Software Engineer - Napier Bank Message Coursework

Davide Pollicino – 40401270

40401270@live.napier.ac.uk

School of Computing, Edinburgh Napier University, Edinburgh – Software Engineer (SET09120)[1]

Abstract: The aim of this coursework is to discuss and analyze the design, implementation and testing processes involved in the realization of the Napier Bank Message Application. The NBM (Napier Bank Message) app, has been developed implementing a **WFP application**, and using **C#** as programming languages**, JSON** as file format for the **data serialization**, .**NET Unit testing** for the creation of a Test class and Test methods, in order to validate our Desktop application and draw.io for the creation of the **use cases** and **class diagram.**

Requirements Analysis of the NBM project

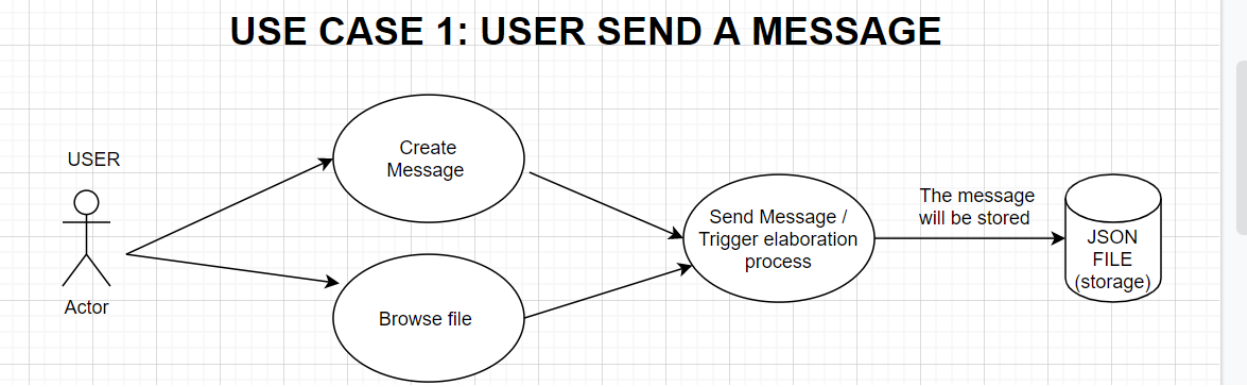
The Napier Bank costumer support application is required to:

- Let the user type a message from a form or allows the user to upload the message from a file chosen via GUI;
- Recognize by the structure of the message ID, the nature of the message: Text Message, email, tweet.
- Extract from the message body the ID of the sender: Mobile phone number (in case of a text message), email ID (in case the given input is an email message) or a twitter id (in case the given input is a tweet).
- Hide any URL present in the email message, extend any message abbreviation presented inside the email and text message (referring to the abbreaviation.csv), and count the frequencies of each hashtag and manage a trending global list of hashtag.
- Serialize and deserialization all the message, list of quarantined hashtags and trending global lists.
- Categorize a given email as priority or non-priority email in according to the emergency criteria.

