

**MASTER THESIS -  
ABSTRACTION LEVELS OF AUTOMATED TEST  
SCRIPTS**

Andreas Lundgren

A Master Thesis Report written in collaboration with:

Department of Computer Science at  
Faculty of Engineering  
Lund, Sweden

and

Cinnober Financial Technology

Stockholm, Sweden

Mars, 2008



## **Abstract.**

Computer software development needs an efficient and secure development process where reliable automated tests are run on a daily basis. However, the largest cost of having an automated testing framework is not the time spent writing the test, but the time spent maintaining it. If the maintenance efforts could be less extensive and the tests made more dynamic in order to work for many different projects, development costs may be decreased.

This thesis focuses on the abstraction of automated tests, that is, how should the tests be designed in order to keep a high level of abstraction. The thesis treats three different testing frameworks; EMAPI developed by the author, JUnit and the Cinnober proprietary framework Grace. The frameworks are analyzed and evaluated; drawbacks and benefits are presented. Further, a case study is conducted in order to test the abstraction level of test cases in JUnit and Grace. In the case study, tests used in one development project are moved into another project with similar protocol design, that is, some attributes are different in one protocol. Effort of making the tests run with the new protocol structure is examined and analyzed. Since the actual difference between the two projects is protocol changes, the case study should be viewed as adding fields in the existing protocol and make necessary modifications to make the tests run again.

The conclusions drawn are that the frameworks have different strengths and weaknesses. JUnit keeps its abstraction on input and Grace on its output. JUnit is focused on verifying only one thing at the time; thus, the output keeps a high level of abstraction. Grace, on the other hand, does by default focus on an extensive validation in every single test case and uses a post validation approach. Thus, the abstraction for the output validation is low.

## Contents

ACKNOWLEDGMENTS.....	4
1 PROBLEM DESCRIPTION .....	5
1.1 Problem statement .....	5
2 BACKGROUND.....	7
2.1 Market theory.....	7
2.1.1 Market participants .....	7
2.1.2 Markets.....	8
2.1.3 Orders .....	9
2.2 Software Testing.....	11
2.2.1 Background .....	11
2.2.2 Static testing.....	12
2.2.3 Dynamic testing.....	12
2.3 Manual tests.....	14
2.4 Test automation.....	15
2.4.1 Unit testing.....	15
2.4.2 Abstraction .....	16
2.4.3 Predicate structure .....	18
2.5 TRADExpress Trading Engine .....	18
2.5.1 Platform layer.....	18
2.5.2 Product layer.....	19
2.5.3 Solution layer .....	19
2.5.4 The trading engine .....	19
3 The frameworks.....	22
3.1 EMAPI.....	22
3.2 JUnit.....	25
3.3 Grace .....	27
4 METHOD .....	30

4.1	Definition.....	30
4.2	Evaluation method.....	30
5	STUDY.....	31
5.1	Exploratory testing.....	31
5.2	The case study.....	32
5.2.1	Background.....	32
5.2.2	Execution.....	32
5.2.3	Evaluation.....	33
6	RESULTS.....	34
6.1	The case study.....	34
6.1.1	JUnit.....	34
6.1.2	Grace.....	37
7	DISCUSSION.....	38
7.1	Summary.....	38
7.2	Conclusions.....	39
7.2.1	Recommendations.....	40
7.3	Future work.....	42

## **ACKNOWLEDGMENTS**

This thesis was made in collaboration with Cinnober Financial Technology AB and the Department of Computer Science at the Faculty of Engineering at Lund University. I would like to thank my supervisor Lars Wahlberg at Cinnober for all support given in order to make this project possible and all invaluable teaching about software testing. Also, I would like to thank all other Cinnober staff involved that coped with all questions.

# 1 PROBLEM DESCRIPTION

Automatic testing is an interesting and topical field within the software testing area. However, the area is far from new. Papers, books and articles have been written that all have their aspects of automated tests.

There is no doubt that a rigorous testing procedure is a critical factor to successful system development. A big task for developers and testers is to speed up the working process and increase accuracy and completeness. When dealing with tests, the main objectives are maintainability and coverage. For a complex system, like a financial trading system, a large number of tests are needed in order to get decent test coverage. Such coverage is nearly impossible to get through manual testing and would also be very time consuming and cost ineffective. However, when automated tests are used, the time spent is decreased and thus also the costs.

An automated test is a script-based stimulus for the system where an expected output is validated. The system under test is fed with input from the script, and then the actual output is compared to the expected. A validation process decides whether the test fails or not. In order to have an efficient automated testing tool, high maintainability is crucial. If an automated test suit needs to be changed in all its test scripts every time a small change is made in the system, the maintenance process would rapidly increase beyond acceptable limits. How can this high maintainability of the testing tool be achieved?

## 1.1 Problem statement

In this thesis, the importance of keeping a high level of abstraction within automated test will be examined. In order to keep an automated test suit running even though an extensive development process is in progress is a crucial task for efficient software development. But how should this abstraction be constructed and used? Is the abstraction design the same for both input and output? Also, when talking about abstraction, the main design of the test suits is important. How should the validation be made in order to keep the high level of abstraction when changes are made to the system under test? The specific goals of this thesis are:

- Study the design of abstraction layers for two existing test frameworks at Cinnober and also make a structural design of an own framework.
- Identify the differences and similarities of existing abstraction layers both at a Cinnober framework and a JUnit framework.
- Identity pros and cons of different designs of abstraction layers.

- Evaluate abstraction dependence in validation process of automated tests.

## 2 BACKGROUND

This thesis treats automated testing of financial trading systems. In order to get a comprehensive view of the testing procedure, the theoretical section starts with a market overview in order to get the reader familiar with how stock exchanges and financial systems work. The theory section then treats the theory of software testing with focus on the automated testing part. Finally, this section gives an overview of the market trading system used and developed by Cinnober.

### 2.1 Market theory

A market is a place where dealers are brought together, electronically or physically, to sell or buy goods or services. The stock exchange offers dealers to trade goods, futures, currencies or securities. The market can be divided in to different sections. The initial offering of the security is called the *primary market*. This is where a company goes public in order to raise new capital for expansion or get a better liquidity. After the company has gone public the shares are traded in the *secondary market* where investors can convert their assets into cash. The revenues from the secondary market go to the people owning the shares whereas the revenue for the primary market goes to the company itself. A further step is where traders can buy and sell with all listed stocks; this market is called the *third market*. The *fourth market* is where a financial institute on its own deals with the traders. No external brokerage is used, thus more profit for the institute. [1]

#### 2.1.1 Market participants

In order to create a market, *customers* are needed. The most important thing when creating a market is the *customers*. A customer may be a bank, a private person or an institute that wants to sell or buy on the market. However, since a market place is a complex system that demands operations and participants to follow certain rules it would be impossible to allow anyone to trade directly on the stock exchange. Hence, a *broker* is used to facilitate trades for the customer. A broker may be either a person or a company and has to execute customer's orders at the best possible price available in the market. A broker does not assume any risk when dealing for his client, instead the broker charges his customers either a flat or a percentage fee. Whereas a broker deals for a customer, another market participant deals for his own purpose; a *dealer*. Since a dealer is trading for him self, a dealer also assumes a risk. Dealers and brokers are often the same people; the only difference between the two actors is for whom they are dealing. [1]

Another participant is the *market maker*. It is a firm who quotes both buy and sell prices in all financial instruments hoping to make money on the bid/ask spread; which is the difference between the price available for a bid and an ask. A bid offer is sent from someone who wants to buy, and correspondingly, an ask offer is sent by someone who wants to sell. A market maker adds a depth and liquidity to the market by buying stock when other wants to sell and selling when other wants to buy. However, the market makers do not aim to prevent prices from falling or rising, but only to ensure price continuity, that is, adding bids and asks that are above and below the current market price.

### 2.1.2 Markets

In the financial world there exist different types of markets. The different types all depend on how big the market is, that is, how many participants and actors there are. If the security is a stock from a big world wide company, a big market is needed with fast trading conditions. On the other hand, if the security rises from a stock of a small and relatively new company, the market is of a smaller size. For such a stock a *direct search market* is used. On this type of market, a customer who wants to put a bid order has to find another participant who wants to put a matching ask order and vice versa. That is, the trades are so infrequent that no broker provides any service for these trades. It can be hard to find a best price at these markets since there are no dealers, which often have better information and lower transaction fees, to help out. The next step of markets is the *brokered market*. It is almost the same as the smaller direct search market but here there exist brokers who can help customers to find a better price.

