Software Engineering Coursework Report

Max McRuvie

School of Computing, Edinburgh Napier University, Edinburgh
40496329@live.napier.ac.uk

Abstract. This paper contains information on of the design, creation, and testing of the Napier Bank Messaging System coursework for the Software Engineering module.

The NBM System's purpose is to take in message inputs, sperate them based of their type (tweet, sms, or email), along with separating specified data depending on the data type (for example; for sms, abbreviation would be a specified data type) and display them on screen. The program must also be able to save files in JSON format to local storage.

Contents

Requirement Specs 3

Use Case Diagram 3

Class Diagram 3

Functional Requirements 4

Non-Functional Requirements 5

Testing 5

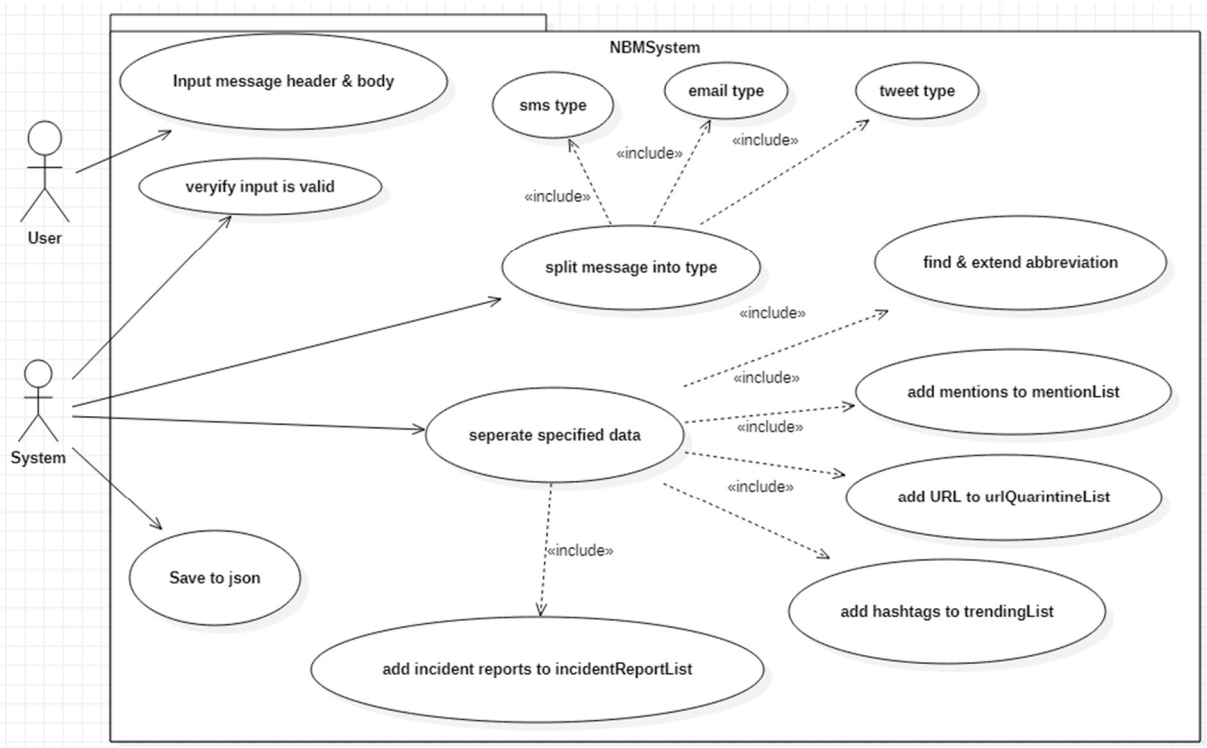
Version Control 7

Evolution Strategy 8

Requirement Specs

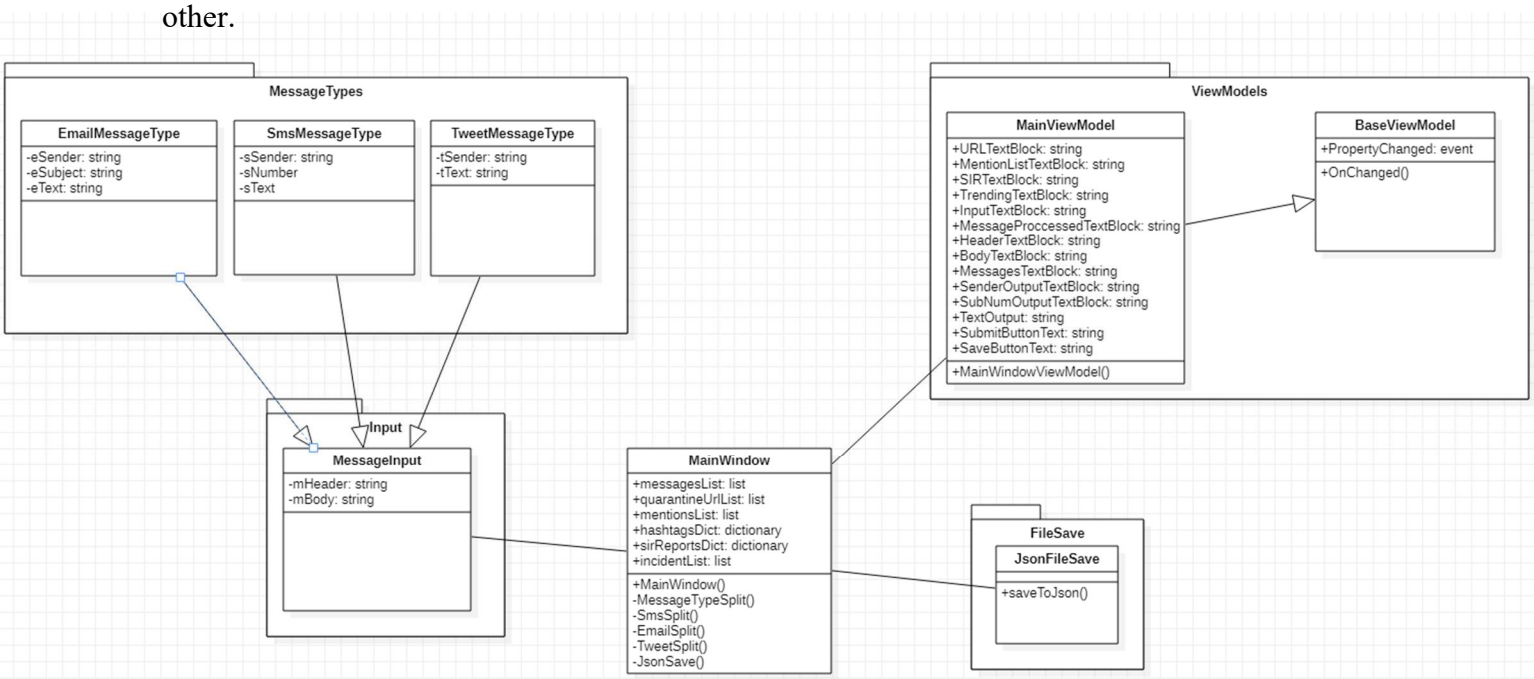
Use Case Diagram

To better grasp the requirement specification needed to create the NBM system I first had to create a use case diagram using StarUML. This process allowed me to visualize the requirements of the project and better plan the next steps in the process of designing/building it



Class Diagram

Another diagram I created to help visualise the requirements of the NBM System was a Class Diagram. This greatly helped when it came to building the system as it is essentially a blueprint for the required classes and the relationships, they will roughly have with each other.



Functional Requirements

For functional requirements there is a lot to unpack. Firstly, the system must be able to take in a user's input message (which will be a header, and a body) and verify that it meets the requirements specified.

