Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

StockHome - Analytical Framework

Underlying analysis framework for a financial analysis tool

Francesco Rigotti

supervised by
Prof. Dr. Michele Lanza

Abstract

Nowadays millions of people invest in the stock market. To make profitable investments, a complicated decision-making process is needed, therefore many investors hire investment consultants. These consultants use various software solutions to perform analyses and then interpret the results to give their customers the solution they believe to be the best.
We built a framework performing analyses on stock market data. The result of these analyses are investment solution proposals. We designed the system in such a way to be extensible with other analysis techniques. The framework includes a system that automatically fetches the data from internet.

Acknowledgments

I would like to thank all the people that supported me during these three years in all different ways, especially my family and friends. A special thanking goes to Michele Lanza, the supervisor of this project, for constantly helping me with my academical decisions and for teaching me those key concepts that helped me during the rest of my bachelor program.

Contents

Abstract **i**

Acknowledgments **ii**

1 Introduction **1**

1.1 Structure of the Document 2

2 Problem & Related Work **3**

3 Solution **5**

3.1 Information retrieval 7

3.2 Database maintainer 7

3.2.1 Scheduling 8

3.3 Database 10

3.3.1 Data classes 10

3.4 The analysis component 13

3.4.1 Calculation core 13

3.4.2 Filtering core 14

3.4.3 Optimization core 15

3.5 Virtual portfolio 16

3.6 Hibernate 17

3.7 Performance of the framework 18

4 StockHome behind the scenes **19**

4.1 Filtering example 19

4.2 Search for a stock 20

4.3 Optimization example 21

4.4 Virtual portfolio creation 22

4.5 Buy a stock 23

5 Conclusions **24**

5.1 Future Work 24

5.2 Applications of the framework 25

5.2.1 Gilad Geron's web interface 25

A Implementation **27**

A.1 Technologies used 27

A.1.1 Java 27

A.1.2 Ruby 27

A.1.3 MySQL 27
A.1.4 Hibernate 28
A.2 Technology choices 29
A.3 Framework structure in deep 30
A.3.1 Filtering 31
A.3.2 Optimization 31
A.3.3 Hibernate 32
A.3.4 Calculation Core 33
A.3.5 Fetchers 34

Chapter 1

Introduction

Nowadays around 15 thousand companies have shares in the stock market and millions of people invest in this market. Such massive markets generate an enormous quantity of data, making them more difficult to observe and understand. In fact a large number of investors, both private and firms, hire investment consultants.

At the same time this massive amount of generated data is however extremely useful for those who know how to interpret it : through statistical analysis techniques, various factors determining the performance of each stock can be discovered, ad-hoc laws can be inferred and investment strategies can be designed. In fact, investment consultants use some software solutions to perform such analyses and give their customers a promising investment proposal.

Stock markets are extremely complex and are very sensitive to changes in the global economy situation. Researchers keep inventing new techniques to interpret the aforementioned data. Many new techniques take inspiration from approaches used in other sciences like artificial intelligence and biomedicine.

The goal of the project was to develop a framework performing statistical analyses on stock market data and proposing various investment solutions. The framework includes also a component in charge of fetching stock market data from internet, collect it in a database and keep this database up-to-date.

Because of the continuous increase in the number of analysis methods, we designed a framework which is easily extensible with other analysis techniques.

In order to test the quality of the system, we went to the financial department in the economics faculty of USI and asked a researcher for his help. We had him use our framework for some days and compare it, in terms of precision and speed, to the software solutions he usually uses.

1.1 Structure of the Document

In Chapter 2 we give a detailed description of the problem we wanted to solve and of its context. We then describe our solution in detail from a conceptual point of view (Chapter 3). In Chapter 4 we explain how the system accomplishes its tasks by showing the internal activity with UML sequence diagrams. We draw our conclusions in Chapter 5.

Chapter 2

Problem & Related Work

Stock markets generate massive amounts of data, which gets outdated and overwritten very fast; some data becomes old in a matter of minutes. It is extremely important to collect, organize and manage this data, because, if correctly analyzed, it can reveal many trends and laws of the market, therefore leading to better investment strategies.

The importance of stock market data is so big that there are companies collecting and managing this information as a business. These companies maintain massive databases and ask money in change of access to the data. The peculiarity of these databases is that they are updated in real-time and are therefore very expensive.

stock_name	ask_price	bid_price	closing_price	date
BIOS	7.07	6.00	6.36	2008-04-22

Figure 2.1: Sample of data about the stock BIOS

The picture above shows a sample of data generated for the stock BIOS on 2008/04/22. Out of the three prices in the sample, only the closing price is used to analyze the stock, while ask and bid are used when trading the stock.

Stock exchange data is very useful if analyzed. One of the most common category of analysis is optimizations. Optimizations are used when we want to perform a replication, the procedure of reproducing the performance of an expensive stock using a set of cheaper stocks. To find out how many shares of each cheap stock we should buy in order to obtain the best replication, we use optimizations.

We said before that all analyses have to be run on massive data sets and, therefore, cannot be carried out by humans. Because of this, in the last decade, many applications performing such analysis have been developed.

If we take a look at the current set of tools on the market, we notice, first that many of these are like add-on components to Excel, second that Microsoft's tool itself is broadly used.

Although flexible, Excel is a program with a more generic purpose and formulas for the analysis need to be specified by hand. This is certainly an error prone procedure and is used in an environment where correctness is crucial. Moreover, because of the complexity of formulas, only users with a good experience in the field can use such solutions.

When formulas are built in the software, unlike in Excel, users cannot corrupt them by mistake and the software can be used also by people without an in-depth knowledge of statistics. Another benefit of this approach is that user interfaces become simpler, because less input has to be specified: formulas are no longer part of the input.

However, even such a tool, like all the others, still suffers from a great lack of automation on the input side. In all existing software solutions, users have to manually feed the application with the data they want to analyze. In Excel, for example, the data has to be copied into a spread sheet.
Given the amount of data and its mutability, users need to keep refetching data to avoid analyzing out-dated information. This continuous refetching is extremely expensive in terms of time and, for firms, of money.

During the market day, the price of a stock keeps changing; the price at the end of the market day is called closing price. This price is the most useful for people willing to analyze a stock. By keeping a record of the closing price of a stock for each day, we obtain the history of this stock. The longer is the history of the stocks we analyze, the more precise are the results of the analyses.
Therefore, in order to obtain results that are precise enough, we have to build the history of each and every stock we want to analyze. Building the history of a stock requires fetched data to be collected and organized.

Because of the reasons mentioned afore, users must deal with the process of data collection and storage; in other words they have to maintain some sort of a database by themselves.

From the problems we illustrated in this chapter, we produced a set of requirements that an application has to satisfy in order to be a possible solution.

- The application must completely automate the process of data fetching, organization and storage;
- Users must have no access to formulas
- The application must be easily extensible with new analysis techniques

Chapter 3

Solution

As a solution to the problems illustrated in Chapter 2, we built a framework performing statistical analyses on stock market data. The result of these analyses is an investment solution proposal. We designed the framework in such a way to be extensible with other analysis techniques. The framework includes a system automatically downloading, managing and updating stock market data.

In order to illustrate the structure of the system we divide it in components, grouping in one component all entities that accomplish a determined task by collaborating with each other. On the next page we present a scheme to give an idea of the framework's structure and to show how components interact with each other. After a brief description of the scheme, we proceed by explaining in detail each component.

Figure 3.1, on the next page, shows a conceptual overview of the system. We describe this picture by illustrating the dataflow in the framework.

We start from the top with the *database maintainer* that tells the *information retrieval* component to download stock market data from internet and store it in the database. The same core now instructs the *calculation core* to calculate intermediate values used to speed-up the analysis procedures. The *calculation core* reads the necessary data from the database, performs the calculations and writes the results in the database.

Now the database contains all data necessary for the analyses: the *filtering* and *optimization* cores are ready to offer their functionality when requested by the user through the interface. Together with the analysis component also the *virtual portfolios* component is ready to satisfy the requests of the user.

