FINAL REPORT, SENG3011 - SOFTWARE ENGINEERING WORKSHOP 3

Authors: The Gs - Saishav Agarwal, Yuwei Chen, Rashim Rahman, Hariharen Veerakumar

---

REQUIREMENTS AND USE CASES FOR ALL ASPECTS OF OURS IMPLEMENTATION

In order to complete the module, and GUI successfully, we created requirements that if met, would ensure the project following the spec of the course. After creating the requirements and molding our project to them, we ran some use cases to ensure the consistency between our requirements and implementation. All use cases used test that the requirements have been made, therefore the requirements demonstrate what our project has achieved.

REQUIREMENTS AND USE CASES FOR THE MSM MODULE

Requirements:

1. Invoke the module, as a standalone executable, with the input trades file in CSV format, parameters file within the same directory, outputting a correct output trades file in CSV format and a log file containing information about the input window size, threshold size, time taken for module to run and the number of trades made.

Use Cases:

1. In the directory of the executable module, include a valid small CSV trades file, and a parameters file. Run the module by clicking on it. Check that a log file has been created, check that the parameters mentioned in the log file are the same as the parameters provided in the parameters file, check the time taken to run the module, and check the number of trades made. Next open the created CSV new trades file, check that the number of trades in this file is the same as the number of trades conducted in the log file. Finally manually calculate the momentum strategy using the specified parameters on the input trades CSV file, compare the manual results with the executable module results to ensure correctness.

2. In the directory of the executable module, include only a valid parameters file; leave the required input CSV trades file missing. Execute the module and ensure that a new

trades CSV file has not been created. Check that a log file has been created, and within it an appropriate message detailing the missing input trades exists.

3. In the directory of the executable module, include only an input trades CSV file, leaving out the parameters file. Execute the module and ensure that a new trades CSV file has not been created. Check that a log file has been created, and within it an appropriate message detailing the missing parameters exists.

REQUIREMENTS AND USE CASES FOR THE GUI RUNNING THE MSM MODULE

Requirements:

1. The GUI should load a selected MSM module from a directory
2. The GUI should load a selected input CSV trades file from a directory
3. The GUI will allow users to enter the window size
4. The GUI will allow users to enter threshold value
5. The GUI will allow users to enter output file name
6. The GUI will invoke the MSM module, using user entered parameters, to output a trades file having the user specified name, a graph visually demonstrating the new trades file and a log file containing information about the input window size, threshold size, time taken for module to run and the number of trades made.
7. The GUI should be aesthetically pleasing, and easy to use for the average user.
8. The GUI should be able to load and run another group's MSM module.

Use Cases:

1. Modify the GUI to print out the module name, and directory of the loaded module. Run the GUI, selecting the desired module. Check that the name and directory output by the GUI is the correct name and directory location of the input module.
2. Modify the GUI to print out the input file name, and directory of the loaded file. Run the GUI, selecting the desired module. Check that the name and directory output by the GUI is the correct name and directory location of the input CSV file.
3. Modify the GUI to print out the user entered; window size, threshold value and output file name. Run the GUI, entering an integer value in the window size text box, a float value into the threshold text box, and a string value into the output file name text box. Ensure that the values printed by the GUI, correspond to the values entered.
4. Start the GUI, select our group's GUI module, select a CSV input trades file, and then input values for the window size, threshold and output file name. Run the GUI. Check that a log file has been created, check that the parameters mentioned in the log file are the same as the parameters entered into the GUI, check the time taken to run the module, and check the number of trades made. Next ensure that a new CSV file has

been created having the name of the GUI user entered output file name and check that the number of trades in this file is the same as the number of trades conducted in the log file. Manually calculate the momentum strategy using the specified parameters on the input trades CSV file, compare the manual results with the executable module results to ensure correctness. Check that a graph corresponding to the data in the newly created file is generated as a result of running the application.

5. Start the GUI, and load the input trades CSV file, and enter the window size, threshold value, output file name and run the application. Ensure that the GUI generates an error message about missing module.
6. Start the GUI, and load the MSM module, and enter the window size, threshold value, and output file name and run the application. Ensure that the GUI generates an error message about missing input trades CSV file and no new CSV file has been created.
7. Start the GUI, and load the MSM module, and enter the window size, threshold value, and output file name and run the application. Ensure that the GUI generates an error message about missing input trades CSV file and no new CSV file has been created.
8. Start the GUI, and load the MSM module, load the input trade CSV file and enter the window size, threshold value, and output file name and run the application. Ensure that the GUI generates an error message about missing parameters and no new CSV file has been created.
9. Use a random person, giving them the location and name of MSM file, location and name of input trades CSV file, and information about what a window size, and threshold value is. Start the GUI, and allow them to follow through it, without any more help except those on the GUI. Ensure that the GUI includes information that allows the person to run the application correctly.
10. Start the GUI, load the MSM module of another group, load the input trades CSV file, and enter the window size, threshold value and output filename. Run the application and check the log file such that the execution of the module ran without error.