Another market is the *quote-driven market*, or a *price-driven market*. In such a market, market makers are obligated to announce bids and asks at any time during the market's opening hours. On this kind of markets, a dealer does not need to wait any time finding an order that complements the dealer order. Since the market makers must assure continuous quotations, the market is fast and has a high liquidity. The market is said to be price driven since it is the price offers that define the market prices. An example of a price driven market is NASDAQ stock exchange.

The last market, presented here, is the *order-driven market*. In this market all seller and buyers submit their orders for a security to a single centralized location. All bids and asks are displayed for all actors and the price and order size are defined. This is also why the market is called an order-driven market, because it is the specific orders that make the market. The order-driven market can be divided into two parts. Either the orders are matched immediately after arrival, which is called a *continuous orders*, or the order are saved and listed; and then after a while in a predefined interval matched together. This is called *discrete orders*.

Many markets are hybrids between the quote-driven and the order-driven systems. A hybrid market is used to provide a high liquidity, and a market where the actor can

trade at any time to any price within the quote range. An example of a hybrid market is the NYSE (New York Stock Exchange).

In order to make the difference between the markets more obvious, a small example is presented in *table 1*. In the example an order book is examined from an order-driven market. Orders can be executed at the prices noted in the order book, no prices in between. On the other hand, if the market was a price-driven market the bid quote would be SEK 15,74 and the ask quote would be SEK 15.76 however, all prices in between are also available bids and asks. [1]

Ericsson B	Round lot: 100			Time: 09:00:43
+/- %	<b>Bid price</b>	<b>Ask price</b>	<b>Last paid</b>	<b>Total amount</b>
-1,99	15.74 SEK	15.76 SEK	15.76 SEK	3247471
<i>Order depth</i>				
<b>Shares</b>	<b>Bid price</b>	<b>Ask price</b>	<b>Shares</b>	
197 000	15.74	15.76	30 000	
26 000	15.72	15.78	116 000	
101 000	15.70	15.80	105 000	
26 000	15.68	15.84	100 000	
34 000	15.66	15.86	122 453	

Table 1: Example of order book for order-driven market, Ericsson B 2007-12-17 [2]

### 2.1.3 Orders

Testing of financial systems could seem to be a pretty easy task. The layman may think that as long as the order size and the order price are correct, the trade could be executed. This is of course not the case. In order to convince the reader of the complexity of a financial system some different types of orders are viewed here.

#### *Market orders*

The simplest order is the *market order*. It is an order without price specification but just a quantity that the actor wants to sell or buy. The order is executed to the best available price. The best available price means to buy at the lowest ask price and to sell at the highest bid price. However, the current best price is seldom the best price over a whole trading day; thus, the actor may not earn as much money as possible when setting a market order. On the other hand, a market order has the highest priority of all order types and therefore it could be of great use if the situation is right.

### *Limit orders*

If the actor wants to get a better price than what a market order gives him, he can put a *limit order*. In such an order the actor specifies the order size and the limit price. Then the order is executed at the limit price or better. Normally an ask order has a limit price above the last paid price and a bid order has a limit price below the last paid price. If the bid or ask price, respectively, does not reach the limit order price, the order is not executed but resides in the order book. Furthermore, even though the limit price is reached the order may not be executed since the limit order has low priority compare to other types of orders. The limit order is by far the most used order traded in a market place.

### *Stop-loss orders*

Another type of order is the *stop-loss order*. This order has a trigger price where the order automatically converts into a market order and is put on the market. First the order is waiting for the stock price to reach the trigger price; then, when the trigger price is reached, the order is converted to a market order and is executed to the best current stock price. There also exist orders where the order is converted into a limit order when the trigger price is reached. Here the actor has to set two prices; first the trigger price and then the limit price. Just like a normal limit order the stop-loss limit order may not be executed if the limit price never is reached.

### *Time condition orders*

Additional to the price limit orders listed above; there are orders where the validity time also is set. Validity time orders could be orders that are valid for a certain time e.g. *good-for-a-day order* and *valid-to-cancel order*. Other types are orders that have to be executed at a certain time e.g. *market-on-opening orders* and *market-on-close order*.

### *Quantity condition orders*

There also exist conditions for the order quantity. Such an order is the *fill-and-kill order (FaK order)*. The actor sets the price and a quantity of how many stocks the order concerns. Then the order is matched for the largest quantity possible. If not the entire order size is fulfilled the rest of the order is canceled but the matched quantity is executed. A close lying, but different, order is the *fill-or-kill order (FoK order)*. This type of order is only executed if the entire order size can be matched. If only a small part of the order can be fulfilled the whole order is canceled and no trade is made.

### *Iceberg orders*

Finally, there are some orders dealing with the quantity transparency. A normal limit order has two different volumes; the order volume and the open volume. If the trader does not want to show how much he wants to buy, he can for example write

an open volume of 10 shares and a volume of 100 shares. This means that from the outside, only the 10 open quantity is seen and nothing more. But if the order could be matched against a quantity of 30, three of these open quantity batches are matched and a new 10 quantity batch is put in the order book waiting for a new matching order. [1]

## **2.2 Software Testing**

Since this thesis concerns automated software testing, a theoretical part is presented about the foundation and the current knowledge of automated tests. However, to get the reader more familiar with the software testing environment, other types of tests are also presented, nevertheless, not as thoroughly as automated tests.

### **2.2.1 Background**

Tests are a very important part of software development and crucial when a system with a high reliability is constructed. Many times, the system is not under any circumstances allowed to fail or crash. Imaging for example a space shuttle that uses an automatically navigation system. With such a system, no faults are accepted. At crucial parts like take off or landing, the system has to work perfectly. Therefore, a large number of tests need to be executed on the system before the system is put into use. However, testing can never prove the absence of faults in the system. In order to do this, all types of situations have to be tested for; and such a test is not feasible. The important aspects of testing concerned in this thesis are; reliability, usability, efficiency, functionality, maintainability and portability. It is obvious that only one single testing method could not cover all these aspects and the whole test outcome space. Instead, it is just possible to cover a part of all imaginable situations and that is where the theory of software testing starts.

Depending on what kind of system that is concerned, it must be decided for how intensively and thoroughly it should be tested. This decision must be made based on what risk the system contains and what a failure would cost, both economically and healthily. E.g. if a your personal home page fail and break down no one will get hurt or affected economically, probably. However, if a stock exchange crashes, the economical losses are millions of dollars and the consequences fatal. Or, if an airplane is about to land and the landing gear does not work, the consequences could be deadly. Thus, the level of testing must be decided related to the system concerned.

There exist many different methods and procedures for software testing. Every method focuses on a certain aspect of the system and none of them is able to cover the whole spectrum. They all have their strengths and weaknesses in finding faults. Therefore, a combination of different test techniques is always necessary to detect failures with different causes. Some techniques are reviewed in this thesis. [3] [11]

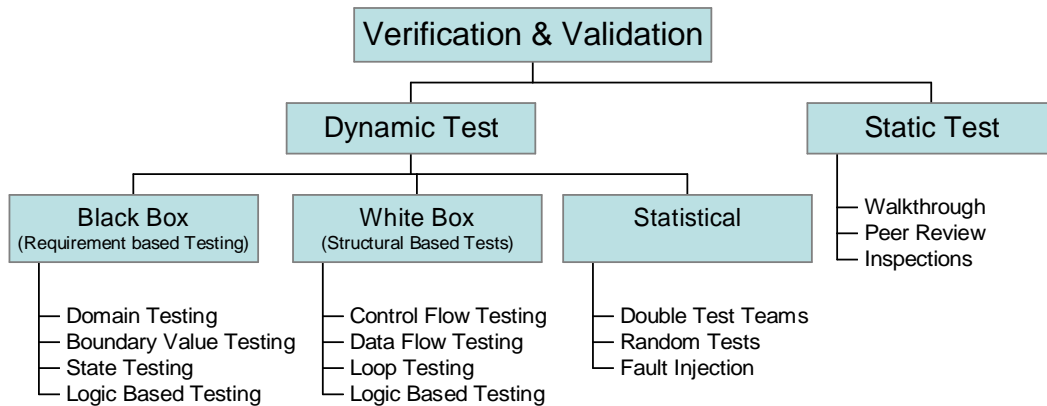


Figure 1: Overview of test methods [3]

### 2.2.2 Static testing

Before test scripts are written it could be of great importance to examine the system and its specifications. This is called *static testing*. The main goal of a static test is to find defects and faults from the requirements and specifications of the system. The test is performed through reviewing documents and code. The three types in the static test branch in *figure 1* are different levels of ambition where an inspection is more serious than a walkthrough. Contrary to dynamic tests, the test object is not executed with test data, but only viewed and examined for logical errors and misleading commands. Results of the static testing procedure are used as improvements for the development process and because faults are found at an early stage, before they have started to infect the whole system, the forthcoming developing process works smoother and faster.

