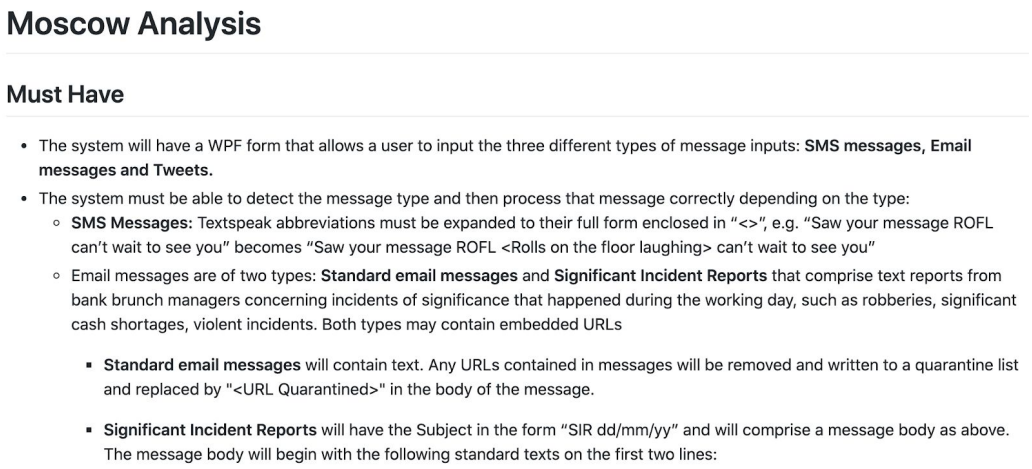
Napier Bank Message Filtering Service - Report

1.1 Requirement Specification

1.1 Moscow Analysis (see ./Report/RequirementAnalysis/MoscowAnalysis.md)

My project began with a requirement analysis to identify all the features that the application would need to have in order to meet the satisfaction and fidelity of Napier Bank's proposed application idea.

After reading over the coursework and analysing the scenarios, requirements, message types etc., I conducted a MOSCOW analysis. I find this a great place to start as you are mitigating the requirements into achievable tasks. Furthermore, you are then categorising these tasks by priority (Must have, should have and could have) which allows you to plan ahead and make sure the most important tasks are implemented first. Here is a sample of this document:



Moscow Analysis

Must Have

- The system will have a WPF form that allows a user to input the three different types of message inputs: **SMS messages, Email messages and Tweets.**
- The system must be able to detect the message type and then process that message correctly depending on the type:
  - **SMS Messages:** Textspeak abbreviations must be expanded to their full form enclosed in "<>", e.g. "Saw your message ROFL can't wait to see you" becomes "Saw your message ROFL <Rolls on the floor laughing> can't wait to see you"
  - Email messages are of two types: **Standard email messages** and **Significant Incident Reports** that comprise text reports from bank brunch managers concerning incidents of significance that happened during the working day, such as robberies, significant cash shortages, violent incidents. Both types may contain embedded URLs
    - **Standard email messages** will contain text. Any URLs contained in messages will be removed and written to a quarantine list and replaced by "<URL Quarantined>" in the body of the message.
    - **Significant Incident Reports** will have the Subject in the form "SIR dd/mm/yy" and will comprise a message body as above. The message body will begin with the following standard texts on the first two lines:

This sample shows some of the must have requirements that I identified. My must have requirements are tasks that have to be implemented before my project submission. They include tasks such as the processing of messages or displaying of message metrics in the GUI after an input session. The Should Have section outlines tasks that don't have to be implemented but should be. In this case, the inputting of messages by file was one of these tasks. Finally, the Could have section outlines tasks that don't have to be implemented but could be something to consider in the future.

Prioritising these tasks that early on helped me plan my project workflow more effectively and gave me a better idea what I would need to get done in order to have a complete application. for submission

1.2 Functional & Non-functional Requirements (see ./Report/RequirementAnalysis/FR&NFR.md)

After writing my Moscow Analysis, I needed to create my use cases. To help me design the use cases, I created a document that outlined my functional and non-functional requirements.