Design, implementation and testing

OVERALL SOFTWARE ARCHITECTURE OF APPLICATION

The architectures of each application are described with Figures 1 and 2 respectively.

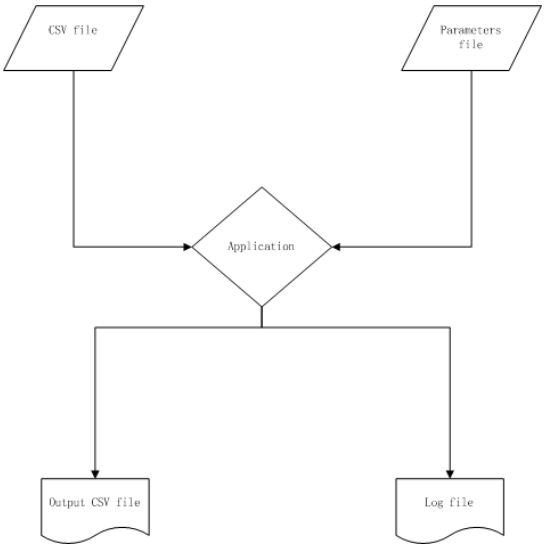

Figure 1: Visual description of how the strategy application interacts with its inputs and outputs

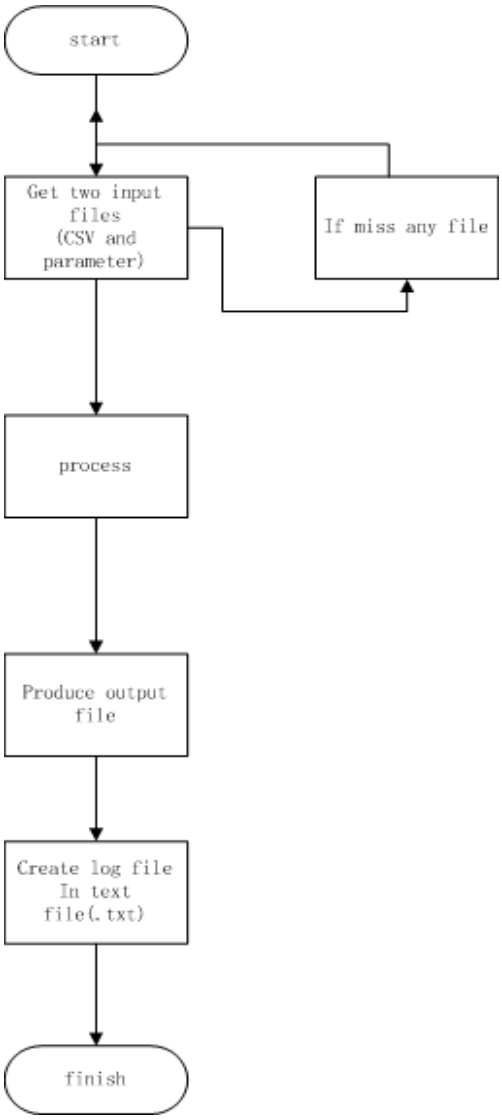

Figure 2: the lifecycle of the GUI application's usage from start of use to success

The architecture was designed in a manner such that it was in line with the methodology required for applying a financial strategy. The inputs include:

- A parameters file
- Input SIRCA CSV file

The parameters file is included to allow a more versatile use of the module. Although it is recommended not to edit any files, it can be done at their own risk – these errors cannot be controlled by anyone but the user. Nonetheless, the application allows the use of flags to change the parameters safely.

The log file, despite being a requirement, proved to be useful for analysis and comparison of the module.
The lifecycle of the application within the module is designed to account for:

- Catching potential exceptions
- Ensure correctness and safety through the lifecycle
- Designed to achieve the result using minimal steps

The process is defined as an algorithm, known as a 'financial strategy', and is currently limited. However, the order in which the files are read and produced are imperative and, hence, are placed in their respective positions accordingly in the lifecycle.

