**Software Engineering Coursework Report**

**40233308**

**Subject: Napier Bank Message Filtering System**

**Introduction**

The Napier Bank requires a messaging service, namely Napier Bank Messaging App (NBMA), which will validate, sanitize and categorise incoming messages to the bank in various forms.

The purpose of this document is to demonstrate the Software Engineering process underlying the NBMA development, accompanied by critical discussions and evaluations.   
We will do so by using the four fundamental process activities outlined by Sommerville:

* Software Specification
* Software Development
* Software Validation
* Software Evolution

In order to avoid problems related to traditional Software Development, we will adopt an Agile workflow.   
We understand that this type of methodology is characterized by an iterative approach, with self-organizing cross functional teams aiming to deliver quality software by addressing user’s needs.  
More on this topic will be discussed further down in the report, for now we will talk about it in respect to its natural disposition in welcoming new requirements.

**1. Requirements Specification**

Requirements are often subject to change as users rarely have clarity over their needs are.  
Whereas the traditional waterfall model could not efficiently make front to the above problem, the modern agile approach offers a more flexible solution, of which manifestos’ values (among others) customer collaboration, therefore welcoming new requirements even during development.  
In addition, agile methods are quite reluctant to producing a requirement documentation as often this becomes obsolete, marking the success of agile in non critical business systems.   
Keeping in mind that requirements should aim for completeness and consistency, we will have a look at:

* Business Requirements
* User Requirements
* System Requirements

Requirement validation will be done towards the end, verifying the system meets stakeholder’s requirements by applying acceptance testing.

**1.1 Business Requirements**

The purpose of the business requirements is to delineate the reasons the organization is undertaking the project. (Enfocus Solutions, 2012)

The Napier Bank needs a messaging system which will streamline the process of validating, sanitizing and categorising different types of incoming messages. This will eventually lead to a more efficient process, with higher message processing speed and lower errors.  
The system will also automatically output the messages to a file in JSON format for later retrieval; here we understand that the system performance might degrade as the size of the file increases, a better alternative could be implementing a database, but this will be part of the system evolution.

In order to acquire information about the system we must first identify the stakeholders, the people who have an interest in the system; the list of stakeholders can be very long, here we will illustrate the primary roles for the NBMA in a real world scenario.

|  |  |
| --- | --- |
| **Stakeholder** | **Reason** |
| Users | interact with the NBMA inputting the messages |
| Designers | aim to build a perfect messaging system |
| System Analysts | concerned with gathering correct requirements |
| Training and user support staff | their main purpose is to ensure a usable messaging system |
| Business analysts | they perform competition analysis aiming to develop the best messaging app product |
| Project manager | concerned with timing and expenses |

**1.2 User Requirements**

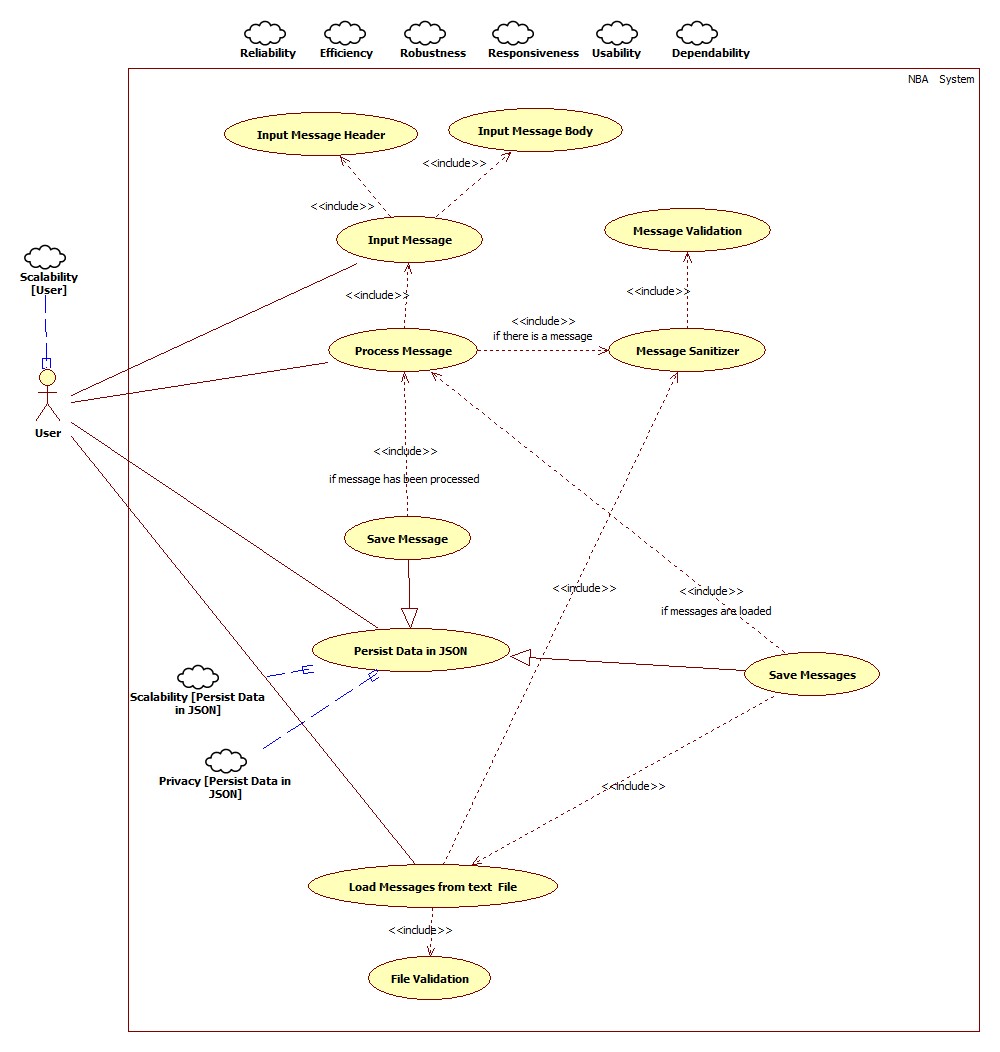
The domain of the problem can be understood by implementing some elicitation techniques, although there are many available out there (questionnaires, prototyping, interviewing, etc..) the only feasible approach is the use of documentation to extrapolate user requirements.User requirements are a high-level view over the functionalities to be provided by the system (both functional and non-functional), derived from a study with the user (performed by the Business Analyst).   
This is a crucial phase in the development process as we have to deal with soft entities who often do not know what they want, providing incomplete and inaccurate information. (Enfocus Solutions, 2012).  
Various methods are available for user requirements extrapolation and definition, here we implement user stories and the popular use case diagram where non-functional requirements are also included.

