**Software Engineering**

**Software Development Report**

*Aaron Campbell*

Table of Contents

[Requirement Specification 2](#_Toc24370570)

[User Stories 2](#_Toc24370571)

[Brief Descriptions of Use Cases 2](#_Toc24370572)

[**Actors** 2](#_Toc24370573)

[**Requirements** 2](#_Toc24370574)

[Detailed Use Case Descriptions 2](#_Toc24370575)

[**Add Message** 3](#_Toc24370576)

[**Determine Type** 3](#_Toc24370577)

[**Validate Message** 3](#_Toc24370578)

[**Write Message** 4](#_Toc24370579)

[**Display Lists** 4](#_Toc24370580)

[**Display Message List** 4](#_Toc24370581)

[Use Case Diagram 5](#_Toc24370582)

[Class Diagram 6](#_Toc24370583)

[Implementation 7](#_Toc24370584)

[Testing 7](#_Toc24370585)

[Testing Strategy 7](#_Toc24370586)

[Test Plan 7](#_Toc24370587)

[Version Control 7](#_Toc24370588)

[Evolution 7](#_Toc24370589)

[Database Implementation 7](#_Toc24370590)

[Read Messages Directly 7](#_Toc24370591)

[Overall 8](#_Toc24370592)

# **Introduction**

As outlined in the brief, the aim of this coursework is to intake, categorise, validate and sanitise messages for Euston Leisure Centre. The messages will take the form of SMS messages, Tweets and Emails. The purpose of this report is to:

* Produce a requirement analysis of the problem.
* Produce a class diagram that illustrates the classes required to match these requirements.
* Outline, perform and analyse the detail of the tests required for the program.
* Propose a version control plan.
* Prepare an evolution strategy for the system.

# **Requirement Specification**

## **User Stories**

1. As a user, I want to be able to add a message to the messaging service.
2. As a user, I want my message to have its type determined and be validated according to its type.
3. As a user, I want to be able to view the hashtag list, mentions list and significant incidents reports list.
4. As a user, I want to be able to view every message added to the system.
5. As Euston Leisure Centre, I want hyperlinks to be added to a quarantine list.
6. As Euston Leisure Centre, I want text speak abbreviations to be expanded to their full meaning.

## **Brief Descriptions of Use Cases**

### **Actors**

1. User – The user of the system i.e. the person who is inputting messages.
2. System – The program itself, and how it works i.e. how it validates and sanitises messages.
3. UI – The graphical interface of the program.
4. File – Where the data is written to and read from.

### **Requirements**

1. Add Message – Allow the user to enter a message to the system. This will take the form of two text boxes – a message header and a message body text box – and a send button.
2. Determine Type – The system determines the type of message the user sent. This is an important requirement, as each message type will have a different body format i.e. a sender and a body for an SMS, and a sender, subject and body for email.
3. Validate Message – The system validates the message based on its type, to ensure that it is a valid message for its type i.e. a significant incident report will have a very different body to an SMS, and both must be validated differently.
4. Write Message – The system writes the message to a list, and the list is then written to a file in JSON format.
5. Display Lists – The user is shown the trending list, the list on mentions and the SIR list at the end of every input session.
6. Display Message List – The user is shown a list of every message ever added to the system.

## **Detailed Use Case Descriptions**

Now that the use cases have been identified, they can be analysed further so that they are implementable and so it is certain that the envisioned implementation of the project is viable. Detailed use cases are described by Cockburn as:

*“A fully dressed use case is written with one of the full templates, identifying factors, scope, level, trigger condition, precondition, and all the rest of the template header information, plus project annotation information.”* [1]

### **Add Message**

**Name:** Add Message

**Description:** Allow the user to enter a message to the system. This will take the form of two text boxes – a message header and a message body text box – and a send button.

**Actors:** User

**Trigger:** User hits the ‘add message’ button.

**Pre-conditions:** User navigated to the ‘add message’ form.

**Post-conditions**: The user should have successfully sent a message, which will then have its type determined.

**Flow of Events:**

*Normal Flow*

1. User navigates to the ‘Add Message’ form.
2. User enters a valid message header i.e. “S123456789” and a valid message body i.e. “+44 5756 5655, test message”.
3. User hits the ‘send message’ button and the message’s type is then determined.

### **Determine Type**

**Name:** Determine Type

**Description:** – The system determines the type of message the user sent. This is an important requirement, as each message type will have a different body format i.e. a sender and a body for an SMS, and a sender, subject and body for email.

**Actors:** System

**Trigger:** User sends a message.

**Pre-conditions:** A message has been sent.

**Post-conditions**: The message’s type has been determined.

**Flow of Events:**

*Normal Flow*

1. User adds a message.
2. System determines the message type i.e. checks that the message header is valid and which letter it starts with.
3. The message is then validated depending on its type.

### **Validate Message**

**Name:** Validate Message

**Description:** – The system validates the message based on its type, to ensure that it is a valid message for its type i.e. a significant incident report will have a very different body to an SMS, and both must be validated differently.

**Actors:** System

**Trigger:** User sends a message and its type is determined

**Pre-conditions:** The message header was valid.

**Post-conditions**: The message has been valid, with its type determined.

**Flow of Events:**

*Normal Flow*

1. System reads the message body.
2. The body is validated depending on which message type the message was determined to be.
3. The message body is validated and written to a file along with its header and type.

### **Write Message**

**Name:** Write Message

**Description:** – The system writes the message to a list, and the list is then written to a file in JSON format.

**Actors:** System, File

**Trigger:** A valid message is written.

**Pre-conditions:** The message was valid.

**Post-conditions**: The message has been written to a file containing all other valid messages.

**Flow of Events:**

*Normal Flow*

1. The existing file is deserialised to a list
2. The valid message is written to the list.
3. The list is serialised in JSON.
4. The JSON object is written to a file.