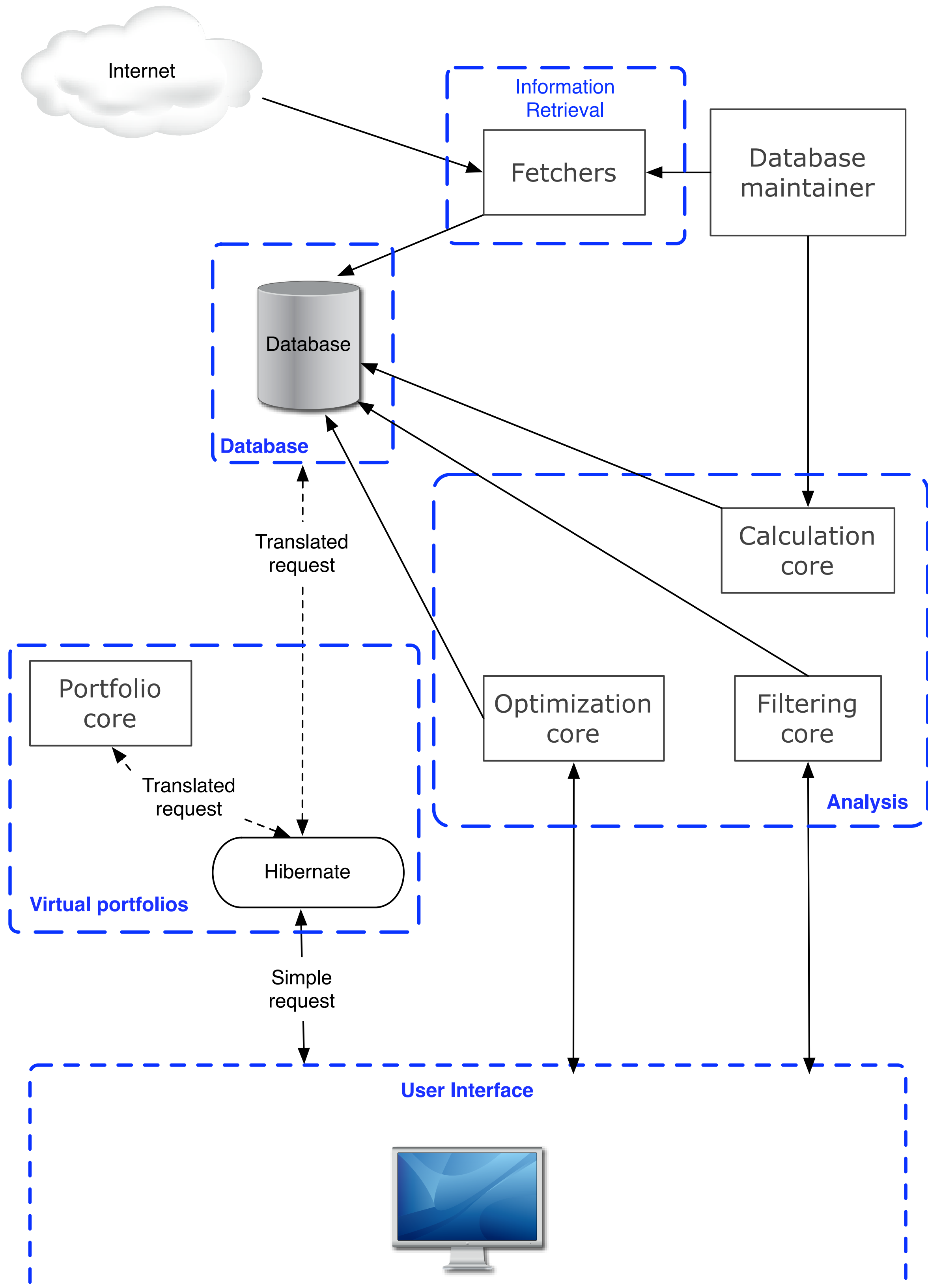

Figure 3.1: Conceptual overview of the system

3.1 Information retrieval

This component is a set of *Ruby* scripts, the fetchers, which are used to download data from internet. All the data fetched by the system is obtained using these scripts. For each data class introduced in Section 3.3 belonging to the fetched group, we have one script in charge of getting the data.
All the scripts download raw data from *Yahoo! Finance*[1], parse it, clean it and validate it; after these checks, the scripts can write data in the database, without the risk of polluting it with wrong values. Some scripts take advantage of the remote API offered by *Yahoo!*; but for some data classes, such as company, there's no remote API and scripts have to parse HTML pages in order to get the raw data.

3.2 Database maintainer

In Chapter 2 we expressed the need for a tool that automates the process of fetching, managing and updating the data.
This core is in charge of these tasks and, in order to accomplish them, collaborates with the fetchers and the calculation core. A schematic view of the collaboration follows.

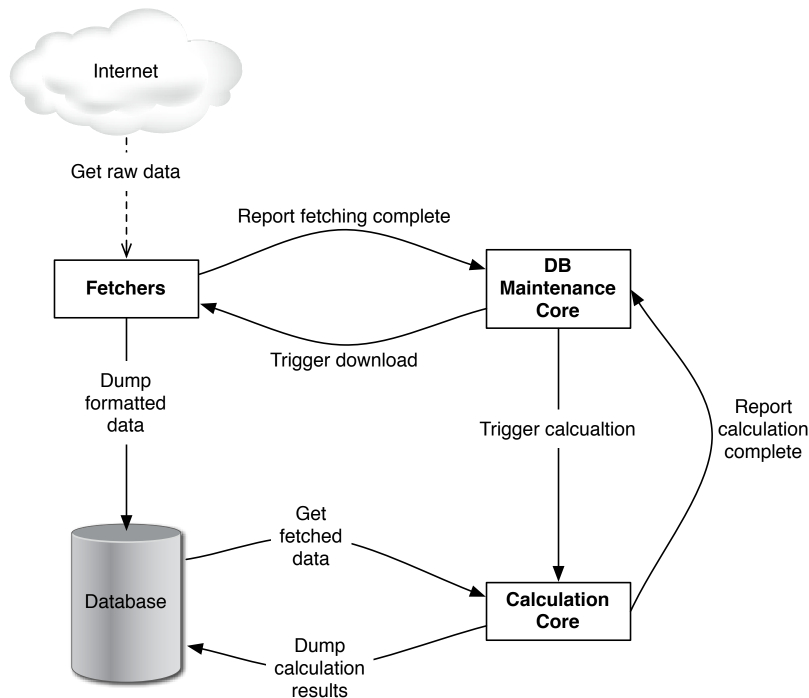

Figure 3.2: Database maintainer and collaborators

This core has its own tasks organized into sets of operations executed at the scheduled time. Each task covers a certain class of data in the database. Depending on the scheduled task, the core will trigger either one or both its collaborators.

[1]See: http://finance.yahoo.com/

3.2.1 Scheduling

Depending on the frequency with which tasks are performed, we distinguish them in two groups: high frequency tasks and daily frequency tasks.

High frequency tasks (every 15 minutes)

Up to now, the only task performed so frequently is updating real-time data.
Notice that in order to keep the latest correct values of real-time data, we don't update it when the market is closed.

Daily frequency

Tasks in this group are performed each day after the market has closed. The first task performed in this group consists of checking whether new stocks were put in the market or whether some stocks left it. In the first case the core will have to run a set of initialization tasks. In the latter case a set of termination tasks will be performed. After the former sets of tasks are completed, the system can proceed to perform a set of updating tasks.
A description of which tasks the mentioned sets contain follows.

Initialization tasks

For each new stock *s* in the market, do the following:

- Fetch nominal data for *s*
- Fetch company information for *s* - only in case that *s* belongs to a new company entering the market
- Fetch historical data of *s* - only one day is fetched since *s* is a new entry.
- Fetch daily data of *s*
- Calculate data belonging to common and filtering data classes

Note that analysis run on new stocks will not be very helpful until the mass of their historical data grows with time.

Termination tasks

For each stock *s* leaving the market, do the following:

- Delete nominal data for *s*
- Delete company information for *s* - in case *s* is the only stock its company has in the market
- Delete virtual portfolios data regarding *s* - as in reality, users owning bankrupt stocks will lose the money invested on them
- Delete historical data about *s*
- Delete daily data about *s*
- Delete real-time data about *s*
- Delete common and filtering data about s

Updating tasks

For each stock *s* in the database do:

- Update historical data of *s* - add data of the market day that just ended
- Update daily data about *s*
- Recalculate data about *s* belonging to common and filtering data classes - new historical data has to be considered in common and filtering data

3.3 Database

3.3.1 Data classes

We divided data classes in two groups: data fetched from internet and data generated by the system, such as cached values for calculations.

Fetched data

Historical data
: This class contains max, min and closing price of a stock for each day in the past. Since it contains the essential data about the history of a stock, it is extremely useful for any statistical analysis. Because of the quantity of data belonging to this class, we limit the maximum history length of each stock to at most ten years.

Real-time data
: Beyond some other values, this class of data contains the ask (buy price) and bid (sell price), which keeps changing throughout the entire market day. Such data is the most mutable in the whole system and is refetched every 15 minutes.

Daily data
: Data in this class informs users about various factors related to the performance of stocks. Because of this, such data is useful when filtering.

Company Information
: This class contains data about the companies stocks belong to. Some of this data, like the industry in which the company works, are used to filter stocks, while other information, like the companies' home page are used as additional source of information offered to the user.

Nominal data
: Data in this class can be seen as a simple list of stock and index names. In other words this data just declares the existence of a stock or an index. Moreover, in the database tables, this data is referred to from almost all the tables of the other stock/index related classes (e.g. historical data).

System generated data

Common
: In this class are contained the cached values calculated by the calculation core, which are used by filtering and optimization cores. It is recalculated at the end of each day because it depends on *historical data*.

Filtering
: This class contains data calculated by the calculation core, which is used when filtering. This data is recomputed at the end of each day, because it depends on *common data*.

Virtual Portfolio
: In this class lies all data necessary to maintain the virtual portfolios of each user.

Logging
: Logged transactions of the virtual trading made by users (e.g. user *i* has bought a certain amount of shares of stock *j*) are contained in this class. These transactions are a sort of backup from which portfolios can be reconstructed in case of non-hardware related failures.

User
: Data like user login, password, e-mail etc. . .

