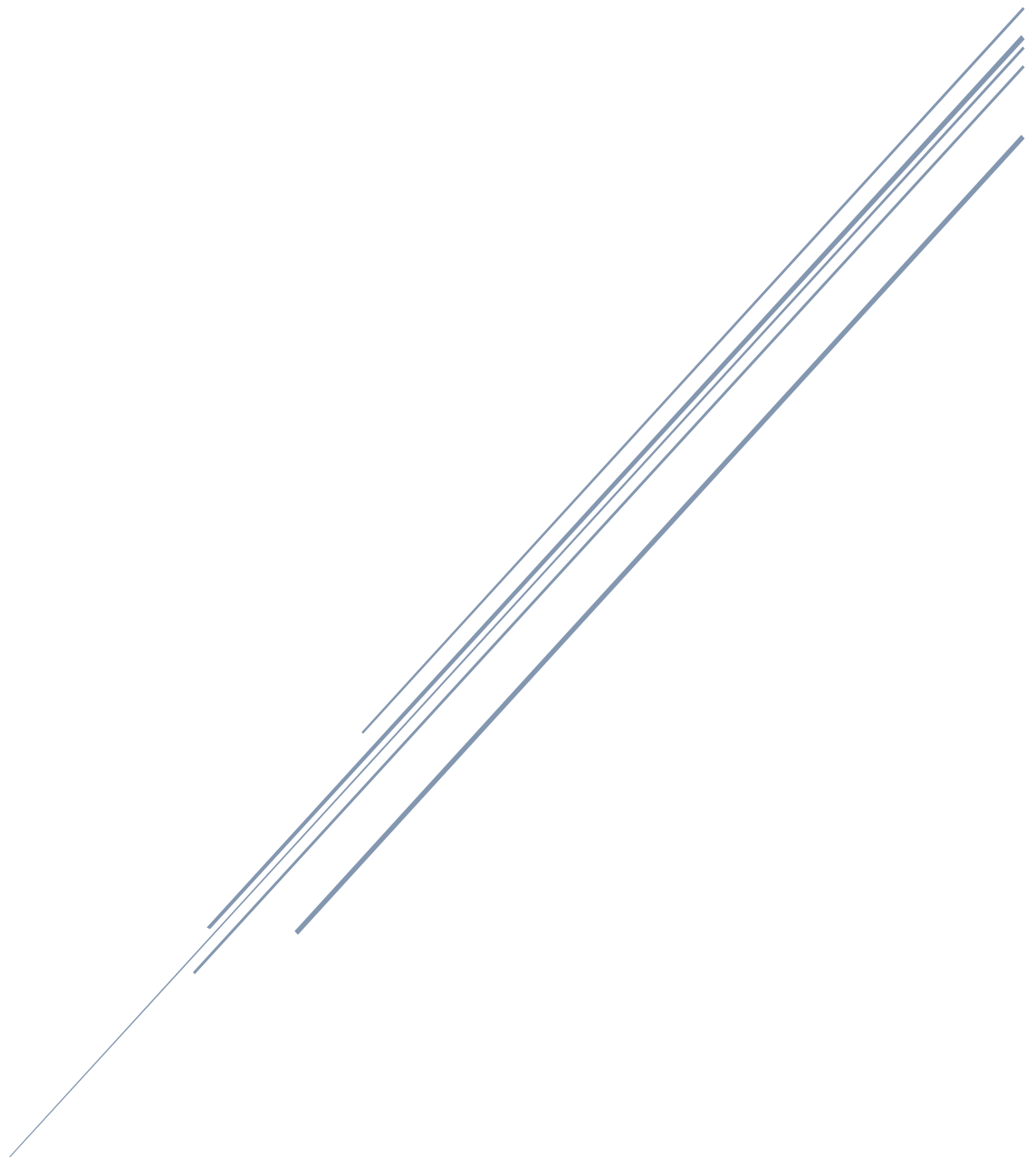


# SOFTWARE ENGINEERING

SET09102 2018-9 TR1 001



40315147

Edinburgh Napier University, Merchiston Campus

## Contents

|                                |    |
|--------------------------------|----|
| Requirement Specification..... | 2  |
| System Design .....            | 3  |
| Class Diagram.....             | 3  |
| Use Case .....                 | 5  |
| Testing.....                   | 6  |
| Unit Testing .....             | 6  |
| Integration Testing.....       | 8  |
| Acceptance Testing .....       | 11 |
| Evolution .....                | 13 |
| References .....               | 14 |

## Requirement Specification

The Edinburgh Napier Bank has assigned a task to create a message filtering service to allow users to communicate through the three means: SMS, Email and Tweets. Much like any messaging service, the desired service should validate, sanitize and categorise incoming messages to better the quality of the service.

