Adapter

CATSBF currently uses SQLite [SQL11] as the database engine to hold market data due to its simplicity. SQLite is a self-contained, serverless, zero-configuration transactional SQL database engine, and it integrates well with Java (using JDBC [Ora11]). The *Adapter* pattern is employed for the database integration. Hence, if one prefers to use another database engine, this is achievable by implementing a custom database adapter that conforms to the `IDatabaseAdapter` interface. In most cases the existing database adapter will be sufficient.

4.3.4 Reporting

For reporting the *Observer* [Eri, pp. 293-303] pattern is used in conjunction with *Polymorphism* [Eri, pp. 14]. An abstract `Report` class defines the basic reporting functionality, such as which methods to call on specific events, and methods to get the file path and file extension.

An arbitrary number of reports can be added to the simulation. Custom reports must extend the abstract `Report` class. The reports are set to observe the client portfolio, but can, manually, be added as observers of the `OrderBook` as well. Two default reports are implemented in *CATSBF*[9]:

- `CcfeaTradeLogReport`: Logs all actions performed by the client. Logs new orders, partially filled orders and filled orders and saves it to a text file.

[9] *CATSBF* actually comes with 3 reports, but the third is implemented specifically for the sample trading strategy, see more in section 6.

- `CcfeaSumCalcReport`: Calculates summarised data such as, total price and amount of sold and bought assets, average buy and sell price and total profit/losses.

More advanced reports, or trading strategy specific reports are easily added to the back-test.

4.3.5 Order Book

A cached order book, `OrderBook`, is kept up to date at all times during the simulation. For each market event in the historical data, and each order placed or deleted by the client, a change is made to the order book. It is implemented as an `Observable` *Singleton* [Eri, pp. 127-134] because numerous objects needs to interact with it. Making it a *Singleton* allows easy access to the same object, plus we can easily make it thread safe (see section 4.3.10). All changes to the order book needs to be reported to *Marketcetera*. A `MarketDataReporter` is implemented as observer of the order book and notifies the client of all changes via FIX messages.

Separate sorted lists for offers and bids are kept in the order book. The lists are sorted with the most competitive limit orders at the top, to less competitive limit orders at the bottom. That is, high buy prices and low sell prices are on top. This makes it possible for the *Order Execution Strategy* to easily find matches without searching the entire list.

4.3.6 Client Portfolio

All orders placed by the client, *Marketcetera*, are recorded in a `ClientPortfolio`. It keeps track of all currently open orders, partial fills and complete fills belonging to the client. When a partial, or complete fill, is added to the portfolio, it automatically updates, or deletes, the existing open order, if any match exist.

It is implemented as an `Observable`, and notifies observers on any changes made to it. An `OrderReporter` is implemented to observe the portfolio and send execution reports (FIX *Execution Reports*) to *Marketcetera* on all updates.

4.3.7 Simulation Time

During a backtest the market events are replayed in sequential order. The replayed events has timestamps that correspond to the actual time they were executed, according to the historical data. The orders placed by the client in the market gets a timestamp as well. In *CATSBF* a *Simulation Time* is introduced, which is continuously updated according to the time in the replayed market events. The user can choose to give the client orders timestamps that matches the simulation time or the actual time. This is configurable in the *Spring* configuration.

4.3.8 Order Book Simulator

An `OrderBookSimulator` is used to replay the historical market events. When the simulation begins it fetches all market events from the database, within the configured time interval, and replays them in order. It updates the cached order book according to each event.

The implemented order book simulator, `CcfeaOrderBookSimulator`, is meant as a general order book simulator that can be used for any external data source, but it is exchangeable with custom simulators. Custom simulators have to extend the abstract class `OrderBookSimulator`, which implements `Runnable` (see 4.3.10) and extends `Observable`. Simulators should notify observers when the simulation starts and stops.

4.3.9 Integration

All integration between the *CCFEA Exchange* and *Marketcetera* is managed via FIX messages. All FIX messaging is implemented using the open source *QuickFIX/J* [Qui11] FIX engine. The use of FIX potentially allows any FIX enabled application to integrate with, and use, *CATSBF* for backtesting. If an application does not use FIX, it would be possible to implement a wrapper that converts the application specific messages to FIX messages, and the other way around.

4.3.10 Multithreading

When the *CCFEA Exchange* is started it runs a single thread that loads the market data, and initiates the FIX servers. Each FIX server has its own thread to manage dequeueing of received messages. The *QuickFIX/J* engine manages all FIX connections and the actual sending and receiving of messages. The FIX server threads are running and managed within this engine as well. Received FIX messages are handled in the main thread one by one, forcing the order book simulation to be run in a separate thread, hence simulators has to implement `Runnable`, because it otherwise would block the main thread.