**1.2.1 User Stories**

User stories are discursive requirements descriptions from the user’s viewpoint used to start a conversation.They derive from the popular agile technique called Extreme Programming and their aim is to increase the interaction between customer and developers.

|  |  |
| --- | --- |
| 1 | As a user I want to to input a message into the system so I can process and save it to a file in JSON. |
| 2 | As a user I want the message I input to be validated and processed by the system upon request so I can see it processed. |
| 3 | As a user I want to load messages from a file so I can see them already validated and processed. |
| 4 | As a user I want the system to automatically recognize and categorize any message I might input into the system. |
| 5 | As a user I want to be able to save the processed message to a file. |
| 6 | As a user I want to be able to save the loaded processed messages to a file. |
| 7 | As a user I want to be able to see a trending list. |
| 8 | As a user I want to be able to see a mention list. |
| 9 | As a user I want to be able to see a SIR list. |

**1.2.2 Use Cases Diagram**“Use Cases” is a scenario base technique in UML which describes the interactions within the system. They give us a deeper understanding of the product and of any functional and non-functional requirement.



In the above diagram we can see functional requirements outlined by ellipses, these are services the system should provide and what the system should not do. For instance, the system should load messages from a text file only if this has been validated.  
As regards non functional requirements, they are constraints on the services and functions offered by the system. (ReQtest, 2012) They are outlined by little clouds in the diagram.

**1.3 System Requirements**

Whereas the user requirements are a simplistic way of describing the system functions, the system requirements aim to provide with a detailed description of functional and non-functional specifications to be used by the technical staff in order to develop the system.   
They support the systematic review, evaluation and approval of the system.

Non-functional requirements can be classified in:

* Product requirements: concerned with aspects such as security, efficiency (performance and space), dependability and usability.
* Organizational requirements: including environmental, operational and development requirements.
* External requirements: dealing with ethical, regulatory and legislative aspects.

Below each system requirement has functional requirements (F) and non-functional requirements (NF). Some NF requirements have a \* near them, this indicates that the requirement is not required for this particular coursework, although it is very likely to be integrated in a real life scenario.