The database tables for the *User* and *Logging* data classes have been designed in strict collaboration with Gilad Geron, because they are tightly coupled with the interface. *Virtual Portfolio* tables where designed with Mr. Geron too, since he implemented the virtual portfolio feature in his interface .

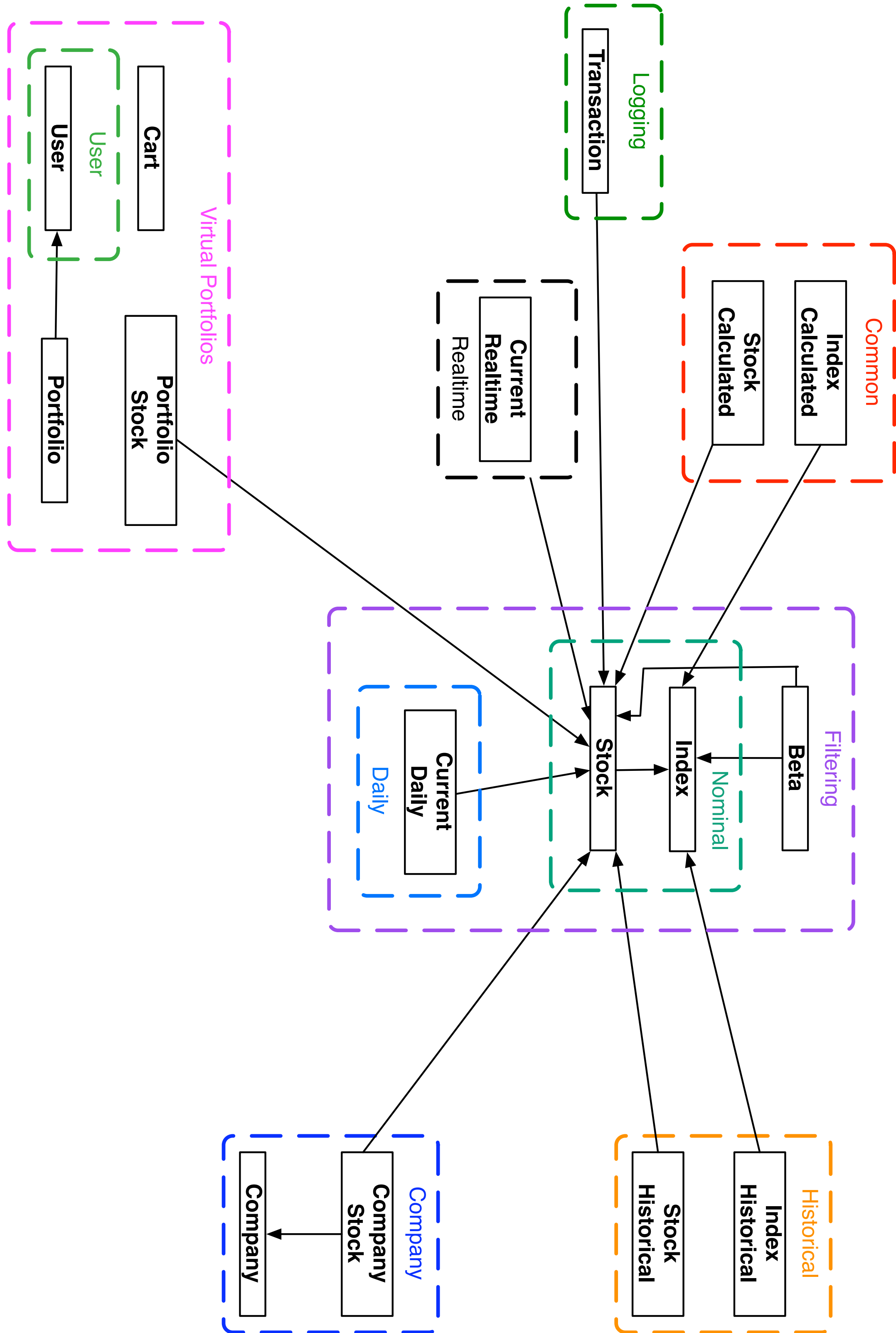

Figure 3.3: Detailed database structure, with tables grouped in the various data classes

3.4 The analysis component

This component is in charge of performing all analysis on the data contained in the database. While going through each of the subcomponents, we give details about what these analyses and their results are.

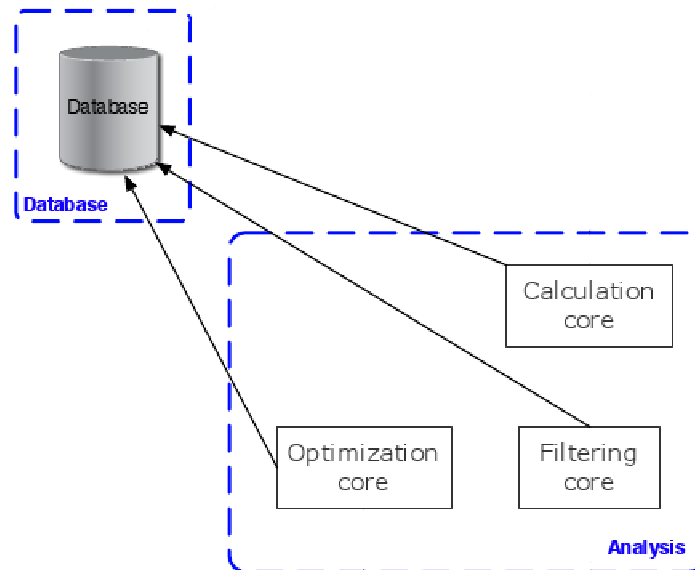

Figure 3.4: The analysis component

3.4.1 Calculation core

This subcomponent reads from the database data that was previously fetched, runs various calculations on it and writes the results back in the database. The purpose of this subcomponent is to calculate data which will be subsequently used by the other two cores to accomplish their tasks. We refer to the calculated data as "cached values", because it is made persistent in the database and does not have to be recalculated each time by the other subcomponents.
Moreover, after new data has been fetched, this component is activated by the database maintenance in order to keep the calculated data up-to-date with the fetched information. A list of the cached values and their definition follows.

Daily rate of return (daily RoR)

It is the percentage change of a stock's price between two subsequent days.
This data is useful to describe the trend of a stock.

Variance

It measures how much a variable is away from its expected value (mean). It is defined as:

$$var(X) = \sum_{i=0}^{N}(X_i - \mu)$$

where μ is the mean of x and N is the number of observations of x

Beta

$$\beta = \frac{cov(X,Y)}{var(X)}$$

where $cov(X,Y)$ is the covariance. [3]

Covariance

It is the measure of how two variables change together; if variables x and y have a positive covariance, it means that they are both above or below their means.
When one is above and the other down, and viceversa, the covariance is negative.
Covariance is defined as:

$$cov(X,Y) = \sum_{i=0}^{N}(X - \mu)(Y - \nu)$$

where μ is the mean of x, ν is the mean of y and N is the number of observations of x and y.

3.4.2 Filtering core

When starting a new investment project, investors might have some preferences and/or some constraints on the stocks to buy. For example one might not like to invest in firms of the war industry or might be constrained, for some reasons, to invest only in the Swiss market. Such an investor has no idea in which stocks to invest yet, so he has to search through the stocks.

In order to restrict the set of stocks to examine and exclude the ones that do not suit the investors' needs, we included a filtering system in the framework. After the user expressed his preferences and constraints as requirements, the system will make a parameterized search in the database and return back the set of stocks that satisfy the user's requirements.
It is important to notice that filtering is using not only the data fetched from internet regarding the stocks, but also data generated by the Calculation core like, for example, *beta*.

3.4.3 Optimization core

Between the stocks on the market, we can identify various stocks that have a good performance, but that are too expensive for many investors. What investors usually do is to replicate the performance of expensive stocks using a set of cheaper stocks.

In a replication, the investor chooses the stocks he wants to use, then he invests different percentages of his budget in each of them. In order to find the percentages that lead to a good replication, the investor needs to run a type of analysis called optimization.

There are many different optimization techniques and thanks to the way we designed the framework, it is easy to add them. Due to time constraints, at the current version, the framework features only one optimization technique, the Style Analysis.

Style Analysis

The purpose of this analysis is to help investors in performing good replications.
Here's an example:

An investor wants to replicate the expensive stock *e* with a set of cheap stocks *c*. Style Analysis is used to find out how much of the budget, in percentage, should be invested in each one of the stocks in *c*, in order to best replicate *e*.