4.3.11 CCFEA Order Execution Strategy

The default order execution strategy is called `CcfeaOrderExecutionStrategy`, and this section explains it in detail.

When an order is received from the client, this can be a *buy (bid)* or *sell (offer)* order, it is matched against orders in the cached order book.[10]

On a bid order, a complete match (fill) is when the order book contains one or more opposing offers that can fill the bid. This happens when a number of offers have a cumulative size larger than the size of the bid, and the offer prices all are less than or equal to the bid price.

It is a partial match (partial fill) if there exist one or more opposing offers where the price is less than or equal to the bid price and the cumulative size is less than the bid size.

The same goes for a received offer except that the matching bid price has to be greater than or equal to the offer price.

Orders that are not matched are placed as open orders. Every time a new order is replayed from the historical data, the open (including partially filled) orders are tested as a match against the historical order, and executed if possible.

[10] See [Gen01, pp. 75-78] for definitions of market order, limit order, fill, bid and offer.

4.3.11.1 Limit Orders

A limit order is when a maximum price is given for a bid, or a minimum price for an offer. The strategy executes the bid at the best price possible, hence the execution price will always be less than or equal to the bid price.

GOOG			
BUY Orders		**SELL Orders**	
Qty	Price ($)	Price ($)	Qty
100	605	606	250
250	604	607	50
150	603	608	550

Figure 4: Example of a possible order book for the Google stock (GOOG). The data is fictional.

The strategy for a client offer is much alike, except that the offer is matched at a price equal to the offer price, even though the matching bid has a larger limit price.

The execution price is calculated as the average price matched at the opposing orders. Figure 5 provides examples that explain the limit order execution strategy in more detail.

4.3.11.2 Market Orders

Market orders are orders to be executed at the best possible price in the market. They are matched with the most competitive limit orders on the opposing order book. Market orders are always executed, assuming that there exist opposing orders with a cumulative size larger than, or equal, to the market order.

If an order cannot be filled, this goes for both market and limit orders, it is either partially filled or placed as a new order in the order book, and will be filled or partially filled as soon as new matching orders arrive.

4.3.11.3 Client Orders Impact on the Order Book

When a client order is matched against one, or more, historical orders, only the client order is updated. This means that the historical orders are not removed from the order book, even though the are partially or completely filled, nor are their order size values

Complete Fill Limit Orders	
BID	**OFFER**
The client places a *bid (Qty: 300, Price: 607.5)*. The *bid* is matched against the first *offer (Qty: 250, Price: 606)*, and the second *offer (Qty: 50, Price: 607)* in the order book. Hence the *bid* is fully matched with a price of $(250 * 606 + 50 * 607)/(250 + 50) = 606.167$, and size 300.	The client places an *offer (Qty: 200, Price: 604)*. The *offer* is matched against the first *bid (Qty: 100, Price: 605)*, and the second *bid (Qty: 250, Price: 604)* in the order book. Hence the *offer* is fully matched with a price of (100∗604+100∗604)/(100+100) = 604, and size 200.

(a) Complete Fill Limit Orders Examples

Partial Fill Limit Orders	
BID	**OFFER**
The client places a *bid (Qty: 350, Price: 606)*. The *bid* is only matched against the first *offer (Qty: 250, Price: 606)*, because the price of the second offer is too big. The bid is partially filled with a price of $250 * 606/250 = 606$ and size 250. This leaves an open *bid (Qty: 100, Price: 606)*.	The client places an *offer (Qty: 150, Price: 605)*. The *offer* is only matched against the first *bid (Qty: 100, Price: 605)*, because the price of the second bid is too low. The offer is partially filled with a price of $(100 * 605)/100 = 605$ and size 100. This leaves an open *offer (Qty: 50, Price: 605)*.

(b) Partial Fill Limit Orders Examples

Figure 5: Examples explaining the *CCFEA Order Exection Strategy* for limit orders. The examples are based on the Google order book in figure 4.

updated. This is due to the potential later match against another historical event, and at this point will the historical order be removed, or updated, because that is what happened in the real market. This might not be the best approach, but the best execution strategy depends on the strategy under test. Thus, custom order execution strategies can be implemented, that corresponds to the users requirements.

There is one obvious problem with this approach. Consider the following scenario: The order book is as in figure 4. The client places a *bid (Qty: 300, Price: 607)*, which is filled at $price = 606.17$. Immediately after an identical bid is placed by the client, which is also filled against the same orders in the order book. This would not have happened in the real market, because the two top offers would have been removed from the order book, an the second bid would not be filled.