For verifying the message header by ensuring start with the letter 'S', 'E' or 'T', this stands for the three different message types; Sms, Email, and Text. The header must then be followed by 9 numeric characters. The system must be able to check this to ensure that the message header is valid. After doing this it must then split the message into its specific data type based off the first letter in the message header.

After the message is split into its data type the class it is split into then verifies that it meets the requirements of that said data type. It does this by ensuring that the data is the correct length, contains the correct characters (for example emails must contain the @ char)

After the data types are verified the message must be checked for characteristics. Each message type has its own specific characteristics that the program must check for and then proceed to separate into either lists or dictionaries.

There may be two characteristics for tweets, mentions and hashtags. Mentions start with the '@' char, and hashtags start with the '#' char. The system must be able to recognise these and place them into their corresponding lists and dictionary.

For emails the characteristics are URL's and SIR's within the messages body. The URLs must be separated into a url quarantine list, and the SIR's must be separated into an SIR list.

After all the data is separated, it must then be displayed on screen. For the message the Sender, Number/Subject, and text are all displayed beneath the messages drop box (which contains every message header entered). Along with this the characterisations must be displayed, i.e url quarantine, SIR list, mentions list, and trending list. All of these must be displayed.

The system must also be able to save this data at the press of a button to a JSON file. Another requirement was to allow for the system to read in JSON files, I haven't managed to complete this task as I don't have enough time.

