# **W1640643 Report**

# Design changes and reasons.

Initially, in my design decisions within my coursework 1, I decided to make Finance a parent class, having two child classes of Income and Expense, and another parent class, Contact, having two child classes of Individual Contact and Business Contact.

Initially, I made these design decisions in mind with the idea of ensuring that my program was easy to maintain. Believing that separating the different finance types and contact types, into their child classes, would ensure my program was better maintainable. However, when implementing my design decisions for coursework 2, I felt that the properties of the parent classes and the child classes were too similar to justify splitting them up into parent and child classes. I felt that the child classes were redundant and focusing on the YAGNI (You ain’t gonna need it) practice, I decided to get rid of the child classes, and make Finance and Contact no longer parent classes.

Following the YAGNI practice, I feel, in this case, this has helped with the maintainability of my program as it has ensured implementing features that are needed for my program. Not implementing features that you foresee needing when in reality, you do not require these features. The Finance and Contact classes still ensure that the program is maintainable, as they are both self-contained classes, solely responsible for their data.

An initial design decision I made in my coursework 1 was to have several Models and Views be handled by one Controller.

I felt that this would initially help with maintainability and extensibility of the program. I felt this would help with the extensibility of the program, as adding additional features to the program would only require modification of one Controller. I also felt that this would help with the maintainability of the program, as having one Controller for several Models and Views I felt would allow for better handling of the program features and code.

However, I overall felt that this decision was harmful in the ideas of ensuring that my program was maintainable, extensible, readable and the overall principles of Object-Oriented Programming (OOP). When implementing this design decision for coursework 2, I felt that this initial design decision was harmful to these principles as it was forcing me to create a class which was handling too many responsibilities for the program. This idea is known as a God class.

When implementing and programming my program, I decided in the end too, with the thought of ensuring that my program is maintainable, extensible, and readable, that it would be best to create a Controller for each Model and View. Ensuring that each Controller is responsible for only its specific Model and View. An important concept of the MVC (Model-View-Controller) architecture pattern, which is followed to ensure separation of critical information of a program.

Initially in my design, I mentioned having a Support and User Details view. Initially, I felt that the Support view could be used to create tickets for suspicious expense records, find the details of a support team and be able to contact the support team.

However, I believe it was wrong for me to have this view in my initial design. This is because I overall did not understand fully the application to be designed and the overall requirements needed for the application. The Support view would be a view more suited to a banking application – initially what I assumed the application to create was meant to be – this was before understanding the application to create was a personal financial tracking application.

The User Details view was initially designed for the user to be able to view their details and update their details. However, I felt that this was not required into the application and was a view and functionality that was not critical for the application. With this, in my initial design, I also had Forgotten Password and Forgotten Email Address views – which I implemented in my application - that takes away the need for the User Details view, as the update functionality of this view was already designed to be part of the Forgotten Password and Forgotten Email Address views.

Alongside this, I believe splitting up the functionality into their separate views has helped for me to refactor the views and the code throughout my implementation. Which has allowed for my application to be maintainable and extensible.

Overall, during the implementation of the application, I removed the User Details view when focusing on the YAGNI practice. Realizing that overall, this was a view not required for my application and had functionality that was already documented in the design, within other views, which are standard practice views and functionality within applications that users already know and expect. The removal of this view has allowed for my application to be better maintainable and extensible.

# Evaluation: Implementation choices:

When implementing my application, I ensured to follow standard practices, such as:

**Readability:**

To ensure that my code was readable to humans, alongside the computer, I ensured I followed practices to ensure my code is readable.

The first thing I did to ensure my code is readable for humans, is that I ensured that my code is correctly indented. This indentation, or separation, allows for my code to be structured, improving readability. Alongside this indentation, I also used white space, that allows for the code to be better readable for humans.

To allow for better separation of my code, I used methods to help separate the functionality of the code. This allows for better readable of my code for humans and makes it easier to find the underlying functionality of the code and what it is doing. I ensured that these methods had appropriate names so that humans can understand what their purpose is. This the same for the variable within my code, ensuring they have appropriate names, easy for human reading and understanding.

I also ensured to comment and document my code, so that humans who are needing to understand what the code is doing, will be able to do so. Improving the readability of my code for humans.

**Maintainability:**

Many of the practices I followed for readability apply for maintainability of an application. Therefore, commenting, the indentation of my code, separation of code and appropriate names for methods and variables, all have allowed for better maintainability of my application.

However, for better maintainability of my application, some further practices I followed are, to create helper classes within my application. This includes my JSON, ThreadHandler (a class that was designed to be used for my attempt at allowing threading to be used in my application) and FileExceptionHandler classes. Noticing that I was using the same code multiple times – also known as code smell – I decided to create smaller classes known as helper classes to implement these methods within and call the class and the methods needed when desired. This allows for better maintainability as it allows for less duplication of code, meaning that when code needs to be debugged, iterated, or added upon, it will be easier for a programmer to do so.

