Coursework Report

40280156@live.napier.ac.uk
Edinburgh Napier University – Software Engineering (SET09102)

1 Introduction

The purpose of this coursework is to develop a prototype that will validate, sanitise and categorise incoming messages in the form of SMS, emails and tweets. This report is required to detail:

1. a requirement analysis of the prototype created
2. a class diagram of the system
3. testing of the prototype
4. using version control to support development iteration and team members collaboration
5. an evolution strategy, maintainability of the system, predicted maintenance costs and evolution process and methods

2 Requirement Specification

As Steve McConnell said, the main aim of requirements gathering is helping users figure out what they want, since at the start they will not be completely sure of what the program has to do. This is why during Agile development the team maintains constant contact with the product owner and has to continuously respond to change.

There are two types of requirements: Functional (describe what the system should do) and Non-Functional (describe how the system works).

Functional requirements are services that the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. It may also include what the system should not do.

2.1 Functional Requirements

These requirements should be specifically detailed or they might be interpreted in a different way depending on who is reading them (developer or user).

For the Napier Bank Message Filtering System the functional requirements are as follow:
Read file (.json or .txt) containing messages or manual input of messages from user

- Validate the input and identify the type of message (SMS, Email or Tweet)
- Sanitise the input:
    - detect abbreviations in texts and add their definition after them
    - quarantine URLs and add them to a list
    - add twitter mentions to a 'mentions' list
    - add twitter hashtags to a 'hashtag' list
    - identify Significant Incident Report emails and add sort code and incident to a SIR list
- Categorise the input: SMS, Email or Tweet
- Output the processed messages to a JSON file

These functional requirements are what then will be the use cases, which are also the user stories, even though the client will not know the technical details that each part requires.

2.2 Non Functional requirements

Non-functional requirements are constrains such as timing, development processes... These often apply to the system as a whole, not to just one feature or service.

The main non-functional requirements that the NBMFS must provide are:

- reliability
- efficiency
- storage requirements must be more than enough to cope with all the data, the system must have scalability
- security

The security requirement generates a number of related functional requirements that define system services that are required. A single non-functional requirements has the power to create many functional requirements.

These requirements may be more critical than functional requirements since if these are not met, the system could be useless. They can be divided into product requirements, which specifies the way in which the delivered product should behave, organizational, this requirements are a consequence of policies and procedures of the organisation, or external requirements, which arise from legislative requirements or interoperability requirements.

The product requirements for the NBMFS are:

- **usability**: the program must be easy to learn and easy to train the staff that will be using the program, user friendly and should have error tolerance, the program should never crash, if any it should display the error to tell the user that something is wrong but never crash the entire system
- **robustness**: if there is an error that crashes the whole program, the system should not take long to restart after the failure
- **efficiency**: the system must not occupy a lot of space and should pass the performance test
- **dependability**: making sure the program delivers the expected outcome on time and detailed
- **security**: the program's data should not be able to be accessible to anyone who is not authorised to see or manipulate said data

The organisational requirements includes environmental, operational and development requirements. In the NBMFS the users will only be the staff working in the organisation, and they must be able to log into the program with their organisation's ID and password or identity card, which is not related to the program itself but must be implemented and therefore is a non-functional requirement, since it accounts for how the system should work.

And finally the external requirements include requirements such as regulatory, ethical, legislative, accounting and safety/security requirements. The NBMFS should implement data privacy policy of the country it is used in.

3 System Design

3.1 Use Case Diagram

The purpose of a use case diagram in UML is to demonstrate the different ways that a user might interact with a system. The use case diagram for this system is shown in Figure 1 in the Appendix.

3.2 Class Diagram

Class diagrams represent the structure of a system by modelling its classes, attributes, operations and relationships between objects. They describe what must be present in the system. The class diagram for this system is shown in Figure 2 of the Appendix.

3.3 Activity Diagram

An activity diagram is a flowchart used to represent the flow from one activity to another one. The activity can be described as an operation of the system. An example activity diagram for how the system handles tweets processing is shown in Figure 3 of the Appendix.

4 Implementation

4.1 Version Control Plan

This system is to be developed in agile approach, this means, dividing tasks into short phases of work and frequent reassessments and adaptations of plans. The Agile Manifesto specifies the four key values that software developers should use in order to guide their work. They state that individuals and interactions, working software, customer collaboration and responding to change are the most important aspects in agile development.

For this project, agile methods would be the best since it would allow the team to deliver high-quality software quickly and continuously while promoting team member collaboration and communication with the client.

During the implementation part, version control should be used to help the software team manage changes to the source code over time, this allows the developers to go back and compare earlier versions of the code to help fix the mistakes without interrupting the rest of the team members.

The plan would be to start with an iterative development, which means using Agile Methodologies. This development breaks down the project into smaller chunks of code that is designed, developed and tested in repeated cycles, and with each iteration, additional features can be added until the final product is delivered.