The service should:

- Allow the user to define the type of message being sent & insert their ID
- Validate the input depending on the message type
- Output the message with abbreviation descriptions
- Allow the user to report an incident by using the SIR list through Email
- Quarantine URLs within Emails
- Record a sent message & output a message to the user

When sending a message, the user should input their ID which comprises of the message type for the first character ('T' for Tweet, 'S' for SMS & 'E' for Email) and the rest being the 9 figure ID code.

If the user decides to send a message by **SMS**, all the user needs to input are his/her ID, the phone number that the message is being sent to and the desired message while keeping to 140 characters max. Any Abbreviations such as "HF" or "LOL" will be translated next to the word with a "<>" on the outside of the description

However, if the user decides to send a message through **Email**, an ID, the email of the messaged person, Subject and body text must be provided. The two email types that exist within the service are the **Standard Email Messages** and the **Significant Incident Reports**. The Standard Email Message will allow for text with any URLs contained will be removed and replaced with "<URL Quarantined>" label instead. The Significant Incident Report will include its Subject section containing an "SIR dd/mm/yy" definition to start with. Following the subject box, the first two lines of the text body will contain the sort code "99-99-99" and then the following **Nature of Incident** definition. The incident list will contain events such as "Theft", "Staff Attack" & "Raid" and will be displayed after a message was sent. Just like the first email type, any URLs detected will be replaced by a quarantined label but the character limit for this message type is no more than 1028.

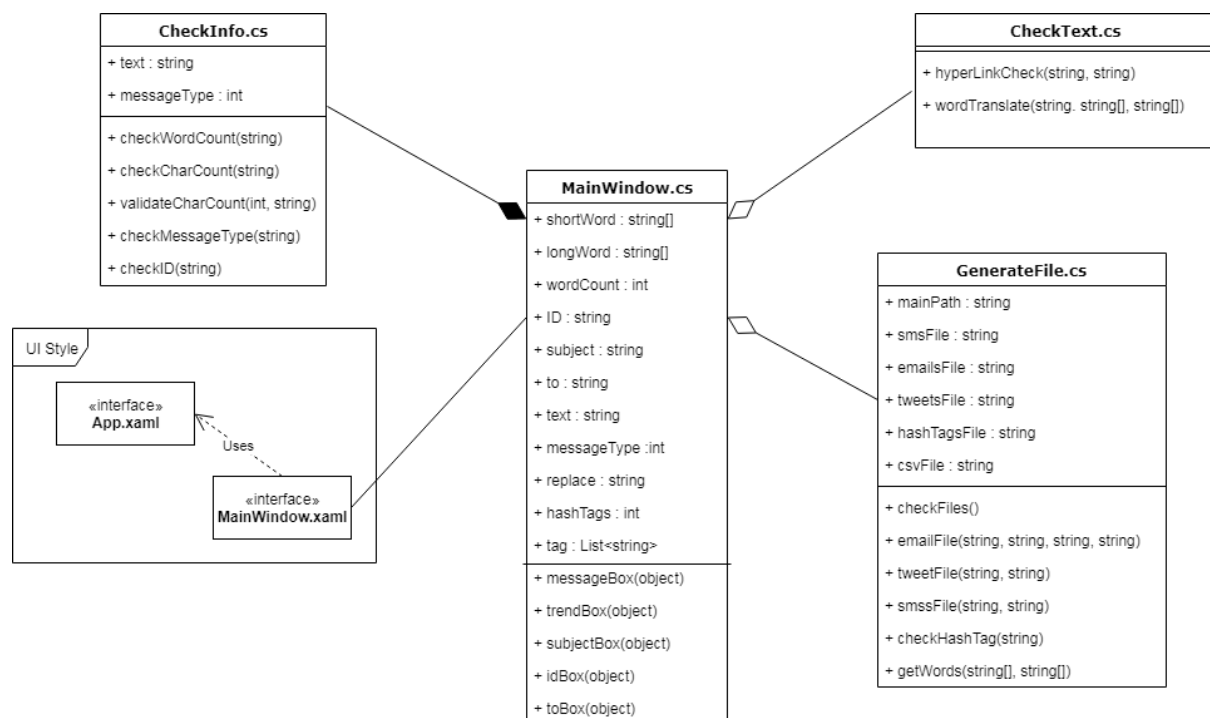
Much like SMS messages, **Tweets** will allow the user to input their IDs, the Twitter ID they are sending the message to and the text body. The sender will be defined by a "@" symbol followed by characters (15 max) while keeping to 140 characters. Abbreviations will be validated just as listed above and any hashtags detected will be stored in a separate list. After sending a message, the hashtags will then be displayed to the user in a manner of trend. Hashtags are defined with a "#" before the start of a word and will be detected if used.