### **Display Lists**

**Name:** Display Lists

**Description:** – The user is shown the trending list, the list on mentions and the SIR list at the end of every input session.

**Actors:** System, UI

**Trigger:** User ends the input session.

**Pre-conditions:** The files exist.

**Post-conditions**: The lists are displayed to the user.

**Flow of Events:**

*Normal Flow*

1. The user ends the input session.
2. The list containing all hashtags is populated from the hashtag file, and is grouped by count.
3. The list containing all mentions is populated from the mentions file.
4. The list containing all SIRs is populated from the SIR file.
5. The lists populate three ListBoxes.

### **Display Message List**

**Name:** Display Message List

**Description:** – The user is shown a list of every message ever added to the system.

**Actors:** System, UI

**Trigger:** User navigates to the message list form.

**Pre-conditions:** The file exists.

**Post-conditions**: The full list of messages is displayed to the user.

**Flow of Events:**

*Normal Flow*

1. The user navigates to the message list form.
2. The messages are displayed by ID in a ListBox.
3. The user clicks on a message.
4. The message’s details are displayed in textboxes, similar to the add message form.

## **Use Case Diagram**

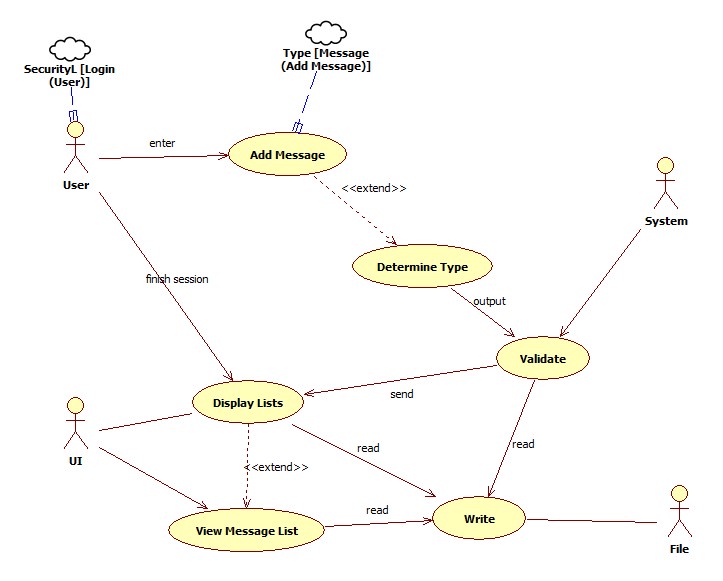


Figure 1 – Use Case Diagram for Euston Leisure Messaging service.

The top level use case diagram can give a good overview of what the user will expect to happen in the system and what features are included in the system. The diagram does a good job of illustrating how the use cases are linked, and the general behaviour of the system.

The first use case is Add Message. This is entered by the user, as in the user fills out the text boxes on the add message form and sends it using the add message button. This extends Determine Type, which will determine the type of message sent at the xaml level by looking at the first character.

The next use case is Validate. The system gets the header and body from the Determine Type use case, and validates them according to their type. This will take place inside of the class related to the type of message that was sent. If the message is invalid, an error will be displayed to the user. Messages will be validation according to their type i.e. an email may have up to 5 items (for a Significant Incident Report) in its body and has a header which in the format email@email.com while a Tweet has 3 items and a sender in the format of a Twitter handle.

If the message is valid, it will be passed to the Write use case. This use case will add the message to a list, serialise the list to JSON and write the JSON object to a file. One feature of the Validate use case is that it checks every message for specific items related to their message type. For example, a Tweet must be checked for the occurrence of hashtags or mentions, and these must be written to a file. Both Tweets and SMS’ must be checked for textspeak abbreviations and expanded to their full meaning in the format “LOL <Laughing Out Loud>”. Emails must be checked for URLs, which will be added to a quarantine list and be replaced in the message body by “<URL Quarantined>”. These must all be written to a file on occurrence.

When the user finishes the input session, they will be taken to the Display Lists use case. This displays the list of hashtags, mentions and significant incident reports – all written to a .txt file every time one occurs in a message. These will be read from a file and displayed to the user. This use case extends View Message List, which works in a similar fashion. The JSON file containing all of the messages written to the system will be deserialised and displayed to the user.

## **Non Functional Requirements**

Non-functional requirements are requirements that are not instrumental to the actual function of the system. There are many common non-functional requirements that are present in most systems, such as reliability, scalability and security. While these requirements do not refer to the actual processes of the system, they can be just as instrumental. For example, in this program we can broadly refer to its non-functional requirements as:

* Reliability – the program must deliver the intended result in the same way each time, so as to ensure each message is processed correctly.
* Usability – the program must have a good ease of use, and should handle errors rather than crash so as to ensure the staff of Euston Leisure Centre are able to use it effectively.
* Scalability – the program must be able to take in multiple input sessions, and must be persistent.
* Security – the program’s data should be safe and secure.

I also identified these two non-functional requirements:

1. Scalability of users.
2. Scalability of message types.

Generally, these requirements are extremely important in terms of how the program will work, despite not having much to do with how the program will be designed functionally/technically. It is good practice to keep these requirements in mind during the implementation of the system so as to ensure they are met at each step and to ensure the program’s general usability.

# **Class Diagram**



Figure 2 – Class Diagram for Euston Leisure Service.

The class diagram is the most commonly used UML diagram, and one of the most useful. Classes describe the type of objects that make up the system and their relationships. As Miles and Hamilton describe them, “Classes do not live in a vacuum – they work together using different types of relationships.” [2] From this diagram, we can start to see how the system will work.