The first step is requirements gathering. In Agile development the owner describes what the final product should do using user stories, since many agile methods argue that producing a requirements document is a waste of time as requirements change all the time. In SCRUM, the user stories requirements are put on a backlog as product backlog items in an ordered list with the most important requirements at the top. The product owner is the one who orders the backlog. This backlog should be finished at the end of a sprint (2-4 weeks), and a MoSCoW Analysis should detail what features should be implemented in the next sprint backlog.

Every day there will be meetings where the progress will be discussed. This promotes collaboration amongst team members since everyone will talk about what they have done since the last meeting, what they will do for the next meeting and what is stopping them from doing it. Another way of promoting collaboration is by doing paired-programming, where 2 developers use one computer and they learn from other people's ways of working and allows them to be more open to constructive opinions.

Version control is used during the whole process, the iterative development and the team's collaborations. There are three types of version control:

- Local version control system: it only keeps the differences between the file's content at progressive stages and then recreates the file's contents
- Centralised version control system: keeps the files in a common place where all the team members have access to from their local machine. Only the last version of the files is retrieved.
- Distributed control system: stores the history of the files on each and every machine locally and syncs the local changes made by the user back to the server. This would be the best option for this system since it provides a collaborative working environment and it is easy to use and manage.

During version control, the master copy of all versions is stored in a repository, which acts as a server, while the version control tools act as the clients. To see the different versions of the NBMFS system, each commit can be accompanied by a tag, which to make it easier to understand and remember will be written as a number and a description of the changes. The version control system will also record who the author was and the changes that have been made. If any team member has any problem with another member's commit, they can see who it was and discuss it together face-to-face, as well as go back to a past version if they decide it was better before.

After each sprint of the development iteration, other features should be implemented in the current version. Team members will be working on different parts of the system, together or alone (paired programming) but they all have to make sure that their parts work well with the current version before committing to the version control system, so that if they ever have to go back, they retrieve a program that does not crash. For this to happen, the version control system provides the branching tool. A branch is like having a totally separate repository where the developer can see the source code it contains and operate independently of the rest of the team members working on the master branch or other branches. When the feature is implemented with the current version and it works like required, then they can merge their branch with the master branch. Merging eliminates the need to cut and paste changes back.

The conflict resolution method used will be Optimistic locking, where every developer gets to edit any file, but the version control system will not allow them to check in a file that has been updated in the repository since they last checked it out. This encourages collaboration among team members because the version control system will sometimes attempt to merge both changes but it might not be the desired outcome, therefore the team members must collaborate together to produce a version that meets both developer's requirements and works well together.

5 Testing

Software testing is used to find errors in a program. It is an investigation that provides the client with information about the quality of the system and gives developers the chance to fix them and create better versions of it.

Testing requires test cases, test data and test results, which compared to test cases provide the test reports. When we run a software test, we execute it with test data and observe the system's operational behaviour.

Software inspections should be done to discover anomalies and defects, as well as considering broadening quality attributes of the program. The inspections will check conformance with a specification but not non-functional requirements.

5.1 Test plan

The NBMFS was a test-driven development system. Test were written before the code using the requirements and then the code was written in order to pass those tests. For example, the message ID consists of a letter which can only be S, E or T and 9 numbers, therefore I designed a test for it and then wrote the validation code. If the code failed it would be changed until it passed the tests. For example, there was an error if the letter that was input was lowercase (s123123123 instead of S123123123) therefore I changed the code so that all the text is transformed to upper before submitting it to validation and therefore the test would be passed, and that error won't appear again.

The benefits that test-driven development has are that the tests cover all the system since the system is created around them, this simplifies regression testing, debugging and the system documentation.

The objective of software testing is to find the most errors with the minimum amount of time and effort. All the tests should be traceable to customer requirements, planned long before the actual testing starts and should be conducted by someone who wasn't involved in the system's development.

The test items in the NBMFS are mainly the classes (messages, sms, text and tweet) and the methods that process abbreviations, URLs or SIR emails. The task is to make sure all their validation is correct and the deliverables should show errors so it can be developed differently. The approach to these test will be first to do some unit testing. The testing done for this system is demonstrated in 5.4 Test outputs of this report. The plan is to fix as many errors as possible so that the staff won't be able to make any mistakes and the program is easy to use and works efficiently. The more the system fails, the more staff that will be required to maintain the system.

Testing is applied at 3 levels: Unit Testing, Integration Testing and Acceptance/Validation Testing.

In Unit Testing each module is tested individually to find bugs in logic, data and algorithms. White Box testing is used for this, where knowledge of the software's internal design is used to develop tests. The test data is derived from the component code, and then the test data is used to test the component code, which then produces the test outputs.

The objective of testing the NBMFS units is to check if exceptions are being raised if there is an error or if the value has been correctly set to what it is supposed to if the input is in a valid format.

Integration testing finds bugs in interfaces between modules. Black box testing, top-down and bottom-up are the methods used to find out these test results. Black box testing is different from white box because it requires no knowledge of the code and focuses only on the requirements and functionalities of the system. Incremental integration tests small units and only adds them to the main program when they pass all the tests. This can be done by the top-down or bottom-up methods. Top-Down integration testing integrates modules downwards through the control hierarchy while Bottom-Up tests the lowest levels first and works upwards.