Functional & Non-Functional Requirements

Functional Requirements

- Two Actors:
  - **A user,** who inputs a message using the form or inputs a file (containg messages).
  - **A JSON file,** where processed messages will be saved.
- Four Use Cases:
  - **Input message(s).**
  - **Detect and process message(s).**
  - **Write messages(s) to JSON file.**
  - **Display trending list, list of mentions and the SIR list.**
- Three Directed Associations:
  - **Message(s)**
  - **Processed message(s)**
  - **Lists (trending list, list of mentions and the SIR list).**
- Three extends.

Non-Functional Requirements

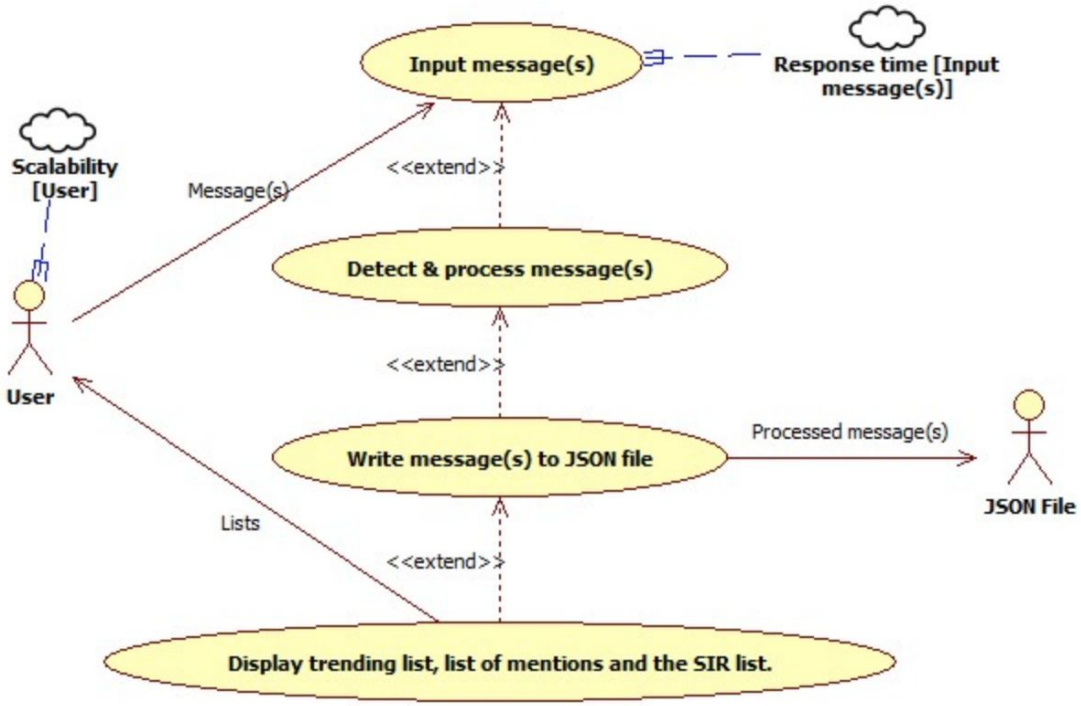
- Two NFR Soft Goals:
  - **Scalability**
  - **Response Time**

The functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. Non-functional requirements are constraints on the services or functions offered by the system. This document helped me better understand the behaviour, inputs of the application. It helped convert a long list of requirements identified in my Moscow analysis into a refined representation of the system logic. For example, the use cases in this document show that there are 4 main behaviors of this system: inputting messages, detecting and processing messages and display message metrics.

I represented the functional requirements inputs by actors and the system behaviours by use cases. This made it far easier to create my use case diagram in the next step. I represented the non-functional requirements by “soft goals”. The soft goals I identified

were scalability and response time. I the felt response time of message processing was important in this application as there may be lots of messages imputed at once.

1.3 Use Case Diagram (see ./Report/RequirementAnalysis/UseCaseDiagram.md)
After outlining my functional and non-functional requirements, I then used them to create a use case diagram. This diagram gives a representation of a users/actors interaction with the system and shows the relationship between the user/actor and the different use cases in which the user is involved