The most important class is the Message class – this contains the messageType, header and body of every Message object, along with a constructor. This class is inherited by three classes – SMS, Tweet and Email. These classes are very similar to the Message class, with some additional properties containing the Regex objects that will be used to validate the body of each message type. As validation will happen at the class level in the mutator methods, these Regex objects will be matched against various parts of the body to ensure that it is in a valid format for its message type.

The DataManager class represents the storage of the Message objects, in the format of a List of Messages. It also contains the methods that are called when a new Message is added – newSMS, newTweet and newEmail. These methods will take in SMS, Tweet or Email objects and validate them at the class level. If the Message is valid, it will then be added to the messageList.

The MsgForm.xaml.cs class represents the form containing the two textboxes (Header and Body). It also has a WriteList() method, which is where every valid Message will be appended to the list. The method will work as follows – the file containing the existing list will be deserialised to a List. The message will then be added to the List. The List will then be serialised and will overwrite the existing file.

There are two other classes in the diagram – FinishedScreen.xaml.cs and ViewMessages.xaml.cs. These classes represent the screen that displays the hashtag list/mention list/SIR list and the form that shows all of the messages added to the system respectively. The FinishedScreen class has a DisplayLists() method, which will read the individual files and display them in textboxes. The ViewMessages class has a loadLists() method, which will read in the Message file and display their ID in a listbox. When the user clicks on a Message, its respective information will be displayed in a textbox.

# **Implementation**

The implemented program is very similar to the system design outlined in the Use Case and Class diagrams. I took a three-tiered architectural approach – I have a Presentation layer, a Business layer and a Data layer. The presentation layer contains all forms implemented in the system, which will be outlined in further detail below in this section. The Business layer contains all of the entities present in the system - in this program, this describes the Message class and its subclasses SMS, Tweet and Email. The Data layer contains our DataManager, which contains the Message list and the methods that add the various types of Message to the list. I will describe these methods and how these layers interact below in the form of images.

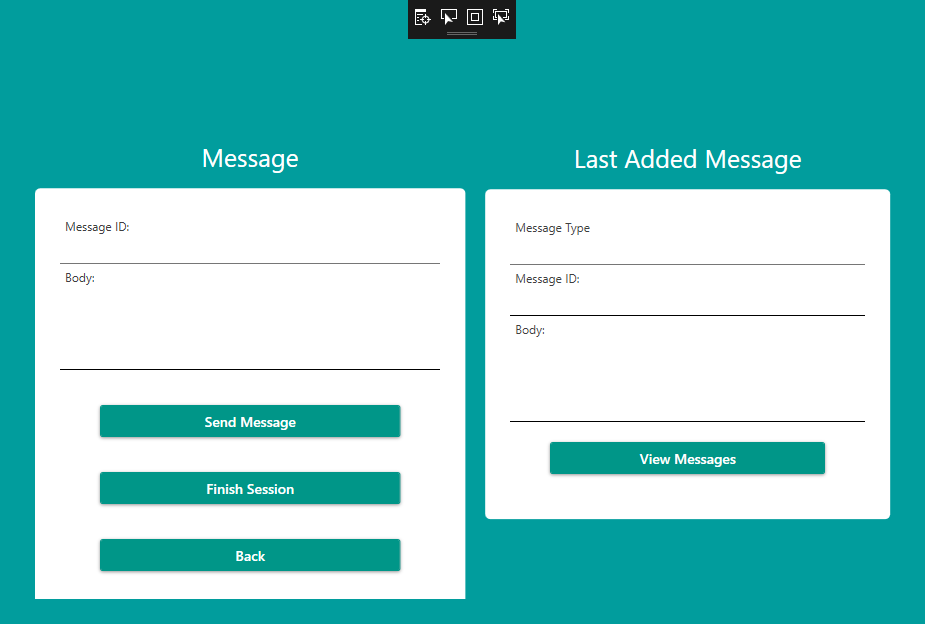


Figure 3 – The MsgForm xaml. Contains an add message form, and a last added message form. Allows the user to finish the session, and to view the list of messages added to the system.

This is the MsgForm i.e. the Add Message form. There is a form on the left containing two textboxes, a header textbox and a body textbox, conforming to the specification in the brief. There is also a Send Message button, which will call the onClick method and attempt to add the message inputted by the user to the system. In addition, there is a Finish Session button which will take the user to the Finished Screen (Figure 5).

On the right there is another form, with three textboxes. In addition to the header and body textboxes, there is a message type textbox. This form will display the message most recently added to the system in the user’s input session. There is also a view messages button, which will take the user to the View Messages form, where they will be able to view every message the user has added to the system.