It is guaranteed that a client order is not checked as a match against the same historical order more than once. Meaning that a client order, *CO*, that is partially matched against the historical order *HO*, is never tested as a match against *HO* again. Neither is it

possible for a client order to be matched against his own orders. E.g. if the client places the *bid(Qty: 300, Price 610* and the *offer(Qty: 100, Price 609)*, they will not be matched against each other, thus making it possible for the client to have outstanding orders on both sides simultaneously.

4.4 Running a Backtest

In the following the flow of a backtest is described along with some implementation details. Figure 6 visualises the flow.

1. When starting the *CCFEA Exchange* the data loader is executed to populate the database. Whether or not to run the data loader is configurable, so if multiple tests are run against the same data, maybe in different time intervals, one can disable the data loader.

2. Then the *CCFEA Order Server* and *CCFEA Market Data Server* are started, and awaiting requests.

3. The actual market data simulation does not start until a market data request is received. When a request is received, the symbol is checked to match the symbol specified in the configuration.

4. When a correct request is received, the market events, from the configured time interval, is fetched from the database and replayed in order and the order book is continuously updated.

5. Each update to the order book is reported to *Marketcetera*.

6. Orders received from the client are matched against the orders in the order book, and the `ClientPortfolio` is updated accordingly.

7. On changes to the `ClientPortfolio` execution reports are send to *Marketcetera*, and the backtesting reports are notified.

8. When all historical market events are executed the server stops and backtesting

reports are closed.

