SOFTWARE DEVELOPMENT

Contents

Introduction

This report aims to explain the methodology of this project development project. The class breakdown for this project, explains the program's structure. The Test plans were put in place for this specific project to enhance the robustness of the prototype. The version control plan for this project, future expansion as well, and the evolution strategy of this project moving forward. This report aims to inform the reader and explain the thoughts and process of delivering this task.

Requirement specification

Prototype Requirement Specification for Napier Bank Messaging (NBM) Service

1. Objective

- Develop a prototype application that enables the inputting of messages in various forms, detects the message type, and outputs each message in JSON format.

2. Message Types

- SMS Messages: Accept and process SMS messages with ASCII characters.
- Email Messages: Handle standard email messages and Significant Incident Reports.
- Tweets: Process Tweet bodies with specific attributes like hashtags and Twitter IDs.

3. Functionality

- Message Input: Allow the inputting of messages in the forms specified in section 2.1.
- Message Type Detection: Detect the type of message (SMS, Email, Tweet) and process it accordingly.
- JSON Output: Output each processed message in JSON format.

4. Message Processing

- SMS Messages: Expand textspeak abbreviations to their full form.
- Email Messages: Remove URLs and write them to a quarantine list, replacing them with "<URL Quarantined>" in the message body.

- Tweets: Handle textspeak abbreviations, hashtags, and Twitter IDs.

5. Technology Requirements

- Research JSON and identify an appropriate API for serialization in a JSON file.
- Development in WPF application using C# or other acceptable techniques.

6. Additional Requirements

- Modify the system to read messages from a text file and process them one by one on screen.
- The structure of the input text file can be designed by the developer, but it shouldn't be a JSON file.

7. Testing

- Verify that messages are processed correctly for each type of message.
- Utilize Visual Studio testing facilities or equivalent to conduct tests.

8. Compliance with Standards

- Follow best practices for coding, including code format, clarity, and comments.

9. Documentation

- Provide clear documentation for the code and the overall functionality of the prototype.

Class diagram

EmailMessage

Description: Represents an email message with properties like sender's email, subject, text, and methods to process the email content.

Methods:

DetectType(): Detects the type of the message.

RemoveURLs(): Removes URLs from the text.

Process(): Processes the email message.

ExtractSIRDetails(): Extracts specific incident report details from the email.

App

Description: Represents the main application class.

Methods: No specific methods found.

DataStorage

Description: Provides functionality to read messages from a JSON file.

Methods:

ReadMessagesFromJson(): Reads messages from a JSON file.

MainWindow

Description: Represents the main window of the application and contains methods related to UI interactions and processing.

Methods:

MainWindow(): Constructor that initializes the main window.

ReadMessagesFromJson(): Reads messages from a JSON file.

GenerateNextSortCode(): Generates the next sort code.

LoadAbbreviationsFromCSV(): Loads abbreviations from a CSV file.

ExtractURLs(string message): Extracts URLs from the provided message.

SendButton_Click(...): Handles the send button click event.

ExpandAbbreviations(string message): Expands abbreviations in the provided message.

CheckForMentionsAndHashtags(string message): Checks for mentions and hashtags in the provided message.

UpdateSIRList(string incidentType): Updates the specific incident report list.

SaveMessagesToJson(): Saves messages to a JSON file.

... and several other UI-related methods.

Message

Description: Represents a base message class with properties and methods common to all message types.

Methods:

DetectType(): Detects the type of the message.

Process(): Processes the message.

ExpandTextspeak(string inputText): Expands text speak in the provided input text.

MessageProcessor

Description: Represents a class that processes a list of messages.

Methods:

ProcessMessages(): Processes all the messages in the list.

AddMessage(Message message): Adds a message to the list.

MessageReader

Description: Represents a class that reads messages from an input file.

Methods:

ReadMessage(): Reads a message from the input file.

SMSMessage

Description: Represents an SMS message with properties like sender's phone number and text.

Methods:

DetectType(): Detects the type of the message.

Process(): Processes the SMS message.

TweetMessage

Description: Represents a tweet message with properties like Twitter ID, content, hashtags, and mentioned Twitter IDs.

Methods:

DetectType(): Detects the type of the message.