Finally, validation testing assures that the software meets all functional and performance requirements. This testing is done using the Black box testing method, a regression testing can also be done in order to see if the software still meets its requirements in light of changes to the software or use stress and back-to-back testing. Stress testing exercises the system beyond its maximum design load and checks if the system suffers degradation as a network becomes overloaded.

Stress testing can be used to test the NBMFS to find the scope of the program and see if the objectives are being met. A text or JSON file with many, many entries will make the program have to cope with a lot of data and should not degrade in efficiency.

Non-functional aspects of the system can also be tested, to provide evidence that it can be reused immediately, or only some parts can be reused or that it cannot be reused.

5.2 Test methods

The unit tests are made by testing every single attribute of the classes and asserting the expected result with the actual result or catching the exceptions when they are supposed to appear.

Manual alpha testing was done to make sure all the buttons in the program trigger an action and there are no program crashes. Alpha testing takes place at the developer's site by an end-user so the tests are in a controlled environment. Some people tried the program to see if they were able to crash anything or discover something that wasn't working properly. All failed test where listed in the test table below.

Some common interface errors are misusing the interface, like for example calling a method with the wrong parameters, or misunderstanding the interface, one method could be waiting for another method to do something while the other method was actually waiting for that method so none of them do anything. Infinite loops that will never work can be discovered during testing.

To make sure all the testing was appropriate, I designed tests so that the parameters to a called procedure are at the extreme ends of their ranges, I designed tests which caused the system to fail and I ran the system in a different hardware to make sure the installation would work. Using boundary values, for example, if the twitter name had to be between 1 and 15 characters long, I would do 5 tests, twitter names that are 0, 1, 7, 15 and 16 characters long. This is called equivalence partitioning.

5.3 Test cases

A test case consists of a description of the objective of the test, the input test data needed to conduct the test and a description of the expected result.

For the NBMFS there are two approaches in unit testing to verify them: one is to expect an exception and the other to assert two values (the expected value versus the actual result) and if they are the same then the test is passed.

5.4 Test outputs

In this table, the tests can be expected to pass or fail the tests, failure can be expected so even if the test fails, it can mean that that was exactly what was expected. The 'PASSED?' column is the one that states if the actual test was passed, which are highlighted in red.

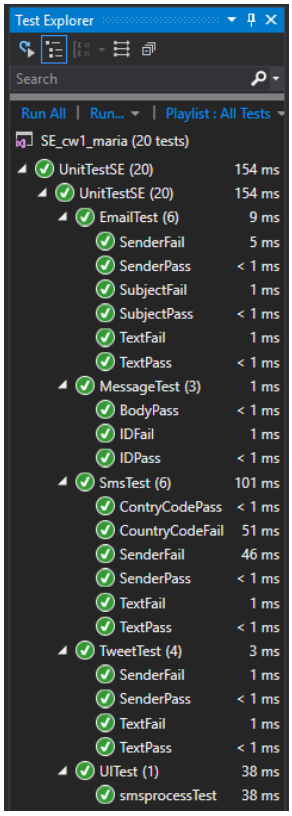
INPUT	EXPECTED	PASSED?	ISSUES	RESOLUTION
`message.id = "s123456789";`	pass	No	If string has a lowercase letter it won't accept it since ID will always be an uppercase letter	Changed _id = value To _id = value.ToUpper()
`message.id = "S123123123";`	pass	yes		
`message.id = "";`	fail	yes		
`message.id = "S123";`	fail	yes		
`message.id = "S1234567890"`	fail	yes		
`message.body = "";`	fail	yes		
`message.body = "hi";`	pass	yes		
`tweet.Sender = "does not start with @";`	fail	No	The class was set to throw an Exception while the test was expecting an ArgumentException	Changed from Exception to ArgumentException
`tweet.sender = "@";`	fail	no	Didn't validate if it was more than 0 characters long	Added validation `(value.StartsWith("@") && (value.Length < 16) && (value.Length > 0))`
`tweet.sender = "@maria";`	pass	yes		
`tweet.sender = "@1234567890123456";`	fail	yes		
`tweet.text = "";`	fail	no	Didn't validate if text was more than 0 characters long.	Added validation `((value.Length < 141) && (value.Length > 0))`
`tweet.text = text with more than 140 chars`	fail	yes		