Any message sent will be stored in a json file format. The file will be named based on the sender's ID and have its contents populated by the contents of the message itself. The files then should be available for the user to search through and output onto the application. The user Interface should be created using an input form such as C# WPF or Java Form.

# System Design

## Class Diagram

The language chosen to undertake this task is C# along with the Visual Studio Environment. Visual Studio offers the ability to create WPF applications through the use of visual objects and xaml style sheets. C# WPF offers a wider range of tools and visual objects compared to Java's Form environment, making it much easier to implement and edit UI features. Xaml files allow the developer to implement features in a simpler and more systematic way thanks to its HTML-like code style and easily definable objects.



The class diagram represents the structure of the bank messaging system. The 'MainWindow.cs' is the main class within the project that contains its own xaml file. As the main class will contain the definitions for UI objects, it will need to connect to the main screen of the application and initialize used tools from there. The '**MainWindow.xaml**' inherits from a global xaml files which contains styles for the UI. These styles work much like css when developing a website, in that the created methods overwrite the properties of tools and allows then to be referenced from any xaml file. In this case **App.xaml** will contain properties for the buttons and background colours which will be used for the application.

'**MainWindow.cs**' will contain standard variables to hold user input, lists of data and abbreviations. However due to better modularity, it is better to keep as least amount of methods in the main class as possible and separate them out depending on their purpose and functionality. The only methods the main class will inherit are the UI elements in order to allow for data to be transferred from the screen to the variables within the class. The main class will then instead reference other classes by passing values in order to perform algorithmic decisions based on the user's input.

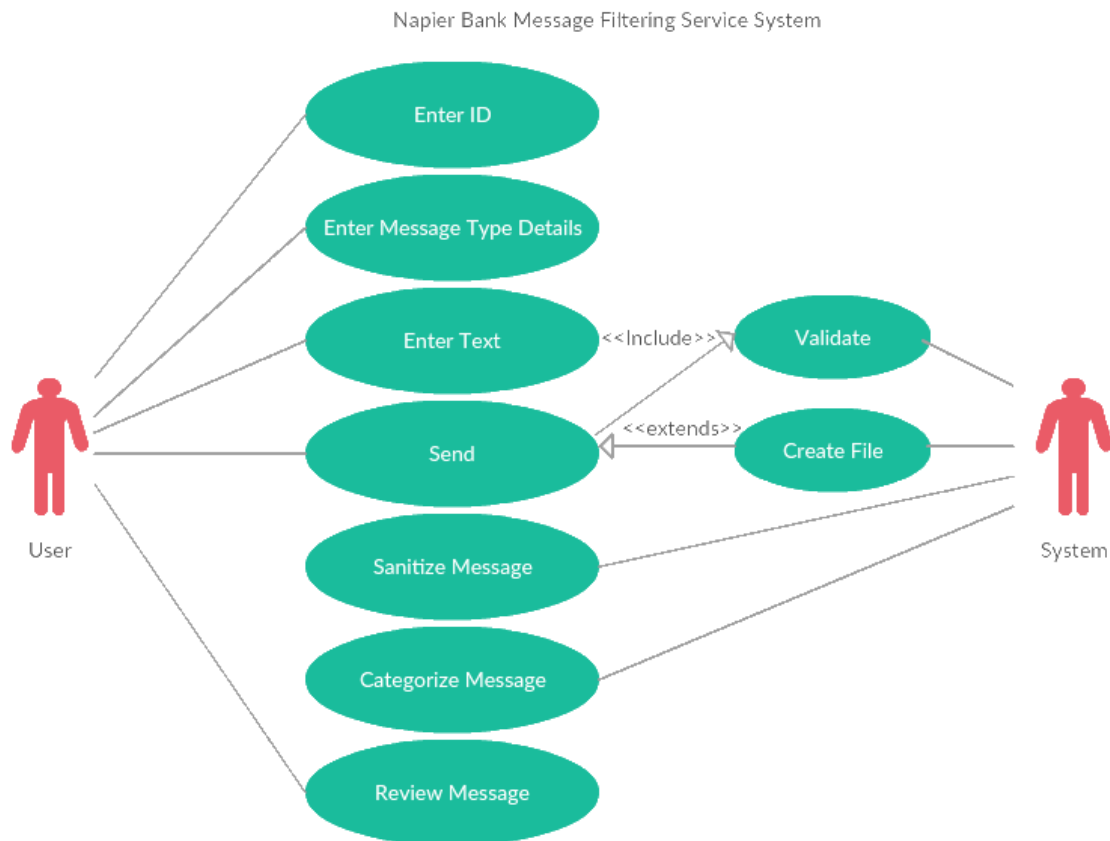
**'CheckoutInfo.cs'** will inherit data from the main class in a form of a composition relationship as there is a strong life cycle between the two classes. This class has a purpose to check all the input from the user and validate it so it may meet the standards of the client. The **'checkCharCount()'** method has the task of ensuring that the text body retrieves the character amount from a text file while **'checkWordCount()'** keeps track of the number of words within the message to make it simple to find an index of a particular word that is needed to be changed. **'validateCharCount()'** then simply returns a value representing whether the current message is within its character limits and whether the program can continue onto the sending phase. **'checkMessageType()'** takes the ID input from the user and checks the first character to dictate to the program which type of message out of the three is being used, if the character represents anything other than 'T', 'E' or 'S', the user is notified that the ID is incorrect. Finally, **'checkID()'** checks for the same ID input but instead checks the number values that come after the first character by ensuring the numbers are equal to 9 characters in length and are integers.