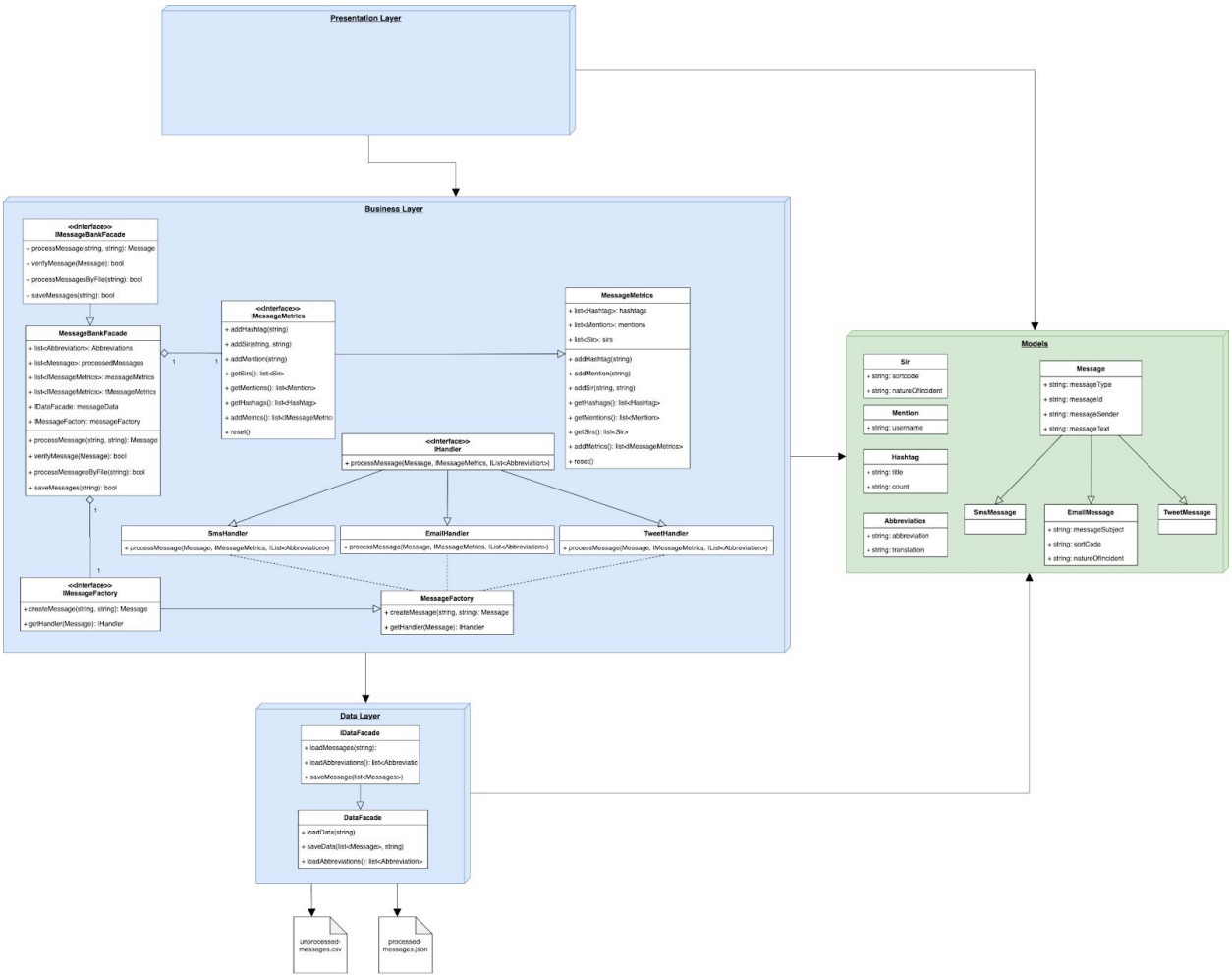


This diagram made the system conceptually easier to understand as it gave a visual representation of how all the use cases interacted with each other and also how they interacted with the actors of the system. One factor it identified was that there was a very linear interaction between use cases in the application. Initially triggered by the inputting of messages by the user, the system shows how these messages are processed, saved and then the message metrics are displayed to the user. This linear design was something I considered when implementing the application.