With this idea of code smell, I also used methods to repeat common functionalities of the application in multiple sections of my code. This idea allows for better maintainability as if a change is needed, you will need to only change the called method, which will update throughout many areas of the code. Instead of having to find code in multiple areas of the code to refactor or update the code and application.

Furthering my application maintainability, I followed the MVC design pattern, to separate my UI from my backend code and my data manipulation. This helps with the maintainability of the code as it allows for it to be easier on updating and modifying the different aspects of the application, without the fear of changing one aspect of an application breaking another aspect of the application.

**Usability:**

To ensure my application meets the needs and demands for the end-user, ensuring that my application ensures high standard usability, I created my application using Microsoft’s WPF (Windows Presentation Foundation) framework. Using the WPF XAML (Extensible Application Mark-up Language) to create the UI (user interface) has ensured my UI is intuitive and responsible to the user.

To ensure that my application meets the user's demands, I have created a dynamic UI for two views in my application – CreateContactView and CreateFinanceView. The dynamic UI allows users to create multiple contact or finance records at once.

To allow for better usability of my application, the functionality of my application has been streamlined to ensure that the functionality of my application is efficient and quick. Ensuring end-users do not have to wait for too long for the application to finish executing a request. I also ensured the end-user is notified when an execution has been completed and its results. This includes notifying the end-user of an encountered error. This includes displaying why and how this error has been encountered. I have also ensured that my application has user input validation for whenever the user is required to input details. Ensuring this does not break the application or cause any unwanted errors for the user.

I decided to use JSON for the reading and writing of data instead of using a database, known as a NoSQL database. I favoured JSON due to not requiring accessing a database, which can cause issues for the end-user, e.g., the database is down, user cannot access the database, etc. Using JSON I believe allows for better usability as it ensures that the data can be accessed and modified at any time for the user. This ensures that my application is accessible 24/7, a requirement that I found during my initial design stage. As end-users wish to use an application 24/7, improving the usability of my application.

**Portability:**

To ensure that my application is portable so that it can be used on multiple devices, I used WPF, which allows for the UI to be made with XAML. When done so, the application UI will be able to be used on multiple resolution rates and allowing for it to be scaled to work on multiple devices.

The event handlers used within my application can be registered and work on mobile and tablet devices. Allowing for the functionality, event handling and user input to be able to work within touch screen devices, allowing for it to work on mobiles and tablets. Allowing for the application to be portable.

Another aspect that allows for my application to be portable is the fact that the JSON, the way I decided to read and write data, can work with mobile devices.

**Reusability**

To ensure that my code was reusable, I ensured that I would use a practice known as a code smell. This is where you find the same or similar code being used in multiple areas of your code and find a method to ensure that this piece of code can be used multiple times in different areas of the code.

The first way I achieved reusability concerning my UI is to use the Apps.XAML within WPF to ensure that styles were incorporated into components of the UI. I used this to make a default and overloaded styles for all the components used within my UI. This means that if I wished to change the style of a UI component I can make modifications in the corresponding style within my Apps.XAML and this will take effect for each component type that was using the style, either by default or explicitly binding a style to a component, using overload styles. This reusability allows for modifications to the UI to be done quickly and efficiently.

Regarding code smell, to ensure that code was not commonly being repeated, I ensured to create methods that could be called when required. This means that this method could be reused/called when required to perform the methods required outcome. Instead of repeating the same code in different areas of my code to get the same result.

**Scalability:**

Scalability is to ensure that your application can meet the demands of increased stress due to increase usage and demand. To ensure that my application is scalable with user demand, I did some of the following.

One thing I did was use JSON instead of using a database. This is because I felt with a database, with more users and data, that it would become more difficult to manage and handle. Therefore, I used JSON, which reads and writes data off of a file from the user's computer. I believe that with large amounts of data, JSON will be able to handle this better than a database at it will only read and write data for one specific user. I also believe JSON is better for scalability as there is no idea of JSON having downtime. With databases, to ensure that the database can manage large influxes of users, maintenance will need to occur, which generally requires periods of downtime. This leaves users without the ability to use the application for a certain period. As JSON will be stored on a single user’s computer, there is no idea of needing this downtime for maintenance of the data.

To ensure scalability, nowadays the application needs to work on a variety of devices, computers, mobiles, and tablets. To ensure that my application would be able to scale with all these devices, I made my application using WPF. This allows for the UI interface and its intractability be able to be transferred over to different devices. This allows for my application to work for end-users, no matter what devices they are using. Allowing for my application to scale towards the needs of different customer products that applications are expected to work on from the end-user’s perspective.

Further, expanding on my ThreadHelper class, -previously mentioned in my maintainability, this was also being used for my attempt to improve the sustainability of my application. Threading would have allowed for my application to read, write, and manage a huge amount of data whilst still allowing for the application to run smoothly for the end-user. However, my attempts of trying to incorporate threading into my application resulted in unexpected results for my application, which I found difficult to debug. A common issue when incorporating threading into an application. In the end, I had taken a tough decision and remove threading from my application, to ensure my application worked and had a smaller number of bugs for the end-user. This results in my application being hard to scale when a user has lots of data needed to be read, written, and manipulated.