|  |
| --- |
| 1-The NBMA shall accept an input message, validate and categorize it. |
| **F.1** The messages can be in three main forms: SMS (S), Tweet (T) and Email (E). In addition, email can be of two types: Standard Email Message (SEM) and Serious Incident Report (SIR).  **F.2** Each message presents itself in the form of a header and a body, hence the header and body must be entered into two separates text fields.  **F.3** The header of the message has a unique ID representing a unique message entity, it starts with a letter according to the message type (S: SMS, T: tweet, E: email), followed by 9 numeric characters.  **F.4** The body of the message is comprised of different sections according to the message type.  **F.5** SMS’s body has a sender section in the form of an international number, followed by a text section which mustn’t exceed 140 characters.  **F.6** Tweet’s body has a sender section in the form of a tweet ID starting with @ and followed by a maximum of 15 characters; it follows a text section which mustn’t exceed 140 characters.  **F.7** Email’s body has a sender field in the form of an email address, followed by a subject field no longer than 20 characters. If the email is a standard type then a text section no longer than 1028 characters will follow, otherwise if it is a significant incident report the subject will be in the form “SIR dd/mm/yy” followed by a sort code section in the form “nn-nn-nn” , a section on the nature of the incident report and a text section no longer than 1028 characters. The nature of the incident must belong to one of the followings: Thefts, Staff Attack, ATM theft, Raid, Customer Attack, Staff Abuse, Bomb Threat, Terrorism, Suspicious Incident, Intelligence, Cash Loss.  **NF.1** \* Scalability: The system shall handle multiple input sessions where users input messages at once, hence response time must not degrade.  **NF.2** Reliability: The input session must be error free.  **NF.3** Usability: The format of the input message shall not be too complex for the user.  **NF.4** Speed: validation and categorization mustn’t take any longer of a second of computation. |

|  |
| --- |
| 2-After the user has input a message, the system (after validation and categorization) shall process the message and display it. |
| **F.1** Both SMS and Tweets might contain abbreviations, these (after processing) must be followed by two brackets as “<>” inside which the abbreviation text fill be expanded.  **F.2** Email messages might contain URLs, these must be removed, added to a list and replaced by “<URL Quarantined>”.  **F.3** Sort Code and Nature of Incident will be written to a SIR list for any of the Significant Incident Report Emails.  **F.4** For tweet messages, hashtags will be added to a hashtag list that will count the number of uses of each hashtag to produce a trending list; mentions in the body (embedded twitter ids) will be added to a mention list.  **F.5** Error messages shall inform he user of the nature of any failure.  **NF.1** \* Scalability: The system shall handle multiple sessions where users process messages at once, hence response time must not degrade.  **NF.2** Speed: the message process time should be lower than a second.  **NF.3** Ease of use: the processed message should be easy to understand. |

|  |
| --- |
| 3-After the user has selected a file with messages to load, the system shall load, process and display the messages. |
| **F.1** The file should be validated to ensure the entities are in the correct format.  **F.2** Validation, categorisation and processing shall be executed as specified in requirement 1 and 2.  **F.2** Error messages shall inform he user of the nature of any failure.  **NF.1** Speed: loading the messages shall take a reasonable amount of time.  **NF.2** Ease of use: the processed message shall be easy to understand.  **NF.3** \* Scalability: The system shall handle multiple input sessions where users load from files at once, hence response time must not degrade. |

|  |
| --- |
| 4-The NBMA should have a UI for the user to interact with. |
| **F.1** The UI shall have two input fields for message header and message body.  **F.2** The UI shall offer a mechanism for processing and displaying a message.  **F.3** The UI shall offer a mechanism for saving a message.  **F.4** The UI shall offer a mechanism for loading messages from a text file.  **F.5** The UI shall display the messages loaded from a text file already processed.  **F.6** The UI shall offer a mechanism of saving messages loaded from a text file.  **F.7** The UI shall display error messages to the user in a user-friendly manner.  **F.8** The UI shall display the three lists: trending, mentions and SIR.  **F.8** The UI shall display success messages to the user in a user-friendly manner.  **NF.1** Responsiveness: The UI mustn’t take too long to respond to any of the user input, half a second is accepted for click-action response.  **NF.2** Speed: the screen refresh time between different actions should not take longer than half a second.  **NF.4** Usability: the UI shall be easy to use, and training time shall not exceed 1 hour.  **NF.5** Reliability: the UI shall be as much error free as possible.  **NF.6** Robustness: if any of the UI components fail to function, it must restart promptly. |

|  |
| --- |
| 5-The system shall be able to save one or more messages to a file in JSON format. |
| **F.1** No more than one message with the same header id must be saved to the file.  **F.2** Success or error messages shall be output to the user to inform him/her of the system status.  **NF.1** Scalability: The system should accept more than one message; \* it shall also cope with many sessions at a time.  **NF.2** Responsiveness: the system shall save messages quickly, no more than a second is allowed for each message.  **NF.3** Privacy: the system shall store personal data according to the latest GDPR regulations.  **NF.4** Size: the system shall cope with a high volume of data.  **NF.5** Reliability and Robustness: failures should not occur, and in case of failure during persistence the system shall restart promptly. |

The system should also be robust, meaning there should be a very low percentage of failures; in the event of a system failure, the NBMA should restart promptly.   
In a real-life scenario users should also authenticate themselves into the system using an encrypted password; authorization will then allow specific user roles to perform various operations such as deleting or modifying records. Finally, dependability allows the system to deliver at all time its services.