**'CheckText.cs'** is the class that is used for checking strictly only the body text input from the user and modifying it if needed. The **'hyperLinkCheck()'** method has the goal of ensuring that all URLs are quarantined within the body text. The only time this method is called is if the user wishes to send an Email with an URL located within the main text. The method first locates the word and deletes its place within the text, making sure to replace it with a "<URL Quarantined>" tag in its place. **'worldTranslage()'** is the method used primarily for reviewing the body text and providing any abbreviated words with their translations respectively. The method checks variables containing abbreviations and their definitions and checks each word within the body text for whether it contains any of the abbreviations listed within it. If the method does find a word that is listed within the containers, the method will then get that word's index and place the translation on its right side whilst being encapsulated within the "<>" parameters.

**'GenerateFile.cs'** is the class which supervises the creation of folders and files. The class provides methods which create folders for the right files dictated by the message type from the user's ID. Each message type has a separate container to allow for easier file locating and manipulation. As most of the methods revolve around directories, some have the task of receiving and writing data to and from these text files. Methods such as the **'emailsFile()'** have the task of storing data into a json file depending on the message type declared by the user. If the user decides to sent a Tweet, the **'tweetsFile()'** method is then activated instead. These methods locate their folder directory and insert a Json file containing the message details provided by the user. The **'checkFiles()'** method has the sole purpose of checking whether a directory exists for the insertion of Json files. If a directory could not be found, the needed directory is then created and ready for usage. The **'getWords()'** method has the task of taking in the details of an abbreviation list and inserting them into variables. This is done in order for another method to use these values and manipulate the body text once the user has submitted the message. Finally the **'checkHashTag()'** is triggered only if the user wishes to send a message in a form of a Tweet with hashtags found within its body text. The method scans the message and if a "#" character is found, it is then located and the word connected to that character will then be added to a list of mentions where it will be stored within a text file. Each time this method is triggered, it reads the current hash tag file and stores any saved mentions onto a list. If the list contains any mentions, then it will be shown on the screen to the user only if a message has been sent.

## Use Case

The Use case diagram shows the interactions and relationships between the user and the using system in the form of use cases. In this application the user will go through stages to achieve a valid message that can be sent to another user. The diagram also portrays parts of the system that are resemble functionality in the background, these are connected the actor "System"



When the user starts the application, he or she will be directed to the message screen. From there the user can input their ID containing the message type of their choosing. Once the user completed this, the next step would be to fill in the according text boxes such as "To", "Subject" and the main body text. Once all text boxes have been completed, the user can then send the message and the system will configure it's directories based on the message sent. Once the directories have been created by the system, the next step would be to validate the message that was submitted by the user and respond with any mistakes or incorrect details that we're provided

The system validates the message by scanning its contents and adding in features such as abbreviation definitions or removing URLs if needed. All these functions are dictated by the type of message the user has decided on. Depending on the message, the system will take the validated message and store it into a specific directory that stores messages, this also counts to other features such as hashtags that are found within the messages.

Once the message has been finally categorised, the user is the prompted with the message that has been sent.