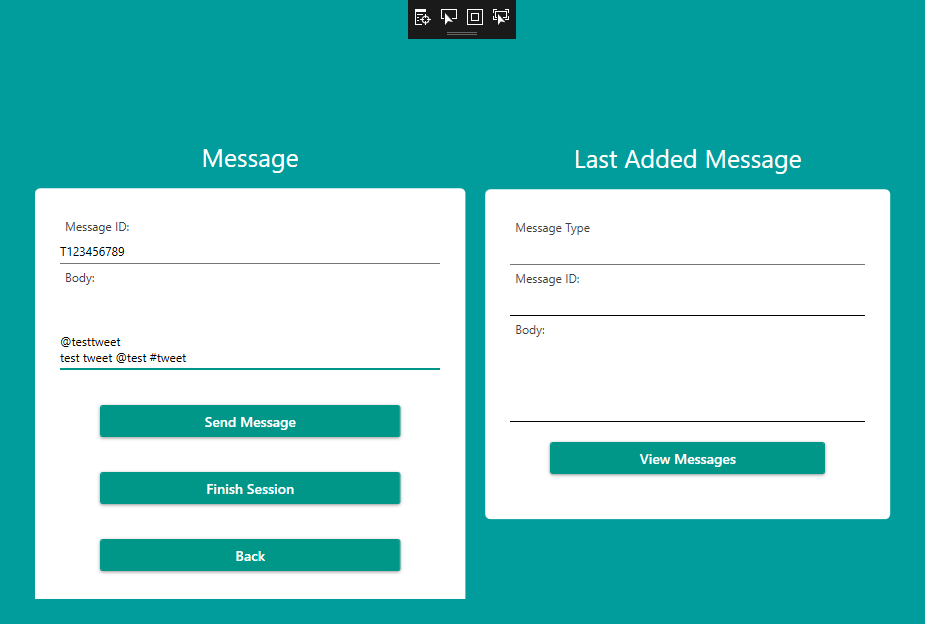
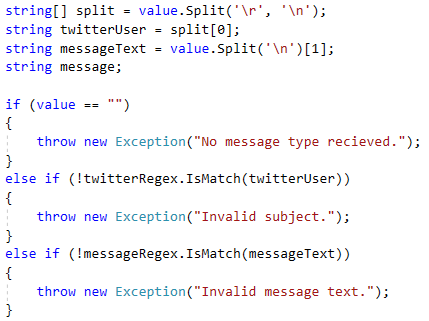


Figure 4 – A preview of a message added to the system. If the tweet is validated correctly, the tweet should be added to a message list, with its mention and hashtag added to their respective lists and displayed on the Finish Session window.

This image shows a typical input format for a Tweet containing a hashtag and a mention. The message ID conforms to the required format – a T followed by 9 numbers. The sender is in the correct format – a Twitter handle. The message body is also in the correct format - body containing a mention and a hashtag which must be recognised. If our system works correctly, the message must be validated and added to our Message list, with the hashtag and mention added to their respective lists and displayed on the Finish Session screen.

The system will take the first char of the message header as a variable, and check if it is a valid letter for the start of a Message ID. In this case, it is a valid letter and so a new Tweet object will be created. The Tweet’s message type will be set to Tweet, its Header will be set to the value of the header textbox and its body will be set to the value of the body textbox. These properties will then be validated inside of the Tweet class’ mutator methods. The Header validation takes the form of a simple Regex pattern. The Body validation will split the value passed to the property by a newline character, so that it splits the sender from the message body. This way, it is simple to match the sender to a Regex pattern and to match the message body to another pattern. I also split the words inside of the body by a space, so that it saves every word in the body as an array of strings. I then do for loops over each word, checking if they match the pattern for a hashtag or a mention, and adding them to a list if they do. The hashtag and mentions list will be written to a file, and the value will be set to the sender appended to the body. I have presented the string split and the validation in further detail below:



1

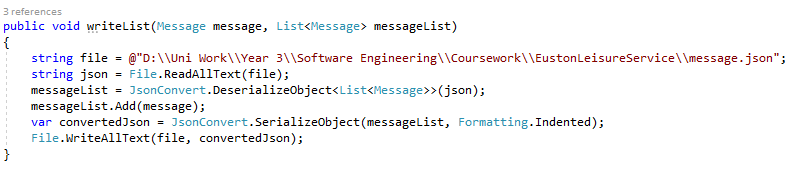
2

3

1 – This is where the value passed to the body is split. It is split by a newline, as this was seen as the most efficient way to take in the input of the user. The first value in the split is set to twitterUser i.e. the sender. messageText is set to the first value after the newline character, and so contains everything on the line under the sender i.e. the message body. Note that this is relatively simple for a Tweet, and takes on a more complex form when adding a message such as a significant incident report. In this case, there are only two lines in the body textbox and only two lines will have to be validated.

2 – twitterUser must match the twitterRegex (^@?(\w){1,15}$). This takes in an @ followed by any 15 characters.

3 – messageText must match the messageRegex(^[ a-zA-Z#@0-9]{1,140}$). This takes in any 140 characters.



When the message has been validated, it will be passed to the addTweet method inside of the dataManager class. This simply takes the newTweet object and adds it to the list of Messages. It will then be passed to the writeList() method inside of the msgForm class, which takes the existing message file and deserialises it to the Message list. When it has been deserialised, it will add the new Tweet to the list and re-serialise and write the JSON object to the file. This ensures that the list will never be overwritten, and the new Message will always be appended to the end of the file.