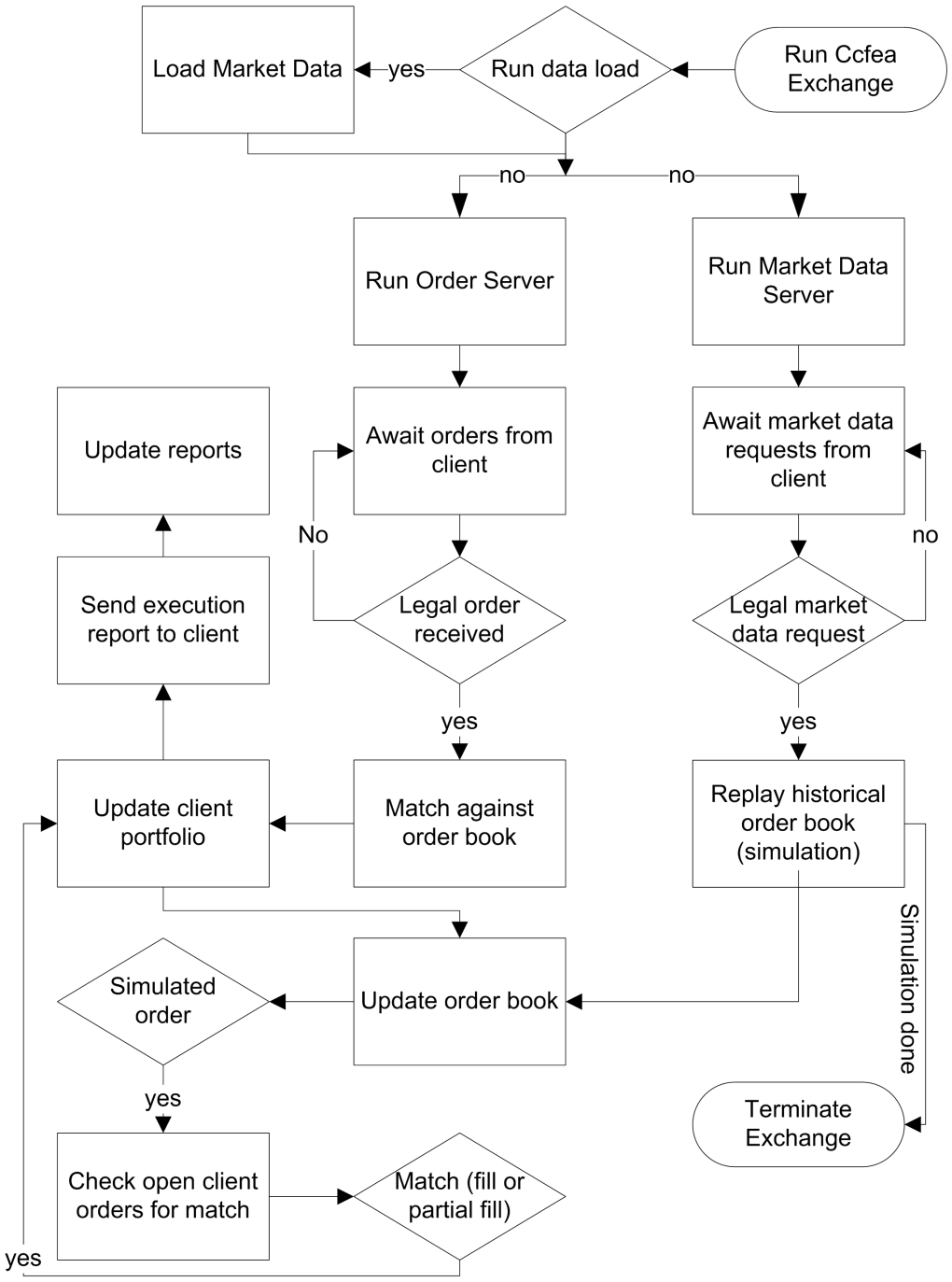

Figure 6: Flow chart visualising the execution flow of the exchange.

4.5 Simplifications and Limitations

Due to the fact that *CATSBF* is meant as a basic, extensible, backtesting framework a number of assumptions, simplifications and limitations have been made. These are

explained in the following.

4.5.1 One Client

The goal of this framework is to provide backtesting functionality to evaluate how well trading strategies would have performed in the past, so the server can only connect to one client at a time. One cannot have multiple clients trading against the same simulation simultaneously.

4.5.2 Securities

Only one stock (one symbol) can be traded at a time. The symbol traded needs to be specified in the configuration file. Only market events belonging to this security should be added to the database, as the framework assumes that all events in the database belongs to the same security. All requests from the client with a different symbol, than the one configured, are rejected.

Ordinary stocks is the only supported security type in *CATSBF*, even though *Marketcetera* supports trading in options as well. All orders are assumed to be on stocks, there is no verification as to whether this is actually true.

4.5.3 Time-In-Force Options

Time-in-force [Inv11] options are limited to *Good Till Cancel (GTC)*. All orders are matched against the order book at arrival. If they are not filled immediately they are kept in the client portfolio (and the order book) as open orders, and tested against all new orders, until they are cancelled by the client, or the simulation terminates. Orders that specify another *Time-in-force* option are rejected.

4.5.4 Order Book Updates

Usually the current state of the order book is send to the client when a market data request is received, whereafter only updates to this state are sent. *CATSBF* is limited to only sending updates, hence no initial state is calculated and reported to the client.

4.5.5 Accepted Orders

Marketcetera only support single order entry entries, and therefore *CATSBF* is limited to only support single orders as well.

4.5.6 Order Status Updates

Execution reports are send to the client when an order is created (new - unfilled order), filled, partially filled, cancelled or rejected. FIX also supports sending *Done for day* reports indicating that no more trading on the stock takes place that day, which might be useful for the client, but this is not supported in *CATSBF*.

4.5.7 Historical Market Orders

It is assumed that there is no open market orders in the historical data. Thus, market orders recorded in the historical data is expected to be filled immediately, and therefore having a price attached. The `CcfeaOrderExecutionStrategy` cannot handle orders with a $price = 0$ correctly. If an unfilled market order is placed in the offer order book with ($price = 0$) it will be matched against any bid order placed by the client at $price = 0$, where it actually should be the current market price.

4.5.8 Short Selling

Short selling is allowed, *CATSBF* does not distinguish between a *sell* and *short sell* order. *CATSBF* has no knowledge about the clients funds, or positions (except those

made during the running simulation) therefore it is possible for the client to sell (short sell) an asset that is not owned, and buy an asset even though he does not have the funds. These matters are assumed to be handled on the client, as they would in a real trading scenario. Note that *Marketcetera* keeps a client portfolio that can be queried in the trading strategy.

4.5.9 Optimisation

Optimisation has not been a priority, because it is a backtesting framework, so there are no requirements for Real-time execution, or other execution time aspects. Especially the *SETS Market Data Loader* is not implemented in the most efficient way because it extends the `AbstractMarketDataLoader` and has to do a lot of data preprocessing before inserting the market data, using the default insertion methods, instead of directly updating the database.

The lack of optimisation will only become a problem for extremely large datasets. Investigation and improvement of performance issues are left to the user.

4.6 Market Impact

How the trading strategy impacts the market and the market price during a simulation is not taken into account in any way. In a highly liquid market the market impact of the trading strategy would be very little and it is therefore ignored, and perfect liquidity is assumed.

Due to the perfect liquidity assumption does *CATSBF* not remove matched orders from the order book, when an order is matched, using the *CCFEA Order Execution Strategy*, nor does it consider the market price changes that the trading strategy might cause.[11] Due to the extensibility of *CATSBF* it is possible to implement a custom *Order Execution Strategy* that simulates market impact.