The static testing also has some other advantages. The tests are done using teams of people all learning a lot when reviewing the documents and finding faults. The comprehension of others work also increases, since the walkthrough or inspection typically is done of someone else than the person that has written the code. [5]

### 2.2.3 Dynamic testing

A *dynamic test* executes the test object during the test. When a static test only views the code and documents looking for logical or operational faults, the dynamic test needs input that is used to execute the test object. Dynamic testing can be divided into three groups; black box, white box and statistical tests.

### *Exploratory testing*

When no test design exists, or the test design is created at the same time as the test is executed, an exploratory test, or a fault guessing test, is performed. The approach is mainly based on intuition and experience of the tester and the test explores the possible functions, tasks and elements of the system. Exploratory tests are executed when hardly any information of the system features are known. That is, the exploratory test is in some sense a black box test, but since there are no rules or definitions for the test procedures, it is hard to place it in only one branch in *figure 1* above. Tests are designed continuously through the test process. When one test has been designed and executed, new knowledge has been gathered in order to form another test. In this way knowledge of the object under test is collected and it becomes clearer how the system works and which tests are necessary. However, exploratory test are non repeatable, that is, failures that are found cannot be recreated in the exact same way again since it does not exist any record of the test performed, a feature which often is pleasurable in fault corrections. [3]

### *Black box testing*

When a *black box test* is executed, the system is looked at as a black box where no concern is taken to what is inside the box. The test is focused on the functionality of the system and does not explicitly use knowledge of the internal structure of the system. The test cases used within the black box technique are derived from the specifications of the system. A complete test would be all possible input combinations; however, this is of course unrealistic since an enormous amount of tests then is needed. Instead, the test design needs to contain a reasonable selection of important test cases that covers most of the function space. To select these tests all possible cases are sorted into equivalence classes, where a class represents many different combination of the same function. Besides equivalence classes for the correct input, classes for incorrect input need to be tested as well.

Another important aspect that is concerned in the black box technique is the boundaries. Faults often appear at the boundaries of input values because boundaries are often not defined clearly or developers misunderstand them. Boundary value tests check the border of the equivalence classes defined in the previous step. An effective boundary test examines the boundary value and both nearest adjacent, one inside and one outside the boundary.

Any of the branches under dynamic testing could be automated; however, in this thesis focus is put on black box tests. [6]

### *White box testing*

Whereas the black box test does not explicitly look at the code produced, the *white box testing* is the contrast where the code structure itself is examined. In a white box tests all individual paths and loops in the code structure are tested for logical incorrectness or boundary conditions. The white box technique is an important tool

for the programmers within the developing process, where logical conditions and loops are tested. [6]

### *Statistical testing*

*Statistical tests* are concerned with functionality for the system as whole more than functionality of the different parts of system. The main purpose of the statistical tests may not be to find faults, but get an overview of the quality of the system. That is, get a statistical value of the fault rate per execution or mean run time before failure. One statistical test is the *random test*. The idea of a random test is that input is automatically and random generated from a predefined statistical distribution, put in to the system and then compared with an expected output. Since the input is randomly generated, the boundaries of the selected data could be tested without restricting it to any specific values. A so called *Oracle* is used to examine the output. An Oracle is something who knows what is expected from a certain input. For an exchange system, typically, many traders are simulated and random orders with an arbitrary size and price are entered. At the same time, an Oracle for each trader, or each group of traders, is listening to the traders knowing what is expected of the output. If the true output is not consistent with what the Oracle expects, something has gone wrong and a fault has been discovered.

Random tests are mainly used for investigating the reliability, usability and efficiency of a system. The explicit functional features are left for the black box part of testing since it is more straight on than a random test. However, when coming to reliability and efficiency, a heavy load is needed to test the system. Such an arbitrary load could be simulated randomly, thus cover a big part of the testing space. Usability is also tested through random tests since many different types of features and operations are used from the randomized input and thus examined by the Oracle.

Statistical tests are a very important part of the system testing which increases the coverage and the reliability of the system. However, it is of great importance to state that random tests could never replace automated and manual testing. The randomized part is only one of other important aspects that have to be used in order to get a necessary coverage of the test. [6]

## **2.3 Manual tests**

All testing methods in any of the branches of dynamic tests in *figure 1* could be performed manually. Manual tests can be used to test all types of systems. The system is manually fed with input that is generated by the user and then an output is examined and compared to the expected output. Normally a GUI (Graphical User Interface) is used. The advantage with manual tests is its reachability. If a certain scenario is wanted, it is easy to test it manually just by simulating the situation and examine the output. However, the drawback is to reach a high coverage of the test, and also, significant human experience and time is needed. The tests need to be

designed of someone who really knows the system and also knows were the fall pits are and how they could be found. However, once the tests are designed, they could easily be executed by someone who is relatively novice to the system. Manual tests are very important in a development process, and even though other tests stand for most of the coverage, manual test could never be excluded or substituted. [5]

## 2.4 Test automation



Figure 2: Process of testing [7]

When identifying a test process in order to automate it, the different steps in *figure 2* have to be analyzed. Which parts can be automated and which cannot? There are white box tests where a testing tool automatically generates test cases from the code under test, that is, then the test is executed and checked, that is, all steps in *figure 2* are automated. Also, there are tests where the test case is auto build after identification and design and then automatically executed and checked. The tools examined in this thesis treat how the execution part and the check part could be automated. The tools that are under examination here treat black box functional test.

An automated test is a script based input to a system which creates a response that is compared to what is expected from the system. The input data could be random generated or explicit selected of the test author. The validation is done through either a comparison with; a predefined output, an examined and approved output or an output from an Oracle.

The purpose of automating functional testing is to replace repetitive and often error-prone manual testing and also decrease the time spent on defining and writing cases for every single project. An automated test tool makes it possible to verify functionality across different type of systems. Whereas a statistical test primary is focused on reliability, efficiency and usability of the system and a structural test's main coverage is on the code structure and its functionality, an automated test is mainly concerned with the functionality, maintainability and portability of the system. [7] [12]

### 2.4.1 Unit testing

JUnit tests make it possible for a development process to check the basic functions and operations of the code written as the process evolves. The unit testing framework used in this thesis for systems programmed in Java is called JUnit. A

deeper description of JUnit is presented in *section 5.2*. Just as all automation processes, the early parts of building an automated test demands time and skills. This is the case for JUnit tests as well. But once the test has been identified, designed and built it will be very useful since it only takes a few second to run and can be executed at an early stage of the development process. The real environment is not needed in order to perform a JUnit test, but a mock up is used which simulates the real environment an all its features. In this way tests could easily be launched in order to test a certain unit of the system.

During a development cycle, code is written by different individuals, and features within the code depend on other parts of the system. This leads to a process where errors are a common consequence. If a JUnit testing framework exists, the tests could be launched at any time of the development cycle, finding faults in the system, which has been merged from different development traces in the project. [7] [8]

### **2.4.2 Abstraction**

When talking about abstraction within automated software testing, one refers at the level of which the input is given or the output is extracted. E.g. if a transaction insert is to be done, the actual input for the system under test could be specified at another level of the test script than where the insertion to the system under test is actually made. That is, the insert method that is used in the test script is not the method that makes the explicit insertion. The method called in the test script calls another method, which calls even another, which then calls the actual insert method used to insert the transaction into the accurate part of the system. If test are kept on a high level of abstraction, that is, the explicit insertions are not made within the test case, automated tests demand less effort to maintain and are easier moved from one system to another.

An important part of test automation is to create different levels of abstraction for the testing tool. Abstraction levels are used to enable the test case to be run on many different system configurations, to facilitate for the user to define test cases, but also, to make the tool easy to support. If, for example, a simple trade match test is made where a trade is executed for two matching limit orders, it is unnecessary and tedious to define all required fields for a transaction. Instead, the level of abstraction is such that the fields which will not be used in the test are automatically field in order to allow the test to be executed. For each commonly used transaction there is an abstraction mechanism that is used by the test writer to define the transaction on a higher level of abstraction than the underlying insertion action. In the same way, abstraction is used when a test case is verified. When a simple limit order match has generated one trade, the validation method does not need to check all the fields specified in the transaction. A validation on a higher level of abstraction could select e.g. the price, volume and active users in the trade and verify them to what is expected. If the test case needs to be verified in all its field for every test, it leads to superfluous work and large maintenance efforts. Thus, if an abstraction mechanism is used also in the verification process the appearance of the output could change

but the test could still remain and work as it has done before. If a small change, e.g. a field is added to a transaction, all test cases do not need to be changed but only the method in a lower layer of abstraction that calls the output. This decreases the maintenance cost heavily and makes the JUnit framework dynamic to work with.