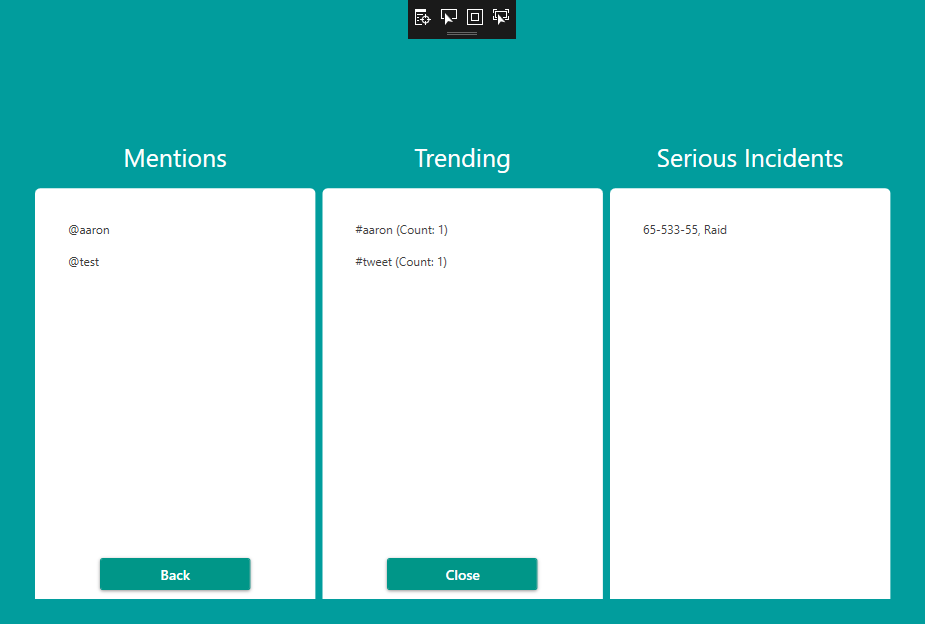


Figure 5 – The Finish Input screen. Shows the list of mentions, the trending list and the list of serious incidents. Note that our tweet was validated properly, and our mention and hashtag is now in the list.

This image shows the Finished Screen, which displays the mentions and trending list. As we can see, our message has been processed correctly and our hashtag and mention are being displayed in the listboxes. The FinishedScreen class has a displayLists() method, which is called in the constructor method and populates the mentions list by reading the mentions file toList(). The method uses a foreach loop to iterate over every item in the list, so it can display the items one by one.

It works similarly for hashtags, however hashtags are sorted using the GroupBy method to get the count of the hashtags and display them in descending order.

The serious incidents listbox also works in a similar fashion, however it splits the sir file by a newline character so that the incidents have both their code and SiR type displayed. The back button takes the user back to the add message form, and the close button closes the program.

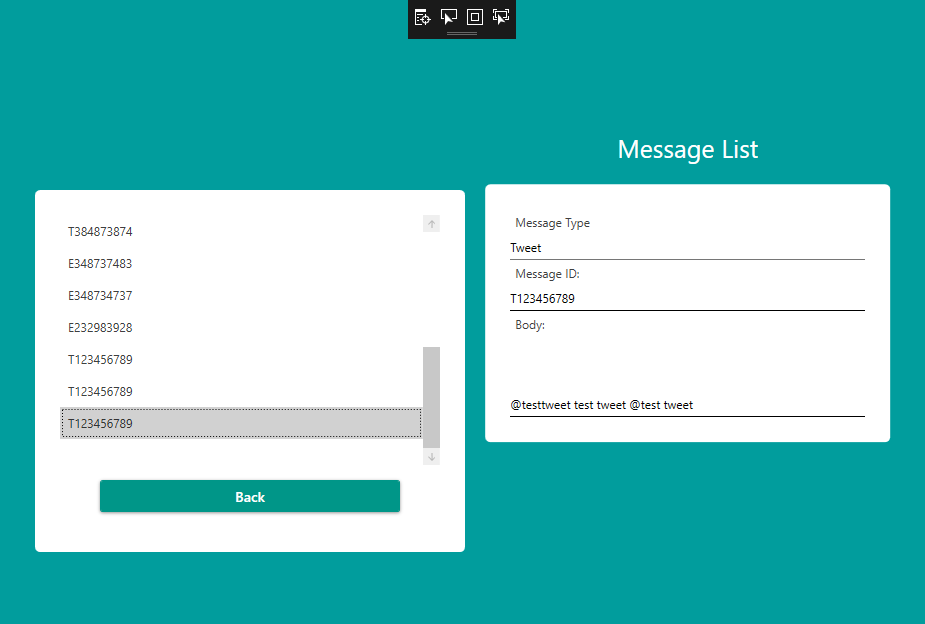
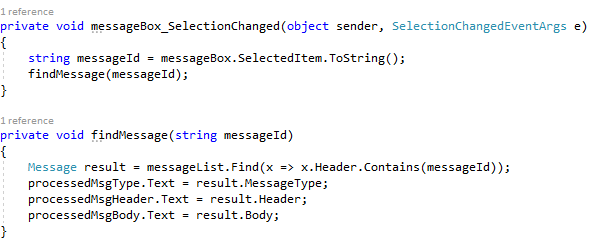


Figure 6 – The View Messages screen. This displays the ID of every tweet in a listbox, and displays its information in a form when it is clicked. This shows the last message added – the tweet we added, along with its type, header and body.

This image shows the View Message screen, where the user is taken when they click the View Messages button on the Add Message screen (Figure 3). This screen displays the ID of every message added to the system, and when a message is clicked it has its details written to the form on the right.



This class reads in the message file, and deserialises it to a JSON object. It then uses JsonConvert, a method included with the NewtonSoft.Json API [3] to save the object to a list. The message are then added to the listbox on the left in the typical fashion, with their ID displayed to the user.



When the user clicks on an item in the listbox, a SelectionChanged method is used to get the selected item. As this is the ID of the message, this can be used to find the message’s corresponding information. I passed the ID to a method called findMessage. This method takes in takes in the message ID and looks for the message that contains it, then sets this to a message called result. Result’s corresponding information is then set to the information inside of the text boxes.

# **Testing**

## **Testing Strategy and Plan**

Errors are a natural part of software development. Testing is the process of executing a program with the intent of finding an error. Defect testing refers to confirming the presence of errors, while debugging refers to locating and repairing them.

Testing requires test cases, data and results. Cases refer to the tests themselves, data refers to the data tested against the method and the result is the outcome of the test. Generally, there are some testing principles that should be kept in mind throughout the process:

All tests should be traceable to customer requirements

Tests should be planned long before testing begins

Exhaustive testing is not possible

From this, I developed a test plan derived from the lecture slides.