`tweet.text = "hi there";`	pass	yes		
`email.sender = "maria@hello.com"`	pass	no	Error in the regular expression used	change to var addr = new System.Net.Mail.MailAddress(value);
`email.subject = ""`	fail	no	Subject has to be less than 20 characters but more than 0. Validation only checked for less than 20.	Added to validation `((value.Length < 21) && (value.Length>0))`
`email.subject = "12345678901234567890"`	pass	yes		
`email.subject = "123456789012345678901"`	fail	yes		
`email.text = ""`	fail	yes		
`email.text = a text longer than 1029 characters`	fail	yes		
`email.text = "hello"`	pass	yes		
`sms.CountryCode = "+44";`	pass	yes		
`sms.CountryCode = "12345";`	fail	no	Country codes are max 3 numbers long	Changed validation
`sms.CountryCode = "";`	fail	pass		
`sms.Sender = "633675139";`	pass	no	Regular expression wasn't working properly	Change it to a new regex `string _regex = @"^(?<countryCode>[\+][1-9]{1}[0-9]{0,2})?(?<areaCode>0?[1-9]\d{0,4})(?<number>[1-9][\d]{5,12})(?<extension>x\d{0,4})?$";`
`sms.Sender = "1";`	fail	yes		
`sms.text = "";`	fail	yes		
`sms.text = "hello";`	pass	yes		
`sms.text = something longer than 140 characters`	fail	yes		
`sms.text = something exactly 14o characters long`	pass	yes		
`abbreviations("Hello LOL");`	Hello LOL <Laughing out loud>	no	Instead of adding the definition to the abbreviation, it replaced it	Create a new string containing the work + definition and replace abbreviation word with new string
`abbreviations("Hello LOL LOL ASAP");`	Hello LOL <Laughing out loud> LOL	no	Result was: Hello LOL <Laughing out loud> <Laughing out loud> LOL <Laughing out loud>	Check if the word after the abbreviation is a definition. If it is then don't add the definition, but if the next

	<Laughing out loud> ASAP <As soon as possible>		<Laughing out loud> ASAP <As soon as possible> Because since there are 2 LOLs, the process would add the definition everytime the word appeared and this was done twice so the word was added twice	word didn't have the definition after it then add it.
abbreviations("Hello LOL LOL ASAP")'	Hello LOL <Laughing out loud> LOL <Laughing out loud> ASAP <As soon as possible>	yes		
url_search("http://google.com");	pass	yes		
url_search("https://google.com");	pass	no	Method was working for only http but not https	Added https to validation
url_search("hello");	fail	yes		
url_search("https://google.com https://google.com");	pass	yes		

Manually tested all buttons in the User Interface to make sure they were working

Process message button	works	
Upload files button	works but crashes if no file is selected	added if statement to check if file is selected
double click textboxes	no crashing	
double click listboxes	crashed if empty - system thinks user is selecting an item from the listbox but in reality it is empty and program crashes	added an if statement to check if the listbox is not empty
selecting a file that is .txt	works and processes it perfectly	
selecting a .json file	works and processes it perfectly	
selecting another type of file	works, displays error message but system doesn't crash	
URL being added to quarantine list effectively	they are added, but if a URL appears twice, it will be added twice	Fixed it so that it checks if the URL already exists in the list then don't add it
twitter mentions added to mentions list	yes and they're not repeated	
twitter hashtags added to mentions list	yes but not in descending order	fixed it and now shows top mentions at the top
SIR sort code and subject added to list	yes works	



The system is made so that if there is any problem, it won't crash, it will only tell the user what the problem is but it won't stop working. The solution to the risks are having some technician in the company that knows how to fix the problems that the users find. The system is designed so that it is not easy for the users to find a problem. It is clear and specifies what the user is doing wrong. For example, if they choose a file that is not in .txt or .json format, the program will tell them. If the format of the ID of the message of any other part of the message is not the correct one, the system will display a message explaining what the user is doing wrong.

6 Evolution

Evolution, software updates, or new requirements emerge when the program starts being used. Errors are found by users that developers couldn't find; performance or reliability might need to be improved; the environment might change or new computers might be added to the system. All of these should be maintained and should keep the program working and up-to-date.

The evolution processes depend on the type of software of the system, the development processes used and the knowledge that the current workers involved have of the system.

There are three types of maintenance: corrective (to repair bugs), adaptive (to adapt software to a different operating system) or perfective (add or modifying a system's functionality. While the first two are only maintenance, the third type (perfective) can also be called Evolution, since it is making the system more and more effective.

Normally, the higher the development costs, the smaller the maintenance costs, since developers have put more effort on it. The costs also depend on the contractual responsibility of the developers: if the developers don't have contractual responsibility for maintenance there is no incentive to design for future changes, since they just concentrate on creating a working system, not a system that is easy to maintain.

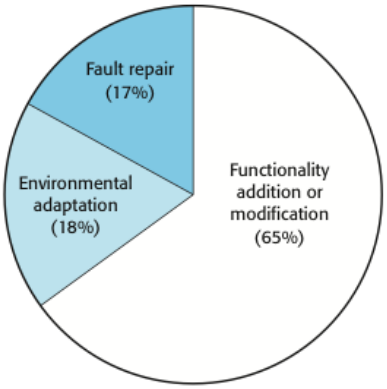
6.1 Evolution Strategy

The Napier Bank Message Filtering system has been developed with modularity, so that it is easy to understand the program since the maintenance staff wasn't involved in the development. The program is divided in 3 layers: business, data and presentation, each dealing with different parts of the system.