*Fester and Graham*, [7], are describing the aspects of robustness and sensitiveness. They refer to the validation of automated tests, where in the sensitive case, all of the output produced of the test is examined and verified, whereas in the robust case, only a small part of the output is validated. E.g. if the output from a test case is a string of a known size, the sensitive case has to verify all characters in the string and all of them have to correspond to what is expected. However if the case used a robust validation approach, only a small part of the string is compared to what is expected, that is, the other part of the string is not concerned.

In *figure 3*, a brief picture describes the aspects of robustness and sensitiveness, from a *Fewster and Graham* point of view. The picture presents two different test verification approaches, where the left one uses a robust testing procedure and the right a sensitive one. The big circles correspond to the coverage that is wanted for a test suit. The small circles within the big ones are equivalent to the coverage of each test within the test suit. As can be seen in *figure 3*, the same area of the output space is covered for both the suits, however, in the right circle test cases are overlapping each other and the same verification is made several times. For the left circle, each test focuses on one certain aspect and only a small overlap is made. The triangular represents a fault in the system and the mission for the test cases is to find the fault. As can be seen, both suits find the fault, the left circle capture the fault with one test and the right with several. [8]

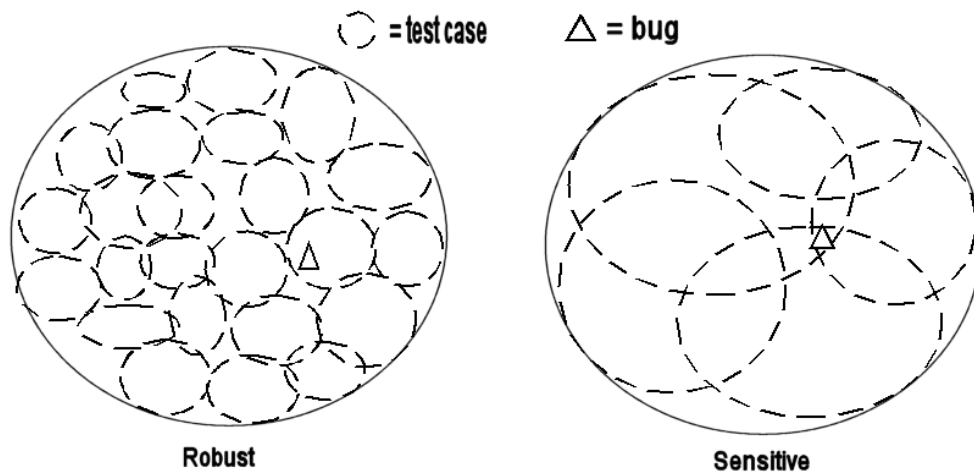


Figure 3: Robust and sensitive testing

### 2.4.3 Predicate structure

A predicate is a predefined property for the test that is essential in order to make the tests executable. E.g. if the test suit needs an order book which uses the currency USD in order to be able to execute the test, then a predicate states that a order book which has a USD currency is wanted and the test suit makes sure such a book is chosen. If an order book is arbitrary chosen, the case could be such that an incorrect order book is chosen. This may cause the test case to fail of irrelevant reasons and a process of finding the fault is processed. The alternative to use a predicate structure is to specify the stimuli manually for each test case. If instead an order book is got where its characteristics correspond to what is predefined through the predicates, the test could be executed without any problems. The predicate states e.g. what kind of order book that is wanted and such an order book is returned and the test could be run. However, if no order book is found with the correct characteristics, the test is skipped and the next test in line is run. [7] [8] [14]

## 2.5 TRADExpress Trading Engine

Market places today are competitive and demand top-performing, low-latency trading engines that easily could support the in-house-platform. The engine needs to be quickly adapted to the changing needs and unique market models. The solution for these demands by Cinnober is called TRADExpress. The entire trading engine could be viewed as a pyramid with three layers, *figure 4*. [9]

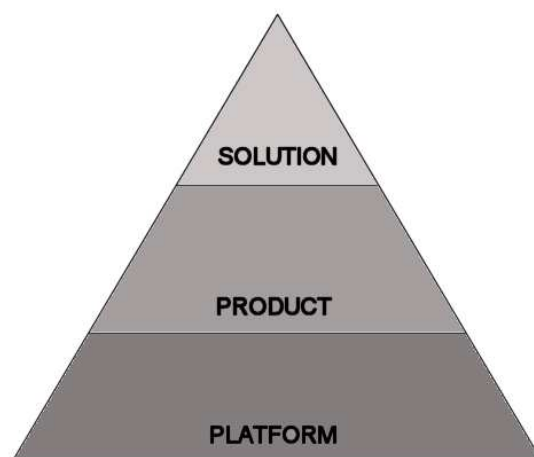


Figure 4: TRADExpress Trading Engine three layer architecture [9]

### 2.5.1 Platform layer

All Cinnober products are based on the TRADExpress platform which offers the technical infrastructure needed in a high performance transaction system. The main

tasks for the platform is to deal with; high reliability, low latency, good scalability, good operability, high flexibility, and state of the art messaging. These tasks are fulfilled using different techniques which are company confidential and can hence not be published. [9]

### **2.5.2 Product layer**

The product layer is divided into a few different types, depending on the market place it should work for. If the customer is an investment bank or a stock exchange, the product layer looks different. The platform itself needs a modified product layer in order to work according to the specified needs from the customer. The product layer provides the application components, the data models and the business libraries that is specific exactly for the concerned customer. The components are constructed to be extensible and replaceable in order to get a high flexibility in supporting refined market models. [9]

### **2.5.3 Solution layer**

The solution layer is the top layer in the pyramid and it is the explicit solution for the customer concerned. The solution layer is an extension of the other layers and makes the product unique for every customer. Since customers use different hardware, operative systems and other external connected systems have gateways that only allow a certain type of protocol, modifications are needed in order to make the whole solution fit to the existing solution system. [9]

### **2.5.4 The trading engine**

The trading engine, that is, the pyramid above, *figure 4*, consists of a number of servers operating together. *Figure 5* below shows the architecture of the trading engine. During a trading day there are hundreds of thousands of orders in the engine. If the engine crashes all entered orders would be lost and the exchange would suffer massive losses and problems. In order to come to terms with this a redundant system is used where a secondary site could take over after a crash of the primary site. The different parts and the two sites can be viewed in *figure 5*. [10]

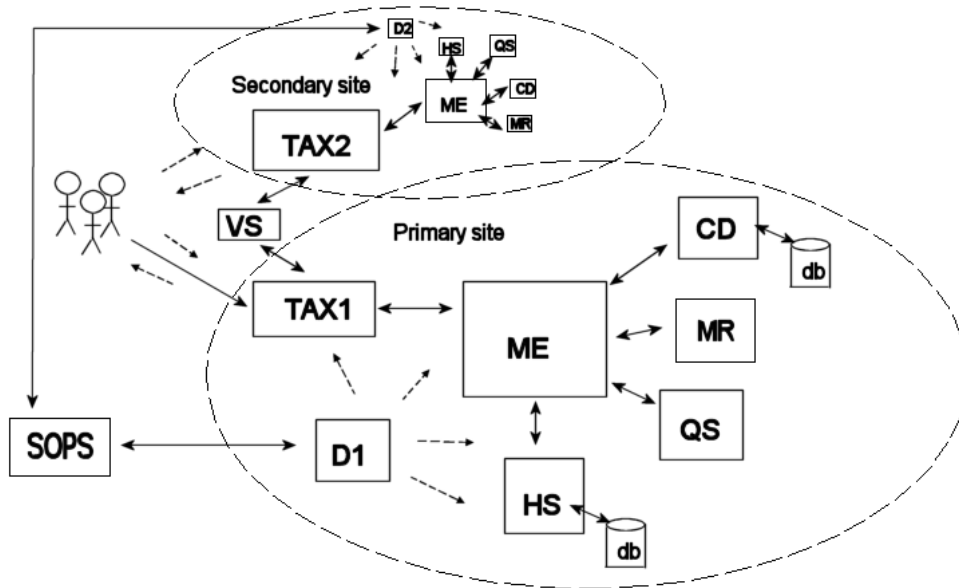


Figure 5: Trading Engine main servers, simplified setup

#### *Trading Application Multiplexor (TAX)*

Trading participants connect to a TAX gateway. The TAX works as a gateway for both incoming and outgoing traffic. The TAX gets an order from a user and sends it to the ME where all order books and orders are maintained.

#### *Matching Engine (ME)*

The ME is the core of the trade engine and it is in the ME where all deals are made, that is, orders are matched.

#### *Common Data (CD)*

Connected to the ME is a CD. In the CD, things like; user information, orders, order books and trading schedules are stored. The CD uses a database to persist settings.