2. System Design

2.1 Class Diagram (see ./Report/SystemDesign/ClassDiagram.md)

After my requirement analysis I then designed my system. I used a class diagram to represent my application



I decided to design this application using three layer architecture, where the system is broken down into a presentation layer, business layer and a data layer. The presentation layer contains all the logic for the interface/GUI and handles user actions, inputs and outputs. The business layer handles all the logic of the application like message identification and processing. It is also where all the message models are instantiated. The data layer handles the saving and loading of data from files such as abbreviations, processed-message JSON files. The reason I chose to use this particular architecture was to prevent a tightly coupled system from occurring. Each layer is logically detached from the others. The benefit of this is to allow layers to be modified or swapped out without having to change the logic of the other layers. For example, if the client asked for the application to start saving processed messages to a SQL database,

the data layer could be modified to handle this and you would not have to change any code in the business layer to achieve this. This not only makes development easier but it also makes maintenance of the application easier.

You can also see that my class diagram contains a models system/class library. One disadvantage of the standalone three layer model is that the data layer does not know about the data models usually created in the business layer. Although one could argue this leads to a less tightly coupled system; as the data layer knows nothing about the business layer, it actually causes dependency issues. This is because the business layer has to convert data into a data-layer friendly format before passing it down. This means the business layer has to know more about the data layer than it should and has to contain unnecessary logic. It would also result in migration issues when modifying the data layer as the business layer would have to also be modified. My solution was to create a models class that every layer had a reference to. This resulted in a less tightly coupled system and will make any maintenance in the evolution of the application far more seamless.

I also wanted to use dependency injection in this system. It is a technique to remove dependencies by separating the usage from the creation of the object. This reduces the amount of required boilerplate code and improves flexibility. To achieve this I created interfaces for relevant classes and followed the “Dependency Inversion Principle”. In my presentation layer I then created a container class that had a constructor where all my injected classes were registered. When the application is run, all the interfaced classes are injected at runtime. The benefit of this is to allow more flexibility in the development and testing of the system. For example when I want to create integration tests, instead of setting up the application objects methods in each test method I can just instantiate my container class (or a test version of the container class) and build the whole system with injection. Another powerful benefit is that I can register my classes in my container differently in a testing environment than I do in a runtime environment. This allows me to switch out classes with test classes. Injection allows these modifications to be done with zero effort.

Another design aspect of my system was the use of handlers for the messages. The reason I chose to do this was to prevent my model classes from containing heavy methods that handled message processing. This was crucial as there are a lot of message objects created in runtime and I didn’t want the same processed message method to be created in memory for every instance of a message. The handlers fixed this problem.

I also used some design patterns. The business and data layers had facades. These facades were how the layers above could communicate with the layer below it. This resulted in a less tightly coupled system.

3. Implementation

3.1 Visual Studio & C#

I implemented my program in the C# language and used Visual Studio as a development environment. I have experience with both, which was the reason I chose to use them.

3.2 Implementation Workflow & Timeline

I used Zube to track my progress and plan my application workflow. Zube is connected to your github repository. All my use cases which I defined as Github issues were automatically added to the Zube Kanban board. I was able to manage sprints and create multiple tickets to help me mitigate tasks and use cases. It also allowed me to easily see what needed to be done as I could prioritise tasks and set deadlines for sprints. The kanban board also helped me organise all the tasks in every sprint by separating them into separate categories: To Do, Doing, Done, etc.

I began implementing by creating each system in the class diagram by adding a new project to the Visual Studio solution. The presentation layer was a WPF project which allowed me to build the GUI and handle user interaction. Class library projects were used to represent the other subsystems. I then added dependencies to each project to reflect the 3 layer and model architecture of the system.

Once my projects were created, I then implemented the model classes, followed by other classes. Once all my classes were added I then started implementing my use cases. I would tend to start at the highest level in the presentation layer and add elements to the GUI and then work down into my business logic and implement the functionality of the methods required for that use case. Dependreferredency injection meant I didn't have to new up any objects apart from lightweight ones such as the model classes. This made developing quicker and resulted in more refined and understandable code. My evolution strategy was always referred to when developing and I was also always adding comments to make my code more readable and maintainable. My rigorous planning made implementation quick and easy.

4. Testing

4.1 Test Plan (see ./Report/TestingDocuments/test-plan.md)

My test plan outlined my testing strategy for this application. It included Objectives and Scope, test items, tasks and deliverables, testing methods, environmental needs, possible tools, test schedule, and possible risks and solutions.

