# INITIAL TESTING DOCUMENTATION, SENG3011

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

# Latest version of the software architecture

Components

The main necessary component, given a client, is a variety of server components and systems within a computer itself. Provided this server can run indefinitely, the design will work such that the client can access the information at any time necessary. The client view is generated as a web application for dynamic use by the user. The client will also be able to send requests to the server.

A web server essentially creates the website and sends the information to the client. Various templates, and other data, will be accessed by this server and transmitted to the client upon request from the “controller”. The web server will send requests to the application server to retrieve information by contacting other servers, such as from www.aec.gov.au (Australian Electoral Commission Official Site) to collect details regarding seating positions in Houses, for example. Considering some of this information can change extremely readily, we will have to collect this dynamically. The web server then receives this data and uses it to update the client.

Relationships between components, and deployment

The web server will require applications for their respective tasks, mainly concerning the sending and receiving of data to and from other components. In particular, the web server will have to retrieve data from other servers, as well as receive requests and send replies back to the client. Hence, it acts as a ‘model’, in our MVC framework, while the client incorporates the ‘view’ and ‘controller’. This relationship is more so evident in the figure above.

Lastly, the type of browsers to use was chosen based on a universal scoring system used to judge the rendering ability of HTML5 and CSS3 in particular. From experience with all browsers throughout time, we decided that the fairest minimal score, considering the available upgrades for each browser, would be 300. This was also a key determining factor for which OS compatibilities we will be catering for, as seen in the next section.

The browsers chosen include Internet Explorer 10 (scoring 322), Chrome 7 (308), Firefox 6 (333), Safari 5.1 (319) and Opera 11.10 (301).

## 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.

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.

## 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.

COMPARISONS:

* Group 4 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
* Group 3 found same number of trades. Produced 1515ms in log file. 1890 ENTER records produced
* Group 7: 1698 ENTERS produced in 357ms, also 16485 trades