ProcessHashtags(): Processes hashtags in the tweet.

ProcessTwitterIDs(): Processes mentioned Twitter IDs in the tweet.

UpdateHashtagsAndMentions(): Updates hashtags and mentions dictionaries.

Process(): Processes the tweet message.

Class Descriptions:

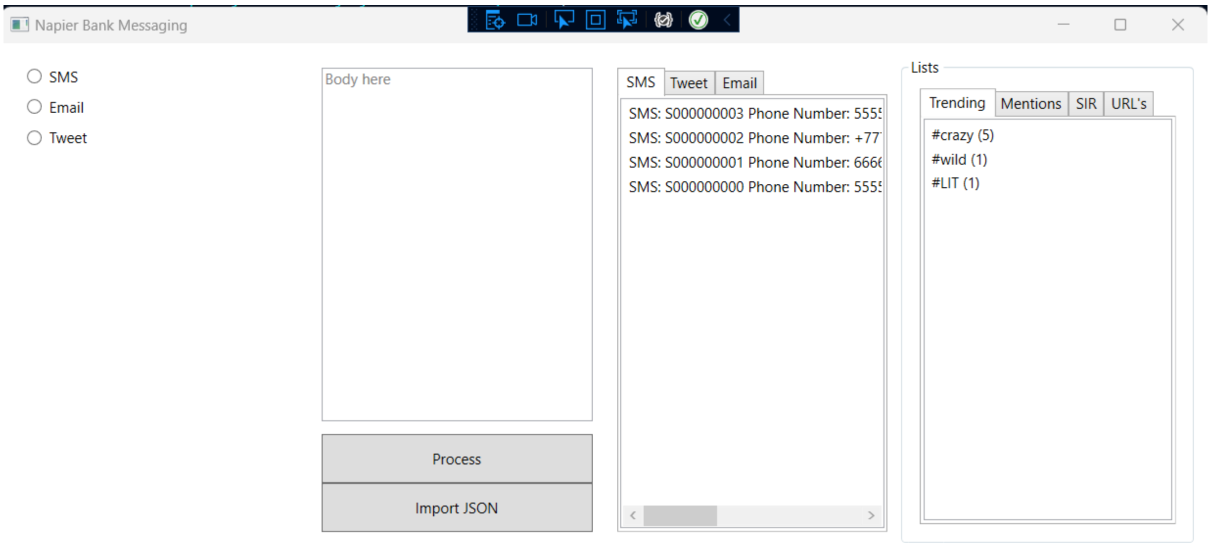
- Message: This is an abstract class that represents a generic message. It has properties like MessageID and Body. It also has abstract methods DetectType() and Process() which are implemented by its derived classes.
- EmailMessage: This class inherits from Message and represents an email message. It has properties specific to emails like SenderEmail, Subject, and Text. It also has methods to process the email content.
- SMSMessage: This class inherits from Message and represents an SMS message. It has properties like SenderPhoneNumber and Text.

- TweetMessage: This class inherits from Message and represents a tweet message. It has properties like TwitterID, Content, Hashtags, and MentionedTwitterIDs. It also has methods to process hashtags and mentions.
- MessageProcessor: This class is responsible for processing a list of messages. It has a method ProcessMessages() that processes each message in its list.
- MessageReader: This class is responsible for reading messages from a given input file. It has a method ReadMessage() that reads a message from the file.
- DataStorage: This class represents a storage mechanism with a property OutputFile.
- MainWindow: This class represents the main window of the application. It contains various fields and methods related to the application's UI and functionality.
- App: This class represents the application itself.

UI Design

This section aims to explain the different thought processes that went into the creation and implementation of the UI (user interface) system for this project. This project did not have a large emphasis on UI however as a developer I wanted this to be built to the best of my ability with the knowledge I have gained over the years in this industry. The UI implementation of the project is primarily centered around the MainWindow.xaml file, which defines the layout and visual elements of the main application window. The main window, titled "Napier Bank Messaging," is structured with a grid layout that divides the window into four columns and three rows.