#### *Query Server (QS)*

Connected to the ME is also a QS. The QS maintains a copy of the active orders and order books from the ME. The main task for the QS is to offload the ME from serving large queries.

### *History Server (HS)*

The HS keeps all orders and trades overnight and serves queries for historical information. The HS uses a data base and is not part of the transaction critical part.

### *Additional components*

A Vote Server (VS) is used to select which server is the primary and standby respectively.

There is a daemon on each site and this daemon runs TRADEExpress, that is, the daemon handles all servers belonging to it.

Management Repository (MR) maintains system operations data such as status event and statistics.

System Operation Application (SOPS) is used by system operators to monitor TRADEExpress. The SOPS is connected to the servers through the daemon on each site.

### 3 The frameworks

This thesis concerns three different automated testing frameworks. These three are presented in this section

#### 3.1 EMAPI

EMAPI, Extending Messaging Application Programming Interface, is the name of the external protocol used in TRADExpress. It is intended for communication between the Trading System and external clients, e.g. trading clients, market operations clients and customer system such as clearing systems. The EMAPI protocol is an XML-based protocol using TCP/IP as its transport layer.

Clients can connect directly on a TCP socket and send or receive XML messages. EMAPI uses a helper code (“API code”), which is a convenient way of handling the EMAPI communication with TRADExpress. When using the EMAPI code, the client does not need to create an XML message, but can just instantiate a Java object and let the helper code handle the conversion to XML and the TCP sessions. The helper code also handles the XML parsing for received messages.

By implementing and creating a testing framework called EMAPI, a more comprehensive picture of the predicate structure and the abstraction mechanism was achieved. Since the framework is working directly against the EMAPI protocol and not via any type of client, transactions are sent directly into the system, limiting the probability of defaults caused by the client.

The framework design was such that the orders were sent into a local TRADExpress system, containing all parts described in 3.5.4. Orders were entered and the output was verified through the broadcasts leaving the TAX.

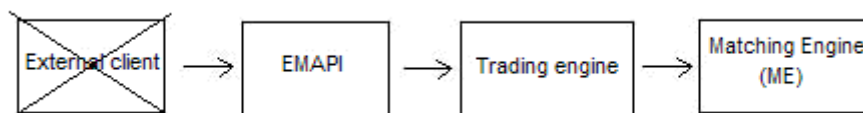


Figure 6: Scheme over the EMAPI framework. EMAPI transactions are executed directly on the system, where the ME is under test.

The framework used a predicate structure for getting a valid order book and a user with the rights to execute orders. Conditions of interest were specified and then a random order book and a user were got through get-methods. See *example 1* below. When the methods were used, the test case could be certain of having an order book

and a user that had the desired characteristics in order to run the test. If no order book or user with the expected properties was found, the test was skipped.

In order to increase the level of abstraction, convenience methods were used. A convenience method is a method that is only there to facilitate the usage of the framework. The method is not necessary for the execution of an action, e.g. sending a transaction. However, the method does not only increase the level of abstraction, it also makes the test easier to read since the name of the method often describes what it is happening in the test. E.g. in order to insert a bid order, the EMAPI protocol uses an `orderInsertRequest`. The protocol requests a mandatory attribute; `isBid`, however, this boolean attribute could be set to false for a convenience method called `enterAskOrder()`. [14]

Tests were written with defined predicates and convenience methods used. The tests were robust tests of the type dynamic comparison where no validation of parameters and states could be checked within the order execution. All validation was made after the orders had been matched. [7]

The EMAPI framework developed is presented in *example 1*. The example contains two classes. The first is the test class where the tests are specified and the output validated. The test class starts by getting an order book and a user that have the properties described in the predicates. The properties defined in this example are for the order book; currency USD, properties not valid for combination orders, tick size 0.01 USD (that is, the smallest price change allowed for an order) and round lot size of 100 shares, and for the user; a trader user. The first class executes the test after calling the methods `getRandomOb()` and `getRandomUser()`. `getRandomOb()` is specified in the second class.

### Example 1

```
public void Test1() throws InterruptedException {

    LinkedList<Predicate> tPredicateListOb = new LinkedList<Predicate>();
    LinkedList<Predicate> tPredicateListUser =
        new LinkedList<Predicate>();

    tPredicateListOb.add(new PredCurrency("USD"));
    tPredicateListOb.add(new PredIsCombOB(false));
    tPredicateListOb.add(new PredTickSize(0,01));
    tPredicateListOb.add(new PredRoundLot(100));
    tPredicateListUser.add(new PredProperties("trader"));

    int tMyOb = getRandomOb(tPredicateListOb);
    String trader1 = getRandomUser(tPredicateListUser);
    String trader2 = getRandomUser(tPredicateListUser);

    if (tMyOb != 0 || trader1 != null) {

        //users are logged on and orders are entered
        ...
        tTrader1.enterAskOrder(tMyOb, 80, 18);
        tTrader2.enterBidOrder(tMyOb, 100, 18);
        ...
        boolean check1 = tTrader1.checkTrade(tMyOb, 80, 18);
        testResults(check1)
    } else
        System.out.println("test skipped since predicate(s) not
            fulfilled");
}

public int getRandomOb(LinkedList <Predicate> PredicateList) {

    boolean tState = false;
    for(Orderbook tBook : mOrderBookList){
        for(Predicate tPred : tPredicateList) {
            if(tPred.check(tBook)) {
                tState=true;
            }else {
                tState=false;
                break;
            }
        }
        if(tState) {
            mValidOrderbooksList.add(tBook);
        }
    }
    try {
        Random tRandom = new Random();
        Integer tRandomElement = new Integer((int)(mValidOrderbooksList
            .size()*tRandom.nextDouble()));
        return mValidOrderbooksList.get(tRandomElement).ObName;
    }
    catch(Exception e) {
        System.out.println("No matching orderbook was found");
        return 0;
    }
}
```

## 3.2 JUnit

The JUnit framework that was used in this thesis was a model of the Matching Engine (ME). Instead of setting up the whole system to perform the tests, as was made for the EMAPI framework, the JUnit framework creates a mock up that gets under the skin of the ME and simulates the environment of the ME, and thus is able to verify all its matching logics.

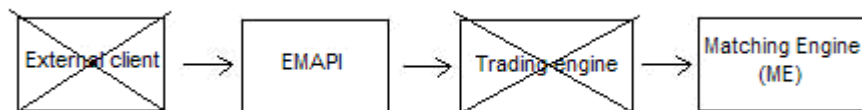


Figure 7: Scheme over the JUnit framework. The ME is under test through EMAPI transactions executed on a mock up of the ME.

JUnit is an easily handled tool if the user is familiar with Java. The framework works best if suitable convenience methods exist, as these could be used instead of doing the same tedious insert several times in the same way. It is also optimal if these convenience methods could include e.g. an insert verification, where a validation of correct insert is made. This saves time and makes the code more clear and strict.

A dynamic comparison is used where the states of the test are checked through the test. The tests are quite specified, that is of the robust type according to *Fewster and Graham* [7] and do not concern checking other fields than the one that the test concerns. The test does not validate all available fields in the test but only the ones specified in the test description, that is the requirements.

In *example 2* the JUnit framework is exemplified. The example illustrates the increased level of abstraction when a convenience method is used in order to insert a hidden volume order. All attributes inserted in the second method are just set there, and not in all inserts within the test. *Example 2* is just a part of the abstraction used in the test cases. The same approach is used to assert a certain condition. If a trade of a residing bid order is to be verified, the method `verifyBidPrivateTrade()` is used. The method calls the method `verifyPrivateTrade()`, which then calls the method `findPrivateTrade()`. In this way, only the last method, on a low level of abstraction needs to be updated if any changes are made in the trade event protocol.