## Testing

The tests that we're employed for this application were:

- Unit Testing
- Integration Testing
- Acceptance Testing

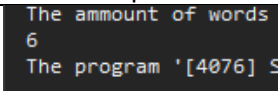
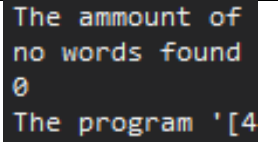
These tests will all be conducted with the white box testing methodology. White box testing consists of the programmer or person who knows the internals of the program to carry out the tests (Software Testing Fundamentals, 2018). It is useful to note that some testing stages will have more tests than others depending on the need and complexity of some functions.

### Unit Testing

Unit testing is a form of white box testing where the programmer him/herself tests the inside methods of a program one by one in order to check for desired outputs (Software Testing Fundamentals, 2018). Tests will be made containing expected input against given input to validate the workings of specific methods.

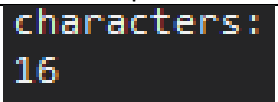
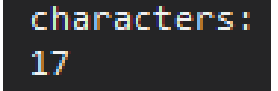
#### **Test 1 (checkWordCount())**

This method has the primary purpose of providing an integer containing the amount of words within a string.

| Test   | Input                     | Expected Output                  | Actual Output   |
|--|---------------------------|----------------------------------|---|
| Check whether the method can return a valid number of words within a string                | "This Is a Piece of Text" | 6                                |  |
| Check whether the method will return the right amount of words when providing a non-string | "23464567564352332533"    | "No words found"<br>Or<br>0/null |  |

#### **Test 2(checkCharCount())**

This method is very similar to the "checkWordCount()" method, but instead of counting words, it counts how many characters are present within a string.

| Test  | Input              | Expected Output | Actual Output   |
|---|--------------------|-----------------|---|
| Check whether the method can return a current character amount when given a typical input | "This should work" | 16              |  |
| Check whether the method will calculate white space as a character                        | "2" "6"            | 17              |  |

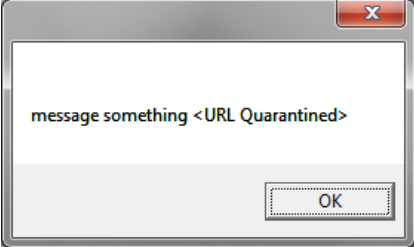
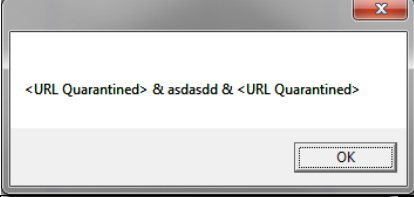
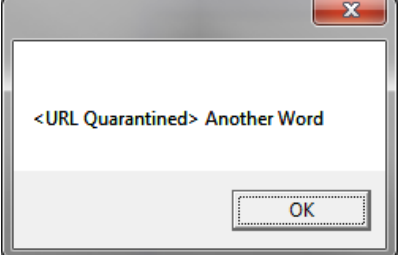
### **Test 3(checkMessageType())**

The "checkMessageType()" method checks for the first character of the userID in order to define which message type the user wishes to send.

| Test  | Input        | Expected Output | Actual Output   |
|---|--------------|-----------------|---|
| Check whether the method can detect the first character of a user ID                              | "T987637456" | "Tweet"         | 'SoftwareDev.exe' (CLR v4.0.30319.1) Tweet                  |
| Check whether the method will output a message saying whether the first ID character is not valid | "F987637456" | Wrong ID        | 'SoftwareDev.exe' (CLR v4.0.30319.1) Incorrect Message Type |
| Check whether the method will output a error message if the first character is a number           | "1987637456" | Wrong ID        | 'SoftwareDev.exe' (CLR v4.0.30319.1) Incorrect Message Type |

### **Test 4(hyperlinkCheck())**

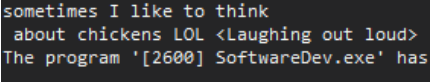
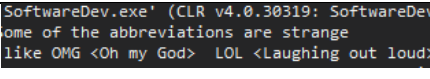
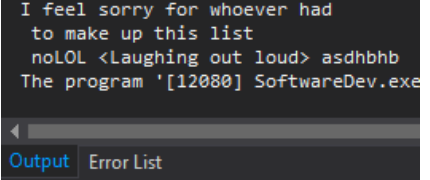
This method is designed to locate a hyperlink within an email and then proceed to remove them while replacing it with a <Quarantined URL> label. The method checks for the usual "www." Or "http" labels within each word of the main text in order to remove all links

| Test  | Input   | Expected Output         | Actual Output  |
|---|---|-------------------------|--|
| Check whether the method can detect a simple URL and replace it | "message something www.Something.com"           | Words + Quarantined     |  |
| Check whether the method will replace more than one URL         | "www.Something.com & asdasdd & www.youtube.com" | Words + URL Quarantined |  |
| Check if the method can detect a URL within a message           | "1sadsadassdwww.saesom.com6 Another Word"       | URL Quarantined + Words |  |