Within the first column, there are radio buttons allowing users to select the type of message they wish to process: SMS, Email, or Tweet. The second column contains input fields corresponding to the selected message type. For instance, if the user selects "SMS," they are prompted to input a phone number; if "Email" is chosen, fields for the recipient, subject, and an optional incident type dropdown appear; and for "Tweet," a username input is provided. A text box for the message body and buttons for processing the message and importing JSON data are also present.



```
SMS: S000000003 Phone Number: 555
SMS: S000000002 Phone Number: +77
SMS: S000000001 Phone Number: 666
SMS: S000000000 Phone Number: 555
```

```
#crazy (5)
#wild (1)
#LIT (1)
```

(Figure 1 UI final design)

The third column showcases a tabbed output window where processed messages are displayed under their respective tabs: SMS, Tweet, or Email. The fourth column displays lists of trending topics, mentions, significant incident reports (SIR), and quarantined URLs.

The logic behind the UI is encapsulated in the MainWindow.xaml.cs file. This code-behind file contains event handlers for UI interactions, such as radio button selections, which dynamically adjust the visible input fields based on the user's choice. Additionally, there are methods for processing and displaying messages, handling mentions and hashtags, and managing quarantined URLs. The application also has the capability to save and load messages in JSON format, and there's a mechanism to expand abbreviations found in the messages.

A well-designed user interface (UI) is paramount in modern software development, offering both tangible and intangible advantages. It enhances user efficiency, reducing the time and effort required to complete tasks and leading to improved productivity. A good UI also minimizes user errors, as intuitive designs guide users seamlessly, reducing the likelihood of mistakes this is showcased in the message validation, present in the input limitations i.e., the character length. This, in turn, diminishes the need for extensive training or user manuals. Furthermore, user satisfaction is directly influenced by the UI; a streamlined and intuitive interface can lead to increased user retention and loyalty. From a business perspective, a superior UI can be a differentiating factor in a saturated market, providing a competitive edge. Lastly, a well-constructed UI can reduce development costs in the long run, as fewer revisions and post-launch fixes are required. In essence, investing in a robust UI is not merely about aesthetics but has profound implications for user efficiency, satisfaction, and overall business success.

Test Plan

The test Plan for this prototype was to create test cases based on the ideal outcome of this prototype. The ideal running scenario is a case where the prototype runs to meet all specifications. This test plan outlines the testing strategy for the Napier Bank Message Filtering Service. It includes the objectives, scope, test items, tasks and deliverables, testing methods, environmental needs, possible tools, test schedule, and possible risks and solutions.

1. Objectives and Scope

The main objective of the testing phase is to verify that messages are processed correctly for each type of message. The scope includes various types of testing to ensure the functionality and maintainability of the system.

2. Test Items

- Class diagram
- WPF application written in C#.
- Text file processing and display

3. Tasks and Deliverables

- Testing Strategy: Brief description of the overall testing strategy for the system.
- Test Plan: Including Objectives and Scope, Test Items, Tasks and Deliverables, Testing methods, Environmental Needs, possible Tools, Test Schedule, and Possible Risks and Solutions.
- Test Cases: Develop test cases and construct tests to verify that messages are processed correctly for each type of message.
- Use Visual Studio testing facilities: (or equivalence on the platform chosen) to conduct the tests where appropriate.

4. Possible Testing Methods

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

5. Environmental Needs

- Development environment (e.g., Visual Studio)
- Testing tools and platforms
- Access to source code and related documents

6. Possible Tools

- Visual Studio testing facilities
- Other equivalent platforms

7. Test Schedule

- Incremental testing as the project develops.
- Final testing once all code is running to make sure all deliverables are met.
- Post final testing, if any issues need to be addressed, they will be then re-tested.

8. Possible Risks and Solutions

- Identify potential risks in the testing process.
- Propose solutions to mitigate those risks.

test methods

The methods for Testing used in this prototype are Pass-Fail testing. Where a list of requirements shall be written and then cross referenced if they passed or failed the testing process. The pass-fail method is a common approach used in software testing to determine whether a test case has passed or failed. A test case is considered to have passed if the actual output matches the expected output. Conversely, it is considered to have failed if the actual output does not match the expected output. The pass-fail method provides a straightforward way to assess the success or failure of individual test cases. It can be an essential part of the overall testing strategy for the Napier Bank Message Filtering Service, ensuring that the system meets the specified requirements.