[11] See 4.3.11 for a detailed description of the *CCFEA Order Execution Strategy*

One way to achieve this could be to remove matched orders from the order book, and keep a price delta, in memory, that describe cumulative price changes, and apply the delta to the price henceforth [PV08]. Another approach could be to completely ignore historical information about matches, and do the matching in the simulation. This way historical data would be mixed with the strategy's orders and whether or not to fill historical orders is dependant on the orders placed by the strategy, thus letting it impact the market. In any case it has to be validated that the simulated market still resembles a real market, hence exhibits stylised facts [Con01].

5 Future Work

By releasing *CATSBF* as open source it is the author's hope that the framework will continuously be evolved and improved upon. In this section suggestions for future work is described.

5.1 Feature Improvements

In this section a number of feature extensions to *CATSBF* are suggested, along with brief discussions on how to implemented them.

Market Data Loaders for other market data providers, this could be other exchanges or Bloomberg [Blo11], would make *CATSBF* attractive to a larger group of users. Enabling *CATSBF* to use a live market data feed could also be interesting, and would probably be doable by implementing a custom *Order Book Simulator*.

To make the simulation resemble a real-life situation even more *CATSBF* should be extended to calculate the initial state of the order book and push it to *Marketcetera* before sending updates. This will require changes to both the *CCFEA Exchange* and the *CCFEA Marketcetera Market Data Adapter*. The adapter should be extended to support FIX `MarketDataFullSnapshotRefresh` messages, but this should be a simple

improvement as it is parsed in a similar way to the currently accepted `MarketData-IncrementalRefresh` messages. The update to the exchange on the other hand imposes a new problem. It is not possible to check whether or not all the information to calculate the initial order book state is available. It could either be assumed that the database contains all market events needed, or a strategy to handle missing data could be implemented.

Adding support for more *Time-in-force* options, would definitely make the backtest more realistic. Strategies will, in most cases, be backtested against data that spans several days, maybe even several years, therefore it would be more realistic if the strategy could specify other *time-in-force* options, such as *DAY*, hence specifying that the order expires at the end of the day. *CATSBF*'s internal clock, *Simulation Time*, could be observed by the client portfolio and check open orders for expiry at specific times. Using this internal clock it would be possible to implement "Done-for-day" execution reports as well. These would notify the client that a day has passed and that no further executions are forthcoming that trading day.

Any market should be able to handle *market orders*, thus the *CCFEA Order Execution Strategy* should be updated to handle *market orders* correctly. It would probably be a good idea to update `MarketEvent`s to include a field specifying whether they are a *market* or *limit order*. That would generally give more information about the orders within *CATSBF*, and could potentially provide more information to the client. This would include adding this information in the database on market data load.

Implementing new backtesting reports that calculate more advanced statistics would contribute greatly to the analysis capabilities of *CATSBF*. It might be possible to create a report that automatically looks for stylised facts in the trading data, and thus validate (or invalidate) the results.

5.2 Framework Validation

If the framework is to be used in production, one should somehow validate that it resembles an actual market. A few validation methods are discussed in this section.

5.2.1 Replicate Another Study

One way to verify the framework would be to find another backtesting study and replicate it. That would include implementing a similar strategy, to the one used in the existing study, in Marketcetera and then backtest it using *CATSBF*. If the results from the two studies are equal, with some margin, one could be assured that the framework works correct. Assuming that the replicated study is correct.

5.2.2 Compare Order Books

CATSBF rebuilds the order book from the historical data. If one used another tool to rebuild the same order book, and compared the two order books, at a given time, the order books should be identical. Some assumptions has to be made for this approach. First, the used tool should be guaranteed to generate the correct order book. Second, *CATSBF* starts populating the order book from the given start time, it does not take into account orders that might be in the order book already, hence the comparison has to be at a point so far from the start date that the actual state of the order book is expected to be in cache.

Another downside to this approach is that only the order book reconstruction is verified, it does not guarantee that the simulation will exhibit real market characteristics when a strategy is trading during the simulation.

5.2.3 Stylised Facts

Stylised facts are empirical statistical properties common to asset returns which are observable in real markets [Con01]. If stylized facts can be found in the high frequency data

from the simulation, *CATSBF* is likely to resemble a real market.

Assuming that the simulation of historical market events is correct, stylized facts should always be exhibited when only the data is replayed, because this should be exactly as the orders were placed in the real market. So, to be sure that the simulation is correct when running trading strategy during the simulation, one should look for stylised facts in data collected from a simulation where a strategy is trading. Such simulation generates high frequency data that is a combination of the historical events and the events generated by the strategy. The result of such a study will greatly depend on the order execution strategy, as that defines how the strategy impacts the market. The strategy can potentially corrupt the simulation, thus keeping the simulation from exhibiting stylized facts.