### **Objectives and Scope**

The objectives of these tests are to ensure that the requirements of the system have been met, and that the program works in the expected way. Another aim of this testing is to uncover previously unknown bugs. The majority of my tests have been developed from the requirements in the specification i.e. checking a message header length, checking a sender format.

### **Test Items, Tasks and Deliverables**

|  |  |  |
| --- | --- | --- |
| **Class** | **Test Items** | **Deliverables** |
| SMS.cs | Header, Body | Validation working as expected |
| Email.cs | Header, Body | Validation working as expected |
| Tweet.cs | Header, Body | Validation working as expected |
| DataManager.cs | addEmail, addSMS, addTweet | Methods working as expected. |

### **Testing Approaches**

Generally, we will be testing the program with a strong knowledge of the system’s internal design. This is known as white box testing.

I wrote my tests before writing my code, and then wrote code that passes this tests. This approach in known as test-driven development, combining testing and code development. Code is developed incrementally, and you don’t move onto the next requirement until the code that has been developed passes its test. A major benefit of this approach is its strong code coverage – in theory, every code segment that has been written will have at least one associated test.

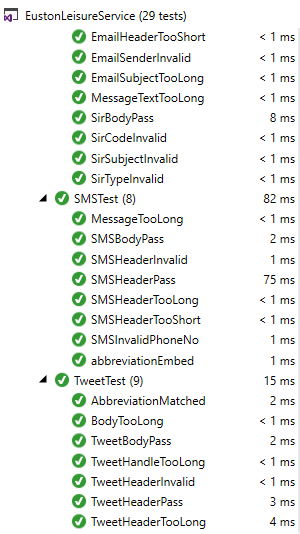
### **Environmental Needs**

As this program was developed using C# in Visual Studio, I used the Visual Studio Unit Testing Framework. I mostly used Assertions to test the behaviour of my validation, checking that the exceptions were thrown for invalid values and that valid values passed.

## **Test Cases**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Test Class** | **Test Method** | **Input** | **Expected Output** | **Reason for Test** |
| **1** | **Email** | EmailHeaderPass() | Tweet.Header = “T123456789**”** | Accepted | Check that the setter accepts a valid header value. |
| **2** | **Email** | EmailHeaderInvalid() | email.Header = "S123456789 | Exception thrown | Check that the setter rejects an invalid header value. |
| **3** | **Email** | EmailHeaderTooLong() | email.Header = "E1234567891" | Exception thrown | Check that the setter rejects a header value over 15 characters |
| **4** | **Email** | EmailHeaderTooShort() | email.Header = "E12345678" | Exception Thrown | Check that the setter rejects a header value less than 10 characters |
| **5** | **Email** | EmailBodyPass() | email.Body = "test@gmail.com\r\nsubject\r\nbody" | Accepted | Check that the setter accepts a valid body value |
| **6** | **Email** | EmailSenderInvalid() | email.Body = "testgmail.com\r\nsubject\r\nbody" | Exception thrown | Check that the setter rejects an invalid email address. |
| **7** | **Email** | EmailSubjectTooLong() | email.Body = test@gmail.com\r\nsubjecttttttttttttttt\r\nbody" | Exception thrown | Check that the setter rejects an email subject over 20 characters |
| **8** | **Email** | MessageTextTooLong() | email.Body = “Lorem Impsum…”(1029 characters) | Exception thrown | Check that the setter rejects an email message body over 1028 characters |
| **9** | **Email** | SirBodyPass() | email.Body = "test@gmail.com\r\nSIR 20/20/20\r\n11-111-11\r\nRaid\r\nbody" | Accepted | Check that the setter for aN SIR accepts a valid body |
| **10** | **Email** | SirSubjectInvalid() | email.Body = "test@gmail.com\r\nSIR 20/200/20\r\n11-111-11\r\nRaid\r\nbody" | Exception thrown | Check that the setter for an SIR rejects an invalid email subject i.e. SIR 20/200/20 |
| **11** | **Email** | SirCodeInvalid() | email.Body = "test@gmail.com\r\nSIR 20/20/20\r\n11-111-111\r\nRaid\r\nbody" | Exception thrown | Check that the setter for an SIR rejects an invalid SIR code i.e. 11-111-11 |
| **12** | **Email** | SirTypeInvalid() | "test@gmail.com\r\nSIR 20/20/20\r\n11-111-111\r\nInjury\r\nbody" | Exception thrown | Check that the setter for an SIR rejects an invalid SIR type |
| **13** | **SMS** | SMSHeaderPass() | sms.Header = "S123456789"; | Accepted | Check that the setter accepts a valid header value. |
| **14** | **SMS** | SMSHeaderInvalid() | sms.Header = "E123456789" | Exception thrown | Check that the setter rejects an invalid header value. |
| **15** | **SMS** | SMSHeaderTooLong() | sms.Header = "E1234567891" | Exception thrown | Check that the setter rejects a header value over 15 characters |
| **16** | **SMS** | SMSHeaderTooShort() | sms.Header = "E12345678" | Exception Thrown | Check that the setter rejects a header value less than 10 characters |
| **17** | **SMS** | SMSBodyPass() | sms.Body = "+44 2727 2727\r\nhello" | Accepted | Check that the setter accepts a valid body value |
| **18** | **SMS** | SMSInvalidPhoneNo() | sms.Body = "+54 4444 4444 22 22 222\r\nhello" | Exception thrown | Check that the setter rejects an invalid phone number |
| **19** | **SMS** | MessageTooLong() | sms.Body = "+434 2727 2722\r\nLorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque interdum rutrum sodales. Nullam mattis fermentum libero, non volutpat.." | Exception thrown | Check that the setter rejects a body value over 128 characters |
| **20** | **SMS** | abbreviationEmbed() | sms.Body = "+43 2727 2722\r\nLOL" | Assert.AreEqual(sms.Body, "+43 2727 2722 LOL <Laughing out loud>"); | Check that the setter recognises and expands textspeak abbreviations |
| **21** | **Tweet** | TweetHeaderPass() | tweet.Header = "T123456789"; | Accepted | Check that the setter accepts a valid header value. |
| **22** | **Tweet** | TweetHeaderInvalid() | tweet.Header = "E123456789" | Exception thrown | Check that the setter rejects an invalid header value. |
| **23** | **Tweet** | TweetHeaderTooLong() | tweet.Header = "T1234567891" | Exception thrown | Check that the setter rejects a header value over 15 characters |
| **24** | **Tweet** | TweetHeaderTooShort() | tweet.Header = "T12345678" | Exception Thrown | Check that the setter rejects a header value less than 10 characters |
| **25** | **Tweet** | TweetBodyPass() | tweet.Body = "test\r\nhello" | Accepted | Check that the setter accepts a valid body value |
| **26** | **Tweet** | TweetHandleTooLong() | tweet.Body = "@testtesttesttestes\r\nhello" | Exception Thrown | Check that the setter rejects an invalid Twitter handle |
| **27** | **Tweet** | BodyTooLong() | tweet.Body = "+434 2727 2722\r\nLorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque interdum rutrum sodales. Nullam mattis fermentum libero, non volutpat.." | Exception Thrown | Check that the setter rejects a body above 128 characters |
| **28** | **Tweet** | AbbreviationMatched() | tweet.Body = "@test\r\nLOL"; | Assert.AreEqual(tweet.Body, "@test LOL <Laughing out loud>"); | Check that the setter recognises and expands textspeak abbreviations |