test cases

1. Message Input Validation

TC1.1: Verify that the system accepts valid SMS messages.	PASS
TC1.2: Verify that the system accepts valid Email messages.	PASS
TC1.3: Verify that the system accepts valid Tweets.	PASS
TC1.4: Verify that the system rejects messages with invalid formats.	PASS

2. Message Type Detection

TC2.1: Verify that the system correctly identifies SMS messages.	PASS
TC2.2: Verify that the system correctly identifies Email messages.	PASS
TC2.3: Verify that the system correctly identifies Tweets.	PASS

3. JSON Output

TC3.1: Verify that the system outputs messages in the correct JSON format.	PASS
TC3.2: Verify that the JSON output includes all required fields.	PASS

4. SMS Message Processing

TC4.1: Verify that textspeak abbreviations are expanded to their full form.	PASS
TC4.2: Verify that the message structure complies with the specified format.	PASS

5. Email Message Processing

TC5.1: Verify that URLs are removed and written to a quarantine list.	PASS
TC5.2: Verify that "<URL Quarantined>" replaces URLs in the message body.	PASS

6. Tweet Message Processing

TC6.1: Verify that the system handles textspeak abbreviations, hashtags, and Twitter IDs correctly.	PASS

7. File Reading (Additional Requirement)

TC7.1: Verify that the system reads messages from a .JSON file correctly.	PASS
TC7.2: Verify that the system processes and displays messages one by one on screen.	PASS
TC7.3: Verify that the abbreviations are read in from the .CSV file correctly	PASS

8. Error Handling

TC8.1: Verify that the system handles incorrect message formats gracefully.	PASS
TC8.2: Verify that the system provides meaningful error messages.	PASS

9. Performance Testing

TC9.1: Verify that the system can handle a large volume of messages without degradation in performance.	Pending

10. Security Testing

TC10.1: Verify that the system does not expose sensitive information.	Pending
TC10.2: Verify that the system handles malicious input safely.	Pending

test outputs

The Test outputs are a simple pass or fail. If the point achieves the desired and defined objective, it will achieve a passing grade. Contrastly if the objective does not meet the required objective, then it will achieve a failure in the testing stages of this development. have completed two rounds of testing to negate any outliers. This approach can have several benefits in the context of software

development:

1. **Simplicity**: Pass-fail testing simplifies the evaluation process by boiling it down to two possible outcomes. This can make it easier to understand and communicate the results of the testing.
2. **Focus on Minimum Requirements**: By using a pass-fail approach, the focus is placed on whether the software meets the minimum requirements for functionality and performance. This can help ensure that essential features are working as intended.
3. **Speed**: Pass-fail testing can be quicker to execute and analyse since it doesn't require detailed grading or scoring. This can be particularly beneficial in agile development environments where rapid iterations are common.
4. **Objective Evaluation**: The binary nature of pass-fail testing can reduce subjectivity in the evaluation process. Either the software meets the criteria, or it doesn't, which can lead to more consistent and unbiased results.
5. **Integration with Automation**: Pass-fail testing can be easily integrated with automated testing tools. Automated scripts can quickly determine whether a test has passed or failed, allowing for more efficient testing cycles.
6. **Clear Decision-Making**: The clear-cut nature of pass-fail results can facilitate decision-making processes. If a test fails, it's evident that something needs to be fixed. If it passes, the development can move forward.
7. **Reduction of Noise**: By focusing only on whether the test passes or fails, extraneous information that might not be relevant to the immediate development goals can be filtered out. This can help in focusing on what's most important.
8. **Encouragement of Thorough Testing**: Since the only outcomes are pass or fail, testers may be encouraged to create comprehensive tests that thoroughly examine the functionality and performance of the software.

9. **Alignment with Acceptance Criteria**: Pass-fail testing can be closely aligned with the acceptance criteria defined by stakeholders. This ensures that the software is evaluated based on the specific requirements and expectations of the end users.

10. **Resource Efficiency**: Without the need to delve into detailed analysis or grading, resources can be more efficiently allocated to other areas of development or testing.

These points are legitimised by this supporting research conducted before and during the testing phase of the project (W. W. Peng, 2013) (Jan Wloka, 2008).