### Test 5(wordTranslate())

This method has a simple task of finding a word that is listed within the abbreviations list and Then providing a description of the word next to it.

| Test   | Input  | Expected Output   | Actual Output  |
|--|--|---|--|
| Check whether the method can detect a common abbreviation                        | "sometimes I like to think about chickens LOL"                   | Words + Abbreviation  |  |
| Check whether the method will assign descriptions for more than one abbreviation | "Some of the abbreviations are strange like OMG LOL so 2012"     | Words + 2 abbreviations   |  |
| Check if the method can find abbreviations hidden within concatenated text       | "I feel sorry for whoever had to make up this list noLOLasdhhbb" | Words + abbreviation + Words while the abbreviation is within a concatenated string |  |

## Integration Testing

Integration testing is the process of testing or validating the output of 2 or more methods that work together in order to generate an output (Software Testing Fundamentals, 2018). In this case both internal and UI methods can be tested together to see whether a more complex algorithm is working correctly while making sure the output is up to standards.

### Test 1(wordTranslate() & hyperlinkCheck())

These methods we're tested separately but are used together when the user wishes to create an Email. If so, the system needs to check for both abbreviations and URLs in order to send the functional email.

#### **Test:**

Check if both methods work together to output a sorted message from the main body text

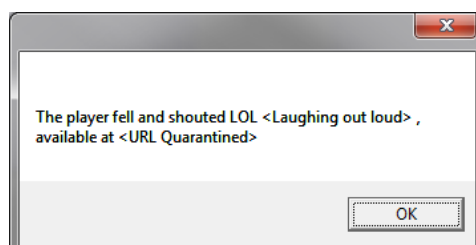
#### **Input:**

"The player fell and shouted LOL, available at www.notavirus.com"

#### **Expected Output:**

Text + abbreviation + Text + Quarantined URL

#### **Actual Output:**



**Test:**

Check if the output test will delete a URL with an abbreviation inside of it.

**Input:**

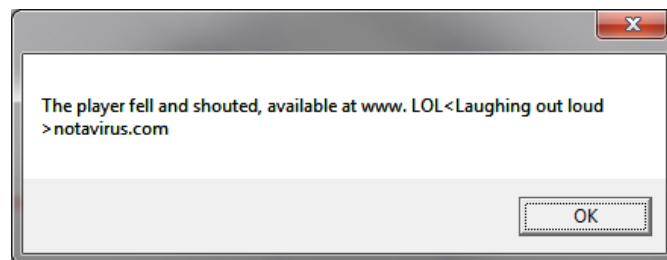
"The player fell and shouted, available at www.LOLnotavirus.com"

**Expected Output:**

Text + abbreviation + Text + Quarantined URL

**Actual Output:**

(Fail)

**Test 2(checkID() & checkMessageType())**

These 2 methods are used by the application to fully validate the user ID. Once the user enters the ID containing the message type and the 9-figure code, the field will be validated by these two methods and will dictate whether the user can continue to send a message or not.

**Test:**

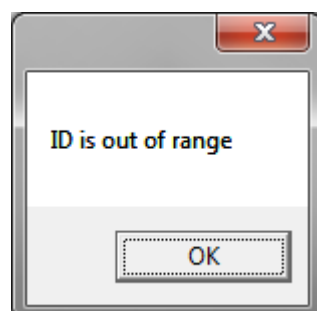
Check if the system can validate an ID that is out of range whilst containing a valid message type

**Input:**

"E12345678910"

**Expected Output:**

Message saying that the ID is not within range

**Actual Output:**

**Test:**

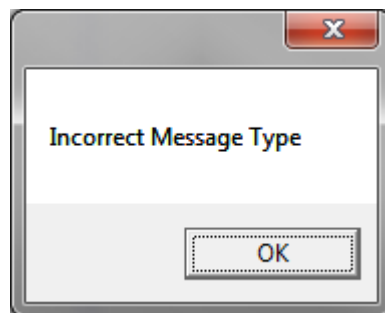
Check if the system can validate an ID that has a valid ID number but an incorrect message type

**Input:**

"G123456789"

**Expected Output:**

Message saying that the ID type is invalid

**Actual Output:****Test:**

Check if the system can validate an incorrect ID number and an incorrect message type