## Example 2

```
public static OrderInsertReq createHiddenBidOrderInsertReq(long
                                                            pOrderBookId,
                                                            long pVolume,
                                                            long pVolumeOpen,
                                                            long pPrice) {

    return (createOrderInsertReq(pOrderBookId,
                                  true,
                                  pVolume,
                                  pVolumeOpen,
                                  pPrice,
                                  VALIDITY_PERIOD.GOOD_FOR_DAY));
}

public static OrderInsertReq createOrderInsertReq(long pOrderBookId,
                                                  boolean pIsBid,
                                                  long pVolume,
                                                  long pVolumeOpen,
                                                  long pPrice,
                                                  int pValidityPeriod) {

    OrderInsertReq tOrder = new OrderInsertReq();
    tOrder.activeUserId = new String("User1");
    tOrder.messageReference = new String("User1 inserts order");
    tOrder.source = (byte) 1;
    tOrder.partition = new Integer(1);
    tOrder.addTimestamp(BdxMessage.Station.SENDER_SEND);
    tOrder.ownerData = new OwnerData();
    tOrder.ownerData.member = "Member1";
    tOrder.ownerData.user = "User1";
    tOrder.orderDataIn = new OrderDataIn();
    tOrder.orderDataIn.isBid = new Boolean(pIsBid);
    tOrder.orderDataIn.orderBook = new Long(pOrderBookId);
    tOrder.orderDataIn.orderDataInExt = new OrderDataInExt();
    tOrder.orderDataInPrivate = new OrderDataInPrivate();
    tOrder.orderDataInPrivate.orderQty = new Long(pVolume);
    tOrder.orderDataInPrivate.orderQtyOpen = new Long(pVolumeOpen);
    tOrder.orderDataInPrivate.infoText = "4711";
    tOrder.orderDataInPrivate.isSuspended = new Boolean(false);
    tOrder.orderDataInPrivate.orderQtyOpen = new Long(pVolume);
    tOrder.orderDataInPrivate.validityPeriod = new
        Integer(pValidityPeriod);
    tOrder.orderDataInPrivate.orderDataInPrivateExt = new
        OrderDataInPrivateExt();
    tOrder.orderDataIn.price = new Long(pPrice);
    tOrder.status = getDummyTapStatusCodeOK();

    return tOrder;
}
```

### 3.3 Grace

Grace is an automated testing framework developed and used by Cinnober. Grace was developed a few years ago and is used in many of the Cinnober projects. Grace is a Java tool based on xml-scripts. The test cases are defined in xml and java code is auto generated. The methods generated from the xml-scripts have to be implemented manually in order to make the framework work properly. While EMAPI executes directly to the TAX without a client and JUnit is focused on the ME, Grace uses a client that is connected to the TAX and communicates to the system through it. Grace allows a high level of abstraction for the tests, and also uses a predicate structure in order to set up the test environment. Before a test is executed, predicates for the test suit are defined in order to supply the tests with accurate input. The preconditions provide the test suit with accurate broadcast flow, users and order books. The conditions setup is base on predicates defined for the test. When all preconditions for the test are fulfilled, the test scripts are executed. The test contains what type of action that is wanted and all its parameters. In order to verify the test, the output is manually overviewed and confirmed after the first execution. Then, when the test is run again, the output from the first run is saved as a key, which the new output could be compared to. If the output is the same, the test passes. If it is not, the test fails. [13]

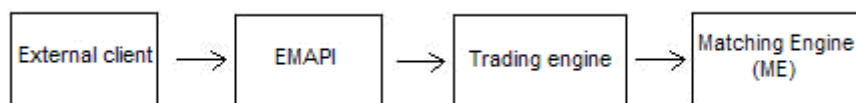


Figure 8: Scheme over the Grace framework. An external client (Grace) executes EMAPI transactions on the trading engine, where the ME is under test.

Since Grace enables a very high level of abstraction, the validation can be made through looking on either all of the field or just on the fields wanted from the output protocol. With the *xpath* function, unwanted fields can be hidden and only the ones wanted compared and verified to the key output. Without the *xpath* function the testing framework is what *Fewster and Graham* [7] wrote, of a very sensitive type. All the information in the output is validated and a small insignificant difference between the actual and the expected output causes the test to fail. However, with the *xpath* function, the validation could be of a strictly robust type where only the output that the test concerns is validated.

As for the other frameworks, an example is presented in order to make the reader more familiar to the framework.

In *example 3*, the private flow of the match for an iceberg order and a limit order is verified. The hide-tags at the end of the script hide all flows and their attributes except the wanted ones; volume, price, order event and order priority. The only fields examined in the output key file are those listed above.

*Script 1* in *example 3* is just a part of the test script used. This *xml*-script is run through a client called Grace. The code is generated after the IceBergOrderInset action is defined. For example, the tag IceBergOrderInsert is constructed in an *xml*-script, *script 2*, where all its parameters are defined. Then, the script builds a java interface that needs to be implemented manually in order to use the method.

Grace is a tool that demands quite an effort and knowledge to handle. The tool demands certain programming skills, knowledge of *xml*-programming and has a fairly cumbersome starting up phase before it gets easy and useful. But, as the environment looks more familiar and the test stage with its predicates and inputs are set up, Grace-tests are written fairly easy and fast. Whereas certain specific fields could be pointed out in EMAPI and JUnit, Grace has the ability to look at both the whole protocol at the same time, but also pinpoint a certain field to verify.

### Example 3

#### Script 1

```
<TestCase id="IceBergOrderBasic0004" desc="IceBergOrderInsert, partial
match, Private" type="positive" reqRef="Order">
  <StageRef ref="subscribePrivateOrder">
    <Param name="userType" value="SUPER_TRADER"/>
  </StageRef>
  <PROTSession ref="user1">
    <IceBergOrderInsert contractRef="contract1" isBuy="false"
price="85" volume="100" volumeOpen="10" orderId="order1"/>
  </PROTSession>
  <PROTSession ref="user2">
    <OrderInsert orderId="order2" isBuy="true" volume="25" price="85"
contractRef="contract1"/>
  </PROTSession>
  <Check fileName="IceBergOrder">
    <Status name="EmapILog" enabled="true">
      <Hide id="all except volume, price, eventType and priority"
xpath="/Output/Sessions/Session/Flow/OrderEventPrivate/
attribute::*[not(name()='orderQty' or name()='price' or
name()='eventType' or name()='priority')]" />
      <Hide id="InFlow" xpath="/Output/Sessions/Session/
Flow[@key='In']" />
    </Status>
  </Check>
</TestCase>
```

#### Script 2

```
<Action name="IceBergOrderInsert">
  <Param name="contractRef" type="Contract" required="true"/>
  <Param name="isBuy" type="Boolean" required="true"/>
  <Param name="price" type="String" required="true"/>
  <Param name="volume" type="Integer" required="true"/>
  <Param name="isActive" type="Boolean" required="false"
default="true"/>
  <Param name="validityPeriod" type="VALIDITY_PERIOD" required="false"
default="GOOD_FOR_DAY"/>
  <Param name="messageRef" type="String" required="false"/>
  <Param name="orderRef" type="String" required="false"/>
  <Param name="infoText" type="String" required="false"/>
  <Param name="volumeOpen" type="Long" required="false"/>
  <Result name="orderId" type="Order" required="true"/>
</Action>
```

## **4 METHOD**

This thesis is focused on a comparing case study concerning two different frameworks, JUnit and Grace. However, the report first looks at the existing abstraction in three different testing frameworks through implementation, experimentation and evaluation. The method can be divided into four steps, which are presented here. [15]

### **4.1 Definition**

As stated in the problem statements, this thesis studied three frameworks for automated tests, find similarities and differences between them, state pros and cons for the frameworks and evaluate the abstraction dependency for input and output validation. The abstraction is an important aspect when talking about maintenance and portability of the tests. A written automated test is useless if it has to be modified every time it should be run or every time the system under test changes.

### **4.2 Evaluation method**

The method used in this thesis was a case study where first the system as whole was examined and necessary knowledge was gathered through reading and trial and error testing. The preliminary work can be divided into three processes. The first involved getting to know the system and its features, the second concerned testing with an EMAPI framework implemented and developed by the author. The third dealt with automated tests with two different in house frameworks.

When the system had been examined, a case study was started where the level of abstraction for two frameworks was evaluated. The case study treated two different solutions developed at Cinnober and the process of moving tests from one of the solutions the other. Was the abstraction for the frameworks such that no extensive implementations and modifications were needed?

## 5 STUDY

Three different automated testing frameworks were examined, implemented and evaluated. When the frameworks felt usable and enough knowledge about the system had been gathered, a case study was executed in order to state the robustness of the abstraction.

### 5.1 Exploratory testing

In order to get a comprehensive view of the platform and TRADExpress, a trading system setup was made. To understand and construct automated testing tools, knowledge of the system itself was crucial. To get such a basic knowledge, a TRADExpress system was built on a computer that could be used and examined through trial and error methods. The system contained all real components explained in *section 2.3.4*, which were:

Daemon (D1)

Vote Server (VS)

Common Data (CD1)

Matching Engine (Primary ME1)

Matching Engine (Secondary ME1S)

History Server (HS)

Manage Repository (MR)

Trading Application Multiplexor (TAX1)

Trading Application Multiplexor (TAX2)

System Operation Application (SOPS)

The product used was developed for internal matching of stocks and was intended for an investment bank.

The point of this part of the thesis was to create a shallow foundation of the TRADExpress system, the EMAPI protocol used within the system, as the features and operations that could be used to handle and navigate the TRADExpress system.