Analysis

Test analysis in software development refers to the systematic examination of the requirements and design documentation to determine the testable features and derive test objectives. It's a critical phase in the software testing life cycle, laying the groundwork for subsequent stages like test design and implementation. By scrutinizing the software's specifications, test analysis ensures that all functional and non-functional requirements are accounted for in the testing process, thereby enhancing the software's reliability and robustness. This meticulous approach ensures that testing efforts are directed towards areas of highest risk and importance, optimizing resource allocation, and maximizing the efficacy of the testing process.

After reflecting on the Test process, the developer wanted to implement Unit testing. However, they lacked extensive experience in this area and did not have ample time to learn and apply these skills effectively. They have earmarked this for future functionality enhancements. Nevertheless, the developer found the PASS/FAIL method of testing to be satisfactory for a project of this scale. The pending status of the testing arose from certain sections remaining untested. To test these sections comprehensively, the developer would either input data for extended periods or utilize an API to which they don't have access. Regarding security, while they didn't possess the capability to test it thoroughly, no specific security mitigations were implemented, leading to the assumption that this aspect might be a potential vulnerability.

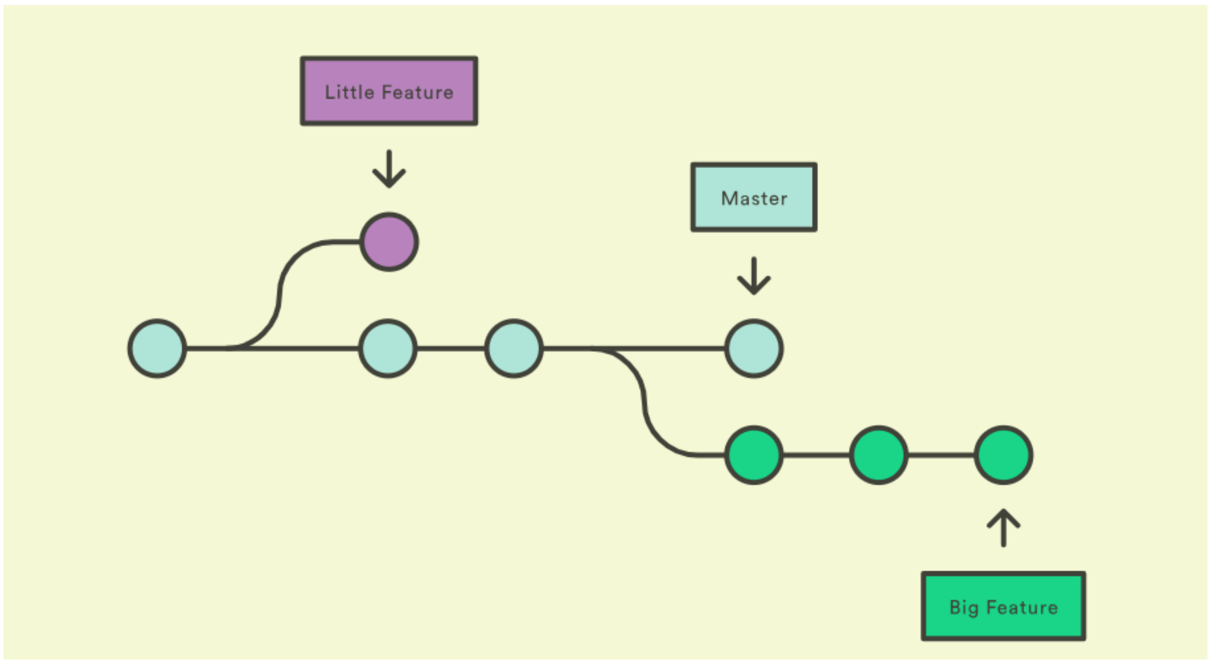
Version control plan

What is version control “Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.” (Scott Chacon, 2023).

This explains the basic system of version control within this system there are more intricate nuances for different specifications. For this project, a simple version of version control will be used, as this is such a small project a simple version where each function section of my project will be saved to my local computer to negate any regression. This will be done by storing multiple files within my project to help manage each section and then tie them together easily when everything is ready for compilation.

