Student number: 40284739

# Napier Bank Message Filtering Service

## Table of Contents

# Requirement Specification

Requirements engineering was conducted on the coursework specification and the following software requirements specification and use case diagram were produced.

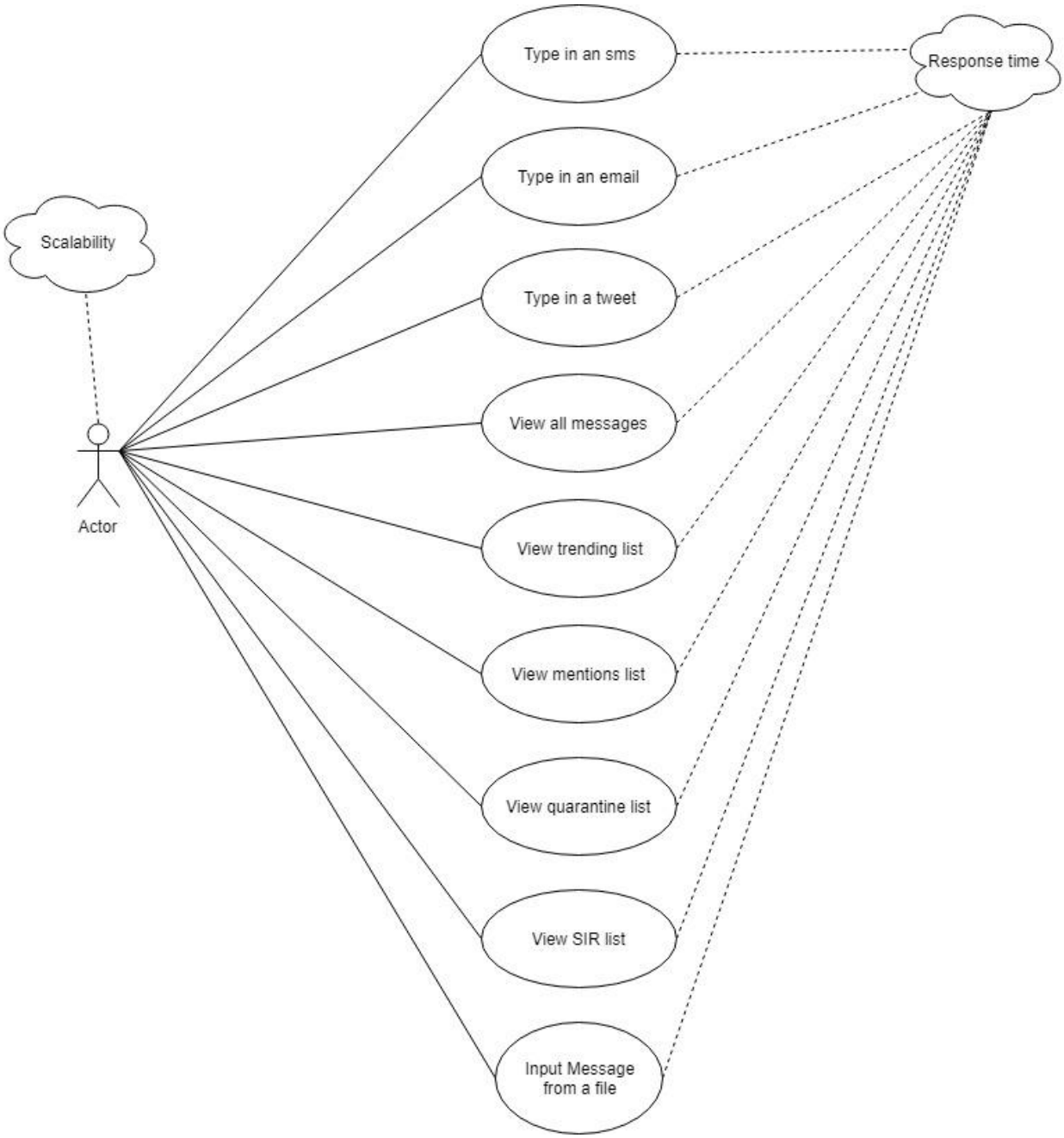
## Software requirements specification

1. The system must deal with three types of messages – Sms, Email and Tweet
2. Automatically detect the message type. That would be done with the help of drop down menu.
3. Every message has an ID in the form of an indicating letter (S, E or T) followed by 9 numeric characters (e.g. E123456789). The ID is automatically generated and assigned from the system. They are all unique.
4. The system should validate, sanitize and then re-display the processed message.
5. Sms:
    a. Sender is international phone number
        i. Cannot be empty
        ii. No shorter than 5 chars
        iii. No longer than 15 chars
        iv. Allowed characters “0-9”, ‘+’, ‘-‘ and spaces allowed
        v. Must start with ‘+’ (because it’s international)
    b. Message
        i. Cannot be empty
        ii. 140 chars before expanding the abbreviations
        iii. When Abbreviation is encountered, it will be expanded after its instance encapsulated with “<>” e.g. “ROFL <Rolls on the floor laughing”. Abbreviations are specified in a csv file.