## 5.2 The case study

At Cinnober there are a lot of parallel projects going on. Each Cinnober customer is its own project and there is always extensive development in progress. The projects are derived from the same platform and each project has its own customer specific solution. At the time for this thesis, there was a process where a solution was developed at Cinnober, a solution that was to be a foundation of many forthcoming projects. The thought was that if this solution was well specified and tested, then the modification for each future project would demand less effort. To benchmark the level of abstraction of JUnit and Grace used in this thesis, a case study was conducted. The EMAPI framework was never meant to be included in the case study. This, due to the obvious reason that the development of the EMAPI framework was on a complete different level than the development of JUnit and Grace.

### 5.2.1 Background

Two different project solutions were examined. The first one was the solution to a London investment bank; let us call it the *Bank solution*. The solution was built from the head branch of the developing trace at Cinnober and was focused on internal stock matching. The other project that was examined is the product project developed at Cinnober; this project is called the *Product*. The product was modified in a way that would make it more dynamic to all other projects, that is, it is extended compared to the Bank solution. The specific changes that were noticed within this experiment were some business logics and the attributes in the protocol that had been changed in the Product compared to the Bank solution. This change from one project to another affected the automated tests that were run on the solutions. The experiment was focused on the robustness of the predicate structure. Does it hold for the two frameworks? That is, were the changes and modifications extensive in order to move the tests from the Bank solution to the Product?

### 5.2.2 Execution

The case study concerned JUnit and Grace tests. The JUnit tests were a 4600 lines suit containing 61 tests that were focused on order updates and were examining the matching engine. The Grace tests were separated suits, all together an amount of 1500 lines with 55 tests included. The Grace test did focus on order action e.g. order insert, order update and order cancel. The tests were examined, executed and validated for the Bank solution. The two different testing frameworks used at Cinnober were utilized and compared, JUnit and Grace. When this was done, the tests were adapted to the Product and implemented. The modification and changes necessary for the Product in order to make the test run smoothly were performed.

### **5.2.3 Evaluation**

Differences between the projects were stated and explained. The effort of modifying the Product was estimated and evaluated, as conclusions and remarks were noticed and stressed.

Furthermore, in order to confirm or reject the pros and cons noted from the experiment, interviews were held with 'experts' at Cinnober. Developers and testers that during an extensive period have worked with test and development gave their perspective to the frameworks examined.

## 6 RESULTS

This section presents what was noticed when working with the JUnit and Grace testing frameworks. Conclusions are also drawn from the case study where the robustness of the abstraction of two frameworks was tested.

### 6.1 The case study

The case study started with an examination of JUnit for both projects. Test were written, existing test were examined and the test suit were evaluated in both the projects. Then, the same thing was tried for Grace, however, some problems were noted and the experimental plan had to be changed. The procedure is explained below.

#### 6.1.1 JUnit

When all the test cases first were transferred into the Product from the Bank Solution, lots of compilation errors were presented, more than one per test. Many of the errors concerned the same thing but were displayed in many replicas. The errors could be divided into three bullets.

- Variable type change
- Attribute definition change
- Class import change

Some member variables had been changed to class variables. This cannot be viewed as a maintenance problem since it is a change of what is classified as a member variable and a class variable that caused the problem. This was due to earlier negligent development work and should not be taken into account here.

The attribute definition change is the most crucial change between the two projects and also becomes the essential part of this study. Since the Product is meant to serve a wider perspective of business functions, the appearance of some attributes in the protocol had been changed. When the trigger price of a stop-loss order was to be set, the Bank solution called `STOP_LOSS_CONDITION` when the Product called `TRIGGER_CONDITION`. This is a change in protocol which demanded modification in the `createStopLossOrderInsertReq()` and `createStopLossHiddenVolumeOrderInsert()`. Changed were also demanded for all trigger condition attributes within the test cases. That is, the attribute had to be changed in two places.

The same happened to the validity period in the protocol. In the Bank solution the attribute was called VALIDITY\_PERIOD, whereas the Product called the field VALID\_TILL. The attribute validity period had to be changed in the createOrderInsertReq() but also in the creation of the order inserts in the test case. That is, even this attribute had to be changed in two places.

The import change errors were due to the change of protocol attributes in the bullet above.

### *Modifications*

The modification performed in order to make the tests work for the Product solution environment were focused on the change of some attributes in the order insert protocol. The reason why the protocol structure had been changed is that the protocol with the new look is more dynamic to whatever orders that are implemented, that is, the solution is more dynamic for new development and changes of business functionality.

What could be noted from the implementation of the tests in the Product framework is that modifications due to the attribute name change in the protocol were extensive. The methods used to execute inserts and updates had to be modified, which was expected and accepted. But, the test scripts them self had to be changed in all its insert methods as well. This led to quite an effort when all inserts had to be modified and since every single test case had at least one order insert the number of change was linear proportional to the number of tests. The reason why all inserts also had to be modified is that the insert methods were written on a too low level of abstraction. The attribute, validity period, was specified two times with the constants predefined. In example 4, the situation is shown and a suggestion of improvement is proposed.

When the methods of insert are written in the way that is exemplified in *case 1* of *example 4*, and as it was done for the bank solution. The level of abstraction is low and a change to the protocol attributes demands modifications in all tests since they all use the order insert method. If, on the other hand, tests are written as is shown in *case 2*, the abstraction is increased. The method used in the test case is enterBidLimitOrder() and the filed validity period is filled once in the createOrderInsertReq() method. The tests use enterBidLimitOrder() and are not affected by the changed attribute. If this had been the implementation in the Bank solution, the change of the field validity period would have been needed only once in order to make the tests work.

## Example 4

### Case 1

```
-----  
(within test case)  
OrderInsertReq tReq = BeanTestUtil.createOrderInsertReq(101, // ObId,  
    true, // Is bid  
    100, // Volume  
    695, // Price  
    VALID_TILL.GOOD_TILL_DAY);  
-----  
  
public static OrderInsertReq createOrderInsertReq(long pOrderBookId,  
    boolean pIsBid,  
    long pVolume,  
    long pPrice,  
    int pValidityPeriod) {  
  
...  
  
    tOrder.orderDataInPrivate.validityPeriod =  
        new Integer(pValidityPeriod);  
  
    return tOrder;  
}  
}
```

### Case 2

```
-----  
(within test case)  
OrderInsertRsp askOrder1 = enterBidLimitOrder(myOb,  
    400,  
    200);  
-----  
  
public OrderInsertRsp enterBidLimitOrder(long pOBid,  
    long pVolume,  
    long pPrice) {  
  
    return enterLimitOrder(pOBid, true, pVolume, pPrice);  
}  
  
public OrderInsertRsp enterLimitOrder(long pOrderBookId,  
    boolean pIsBid,  
    long pVolume,  
    long pPrice) {  
  
    OrderInsertReq tReq = BeanTestUtil.createOrderInsertReq(pOrderBookId,  
        pIsBid,  
        pVolume,  
        pPrice,  
        VALIDITY_PERIOD.GOOD_FOR_DAY);  
  
...  
    tOrder.orderDataInPrivate.validityPeriod =  
        new Integer(pValidityPeriod);  
  
    return tRsp;  
}  
}
```

### 6.1.2 Grace

Circumstances did not make it possible to make the comparing experiment identical for both frameworks. Grace tests could not simply be moved from the Bank solution to the Product due to too extensive modification of the underlying systems. That is, the system as whole did not reject the adoption of Grace-tests, but the task to set up the Grace framework in order to be able to run Grace-test on it, was too extensive. The case study examined how the abstraction of frameworks handled a change in the protocol structure, not how extensive the work should be to set up a Grace framework for a system that is not prepared to the framework.

However, the same experimental setup was reached through implementing tests after a merge of the Product and another project at Cinnober, this project, very similar to the Bank solution. The merge could be viewed as moving test cases from the Bank solution to the Product since the tests had been written in one project and then, when the two projects were merged, the tests were moved into the Product. After the merge, the same problem as for the JUnit environment occurred. The attribute fields had been changed and modifications had to be made in order to make the test run.

The change of attributes in Grace is much easier than in JUnit. Since the interfaces are auto generated from an *xml*-file where all attributes are defined, the additional attributes could be submitted and the code was regenerated. Methods had to be modified and changed in some fields, but this was only made once and worked through all tests. In order to make the tests work, very little work had to be done.

However, the validation was not easy to accomplish. Since the validation is a comparison of the expected and the actual output, all tests would fail if no changes were made to the expected output. This seemed to be an extensive task, just as the one made for all test cases in the JUnit. But, since *xml*-files are used, a replace script could be used to replace the old fields with the new ones.