**2. Version Control**

Version control is a system that records changes to a file or set of files over time so that specific versions can be later recalled (Pro Git, 2014), as a side effect this facilitates the development particularly in team environments.  
Many VCS have been proposed historically, although today the most used is the Distributed VCS which combines the benefits of Local and Centralised approaches.   
The entire history of files is stored not only on the central server but on every local machine allowing users to synchronize local changes with the repository on the Central Repository Server.

The development of the NBMA is to adopt the widely adopted Git, developed by Linus Torvald in 2005 to solve many problems present in the CSV. The distributed architecture avoids development bottlenecks, promoting branching and efficiently handling a large number of users working on the same project.

As previously mentioned, the development for the NBMA adopts an Agile approach, following the SCRUM methodology which promotes collaboration amongst team members. This is a process skeleton running in work cycles called sprints, with the duration of two to four weeks. In a real scenario, the NBMA team would be composed by about 7 people, amongst which there are the Scrum Master, who maintains the process, and the Product owner who represent the stakeholder.   
The requirements for the NBMA are pushed to the Product Backlog, a list ordered by requirement importance, looked after by the Product Owner. From here the development team selects a number of Product Backlog Items for the sprint Backlog, after which the sprint starts. Each day the team meets to communicate what has been accomplished, what are the next intentions and if there are any difficulties in the work.  
Once a sprint has been completed, a shippable increment of the software is signed off by the Product Owner, effectuating a Sprint Retrospective to improve on the work done.  
For the NBMA three sprints have been designed each of them taking care of the following aspects:

1) Message Validator and File Parser

2) Message Processor and Serializer

3) Merging Components into User Interface

During each sprint the developers work with a main branch, merging through an Optimistic Locking approach which resolves code conflicts and encourages team collaboration. Each sprint is represented by a branch with additional features, merging back into the main one at the end of the sprint and releasing the code under a specific version name.  
If any bug is discovered after release, a team member can make the fixes whilst the other developers keep working on the next sprint.