**Input:**

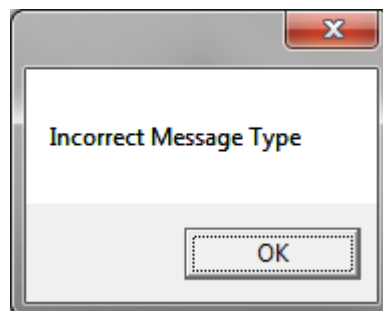
"A123456789344323423"

**Expected Output:**

Message saying that the ID type and ID number is invalid

**Actual Output:**

(Fail)



## Acceptance Testing

Acceptance testing is a form of testing where the system is tested for 'Acceptability'. This form of testing relates to the requirements specification and in doing so, results in wider tests containing more functionality than the other testing strategies. These test results will be checked whether they match up with the requirement specification and check whether the application is suitable for shipping (Software Testing Fundamentals, 2018)

### Test 1 Checking the output of a message

#### Test:

This test is performed in order to make sure that a message is validated and stored within a created directory

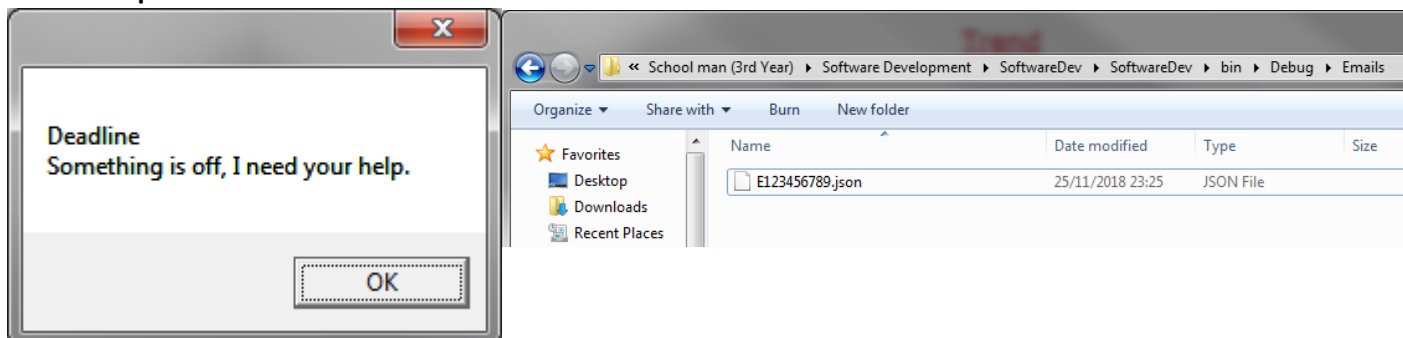
#### Input:

The screenshot shows a web application window titled 'MainWindow'. The header features the 'Edinburgh Napier UNIVERSITY' logo. The main content area is divided into two sections: 'Message' on the left and 'Trend' on the right. The 'Message' section contains a form with the following fields: 'ID' (containing 'E123456789'), 'To' (containing 'someone@gmail.com'), 'Subject' (containing 'Deadline'), and 'Message' (containing 'Something is off, I need your help.'). A red 'Submit' button is located at the bottom right of the form. The 'Trend' section is currently empty. The bottom right corner of the window displays the Edinburgh Napier University logo.

#### Expected Output:

Message saying that the ID is not within range

#### Actual Output:



**Test:**

This test is performed in order to check whether hashtag trends are stored and outputted to the user

**Input:**

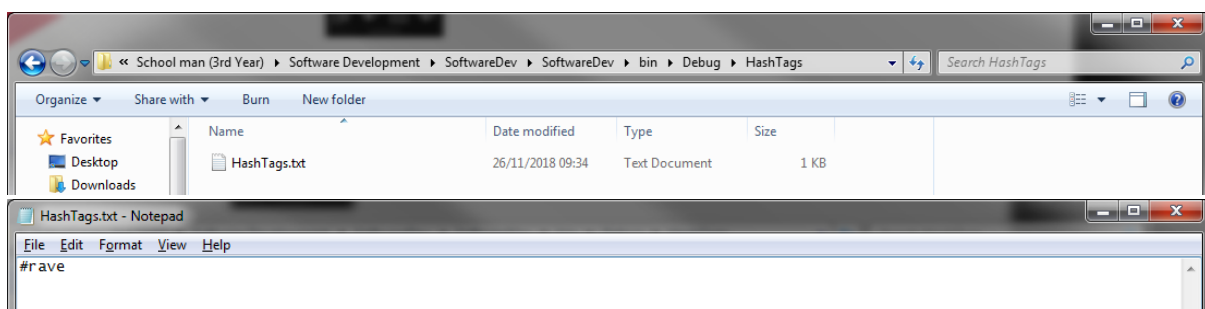
"T444445555" + To text + Message + "#rave"