## **Test Analysis**



The above image shows all the test cases detailed above being applied and passing.

In retrospect, my tests could have been more thorough. For example, I don’t check whether the emails have been quarantined when written to an email. I could also have been more thorough in my validation of the message headers i.e. checking whether it accepts 10 letters, or if it would accept a small letter at the start of the value.

It is important to note that testing cannot be wholly comprehensive, and there will likely be a number of bugs that have not been detected in the system. As I used test driven development throughout the development of the program, I would discover logical errors that would have otherwise been undiscovered. For example, I found that I wasn’t fully validating phone number message senders i.e. international phone numbers and that an 18 digit number would be accepted (SMSHeaderTooLong). This turned out to be an issue with my Regex, and when I reworked it the issue was solved and the test would pass.

# **Version Control**

## **Agile Development**

This system would benefit from agile development, in which work is completed in short phases with frequent revision and adaption. Through my work in the Software Engineering Methods module last year, I came to see the benefits of an agile approach. The Agile Manifesto is as follows:

*Individuals and interactions over processes and tools*

*Working software over comprehensive documentation*

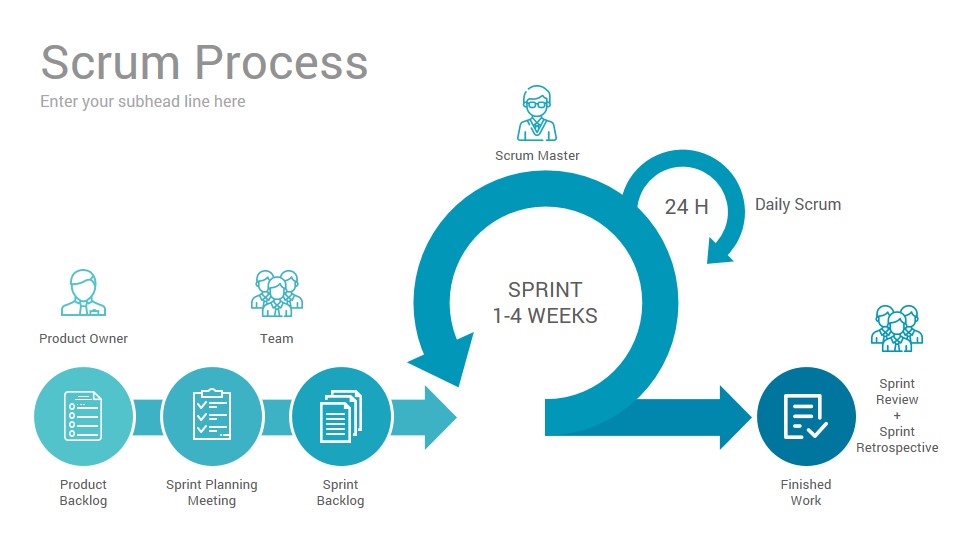
*Customer collaboration over contract negotiation*

*Responding to change over following the plan*

An agile approach is generally the best approach, in the case of this project it would be beneficial as it would allow frequent evolution and improvement of the various problems within the project, along with collaboration among the team and flexibility to change. Agile methods are built on human-centric ideas, and thus are more natural and adaptive to requirements as they change. There is a focus on users, as user stories play a large role in determining and prioritising requirements.

The development of this program would be broken down into small chunks, developed and tested and iterated over by a team, adding to the chunk on each iteration, to ensure that the requirement is met.

## **Scrum**



Scrum is a project management and teamwork philosophy that underpins Agile development. Scrum works in iterations, referred to as Sprints. A Sprint occurs for a fixed length of time – typically a week but can be more or less. The team will work on a set of tasks, and at the end of the sprint will come together to analyse how they performed in terms of completing these tasks. This allows the team to work towards the final goal (the finished program) in iterations, gradually completing tasks and coming closer to the end product. There are a number of scrum methods that can be utilised in our approach to our program, which I have detailed below:

Product Backlog – In Scrum, the list of tasks is identified and prioritised by the Product Owner.