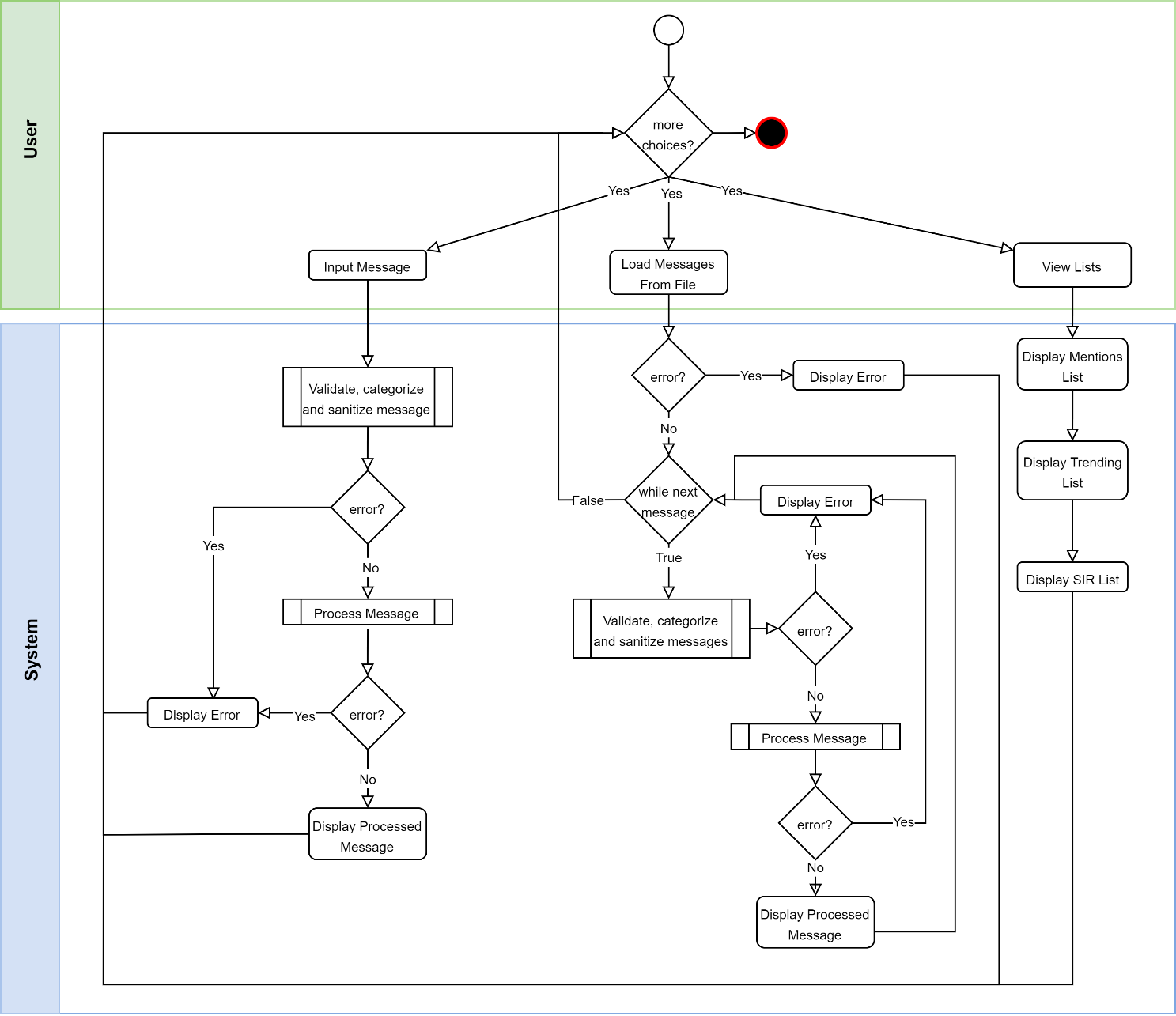
**3. Design and Implementation**

The increasing complexity of software accompanied by the recent technological progress has led to problems with software development.   
“Real quality improvements are only possible if the increase in program complexity is overcompensated”. (Avarage value, Balzert ’96)  
The answer to these problems is Model Driven Software Engineering, thanks to which software is delivered on time, the requirements are accurate, and the software can evolve automatically after release. MDSE abstracts the real-world problem into the model, placing the latter as the central artefact of software development, allowing for better software portability, interoperability and increasing productivity, efficiency and reusability.  
Although there are available industry standards as the MDA proposed by the OMG (generating three modelling levels as CIM, PIM and PSM), for simplicity we will define only a state diagram and a class diagram adopting the UML as the design language.   
Automatic code generation has been used for the base classes obtaining part of the production code, further development included adding a MVVM on top of it.

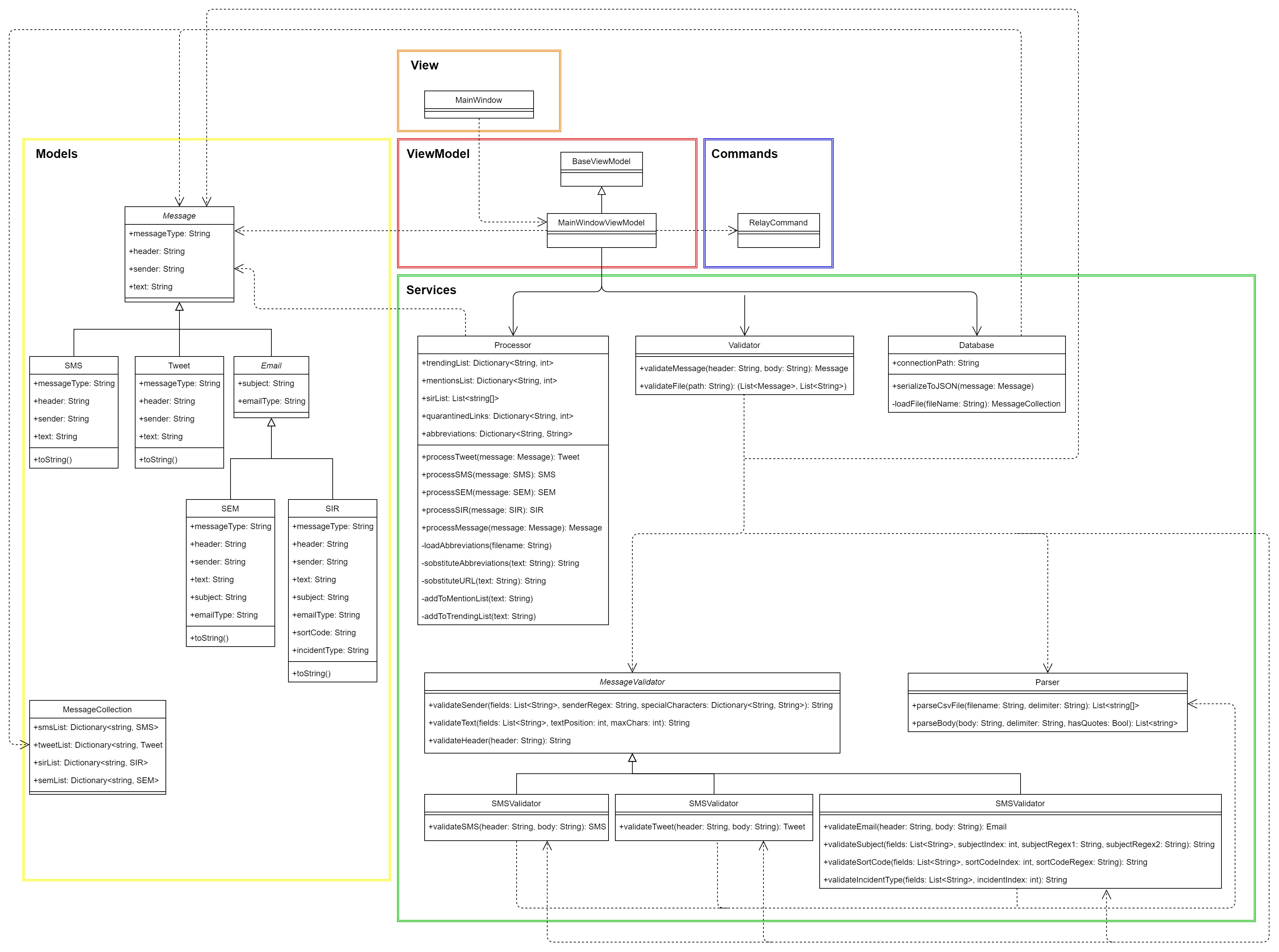
* Describe the MVVM and talk about the benefits