The first maintenance that the NBMFS will require is corrective. The end-users will find some bugs that the developer of the system (me) couldn't find and therefore those will need some assistance. Then adaptive maintenance will be required since this program has only been tested in one operating system and some end-users might prefer to use a different one. Finally, once the program is working well in all environments, evolution will happen. The business requirements might change, and they might want to

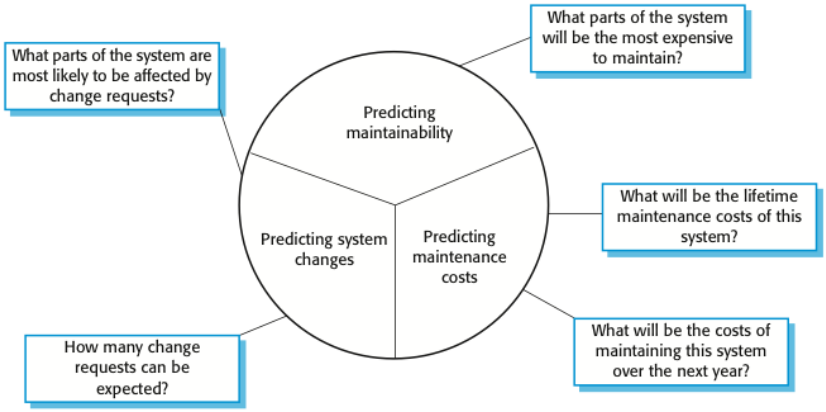
add new features and functionalities to the NBMFs which will require staff to understand the code, restructure it, modularise it even more or clean up data and restructure system data.

During maintenance, 65% of it is for adding functionalities or modifying the program, 18% is for environmental adaptation and 17% for fault repairs.



For maintenance predictions, developers should always consider several points:

- What parts of the system will be more expensive to maintain: the NBMFs is not expensive to maintain since it is a simple yet efficient program. If the business wants to upgrade data storage or the hardware, then they will need to invest on it. However, the software part of the program will not require a lot of cost
- What will be the lifetime maintenance costs of the system: usually these costs are from x2 to x30 times bigger than the development costs since they involve technical and non-technical factors and maintenance corrupts the software structure so it makes further maintenance more difficult. Also if the program is still being used after a couple of years and the programming languages used at that time have changed since the program was developed, this will require higher support costs.
- What will the costs of maintaining the system over the next year be: the first few months will require a higher cost for adaptations and bug fixes but once that is done, all the maintenance there is to do is for the evolution of the system
- How many change requests can be expected: normally then the clients define the requirements, they are not completely sure of exactly what they need, and they find out they need more functionalities once they start using the program. Therefore, the first year of using the system will probably have many requests for change, and as the program keeps being changed it will be more suitable for the business. Maybe the business requires to read the messages from an actual twitter account instead of from a text or json file and that functionality should be added accordingly
- What parts of the system are most likely to be affected by change requests: the user interface is what is most likely to change in the NBMFs since if new functionalities are added, they should be shown in screen somewhere so the users can notice the changed and work with them



Reengineering can take place after maintaining the system for a long time if the maintenance costs increase but refactoring the system will be a better option for the NBMFs since this will involve the continuous process of improvement throughout the development and evolution process to avoid the structure and code degradation that increase the costs and difficulties of maintaining a system.

All organisations can decide if they want to scrap the system completely and modify business processes so that they don't need it anymore, they can choose to continue maintaining the actual system if it is necessary, re-engineer it to improve its maintainability or replace the system completely with a new system. For this they should take into account the system quality and the business value it has.