Use Case Diagram

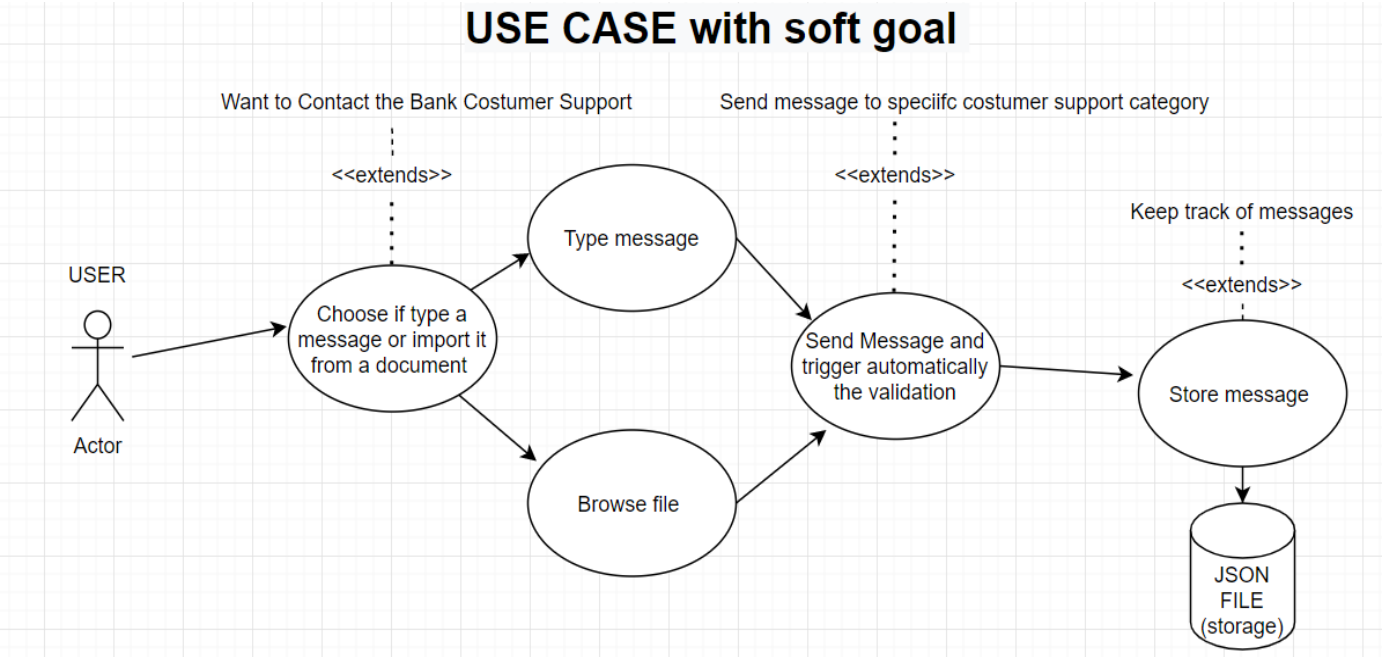
A UML use case diagram is the primary form of system requirements for a new software program under development. Each use case diagram specifies the expected behavior of the application, that summarize some relationship between use case (specific action that the user is executing), actors (the users) and the system (our bank message application).

Below, we can find a use case diagram of our application.



Use case with soft goals (NFRs)

A **soft goal,** is a system an implicit system requirement. For implicit requirement, we indicate a software functionality or characteristics that has not explicitly indicated in the **development plan**, but that must coexist in our environment in our to achieve our goals. Use case with soft goals in fact, represent a **non-functional requirement** and the relations between non-functional requirements. Below, we can find a use case with soft goal diagram related to our application.