After this, the graph is created using the GUI application, which has extra functionality inside to do so. This does not require any error-checking, since all data provided is validated using the strategy application.

Module and interaction.

As a standalone application, any third party software system can download the application via the download link on our webpage. The application will be packaged in a zip file, and can be extracted to obtain individual files as follows:

- A CSV file which is essentially the main input
- A CSV parameters file
- The executable application file itself
- A JAR file that executes the application

STRATEGY APPLICATION

The application can be invoked in different ways using a relatively current Windows OS i.e. XP or above. We can type the following command in the command line terminal:

msm_v_1_2.exe

This will process the input and generate the required output file and a log file, as shown in the diagram previously. A default input file from the package will be run with the application. This will also be the course of action if the sample application is double-clicked.

The user is able to select an input file of his/her choice. Finally, using these files as input the application will allow the user to invoke the module to process the file and generate the output files. The output CSV and log files will be stored in the user's computer.

For further options, we have the following flags for custom use:

- -w: Window size
- -t: Threshold size
- -i: directory of input file
- -o: directory of output file
- -h: help

The strategy application has a very strong point in being flexible for necessary parameters, and hence the application is available to be integrated effectively into third party applications. The commands to be used are very simple, and do not so much as require a specific order in which parameters are entered. It also does error validation to ensure the application does not crash.

The results produced are accurate, as seen below in the Testing Documentation. Hence, we have created a stable application that works as a standalone, as well as being easily integrated into other modules.

GRAPHICAL USER INTERFACE (GUI)

With a GUI, it is more user-friendly to modify the parameters. Hence, the module also contains a GUI component. All of this can be done via an interactive graphical interface, which can be accessed publicly using our webpage.

The user interface is built by Java which contained by a java class called GUI. The GUI of our project is built to be able to select an input CSV file and a module. The file path will be show on the user interface. We can also set the output file name, N(windows size) value and TH(threshold value) value by input the number into text boxes, as well as the output directory. The order of operations is shown in an easy-to-use order, such that users and follow the order and run the application step by step.

While the program is running a "Loading" text will be shown at the bottom of the window. After processing a window will jump out with a finish statement with a graph shows the trade data. Also the output file path will be opened automatically for the user.

Furthermore, the GUI has a default behavior integrated into it. This is used for demonstration purposes for the user.

The GUI is created in a way for ease of use. It is run simply by double-clicking the module. Alternatively, it can be done through using the command-line command:

java -jar sengGUI_v_1_5.jar

Given the nature of the application, no command line arguments are used – everything is done using graphics.

The GUI also produces a graph for an interpretable result. Whilst the strategy application produces results in a CSV format, the GUI application takes these results and provides a visual interpretation. The features in the graph include:

- Moving average over time show in orange line
- All trade show in blue line over time
- Red and green points shows the type of the trades where red shows the sell and green shows the buy.

The graph will show all transactions provided over time, as well as a graph of the cumulative average. Users can find how much they benefit or lose from each trade. As an example, one can find they have made gains or losses by visually comparing the heights of each of the buy and sell points, as well as making further deductions by comparing it to the average line.

The graph has been created in a user-friendly fashion. The background color of the graph is gray, which can shows the points more clear, because green and red in the gray background is highlighted more easily compared with the blue line. The span of the time on the x-axis and the price on the y-axis is decided based on the maximum and minimum prices and time over all transactions, which can ensure the graph covered the entire layout.

The GUI then directly speaks to an object with reads in our input and output files, followed by creating the graph. The graph is displayed in a separate window for a furthered user-friendly experience. This marks the end of the use-cases of the GUI. The user can then re-use the application using other inputs.

Finally, the application can be integrated with one other module, as implemented by “___”. This has been done in a manner according to their documentation, which has been provided as easy to use. As both use similar parameters, the GUI simply processes the parameter data in a different fashion, and

produces an output of a similar nature.

IMPLEMENTATION LANGUAGE AND ENVIRONMENT USED

The strategy application was written in Ruby. We chose the language for a number of reasons, above simply being most comfortable with the language.

Firstly, the parsing of information is made much easier using Ruby, making it a much more practical language for what we need. It is easily achieved using the regular expression tools that Ruby provides. Secondly, it uses principles of Object Oriented Programming – one which the developers of the module are strongly familiar with. These advantages are all encompassed in another functionality that Ruby provides. The files provided for input are all of a comma-separated value format. Ruby furthers the ease of using these files by providing classes to read in CSV files. Each record is then stored as an object, with characteristics stored as per its description within the CSV file, further ensuring a simplified program to achieve the aim. It also provides a couple feasible methods to develop and execute our module.