Non-Functional Requirements

There aren't many non-functional requirements I can think of for this project. The main one is that the UI must be somewhat pleasing to look at and easily understandable for users.

Aside from appearance of the UI the only other non-functional requirement I can think of would be scalability if this system was to ever go live and allow for multiple people to use it at once.

Testing

For testing my strategy was unit testing, as this allowed me to test specific sections of the system and get results from them. I tested the system on all inputs that can possibly put into the system to see how it would handle them, whether it would run as normal, output an exception handle, or if it would outright fail.

The primary objective of this testing was to ensure that the system works as expected and provides results that is required of it. The scope that is being solely focused on is the user's inputs, as this is the only section that can really be tested.

The test environment for this system was simply using the IDE visual studios, the process consisted of simply running the system and entering the below test data.

I never used the testing facilities provided by visual studios as I never felt confident using them, therefore I just did a basic test case table as seen below;

Inputs	Expected Result	Result	Description	Solution
S123456789	Fail	Fail	Exception is thrown saying the body cannot be empty	N/A
B123456789	Fail	Fail	Exception thrown saying the header must start with 'S', 'E', or 'T'	N/A
+32132113213 SenderTest TextTest	Fail	Fail	Exception is thrown saying header must be length of 10	N/A
S123 +32132113213 SenderTest TextTest	Fail	Fail	Exception is thrown saying header must be length of 10	N/A
S123456789 +32132113213 SenderTest TextTest	Pass	Pass	All info inputted meets the message requirements	N/A

S123456789 +32132113213 SenderTest	Exception should be thrown saying text shouldn't be empty	Exception is thrown saying text shouldn't be empty	Exception is thrown saying text cannot be empty	N/A
S123456789 +32132113213	Fail	Fail	Fails due to no sender	N/A
T123456789 @SenderTest TextTest	Pass	Pass	The header is added to the message list, with the body being displayed below	N/A
T123456789 SenderTest TextTest	Fail	Fail	Exception Handle thrown saying that an '@' must be present in sender	N/A
T123456789 @SenderText	Fail	Fail	Exception Handle thrown saying the text length must be between 1 and 140 chars long	N/A
T123456789 @SenderTest @MentionTest TextTest	Pass	Pass	Displays header, and body as expected. Also displays the mention in the mention list	N/A
T123456789 @SenderTest TextTest #TrendTest	Pass	Pass	Displays header, and body as expected. Also adds the # to the trending list	N/A
E123456789 test@email.com,TestSubject,TestTest	Pass	Pass	Displays header, and body as expected.	N/A
E123456789 testemail.com,TestSubject,TestTest	Fail	Fail	Exception Handle thrown saying the email address	N/A

			is not valid (due to no @)	
E123456789 test@email.com,, TestTest	Fail	Fail	Exception Handle thrown saying the Subject length must be between 1 and 20	N/A
E123456789 test@email.com,TestSubject,	Fail	Fail	Exception Handle thrown saying the email the email text length must be between 1 and 1028 characters long	N/A
E123456789 test@email.com,SIR,55-55-55,atmtheft,textest	Pass	Pass	Displays header, and body as expected. Also adds SIR to SIR List	N/A
E123456789 test@email.com,SIR,55-55-55 ,textest	Fail	Fail	Fails due to no SIR	N/A
E123456789 test@email.com,TestSubject,TestTest https:urltest.com	Pass	Pass	Displays header, and body as expected. Also adds url to url quarantine list	N/A

Version Control

For an agile approach I would highly suggest that GitHub for version control. GitHub allows for you to track every single change made in the form of commits, along with allowing you to create branches off from the main system. This allows you to design, build, and test potential functionality without compromising the main system. A merge can then be performed it is desired for the built branch to become the main system.

GitHub also allows for great cooperation amongst more than one developer as multiple people (if given access) can push commits to the system, all while everyone can see what everyone is doing and adjusting. Therefore, it's a good platform for collaborating with teammates.

Although I had some issues with GitHub (primarily from my lack of experience with it, and one merging issue) during this project, I would still use it as my proposed plan for version control.

Evolution Strategy

The primary evolution strategy I can think of for NBM System would be Agile, as it is one of the most widely used. It allows for rapid development and testing along with easier maintenance. As it allows for the entire project to be broken up into sections. Of course, if this system was to go live it is imaginable that there would be a team or several working on it, therefore sections of the system could be divided for each team using agile.

The main piece of evolution I can see for this system is to build the scalability of it for if it were to ever go live, along with adding more features to it. (Such as the JSON file read in that I didn't complete)