## 7 DISCUSSION

When trying to answer the question of how to design the abstraction for automated tests, one discovers that there are two different parts of the abstraction; input and output. Both depend on what kind of framework that is used but they derive from different locations. The input abstraction is defined through how the test scripts are written and the output abstraction depends on how the validation is performed.

### 7.1 Summary

The case study did indeed show how important it is to keep a high level of abstraction when writing tests. That is, using convenience methods as far as possible when doing the same thing several of times in order to decrease maintenance efforts in the future. If the abstraction is kept high during the whole test suit, the suit is not sensitive for changes such as the attribute changes that occurred in the case study.

If all fields are verified at every single action point in the test, a good coverage of the system under test is reached. This procedure will most likely capture most of the faults. However, the tests become unreasonable big and cumbersome and the test will overlap each other, verifying the same issue lots of times. On the other hand, a test could check only the specific field of interest for this particular case. This will make the test suit more workable and more dynamic. However, the chance is big that some fall pits and complex cases are left out, thus faults are missed during the test procedure.

Additional to the question of what to be verified when using dynamic comparison, we also have to decide how often the verifications should be done. If verification is done after every single step in the test case, then, it is easy to catch logical failure within an operation and also easy to debug the process since the point of failure is easy to find. Nevertheless, if the validation is done after every single action the test case demands a long time to complete and also cause a lot of difficulties when the system under test is changed.

Three different frameworks were examined but only two are extensive enough to be overviewed for benefits and disadvantages.

The JUnit framework uses a dynamic comparison where verification could be done after every keystroke within the test case. JUnit is the framework examined here with the highest flexibility since it is capable of verifying states and variables easily at any point. This makes it suitable for complex operations where a series of event produces an output that is under test. However, the test cases take time to write since verification is done in many places and the expected state of a variable may be hard to predict and verify correctly. It is however easy to keep a high level of

abstraction since JUnit renders possible to use convenience methods when repetitive actions are made.

The Grace framework differs a lot from JUnit. Grace uses a post-execution comparison where predefined fields, or the whole output, are compared to an expected output. When the framework has been used for a while and feels easy to handle, the tests are fast written and verified. The implementation of used actions is only done once and after it is done, tests could be written easily. The procedure where a test stage is set up that suits the tests concerned works effectively and makes the tests easy to modify for different systems under test. The level of abstraction is kept high through the framework since attributes that are renamed can easily be changed and code can be regenerated. However, in the output, Grace shows a low level of abstraction if not the tests are very carefully written. Since the default is to use what *Fenster and Graham* call a sensitive validation, all of the output is compared to an expected one, and e.g. if only a time stamp has changed, the test will fail.

## 7.2 Conclusions

### *Robust vs. sensitive*

In *figure 3*, a brief picture described one of the conclusions of this thesis. The robust test suit has a much higher level of abstraction than the sensitive suit. If, as was the case for Grace, attributes have been added to the output, some tests will fail. Then, the robust suit where only the actual added attributes are checked needs to be changed only for those test where the new attributes are present and changes are not required for all test where the attribute happens to be included. The other tests, that do not concern the added attribute will run and pass. That is, if tests are constructed in such a way that every single test focus on just validating the output relevant for this test, and not all test validate all output every time, then the abstraction would increase and the maintainability effort would be lower.

However, a few more tests are written in the robust test suit than what is written in the sensitive one, but the tests in the left circle are easier to change and maintain since they all focus on one certain thing. This makes the effort of maintaining and writing the tests pretty much the same. A disadvantage with robust test suits is that since the test cases do not always overlap, there is a small probability that a fault is missed if it is located right between two cases. Nevertheless, this probability should be minimized and the faults should hopefully be picked up through manual or random tests.

### *Grace vs. JUnit*

JUnit and Grace are two different environments for automated testing. JUnit is focused on the ME and all its business logics. JUnit makes it possible to check every

single field, and also allows the tester to select which fields are to be checked. This makes the tool dynamic in terms of usability, that is, the tool makes it easy to write the tests wanted. JUnit also allows a test driven development within projects since e.g. trading beans in the ME could be tested at an early stage even though not all the other parts of the system are ready to use. This makes the tool usable for developers during a process where code and tests are written simultaneously, and new business functionality could be tested right after it has been written. However, JUnit tests today have some weaknesses in the abstraction of the input. As the case study showed, the maintenance effort of the JUnit tests could easily exceed acceptable limits when a new field is added to the protocol. Nevertheless, this could be fixed through increasing the level of abstraction with convenience methods explained in example 4. When tests are correctly written and all protocols are consistent, the abstraction of the output is such that it copes with extensive changes.

Grace, on the other hand, does not demand hardly any changes of the tests according to a protocol change. Grace is not focused on one part of the system, but more of the system in general, that is, the tool is focused on what is leaving and entering the TAX. The added attributes, from the case study, are submitted into the definition scripts, the code is regenerated and the tests run smoothly. However, Grace has its weaknesses in the output validation. If a field is added all test will run but they will all fail due to inconsistent validation. The expected file does not contain the newly added attribute, which the new file does. Thus, the two validation files are not similar and the test fails. The process of making the output equal to the expected is however not too extensive. For the tests that do include the added fields, expected files could be changed through a replace-script and only a few cases have to be modified and validated by hand once again. The replace script simply substitutes the old attribute name with a new one. This procedure is possible since the tests are written in xml-files.

In order to increase the level of abstraction the method showed in *example 3* could be used. The *xpath*-method only allows certain selected fields to be displayed in the output file. If this is done correctly, Grace becomes more of a robust tool where maintenance and support is easier.

Finally, the conclusion after this remark is that Grace is more dynamic to changes than what JUnit is, that is, Grace allows for a higher level of abstraction and thus has lower maintenance costs than JUnit. However, JUnit is easier to use and better in terms of advanced business function tests. JUnit is also an important tool when a test driven development process is in progress since it can be used even though the system as whole is under construction.

### **7.2.1 Recommendations**

In order to have an extensive test strategy, several testing procedures are needed. For black box testing, a combination of the two frameworks presented in this thesis is

recommended. However, modifications for both the tools are necessary in order to get a higher level of abstraction for the input and the output respectively.

To give the JUnit framework a higher level of abstraction, convenience methods must be created and used within test cases. The framework allows for a high level of abstraction, but the abstraction needs to be made use of by the person writing the tests. The test cases do not only get more rigid to changes and modifications of the protocol, but the tests also get easier to read since every method name explains what it does.

Changes which are needed for the Grace framework are changes for the abstraction of the output. Right now, Grace uses a very sensitive testing approach where the same field is verified several of times as the whole output is verified for almost every test. If the *xpath* method is used more frequently, where only the fields selected are verified, the tool gets more robust, thus, easier to maintain and support.

### **7.3 Future work**

This thesis has been a study of how the abstraction of an automated testing tool should be designed. The report does not give an extensive suggestion of how a complete design of a tool should be done and what features it should have, but more of suggestions for improvements for the frameworks studied.

There are many different aspects when designing an automated test tool and it would be interesting to study some aspects of validation where the question about robustness vs. sensitiveness could be analyzed more carefully. Also the validation is achieved through two different approaches for JUnit and Grace. Which should be used? Is a combination good or is one preferable?

## REFERENCES

- [1] Philipp Martin Schauer. *Market architecture of the largest stock exchanges*. Innsbruck, 2006
- [2] Avanza, [www.avanza.se](http://www.avanza.se), 2007
- [3] A. Spillner, T. Linz, H. Schaefer. *Software Testing Foundations*. dpunkt.verlag, Heidelberg, 2005
- [4] *Test methods and tool, an introduction*. Cinnober Financial Technology AB, Stockholm 2006
- [5] D. Graham, E. Van Veenendaal, I. Evans, R. Black. *Foundation of software testing*. Thomson Learning, London 2007
- [6] A. Kruthiventy, A. Shah, A. Datey. *Software testing guide book*. SofTReL, 2004
- [7] M. Fewster. D. Graham. *Software Test Automation*. Addison-Wesley, Great Britain, 1999
- [8] L. Kanglin, W. Mengqi. *Effective software test automation*. Sybex Inc, California, 2004
- [9] *TRADEExpress Platform, Thecnical Architecture*. Cinnober Financial Technology AB, Stockholm, 2007
- [10] *TRADEExpress Trading Engine*. Cinnober Financial Technology AB, Stockholm, 2007
- [11] M. Blackburn, R. Busser, A. Nauman. *Understanding the generations of test automation*. Software Product Consortium, NFP, 2003
- [12] Y. K. Malaiya. *Automatic test software*. Computer Science Department, Colorado State University
- [13] *Grace Framework Developers' Guide*, version 1.0, Cinnober Financial Technology AB, Stockholm 2004
- [14] L-I Sellberg, *Mexus-main design ideas, v1.3*, Stockholm 2003