It is defined as $Y = \hat{\beta}X + \epsilon$, where

- Y is a vector of daily RoR, which describes the trend of the stock to replicate;
- X is a matrix of daily RoR built by joining the vectors of daily RoR describing each stock used in the replication; the number of rows in X coincides with $|Y|$;
- β is a vector of coefficients;
- ϵ is the tracking error and represents the imprecisions that prevent us from getting a perfect replication;

We aim to find a $\hat{\beta}$ minimizing ϵ

$$\min(\epsilon) = \min(\,(Y - \hat{\beta}X)^2\,)\ ;$$

To do this, we will use a technique called OLS, which states:

$$\hat{\beta} = \frac{X'Y}{X'X}\ ;$$

To see how well β fits Y, therefore how good the replication will be, we calculate R^2:

$$R^2 = \frac{\sum_{i=0}^{N}(\hat{Y}_i - avgY)^2}{\sum_{i=0}^{N}(Y_i - avgY)^2}$$

where

$N = |Y|, \quad avgY = \frac{\sum_{i=0}^{N} Y_i}{N}$

$\hat{Y}_i = \beta_j X_{i,j}$; with $j = 0 \ldots |\beta|$ and $i = 0 \ldots |Y|$

[4] [1]

3.5 Virtual portfolio

Our framework is meant to help investors by proposing them investment solutions. Typically users would like to deeply inspect the investments proposed by the system and have the possibility to save and modify them in the future.
In order to offer such features, virtual portfolios and a virtual portfolio management system are necessary. As said at the end of Section 3.3 Gilad Geron implemented this feature; since this feature affects the database and is very important, we think it is worth to describe it shortly.

- budget - the amount of money available to buy new stocks or buy more shares of stocks already in the portfolio
- entry - an entry consists of the name of a stock, the number of shares owned for that stock and its price at the time of trading; a portfolio can contain zero or more entries

Since portfolios are meant to group shares of traded stocks, the concept of portfolio requires the existence of a concept of trading, which requires the concept of price.
This gives the need to keep the latest price data in the database; such data belongs to the real-time data class discussed in Section 3.3.
As can be seen in the scheme below, the framework allows interfaces to access virtual portfolios data through the subcomponent Hibernate, which is discussed in the next section.

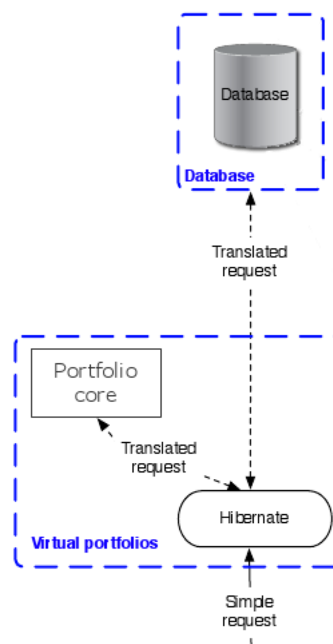

Figure 3.5: The virtual portfolio component

3.6 Hibernate

Hibernate [2] is an ORM (Object Relational Mapping) technology, which allows an application to manage tables in a database by manipulating *Java* objects.
Thanks to this technology, a *Java* application using a database, is made independent from the underlying database technology, being it MySQL, Oracle or SQLite, etc ...

Moreover the application code in charge of database interaction is greatly simplified. As a drawback of *Hibernate*, and all other ORMs, performance decreases.

We decided to include this ORM in the framework when, while designing it, we discovered the existence of a particular set of operations. Operations in this set access the database, are not performance critical, are requested by users through an interface and are preformed frequently.
Although it is an instance of the concept of ORM, for the sake of simplicity, we will refer to this ORM component as *Hibernate* throughout the whole document. A more detailed description of this technology is given in Section A.1.4.

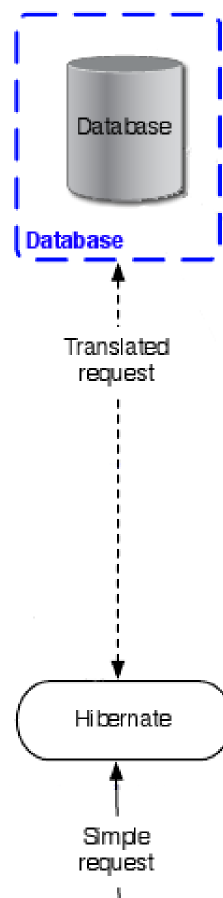

Figure 3.6: Hibernate relation with database

[2]See: http://www.hibernate.org/

3.7 Performance of the framework

In order to show the performance of the system, we give a description of the setup phase and of the functionality access performed by a user.
Before beginning to setup the system, the user has to specify in a configuration file the details for the connection to the database to use. As first step of the setup the user has to run the *initialize-index.rb* script from the fetchers, which downloads data about the specified index and about the stocks it contains.

In our tests we used a line limited at 300 Kb/s. It took 80 minutes to initialize the Nasdaq, one of the largest indexes, which contains more than 3.050 stocks. The table *stock_historical* contained more than 5.710.000 rows and weighted less than 350 MB.

After that data has been downloaded, the user has to activate the *database maintainer* manually. The core instructs the *calculation core* to calculate the cached values for all stocks in the database.
In our tests with the Nasdaq we measured an execution time of almost 255 minutes, which means approximately 5 seconds per stock. The table *stock_calculated* contained more than 4.101.000 rows and weighted less than 95 MB.

After the first two steps, the system setup is complete and users can access the framework's functionality through the interface. When testing the system through Gilad Geron's interface, we realized how effective the cached values are in increasing the speed of analyses: optimizations took less than 5 seconds and filtering the stocks in the Nasdaq was completed in less than 3.

It is important to understand that once the cached values have been calculated from scratch, it takes much less time to update them according to the new historical data fetched at the end of the market day.

Chapter 4

StockHome behind the scenes

4.1 Filtering example

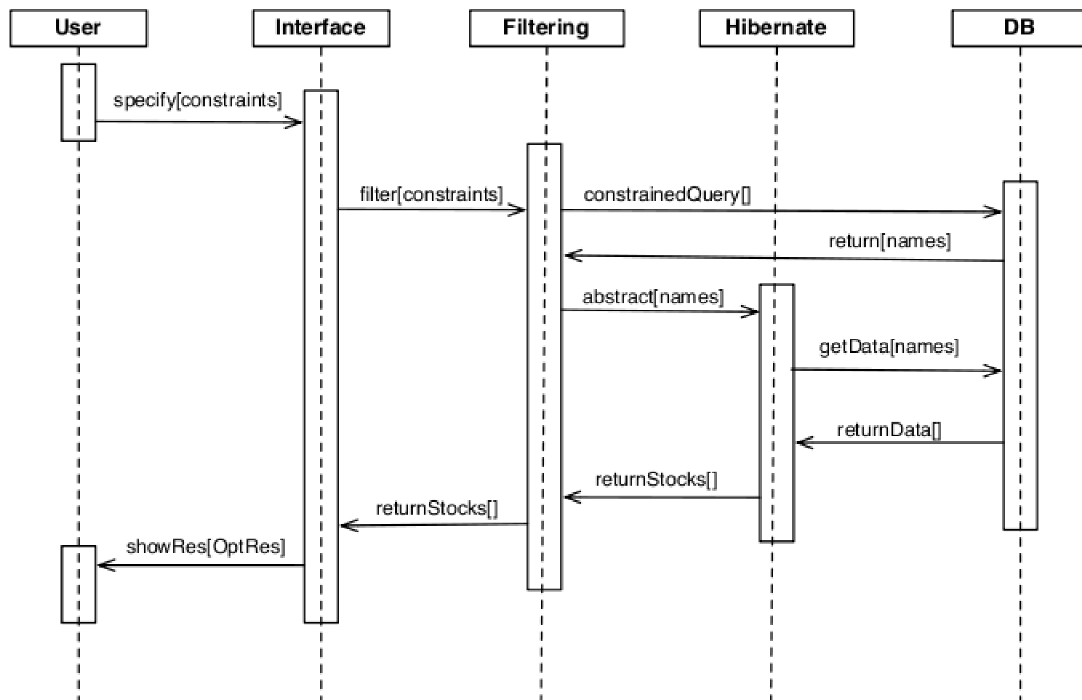

Figure 4.1: Sequence diagram for filtering

The user gives constraints about the stocks through an interface. The interface formats user input and send a request to Filter (filter[constraints]).
To filter the stocks, we rely on data of the daily data class and on cached values belonging to the filtering data class. Since everything is in the database, Filter has just to build a query expressing all the constraints (constrainedQuery[]). As requested by the filtering core, the database returns the names of the stocks that satisfy all requirements (return[names]).
For the sake of easy interface programming, Filter uses *Hibernate* to abstract to objects

the stocks satisfying the requirements (abstract[names]). *Hibernate* requests additional data from the database in order to perform the abstraction (getData[names]); after that the database answers back (returnData[]), *Hibernate* abstracts the stocks to objects and sends these objects to Filter (returnStocks[]), which forwards them to the interface. The interface shows the stocks to the user.