**Expected Output:**

A trend box showing the "#rave" hashtag after its used

**Actual Output:**

The screenshot shows a web application window titled 'MainWindow'. The header features the 'Edinburgh Napier UNIVERSITY' logo. The main content area is divided into two sections: 'Message' on the left and 'Trend' on the right. The 'Message' section contains a form with the following fields: 'ID' (containing 'T444445555'), 'To' (containing '@somebody'), 'Subject' (empty), and 'Message' (containing 'sometimes i feel like i want to dance #rave'). A red 'Submit' button is located at the bottom right of the message form. The 'Trend' section contains a box with the text '#rave'. The bottom right corner of the window features the Edinburgh Napier University logo.



## Evolution

As the current messaging system comprises of 3 message types, it can be said that it could last a couple of years until another popular way of messaging arrives and beats the current ones. For now, the application would benefit most from UI updates as it is very basic and here I say it, too basic. The application does inherit the right colour schemes as they relate to the Napier's colour pallet and have a very basic Interface. In the future the application would greatly benefit from more navigation options such as having multiple pages with more control options, this may include adding a list of blocked users or a simple menu that lets people browse for previous sent messages.

It would be greatly beneficial to the user if the application contained separate pages for each message type. This would allow for a login feature to be implemented to ensure that the current application options are tailored to a specific user. This feature is also important as it allows each user to have his/her own identity and provide security for their account. If the user wants to send a simple email through the application, there would be no need of having a hashtags trend box as that is unnecessary for the user and might find annoying at times. Simple text editing features such as **bold** or underline options would be useful for users when writing more dynamic messages. Functionality such as sorting messages would be very useful when browsing through past entries or finding a specific message as this could save time and sanity depending on the amount of sent or received messages.

As the current application is very basic and only allows for local message transfer, introducing a database for this application would allow for easier message transfers and would provide a central storage for all entries. The database can be manually updated with information that notifies all users of any alerts or notices such as an "Application will be down for maintenance at 4pm" message which can appear on the main menu. A database will also allow the admins to provide privileges to different users by changing their current accounts as some users may be given restrictions or penalties. Unlike other styles of storage, a database acts as a central storage, meaning security will not be that hard to implement as security software is would be much easier to install.

If, however the database plan decides to pull through, maintaining it would take the most effort as it will change as the application is added with more features, permissions and user functionality. Maintaining these servers can be costly and can either be hosted locally or externally by renting, however if hosted locally, usual checks and tests must be applied to ensure the safe running of the servers. Peer to Peer networks do have their benefits but depending on the size of the company and its user base, a database would prove to be more beneficial. Buying a SQL database service from Microsoft would be considered a good choice if you wish to host externally with security already taken care of. An average pay for a very good performance database will cost around \$10,903 monthly (Single Database, 24 vCore, Gen 5, Business Critical). Or a basic database would cost something around \$1,300 a month (Single Database, 8 vCore, Gen 4, General Purpose). The Microsoft azure website allows the buyer to alter these prices depending on his/her needs. (Microsoft Azure, 2018).

As the application evolves, an options menu would be created for its users to customize the application to their liking. Options can consist of changing the colour scheme of the application or presenting a different UI configuration/size. Options can include a logout button, Sign in with different account and/or view profile. These features will all be simple to implement so long as there is a central database and the application allows for more than one page to be accessed.

## References

Microsoft Azure. 2018. Pricing Calculator | Microsoft Azure. [ONLINE] Available at: <https://azure.microsoft.com/en-us/pricing/calculator/?service=sql-database>. [Accessed 26 November 2018].

Software Testing Fundamentals. 2018. Acceptance Testing - Software Testing Fundamentals. [ONLINE] Available at: <http://softwaretestingfundamentals.com/acceptance-testing/>. [Accessed 26 November 2018].

Software Testing Fundamentals. 2018. Integration Testing - Software Testing Fundamentals. [ONLINE] Available at: <http://softwaretestingfundamentals.com/integration-testing/>. [Accessed 26 November 2018].

Software Testing Fundamentals. 2018. Unit Testing - Software Testing Fundamentals. [ONLINE] Available at: <http://softwaretestingfundamentals.com/unit-testing/>. [Accessed 26 November 2018].

Software Testing Fundamentals. 2018. White Box Testing - Software Testing Fundamentals. [ONLINE] Available at: <http://softwaretestingfundamentals.com/white-box-testing/>. [Accessed 26 November 2018].