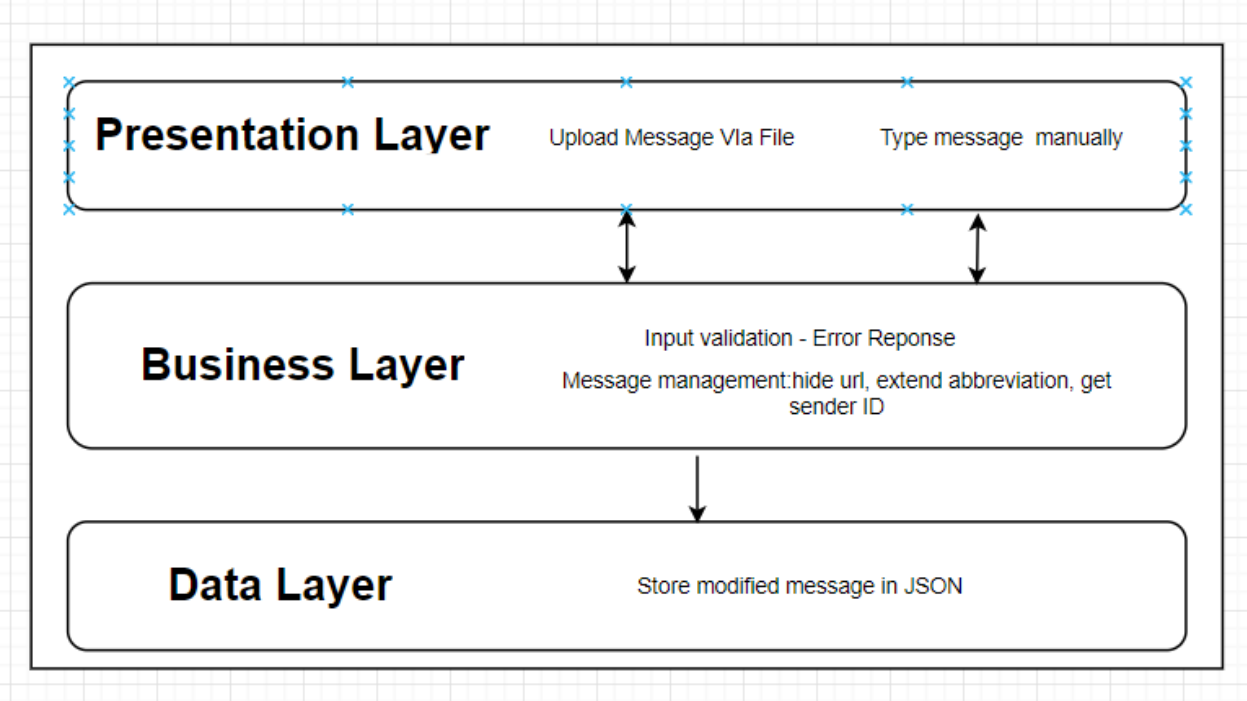
Project Structure and Class Diagrams

Before to process with the implementation of the source code, it is necessary to analyze and approve the structure of the project and the set of classes that will be used within the project. This step, will allows us avoid errors and re-design during the development process, guaranteeing a high level of **scalability** and **modularity** of the application, proven by an **accurate design**.

Our application is divided in 3 logic part, a Presentation Layer, responsible for the interaction between user and platform, a Logic or Business Layer, responsible for the message analysis and message elaboration and a Data layer, responsible for data-serialization and data de-serialization.

Below [Figure 3], we can find the project structure of our application.

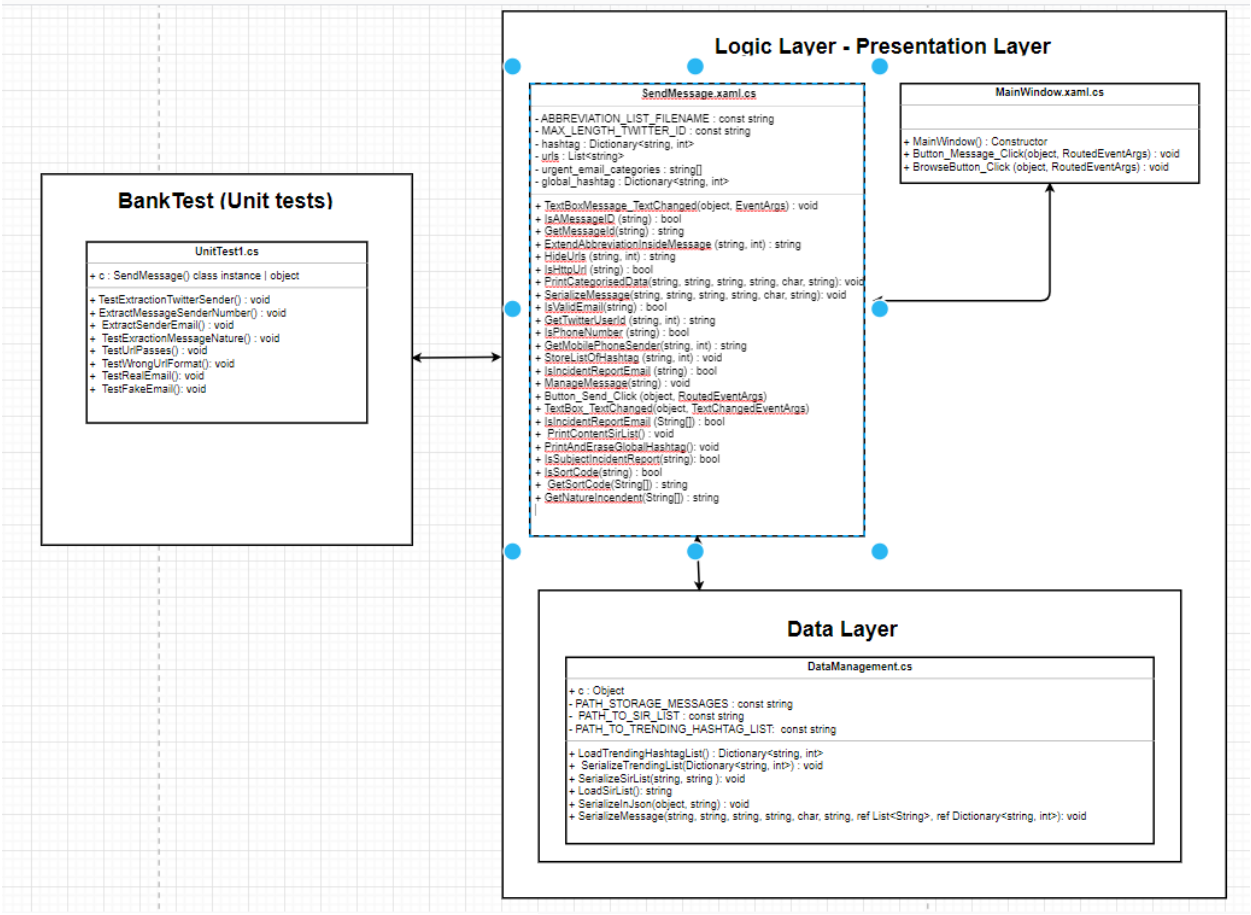
Figure 3: Project structure.



