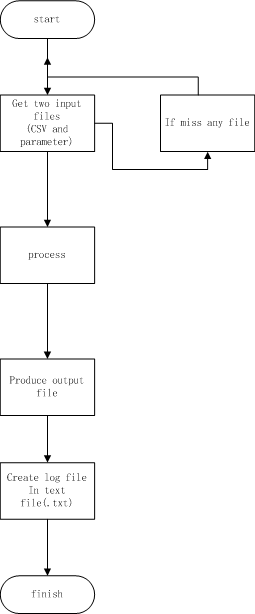
# FINAL REPORT , SENG3011 - SOFTWARE ENGINEERING WORKSHOP 3

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Overall software architecture

The architectures for Sections 1 and 2 are described with Figures 1 and 2 respectively.

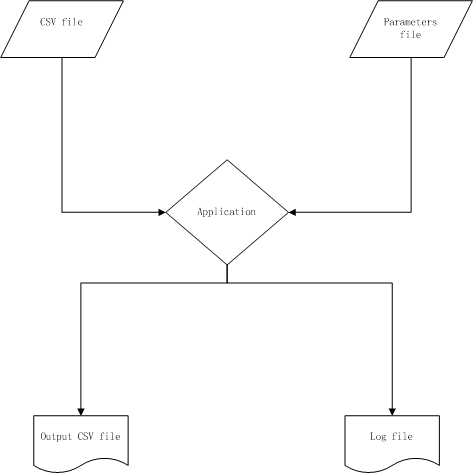


Figure 1: Visual description of how the application interacts with its inputs and outputs

Figure 2: the lifecycle of the 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.

## Development of module and ability to run it in standalone mode

The module was created such that it analysed and, using the Momentum strategy, made choices to buy and sell stocks according to the data provided. The strategy will vary based on personal preference and, as such, we also require parameters. Finally, the module should ultimately be able to produce a result to a certain file, in a presentable manner. Therefore, to accompany the developed module, we will also require:

* the required file to be read in, in the correct format, that holds the input data
* A parameters file, that will hold the parameters required as per the user’s choice
* The path of a directory to output the resulting file and log file to
* An application to produce an interpretation of the information it is given

The application is readily available for download, packaged with a sample file as well as the parameters file. These two input files contain comma-separated value (CSV) data to be read in to complete the application input. The files are to be extracted into a specific location.

The module can be run directly with the application, or through using the user-friendly GUI provided. The application is run with the input file and the parameters file given as parameters for the application. Furthermore, we can provide command-line arguments to specifyThe application will use the data read in from the files it is directed to in order to commit to the financial strategy the application is scripted to do.

The output data will be stored in another CSV file, since this holds the original format of the input. It is also universally used, and easily read in with applications such as Microsoft Excel. The log file produced will be in the form of a text (.txt) file. Again, this is readily legible by most text editing applications. Both these kinds of files are to be written out from the application once the results are produced.

Hence, provided the application within the module is provided with the input files (and their respective paths) in the module, and a path to create output to, the module will be able to function as a standalone application. Further refinement will determine what operating systems specifically will be supported. We will be aiming to cater for Windows, Linux and Apple OSs. (For further details on producing the standalone executable application, please see Section 4).

## 2. Third party software system invocation of our 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 3 individual files which are as following:

* 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

The most feasible decision involves development via Ruby as a module invoking the Windows executable (for Windows and, with Wine, for Linux).

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.

Eventually, we expect our application to be available to third party applications more efficiently. With a GUI, there is more flexibility in terms of modifying the parameters file as per the user’s requirements. 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.

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.

## Implementation language and environment to be used

The module is to be written in Ruby. We have chosen this 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.

Lastly, it provides a couple feasible methods to develop and execute our module – which are applicable for other languages, of course. However, in our considerations, we also considered conversion of scripts to actual usable applications. However, we must first consider which operating system we should use.

If we plan to use just the command prompt/ terminal we would need to run the module from it, and direct out input and output to it manually in our command. This is simple although during testing would be hard to make quick changes. The Mac OS X terminal would be the better option as Ruby is pre-installed, making it easier to get the environment set up.

The decision follows that, using the command prompt/terminal wouldn’t require the installation of third-party IDE’s, and hence would require less training for the developers. However, in using an IDE, which requires an initial installation, we will allow us to handle editing, input/output files and running the module in one window. Our final decision is as per Section 2, whereby there will be two different types of modules to account for Mac OSX, as well as Windows and Linux, separately.

## Testing Environment

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 analyzed:

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 Rt values were calculated in an extra column using an excel formula. Next the SMA values were calculated in a new column using the Rt column and an excel formula. The SMA column was then used to calculate the TSVt value in another column also using an excel formula. Finally we manually compared the TSVt 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.

## 5. 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.