**Activity Diagram**

The activity diagram should be a very high-level abstraction of the system, including system requirements and business objectives, independent from how these are implemented.  
The actions have been divided between user and system for more clarity.



**Class Diagram**



The above class diagram is to be used to generate the code automatically, though after code generation the developers shall manually intervene to fine tune the program.  
The code for the View, ViewModel and Commands has been omitted here for space reasons, although the developers are to follow the MVVM model.

Coupling and cohesion were taken into account; the classes shall have high cohesion, meaning they should focussed only on their specific functions; as for coupling this has been kept low since changing something in one class shall not affect other classes.  
For the NBMA, the proposed architecture implements the MVVM model with important benefits on code modularity. In fact, if we were to swap a view with another one, this would just have to bind its components to the View Model, hence we’d save time on coding a new View Model.   
We can see that MainWindow (the UI) calls MainWindowViewModel which in turns calls other components as the RelayCommand and the classes related to the application business logic.

Models are unaware of the business logic and this makes them more flexible whilst the business logic is handled in the Services folder which will eventually make calls to the models.

The application code is called in the MainWindowViewModel which implements the UI logic, having as attributes: Processor, Validator and Database, among other components needed for the user interface.

When the user wants to process a message or load data from a file the Processor class gets called, which in turn calls the Validator; if the validation is successful then a validated and cleaned message of the designated type will be returned.

The Processor processes the returned message, substituting abbreviations (loaded from a file when the Processor is first instantiated) and keeping track of the required lists (i.e. trending list).

In turn the UI will update the lists each time a message is successfully processed.  
If the user wants to save a message then the Database class will get called, which will use the MessaCollection model to serialize the messages in JSON. Here we can see a private method called “load file”, this deserializes the messages and passes them to the “serializeToJSON” method which will check if the message is a duplicate, hence not serializing it if already present in the file and returning an error.  
The database is not the only class to throw errors in case something goes wrong, by other classes as the Validator do it. Errors are handled by the UI which shall display user friendly messages (avoiding the program to crash).

**4. Testing**

Testing is a process aiming to discover errors in software in order to improve performance, user experience and increase revenue.  
The NBMA testing will adopt the Defect Testing procedure by designing test cases, preparing test data, running the program with the test data and finally comparing the results to the test cases.

Before running a dynamic testing, software inspections should be run, analysing the system in a static fashion; inspections for the NBMA consider requirement specification, software architecture, UML design models and the program.

Software Inspection not only allows to discover errors, but ensures compliance with standards, portability and maintainability.

Dynamic testing can happen at various levels and in a real scenario they should be all implemented:

* Unit Testing: this is the type of testing adopted for this coursework; it verifies individual modules.
* Integration Testing: verifies the design and aims to find errors in interfaces between modules.
* System Testing: verifies requirements; often regression testing is also performed if changes have been made to the system in order to ensure requirements are still met.
* Acceptance Testing: a sort of additional system testing which validates business needs.

The NBMA adopts a Test Driven Development approach (introduced by Extreme Programming) where tests are written before functionalities are implemented, and we are not allowed to move on to the next increment until the code passes its tests. Testing documentation shall be produced adopting the V-model, where testing and development happen in parallel; the benefits of TDD also extend to code coverage, simplified debugging and regression testing.

As regards test tools, unit testing should be automated without requiring manual intervention, developers shall extend the text classes provided by the unit testing frameworks in order to create specific taste cases.