For a larger project or a project with more intricate nuances, it would be preferable to use a system in place for better managing version control. A good platform for this would be Git, this platform is used by some medium-sized companies as well as large more intricate projects. The developer has implemented a remote repository however with a project of this size following the strict branch requirements for such small changes didn’t make sense. The developer did utilize this system for some areas however the main use of VC was not used. Version control allows for branches to be created from a main branch and better manage changes to the main. Branching enables seamless collaboration among teams of developers within a centralized code base. When a developer initiates a branch, the version control system generates a duplicate of the code base at that specific moment.

Any modifications made to the branch do not impact other team members' work. This can be visualised with the graphic below.



(Figure 2 Version control in larger projects)

This system is the most versatile and robust system used by most developers in today's world. The version control in this project is based on this system however it is on a smaller scale. Each version is copied and stored however this is all done locally and therefore does have the disadvantage of being a single point of failure as there is no backup besides the local. Using a system such as Git allows for cloud storage and utilization of a decentralized storage system.

Evolution strategy

Code evolution is a critical aspect of software development, reflecting the continuous growth and adaptation of code to meet changing requirements and technologies. Several methods can aid in this process supporting research found in (McConnell, 1993), (Keith Bennett, 2002)

Version Control Systems (VCS): Tools like Git enable developers to track changes, collaborate efficiently, and manage different versions of the code. This ensures that the codebase remains consistent and recoverable, facilitating smooth evolution.

Automated Testing: Implementing a robust suite of automated tests ensures that changes or additions to the code do not break existing functionality. Unit tests, integration tests, and end-to-end tests provide different levels of validation, enabling developers to refactor and extend code with confidence.

Continuous Integration and Continuous Deployment (CI/CD): CI/CD pipelines automate the build, test, and deployment processes. This ensures that code changes are automatically validated and deployed, allowing for rapid iterations and consistent code quality.

Modular Design: Designing code in a modular fashion, with a clear separation of concerns and well-defined interfaces, makes it easier to modify, extend, or replace individual components without affecting the entire system.

Code Reviews: Regular code reviews by peers ensure that the code adheres to the project's coding standards and best practices. This collaborative process not only improves code quality but also facilitates knowledge sharing among team members.

Documentation and Commenting: Proper documentation and in-line commenting provide context and understanding for future developers who may work on the code. This aids in maintaining and evolving the codebase as it grows in complexity.

Refactoring: Regular refactoring helps in maintaining clean and maintainable code. Tools that support refactoring can automate many of these tasks, making the code more efficient and adaptable to new requirements.

The evolution of this project will be very easy to alter, there are many directions in which this project can be taken. For this project, The Developer has taken a more direct approach but for any future development, the developer would take a more modular approach, where each major section of the project would be hot-swappable or interchangeable. With new functionality based on future requirements. This is a relatively new framework and one that is gaining traction in the global tech sector. Although there are areas of modularity there is room for improvement, with more time the developer would take time to refactor and organise the structure of the program to improve modular efficiency by creating more classes and separating logic into smaller sections. From this article, it is clear to see that this approach is superior for a program such as this where different nodes may be forms of inputs and outputs being easily changed (Abazher, 2023). This framework would be tricky to implement as it would require a small refactor of existing code however it is possible and would be greatly beneficial for the future of this application.

By employing these methods, development teams can ensure that their codebase remains agile, maintainable, and aligned with the evolving needs of the project, thereby supporting effective code evolution.

References

Abazher, B. (2023, Jul 20). *Modular Architecture for Software Development: What are the Benefits for Business*. Retrieved from Triare: https://triare.net/insights/modular-architecture-software/

Jan Wloka, B. G. (2008). Safe-Commit Analysis to Facilitate Team Software Development. *Rutgers University (It appears to be a university publication rather than a journal)*, Whole book.

Keith Bennett, V. R. (2002). Software Maintenance and Evolution: A Roadmap. *ACM SIGSOFT Software Engineering Notes*.

McConnell, S. (1993). *Code Complete.* Microsoft Press.

Scott Chacon, J. L. (2023). *1.1 Getting Started - About Version Control*. Retrieved from Git CSM: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

W. W. Peng, H.-Z. H. (2013). A Bayesian approach for system reliability analysis with multilevel pass-fail, lifetime and degradation data sets. *IEEE Transactions on Reliability*, Whole book.