4.2 Search for a stock

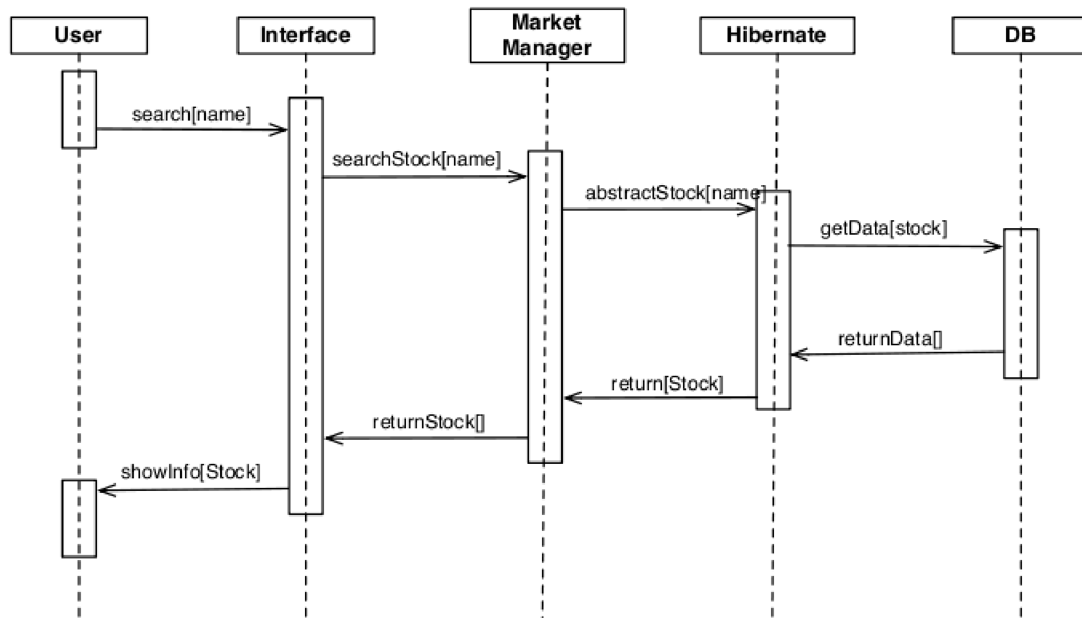

Figure 4.2: Sequence diagram for a stock search

The user searches for a stock by specifying its name via an interface. This interface sends a request to the MarketManager class (searchStock[name]).
MarketManager sends a request to *Hibernate* (abstractStock[name]), which reads from the database data regarding the stock (getData[stock] and returnData[]).
When *Hibernate* has all the necessary data, it creates an instance of the Stock class, which represents a stock in the database. The stock object is returned to MarketManager and forwarded to interface (returnStock[]).
The interface shows the stock's information to the user by accessing the stock object (showInfo[Stock]).

4.3 Optimization example

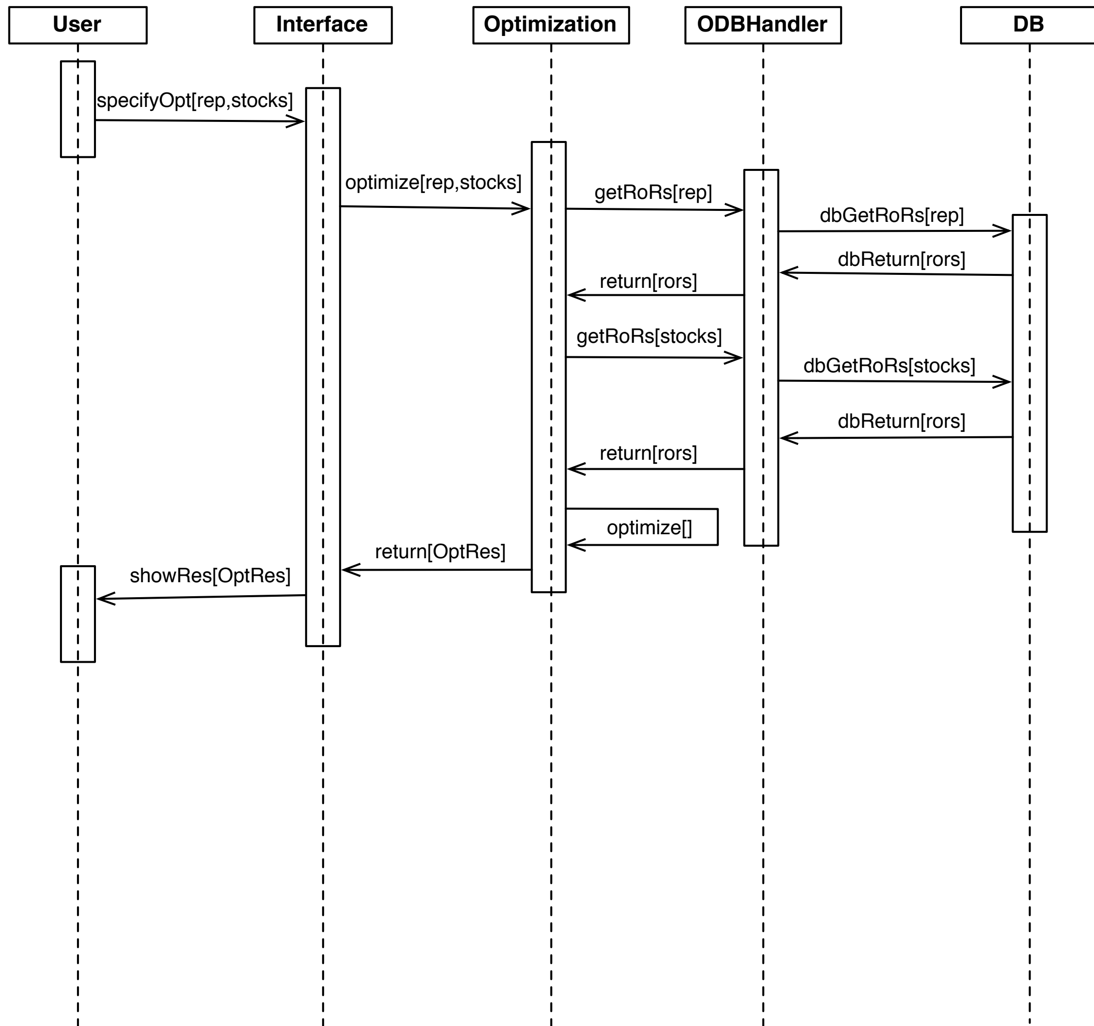

Figure 4.3: Sequence diagram for optimization

The user specifies, through an interface, a stock to replicate and which stocks to use in the replication (specifyOpt[rep, stocks]). The interface formats user input and sends a request to the Optimization class (optimize[rep,stocks]).
To perform its task, Optimization needs to read from the database some data belonging to the common data class. This task is delegated to ODBHandler (dbGetRors[rep] and dbGetRors[stocks]), which queries the database.
After the response from the database, ODBHandler sends the information back to Optimization, which performs the optimization procedure (optimize[]).
The result of the analysis is an instance of the OptResult class. This object contains all the results of the optimization and provides easy access to them.
Optimization sends the OptResult to the interface (return[OptRes]), which shows the results to the user by accesing the OptResult instance.