We made a wise choice in using Java for our GUI application. All graphical elements of the GUI use the commonly-used “Swing” library, where we can set the points easily on the graph. All the points are stored in an object containing the date and the price at each point. We have further created a number of objects to coincide and create an effective GUI.

TESTING OF STRATEGY APPLICATION

Testing was carried out on a Windows machine, since our module is compatible with windows at the moment. We used Microsoft Excel as a tool to manually compute data. The application within our module is a .exe executable file. This can be executed on the command line as well as doubling clicking the application. Tests were performed using both methods, whilst running Windows 7 32-bit and 8.1 6.4-bit as OS's when carrying out these tests.

The limitation was that many of these tests were carried out for a small number of entries, since it would be very time consuming to verify output manually for larger chunks of data. In taking in all different types of records, we assume that if it works for a small input then it should then work for a large input size.

Overview of Test Data

Using 10 trade entries, we tested our module by varying the value of n and th (“window size for simple moving average” and “threshold value” respectively). There were 3 main test cases for which were analysed:

1. Setting parameters n = 3 and th = 0.0005

2. Changing n = 6 and keeping th = 0.0005
3. Changing th = 0.0008 and keeping n = 3

These test cases were repeated with a standardised sample data file that was 100227 records in size.

We must note that the default values for the parameters are set as n = 3 and th = 0.00005. These default values were used upon double-clicking.

We then carried out testing using other teams' modules. We used the same parameter and input file and generated the output file. This helped us compare our results with theirs. To sum it up, we tested our module by varying every parameter one by one, while keeping the others constant.

Details regarding test data

The following are the testing files included with this document:

Input file with 10 trade entries

trades10input.csv

Parameters	**Manual Testing process files**	**Output Files(Manual Testing)**	**Output Files (Module Generated)**
n = 3 th = 0.0005	trades10testingprocess.csv	trades10output.csv	trades10AutoOut.csv
n = 6 th = 0.0005	trades10testingprocessN.csv	trades10outputN.csv	trades10AutoOutN.csv
n = 3 th = 0.0008	trades10testingprocessTH.csv	trades10outputTH.csv	trades10AutoOutTH.csv

Performance Testing against 3 other modules

Our log file	MSM Module A	MSM Module B	MSM Module C
log.txt	logTestA.txt	logTestB	logTestC.txt

Testing Process:

First, we chose 10 trade entries from the original input file, and loaded it onto an excel spreadsheet. Then we manually entered the MSM strategy formulas one by one, to compute

whether a buy or a sell signal was to be generated. The R_t values were calculated in an extra column using an excel formula. Next the SMA values were calculated in a new column using the R_t column and an excel formula. The SMA column was then used to calculate the TSV_t value in another column also using an excel formula. Finally we manually compared the TSV_t value to our threshold to compute whether to buy or sell. We then saved this information into an output file.

We used this process for each of the input data, firstly for 10 inputs, parameters n = 3, threshold = 0.0005, and concurrently ran it using our module. We did this for all three test cases. We then repeated for the larger set of data.

Once that was done, we had 3 output files generated by running the input on the module and 3 output files, which were manually computed. Each pair of files was compared by running the Unix command 'diff' on each corresponding pair. From this, we concluded that the output generated by the module matched the manually computed results.

We then used other teams' modules to do the same, and compared the output. Furthermore, we compared our application for speed performance. We found that our module was taking longer than expected.

COMPARISONS

- A: found same number of trades. Produced 1515ms in log file. 1890 ENTER records produced

- B: produced results in ~90-100ms, however results did not vary with parameter changes. Consistently produced all TRADE transactions from original file. Same number of trades found

- C: 1698 ENTERS produced in 357ms, also 16485 trades

CONCLUSIONS

One of the main attributes to the time it takes is due to the way Ruby is run on Windows in particular. Ruby-generated applications, as a result, do not work well with Windows OS systems.

We can further deduce that Ruby, itself, is not comparable in terms of time performance to many other common languages and, in fact, proves itself to be slower than all other popular languages, namely Java which was primarily used.

In regards to results, there appears to be minor differences in results between the different modules. Given the test cases provided, we are yet to determine where all cases have been accounted for. This will be done via more unit tests.

Conclusions, appraisals and team organisation

PROJECT MANAGEMENT SOFTWARE

One of the major problems that we have encountered with software development in the past was collaborating as a team. To overcome this, we have decided to take project management more seriously.