7 Appendix

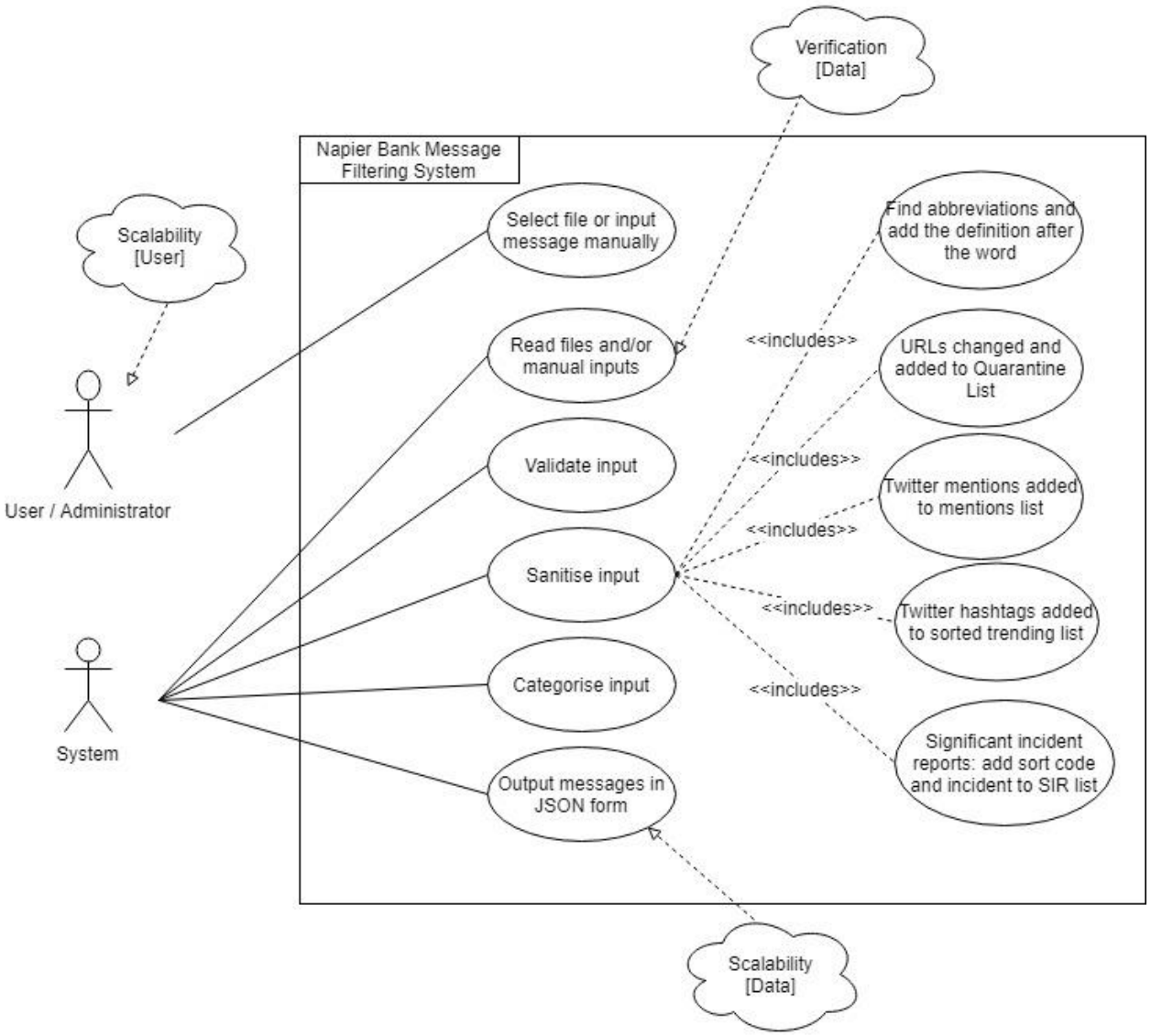


Figure 1. Use Case

Figure 2.
Class Diagram

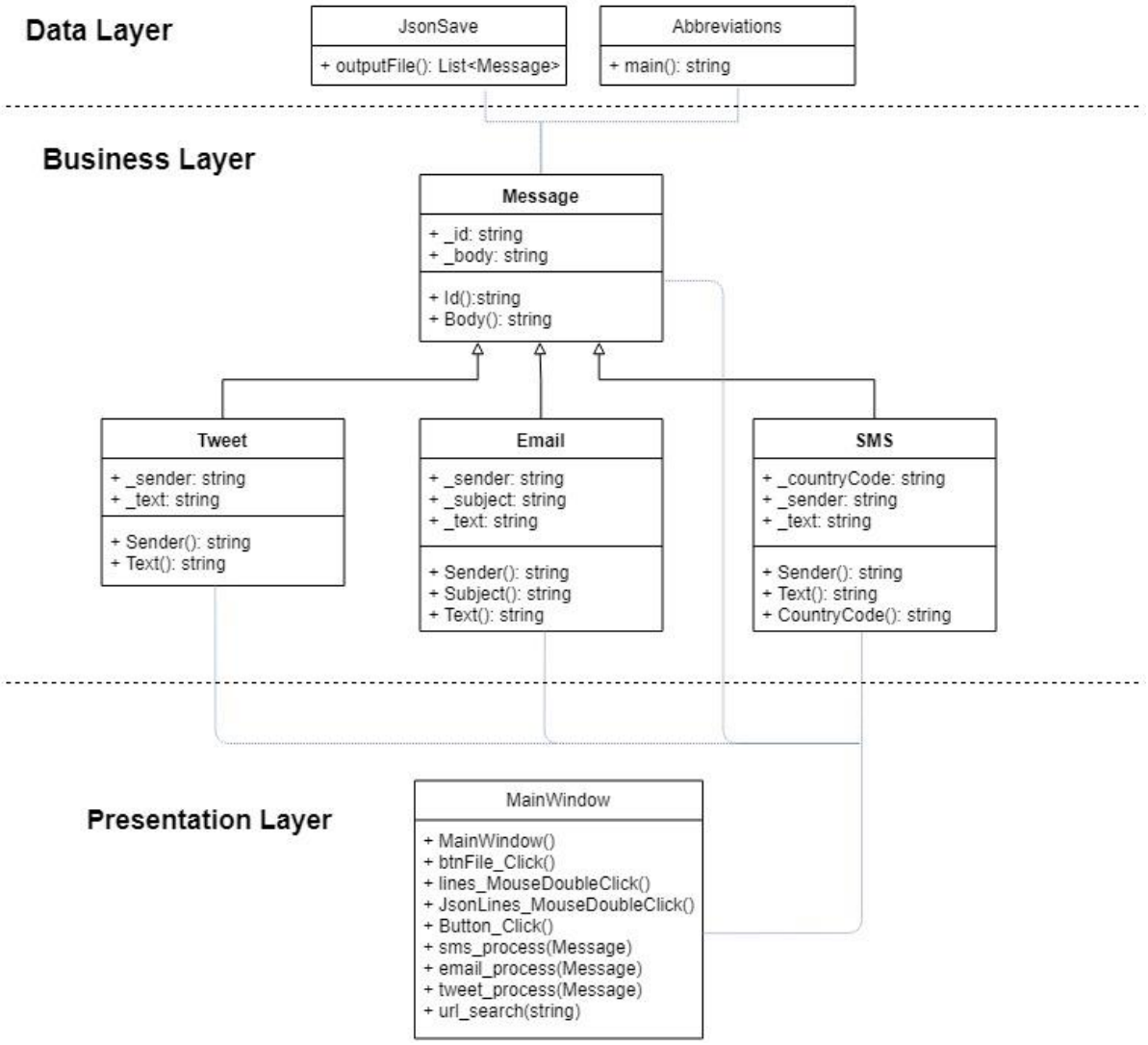


Figure 2. Class Diagram

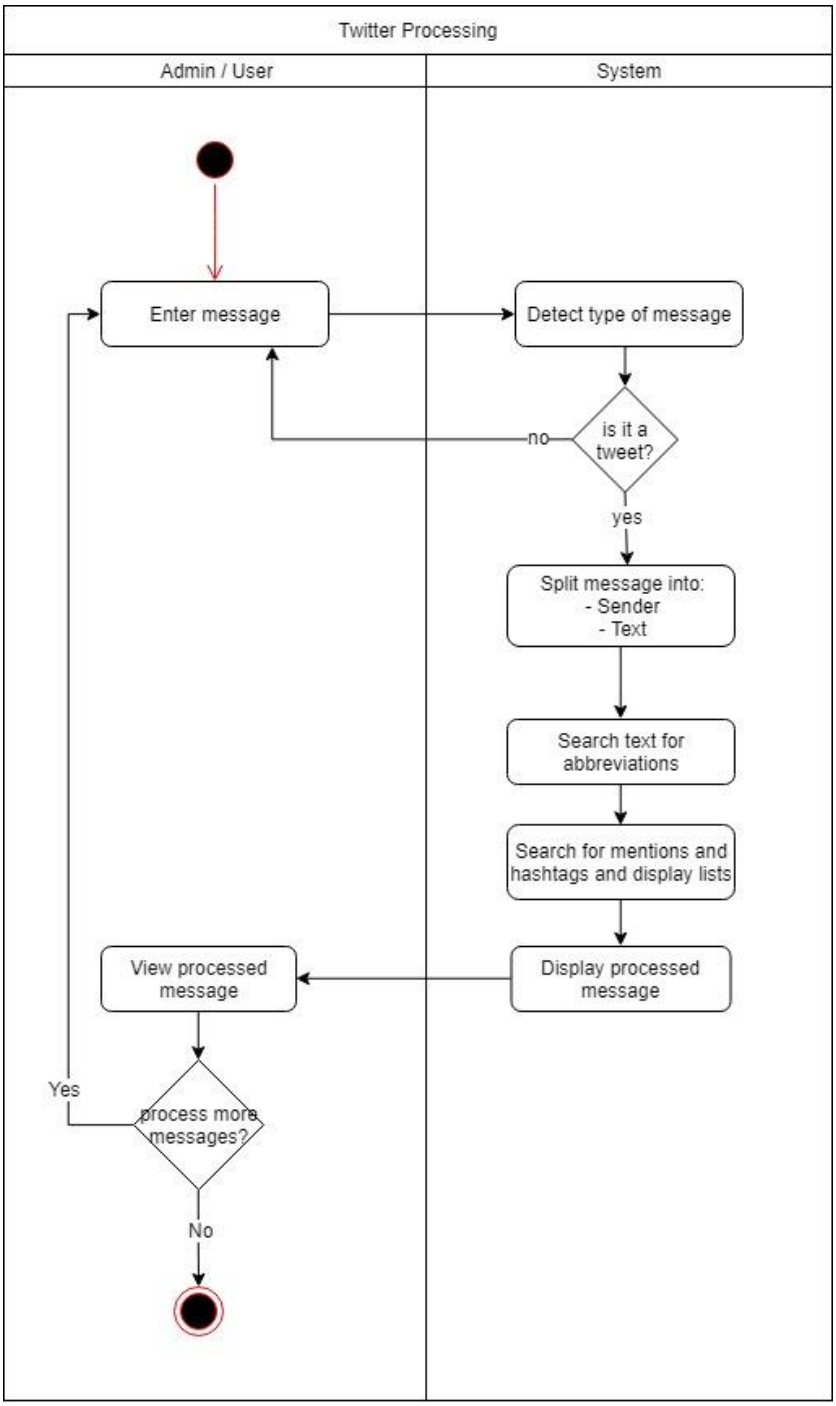


Figure 3. Activity diagram for tweet processing

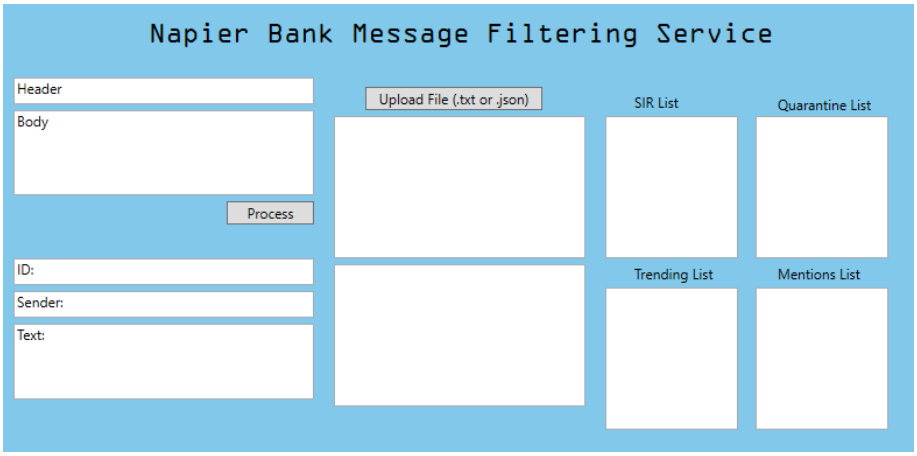