4.4 Virtual portfolio creation

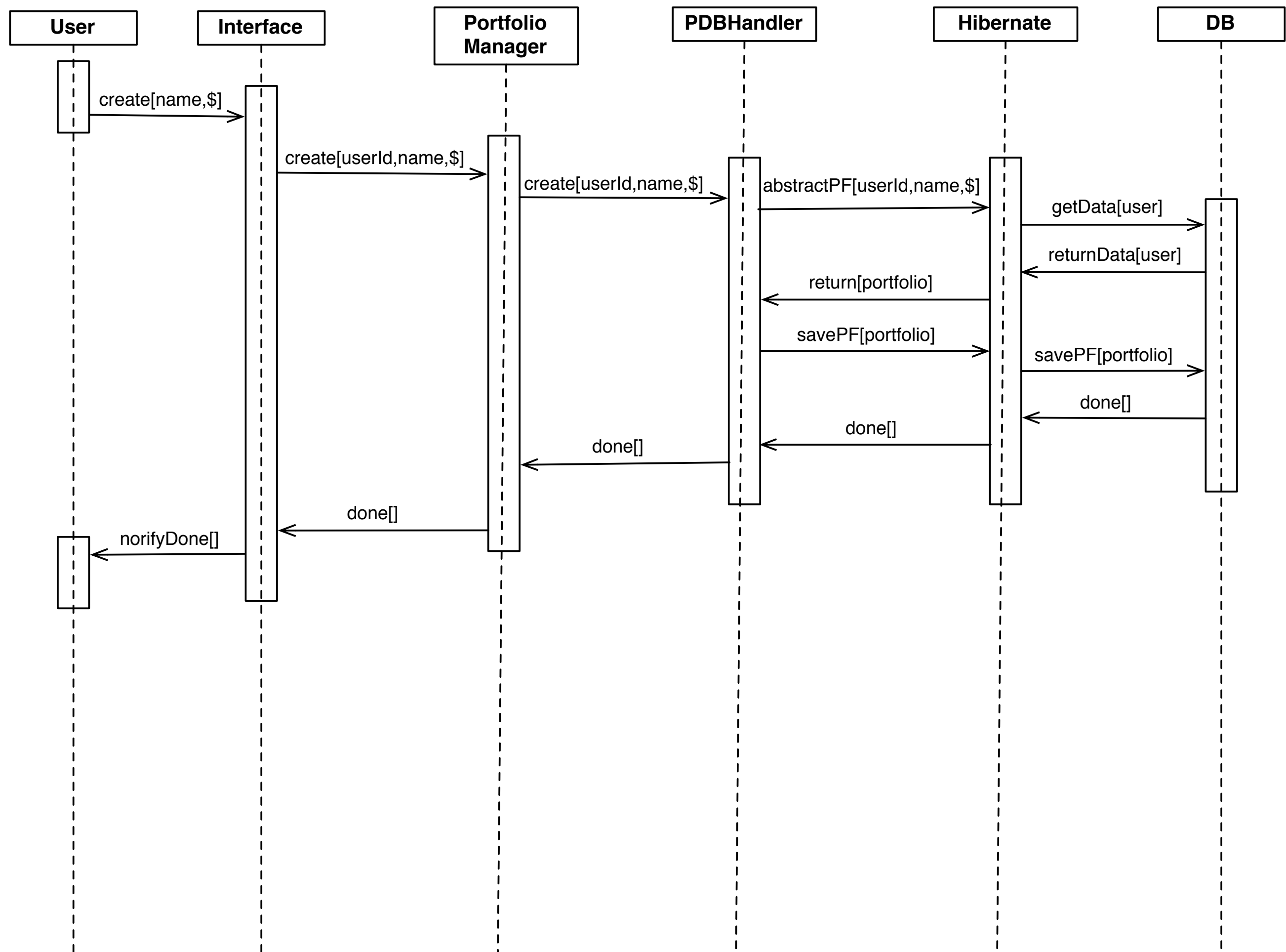

Figure 4.4: Sequence diagram for creation of portfolios

The user specifies a name and the initial budget ($) for the new portfolio. The interface adds the id of the user to the input and sends a request to PortfolioManager (create[userId, name, $]).
PortfolioManager delegates the portfolio creation to *Hibernate*,which , after reading data about the user from the db, initializes a Portfolio. The object is initialized using both information specified by the user and data read from db (getData[user]).
PortfolioManager performs some modifications to the returned Portfolio and then tells *Hibernate* to save these modifications in the database (savePF[portfolio]).
After being informed by *Hibernate* that the portfolio has been created, PortfolioManager acquaints the interface with the procedure's completion (done[]).
The interface notifies the user about the task's completion.

4.5 Buy a stock

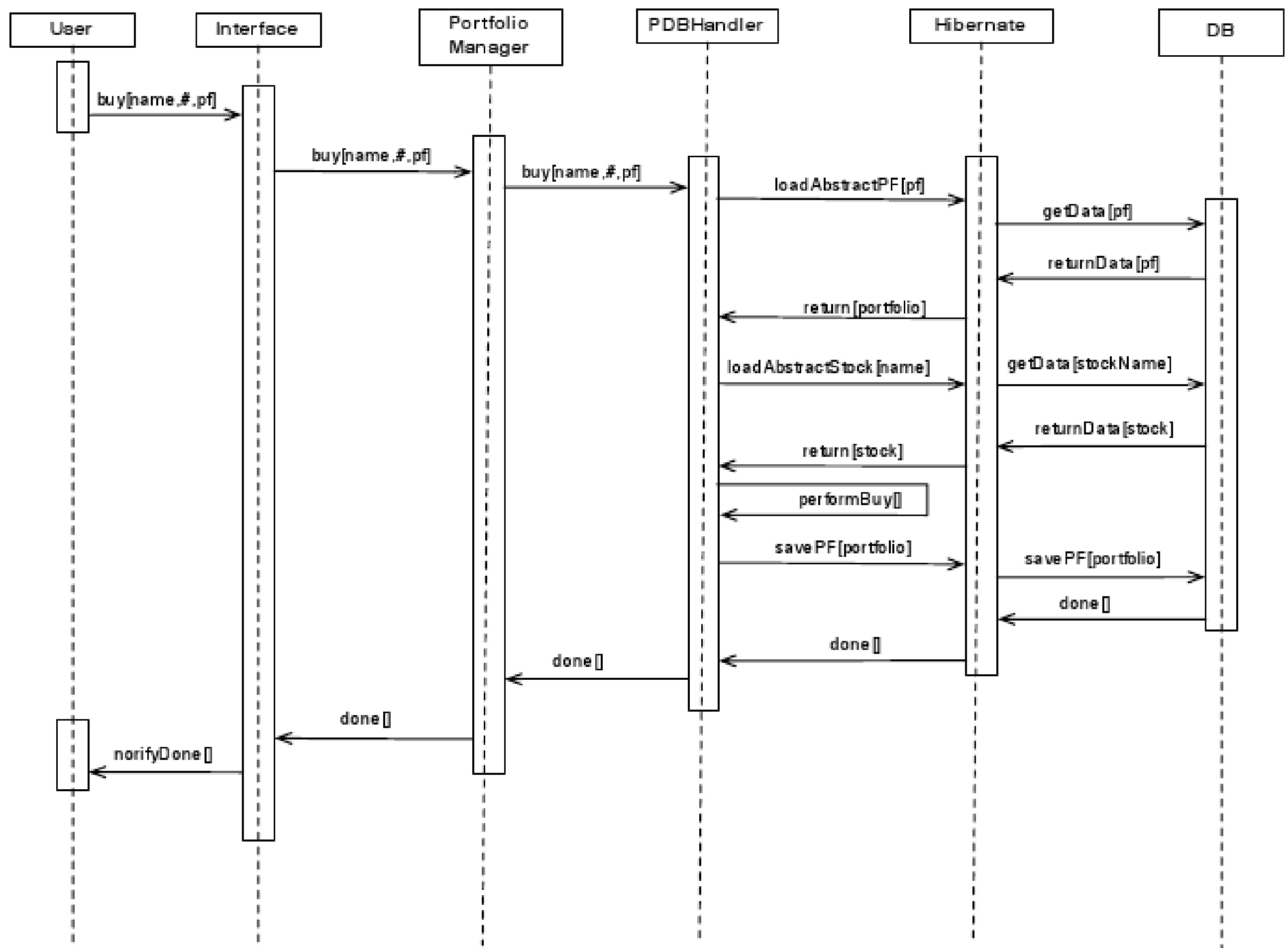

Figure 4.5: Sequence diagram for purchase of stocks

The user specifies the name and how many shares (#) of a desired stock he wants to buy. A portfolio in which to put the stock and take the necessary money from has to be specified too. The interface sends a request to PortfolioManager, which forwards it to PDBHandler (buy[name,#, pf]).
The handler requests *Hibernate* for an instance of Portfolio. Hibernate reads the related data from the database and returns the requested objects.
To obtain the stock price, PDBHandler uses the same collaboration with *Hibernate*, but this time requests a Stock object representing the stock the user wants. When the handler has all the required data, it checks that the transaction can be carried out (e.g. if the budget in the portfolio is enough) and executes it by modifying the Portfolio instance (performBuy[]).
PDBHandler delegates to *Hibernate* the task of writing the changes to the database (savePF[portfolio]).
When *Hibernate* reports completion, PDBHandler passes the message to the interface through PortfolioManager(done[]). The interface notifies the user about completion.

Chapter 5

Conclusions

In four months we obtained a software solution bringing many novelties in the market of financial tools and various advantages to users.
We targeted two groups of users, one formed by investment consultants and one containing people willing to invest without the help of an expensive human consultant.
We started from the problems identified in the market that were discussed in Chapter 2 and built a tool overcoming all these problems.
However the framework is still missing features that other applications in the market already have; these features are of great interest for the users we targeted and are discussed in the next section.

5.1 Future Work

The current version of the framework is able to fetch only one type of asset which is the simple stock. So far we can run analysis on stocks and options (another type of asset). However the database does not support options yet.
As part of the future work, we want to integrate in the framework support for two more asset type: options and futures.
Supporting these assets means expanding the framework by:

- writing new fetchers for data regarding these assets
- expanding the calculation core to generate cached values for analysis on these assets
- reorganizing the database to support the assets
- reviewing Hibernate to have it mapping the new database
- adapting filtering and optimization cores to support the new assets
- modifying the portfolio core to allow users to trade these assets

Another important feature is a way to simulate the solutions proposed by the system. When simulating an optimization for example, we should measure how well the allocation scheme proposed by the system replicates the expensive stock.
The best way to do it is using data from the past: measure how good the replication would have been if done three months ago. This simulation technique is called *backtesting*.
Since we have in the database all the necessary data, what we need to add this simulation feature is a set of algorithms.

We also want to expand the analysis capabilities of the framework by adding new optimization techniques and increasing the number of parameters that can be specified when filtering the assets.

5.2 Applications of the framework

In order to obtain a financial analysis tool, our framework has to be integrated with an interface. Any type of interface can be developed to work with the framework, but if they interface does not bring any innovation, the outcome will be a tool without any novelty.
Gilad Geron, a bachelor student at USI, developed an innovative interface as his bachelor project. We give now a description of Gilad's interface.