A class diagram is a structured diagram that allows us to describe the overall structure of the system, defining the classes, attributes and methods that will be used withing the platform, defining also eventually the relationship between these classes and the accessibility level of each class, attribute and methods, like private, public and protected (the default in C#).

Below, in the Figure 4., it's possible to analyze the class diagram of the Napier Bank platform.

Figure 4: Class diagram



BankTest (Unit tests)
UnitTest1.cs
+ c : SendMessage() class instance | object
+ TestExtractionTwitterSender() : void
+ ExtractMessageSenderNumber() : void
+ ExtractSenderEmail() : void
+ TestExtractionMessageNature() : void
+ TestUrlPasses() : void
+ TestWrongUrlFormat(): void
+ TestRealEmail(): void
+ TestFakeEmail(): void

T**esting Strategy and Test procedures**

To correctly validate and test the entire project, it is has been essential to design and implement different test. Every test has been created with the goal to check any possible test-case (specific situation that can vary in according with the nature of the input given or by the direct interaction with the platform).

A first set of tests has been realized in base **to direct project requirements**. A second set of tests, has been created considering the **indirect requirements** of the project, where, for indirect requirements, we intend all the requirements that has not been explicitly indicated in the project statement, but are essentials in order to avoid elaboration anomalies or bugs. An example of test that has been created for an implicit requirements, is that the message provided by the user by the WPF form, must not be empty.

It is essential that every time a new method is implement or updated, it will be fully tested and approved, showing evidence of test success.

Tested corner cases

- The input provided is empty (Message error will be given)
- The message input does not have any sender id (Unknown ID will be shown during the message output)
- The mobile phone number of the text message sender does NOT have a proper format. (Unknown ID will be shown during the message output)
- The email sender id does not have a current email format. (Unknown ID will be shown during the message output)

Test report

In the current section we will analyze the type of test conducted and an overall report about the result of these tests and bug-fixing apported thanks to the discovery of bugs during the running of tests in the platform.

- **Type of test executed**: two different types has been executing in the project. The first category, during the development process, called Alpha testing, made within the controlled developed environment thanks to the unit test. A second category has been made thanks to the use of random users that tested the platform to detect any kind of possible anomaly and reporting to me as bug fixing.

- **Class testing:** to test all the objects and functionalities used within the platform, a set of unit tests has been created within the solution Bank Test of our solutions. In few cases, for example IsHttps() , has been tested multiple times, where in both cases, it was checked the actually validation of the given string, in case of correct of incorrect input. This lead led to discover that most of the URL validation involves just the use of the startWith() function to actually check if a given string star with http, https or ftp. It has been clear that a given input like “https, “, would

actually pass the test, for this reason, an appropriate and functional regex expression has been used for this kind of string validation.