6 An Application of *CATSBF*

I implemented a simple trading strategy in *Marketcetera*, along with a specific report, to demonstrate an application of *CATSBF*. The strategy, *MACrossingStrategy* [Sch11a], places buy and sell orders in turn based on crossings of a long term moving average, *lma*, and a short term moving average, *sma*. A buy order is placed when *sma* crosses *lma* from below (indicating a positive trend) followed by a sell order when *sma* crosses *lma* from above (indicating a negative trend). All orders are market orders of exactly one stock, thus execution is, almost, guaranteed immediately. *MACrossingStrategy* only trades one asset at a time, and thus every buy/sell pair can be seen as a trade making a profit (or loss), hence the strategy makes a series of individual returns.

The report [Sch11b] implemented to evaluate the *MACrossingStrategy* calculates descriptive statistics on the returns for the series of buy/sell trades, including maximum, minimum, mean and standard deviation. Additionally it calculates the maximum drawdown [Cha09, pp. 21-22] and its duration. Figure 7 provides an overview of the results.

A backtest of the *MACrossingStrategy*, with $sma = 25$ and $lma = 30$, was performed against SETS March 2007 market data for the *GlaxoSmithKline* stock. It earned a total

Test results	
Indicator	**Result**
Maximum return	16.70%
Minimum return	−3.169%
Standard deviation	1.222%
Total profit/loss	£3841.0
Maximum drawdown	67.33%
Maximum drawdown duration	2050
Total number of buy/sell trades	4557

Figure 7: Test results for the *MACrossingStrategy* with *sma* = 25 and *lma* = 30 against *SETS* March 2007 *GlaxoSmithKline* data.

profit of £4651.0, implying that it is a profitable strategy, but when analysing the test results, it becomes clear that it does not provide a consistent return. The mean return was as little as 0.08743%, with values spanning from −3.169% to 16.70%, indicating that there are some outliars.

The standard deviation is fairly low, 1.222%, suggesting that the returns tend to be close to the mean, thus indicating that the strategy generally produces low returns, in the area of $0.08743\% \pm 1.222\%$.

The maximum drawdown is 67.33%, and has a duration of 2050 buy/sell trades of the total 4557 trades. This is a very long drawdown period, suggesting that big profits are made in other periods. When analysing the graph of cumulative returns, see figure 8, it is clear that the strategy makes its profit in spikes, followed by longer drawdown periods, thus it appears that only a fraction of the profits are positive.

Backtests with other values for *sma* and *lma* yielded similar results, hence the strategy seems to consistently provide positive returns, for that specific month, but the mean returns are very low, and the total positive profits are generated from only a fraction of the trades, hence making it a risky strategy. Based on the results from the backtest, it is clear that the strategy relies on making a few big profits, thus making it a poor strategy.

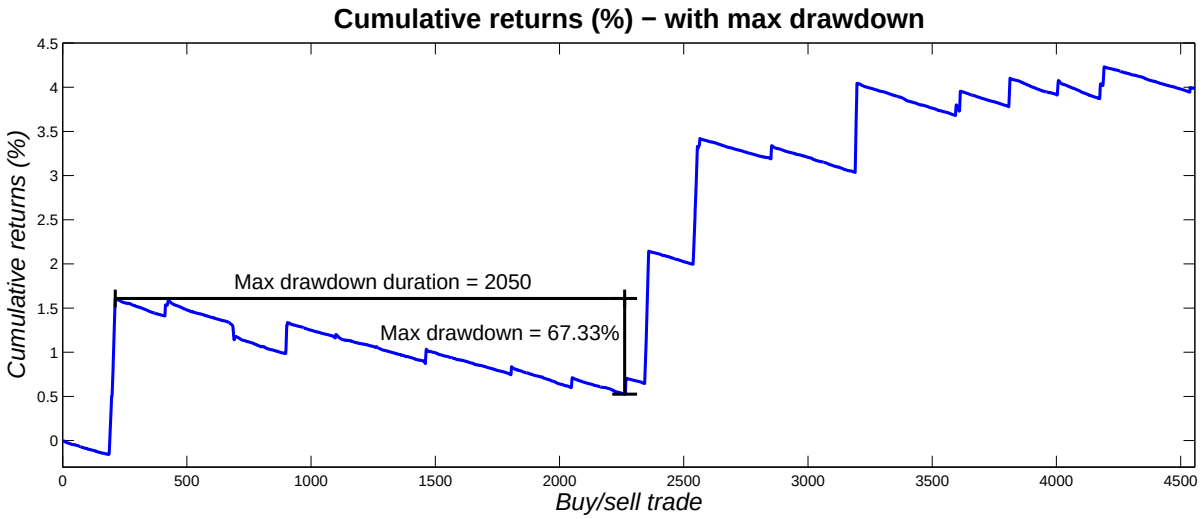

Figure 8: Cumulative returns for *MACrossingStrategy*, with the maximum drawdown indicated. Based on data generated by the backtest report.