5.2.1 Gilad Geron's web interface

The main feature distinguishing Geron's interface from the others, is the absolute portability he obtained. Gilad integrated our framework in *Tomcat*, a servlet container.
By having our framework, running on a server featuring *Tomcat*, offer its functionality through a web interface, he created an on line service.
In this way, users will not need to go through any installation phase and would be able to access the functionality of the framework even through a computer with very small computing power.
In order to prevent his interface from missing the usability of interfaces running locally, Mr. Geron took advantage of the newest web technologies such as AJAX[1], Prototype[2] and JSON[3].
A screenshot of Gilad's application follows.
[2]

[1]See: http://en.wikipedia.org/wiki/AJAX#Bandwidth_usage
[2]See: http://www.prototypejs.org/
[3]See: http://www.json.org/

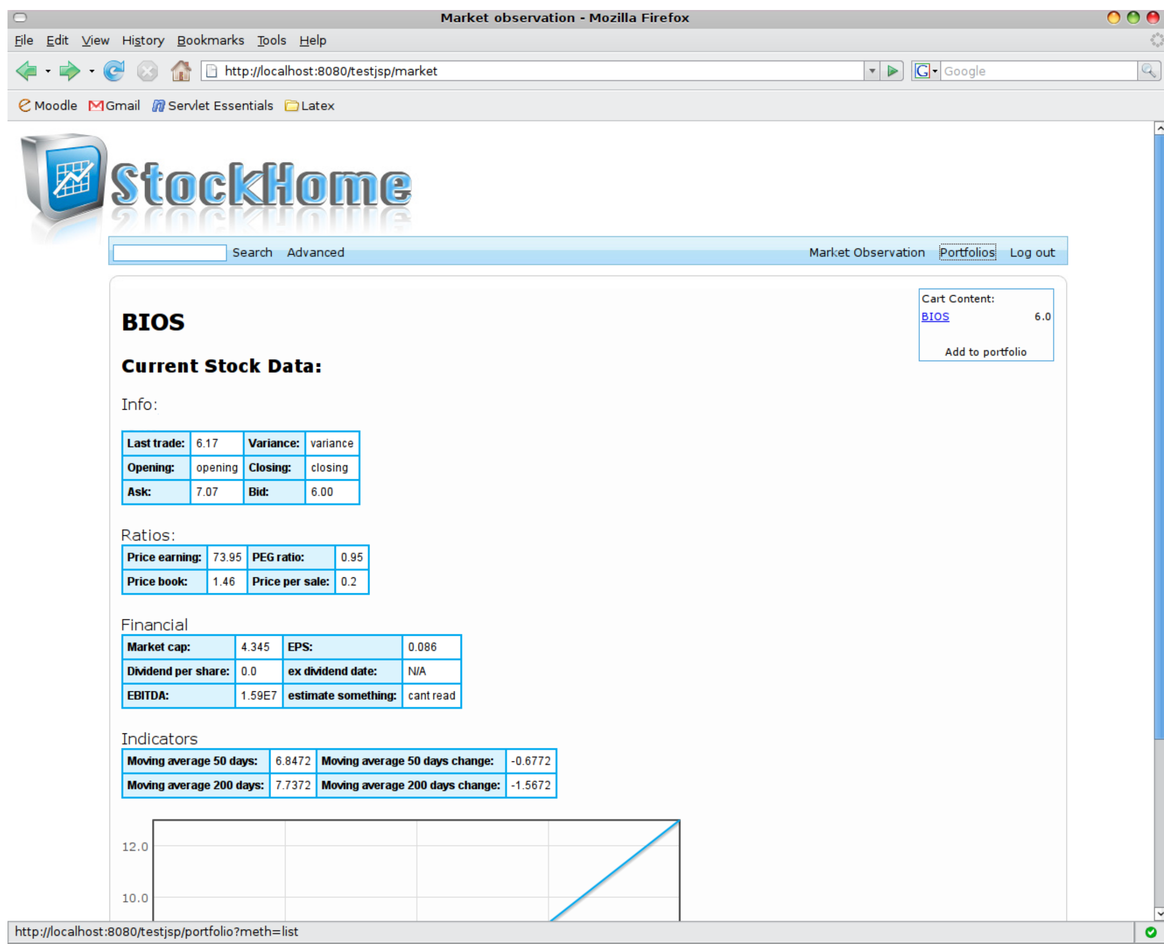

Figure 5.1: Screen from Gilad Geron's web application

Appendix A

Implementation

A.1 Technologies used

A.1.1 Java

Java is a programming language originally developed by *Sun Microsystems* and released in 1995 as a core component of *Sun Microsystems*' *Java* platform.
The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. *Java* applications are typically compiled to bytecode that can run on any *Java* virtual machine (JVM) regardless of computer architecture.[6]
Sun recently released the majority of their *Java* technologies under GPL.

A.1.2 Ruby

Ruby is an object oriented programming language created by *Yukihiro Matsumoto* in 1995. It is dynamic, reflective, open source and features a set of incredibly powerful libraries.
Ruby has many similarities to *SmallTalk* in its object model, but includes also features that belong to scripting languages like *Perl*.
Ruby's impressive growth is largely due to its killer application: *Ruby on Rails*, a very popular open source web framework.

A.1.3 MySQL

MySQL is a multithreaded, multi-user SQL database management system, which has more than 11 million installations. It is popular for web applications and . . . for open-source bug tracking tools like Bugzilla.
Its popularity for use with web applications is closely tied to the popularity of PHP and Ruby on Rails, which are often combined with *MySQL*. PHP and MySQL are essential components for running popular content management systems such as Joomla!, e107, WordPress, Drupal, and some BitTorrent trackers. [7]

A.1.4 Hibernate

Hibernate is an object-relational mapping (ORM) library for the *Java* language, providing a framework for mapping an object-oriented domain model to a traditional relational database. *Hibernate* solves Object-Relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions. [5]

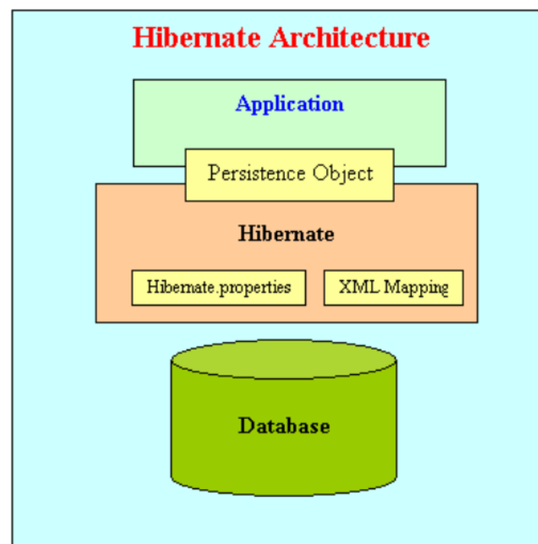

Figure A.1: Where *Hibernate* fits between application and database

Hibernate makes applications independent from the underlying database technology. However it becomes too slow when used with large databases with many references between tables; moreover its setup phase is heavy and time-consuming. In the case of our framework, all the benefits we gained from this technology weren't worth the efforts to set it up.

A.2 Technology choices

- MySQL
  We decided to use this technology for our database because we are used to its syntax and to *MySQL*'s graphical tools

- Java (for core functionality)
  We were determined to implement all the functionality regarding the analysis, as the whole framework, in *Ruby*.
  But in the end we decided to use *Java*, because we needed a certain level of speed, which *Ruby* is not yet able to offer, and a threading model taking advantage of multiple cores (*Ruby* does not support native threads in its stable version).

- Hibernate
  We wanted to have an ORM system in order to facilitate the development of interfaces to the framework. Before choosing *Java* as the language of the framework, we wanted to use *ActiveRecord*, *Ruby*'s ORM. *Hibernate* was the only choice for an ORM for *Java*.

- Ruby
  We decided to write data fetchers in *Ruby* because of its powerful libraries for network interaction.