Figure 4. Screenshot of the application

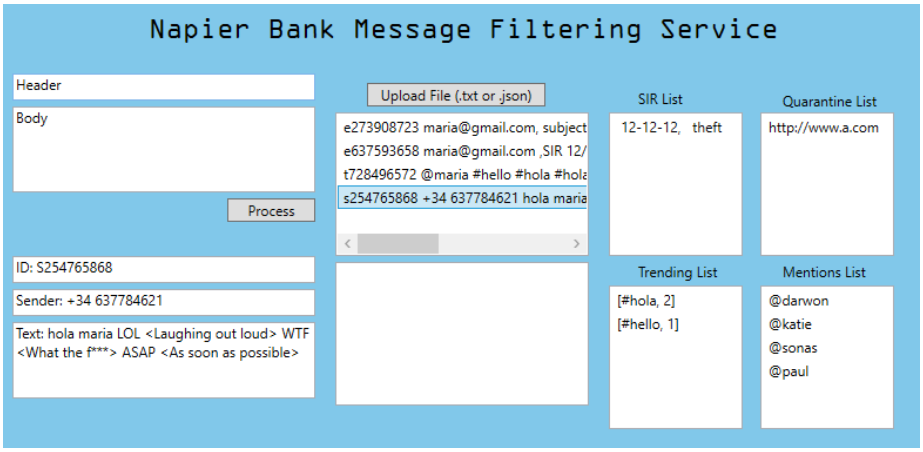


Figure 5. Uploading a .txt file to the app. Each line gets added to the listbox and user selects the line they want to process from there and values are displayed in the Lists.