            1. The check for abbreviations is completely non-case sensitive.
            2. Multiple occurrences of the same abbreviation are allowed.
            3. The abbreviation “dictionary” is loaded each time the app is started in order to be up to date with any changes to the file.
6. Email – can be standard or SIR:
    a. Sender is standard email address
        i. Cannot be empty
        ii. No shorter than 5 chars
        iii. Cannot start or finish with ‘@’ or ‘.’
        iv. Must have only one @ sign
        v. Must have min one ‘.’ sign
    b. Subject
        i. Standard - Can be empty!
        ii. Max 20 chars
        iii. Significant Incident Reports – SIR followed by the date (auto-gen) (e.g. “SIR dd/mm/yy”)
    c. Message contains embedded links
        i. Cannot be empty
        ii. Must be max 1028 before replacing the embedded links
        iii. Links are defined by containing “http”, “https”, “www.”, “.com”, “.net” or “.uk”

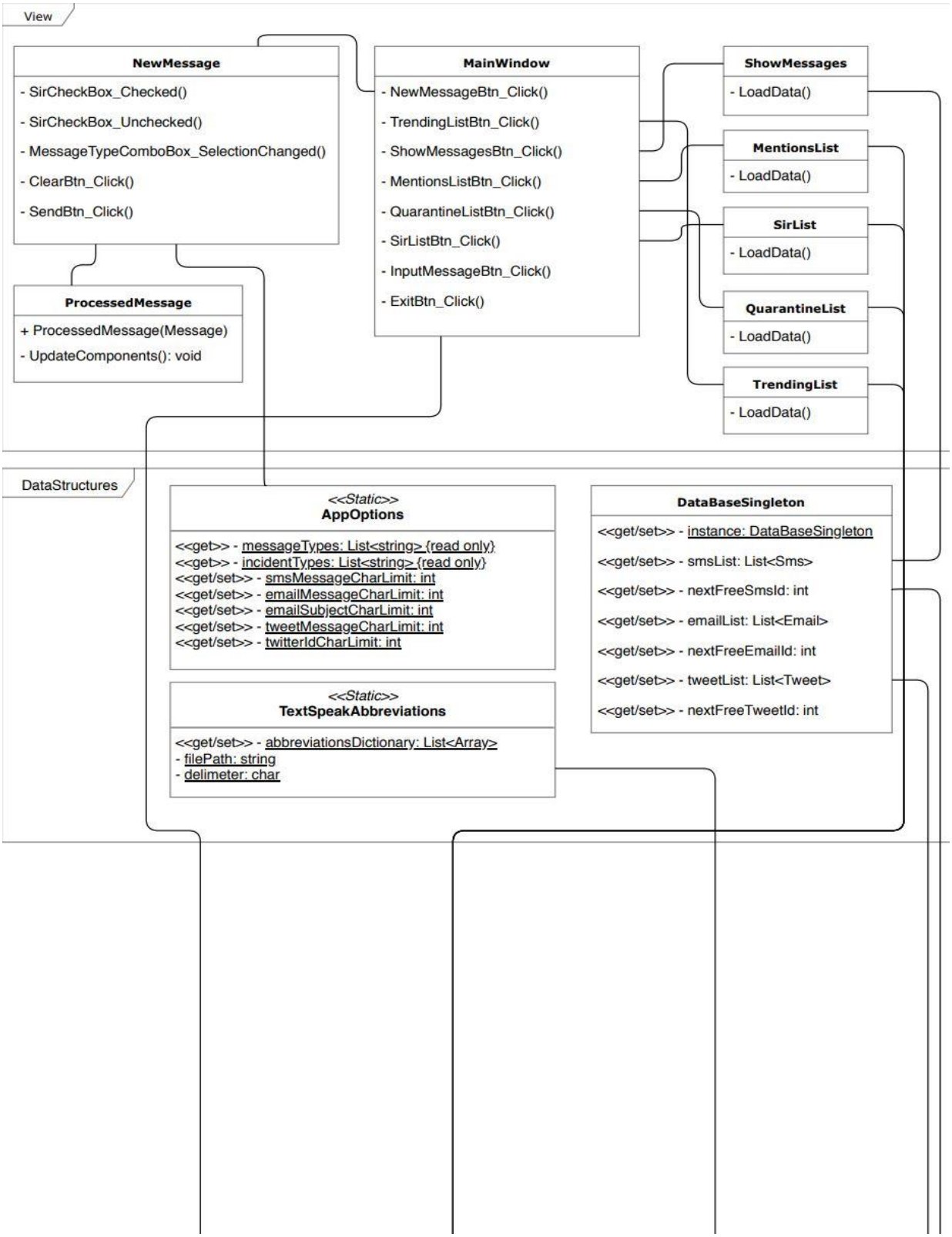
iv. Links are substituted with “<URL Quarantined>”
v. Every Email has a list of quarantined links
vi. SIR - First line of the processed message is the sort code
vii. SIR – Second line is nature of incident (chosen from the provided list)