We are using Pivotal Tracker for managing tasks assigned to each member in the team. Each task will consist of a description and a deadline. We have decided to create stories for each of the features that we expect our application to have. Tasks will be not only for development, but for testing as well.

We are also using Git for version control of the module. This helps us distribute the module easily between each developer, whilst allowing all changes to be made in one central location. Despite its complex nature, all users are proficient in how to use it and can use it in a powerful manner.

Apart from tasks, any bugs in the application reported by members of the team will be added as well. This will help us track our versions, and hence allow a better version control. Thus, Pivotal Tracker and Git complement each other well for our purposes.

With every release, we will include details of the release on the page too. This will enable us to get a clear picture of our current progress and backlog. Using this tool will help us communicate better, monitor each other's performance and hence work more efficiently.

TEAM ORGANIZATION

At the beginning of the project, we had decided to take team organization and collaboration seriously. The main reason for this was to rectify team management issues that we had faced in the past. To do so, we decided to use a project management tool called Pivotal Tracker. Initially, we used it to an extent but as time progressed, the tool did not work much to our advantage. We used the basic functions of Pivotal Tracker for task allocation and project timeline. Often team members failed to update their task statuses, which made it difficult for us to follow the project workflow and backlog. Thus, we decided to switch to other modes of communication for team management.

We used GIT for version control. This was very helpful for the team. Previously, we relied on emails and dropbox to get code that could have been updated by one or more members. We realised that merging and rebuilding code manually wasn't efficient at all. GIT served as a

powerful tool to resolve this issue. By exploiting the various GIT commands, we saved a large amount of time that we would have otherwise spent on organizing code.

Apart from using online tools, we organized meetings in-person where we discussed our short-term project goals, with the team members that turned up to it. We designated tasks to all team members and got feedback about the progress of the project. Often, we used this time to help each other with bug fixing on code that could have been interrupting a team member from progressing with their allocated tasks.

PERFORMANCE

The project did not go as good as we had thought it would. In the initial few weeks while working on the MSM module, we were getting excellent feedback from the course mentors. We were being able to meet with the goals we had set. As we moved to the later part of the project, our performance deteriorated and we weren't able to achieve what we have planned we would. Considering the fact that team participation was an issue, our performance was somewhat justified.

There was a lot to learn from the project as a whole. We did not have a commerce/finance background. The project helped us understand how trading is actually carried out and how it can be made efficient using high performance software applications. It helped us realize the importance of developing software from a customer's perspective. Having had a chance to interact with Optiver engineers helped us to understand more about the software market and what they expect from software engineers. Apart from this, we learnt a new programming language called Ruby. While only one of our team members was proficient in Ruby, now we are all familiar with Ruby programming. However, making Ruby as our choice of implementation did not work out too well for us. We will talk about this in the section that follows.

PROBLEMS ENCOUNTERED

There were a few issues that we encountered during the course of the project. There were two main issues, which are described below.

The first issue was with the language we used for the MSM module. We used ruby and later realised that it wasn't a good choice. While building a GUI for the later part of the project, we figured that the use of Java would have been better. One of the requirements of the GUI was to be compatible with other modules. Since we had implemented the module using Ruby, we had to create an executable file to run it. The way this was coded to work with the GUI was completely different to what it would have been if we used Java and generated a JAR file.

Thus, there was no way we could use our GUI with other modules since most of the teams generated JAR files for the same.

Team participation was the other problem that we faced. From the beginning, some team members did not show any interest in working on the project. Work was being done half-heartedly after repeated pestering by other team members. This definitely lowered the team's confidence as a whole. Some team members did not take any initiative to get work done throughout the course of the project. This led to a lot of pending work to be done by other team members in the last minute. This surely had an adverse effect on our performance.

THINGS WE WOULD DO DIFFERENTLY

Firstly, we would use Java as the programming language for the MSM Strategy Module and generate a JAR file. As mentioned earlier, this would match the executable type for most other modules allowing us to run them from our GUI.

The second reason for not using Ruby is performance. The project dealt with high frequency trading, where code performance was the key. During performance testing, we discovered that our module that was implemented using Ruby wasn't performing as good as other modules that used Java.

Also, we would redesign the graph representation of the output generated by our module. We realised that the using of the Java swing library wasn't a good choice. We would use, for example, JFreeChart or JGraph that has more features and is comparatively very user-friendly. This would allow us more ways to represent the data using visual aids, which would allow customers to analyze data in more detail.