Test cases have been derived on test items (the classes used for validation, parsing and processing) using boundary values and equivalence partitioning in order to cover a wide range of input possibilities, these are available in the table at the end of this section.

Ideally testing should be done by an external party, although for the NBMA this is not possible, so a white box unit testing has been performed as we understand the structure and logic of the system. On the contrary black box testing could be applied at the integration stage in order to discover defects; here a bottom-up approach could be proposed as test cases are easier to design. Here testing works upwards, and a Test Harness coordinates input and output.   
A more modern alternative could e using thread testing, which is object-oriented, hence quite suitable for the NBMA which has several processes interacting with one another under the MVVM architecture.

Interface testing, quite important for object-oriented systems as the NBMA, has been done manually to ensure all the components efficiently communicate with one another.

Stress testing has also been performed on the application as regards loading messages and persisting them to a text file in JSON; as predicted in the requirement risks, as the file content increases the system performance decreases. A solution will be proposed in the evolution section where an optimized database could be used.

Ideally the NBMA should be installed on different platforms and check the tests pass on these as well to ensure correct system compatibility.

Finally, the system testing is to be performed with the customer to ensure the requirements have been met. This can be run either through acceptance testing, installation testing, alpha or beta testing. For the NBMA alpha testing has been performed manually with successful results.

To conclude, testing should be applied at the end of each sprint to ensure the software requirements are met.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Component** | **Objective** | **Input Data** | **Expected Output** | **Output** |
| MessageValidator {ValidateHeader()} |  |  |  |  |
|  | The header must have a max length of 10 chars. | S000000000 | Pass |  |
|  |  | <empty> | Fail | Success |
|  |  | S000 | Fail | Success |
|  |  | S000000000000 | Fail | Success |
|  | The header must start with S, E, or T. | S000000000 | Pass | Success |
|  |  | E000000000 | Pass | Success |
|  |  | T000000000 | Pass | Success |
|  |  | X000000000 | Fail | Success |
|  |  | Y000000000 | Fail | Success |
|  | The header type must followed by only numeric characters. | S000000000 | Pass | Success |
|  |  | S000000XYZ | Fail | Success |
|  | Lower case is capitalized. | s000000000 | S000000000 | S000000000 - Success |
| MessageValidator {ValidateSender()} |  |  |  |  |
|  | SMS body must have a sender specified | <empty> | Fail |  |
|  |  | +1234567 | Pass | Success |
|  | SMS Sender length between 7 and 15 included | +123456 | Fail | Success |
|  |  | +1234567890123456 | Fail | Success |
|  | Tweet body must have a sender specified | <empty> |  | Success |
|  | Tweet sender length max = 15 | @david | Pass | Success |
|  |  | @1234567890123456 | Fail | Success |
|  | Tweet sender in form @<chars> | david | Fail | Success |
|  | Email body must have a sender specified | <empty> | Fail | Success |
|  | Email sender must be an email | david@gmai.com | Pass | Success |
|  |  | david@ | Fail | Success |
| MessageValidator  {ValidateText()} |  |  |  |  |
|  | SMS text not longer than 140 chars | <text longer than 140 chars> | Fail | Success |
|  |  | <text smaller than 140 chars> | Pass | Success |
|  | Tweet text not longer than 140 chars | <text longer than 140 chars> | Fail | Success |
|  |  | <text smaller than 140 chars> | Pass | Success |
|  | Email SEM text not longer than 1028 chars | <text longer than 1028 chars> | Fail | Success |
|  |  | <text smaller than 1028 chars> | Pass | Success |
|  | Email SIR text not longer than 1028 chars | <text longer than 1028 chars> | Fail | Success |
|  |  | <text smaller than 1028 chars> | Pass | Success |
| EmailValidator {ValidateSubject()} |  |  |  |  |
|  | Subject must be specified | <empty> | Fail | Success |
|  | Subject length <= 20 chars | <string longer than 20 chars> | Fail | Success |
|  | If SIR return SIR | SIR 02/07/1995 | Pass | Success |
|  | If SEM return SEM | Hello World | Pass | Success |
| EmailValidator  {ValidateSortCode()} |  |  |  |  |
|  | Sort code must be specified | <white space/empty> | Fail | Success |
|  | Sort Code in format nn-nn-nn | Hello | Fail | Success |
|  |  | 99-99-99 | Pass | Success |
| EmailValidator  {ValidateIncidentType()} |  |  |  |  |
|  | Incident Type must be specified | <empty> | Throw Exception -Fail | Success |
|  | Incident belongs to specified list | Hello | Fail | Success |
|  |  | Theft | Pass | Success |
| Processor  {SobstituteAbbreviations()} |  |  |  |  |
|  | Text abbreviations are processed | LOL | LOL <laughing out loud> -Pass | Success |
|  |  | lol | Fail | Success |
| Processor  {SobstituteURL()} |  |  |  |  |
|  | URL is replaced | http:\\\\mywebsite.com | <URL quarantined> Pass | Success |
|  |  | hello | Fail | Success |
|  | URL is added to quarantined list | http:\\\\mywebsite.com | URL added to list - Pass | Success |
|  |  | hello | Fail | Success |
| Processor  {AddToMentionList()} |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Processor  {AddToTrendingList()} |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Interface Manual Testing**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Component** | **Objective** | **Action** | **Expected Output** | **Output** |
|  |  |  |  |  |
| Tab buttons | On click window is changed to appropriate section | Click | Change to appropriate section | Success |
| Header Text |  |  |  |  |
| Load File Button | Open File Dialog |  |  |  |
|  | Selected file is loaded |  |  |  |