7 Conclusion

The objective of this dissertation was to implement a basic, extensible, backtesting framework for the *Marketcetera* platform, and release it as open source. I constructed a backtesting framework, *CATSBF*, that uses the standard FIX communication protocol, which, potentially, allows any FIX enabled application to integrate with *CATSBF*. For *Marketcetera* I constructed a market data adapter, that parses FIX market data received from *CATSBF*. *CATSBF* is constructed in such a way that it resembles a realistic trading scenario, meaning that nothing "test" specific needs to be defined in the trading strategy, thus successful strategies can be deployed as is.

This work is a first step towards a rigorous backtesting framework, that can be utilised by any FIX enabled platform. *CATSBF* will, ultimately, be continuously evolved by a number of developers in the future. Already within the first few days after releasing *CATSBF* as open source, and posting it on the *Marketcetera* forum, a lot of interest was shown from both *Marketcetera* and users. A user has also requested to join the project as a contributor.

For the practical application of this framework some validation must be performed by the user. I have suggested different methods that could be applied to achieve this, hereunder

analysing the trading data and look for stylised facts.

8 Acknowledgements

I would like to show my gratitude to my supervisor, Dr. Steve Phelps, for his invaluable support and guidance throughout the dissertation process.

Thanks also goes to the users from the *Marketcetera* community who has shown interest in the project. It has been very encouraging and motivational. Special thanks goes to the user Colin for his help on various *Marketcetera* related programming issues.

9 References

[Alp11] Alphacet. Alphacet discovery. `http://www.alphacet.com`, 23-08-2011.

[Bat07] Dr John Bates. Algorithm backtesting. *The Technical Analys*, pages 34–36, July/August 2007.

[Blo11] Bloomberg. Bloomberg - business & financial news. `http://www.bloomberg.com/`, 24-08-2011.

[Cha09] Ernest P. Chan. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business*. Wiley, 2009.

[Con01] Rama Cont. Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance Volume 1*, (223-236), 2001.

[Ecl11] Eclipse. Eclipse rich client platform. `http://www.eclipse.org/home/categories/rcp.php`, 15-08-2011.

[Eri] year = 1995 isbn = 0201633612 publisher = Addison-Wesley Erich Gamma, Richard Helm, Ralph E. Johnson and John Vlissides, title = Design Patterns: Elements of Reusable Object-Oriented Software.

[Esp11] EsperTech. Espertech - Event Stream Intelligence. http://esper.codehaus.org/, 16-08-2011.

[Exc11] London Stock Exchange. Sets, London Stock Exchange. http://www.londonstockexchange.com/products-and-services/trading-services/sets/sets.htm, 24-08-2011.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004.

[Gen01] George A. Fontanills, Tom Gentile. *The Stock Market Course*. Wiley, 2001. 1st Edition.

[GNU11] GNU. The gnu general public license 3.0. http://www.gnu.org/copyleft/gpl.html, 20-08-2011.

[Inv11] Investopedia. Time in force definition. http://www.investopedia.com/terms/t/timeinforce.asp, 24-08-2011.

[Jav11] Java. Java technology. http://www.java.com/en/about/, 23-08-2011.

[Kab11a] Kaburobo. Advanced robot trading platform. http://www.kaburobo.jp/ (in Japanese), 10-08-2011.

[Kab11b] Kaburobo. Kaburobo development. http://www.kaburobo.jp/about/develop/ (in Japanese), 10-08-2011.

[Kab11c] Kaburobo. Trade science corporation. http://www.trade-sc.jp/english.html, 10-08-2011.

[KK95] Raymond Kan and George Kirikos. Biases in evaluating trading strategies. November 1995. University of Toronto and Leap of Faith Research Inc.

[KO03] Michael Kearns and Luis Ortiz. The Penn-Lehman Automated Trading Project. *IEEE Computer Society*, 2003. University of Pennsylvania.

[Mar11a] Marketcetera. Marketcetera Automated Trading Platform, Overview. http://www.marketcetera.com/site/products/marketcetera-platform, 08-08-2011.

[Mar11b] Marketcetera. Marketcetera automated trading platform, Complex Event Processing. http://www.marketcetera.org/confluence/display/CEP210/Complex+Event+Processing, 12-08-2011.