- **Environmental needs:** the entire software must be highly modular, to easily test any possible function and decrease the level of difficulty of any debugging session.
- **Test schedule and plan**: all the test would be run, every time before commit in the project sub-branch. To decrease the changes to get any last minute error or conflict tests will be run before merge the main-sub branch of the repository and the master branch (available to user and to the potential client).
- **Risks and solutions:** An incorrect and non-modular update of the methods could potentially invalid the unit test; For this reason, any unit test must be always directly test one and single method, responsible for one and just one determined function.
- **Use of visual studio for test facilities:** for this project, it has been used the .NET unit test for the creation of **unit test**[3] and the test solution explorer to monitorate, run and check the test of each test individually.

Agile approach development and VCS

An Agile development approach is required in terms of development speed and overall project management.

For allowing the contribution between multiple developers or teams in the same project, it is necessary to use a **Versional Control System,** to allow every developer to:

- Keep track of the files and documentation history.
- Have access to all the versions of any file
- Collaborate on the same project and even in the same files.
- Being able to receive rapidly the updates/changes made by other developer in the project.

A perfect solution for our requirements and needs, is: Git, the most popular and widely used Version Control System (VCS).

In this project, it will be used the GitHub platform, having so access to free infinite numbers of private repository, and have an extremely easy management of the access granted to every development team member, on a specific repository.

Evolution and Maintenance prediction

In this first part of the evolution and maintenance section, we will discuss the next evolution step of the Napier Bank Application.

Just after the release, the core functionalities of the application dedicated to actually recognize the category message, will be improved training a model that will be able to detect with an higher percentage of prevision, the nature of the message, and report it to the dedicated team.

In terms of user experience, it will be created a chatbot that will be automatically loaded when the application will be runner from the user. Here, in this section, thanks to the interaction between user and application, it will be easier for both users and support team manage any support request. With this new update, any user will be able to directly select in the chatbot, the type of support requested: personal account, cash lock, issues log-in to the account.

Thanks to the chatbot presence, we will be able to response quickly to the most popular questions and issue (FAQ), for example: How to access to the personal bank account using the Bank application or, how to get a bank statement. In this way, we will be able to solve user doubt's hopefully immediately, allowing the user team to focus on most urgent and particular cases, that require a full and human support.

In this send part of the main tenement section, we will analyze the maintenance costs predicted for the next year, in terms of hosting and version control systems team upgrade.

To actually keep the application online, we will need:

- A Linux hosting server or VPS (About 8£ for month / about 96£ a year)
- At least 3GB of Storage space and Database administration access (About 3.50£ for year)
- A domain name (About 10£ for year)

The minimal annual cost for keeping the application online and reachable from any costumer, it will be about 109.50£.

Of course, this could potentially increase with growth of server resources required.

Currently, best option in terms of scalability and cost, would be a Cloud Based solution, that we can actually just have costs just for the resources that have been used.

Maintenance details

The development Team will guarantee that at any time, the system will be online and reachable to any Napier Bank client.

Of course, any new feature will be discussed with the entire development team and Project Manager, having its dedicated design process, dedicated time, and release estimation.

At the same time, the developers needs to guarantee that if any malfunction or vulnerability has been found and/or report, it is our duty to actually report this vulnerability to the Project manager, fix it, test again the platform and update the rest of the team about the status of this vulnerability (in review - fixing in progress - fixed).

It will also be in the interest of our client privacy and security, doing periodically code refractory related testing, with relative code refractory in case of any new product version of technologies and/or tools used within the project will be released.

Conclusion

It is essential to recognize that with a detailed design process focused on the analysis of the system requirements, code infrastructure, testing and needs in terms of long-term main tenement, the timeframe dedicated to the software development finalized to the first release of the platform drastically decrease, avoiding error or misunderstanding that are usually discovered during the design process.

References

1. Software enginner 09102 Napier University
2. Unit testing: https://martinfowler.com/bliki/UnitTest.html
3. Unit test in Visual Studio: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019