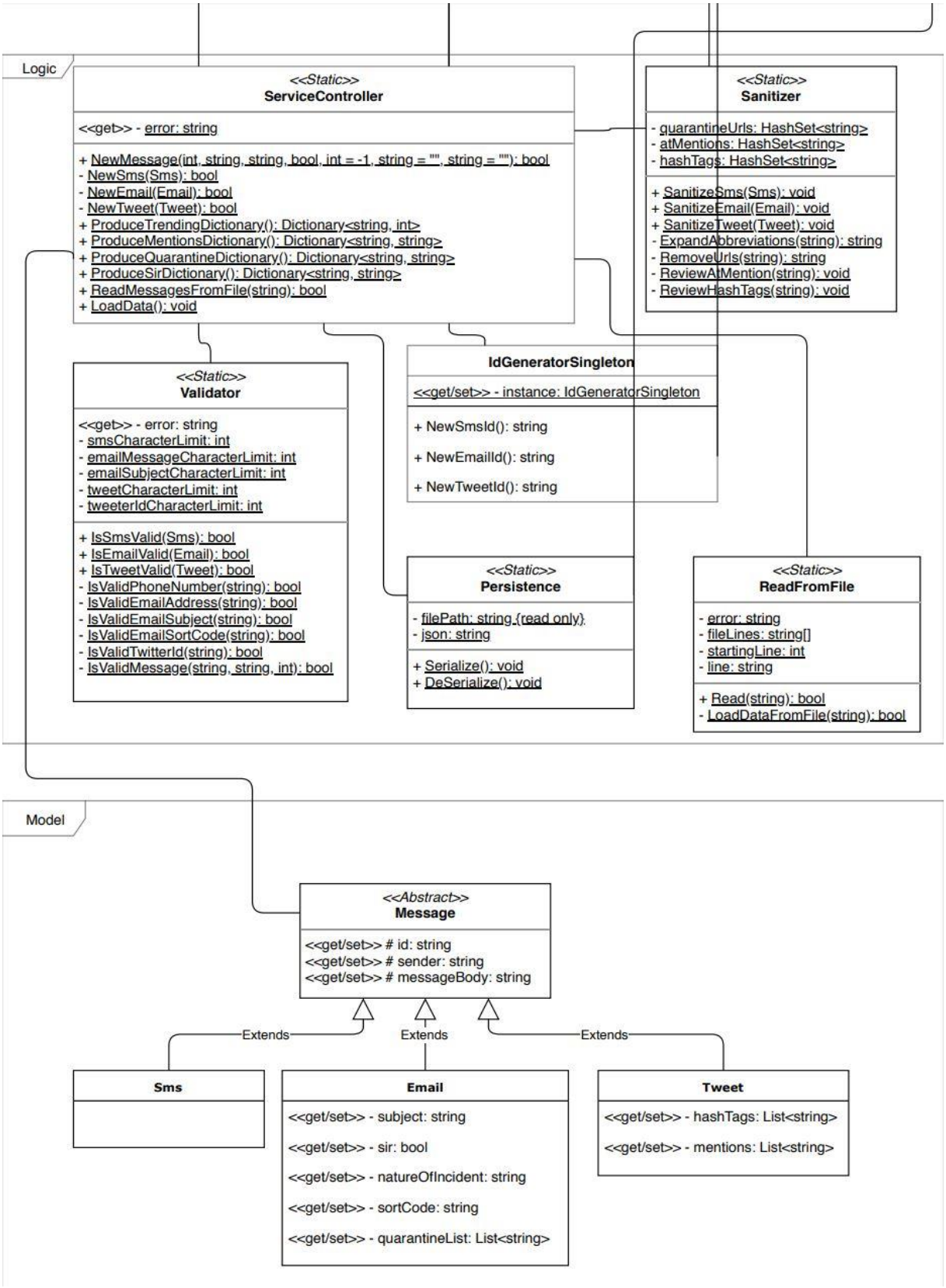
7. Tweet:
    a. Sender is twitter ID
        i. Starts with @
        ii. max 15 chars (total 16 + @)
    b. Message
        i. Max 140 chars before expanding abbreviations
        ii. Abbreviations like in sms
        iii. Hashtags
            1. Each tweet has a list of unique hashtags (that way hashtag has weight of 1 per tweet)
        iv. Mentions – twitter IDs
            1. List of unique mentions so later the users can get a notification.
8. The system must output the processed messages into json file. Every time a new message is entered, the whole data of messages is re-serialized.
9. Provide a SIR list
    a. Sort Code and Nature of incident
    b. Additionally, id of the email so it can be tracked back in future.
10. Provide a Quarantine list
    a. List of quarantined links
    b. Additionally, each url will have a list of email ids where the url has been removed. (In future an url might be evaluated and removed from quarantine. That way you can enable it in all emails where is present.)
11. Provide a Trending list
    a. Each hashtag with the number of instances (still unique hashtags per tweet)
12. Provide a Mentions list
    a. Each tweet id associated with a list of mentions