A.3 Framework structure in deep

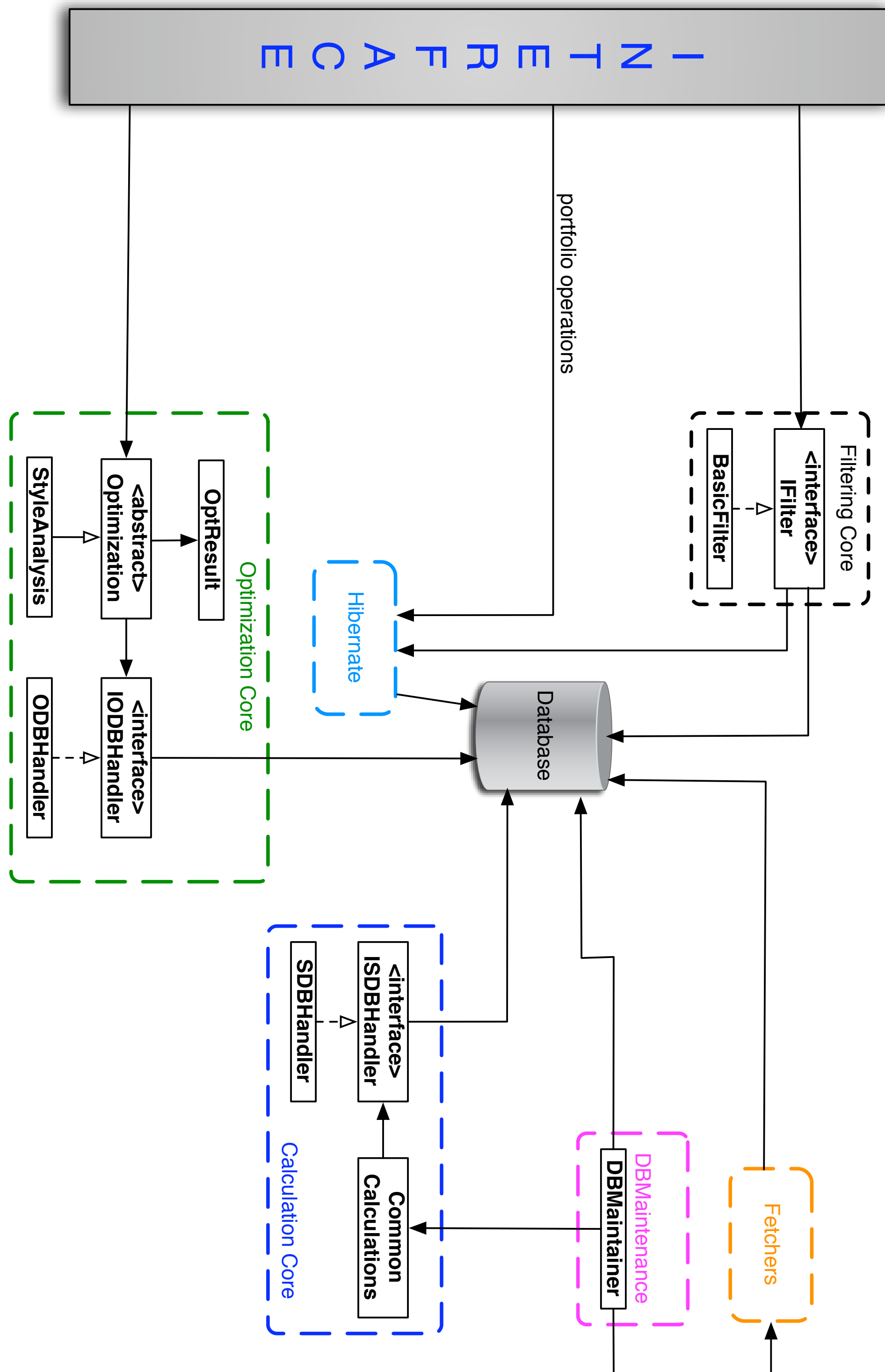

Figure A.2: A detailed view of the framework

A.3.1 Filtering

IFilter offers filtering functionalities to the interface. It contains a single method, *filter*, which allows to filter stocks on data belonging to the company, daily and filtering data classes.
In its implementation of the method, BasicFilter, builds one query for each data class and then nests them using the *in* clause: we want an intersection of the results.

A.3.2 Optimization

Optimization is an abstract class featuring the following methods:

- RoRs(stock)
  a method returning an array of daily RoRs of the stock to replicate; this data is read from the database using an implementor of the IODBHandler interface.

- initializeX(stocks)
  this method returns a matrix of dailyRoRs where rows represents different days and columns represent the different stocks. An implementor of IODBHandler is used in this task too.

- replicateStock(rep, stocks)
  this abstract method returns an OptResult instance

OptResult represents the result of all optimization techniques, and contains the following:

- weightsMap - a HashMap where stock names are keys and budget percentages are values.

- R^2 - the score of the obtained set of weights (more details are given in Section 3.4.3)

In order to add an optimization technique to the framework, a class extending Optimization and featuring an implementation of *replicateStock* has to be written.

IODBHandler features the two methods in charge of building the array of RoRs and the matrix of RoRs; ODBHandler is the only implementor.
When not enough data is available for one stock, 0s can be used to fill the empty cells in the matrix: a value of 0 for a RoR does not affect results, like multiplying a number by 1.

SytleAnalysis contains all methods used to perform this type of optimization.
Some of them follow:

- replicateStock(rep, stocks)
  after getting the RoR matrix and the RoRs for rep, it calculates $\hat{\beta}$ and R^2.

- calculateStimatedY
  starting from $\hat{\beta}$, it builds an array of RoRs representing the performance of a virtual stock built as $\hat{\beta}X$.

A.3.3 Hibernate

This package contains the classes representing the tables in the database and the mapping files from which the ORM generates these classes.
Here is an example mapping file:

```
<?xml version="1.0"?>
<hibernate-mapping>
    <class name="dbobjects.Company" table="tbl_company">
        <id name="id" type="java.lang.Integer">
            <column name="company_id" />
            <generator class="identity"></generator>
        </id>
        <property name="companyName" type="string">
            <column name="company_name" length="20" not-null="true" />
        </property>
        <set name="companyStocks" inverse="true">
            <key>
                <column name="company_id" not-null="true" />
            </key>
            <one-to-many class="dbobjects.CompanyStock" />
        </set>
    </class>
</hibernate-mapping>
```

Figure A.3: An example *Hibernate* mapping file

In this file we map the class *Company* to the table *tbl_company*.
Setting to **identity** the attribute **class** in the **generator** element tells *Hibernate* to let the management of ids to the underlying database.
The **property** element declares an attribute of *Company* of *Java* type String mapped to the column *company_name*.
The **set** element is used to declare that a set of instances of another class is attached to each instance of, in this case, *Company*. A relation in the objects, is representing a relation in the tables of the underlying database. In this case the set is representing a **one-to-many** relation between *tbl_company* and *tbl_company_stock*.

A.3.4 Calculation Core

ISDBHandler offers methods for retrieving historical data from the database, inserting, and removing, calculated data of the common and filtering data classes.
CommonCalculations is in charge of calculating all the cached values and storing them in the database using an implementor of ISDBHandler.
The two main methods of CommonCalculations, *computePropertiesInitial(stocks)* and *computeProperties(stocks)*, calculate the same types of values, but in two different ways and at two different times.

computePropertiesInitial is run only in the initial database setup and when new stocks enter the market. Pseudocode follows:

```
read historical data of all indexes
compute RoRs and variance for all indexes
write RoRs and variance to db

for each stock s do:
  read historical data of s
  compute RoRs and variance for s
  write RoRs and variance to db

  for each index i do :
    compute beta(s, i)
    write calculated beta to db
  end for;
end for;
```

computeProperties is run daily after new historical data has been fetched.
Pseudocode follows:

```
for each index i do:
  read historical data of i
  compute the RoR(yesterday, today)
  read variance of i
  recalculate variance and write it to db
end for;

for each stock s do:
  read historical data of s
  compute RoR(yesterday, today)
  read variance of s
  recalculate variance and write it to db

  for each index i do:
    compute beta(s, i) and write to db
  end for;
end for;
```

A.3.5 Fetchers

constants.rb
Holds configuration data shared by all the fetchers like, for example, database connection information.

data-preparer.rb
Holds methods used by other fetchers to format fetched raw data before storing it in the database.

company-info.rb
Fetches, format and store company data in the database; no remote API from Yahoo! exists for company data, so the script has to parse HTML files.

index-components.rb
Holds methods to fetch, format and store data of index components, including nominal data and company names.

historical.rb
Holds methods used to fetch and store historical data for both indexes and stocks.

historical-init.rb
Uses various methods in *historical.rb* to fetch, and store historical data for indexes and stocks. Can be run independently by specifying a bunch of stock names and a history time slice to fetch.

update-historical.rb
Holds a method to remove the oldest stock in the history. It is run independently to fetch and store new historical data and delete the most obsolete. It uses methods in *historical.rb* to accomplish its tasks.

insert-current.rb
Fetches, formats and store real-time and daily data for a set of stocks. It is used only once for each stock, either when initializing the database or when the maintenance core finds a new stock in the market.

update-current.rb
Fetches, formats and store real-time and daily data for a set of stocks. Unlike *insert-current.rb*, it performs *UPDATE* queries. It is run every 15 minutes for real-time data and every day for daily data.

initialize-index.rb
Given an index name, it calls methods from all the other fetchers to initialize that index. This script is run once for each existing index when initializing the database.

List of Figures

2.1 Sample of data about the stock BIOS 3

3.1 Conceptual overview of the system 6
3.2 Database maintainer and collaborators 7
3.3 Detailed database structure, with tables grouped in the various data classes 12
3.4 The analysis component 13
3.5 The virtual portfolio component 16
3.6 Hibernate relation with database 17

4.1 Sequence diagram for filtering 19
4.2 Sequence diagram for a stock search 20
4.3 Sequence diagram for optimization 21
4.4 Sequence diagram for creation of portfolios 22
4.5 Sequence diagram for purchase of stocks 23

5.1 Screen from Gilad Geron's web application 26

A.1 Where *Hibernate* fits between application and database 28
A.2 A detailed view of the framework 30
A.3 An example *Hibernate* mapping file 32

Bibliography

[1] Advances in Mathematical Programming and Financial Planning.

[2] G. Geron. Stockhome, 2008.

[3] H. M. Markowitz. Mean-Variance Analyses in Portfolio Choice and Capital Markets. 1987.

[4] W. F. Sharpe. Asset allocation: management style and performance measurement. Journal of Portfolio Management, pages 7–19, 1992.

[5] Wikipedia. Hibernate. http://en.wikipedia.org/wiki/Hibernate_(Java), May 2008.

[6] Wikipedia. Java. http://en.wikipedia.org/wiki/Java_(programming_language), May 2008.

[7] Wikipedia. Mysql. http://en.wikipedia.org/wiki/Mysql, May 2008.