Kanban Board – Tasks are split into four distinct columns (Backlog, Sprint, In Progress and Done).

User Stories – Utilised earlier, this helps the team to understand why a task is required.

Sprints – Detailed above, the period of time when the team does work.

Sprint Reviews – A review/retrospective of the sprint.

Every day, meetings will be held to ensure members are happy of their progress. This promotes collaboration, ensuring no team member is left behind and that every requirement will be met effectively.

## **Version Control**

Version control is the monitoring and management of changes to files. It allows you to store code, track changes and view the state of a file at certain points. There are three types of version control systems:

Localised – Developers must be working on the same file system.

Client-server – Multiple developers can work locally and push changes to a single repository.

Distributed – The complete code base and history is stored by each individual developer, thus replicating the repository for each developer. Developers can collaborate via remote repositories.

A key idea is that we are able to track commits – when a change was made and who made that change. We can also rewind back to a previous commit, if a bug was introduced in a commit. Additionally, there is the idea of branching. This allows for the management of multiple versions of the files.

A common approach is to have a master branch and a develop branch. The master branch must always be a working version of the program. The develop branch is where collaborators can push their progress on the development of requirements. When a requirement is implemented and works completely as required, it can be merged with the master branch. This would be an effective approach to take in terms of this project, as there are many requirements that could be broken down into “chunks” and worked on independently.

### **Version Control Plan**

In terms of this project, I would recommend using Visual Studio Online Basic’s Team Foundation Service. Using this, you can configure the workspace of the project to the source for version control. It is fairly simple to add an existing project to a Team Foundation Server, and so we would simply add it to the source control and check in our current classes.

From there, we would create branches as outlined above. We would have a master branch i.e. the current project, a working version of our system. We would create a develop branch, where any changes would be made. This would be especially useful when implementing the new features described below in the evolution section.

I would recommend using Visual Studio’s built in version control, as it is the sole environment we have and will use for the project and would simplify the development of new features. However, the team could also choose to use GitHub or another similar version control service that uses Git.

# **Evolution**

### **Database Implementation**

There are many ways in which I think this program could be improved upon in future implementations.

One obvious improvement would be to store the list of messages in a database, rather than in a file. This could be achieved through MySQL in PHP MyAdmin, although there are several alternatives. A connection could be made between Visual Studio and the database, and the JSON fields could be easily converted to fields in a MySQL database table. This would be an extreme improvement over the current system – saving the data to a database would keep it in a safe, reliable place rather than simply writing sensitive information to a file. Another advantage of this approach is that it would allow the user to run simple queries that would return items such as the hashtag or significant incident report list, rather than saving this information to a list. It would also allow the user to perform more complex queries, opening up a much greater range of information that could be viewed pertaining to the messages.

The process of moving the information to a database itself would be extremely simple and wouldn’t take very long, as there is ample documentation on how to write your information to a database – especially when using Visual Studio and MySQL/PHP MyAdmin. In terms of maintenance costs, there are a variety of cloud hosting services such as Microsoft Azure or Kametara. There are various pricing plans, although the price for hosting such a simple program wouldn’t be too extreme.

### **Read Messages Directly**

Another major improvement that could be made would be to take in messages directly, rather than having a user enter them manually. There are several ways in which this could be done, however it would likely require implementing some forms of C# API. The most obvious advantage of this is that it strongly reduces the time taken to input each message sent to the leisure centre – taking in the messages directly would be a vast improvement over a user having to enter them one-by-one manually. Another advantage of this is that it reduces the possibility of user error, having the process automated would reduce the risk of having a message typed incorrectly. Another advantage of this is that it would eliminate the need for the system to determine the message type – it would simply validate each message depending on its source. The process for implementing this improvement is detailed below:

* One way in which SMS messages could be read directly through Twilio, a cloud communications platform. Twilio allows you to send and receive text messages through the API, if you have a Twilio number set up. There are a number of Twilio .NET libraries available, and through a TwilioController it would be possible to set up a class that returns every message sent to the number as an object. These objects could then be passed to the SMS class and validated there, and then sent to the database.
* In terms of tweets, there are a number of Twitter APIs associated with .NET – including Twitter’s own. Using TweetSharp or something similar, it would be possible to read in all Tweets sent to the leisure centre’s mentions/notifications and store them as an object. The tweets could then be validated and stored in the database, similar to how tweets would work.
* There are also a number of extensive .NET frameworks that would allow you to read in emails sent to the leisure centre’s email address, such as MailKit and MailSystem.NET. This would work in a similar way to tweets and emails – simply reading in each email as an object and validating/storing the email.

## **Overall**

Maintenance for these improvements would be very simple and cost effective, with the only requirements being that the leisure centre has an email address/Twitter account/mobile phone number set up – which they are already known to have from the current system. The only maintenance costs would be the cost of a cloud hosting service, and the cost of a Twilio phone number ($0.75 per text in the UK).

These improvements would drastically change the system, and largely change its focus from a system that interprets and validates various forms of messages to one that automatically takes in messages and allows the user to analyse them. For example, having the system work in this way would allow the user to run queries to determine which types of significant incidents are happening most often and in which area codes. I think this is a natural evolution of the system, and would provide some very useful functionalities to Euston Leisure Centre.

# Bibliography

|  |  |
| --- | --- |
| [1] | A. Cockburn, Writing Effective Use Cases, 2001. |
| [2] | H. Miles, Learning UML 2.0, 2006. |
| [3] | NetwonSoft, “https://www.newtonsoft.com/json/help/html/T\_Newtonsoft\_Json\_JsonConvert.htm,” [Online]. |