## USE Case – NFRs



# Class Diagram




Logic
<<Static>>
ServiceController
<<get>> - error: string
+ NewMessage(int, string, string, bool, int = -1, string = "", string = ""): bool
- NewSms(Sms): bool
- NewEmail(Email): bool
- NewTweet(Tweet): bool
+ ProduceTrendingDictionary(): Dictionary<string, int>
+ ProduceMentionsDictionary(): Dictionary<string, string>
+ ProduceQuarantineDictionary(): Dictionary<string, string>
+ ProduceSirDictionary(): Dictionary<string, string>
+ ReadMessagesFromFile(string): bool
+ LoadData(): void
<<Static>>
Sanitizer
- quarantineUrls: HashSet<string>
- atMentions: HashSet<string>
- hashTags: HashSet<string>
+ SanitizeSms(Sms): void
+ SanitizeEmail(Email): void
+ SanitizeTweet(Tweet): void
- ExpandAbbreviations(string): string
- RemoveUrls(string): string
- ReviewAtMention(string): void
- ReviewHashTags(string): void
<<Static>>
Validator
<<get>> - error: string
- smsCharacterLimit: int
- emailMessageCharacterLimit: int
- emailSubjectCharacterLimit: int
- tweetCharacterLimit: int
- tweeterIdCharacterLimit: int
+ IsSmsValid(Sms): bool
+ IsEmailValid(Email): bool
+ IsTweetValid(Tweet): bool
- IsValidPhoneNumber(string): bool
- IsValidEmailAddress(string): bool
- IsValidEmailSubject(string): bool
- IsValidEmailSortCode(string): bool
- IsValidTwitterId(string): bool
- IsValidMessage(string, string, int): bool
IdGeneratorSingleton
<<get/set>> - instance: IdGeneratorSingleton
+ NewSmsId(): string
+ NewEmailId(): string
+ NewTweetId(): string
<<Static>>
Persistence
- filePath: string {read only}
- json: string
+ Serialize(): void
+ DeSerialize(): void
<<Static>>
ReadFromFile
- error: string
- fileLines: string[]
- startingLine: int
- line: string
+ Read(string): bool
- LoadDataFromFile(string): bool
Model
<<Abstract>>
Message
<<get/set>> # id: string
<<get/set>> # sender: string
<<get/set>> # messageBody: string
Extends
Extends
Extends
Sms
Email
<<get/set>> - subject: string
<<get/set>> - sir: bool
<<get/set>> - natureOfIncident: string
<<get/set>> - sortCode: string
<<get/set>> - quarantineList: List<string>
Tweet
<<get/set>> - hashTags: List<string>
<<get/set>> - mentions: List<string>