My overall strategy was to test my code at the end of every sprint. This tended to be after every use case was implemented. I began with unit tests which used white box testing. Integration testing was then used to test multiple components together. I used black box testing methods (using injection) for my integration. Here is a sample of the test plan:

Test Plan

Objectives and Scope

Objectives

- Check that whether the application is working as expected without any error or bugs in real business environment.
- Verify the usability of the website. Are those functionalities convenient for user or not?
- Set up automated tetsing with Travis Ci

Scope

- **In scope**
  - This project will focus on testing all the functions of the napier bank message filtering application. (In this case, this will require all 3 layers of the application to be tested).
  - Integration of functions will be tested to see that parts of the applications work together correctly.
- **Out of scope**
  - Nonfunctional testing (eg. softcases in use case diagarm) such as stress, scalability or response time will not be tested.

4.2 Test Cases (see ./Report/TestingDocuments/test-cases.md)

I then identified my test cases by referring back to my requirement analysis documents. For every use case I defined the constraints and rules required for the application to have high fidelity in regards to the specification. For example one rule for an email message was that the text message had to be <= 1028 characters.

I then created all my test cases, representing them in a table. They give a rough summary of the test, the inputs, the results, etc. Here is a sample of my test cases:

Test Case ID	Test Scenario	Test Steps	Test Data	Expected Results	Actual Results	Pass
UTD1	Check loadData() with file	1. Create a csv file with correct message fomratting 2. Call loadData()	"Filepath"	List returned with correct length and format		
UTD2	Check loadData() with no file	1. Create a csv file with correct message fomratting 2. Call loadData()	""	Exception with message(File does not exist)		
UTD3	Check saveData() with file	1. Create list of messages 2. Call saveData()	"Filepath"	A json file is created with correct output		

4.3 Test Summary (see ./Report/TestingDocuments/test-summary.md)

After my tests I would update the results in the test case tables. I then gave an analysis and reflection of my results at the end of the document to make it clear what needed to be updated and fixed in the next iteration of development.

4.4 Travis and Continuous Integration

I integrated Travis CI, a tool for continuous integration, into the application. This came very useful when it came to testing because every time I pushed my code to Github, it is then pushed to Travis where it builds the solution. I then set it up so Travis then runs all the unit tests and integrations tests. This allowed for automated testing every time my project was changed and pushed to GitHub. This made it far easier to find errors and bugs in my application as my application was constantly being tested and I would be alerted if there were any tests that failed. I set up how I wanted my application to be built and tested by defining rules in a travis.yml file:

```
language: csharp
mono: none
dotnet: 3.1

env:
    global:
      - CODECOV_TOKEN="5f1d7ac5-6fbd-4947-b2fc-35956fce64b8"

script:
  - dotnet --version
  - dotnet build ./NapierBankMessageFilteringService/NapierBankMessageFilteringService.sln -c Travis
  - dotnet test -c Travis --no-build ./NapierBankMessageFilteringService/XTests/XTests.csproj /p:CollectCoverage=true /p:CoverletOutputFormat=opencover

after_script:
  - bash <(curl -s https://codecov.io/bash)
```

5. Version Control

5.1 GitHub

I used GitHub to handle version control of the application. I chose it as I use it all the time and have experience with it.

5.2 Branches

I had 3 types of branches: main develop and feature. The main branch is only used for release versions. The Develop Branch contains more frequently updated versions of the code. Everytime a new feature or functionality is being implemented a feature branch will be created. Different branches allow multiple people to work on this project simultaneously without clashing.

6. Evolution

6.1 Strategy **(see ./Report/EvolutionStrategy/evolution-strategy.md)**

The evolution plan proposes a strategy on how application will be maintained and evolve over its lifetime. It will give future developers a clear plan and strategy on how to maintain the system. The main goals would be:

- **Improving the usability and performance of the Napier Bank Message Filtering Application.**
- **Improving the maintainability and interoperability of the system.**

Any maintenance on the system would be approached in an agile way, with goals, workflow, costs, team members, etc. set up by a product owner initially. Iterations would be executed in the form of sprints until goals were achieved.

6.2 An example of a modification

I proposed some perfective, corrective and adaptive maintenance ideas for future versions of the application. One perfective modification was the option to allow users to change how messages are processed or the ability to add new message types to the system. I believe this would add more usability to the system and also reduce the amount of work to maintain the system because developers wouldn't have to adjust the program every time the Bank asked for a new message type.