[Mar11c] Marketcetera. Marketcetera automated trading platform, order routing server. http://www.marketcetera.org/confluence/display/ORS210/ORS+Guide, 12-08-2011.

[Mar11d] Marketcetera. Marketcetera automated trading platform, photon. http://www.marketcetera.org/confluence/display/PN210/Photon+Guide, 12-08-2011.

[Mar11e] Marketcetera. Marketcetera automated trading platform, strategy agent. http://www.marketcetera.org/confluence/display/SA210/Strategy+Agent, 12-08-2011.

[Mar11f] Marketcetera. Marketcetera automated trading platform, supported features. http://www.marketcetera.com/site/products/what%E2%80%99s-supported, 12-08-2011.

[Mar11g] Marketcetera. Marketcetera automated trading platform, contribute. http://www.marketcetera.org/confluence/display/MPIO/Contribute, 20-08-2011.

[Mar11h] Marketcetera. Marketcetera automated trading platform, csv market data adapter. http://www.marketcetera.org/confluence/display/MOL/CSV+Market+Data+Adapter, 20-08-2011.

[Mar11i] Marketcetera. Marketcetera automated trading platform, open labs. http://www.marketcetera.org/confluence/display/MOL/Marketcetera+Open+Labs, 22-08-2011.

[Mar11j] Marketcetera. Marketcetera automated trading platform. http://www.marketcetera.com/, 24-08-2011.

[Mar11k] Marketcetera. Marketcetera community, number of users. http://www.marketcetera.org/community/posts/list/359.page#2162, 24-08-2011.

[Mat09] K. Izumi, F. Toriumi, H. Matsui. Evaluation of automated-trading strategies using an artificial market. *Neurocomputing 72*, (34693476), 2009.

[MSC11] MSCI. Barra aegis. http://www.factset.com/products/im/portfoliosimulation, 23-08-2011.

[Mul11] MultiCharts. Multicharts raising the trading standard. http://www.multicharts.com/, 23-08-2011.

[Ora11] Oracle. Jdbc overview. http://www.oracle.com/technetwork/java/overview-141217.html, 24-08-2011.

[Org09] Internation Labour Organization. Impact of the financial crisis on finance sector workers. *International Labour Office Geneva*, 2009. Sectorial Activities Programme.

[Org11] The FIX Protocol Organization. Fix protocol industry-driven messaging standard. http://fixprotocol.org/, 23-08-2011.

[PV08] Srinivas Raghavendra, Daniel Paraschiv and Laurentiu Vasili. A framework for testing algorithmic trading strategies. *Department of Economics National University of Ireland, Galway*, (0139), December 2008. Working Paper.

[Qui11] Quickfixj.org. Quickfix/j - free, open source java fix engine. http://www.quickfixj.org/, 12-08-2011.

[RP11] Minh Khoa Nguyen, Neil Rayner and Steve Phelps. Inferring the state of a double-auction market from empirical high-frequency transaction data. *CCFEA, Computer Science Department, University of Essex*, 2011. CCFEA WP045-10.

[Rub11] Ruby. Ruby programming language. http://www.ruby-lang.org/en/, 24-08-2011.

[Sch11a] Daniel Schiermer. Moving average crossing strategy. http://sourceforge.net/p/catsbf/code/9/tree/trunk/ccfea-marketdata/Ccfea%20Sample%20Strategies/ma_crossing_strategy.rb, 26-08-2011.

[Sch11b] Daniel Schiermer. Moving average crossing strategy report. http://sourceforge.net/p/catsbf/code/9/tree/trunk/ccfea-exchange/src/ccfea/exchange/report/CcfeaSampleMAStrategyReport.java, 26-08-2011.

[Spr11] SpringSource. Springsource. http://www.springsource.org/, 24-08-2011.

[SQL11] SQLite. Sqlite. http://www.sqlite.org/, 28-08-2011.

[Sys11] Factset Research Systems. Alpha testing. http://www.factset.com/products/im/alphatesting, 23-08-2011.

[Wik11a] Wikipedia. Wikipedia, Application Programming Interface. http://en.wikipedia.org/wiki/Application_programming_interface, 11-08-2011.

[Wik11b] Wikipedia. Wikipedia, Complex Event Processing. http://en.wikipedia.org/wiki/Complex_event_processing, 16-08-2011.

[Wik11c] Wikipedia. Wikipedia, Garbage collection. http://en.wikipedia.org/wiki/Garbage_collection_(computer_science), 27-08-2011.

[Wik11d] Wikipedia. Wikipedia, Integrated Development Environment. http://en.wikipedia.org/wiki/Integrated_development_environment, 27-08-2011.