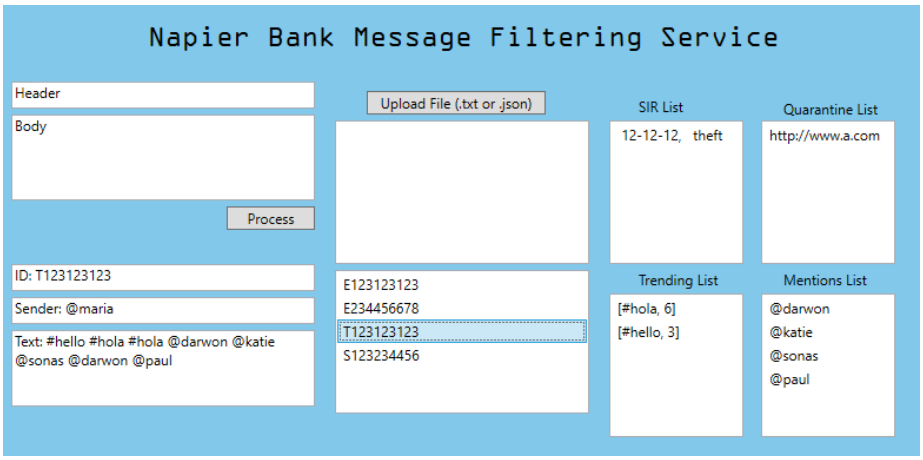


Figure 6. Uploading a .json file. Every object in the file gets its header ID added to the listbox and users select from there what message they want to process. Same as before, values are added to the lists in the sides and processed message in the lower left textboxes.