**\*add stress testing**

**5. Evolution**

Software change is inevitable if we want the system to keep delivering its functionalities efficiently, therefore evolution is a critical aspect in organizations, which often involves higher expenses than the development itself (from 2 to 30 times more); process and costs depend on many factors such as the type of the software, the development process, the skills of the staff.

Before evolution takes place, the system has to be understood, therefore the Napier Bank should stay with the same software company, as the developers already have a clear understanding of the program.   
Although the above may not be a contractual responsibility for the tech team, the bank should keep this in mind in order to avoid unnecessary expenditures.

Furthermore, the NBMA adopts a modular approach to programming which improves software maintainability; evolution can easily happen as part of the Agile lifecycle where transitions from development to evolution (which is just a continuation of the development process) are seamless. Developers shall also continuously refactor the code, so to keep it clean and simple improving maintainability.

Lhnman and Belady proposed several rules which can be applied to systems as they evolve, although they are mainly applied to large systems, we can extrapolate relevant information for the NBMA paraphrasing them into the following: as the system is introduced into the environment, other requirements might arise increasing the system complexity with decline in software quality, unless we evolve and maintain the system.

The various types of maintenance the NBMA will need are:

* Corrective: bug identification and elimination, concerned with repairing software.  
  As the the many sprints reach their release, the customer will do the beta-testing, identifying and reporting bugs which need to be corrected.  
  This step should take the least amount of effort with about 17% of the total maintenance.
* Adaptive: the bank might want to extend the system to different environment (i.e. different devices, computer architecture, Operating Systems), hence the system shall be adapted to these.  
  Environmental adaptation should take about 18% of the maintenance effort;
* Perfective: implementing new system requirements by adding or modifying functionalities.  
  Once the system is released, the customer (Napier Bank) will realize they need more features, or that certain aspects might need to be changed to better suit the business needs, hence the system shall be perfected.  
  Focus must be given to this area which will take about 65% of the whole maintenance.

NBMA is a very simple application, on this account we can say the costs won’t be very high, although they could increase in the future as new components are added.

Over its first year of “life”, the system will undergo several changes which might involve:

* Automating message insertion as these are sent to the bank, improving the overall efficiency of the company (the bank could save on labour effort which has to manually input messages);
* Replacing JSON serialization with a database as we saw in the testing section that the system performance decreases when trying to write to a very large text file.   
  Using a database as part of a possible evolution strategy could dramatically improve the program performance as such systems are optimized for the above tasks, often implementing a lookup table with constant access time. This could potentially be the most expensive part to maintain.
* Changing the user interface to improve user experience. This might involve moving around or adding components, as well as building a colour customization feature.
* Extending support for new type of messages and different file formats.

In the event of the need for system re-engineering, the NBMA shall be restructured with positive impact on costs (as opposed to creating a new system).   
The reasons for re-engineering could be many, some of them involve system age, supplier stability, performance, interoperability, data, programming language, personnel skills. The previous factors grouped under business, environment and application assessment, must be carefully considered in order to understand whether the system needs to be completely scraped, continue system maintenance, re-engineer or replace the system with a new one.

Re-engineering takes place after the system has been maintained for some time and costs have increased; automated tools are available for such tasks which aid in architecture recovery and selection.

After all the documentations has been gathered and the architecture has been recovered, the migration can take finally place, evolving the legacy application to a new system.

**Notes**

Error handling:

The system aims to be user friendly informing the user of the nature of the error when performing a wrong action.

**References**

Enfocus Solutions (2012) *Business, User, and System Requirements* Available at: https //enfocussolutions.com/business-user-and-system-requirements/#:~:text=User%20requirements%2C%20often%20referred%20to,must%20be%20able%20to%20perform.&text=System%20requirements%20are%20the%20building%20blocks%20developers%20use%20to%20build%20the%20system.

ReQtest (2012) *Why is the difference between functional and Non-functional requirements important?*Available at: [https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/#](https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/)

Napier’s supplied slides and reading notes.