## Testing

The overall testing strategy is to conduct user test for each field for each type of message in order to find any logical errors in the design of the methods. To test the validity of those methods, unit tests were chosen.

The scheduling of those tests is based on the version/development plan for the system. At the end of each sprint, the type of message under development would undergo those tests and any problems will be addressed in the start of the next sprint.

After the implementation of the 3 types of messages, the UI will be finalized and tested.

Finally, the feature to input from a file was implemented to produce a final test method for all the type of messages. This will be demonstrated at the practical.

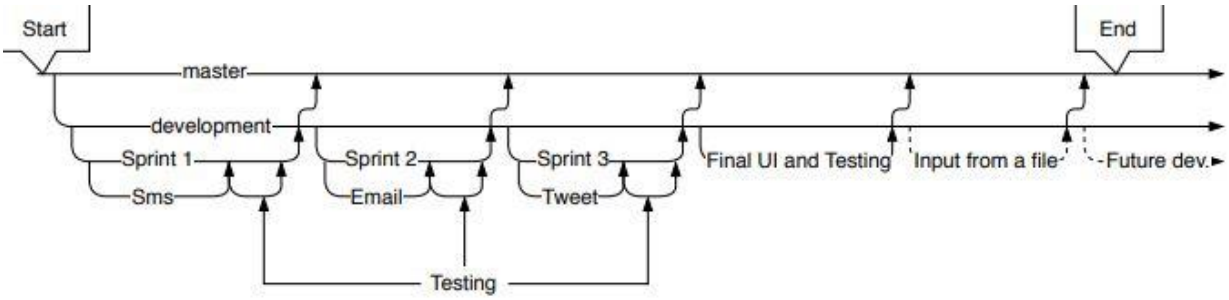
Since the testing methodology for the first 3 sprints is so similar, only one is demonstrated below.

Unfortunately, the implementation of unit tests was not achieved due to time constrains.

| Test | Result | Comment |
|---|---|---|
| Sprint One - Sms | | |
| Empty sender | pass | |
| Long sender | pass | |
| Chracters in sender | pass | |
| Must start with + | fail | fixed in the next sprint |
| Starts with white space | pass | |
| | | |
| message is empty | pass | |
| message is too long | pass | |
| expand abbreviations | pass | |
| expand abbreviations follow by punctuation | fail | still open bug |
| white space as only character | fail | still open bug (needs to validate that characters present are not only white spaces) |

## Version Control Plan

The system was presumed to be developed in an agile approach. If the system were to be developed by a team the structure would be similar, but the prints might be able to be developed in parallel. The proposed plan below can be applied to teams but also a single developer.




The technology used for this project was git hub even though any version control such as the integrated one in visual studio should be sufficient.

## Evolution and Maintainability

The architecture of the software is heavily influenced by considerations for evolution and maintainability. The structure is modular so at any point a new method or API can replace most of the parts of the software. For example, if the validation for a specific type of message was to change, it would be easy as to replace the method or introduce an external API.

The software structure is also build with platform migration in mind. Both the models (entities) and the front (view) layer are left as empty as possible. That way it would be painless to migrate to a web-based application or introduce a data persistence in the form of a database.

Future features that have been considerate are introducing the option for different type of messages, editing and deleting records.

The decision for displaying each type of list (sir, mentions, quarantine and trending) is to they are calculated real time and also that can be used as a stepping stone for future development. Such as approving some of the quarantine list URLs or re-directing messages to all the mentions.

Currently the project has for a cost the space for storing the record of messages as well as the run time memory for buffering this list of messages. Depending on further requirements or features, the memory used can be reduced and optimized. Further design is needed based on feedback to decide the development strategy.